Liam Kolber

Mark Rast

ASTR 3800

7 September 2016

How Random is Random?

In this first project students were tasked with writing code in python that would generate a sequence of 1000 random numbers. This was to be done using two separate methods: the already implemented function within python's numpy library (numpy.random.uniform()), and an algorithm created by John von Neumann. With 3 predetermined seeds, 3 list were generated of random numbers for each method. These generations were then plotted and tested for effectiveness using a χ^2 comparison, and this was checked by calculating a probability to exceed.

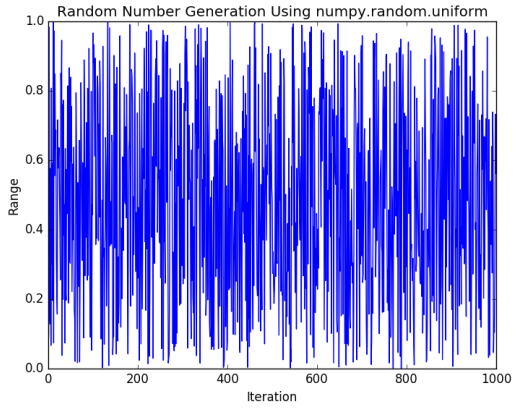

Figure 1: First random number generation using numpy.random.uniform() with a seed of 414929

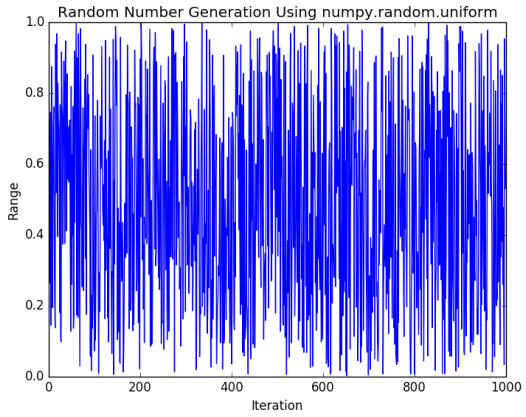

Figure 2: Second random number generation using numpy.random.uniform() with a seed of 581714

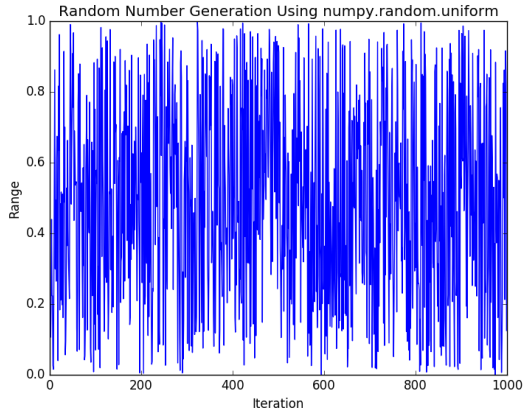

Figure 3: Third random number generation using numpy.random.uniform() with a seed of 698137

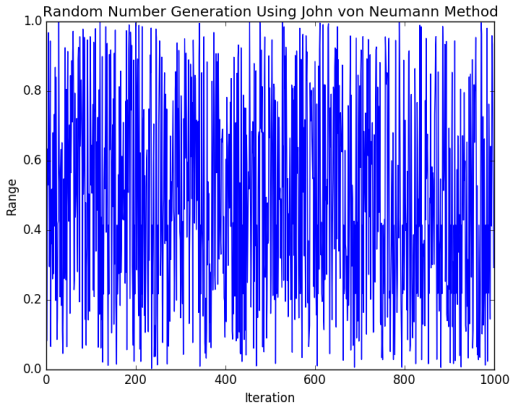

Figure 4: First random number generation using John von Neumann method with a seed of 414929

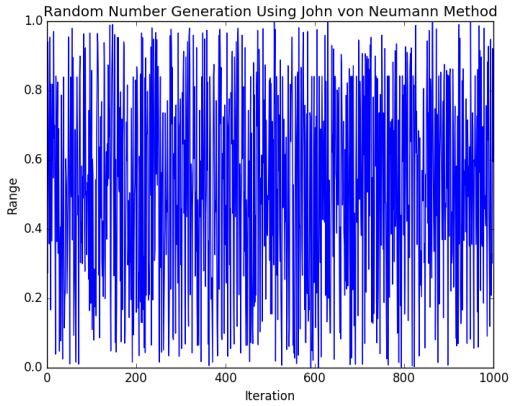

Figure 5: Second random number generation using John von Neumann method with a seed of 581714

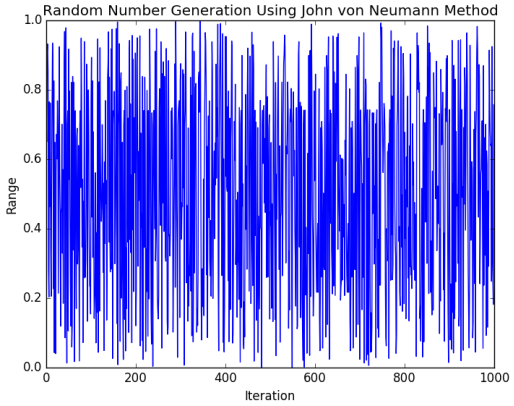

Figure 6: Third random number generation using John von Neumann method with a seed of 698137

For the most part these plots appear to not have any pattern. Figure 3 appears to have a slight sinusoidal pattern over the 1000 generations, but this is a bit of a stretch. One thing to note

in the John von Neumann method is that numbers tend to group toward the center range of numbers and don't frequently reach the edges of the plot.

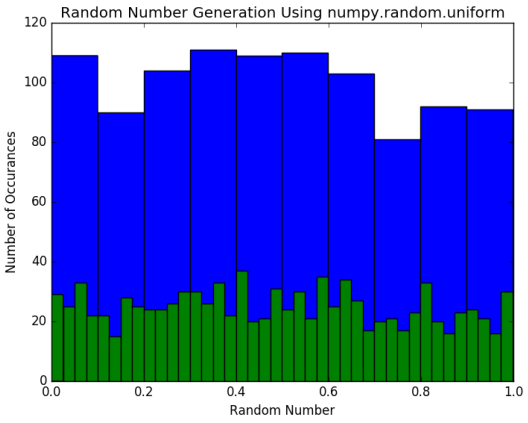

Figure 7: Histogram of numpy.random.uniform() data with seed of 414929 and with 10 and 40 bins.

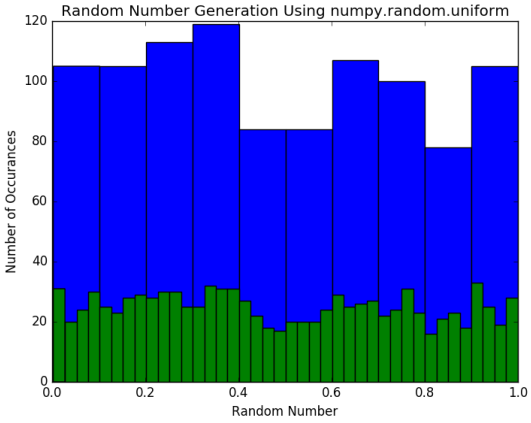

Figure 8: Histogram of numpy.random.uniform() data with seed of 581714 and with 10 and 40 bins.

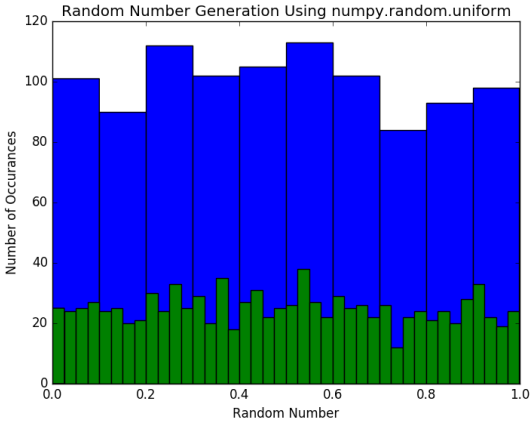

Figure 9: Histogram of numpy.random.uniform() data with seed of 698137 and with 10 and 40 bins.

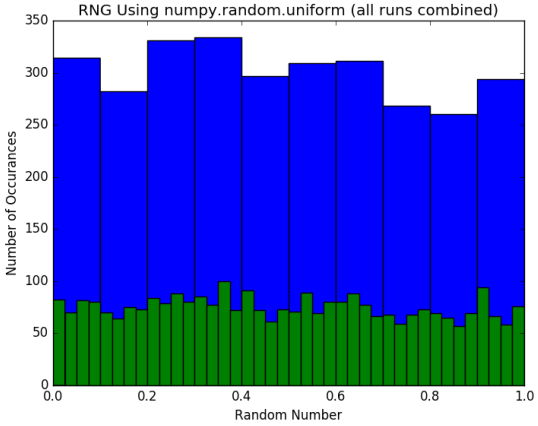

Figure 10: Histogram of numpy.random.uniform() data compiled into single list of data with 10 and 40 bins.

These plots that were generated using python's numpy random number generator function. Figures 7-9 individually have their inconsistancies with the most obvious being in Figure 8. Interestingly enough though, the 10 bin plot for Figure 8 is the most inconsistant, yet its 40 bin plot appear to be the most consistant of the three. When these three data sets are combined, the bin levels appear to level out much more nicely.

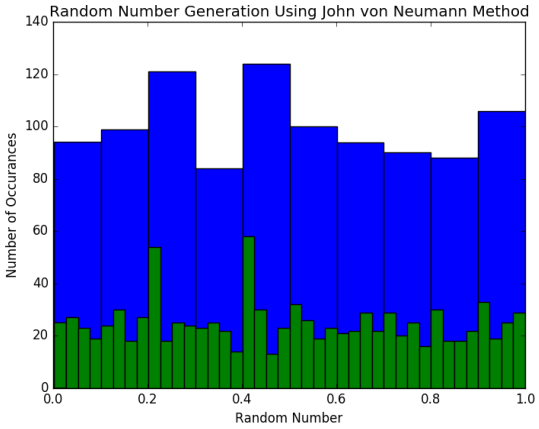

Figure 11: Histogram of John von Neumann data with seed of 414929 and with 10 and 40 bins.

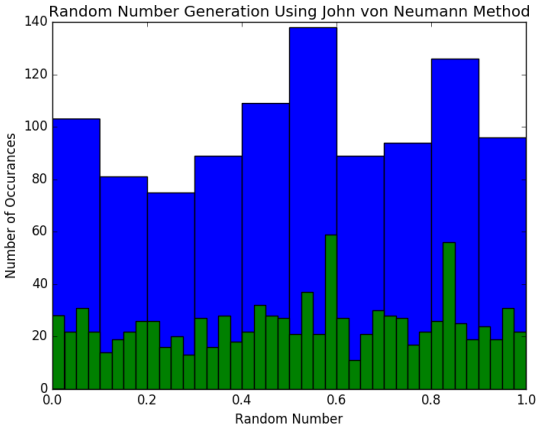

Figure 12: Histogram of John von Neumann data with seed of 581714 and with 10 and 40 bins.

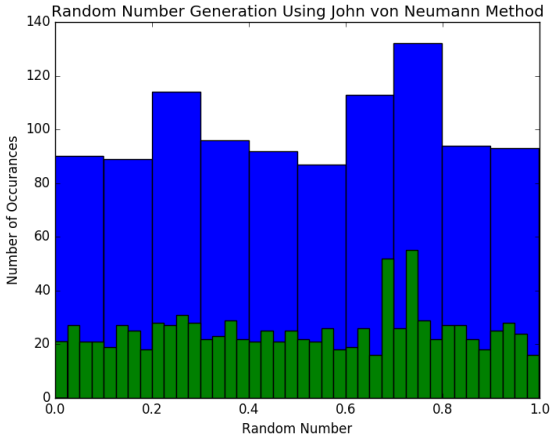

Figure 13: Histogram of John von Neumann data with seed of 698137 and with 10 and 40 bins.

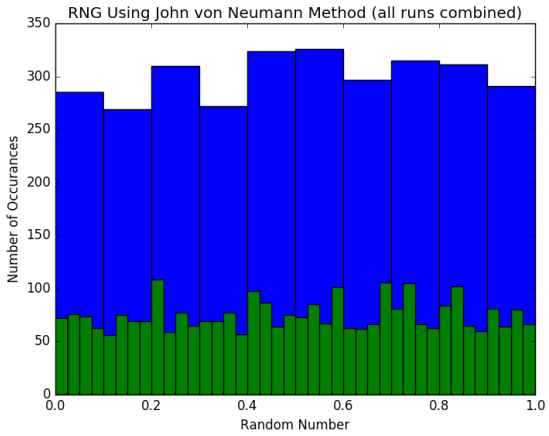

Figure 14: Histogram of John von Neumann data compiled into single list of data with 10 and 40 bins.

These John von Neumann plots are a little more interesting. As can be seen rather clearly, Figures 11-13 have bins that are much more filled than their surroundings which becomes more obvious in the when looking at the 40 bin portions of the plots. Fortunately, when combining all the data together, there appears to be a more uniform randomness to the data generation over 3000 numbers as opposed to 1000 number sequences.

The χ^2 test is used to compare the observed and expected PDFs to determine the effectiveness of these number generators by calculating that χ^2's probability to exceed (P). When P is close to or equal to zero it means that the calculated χ^2 is so great that the chance of getting anything higher is nearly 0%. In other words, the amount in each bin is so drastically uneven or different than what should be expected, that the generation is not uniformly random. When P is close to or equal to one, the distribution of random numbers is so evenly distributed that the generation is not truly random (though technically it is still possible to get a random generation that does have such a distribution). As evident from the χ^2 test of the frequency test (in which we checked the bin distribution of the histograms with 10, 40, and 100 bins) that was printed to the terminal during the program, the distributions for the numpy function are fairly decent with an increasingly worse χ^2 (higher) as the number of bins increased but still low enough that it passes the test. With the John von Neumann method, however, the χ^2 values that were generated weren't as promising, and as the number of bins increased, the probability to exceed (P) approached zero which leads to the conclusion that it fails the test.

We then had to check to see if any two numbers in a row were placed in either the same bin or in adjacent bins by creating our own method of dividing up our numbers similarly to a histogram. When this was done, we expected that with a 10 bin distribution (each bin a size of 0.1) that there should be on average about 10 "same-bin" placements and 10 "adjacent-bin" placements. With these expected values, a χ^2 could then be calculated for each of the randomly generated lists. As evident from the χ^2 and P values printed to the terminal during the program, the values for the "same-bin" placements are fairly decent (not too close to 1 or zero) for the numpy function meaning that it passes the test. On the other hand, when looking at the results for the "adjacent-bin" placements, the values calculated for χ^2 and P were less than acceptable. This must be the

result of poor logic when checking neighboring bins, however, after many attempts at reworking the logic, no success was had. Unfortunately, in regards to the great "same-bin" results, the same couldn't be said for the John von Neumann data which fails this test due to extreme χ^2 values at increasing bin numbers and P values of nearly zero. This means that the numpy function is a much more effective random number generator.

Thought Questions

1. When a sequence of numbers is pseudorandom, it is a sequence in which the numbers aren't truly generated randomly. The numbers are generated through an algorithm that aims to mirror randomness by passing various statistical tests. The main hope when it comes to pseudorandom number generators is that even an infinitely generated sequence of numbers wouldn't fall into any apparent periodicities or flat lines. Fortunately, my sequences don't appear to exhibit either of these qualities. This means that my sequences are at least good enough for a thousand number sequences.
2. The human brain is built around pattern recognition. This trait unfortunately leads to many subconscious pattern creations when entering numbers randomly particularly if the numbers are thought up prior to entering. Likewise, if the keyboard is just hit randomly, the brain might still aim to create some similar patterns in hand movement which would create similar numbers over the course of many entries. I think that thinking up the numbers individually could lead to a more random sequence than randomly hitting the keyboard because one can purposefully choose numbers that could avoid any potential patterns when looking at the whole sequence.
3. The χ^2 comparison is heavily dependent on each individual value that it is given. This means that outliers have a significant impact on the end values of these tests. In order to

improve the χ^2 values that were seen, conditions could be put in the code to remove extreme outlying values. This would create a more uniform distribution with a much smaller χ^2.

4. When a short sequence is apparently random, it is unknown if it maintains randomness outside of the given range. A longer sequence follows the same limitations; however, it is known to hold randomness for a longer sequence of numbers. When taking a small snippet from a larger truly random sequence, no real conclusion can be made of the complete sequence as one can't assume no periodicities or flat lines would occur.
5. With a pseudorandom number generator one has more control over just how random they want their numbers to be while one has no control over the randomness of a radioactive material. In other words, one can place limits more easily on their generation algorithm to fit the need of their program (i.e. ranges or changes in values). Likewise, changes can be made to the sequence (i.e. using seeds to recreate specific sequences) in order to get consistent tests results to ensure other aspects of the program run as anticipated.

Nothing in this project ended up being too overbearing. Everything was doable in the allotted time and actually very helpful in reviewing the language. Overall I'd say this project is good and no changes are necessary.