Tausworthe PRN report

Student: Andrei Goncharov (agoncharov6@gatech,edu)

Problem statement

Implement the Tausworthe pseudo-random number generator. Perform a decent number of statistical tests on the generator to see that it gives PRN's that are approximately i.i.d. Uniform(0,1). Plot adjacent PRN's (Ui, Ui+1), i = 1, 2, . . ., on the unit square to see if there are any patterns. Generate a few Nor(0,1) deviates (any way you want) using Unif(0,1)'s from the Tausworthe generator.

Approach

1. Implement a base class for thr Tausworthe generator.
2. Implement a simple CLI to interact with the class
3. Add commands to the CLI to:
    1. Generate a sequence of PRNs (pseudo-random numbers) and print them.
    2. Generate PRNs and test them. Print the graphs. Generate Nor(0, 1) deviates.

CLI

Using python-nubia create a simple CLI. Read more about it in the README.

> CLI has limitations! It currently does not do any input validation. If you enter bad data, you are on your own!

Tausworthe generator class

Accepts initial `seed`, `q`, `r`, `l`. Exposes a single method `next` that returns a new `U(i)`. Internally, keeps the current number of `q` bits as `seq`. To generate a new PRN it appends `l` new bits to `seq` and pop `l` first bits. `seq` is initialized to `seed`.

Every new bit is calculated by the algorithm: `B(i) = B(i - q) xor B(i - r)`.

Every new PRN requires `l` new bits and is calculated as `l` converted from binary to decimal divided by `2^l`.

Statistical tests

We will perform a goodness-of-fit test (with a = 0.05 and 101 bins) and an independence test (runs "Up and Down") (with a = 0.05).

> We use 101 bins so our chi squared distribution has 100 degrees of freedom.

Choosing constants

Tausworthe algorithm uses several constants:

- `q` - big number of offset bits
- `r` - little number of offset bits

- `l` - little number of offset bits

`q` must be as big as possible as we generally want as big as possible cycles for our PRN generators. So we set `q` to `31` bits by default. It gives us a cycle of `2^31-1` which is the biggest any `32-bit` machine can handle.

`r` and `l` can vary. Due to the limited scope of the project, we set `l` to `20`. It makes the generator produce `2^20` different PRNs which sounds like a reasonable number.

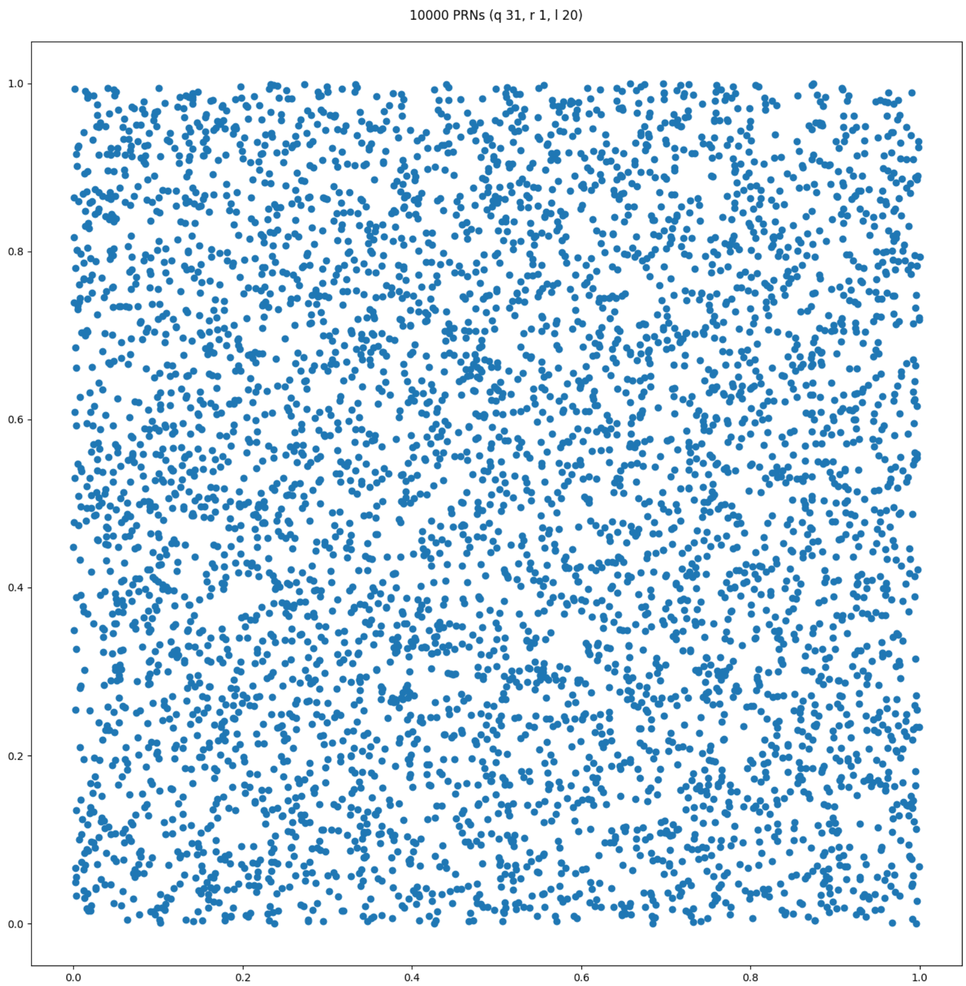
For `r` we will perform a few tests to see if value of `r` significantly affects the result.

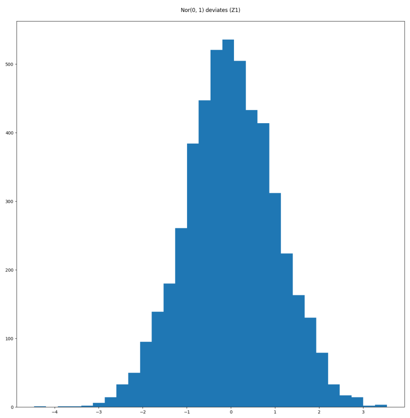
Testing `r`

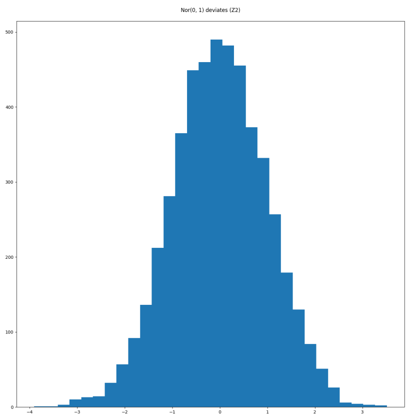
Using `42424242424242` as our seed. We will generate 10000 numbers. Based on the constants listed above, our reference (threshold) Z-score is `1.96`. Reference chi squared value is `124.342`.

`r = 1`

Chi^2 = `97.7578`. Number of runs = `6708`.

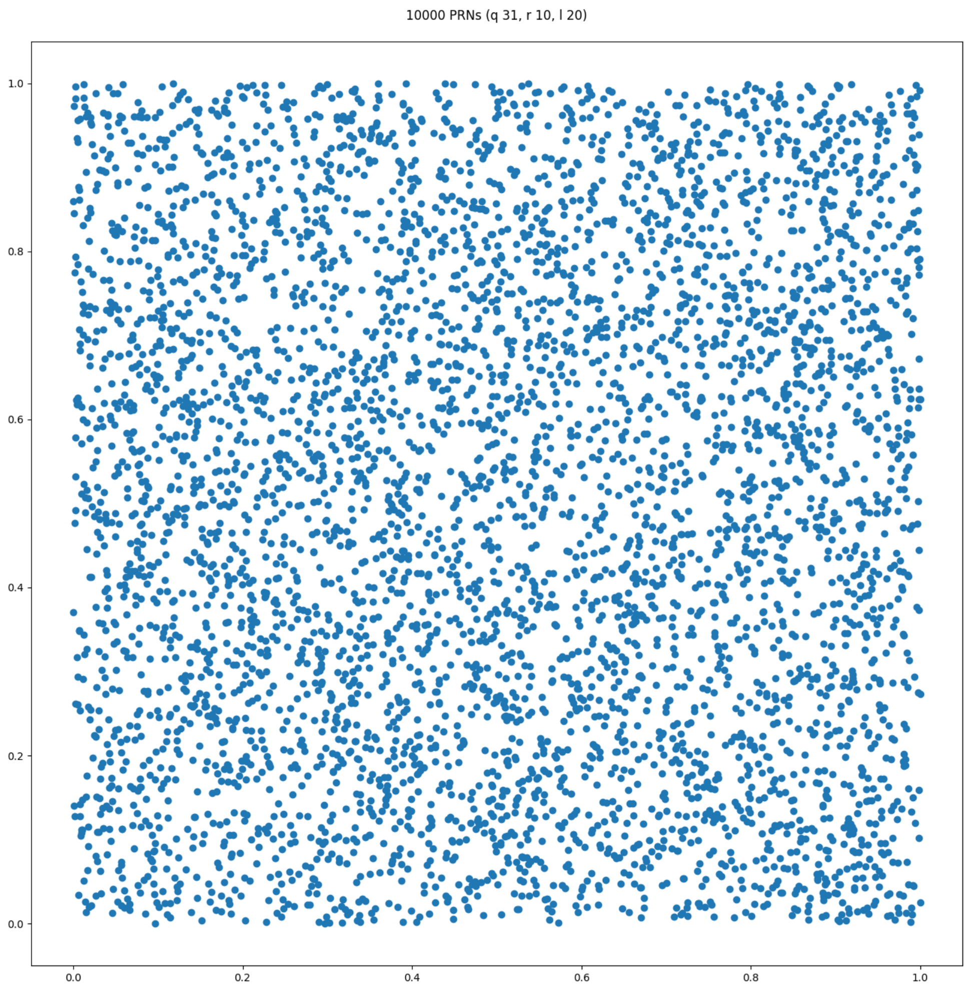


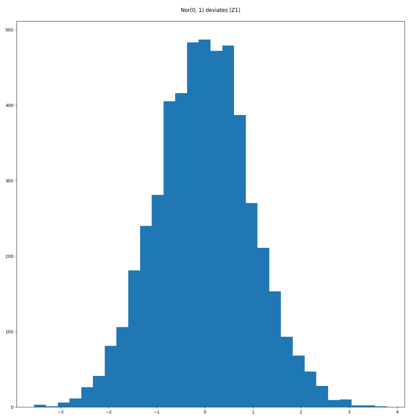


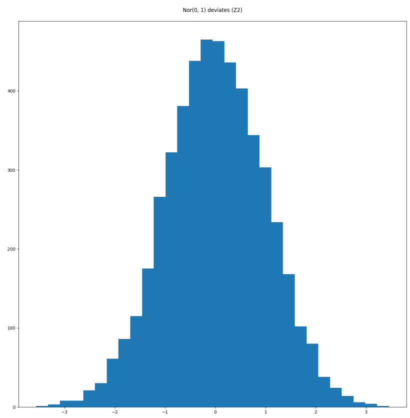


`r = 10`

Chi^2 = `77.75`. Number of runs = `6659`.

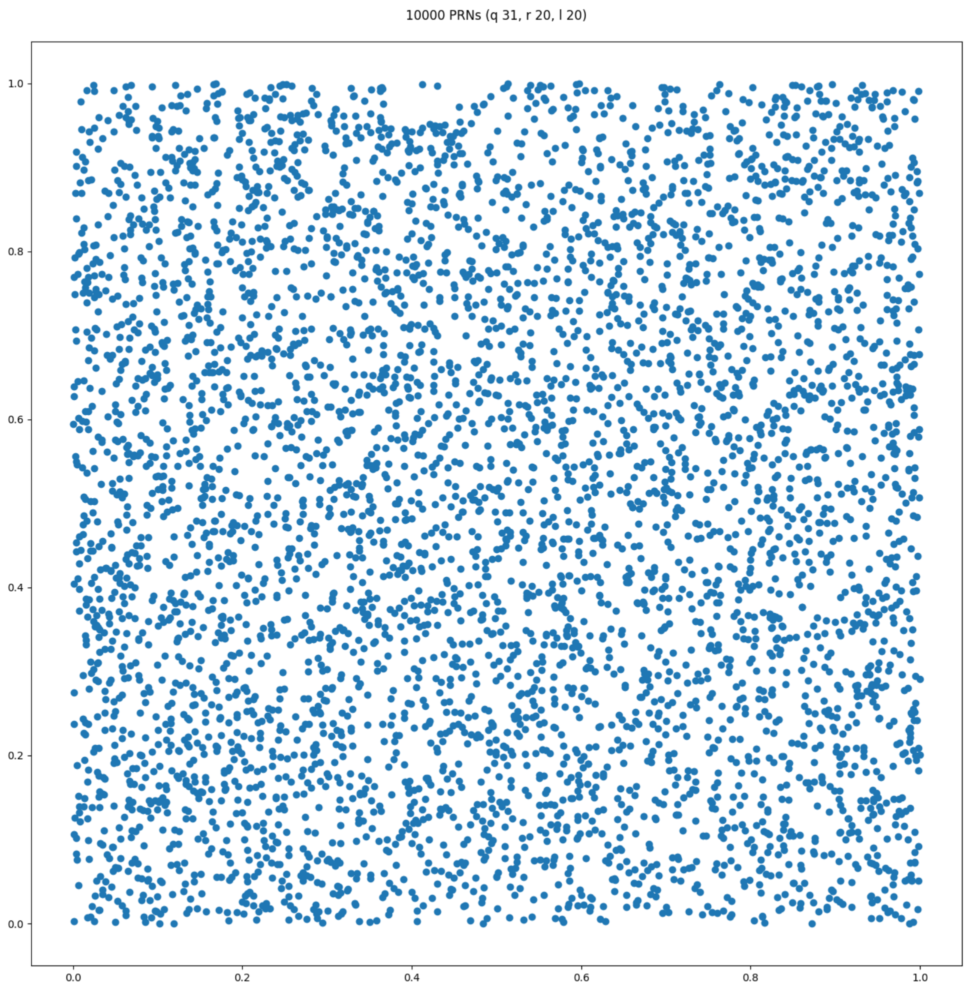


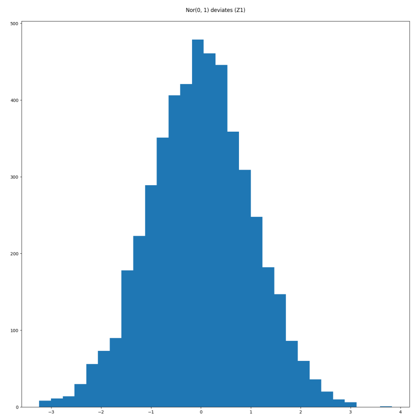


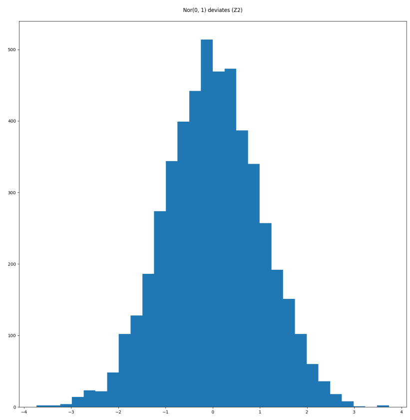


`r = 20`

Chi^2 = `100.6868`. Number of runs = `6612`.

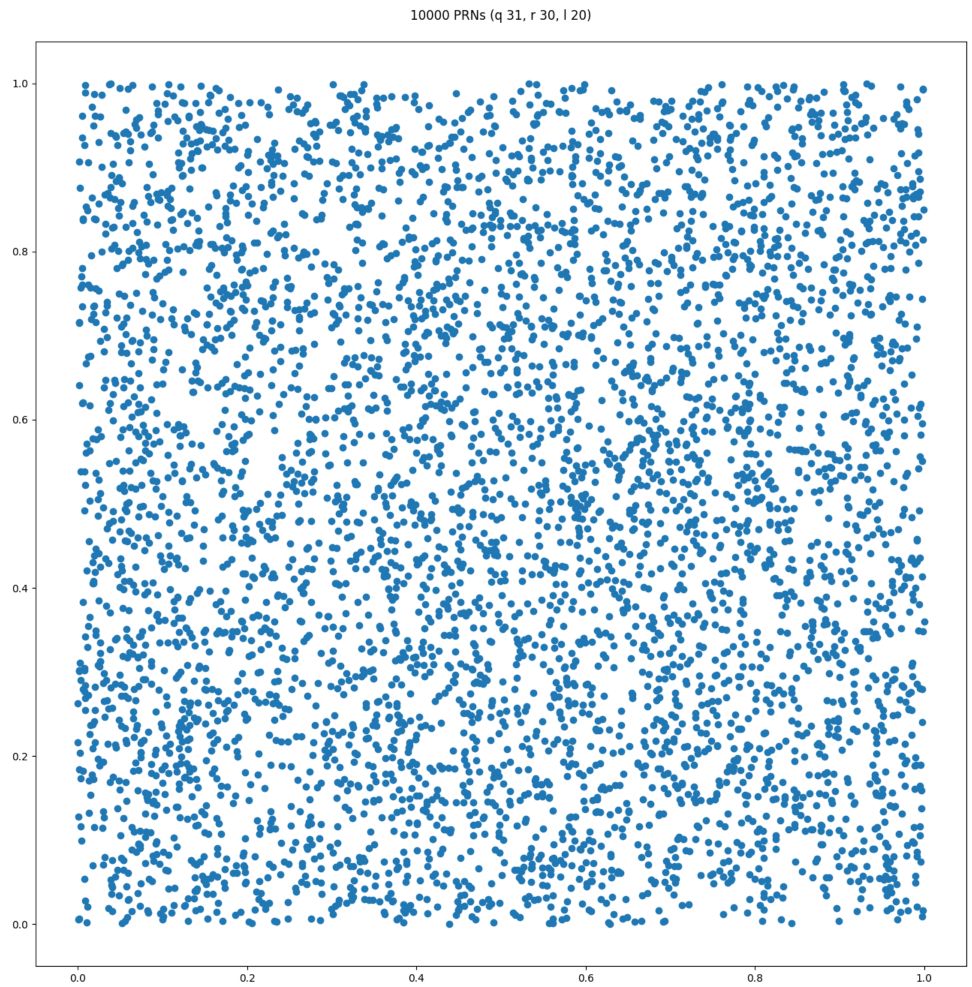


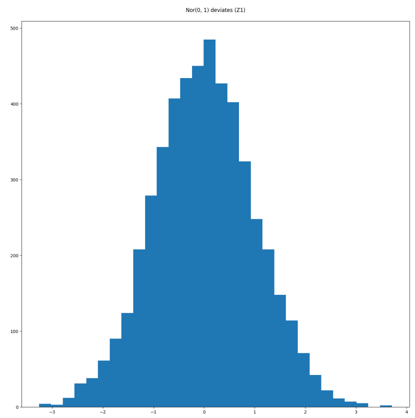


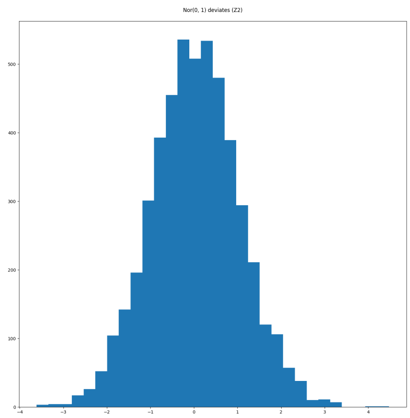


`r = 30`

Chi^2 = `104.89`. Number of runs = `6643`.







Summary

As we can see, our Tausworthe generator produces PRNs that pass goodness-of-fit and independence tests, thus we conclude that the implementation is correct. We can also notice that different values of `r` did not affect the out come significantly, thus we conclude that `r` does not affect generator properties and can be anything between `1` and `q - 1`. Finally, we can clearly see from the graphs that we can generate Nor(0, 1) deviates using Box-Muller method from the PRNs produces by our generator.

References

1. https://studio.edx.org/assets/courseware/v1/f33427e02d57018e66a65baa3c8305c5/asset-v1:GTx+ISYE6644x+2T2019+type@asset+block/Module06-RandomNumberGenerationSlides_180526.pdf