Monte Carlo

Random Numbers

Francesco Banterle, Ph.D. - July 2021

Real Random Numbers
Introduction

- Montecarlo methods require randomness:
  - We have a **match** between our mathematical model and our computational model.
- To draw truly random number is not an easy task:
  - We need special hardware based on thermal noise, shot noise, etc.

Real Random Numbers
Introduction

- There are limitations too:
  - We cannot debug with them.
  - Such generators are computationally slow.
  - Some hardware generators fails some randomness tests —> flaws while readings.

A Pseudo-Random Generator

Introduction

- In this course, pseudo-random numbers will be called random numbers for the sake of simplicity.
- The main reasons to use such generators:
  - **Computationally complexity**: they are computationally fast in drawing numbers.
  - **Debugging**: we can restart the stream of drawn numbers.

A Pseudo-Random Generator

Introduction

- For some random generators, their inner state can be inferred from their outputs:
  - We can then predict the next draws.

- Cryptography requires random generators where **this problem is computationally expensive to solve**.

A Pseudo-Random Generator

Introduction

- Blum Blum Shub generator is defined as:

$$x_{i+1} = x_i^2 \mod M \quad M = p \cdot q,$$

where p and q are large primes (4096-bit), and x_0 is co-prime to M; different from 0 or 1.

- The square root $\mathrm{mod}\ M$ of x_i is not a computationally easy to solve problem.

- For Montecarlo, **we do not need cryptographic security**!

A Pseudo-Random Generator
Properties

- **Computationally Fast**: Montecarlo-based algorithms devour a huge number of random numbers. We need to draw such numbers computationally quickly.
- **Multiple Streams**: Montecarlo-based algorithms typically are executed in parallel (more CPUs or threads), we need different and independent streams.
- **Large Period**: the sequence of random numbers starts to repeat only after P numbers were drawn; where P is a very large number.
- **Quality**: the generated numbers are independent and identically distributed (i.i.d.) in the range [0,1] or (0,1) or (0,1], or [0,1).
    - **Equidistribution:**: when drawing numbers in $x_i \in [0,1]$, we do not want more dense regions of others:

$$\forall_{[y_s, y_e] \subset [0,1]} \, | \{ x_i \, | \, x_i \in [y_s, y_e] \} | \propto | y_e - y_s |$$

Classic Random Generators

The Middle-Square Method

- A very simple method introduced by Von Neumann. This method is considered the first PRNG.

- A 32-bit version would be:

$$x_{i+1} = \left(x_i^2 \gg 8\right) \odot \text{00FFFFFF},$$

where $x_0 \neq 0$.

- Drawbacks: very small period.

Classic Random Generators

LCGs

- A classic and well-known random generator is the **linear congruential generator** (LCG) that is defined as:

$$x_{i+1} = \left(x_i \cdot a_0 + a_1\right) \mod M,$$

where:

- $x_0 \in [0, M-1]$ is called the **seed** or start value,
- M is called the modulus (a positive integer),
- $a_0 \in [0, M-1]$ is the multiplier, and
- $a_1 \in [0, M-1]$ is the increment.

Classic Random Generators

LCGs

- As we could expect, LCGs generates values in the range $[0,M-1]$.

- To get floating-point values in the range $[0,1]$, we divided by $M-1$:

$$f_{[0,1]}(x) = \frac{x}{M-1}.$$

- Typically, we want to avoid to draw 0 and 1:

$$f_{(0,1)}(x) = \frac{x+1}{M+1}.$$

Classic Random Generators

MCGs

- If $a_1 = 0$, we have:

$$x_{i+1} = x_i \cdot a_0 \mod M$$

$$x_{i+1} = x_0 \cdot a_0^i \mod M \quad i \geq 1$$

- This random generator is typically called **multiplicative congruential generator** (MCG) or Lehmer's RNG.

- To have an extra term as in a LCG does not bring any quality improvement. So MCGs are typically used instead of LCGs.

Classic Random Generators

LCGs

- A LCGs and MCGs, more in general other RNGs, will generate a sequence of values. For example, let's draw some numbers using a generator, G_0:

$$[11, 10, 39, 44, 23, \ldots]$$

- After a while, this sequence will restart:

$$x_i = x_{i+P}$$

- For example:

$$[11, 10, 39, 44, 23, \mathbf{11, 10, 39, 44, 23}, \mathbf{11, 10, 39, 44, 23}]$$

- In this case, the sequence restart after 5 numbers are drawn. This means that G_0 has a period $P = 5$.

Classic Random Generators

LCGs Parameters Selection

- To get maximum P:
  - a_1 is relatively prime to M;
  - $a_0 = 1 \mod p$ for all p dividing M;
  - $a_0 = 1 \mod 4$ if M is a multiple of 4.
- If the period is maximized; the period is maximized for all x_0.

Classic Random Generators

MCGs Parameters Selection

- It cannot achieve maximum P:
  - If M is prime and large, it can achieve $P = M - 1$:
    - $M = 2^{31} - 1$ for 32-bit numbers
    - If M is odd, we have an alternation between odd and even numbers.
  - We need to find an a_0 such that $\forall_{x \in [0, M-1]} \exists_i \,|\, a_0^i = x \mod M$

Classic Random Generators

Parameters Selection

- Several publications (papers, technical reports, blogs, etc.) reports how to choose parameters for LCGs and MCGs including:
    - Number of bits to be used;
    - Period length;
    - Quality of the drawn numbers; e.g., statistical test results.

Classic Random Generators

MRGs

- A further generalization of MCGs are Multiple Recursive Generators or MRGs that are defined as:

$$x_i = a_0 \cdot x_{i-1} + \ldots + a_k \cdot x_{i-k} \mod M,$$

where $k \geq 1$ and $a_j \neq 0$.

- A special case of MRGs are the Lagged Fibonacci Generators or LFGs defined as:

$$x_i = x_{i-r} + x_{i-s} \mod M,$$

where r and s need to be chosen carefully.

Combining RNGs

Main Idea

- A typical trick is to combine different RNGs (which can be not too good) to improve the overall performance and to increase its period.

- Given n RNGs, $\mathbf{U}_1, \ldots, \mathbf{U}_n$, we can put their results together as:

$$x_i = \left(x_{i,\mathbf{U}_1} + x_{i,\mathbf{U}_2} + \ldots + x_{i,\mathbf{U}_n} \right) \mod 1$$

$$\forall_{z \in \mathbb{R}} \quad z \mod 1 \longrightarrow z - \lfloor z \rfloor$$

Combining RNGs
The Wichmann-Hill Generator

- A classic example is the Wichman-Hill Generator:

$$x_i = 171 \cdot x_{i-1} \mod 30269$$
$$y_i = 172 \cdot y_{i-1} \mod 30307$$
$$z_i = 170 \cdot z_{i-1} \mod 30323$$

$$w_i = \left(\frac{x_i}{30269} + \frac{y_i}{30307} + \frac{z_i}{30323} \right) \mod 1$$

- This way we can achieve $P = 6.95 \cdot 10^{12}$.

Combining RNGs

MRG32k3a

- L'Ecuyer proposed to combine two MRGs obtaining MRG32k3a.
- The method has combines two MRGs:

$$x_i = (1403580x_{i-2} - 810728x_{i-3}) \mod (2^{32} - 209)$$

$$y_i = (527612y_{i-2} - 1370589y_{i-3}) \mod (2^{32} - 22853)$$

$$U_i = \begin{cases} \dfrac{x_i - y_i + 2^{32} - 209}{2^{32} - 208} & \text{if } x_i \leq y_i \\ \dfrac{x_i - y_i}{2^{32} - 208} & \text{otherwise} \end{cases}.$$

- By combining two MRGs, we can achieve $P = 3 \times 10^{57}$.
- MATLAB has employed MRG32k3a.

Quality Tests for RNGs

χ^2 Test
Main Idea

- If N samples are drawn in the interval $[0,1]$, then the number of drawn samples in each interval has to be equal on average.
- This test the range of data in k subintervals; i.e., discrete distribution.
- We, then, count the sample for each subinterval.
- The number of samples that fall in each subinterval is close to the expected number.

χ^2 Test
Main Idea

- The test is defined as

$$D = \sum_{i=1}^{k} \frac{(o_i - e_i)^2}{e_i} < \chi^2_{[1-\alpha,k-1]}$$

where:

- α is the level of significance;
- k is the number of bin in the histogram;
- o_i is the number of observed values in the i-th bin of the histogram;
- $e_i = \dfrac{N}{k}$ is the number of expected values in the i-th bin of the histogram.

χ^2 Test
Example using RANDU

- We draw 1,000 numbers in $[0,1]$.
- We create a histogram with $k = 10$ bins.
- We compute $D = 14.2$
- $\chi^2[0.9,9] = 14.684$
- 14.2 < 14.684:
  - We accept these values!

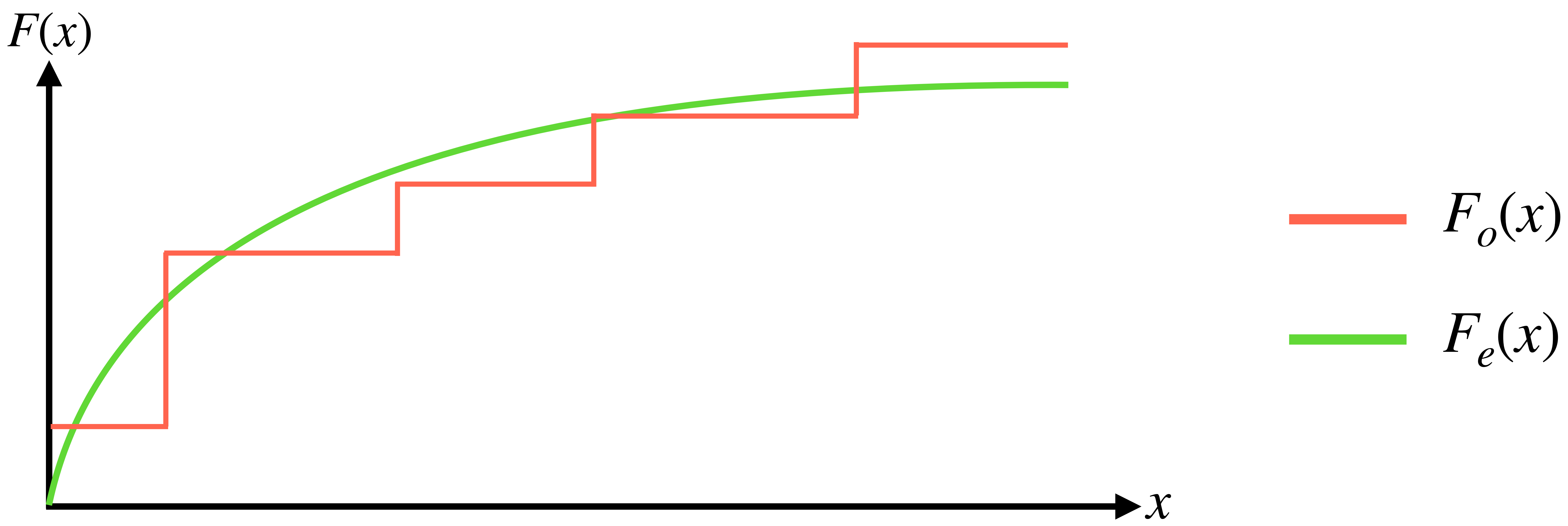
χ^2 Test
Example using RANDU

- We draw 1,000 numbers in $[0,1]$.
- We create a histogram with $k = 10$ bins.
- We compute $D = 14.2$
- $\chi^2[0.9,9] = 14.684$
- 14.2 < 14.684:
    - We accept these values!

Observed	Expected
104	100
89	100
79	100
103	100
108	100
94	100
102	100
126	100
102	100
93	100

Kolmogorov–Smirnov Test
Main Idea

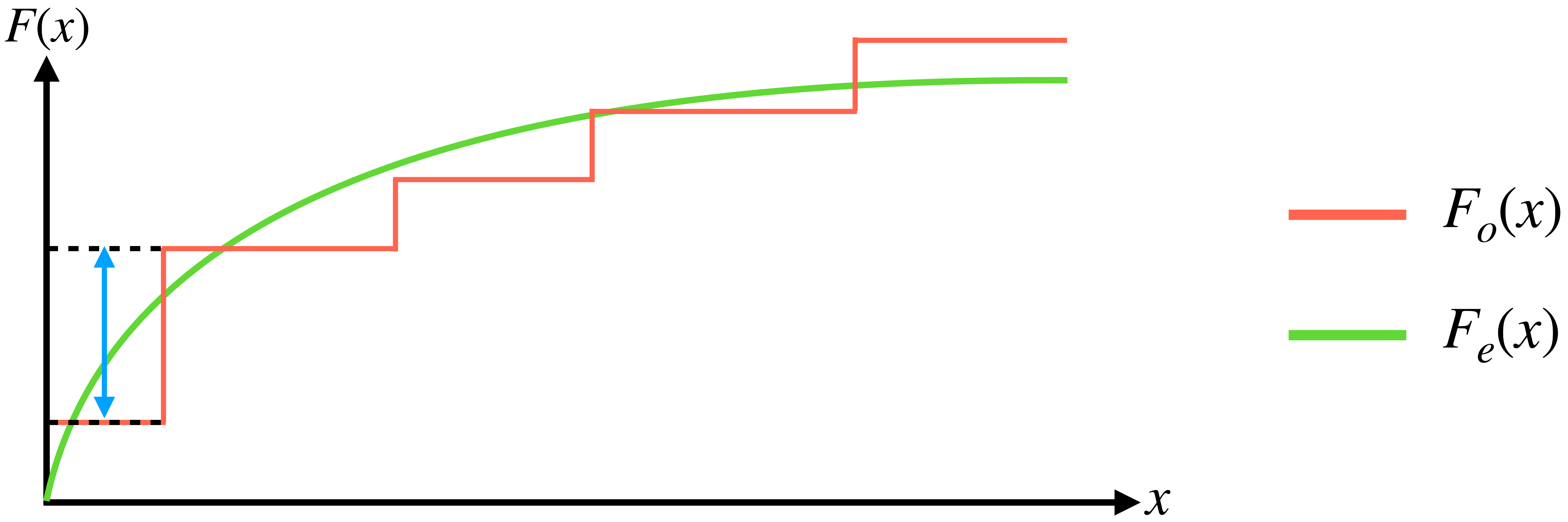
- We measure the differences between the observed cumulative distribution function (CDF) or $F_o(x)$ and the expected CDF or $F_e(x)$.
- This difference has to be small.



Kolmogorov–Smirnov Test

Main Idea

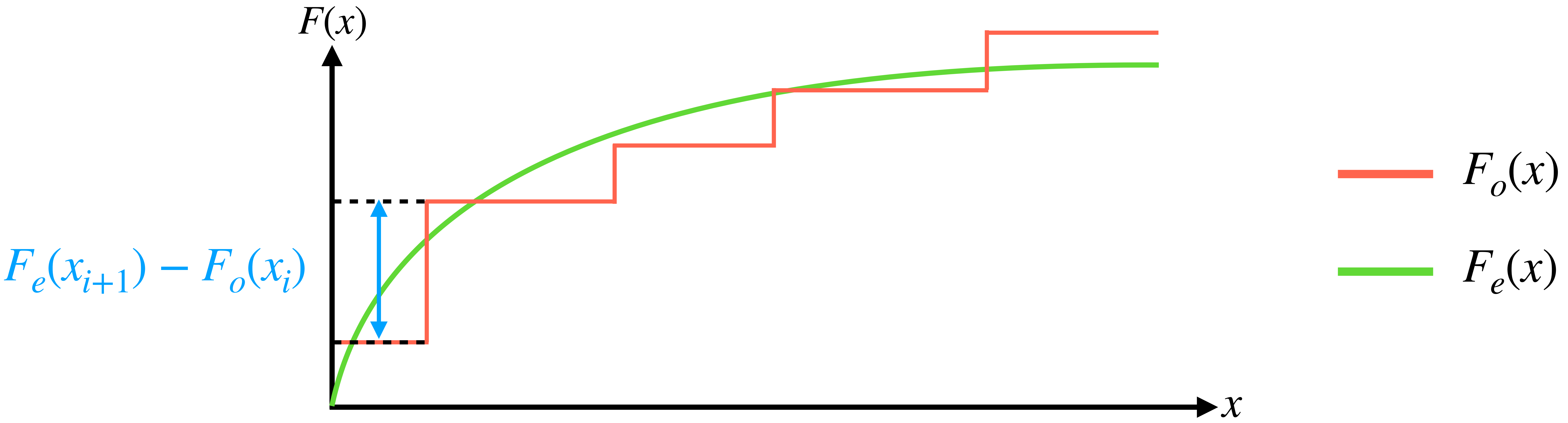
- We measure the differences between the observed cumulative distribution function (CDF) or $F_o(x)$ and the expected CDF or $F_e(x)$.
- This difference has to be small.



Kolmogorov–Smirnov Test

Main Idea

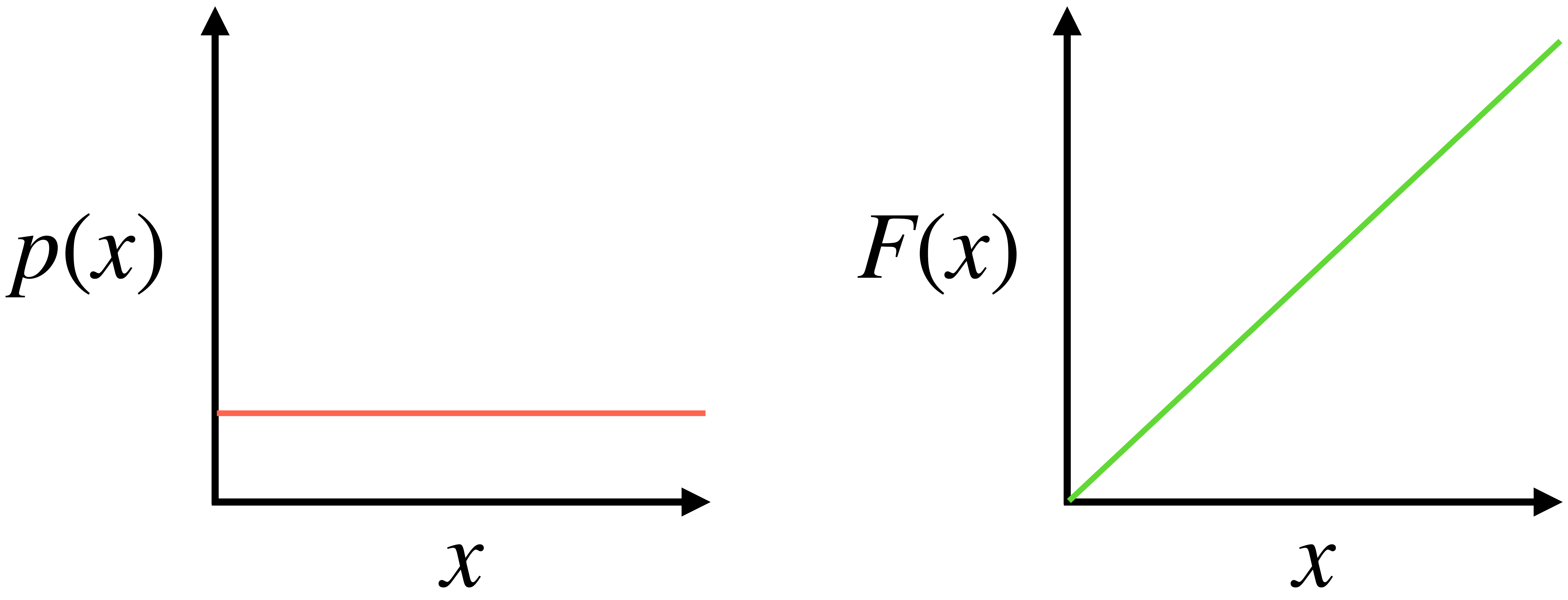
- We measure the differences between the observed cumulative distribution function (CDF) or $F_o(x)$ and the expected CDF or $F_e(x)$.
- This difference has to be small.



Kolmogorov–Smirnov Test

Main Idea

- If N samples are drawn in the interval $[0,1]$, then the graph of the empirical distribution of samples follows the CDF of uniform distribution in $[0,1]$.



Kolmogorov–Smirnov Test

Main Idea

- As first step, we draw some numbers $(n = 30)$ from our RNG that we want to test:
  - X = [0.33967685, 0.05724571, 0.66265701, 0.51043379, 0.14676791, 0.56020847, 0.03633356, 0.70865904, 0.39256236, 0.6442009 , 0.2163937 , 0.56919288, 0.28660165, 0.04716307, 0.41800649, 0.61657189, 0.84608168, 0.41675127, 0.67504593, 0.08985331, 0.06058904, 0.69510391, 0.45404319, 0.31664501, 0.67808957, 0.48707878, 0.27557392, 0.45049086, 0.97062946, 0.30428724]

Kolmogorov–Smirnov Test

Main Idea

- Then, we sort X:
  - X=[0.03633356, 0.04716307, 0.05724571, 0.06058904, 0.08985331, 0.14676791, 0.2163937 , 0.27557392, 0.28660165, 0.30428724, 0.31664501, 0.33967685, 0.39256236, 0.41675127, 0.41800649, 0.45049086, 0.45404319, 0.48707878, 0.51043379, 0.56020847, 0.56919288, 0.61657189, 0.6442009 , 0.66265701, 0.67504593, 0.67808957, 0.69510391, 0.70865904, 0.84608168, 0.97062946]

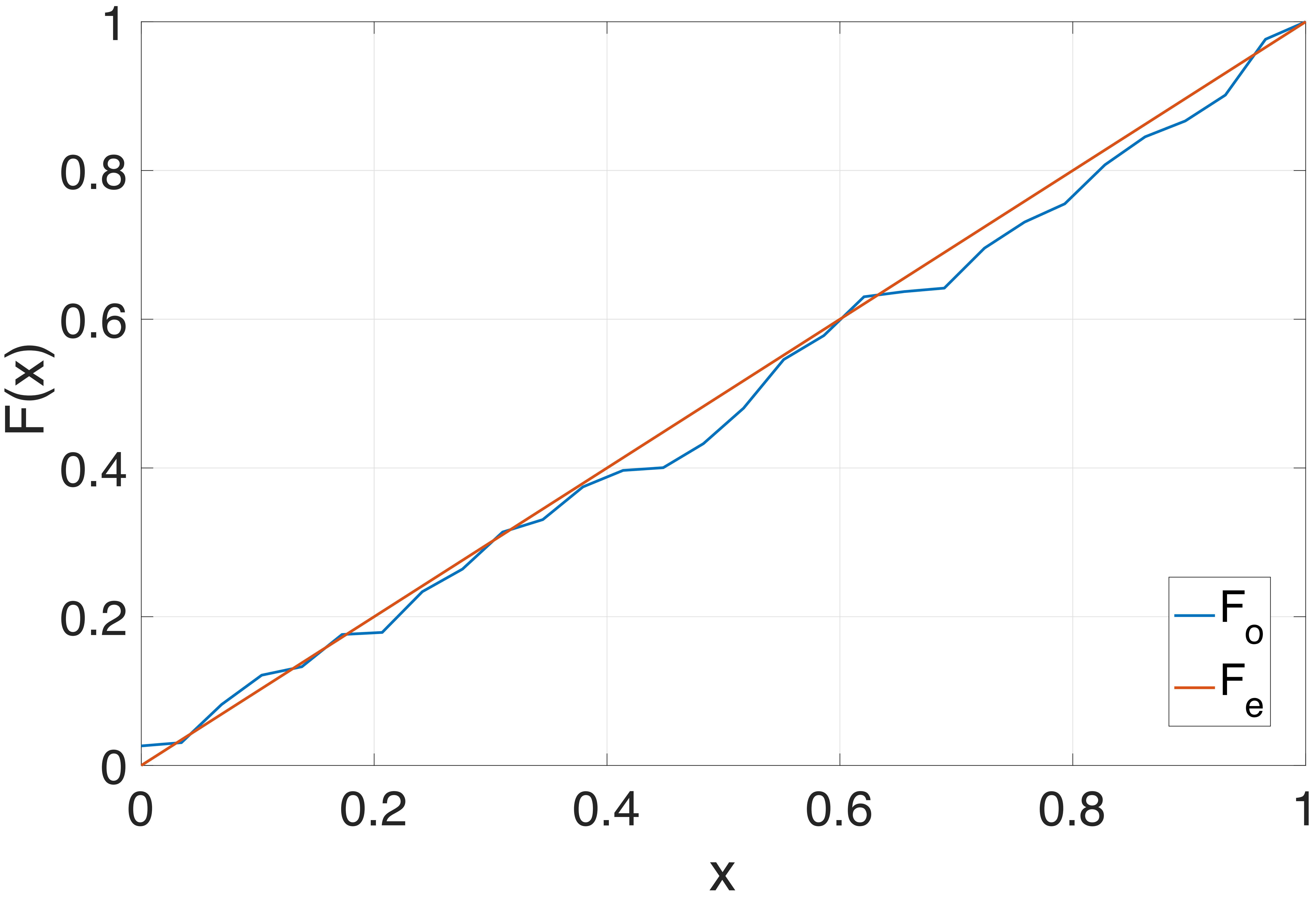
Kolmogorov–Smirnov Test

Main Idea

- At this point, we create our CDF:
  - F_e = [0.02626448, 0.03069083, 0.08192876, 0.12139649, 0.13274487, 0.17606128, 0.17887067, 0.23366556, 0.26401925, 0.31383012, 0.3305621 , 0.3745732 , 0.3967338 , 0.40038054, 0.43270162, 0.48037616, 0.54579684, 0.57802086, 0.63021673, 0.63716436, 0.64184922, 0.69559601, 0.73070352, 0.75518713, 0.80761833, 0.84528021, 0.86658813, 0.90142096, 0.97647192, 1.0]

Kolmogorov–Smirnov Test

Main Idea



Kolmogorov-Smirnov Goodness-of-Fit Test

Main Idea

- We compute D as:

$$D = \max\left(D^+, D^-\right),$$

where D^+ and D^+ are defined as:

$$D^+ = \arg\max_x \left(F_o(x) - F_e(x)\right) \qquad D^- = \arg\max_x \left(F_e(x) - F_o(x)\right)$$

Kolmogorov-Smirnov Goodness-of-Fit Test

Main Idea

- How do we compute D^+ exactly in our case?

$$D^+ = \arg \max_{i \in [1,n]} \left(\frac{i}{n} - x_i \right)$$

- How do we compute D^- exactly in our case?

$$D^- = \arg \max_{i \in [0,n-1]} \left(x_i - \frac{i}{n} \right)$$

Kolmogorov-Smirnov Goodness-of-Fit Test

Main Idea

- Finally, to pass the test (I.e., we accept the Null Hypothesis that X numbers are uniformly distributed in [0,1]) if:

$$D < D_{\alpha,n},$$

where α is the significance value.

- $D_{\alpha,n}$ can be found in tables; but it can be approximated when $n > 35$:

$$D_{\alpha=0.1,n} = \frac{1.22}{\sqrt{n}}; D_{\alpha=0.05,n} = \frac{1.36}{\sqrt{n}}; D_{\alpha=0.01,n} = \frac{1.63}{\sqrt{n}}$$

Kolmogorov-Smirnov Goodness-of-Fit Test

Back to the Example

- In our example, we have:

$$D^+ = 0.0672984 \qquad D^- = 0.0431386$$

$$D = \max\left(D^+, D^-\right) = 0.0672984$$

$$D < D_{0.1,30} \rightarrow 0.0672984 < 0.21756$$

- We have passed the test —> the data is uniformly distributed over the range $[0,1]$.

RANDU

RANDU

A MCG Generator

- RANDU is a MCG RNG[1] defined as:

$$X_{i+1} = X_i \cdot 65539 \mod 2^{31},$$

where X_0 is an odd number.

- This generator is meant to generate uniformly distributed number in the range $[1, 2^{31} - 1]$.
- The generator was designed to generate high-quality tu tuples such as:

$$(x_i, x_{i+L}),$$

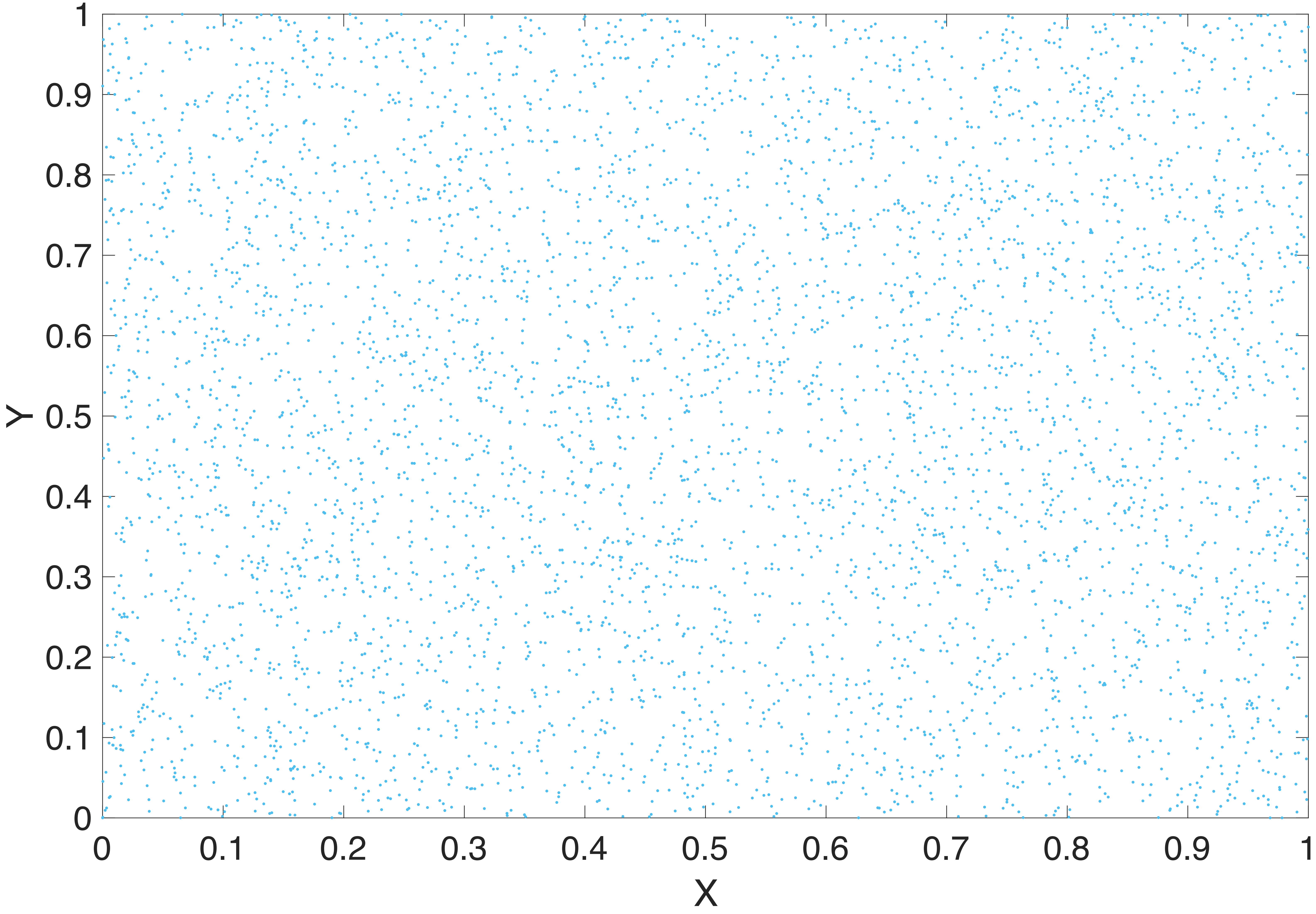
- for $L \in \{1,2,3\}$.

[1]Lewis, Peter A. W., A. S. Goodman and J. M. Miller (1969). “A Pseudo-Random Number Generator for the System 350”, *IBM Systems Journal*, 8(2), 136–45.

Let’s draw some 2D points in $[0,1]^2$ with RANDU

RANDU

2D Points

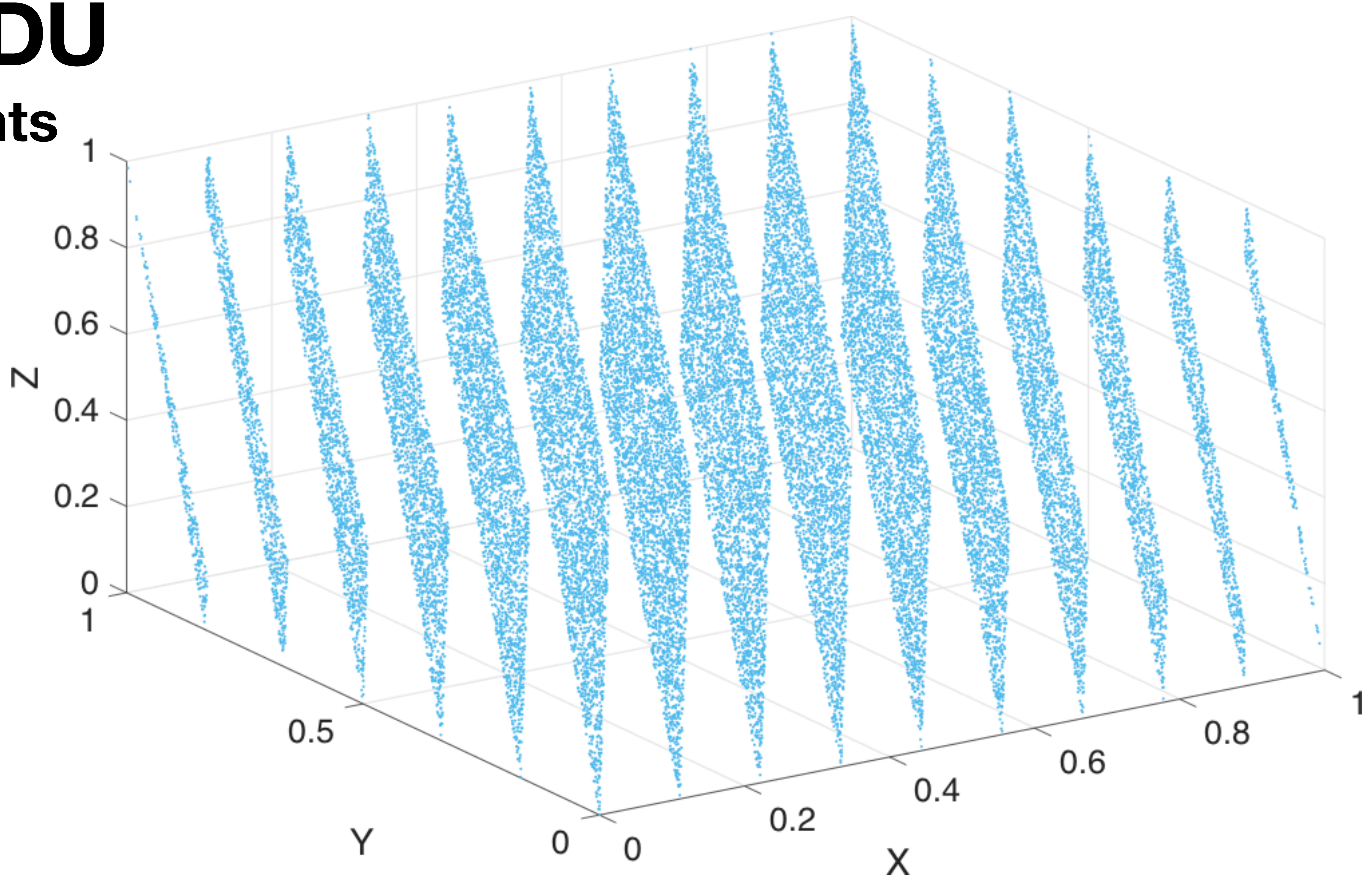


It looks nice! Doesn’t it?

Let's draw some 3D points in $[0,1]^3$ with RANDU

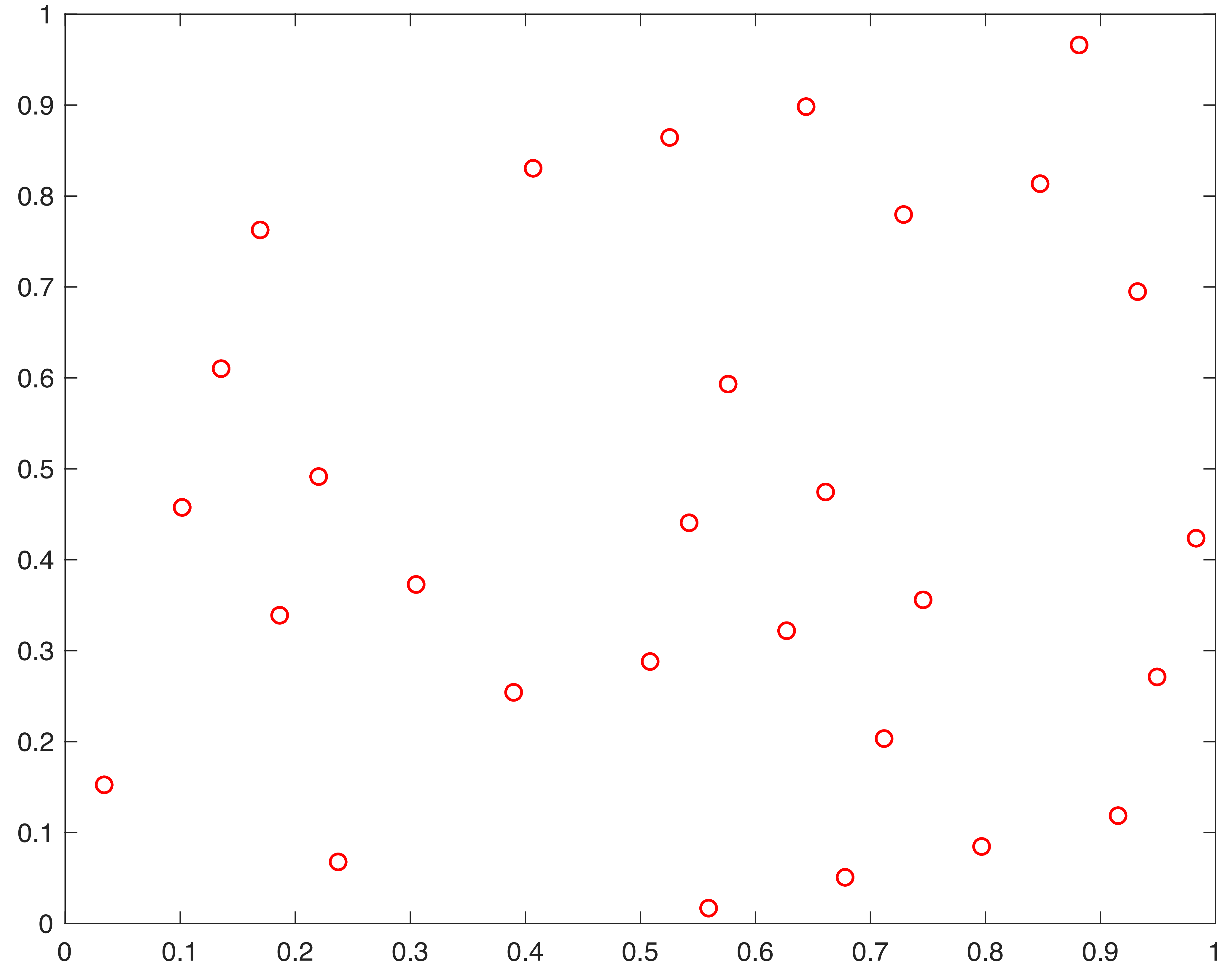
RANDU

3D Points

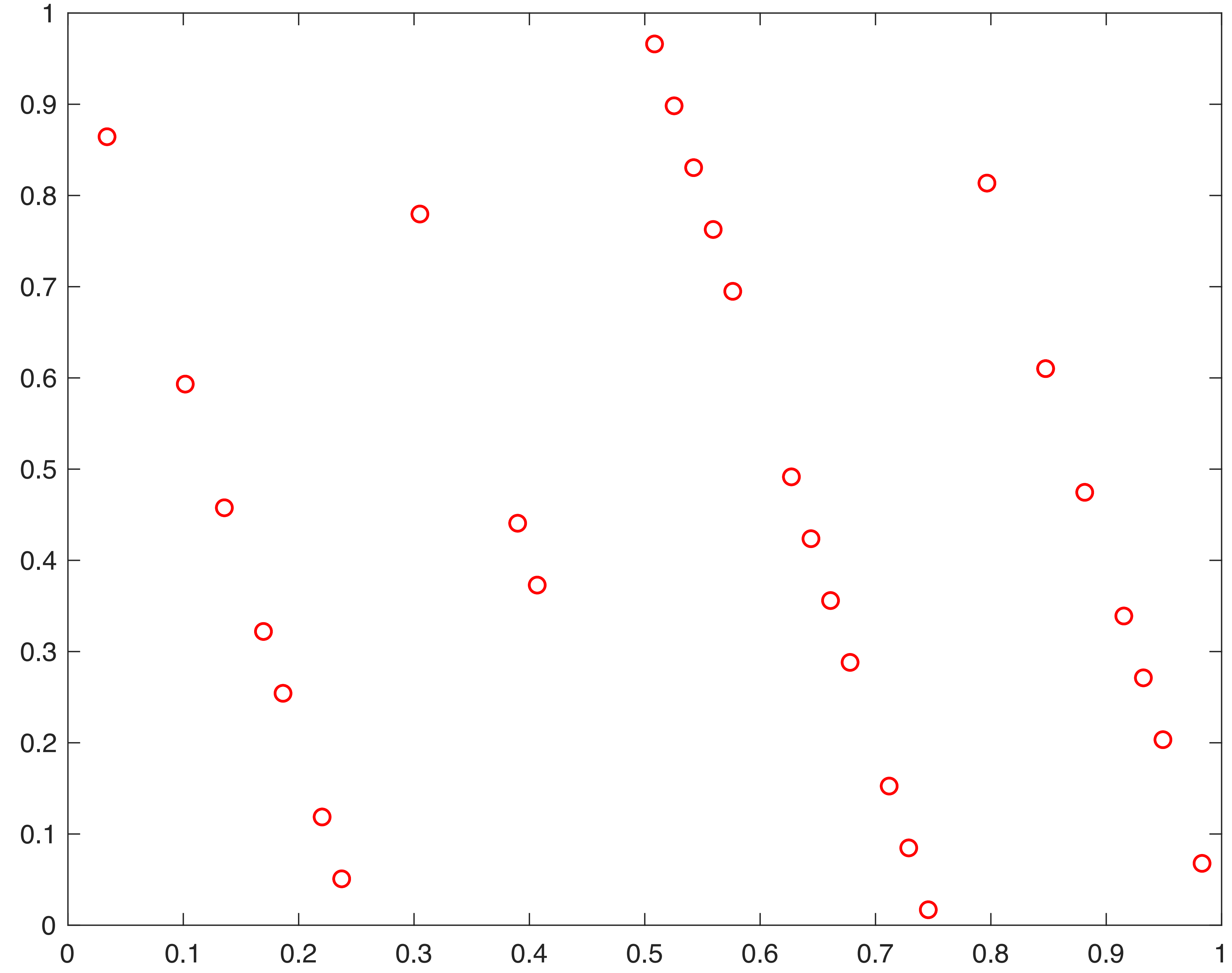


The Lattice Structure in 2D

MCGs in 2D



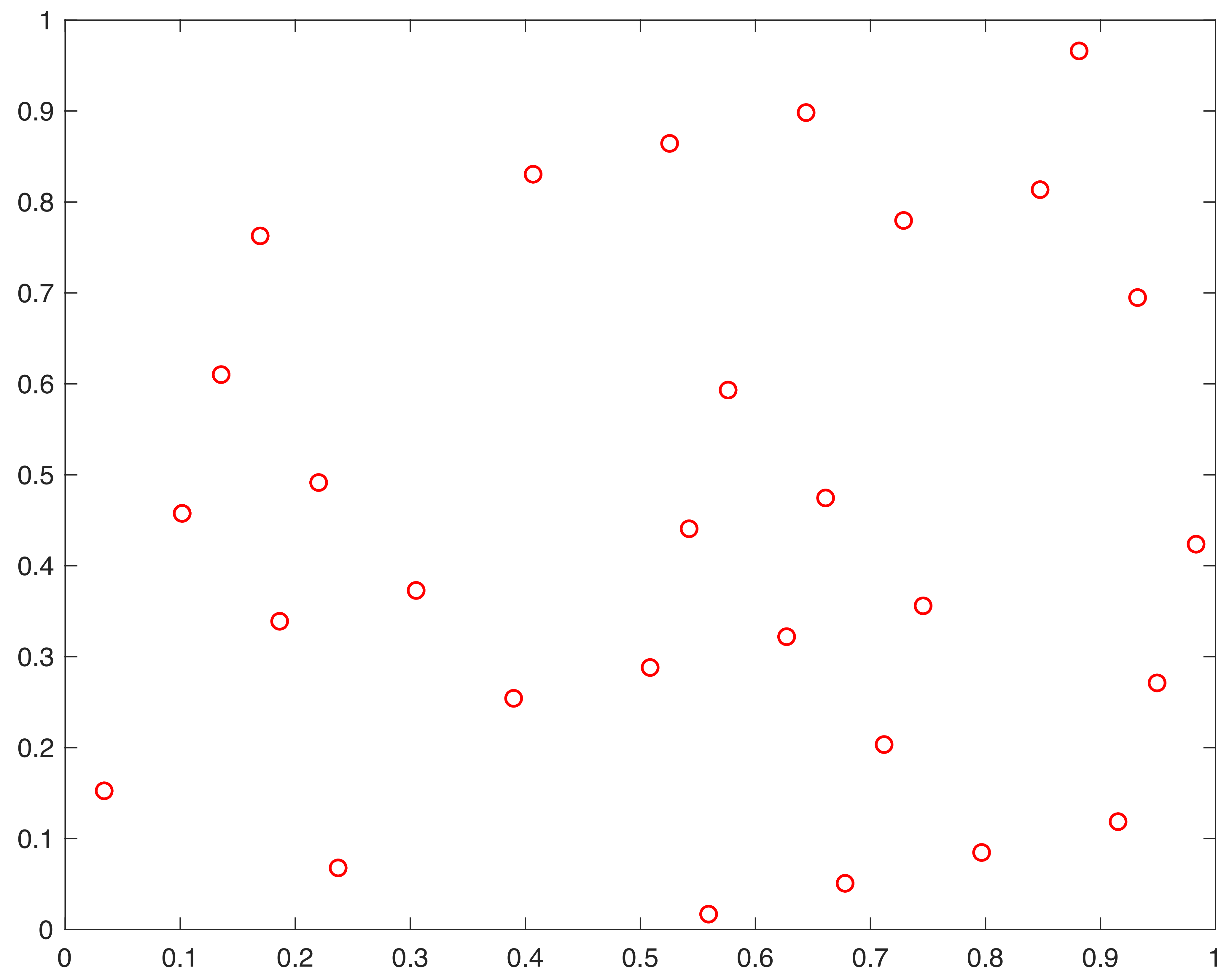
$M = 59; a_0 = 33$



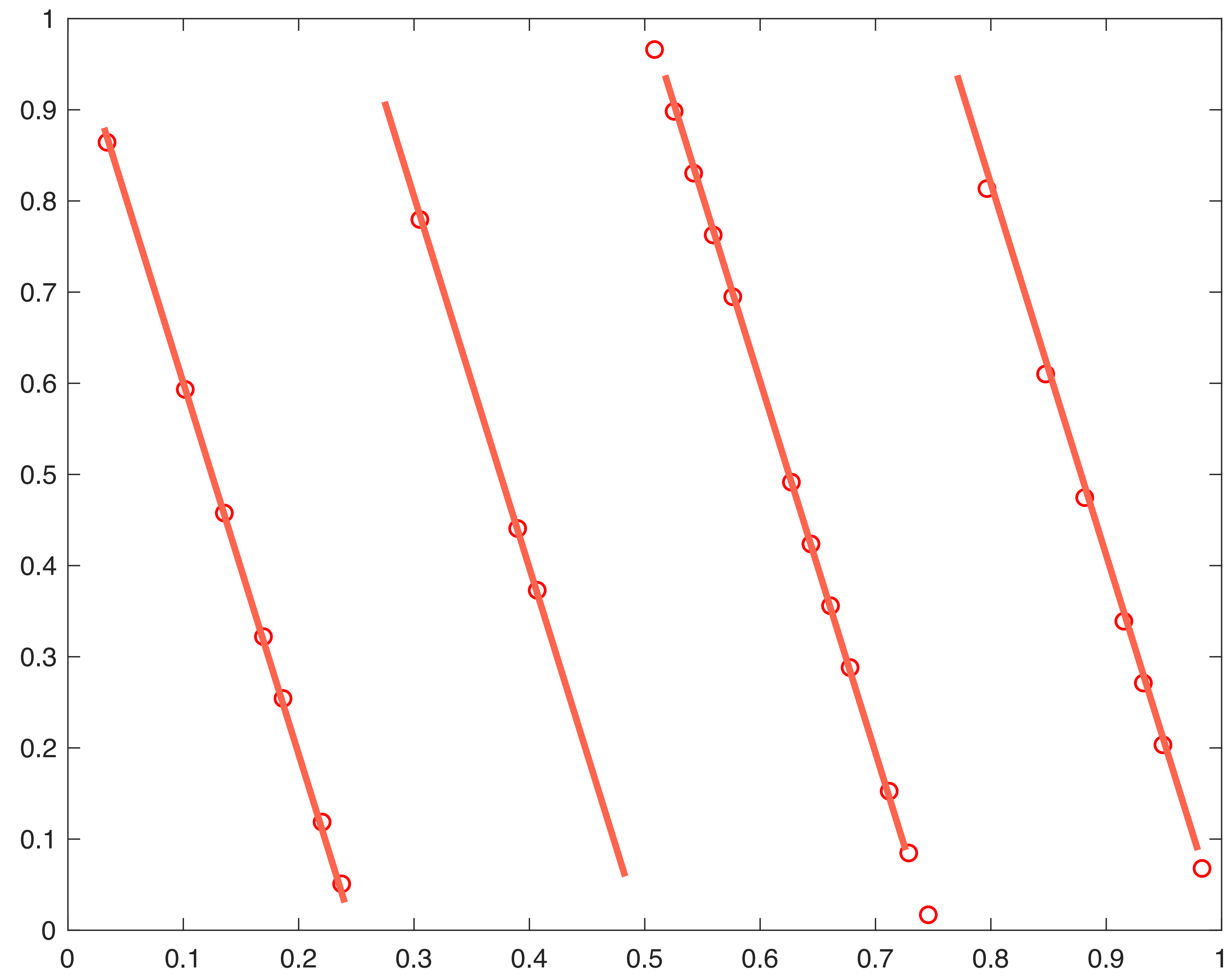
$M = 59; a_0 = 44$

The Lattice Structure in 2D

MCGs in 2D



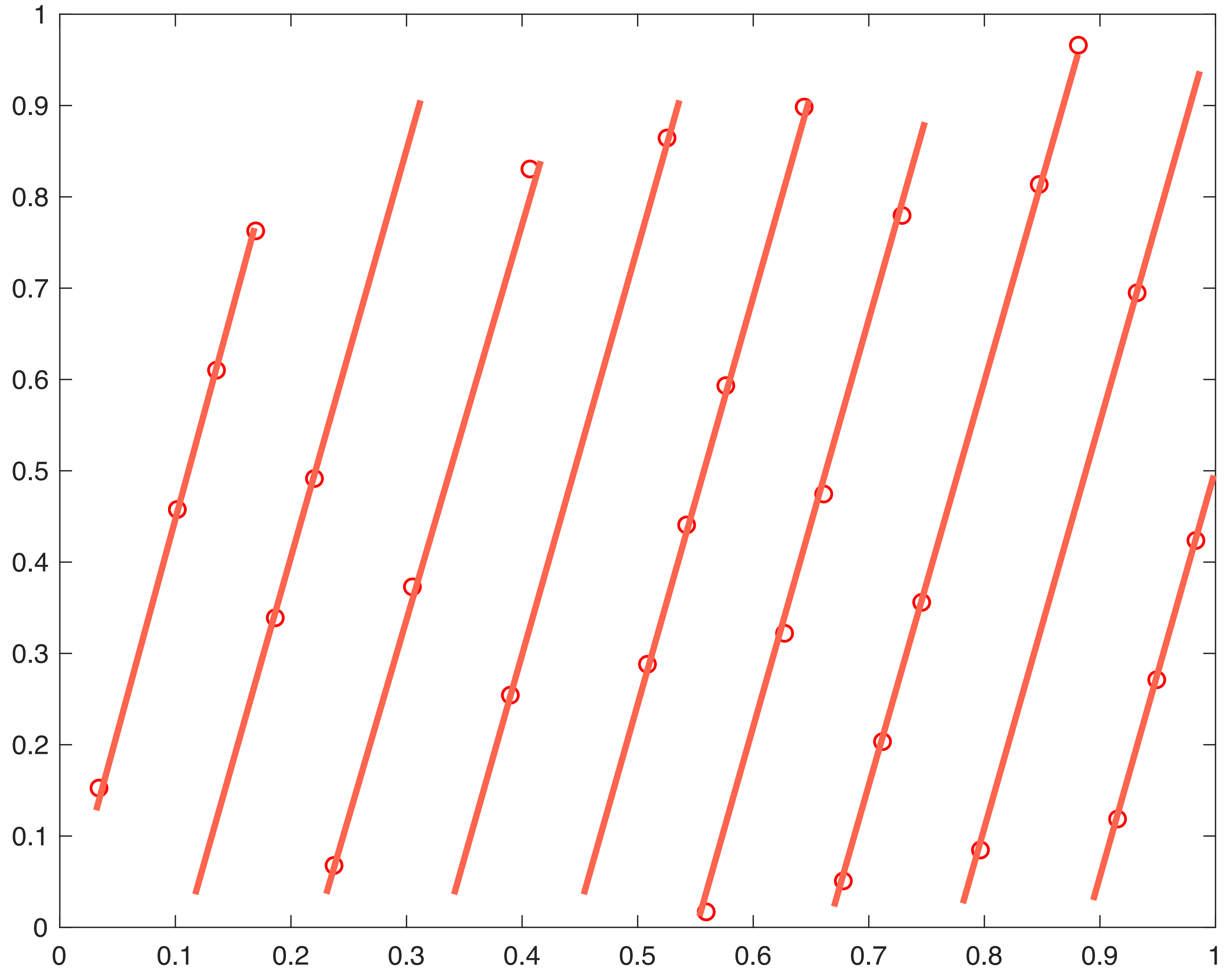
$M = 59; a_0 = 33$



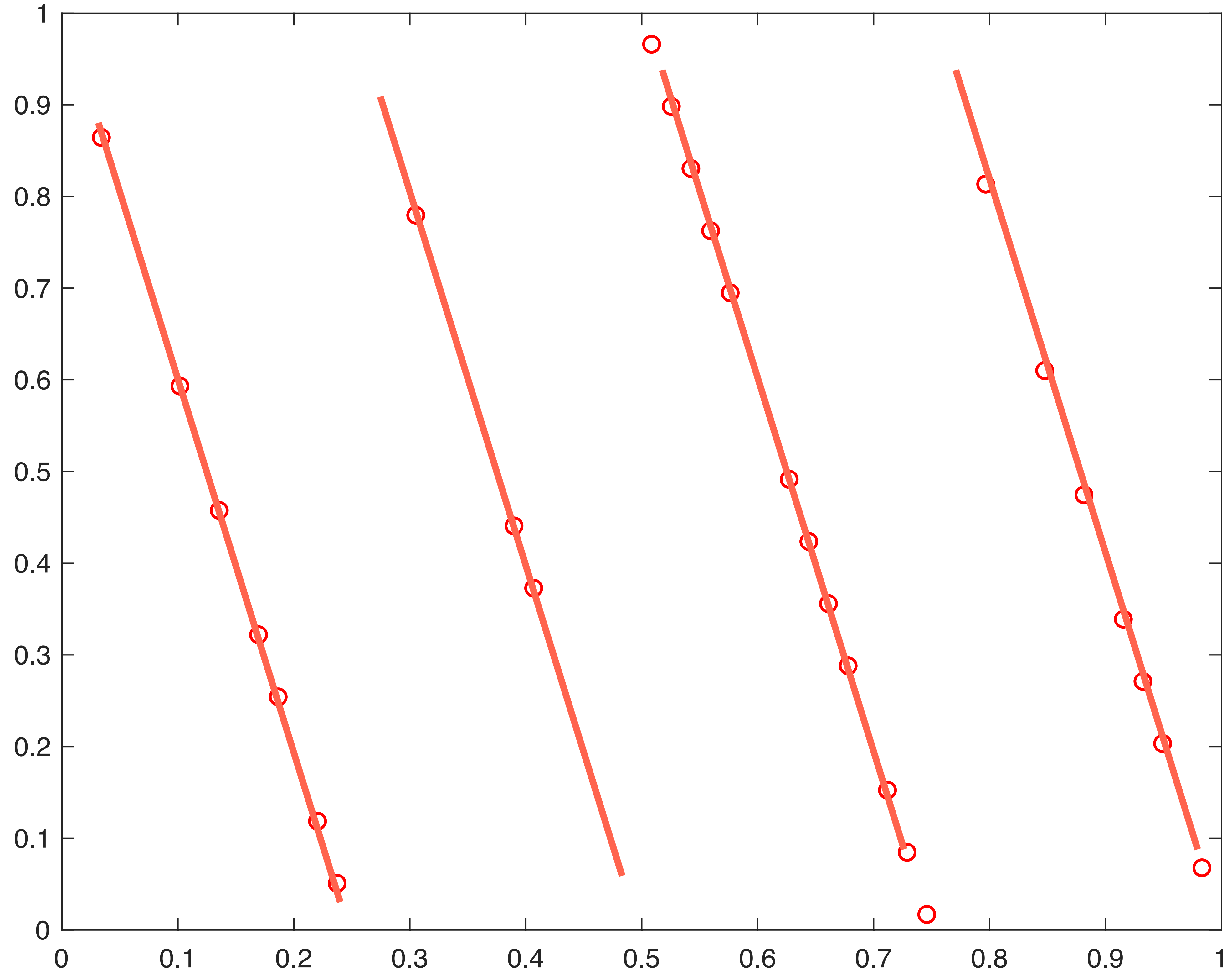
$M = 59; a_0 = 44$

The Lattice Structure in 2D

MCGs in 2D



$M = 59; a_0 = 33$



$M = 59; a_0 = 44$

Marsaglia Theorem

Lattice Structure

- Marsaglia showed that consecutive tuples; e.g.:

$$(x_i, \ldots, x_{i+k-1}),$$

from an MCG have a lattice structure:

$$\mathscr{L} = \left\{ \sum_{j=1}^{k} \alpha_j \cdot \mathbf{v}_j \,|\, \alpha_j \in \mathbb{Z} \right\},$$

where $\mathbf{v}_j$ are linearly independent basis vector in $\mathbb{R}^k$.

- The tuples are the intersection of the infinite $\mathscr{L}$ set with the unit cube $[0,1)^k$.

Marsaglia Theorem

Uniformity in 1D

- In RANDU, all consecutive triples, (x_i, x_{i+1}, x_{i+2}), are all contained within 15 parallel planes in the unit cube.
- What do we want?
  - All the k-tuples, $(x_i, \ldots, x_{i+k-1})$, should be uniform:
    - **At least when k is small**.

Marsaglia Theorem

Uniformity in 1D

- How do we assess uniformity?
    - In 1D, we split $[0,1)$ into:
        - 2^l congruent subintervals:

$$\forall a \in [0,2^l) \quad \left[\frac{a}{2^l}, \frac{a+1}{2^l}\right).$$

        - The subinterval containing x_i can be found from its first l-bits.

Marsaglia Theorem

Uniformity in k-Dimension

- Similarly to the 1D case, we can split $[0,1)^k$ into 2^{kl} sub-cubes.
- An RNG $P = 2^K$ is k-distributed to l-bits accuracy if each box:

$$B_a \equiv \prod_{j=1}^{k} \left[\frac{a_j}{2^l}, \frac{a_j + 1}{2^l}\right),$$

for $a_j \in [0,2^l)$ has 2^{K-kl} of the points (x_i, x_{i+k-1}) for $i \in [1,P]$.

NOTE: many RNGs do not have the point $\mathbf{0}$; so they have $2^{K-kl} - 1$.

More Tests

Further Readings

- L’Ecuyer and Simard’s Test01:
  - “TestU01: A C library for empirical testing of random number generators”
  - http://simul.iro.umontreal.ca/testu01/tu01.html
- Marsaglia’s Die Hard Tests extended by Brown:
  - https://webhome.phy.duke.edu/~rgb/General/dieharder.php

Modern RNGs

Mersenne Twister

Main Idea

- Makoto Matsumoto and Takuij Nishimura introduced this RNG in 1997.
- This RNG takes its name because its period is a Mersenne Primer; i.e., $P = 2^n - 1$ is a prime.
- The most famous Mersenne Twister version is the MT19937 (e.g., C++11):
  - $P = 2^{19937} - 1$.
- MT generates a sequence of word vectors with w-dimension, which are considered to be uniform pseudo-random integer in the range $[0, 2^w - 1]$.

Mersenne Twister

Main Idea

- The method is defined by:

$$\mathbf{x}_{i+n} = \mathbf{x}_{i+m} \oplus \left((\mathbf{x}_i^u \mid \mathbf{x}_{i+1}^l) \cdot A \right) \quad i = 0,1,\ldots$$

  where vectors are w-dimensional vectors, $\mathbf{x} = (x_{w-1}, \ldots, x_0)$, over $\mathbb{F}_2 = \{0,1]\}$:

  - A finite field of two elements (0 and 1):

    - Two operations:

      - $+$: neutral element 0, commutative and associative

      - $\cdot$: neutral element 1, commutative, associative, and distributive

Mersenne Twister

Main Idea

$$\mathbf{x}_{i+n} = \mathbf{x}_{i+m} \oplus \left((\mathbf{x}_i^u \,|\, \mathbf{x}_{i+1}^l) \cdot A \right) \quad i = 0, 1, \ldots,$$

- n is the degree recurrence.
- $m \in [1, n]$.
- $n > m$.
- The first n elements, $\mathbf{x}_0, \ldots, \mathbf{x}_{n-1}$, are seeds and they are cyclically updated.

Mersenne Twister

Main Idea

$$\mathbf{x}_{i+n} = \mathbf{x}_{i+m} \oplus \left(\boxed{(\mathbf{x}_i^u \mid \mathbf{x}_{i+1}^l)} \cdot A \right) \qquad i = 0,1,\ldots,$$

- $\mathbf{x}_i^u$ the upper $w - r$ bits of $\mathbf{x}_i$, where $r \in [0, w-1]$;
- $\mathbf{x}_i^l$ the lower r bits of $\mathbf{x}_i$.
- Parameters need to be picked such that $2^{nw-r} - 1$ is a Mersenne prime.

Mersenne Twister

Main Idea

$$\mathbf{x}_{i+n} = \mathbf{x}_{i+m} \oplus \left((\mathbf{x}_i^u \mid \mathbf{x}_{i+1}^l) \boxed{A} \right) \qquad i = 0,1,\ldots,$$

- A is a $w \times w$ matrix with values in $\mathbb{F}_2$:

$$A = \begin{bmatrix} 0 & 1 & 0 & \ldots & 0 \\ 0 & 0 & 1 & \ldots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \ldots & 1 \\ a_{w-1} & a_{w-2} & a_{w-3} & \ldots & a_0 \end{bmatrix}$$

Mersenne Twister

Main Idea

- The RNG draws:
  - $\mathbf{x}_n$ for $i = 0$;
  - $\mathbf{x}_{n+1}$ for $i = 1$;
  - $\mathbf{x}_{n+2}$ for $i = 2$;
  - etc.
- Note that given, $\mathbf{x} = (x_{w-1}, \ldots, x_0)$ and $\mathbf{a} = (a_{w_1}, \ldots, a_0)$, $\mathbf{x} \cdot A$ can be computed as:

$$\mathbf{x} \cdot A = \begin{cases} (\mathbf{x} \gg 1) \oplus \mathbf{a} & \text{if } x_0 = 1 \\ (\mathbf{x} \gg 1) & \text{otherwise} \end{cases}$$

Mersenne Twister
Tampering

- To improve the distribution properties, the output value is multiplied by $w \times w$ a matrix T:

$$x \cdot T$$

- As before, we implement this transformation with shifts and xors:

$$\begin{aligned}
y &= x \oplus ((\mathbf{x} \gg u) \cdot \mathbf{d}) \\
y &= x \oplus ((\mathbf{x} \ll s) \cdot \mathbf{b}) \\
y &= x \oplus ((\mathbf{x} \ll t) \cdot \mathbf{c}) \\
z &= x \oplus (\mathbf{x} \gg l)
\end{aligned}$$

Mersenne Twister

Seeds

- We need to fill n seeds that are w-vectors.
- n is typically large; e.g., 312.
- Strategy:
  - We supply $\mathbf{x}_0$ as a seed number.
  - The remaining seeds are computed as:

$$\mathbf{x}_k = f \cdot (\mathbf{x_{k-1}} \oplus \left(\mathbf{x_{k-1}} \gg (w-2)\right) + k \text{ for } k = 1,\ldots,n-1$$

Mersenne Twister

Conclusions

- Advantages:
  - MT has a very long period, $P = 2^{19937} - 1$. Some RNGs have very short periods (e.g., $P = 2^{32}$) which can lead to issues during simulations;
  - Computationally fast implementations, and it can exploit SIMD architectures;
  - We can generate points with 623 dimension with equi-distribution to 32-bit accuracy;

Mersenne Twister

Conclusions

- Disadvantages:
  - Initialization needs to be done with care:
    - If there are too many 0s; the sequence may contain many 0s for many generations;
    - If the seeds are picked systematically (e.g., $(0,20,30,\ldots)$) the output may be correlated;
    - Large state; I.e., 2.5KiB ($w = 64; n = 312; m = 156; r = 31$)

Modern RNGs

XORShift Family

- L'Ecuyer proposed a simple RNGs based on XOR and shift operators:

$$
\begin{aligned}
x_t &= x \oplus (x_{i-4} \ll 15)) \\
x_i &= (x_{i-1} \oplus (x_{i-1} \gg 21)) \oplus (x_t \oplus (x_t \gg 4))
\end{aligned}
$$

- The seeds, x_0, x_1, x_2, x_3, can be set to random numbers; not all 0.
- Note: we need to store the latest generated values.
- When 32/64-bit numbers are used we have a $P = 2^{32} - 1/P = 2^{64} - 1$

Modern RNGs
XORShift

- For 32/64-bit, we can achieve a larger period by adding to x_i an additive counter modulo $2^{32} - 1/2^{64} - 1$:
  - 32-bit —> $P = 2^{192} - 2^{32}$
  - 64-bit —> $P = 2^{192} - 2^{64}$
- Furthermore, the method is computationally fast and efficient in terms of memory:
  - XOR/SHIFT operations;
  - Four value state.

Modern RNGs

XORShift

- L'Ecuyer proposed also a simple and fast version 64-bit version:

$$\begin{aligned} x_0 &= 88172645463325252LL \\ x_t &= x_i \ll 13 \\ x_t &= x_t \oplus (x_t \gg 7) \\ x_{i+1} &= x_t \oplus (x_t \ll 17) \end{aligned}$$

- This has $P = 2^{64} - 1$.

Modern RNGs

Other Methods

- Permuted congruential generator (PCG) family:

```
uint32_t pcg32_random_r(uint64_t &state;  uint64_t &inc) {
    uint64_t oldstate = state;
    state = oldstate * 6364136223846793005ULL + (inc|1);
    uint32_t xorshifted = ((oldstate >> 18u) ^ oldstate) >> 27u;
    uint32_t rot = oldstate >> 59u;
    return (xorshifted >> rot) | (xorshifted << ((-rot) & 31));
}
```

  - https://www.pcg-random.org/index.html
- Xoroshiro128+ and more:
  - https://prng.di.unimi.it/

Parallel Random Generators

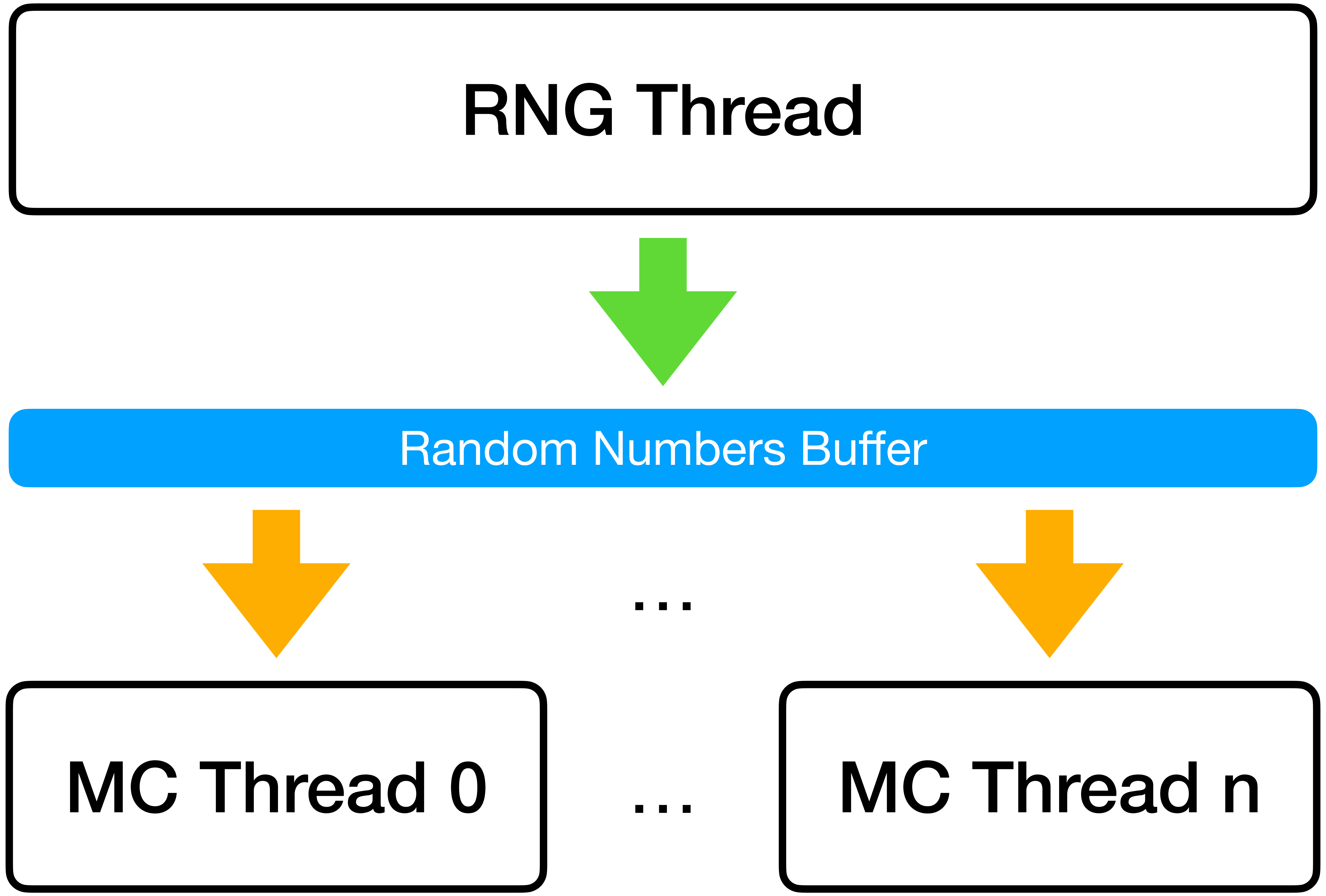
Parallel Random Generators

The Centralized Approach

- We have a single RNG, R_0:
  - Thread safe: locks, atomic operations, etc.
- R_0 draws random numbers for all other threads of the simulations.
- We may precompute a large set of numbers:
  - Single buffer.
  - A buffer for each thread.

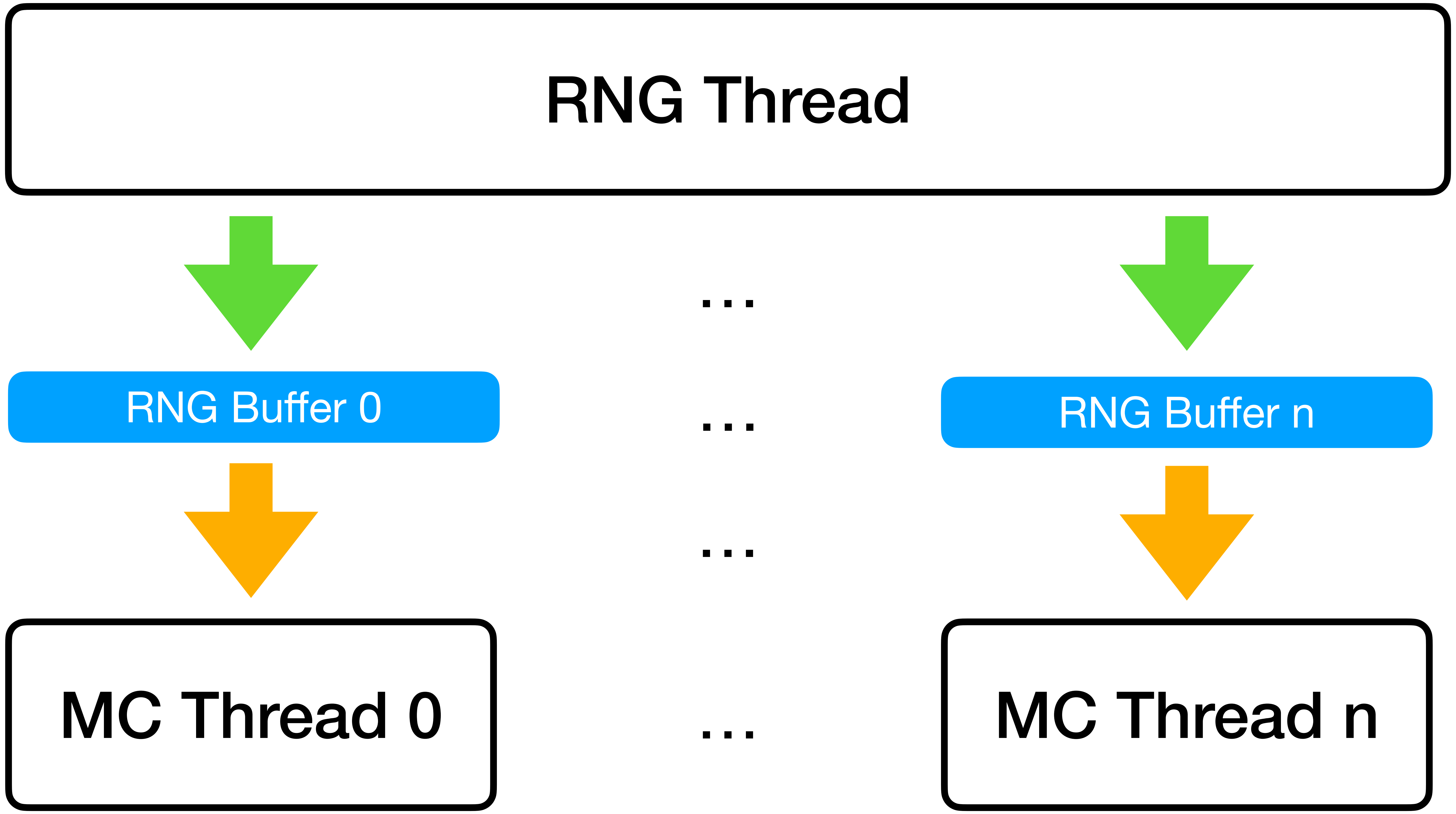
Parallel Random Generators

The Centralized Approach



Parallel Random Generators

The Centralized Approach



Parallel Random Generators

The Centralized Approach

- Advantages:
  - We avoid the problem to generate independent streams.
- Disadvantages:
  - **Not efficient: very slow!**
  - **Reproducibility: hard to debug!**

Parallel Random Generators

The Replicated Approach

- For each thread of our simulation, we have a RNG with the same seed or a unique seed:
  - We may use **parametrization**; i.e., different parameters for each RNG:
    - This is hard for thousands of threads.
- Advantages:
  - Efficiency: very fast;
  - Easy to implement.

Parallel Random Generators

The Replicated Approach

- Disadvantages:
  - **Are the streams independent? We cannot guarantee it; we may have correlation between drawn numbers.**
    - In MT, a solution is that the seed is a mix between the potential seed and the unique ID of the thread.
    - NOTE: if the period is huge, this may be a viable option:
      - There is a possibility of overlap:
        - The probability is often negligible.

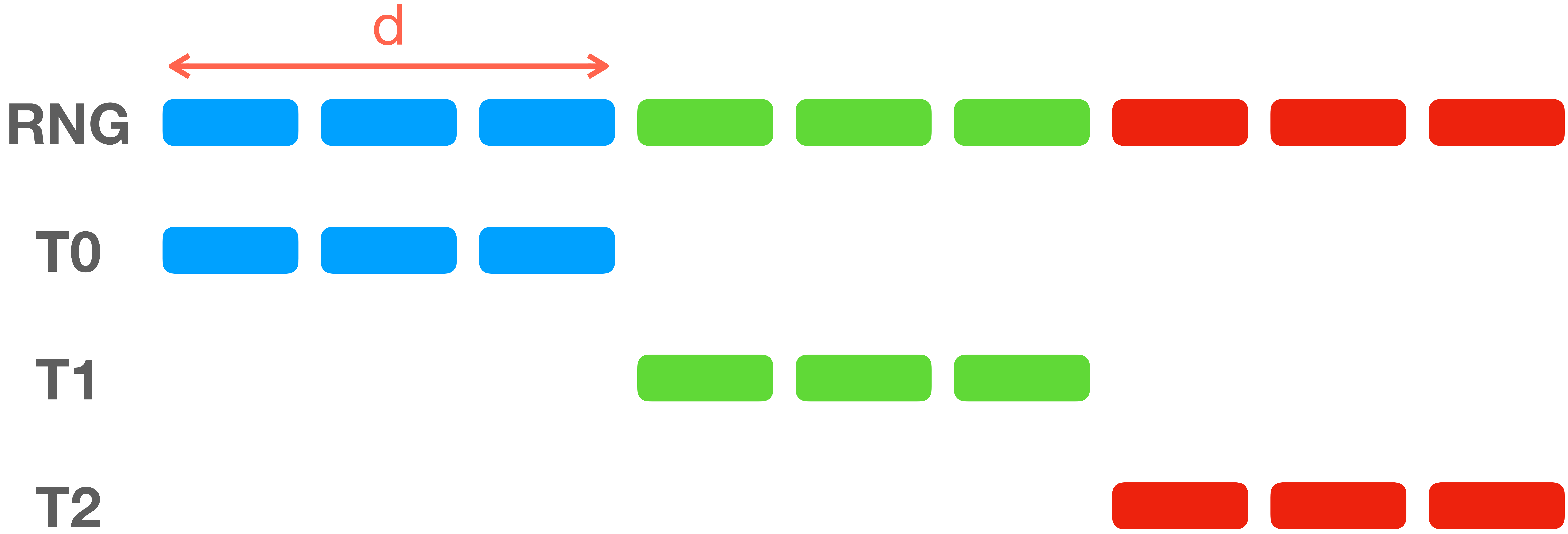
Parallel Random Generators

The Distributed Approach

- The generation of a single sequence is partitioned among many generators; i.e., one for each thread.
- Advantages:
  - Efficient;
- Disadvantages:
  - **Hard to implement it!**

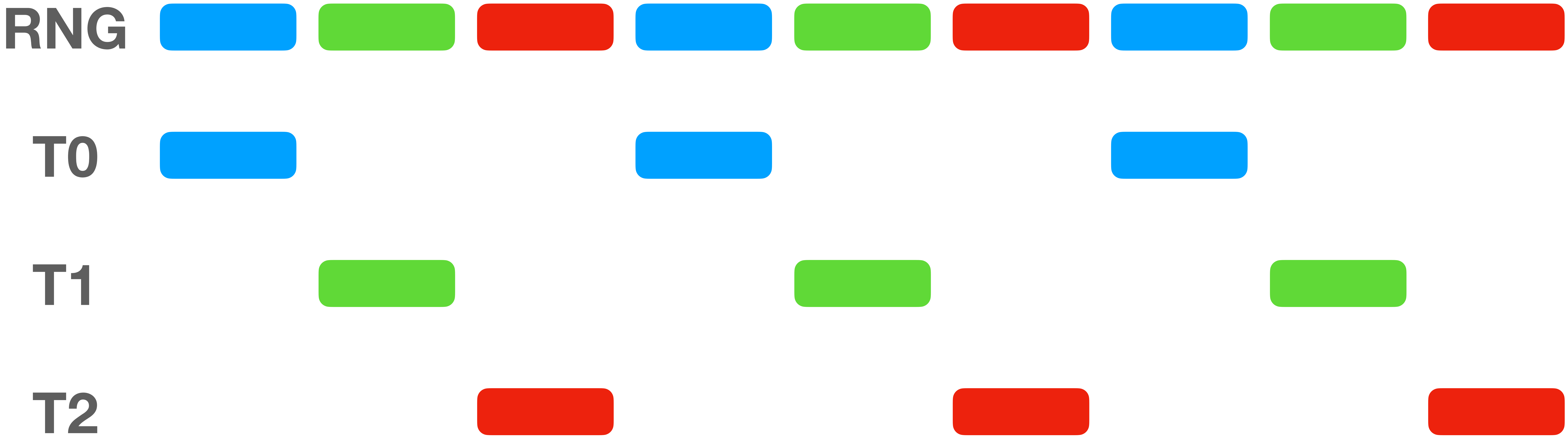
Parallel Random Generators

The Distributed Approach: Block Splitting



Parallel Random Generators

The Distributed Approach: Leap Frog



Parallel Random Generators

The Distributed Approach

- What do we need for implementing these approaches?
  - We need to know how to skip numbers: we need a RNG that can skip to the d-th number:
    - MT can do that.
  - Block splitting:
    - We need to know how many numbers are consumed before synchronization!

Conclusions

Conclusions

Wrapping Up

- Try to use the latest RNGs that works; e.g., Mersenne Twister.
- Write a wrapper RNG class; so you can change your RNG when a better one comes out.
- Try to write a testing example, and test different RNGs.
- Try to avoid using too many numbers from the same RNG.

Bibliography

- Art Owen. “Chapter 3: Uniform Random Numbers” from the book “Monte Carlo theory, methods and examples”. 2013.
- Jeff Wehrwein. "Random Number Generation”. Senior Thesis in Mathematics. 2007.
- Donald E. Knuth. “The Art of Computer Programming, Volume 2: Seminumerical Algorithms (second edition)”. Addison Wesley, 1997.
- Lewis, Peter A. W., A. S. Goodman and J. M. Miller. “A Pseudo-Random Number Generator for the System 350”, IBM Systems Journal, 8(2), 136–45. 1969.

Thank you for your attention!