Solutions to Problem Set 2

Fundamentals of Simulation Methods 8 ECTS
Heidelberg University WiSe 20/21

Elias Olofsson
ub253@stud.uni-heidelberg.de

November 17, 2020

1 Pitfalls of pseudo-random number generation [8pt]

Consider the linear congruential random number generator RANDU, introduced by IBM in the early 1960s. The recursion relation is defined by

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

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

$$u_i = \frac{I_i}{2^{31}}. \tag{2}$$

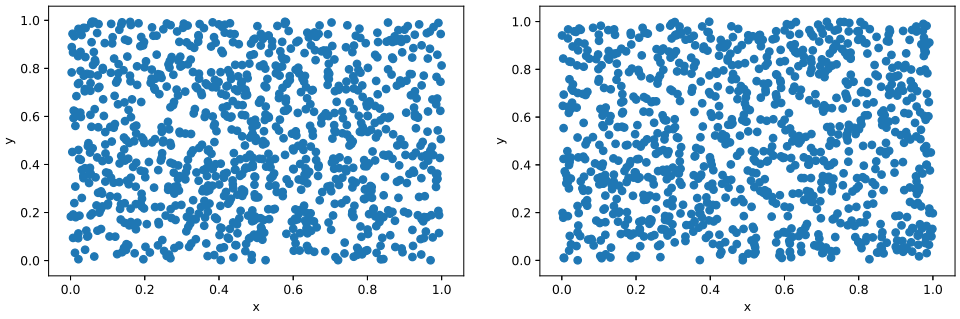
Implement the number generator and generate 2-tuples of successive random numbers from the sequence generated, i.e. $(x_i, y_i) = (u_{2i}, u_{2i+1})$. Plot 1000 points in a scatter plot, how does it look? How does it look in 3D?

Zoom into a small region of the square, e.g. $[0.2, 0.201] \times [0.3, 0.301]$, and generate enough points that there are again 1000 points within this region. Interpret the results. Repeat this process again for your favorite standard random number generator.

For my solutions, please see the submitted Jupyter Notebook `fsm_ex2.ipynb`. First we can note that for small amounts of pseudo-random points, no discernable difference can be found between the RANDU implementation and the native Numpy random number generator, which by default utilizes the Mersenne Twister as the generator. The result when mapping sequential pseudo-random numbers $u_i \in [0, 1]$ to coordinates in the unit square as per $(x_i, y_i) = (u_{2i}, u_{2i+1})$, and generating a 1000 points, can be seen in fig.(1) below.

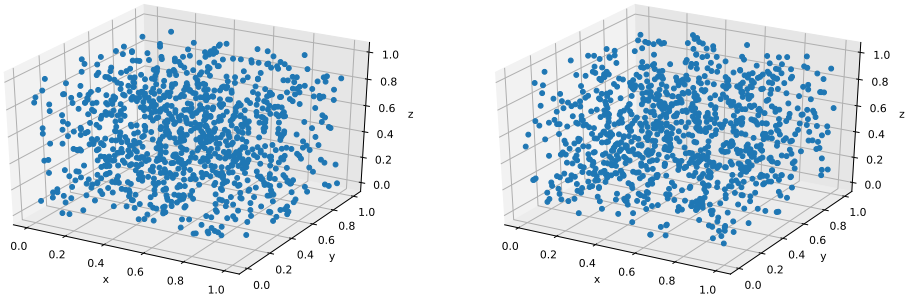
Notably, we cannot identify any problem with either generator from this figure. Even if we plot in 3D, by using 3-tuples $(x_i, y_i, z_i) = (u_{2i}, u_{2i+1}, u_{2i+2})$ as seen in fig.(2), we seemingly cannot draw any conclusion here either about the quality of these random number generators.

However, the problems with the RANDU generator becomes clear if we dramatically increase the number of generated points. In fig.(3) we generate



(a) **(b)**

Figure 1 – Sequential pseudo-random floating point numbers $u_i \in [0, 1]$ mapped to 1000 points in the unit square as per $(x_i, y_i) = (u_{2i}, u_{2i+1})$. To the left we have the RANDU generator and to the right, the Mersenne Twister, the default random number generator in Numpy.

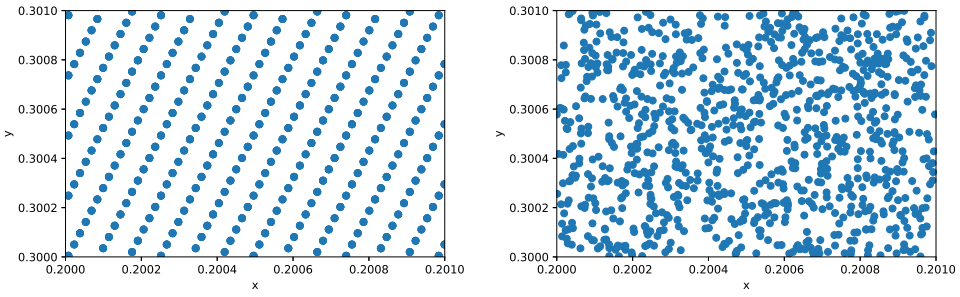


(a) **(b)**

Figure 2 – Sequential pseudo-random floating point numbers $u_i \in [0, 1]$ mapped to 1000 points in the unit cube as per $(x_i, y_i, z_i) = (u_{2i}, u_{2i+1}, u_{2i+2})$. To the left we have the RANDU generator and to the right, the Mersenne Twister, the default random number generator in Numpy.

1 billion random points in the unit square using each of the random number generators.

Since this is 10^6 more points than in the last example, we have to zoom into a 10^{-6} fraction of the unit square, in order to see the underlying result. In our specific case, we show the area $[0.2, 0.201] \times [0.3, 0.301]$, but any similar choice would have sufficed. Clearly, as can be seen in the figure, the RANDU random number generator does not behave very well at all for these large amounts of numbers. Instead, the generated numbers showcase a clear dependence on each other, forming neat rows and a clearly distinguishable patterns. These are not the qualities you desire from a random number generator. Instead you rather want to come as close to "true" randomness as you can get, with each number being drawn seemingly fully independently



(a) **(b)**

Figure 3 – Sequential pseudo-random floating point numbers $u_i \in [0, 1]$ mapped to 10^9 points in the unit square as per $(x_i, y_i) = (u_{2i}, u_{2i+1})$. Due to the large amount of points, the axis are limited to $[0.2, 0.201] \times [0.3, 0.301]$ in order to make the result visible. To the left we have the RANDU generator and to the right, the Mersenne Twister, the default random number generator in Numpy.

from a uniform distribution. The Mersenne Twister, seen for comparison to the right in fig.(3), displays these good qualities. With its insanely long period of $2^{19937} \approx 10^{6001}$ we realistically have a fully non-repeating sequence of random numbers.

2 Performance of Monte Carlo integration in different dimensions [8pt]

Compare the performance of Monte Carlo integration technique with the regular midpoint method. Consider the integral

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

for which the integration domain V is the d-dimensional unit hypercube with $x_i \in [0, 1]$ for all components of the vector $\vec{x} = (x_1, x_2, \dots, x_d)$. The function to integrate is

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

for which the integral in eq.(3) can be analytically be evaluated to 1, independent on the number of dimensions d. Thus, integrate eq.(3) with eq.(4) numerically using the midpoint rule and standard Monte Carlo integration for $d = \{1, 2, \dots, 10\}$ and compare the performance differences of the two techniques. More concretely, subdivide each axis into $n = 6$ intervals for the midpoint method, and use a total amount of $N = 20000$ generated random vectors in d-space for the Monte Carlo technique, and compare the accuracy

of I and the CPU execution time for $d = \{1, 2, \dots, 10\}$.

Please reference my implementation in the attached Jupyter Notebook file `fsm_ex2.ipynb`. Here I have pretty straightforwardly implemented the integration of eq.(3) through the midpoint rule and standard Monte Carlo integration, both for an arbitrary number of dimensions d. Having each of them defined as a function enables me to call them repeatedly with different parameters and measure the execution time needed and judge the relative accuracy between the different methods. Note however that due to the specification of the original question, each of the defined functions take different input parameters. Since N was specified in the context of Monte Carlo, and n for the midpoint rule, we take each of these as the inputs for the corresponding function. Have in mind that we cannot restrict both n and N for the midpoint rule if we want to loop over d, due to the simple relation $N = n^d$.

Letting the code execute with the parameters specified to $n = 6$ for the midpoint rule and $N = 20000$ for Monte Carlo, both for dimensions $d = \{1, 2, \dots, 10\}$, we get the following printout.

```
Midpoint rule:
 1D:  I = 1.00347 ,    Time = 0.000310898  s
 2D:  I = 1.00696 ,    Time = 0.000158072  s
 3D:  I = 1.01045 ,    Time = 0.000241995  s
 4D:  I = 1.01396 ,    Time = 0.00128937   s
 5D:  I = 1.01748 ,    Time = 0.000669241  s
 6D:  I = 1.02102 ,    Time = 0.00114393   s
 7D:  I = 1.02456 ,    Time = 0.00477958   s
 8D:  I = 1.02812 ,    Time = 0.0278454    s
 9D:  I = 1.03169 ,    Time = 0.140695     s
10D:  I = 1.03527 ,    Time = 0.855845     s

Monte Carlo:
 1D:  I = 1.00466 ,    Time = 0.000431299  s
 2D:  I = 1.00585 ,    Time = 0.000947952  s
 3D:  I = 1.00839 ,    Time = 0.00123549   s
 4D:  I = 1.00544 ,    Time = 0.00139928   s
 5D:  I = 1.00397 ,    Time = 0.00140095   s
 6D:  I = 0.999198,    Time = 0.00151944   s
 7D:  I = 0.995385,    Time = 0.00187659   s
 8D:  I = 0.99154 ,    Time = 0.00215268   s
 9D:  I = 0.993583,    Time = 0.00232053   s
10D:  I = 0.989507,    Time = 0.00259113   s
```

Here, we can clearly see how badly the midpoint rule method suits integration in higher dimensions. As we increase the dimensions, not only does the error increase in the numerical result of the integration, the execution

time also increases exponentially. This can be explained by analysing the accuracy and complexity of the integration scheme. For the midpoint rule, the error scales as $\mathcal{O}(n^{-2})$ in each individual dimension, but since the total amount of integration points increases as per $N = n^d$, the error then goes as per $\mathcal{O}(N^{-2/d})$. Clearly, the error characteristics becomes worse and worse with increasing number of dimensions. Simultaneously, the complexity to execute the algorithm increases proportionally to the total number of points N, which explodes as dimensions increase. Thus, the midpoint rule is both inefficient and inaccurate for higher dimensional integration problems.

On the other hand, we notice a stark difference looking at the result of the Monte Carlo integration scheme. Here we seemingly do not lose any significant precision in the numerical integration as dimensions are increased, and execution time seems to increase in a more linear than exponential way. Both of these phenomena can be explained by the good properties for high-dimensional problems that Monte Carlo integration possesses, namely that error rate scales as $\mathcal{O}(N^{-1/2})$, independent of the number of dimensions d. Thus, we get better error characteristics with the Monte Carlo scheme than the midpoint rule, as soon as the number of dimensions $d \geq 4$. We can also note that the complexity for executing the Monte Carlo scheme is proportional to the total amount of random numbers used in the calculation, and since N vectors of random numbers are used, each with d components, we reason that the complexity scales as per $\mathcal{O}(Nd)$. This explains the linear increase in compute time for increasing d and fixed N, as showcased in the example above.

3 Probability Transformation and Metropolis Monte Carlo [4pt]

In this exercise you should transform a flat probability distribution

$$p_1(x) = \begin{cases} 1 & \text{if} \quad x \in [0,1] \\ 0 & \text{otherwise,} \end{cases} \tag{5}$$

into a new distribution on the form $p_2(y) \propto 1/y^2$ for $y \in [1, 20]$. Derive an exact analytical inversion for the given problem and compare with the numerical method of transferring the probability distribution using the Metropolis Monte Carlo formalism.

Let's derive an exact inversion that can transfer the flat probability of $p_1(x)$ on $x \in [0, 1]$ into $p_2(y)$ on $y \in [1, 20]$. First, we need to determine the appropriate constant in front of the $1/y^2$ term in $p_2(y)$, i.e. normalize the function for the given interval. Since any correctly normalized probability

distribution must follow $\int p(y)\,dy = 1$, we then have

$$\begin{aligned} &\int_{-\infty}^{\infty} p_2(y)\,dy = \int_1^{20} \frac{1}{y^2}\,dy = \left[-\frac{1}{y}\right]_1^{20} = \frac{19}{20} \neq 1 \\ &\Rightarrow \quad p_2(y) = \frac{20}{19} y^{-2} \quad \text{for} \quad y \in [1, 20]. \end{aligned} \tag{6}$$

In order to transform one probability distribution to another, we observe that the conservation of probability between two infinitesimal regions dx and dy must hold

$$p_1(x)dx = p_2(y)dy, \tag{7}$$

which we then can integrate to obtain a relation between the cumulative probability distribution functions

$$\int_{-\infty}^{x} p_1(x^\star)\,dx^\star = \int_{-\infty}^{y} p_2(y^\star)\,dy^\star. \tag{8}$$

Using our specific functions for $p_1(x)$ and $p_2(y)$ on their respective intervals, we get

$$x = \int_1^y \frac{20}{19} y^{\star -2}\,dy^\star = \frac{20}{19}\left[-\frac{1}{y^\star}\right]_1^y = \frac{20}{19}\left(1 - \frac{1}{y}\right) \tag{9}$$

$$\Rightarrow \quad y = \frac{1}{1 - \frac{19}{20}x} \quad \text{for} \quad x \in [0, 1]. \tag{10}$$

The expression in eq.(10) is our exact inversion that we previously had requested. Thus by using this function, we can map the uniform distribution $p_1(x)$ on $x \in [0, 1]$ to the non-uniform distribution $p_2(y) \propto 1/y^2$ on $y \in [1, 20]$.

Another method of achieving the same result numerically is by utilizing the Metropolis Monte Carlo formalism. For my specific implementation, consult the Jupyter Notebook `fsm_ex2.ipynb` attached with this PDF. Here, the code generates 1 million randomly generated numbers according to the distribution $p_2(y)$ on $y \in [1, 20]$. This is done following the Metropolis-Hasting's algorithm, where a Markov chain of states y are generated through a repeated process where a new state y' is proposed and either accepted or rejected depending on the relative probability of the new state compared to the old one. Thus, accepted states gets added to the Markov chain if a random number u drawn from the flat distribution on $[0, 1]$ are equal or smaller than the Hasting's ratio

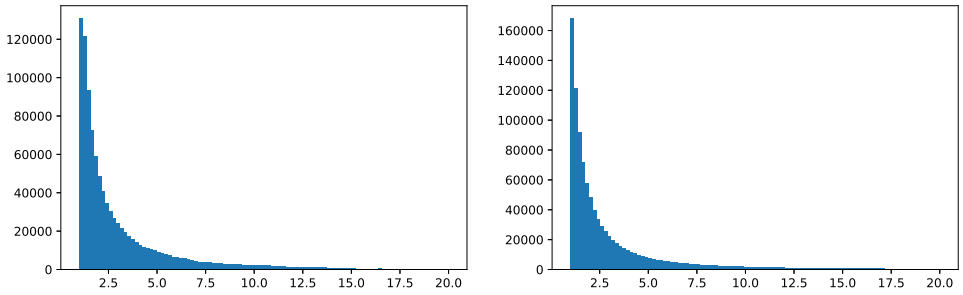
$$r = \min\left(1, \frac{p_2(y')q(y' \to y)}{p_2(y)q(y \to y')}\right), \tag{11}$$

where q are the proposal probabilities from state y to y' and vice versa. Since we are using a symmetric proposal probability function by utilizing a normal

distribution $y_{i+1} = N(y_i, \sigma)$ with $\sigma = 0.1$, we then can simplify eq.(11) to

$$r = \min\left(1, \frac{p_2(y')}{p_2(y)}\right) = \min\left(1, \frac{y^2}{y'^2}\right). \tag{12}$$

Using an initial starting value of $y_0 = 2$ and generating 1 million points, we can visualize the distribution through a histogram shown to the left in fig.(4). Here, we have also for comparison generated 1 million numbers in the standard flat distribution on the interval $[0, 1]$ and then transferred these points to the distribution $p_2(y)$ on $y \in [1, 20]$ by using the analytical expression in eq.(10) derived earlier.



(a) **(b)**

Figure 4 – Histogram with 100 boxes of 1 million randomly generated values in the range $y \in [1, 20]$ following the distribution $p_2(y) \propto 1/y^2$, obtained through two different methods. To the left, using the Metropolis-Hasting's algorithm to create a Markov chain with random values, and to the right, through an analytical inversion from the flat distribution $p_1(x)$ on $x \in [0, 1]$.

Analysing the result, we can see that both methods works for transferring probability distributions. However, one should be vary of the selection of a good proposal function with not a too small of a step-size. By playing around with the value of σ, we can note that the distribution deviates clearly from the exact and analytical $1/y^2$ shape when $\sigma \ll 0.1$. This has to do with the fact that for smaller step-sizes, all states become more and more correlated with each other, and thus you get congregations of values in certain regions.