CS 3430: S24: Scientific Computing
Assignment 10
Random Numbers, Random Number Sequences, Random Art

Vladimir Kulyukin
Department of Computer Science
Utah State University

April 07, 2024

Learning Objectives

1. Pseudorandom Number Generation
2. Linear Congruential Generator (LCG)
3. Xorshift
4. The χ^2 Test of Sequence Randomness
5. Random Images

Introduction

In this assignment, we will experiment with the Linear Congruential, 32-bit Xorshift, and Mersenne Twister pseudorandom number generators (PRNGs), do some random image art to visualize how these generators work, and implement the χ^2 Test as a measure of the randomness of number sequences. You will code your solutions in `prng.py` and `seqrand.py`. The file `cs3430_s24_hw10_uts.py` in the zip contains my unit tests. Try work on one unit test at a time. Easy does it.

Problem 0: (0 points)

Review the Lectures 21 and 22 PDFs and/or you class notes to become comfortable with LCG, Xorshift, Mesenne Twister, χ^2 (chi-square), mathematical expectation, and the χ^2 test of sequence randomness. I made a few additions (examples) to and corrections (a few typos) in the slides of these lectures and included them in the zip for your convenience and ease of reference.

Problem 1: LCG, XORSHIFT, Mersenne Twister (1 point)

Implement the static method `lcg(a, b, m, n, x0=0)` in `prng.py` that generates random numbers with the LCG method we discussed in Lecture 21. This method takes the parameters a, b, m explained in the slides. The parameter n specifies how many numbers will be generated and the keyword `x0` defines the seed. This method returns a generator (i.e., a real Python generator) to generate n random numbers. Here is a trial run in which an LCG generator generates 5 random numbers with the seed of 0.

```
>>> from prng import prng
>>> a, b, m, n, seed = 214013, 2531011, 4294967296, 5, 0
>>> lcgg = prng.lcg(a, b, m, n, x0=seed)()
>>> rns = [next(lcgg) for _ in range(n)]
>>> rns
[2531011, 505908858, 3539360597, 159719620, 2727824503]
```

Implement the static method `xorshift(a, b, c, n, x0=1)` in `prng.py` that generates random numbers with the 32 xorshit method discussed in Lecture 21. This method takes the parameters a, b, c. The parameter n specifies how many numbers will be generated and the keyword `x0` defines the seed. This method returns a Python generator that generates n random numbers. Here is a trial run of my implementation.

```
>>> from prng import prng
>>> a, b, c, n, seed = 1, 3, 10, 5, 1
>>> xsg = prng.xorshift(a, b, c, n, x0=seed)()
>>> rns = [next(xsg) for _ in range(n)]
>>> rns
[1, 3075, 5898885, 3488497534, 2316485042]
```

Implement the static method `mersenne_twister(n, x0=1, start=0, stop=1000)` in `prng.py` that generates random numbers with Python's Mersenne Twister method. This method can be implemented with Python's `random.randint(start, stop)` that to generate random integers in the range `[start, stop]`. This method also takes the parameters n (the number of random numbers to generate), `x0` (the seed), `start` (the lower bound of the range), `stop` (the upper bound for the range). This method returns a Python generator that generates n random numbers. Here's a trial run.

```
>>> from prng import prng
>>> seed, start, stop, n = 1, 0, 1000, 5
>>> mtw = prng.mersenne_twister(n, x0=seed, start=start, stop=stop)()
>>> rns = [next(mtw) for _ in range(n)]
>>> rns
[137, 582, 867, 821, 782]
```

Problem 2: χ^2 Tests of Sequence Randomness (2 points)

Let us compute the randomness of the sequence of the first 10 digits in the π mantissa computed with Chudnovsky (cf. Slides 13, 18, 19 in Lecture 22).

```
>>> from seqrand import *
>>> pim = get_mantissa('pi.txt') ## get the first 99,999 pi mantissa
                                 ## digits into a string.
>>> len(pim)
99999
>>> pim10 = pim[:10] ## get the first 10 pi mantissa digits into a string
'1415926535'
### Get the actual counts in the table on Slide 12, Lecture 22
>>> Y0,Y1,Y2 = pim10.count('0'),pim10.count('1'),pim10.count('2')
>>> Y3,Y4,Y5 = pim10.count('3'),pim10.count('4'),pim10.count('5')
>>> Y6,Y7,Y8 = pim10.count('6'),pim10.count('7'),pim10.count('8')
>>> Y9 = pim_10.count('9')
### Compute V and its p-Value (Cf. Slides 18-19, Lecture 22)
>>> import scipy.stats
>>> scipy.stats.chisquare([Y0,Y1,Y2,Y3,Y4,Y5,Y6,Y7,Y8,Y9],
                          [1 for _ in range(10)])
Power_divergenceResult(statistic=8.0, pvalue=0.5341462169096916)
### A much shorter way of doing the same thing with list comprehension
>>> v, pv = scipy.stats.chisquare([pim_10.count(str(d)) for d in range(10)],
                                  [1 for _ in range(10)])
>>> v
8.0
>>> pv
0.5341462169096916
```

Since the p-value for Vc (i.e., 6) is in [0.1, 0.9], the sequence is considered *sufficiently random* by Dr. Knuth's criteria of number sequence randomness (cf. Slide 17, Lecture 22).

We can test the randomness of an arbitrarily long subsequence of the first 99,999 π mantissa digits. Here is an example. Note that the actual values of the expectations depend on the length of the sequence, because the length of the sequence is the number of trial runs in our experiment.

```
## choose a sequence of 13 trials that starts at 3 and ends
## at 15.
>>> pim[3:16]
'5926535897932'
>>> len(pim[3:16])
13
>>> pim_3_16 = pim[3:16]
>>> Y_counts = [pim_3_16.count(str(d)) for d in range(10)]
>>> seq_len = 13
## Note that seq_len is n in the formula on Slide 9, Lecture 22.
>>> E_counts = [seq_len*0.1 for _ in range(10)]
>>> Y_counts
[0, 0, 2, 2, 0, 3, 1, 1, 1, 3]
>>> E_counts
## We have 13 trials and the probability of seeing
## each digit is 1/10. So, E(Y_i) = seq_len * 0.1 = 1.3
[1.3, 1.3, 1.3, 1.3, 1.3, 1.3, 1.3, 1.3, 1.3, 1.3]
>>> v, pv = scipy.stats.chisquare(Y_counts, E_counts)
>>> v
9.307692307692307
>>> pv
0.4093699783604269
```

Thus, by Dr. Knuth's criteria, the sequence 5926535897932 in the π mantissa that starts at position 3 (0-based counting) and ends at position 15 is also sufficiently random.

Let us run the same sequence randomness tests on the sequences that starts at the same positions in the e mantissa.

```
>>> from seqrand import *
>>> em = get_mantissa('e.txt')
>>> len(em)
9999
>>> em10 = em[:10]
>>> em10
'7182818284'
```

```
>>> import math
>>> math.e ## for illustartion and comparison
2.718281828459045
>>> v, pv = scipy.stats.chisquare([em10.count(str(d)) for d in range(10)],
                                  [1 for _ in range(10)])
>>> v
16.0
>>> pv
0.06688158777412662
```

Thus, by Dr. Knuth's criteria, the sequence 7182818284 is not sufficiently random. We move on to the sequence from position 3 up to position 15.

```
>>> em_3_16 = em[3:16]
>>> len(em_3_16)
13
>>> Y_counts = [em_3_16.count(str(d)) for d in range(10)]
>>> E_counts = [13*0.1 for _ in range(10)]
>>> v, pv = scipy.stats.chisquare(Y_counts, E_counts)
>>> v
9.307692307692307
>>> pv
0.4093699783604269
>>> em_3_16
'2818284590452'
```

Since p-value is in $[0.1, 0.9]$, 2818284590452 is sufficiently random.

Use the teachniques discussed above to implement the function `chisquare_str_digit_seq()` in `seqrand.py`. This function takes a string of digits `str_digit_seq` and returns the χ^2 statistic V and its p-value of the substring that starts at position `seq_start` and ends at position `seq_end`-1 in `str_digit_seq`. A few sample runs are below.

```
>>> from seqrand import *
>>> pim = get_mantissa('pi.txt')
>>> em  = get_mantissa('e.txt')
>>> v, pv = chisquare_str_digit_seq(pim, seq_start=0, seq_end=10)
>>> v, pv
(8.0, 0.5341462169096916)
>>> v, pv = chisquare_str_digit_seq(em, seq_start=0, seq_end=10)
>>> v, pv
(16.0, 0.06688158777412662)
>>> v, pv = chisquare_str_digit_seq(pim, seq_start=3, seq_end=16)
>>> v, pv
(9.307692307692307, 0.4093699783604269)
>>> v, pv = chisquare_str_digit_seq(em, seq_start=3, seq_end=16)
>>> v, pv
(9.307692307692307, 0.4093699783604269)
```

Problem 3: Randomness of Mersenne Twister (1 point)

Implement the function

```
chisquare_mersenne_twister(pval_lower=0.1, pval_upper=0.9,
                           seq_len=10, num_experiments=10)
```

in `seqrand.py` that uses your implementation in `prng.mersenne_twister()` to estimate the percentage of the sufficiently random sequences of the digits from 0 to 9 of specific length generated with the Mersenne Twister PRNG used in Python. The keywords `pval_lower` and `pval_upper` specify the lower and upper bounds of the interval of sufficient values of the p-values. The keyword `seq_len` specifies the length of the random sequence in each experment. This is the number of trials in each experiment. The number of experiments is how many times we perform the experiment, or, to put it differently, the number of random sequences we generate. If the p-value of the V statistic of a random sequence is in `[pval_lower, pval_upper]`, then the sequence is counted as sufficiently random. The function returns the actual count of the sufficient random sequences and the percentage of the sufficient random sequences, i.e., the ratio of the actual count and the number of the experiments.

```
>>> from seqrand import *
>>> count, percent = chisquare_mersenne_twister(seq_len=10,
                                                 num_experiments=100)
>>> count, percent
(100, 1.0)
```

I got between 80% and 100% of the sequences in all the unit tests for this problem as sufficiently random with the p-values in $[0.1, 0.9]$. Can we test the χ^2 randomness of sequences generated with `LCG` or `XORShift`? Yes, we certainly can. But we will skip it for this assignment, because, as you may have noticed after completing Problem 1, random numbers generated with `LCG` and `XORshift` get pretty big pretty fast, which means that we will have to scale, which means we will have to round, which means that we will have to deal with numerical instability. And, since numerical instability is not really the point of this assignment, we will move on.

Problem 4: Random Number Art (1 point)

Let us do some random art! One way to estimate the randomness of a PRNG is to turn the generated random numbers into an image and display it. If we do not see any patterns in the image, then the numbers are likely random. If we see a pattern, then the numbers are likely not random. We can also forget about randomness for a while and enjoy the generated images as works of art.

The method `gen_lcg_data(a, b, m, n, seed=0, option=1)` uses the LCG method to generate n numbers for a PIL image.

```
def gen_lcg_data(a, b, m, n, seed=0, option=1):
    lcgg = prng.lcg(a, b, m, n, x0=seed)()
    ### option 1) generate n lcg numbers.
    if option == 1:
       return np.array([next(lcgg) for _ in range(n)])
    ### option 2) generate n lcg numbers by
    ### varying the seed from i upto n-1.
    elif option == 2:
       data = np.zeros(n)
       for i in range(n):
           data[i] = next(prng.lcg(a, b, m, 1, x0=i)())
       return data
    ### option 3) generate n numbers with numpy arange.
    elif option == 3:
       return np.arange(n)
    else:
       raise Exception('prng.get_lcg_data(): option must be 1,2,3')
```

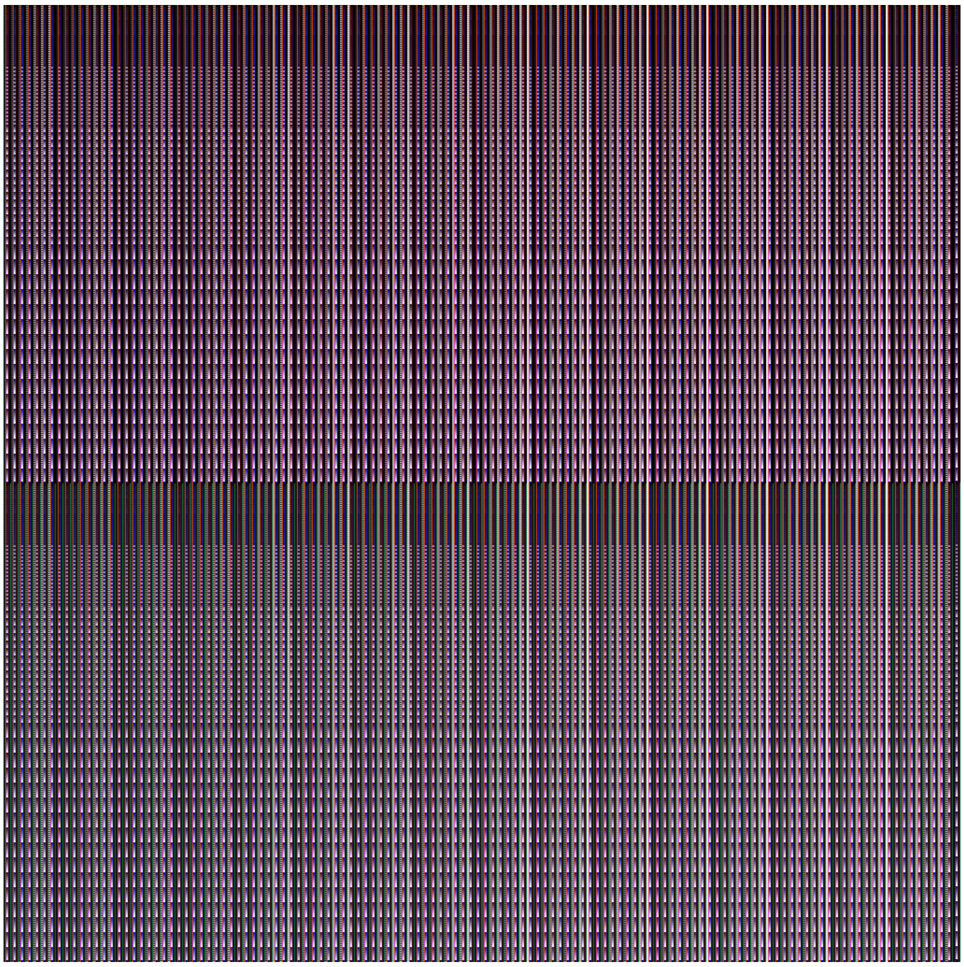
Figure 1: Image generated with random_pil().

When `option==1`, we generate n numbers from one seed. When `option==2`, we generate n numbers with the seed that ranges from 0 upto $n-1$. When `option==3`, we simply generate an array of numbers from 0 upto $n-1$.

Follow the method `gen_lcg_data(a, b, m, n, seed=0, option=1)` as an example to implement the methods `gen_xorshift_data()` and `gen_mersenne_twister_data()` that generate n numbers with the xorshift and Mersenne Twister methods, respectively. Each should implement the same three options.

You can use the method `random_pil()` in `prng.py` to turn an array of random numbers into a PIL image and display it or save it in the current directory. The PIL image constructor will make sure that all the numbers are in [0, 255], which means that there will be some loss of information.

My favorite image for this assignment is the one in Figure 1 generated with on of the unit tests. It is not really random, because I can see a pattern and I love it! I included in the zip my random PILs cooked by the unit tests for this problem.

Quotes on Randomness and Probability Theory

A few famous quotes to think about as you work on this assignment.

```
In no other branch of mathematics is it so easy
for experts to blunder as in probability theory.

                                 Martin Gardner

Probability is expectation founded upon partial
knowledge. A perfect acquaintance with all the
circumstances affecting the occurences of an
event would change expection into ceratinty
and leave neither room nor demand for a theory
of probabilities.

                                 George Boole

No amount of experimentation can ever prove
me right; a single experiment can prove me
wrong.

                             Albert Einstein

God does not play dice.

                             Albert Einstein
```

What To Submit

Submit your code in `prng.py` and `seqrand.py`.

Happy Hacking and Enjoy Random (or should we say Pseudorandom?) Art!