1 Pitfalls of pseudo-random number generation

Consider the linear congruential random number generator RANDU, introduced by IBM in System/3660 mainframes i nthe early 60s. (Donald Knuth called this random number generator "really horrible", and it is indeed notorious for being one of the worst generators of all time). The recursion relation of RANDU is defined by

$$I_{i+1} = (65539 \cdot I_i) \bmod 2^{31} \tag{1}$$

and needs to be started from an odd integer. The obtained integer values can be mapped to pseudo-random floating point numbers $u_i \in [0, 1]$ through

$$u_i = I_i / 2^{31} \tag{2}$$

a) Implement this number generator. Make sure that you do not use the 32-bit integer arithmetic, otherwise overflows will occur. (Use 64-bit integer arithmetic instead, or double precision for simplicity - its precision is sufficient to represent the relevant integer range exactly)

The next "random" integer can be generated from a given input integer via the following python code:

```
from numpy import uint64, power

def get_next_RANDU_int(previous_int):
    # make sure all variables are 64-bit floats
    previous_int = uint64(previous_int)
    a = uint64(65539)
    b = uint64(2)
    c = uint64(31)

    return (a * previous_int) % power(b, c)
```

b) Now generate 2-tuples of successive random numbers from the sequence generated by the generator, i.e. $(x_i, y_i) = (u_{2i}, u_{2i+1})$. Generate 1000 points and make a scatter plot of the points in the unit square. Does this look unusual? How does this look in 3D (using 3-tuples)?

First, we create a list of the first 1000 "random" integers. The first entry in this list is 1, the remaining entries can be calculating using the function we just defined above.

```
def get_list_of_N_random_ints(N):
    list_of_random_ints = [uint64(1)]

    for _ in range(N):
        i = list_of_random_ints[-1]
        j = get_next_RANDU_int(i)
        list_of_random_ints.append(j)

    return list_of_random_ints

first_1000_ints = get_list_of_N_random_ints(1000)
```

Now we can convert the integers to floating point numbers and create a list of tuples. For this, we need a "normalization" function which takes an integer and converts it to a float between 0 and 1 (by dividing it by 2^{31}):

```
def normalize(num):
    return num / power(uint64(2), uint64(31))

def get_list_of_N_float_tuples(N):
    list_of_random_float_tuples = []
    list_of_random_ints = get_list_of_N_random_ints(N+1)

    for idx, item in enumerate(list_of_random_ints[:-1]):
        a, b = item, list_of_random_ints[idx + 1]

        list_of_random_float_tuples.append(
            (normalize(a), normalize(b))
        )

    return list_of_random_float_tuples
```

Let us now plot these points.

```
import matplotlib.pyplot as plt

def plot_points():
    list_of_random_float_tuples = get_list_of_N_float_tuples(1000)

    x = [i[0] for i in list_of_random_float_tuples]
    y = [i[1] for i in list_of_random_float_tuples]

    plt.figure(figsize=(4, 4))
    plt.xlim(0, 1)
    plt.ylim(0, 1)
    plt.xlabel('x')
    plt.ylabel('y')
    plt.scatter(x, y, color='red', s=3)
    plt.savefig('fig1.pdf')
    plt.close()
```

The resulting figure can be seen on the next page.

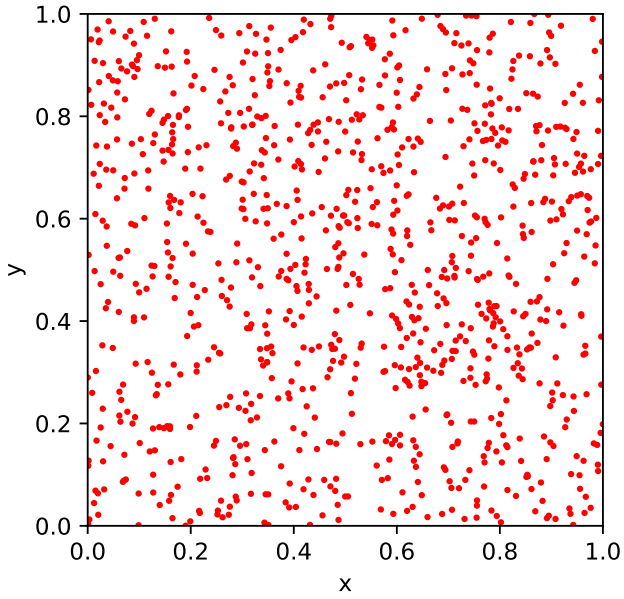

Figure 1: Plot for task 1b)

c) Now zoom in by a large factor onto a small region of the square, for example $[0.2, 0.201] \times [0.3, 0.301]$ **and generate enough points that there are again 1000 points within the small region as before. Interpret the results.**

d) Repeat a) - c) for you favorite standard RNG.

2 Performance of Monte Carlo integration in different dimensions

We would like to compare the performance of the Monte Carlo integration technique with the regular midpoint method. To this end, consider the integral

$$I = \int_V f(\vec{x}) \cdot \mathrm{d}^d\vec{x} \tag{3}$$

where the integration domain V is a d-dimensional hypercube with $0 \leq x_i \leq 1$ for each component of the vector $\vec{x} = (x_1, x_2, ..., x_d)$. The function we want to integrate is given by

$$f(\vec{x}) = \prod_{i=1}^{d} \frac{3}{2} \cdot \left(1 - x_i^2\right) \tag{4}$$

This has an analytic solution of course, which is $I = 1$ independent of d, but we want to ignore this for the moment and use the problem as a test of the relative performance of Monte Carlo integration and ordinary integration techniques. To this end, calculate the integral in dimensions $d = 1, 2, 3, ..., 10$ using

a) ...the midpoint method, where you divide the volume into a set of much smaller hypercubes obtained by subdividing each axis into n intervals, and where you approximate the integral by evaluating the function at the centers of the small cubes.

b) ...standard Monte Carlo integration in d dimensions, using N random vectors (don't use the "wrong" RNG from the previous problem!)

3 Probability transformation & Metropolis Monte Carlo

In this exercise you are asked to transform a flat probability distribution

$$p(x) = \begin{cases} 1 & \text{if } x \in [0,1] \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

into a distribution of the form $p_{\text{new}} = 1/x^2$, realized on the interval $[1, 20]$.

a) Derive an *exact inversion*, so that $y(x)$ is distributed according to $p_{\text{new}}(x)$ if x is drawn from the flat distribution.

b) Use the Metropolis Monte Carlo formalism to achieve the same goal numerically, that is: Create a set of one million random numbers, distributed according to $p_{\text{new}}(x)$ by starting at an arbitrary point in $[1, 20]$ and accepting/dumping new points based on the acceptance criterion you learned about in the script/lecture. To generate new points we suggest using a normal distribution: $x_{i+1} = N(x_i, \sigma)$ with step size $\sigma = 0.1$, but you can also choose other options.

c) Plot your results from a) and b) in the form of histograms and compare to the target distribution $p_{\text{new}}(x)$. Can you observe pathological aberrations, if you set the step size too small for the Metropolis Monte Carlo scheme?