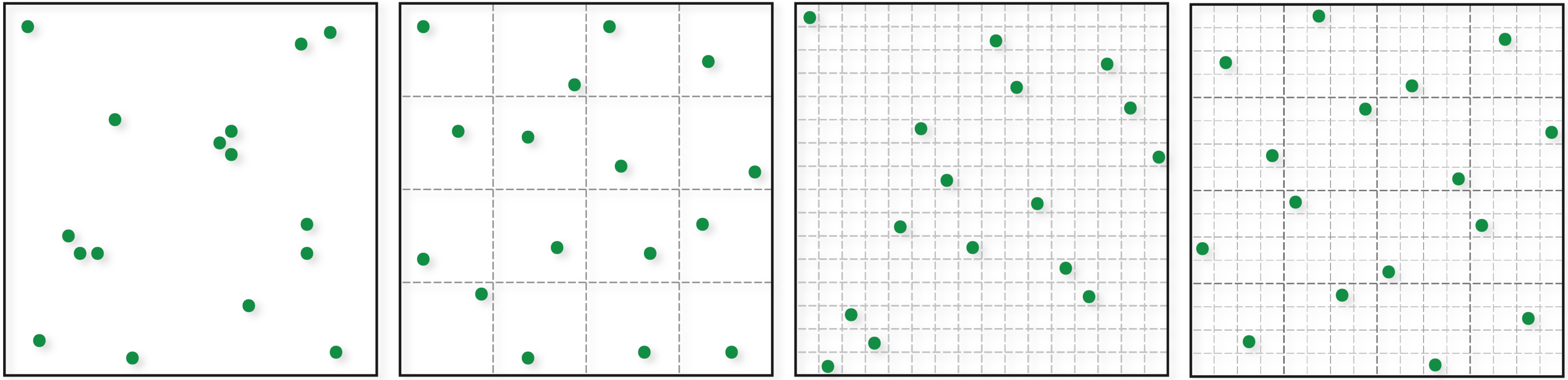
Improved sampling and quasi-Monte Carlo



15-468, 15-668, 15-868
Physics-based Rendering
Spring 2025, Lecture 9

http://graphics.cs.cmu.edu/courses/15-468

Course announcements

- Programming assignment 2 posted, due Friday 2/28 at 23:59.
  - How many of you have looked at/started/finished it?
  - Any questions?

Overview of today’s lecture

- Stratified sampling.
- Uncorrelated jitter.
- N-rooks.
- Multi-jittered sampling.
- Poisson disk sampling.
- Discrepancy.
- Quasi-Monte Carlo.
- Low-discrepancy sequences.

Slide credits

Most of these slides were directly adapted from:

- Wojciech Jarosz (Dartmouth).

Strategies for Reducing Variance

$$\sigma\left[\langle F^N \rangle\right] = \frac{1}{\sqrt{N}}\sigma\left[Y\right]$$ remember, this assumed uncorrelated samples

Reduce the variance of Y

- Importance sampling

Relax assumption of uncorrelated samples

Quick aside: our approach so far

To estimate an integral

$$I = \int_S f(x)\,\mathrm{d}x$$

1. we draw uniform random variates $u_i \in [0,1)^D$,
2. we transform them as $x_i = g(u_i)$,
3. we form the Monte Carlo estimate:

$$\tilde{I} = \frac{1}{N}\sum \frac{f(x_i)}{p(x_i)} = \frac{1}{N}\sum \frac{f\big(g(u_i)\big)}{1/\big|J_u^g(u_i)\big|} = \frac{1}{N}\sum f\big(g(u_i)\big)\big|J_u^g(u_i)\big|$$

Equivalent view: primary sample space

To estimate an integral

$$I = \int_S f(x)\,\mathrm{d}x$$

1. we make a change of variables $x = g(u)$, and *rewrite the integral* as

$$I = \int_{[0,1)^D} f\big(g(u)\big)\left|J_u^g(u)\right|\,\mathrm{d}u$$

This is called the *primary sample space* reparameterization

2. we draw uniform random variates $u_i \in [0,1)^D$,

3. we form the Monte Carlo estimate *of the rewritten integral*:

$$\tilde{I} = \frac{1}{N}\sum f\big(g(u_i)\big)\left|J_u^g(u_i)\right|$$

Same result as before!

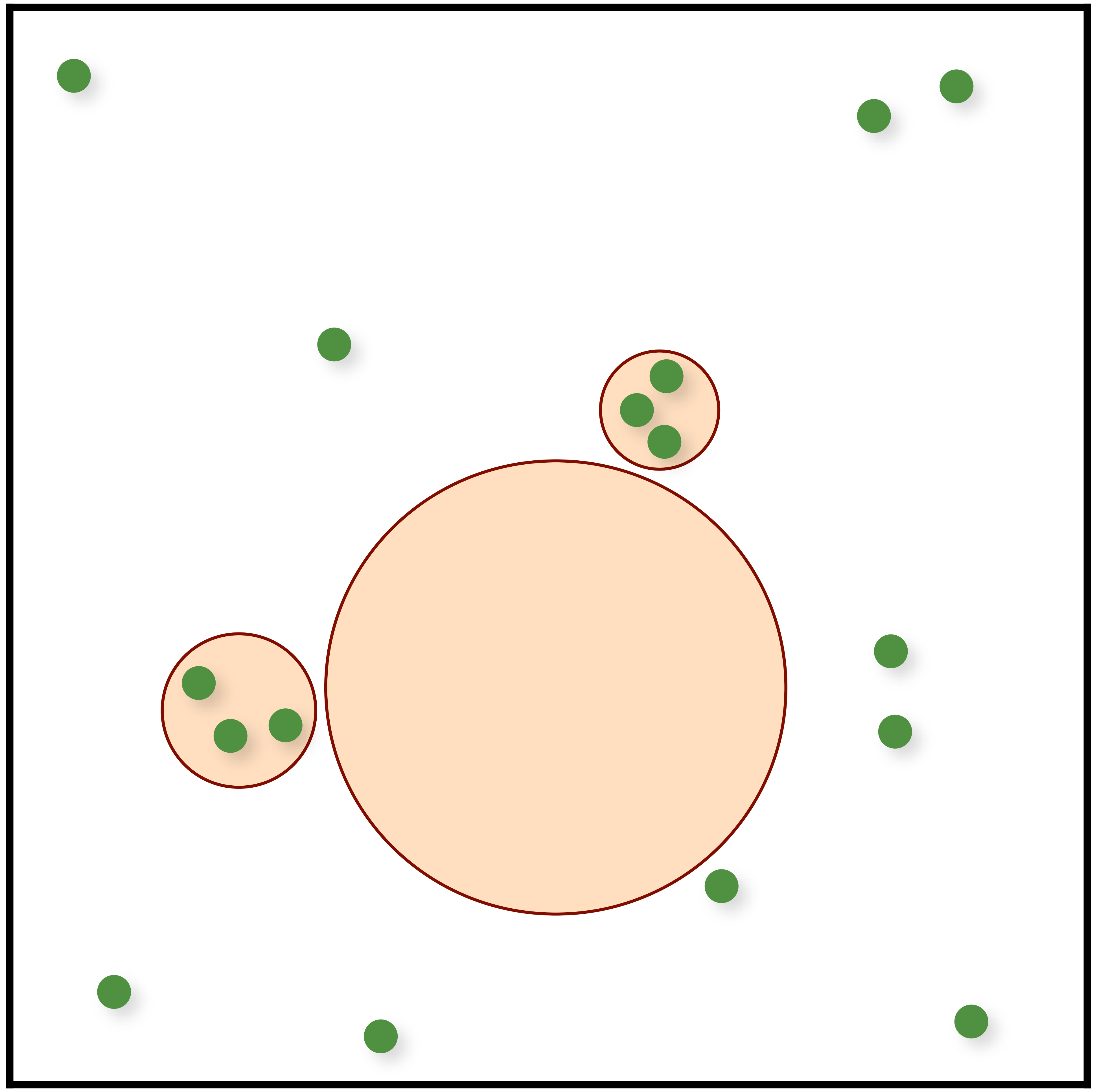
Equivalent view: primary sample space

No matter what integral we are estimating, we can focus our attention on sampling canonical uniform random variables in the hypercube.

This is the approach we take in this lecture.

Independent Random Sampling

```
for (int k = 0; k < num; k++)
{
    samples(k).x = randf();
    samples(k).y = randf();
}
```



✓ Trivially extends to higher dimensions

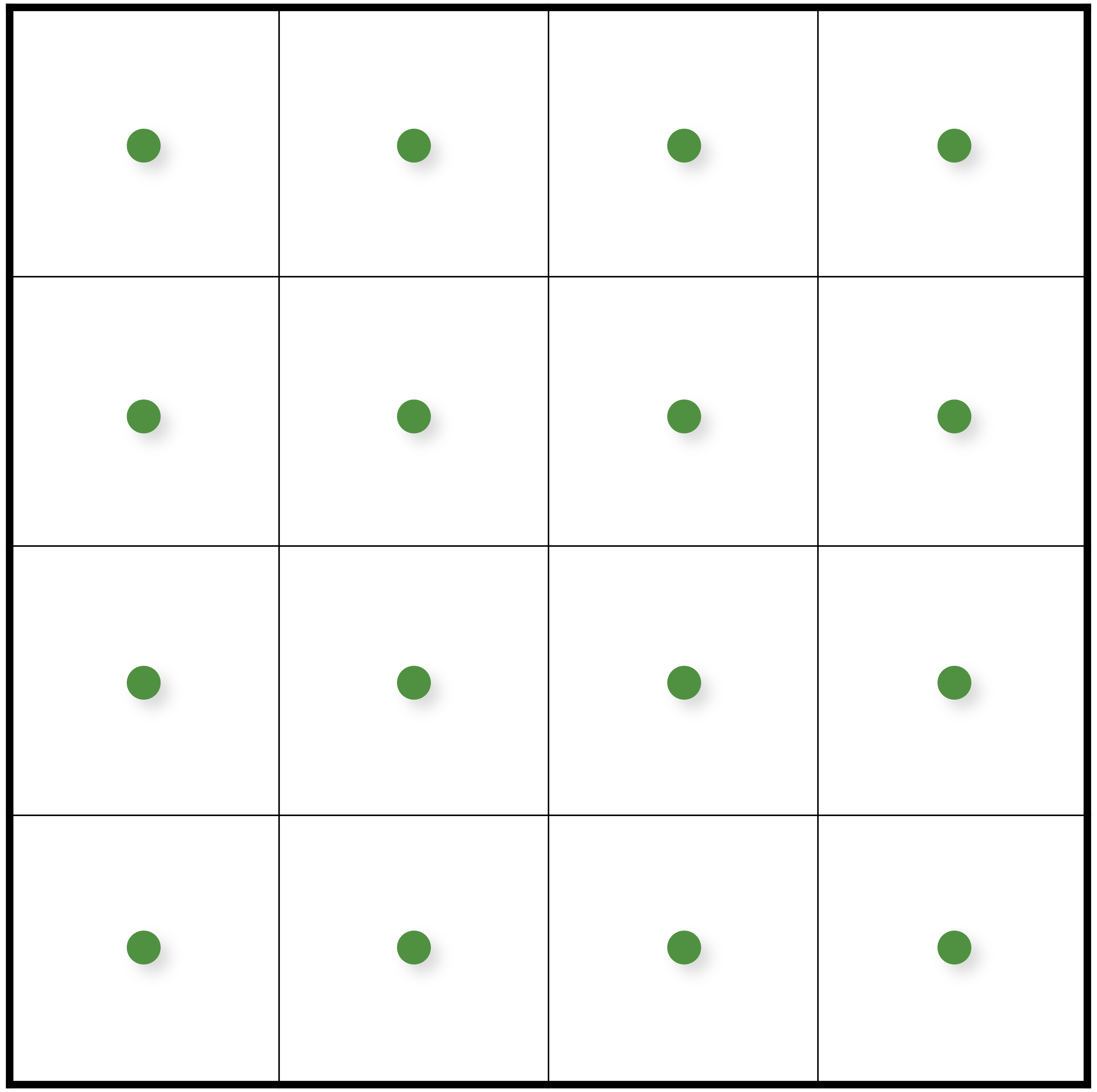
✓ Trivially progressive and memory-less

✗ Big gaps

✗ Clumping

Regular Sampling

```
for (uint i = 0; i < numX; i++)
    for (uint j = 0; j < numY; j++)
    {
        samples(i,j).x = (i + 0.5)/numX;
        samples(i,j).y = (j + 0.5)/numY;
    }
```



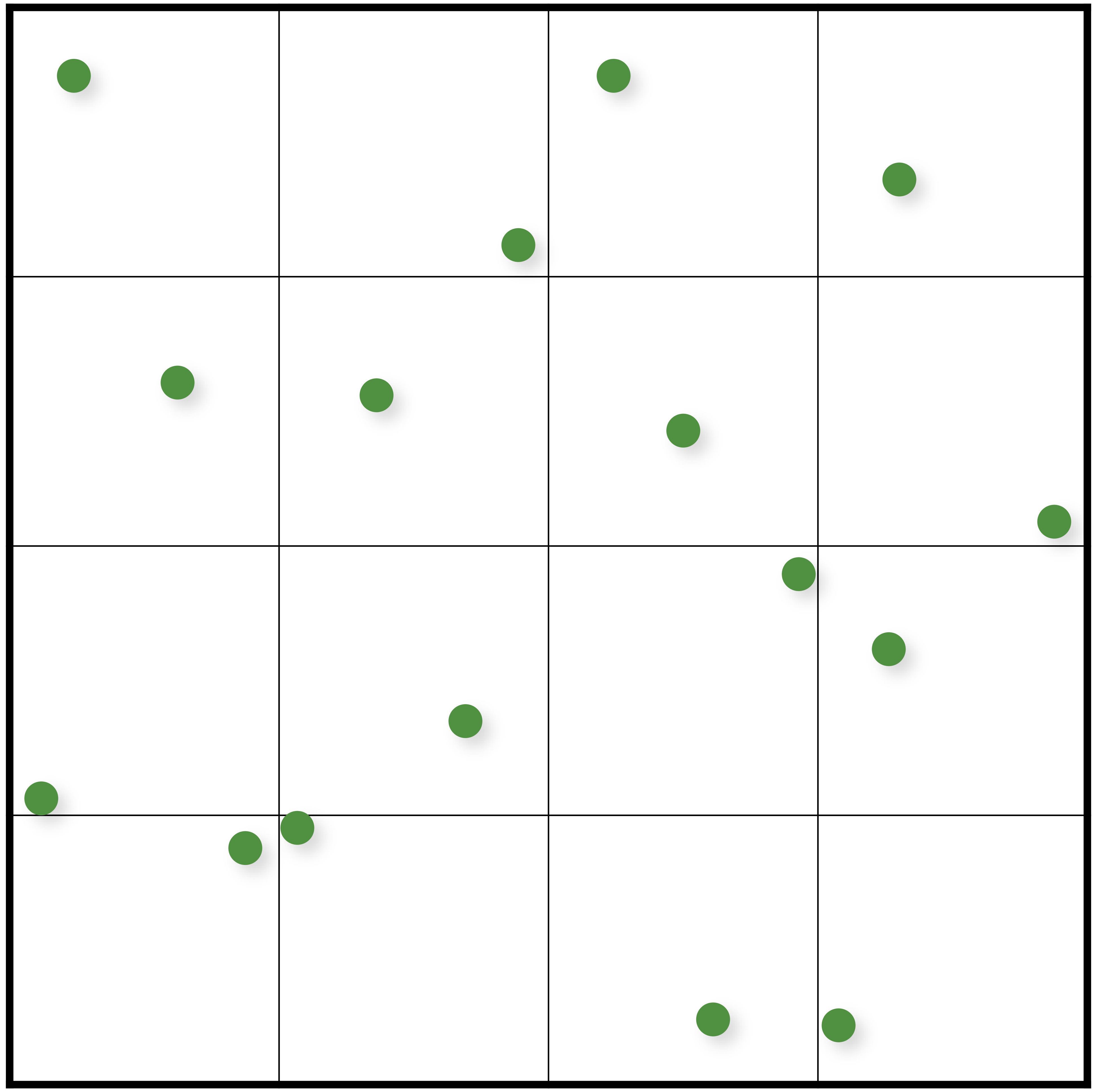
✓ Extends to higher dimensions, but…

✗ Curse of dimensionality

✗ Aliasing

Jittered/Stratified Sampling

```
for (uint i = 0; i < numX; i++)
    for (uint j = 0; j < numY; j++)
    {
        samples(i,j).x = (i + randf())/numX;
        samples(i,j).y = (j + randf())/numY;
    }
```



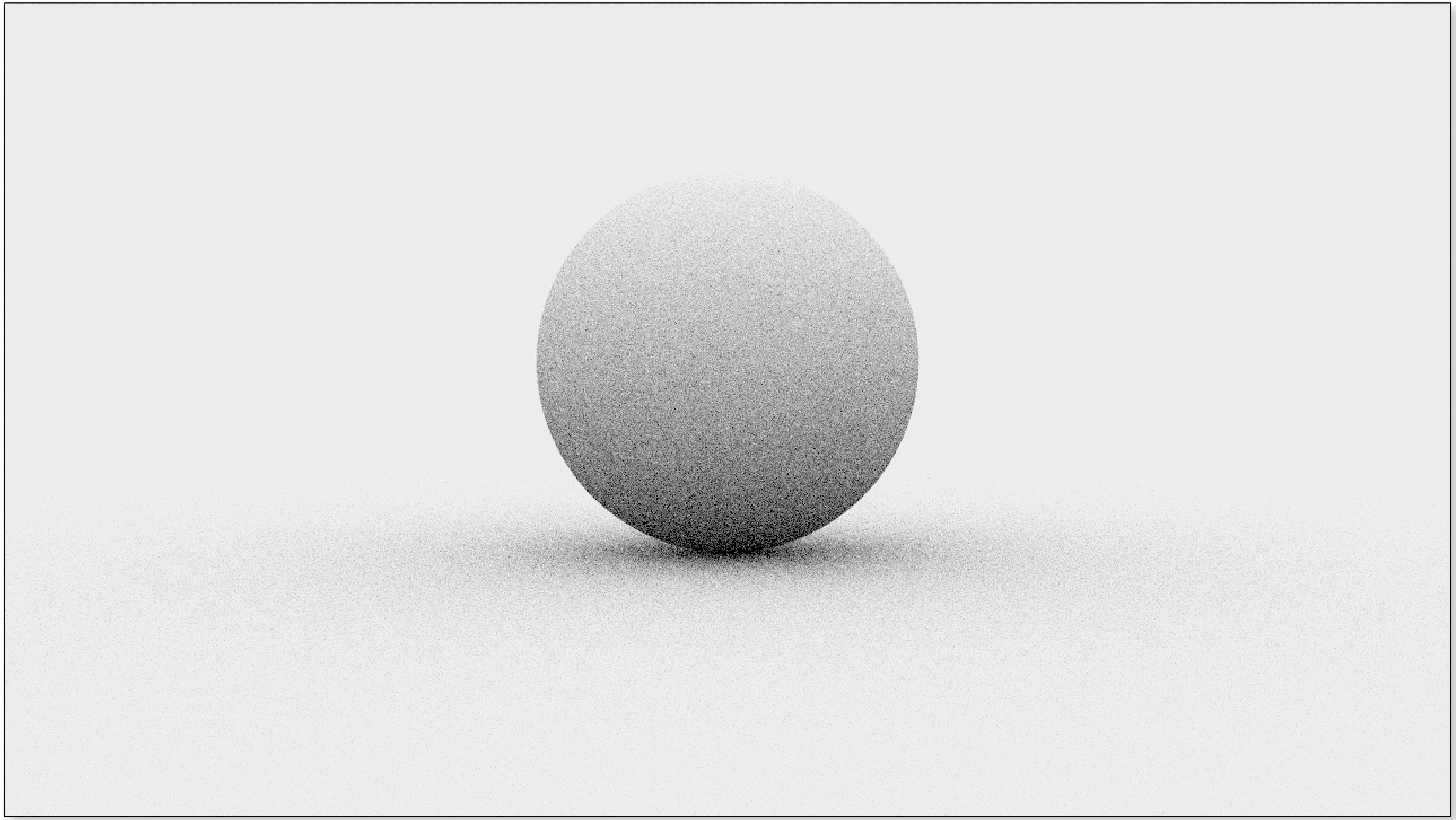
✓ Provably cannot increase variance

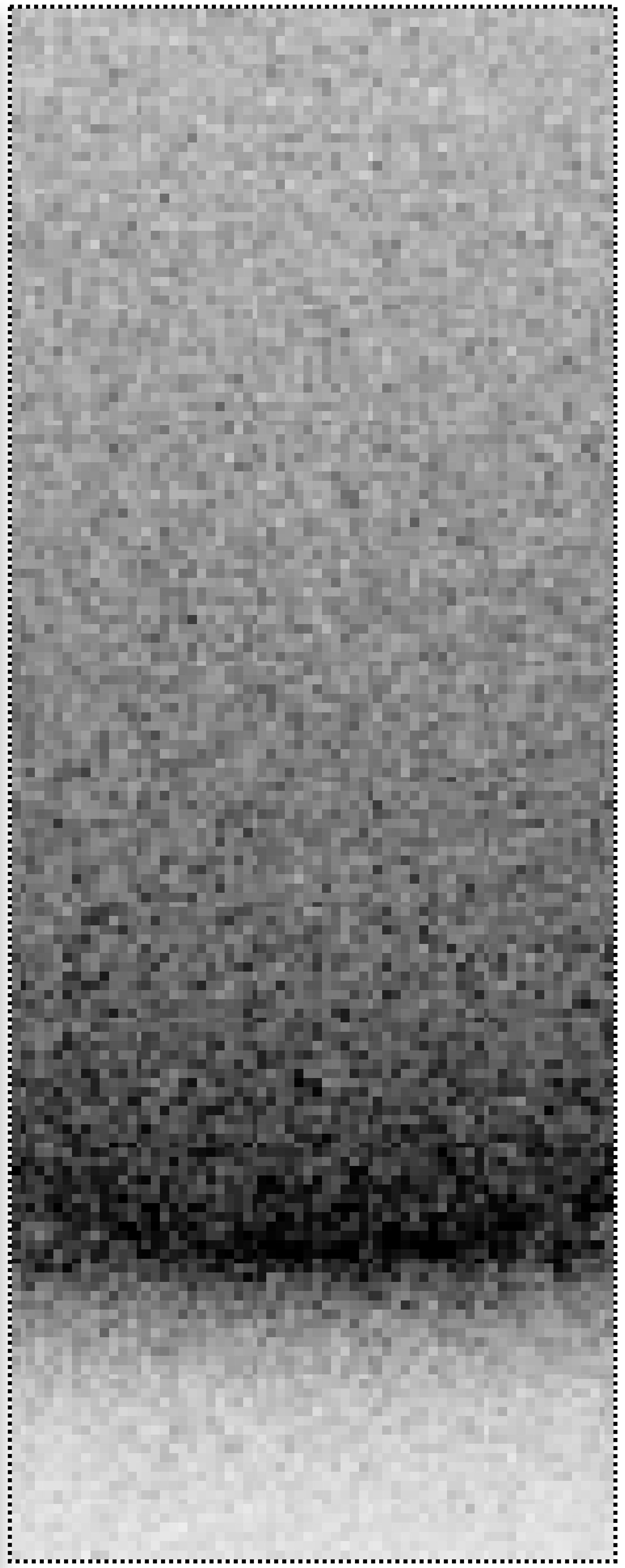
✓ Extends to higher dimensions, but…

✗ Curse of dimensionality

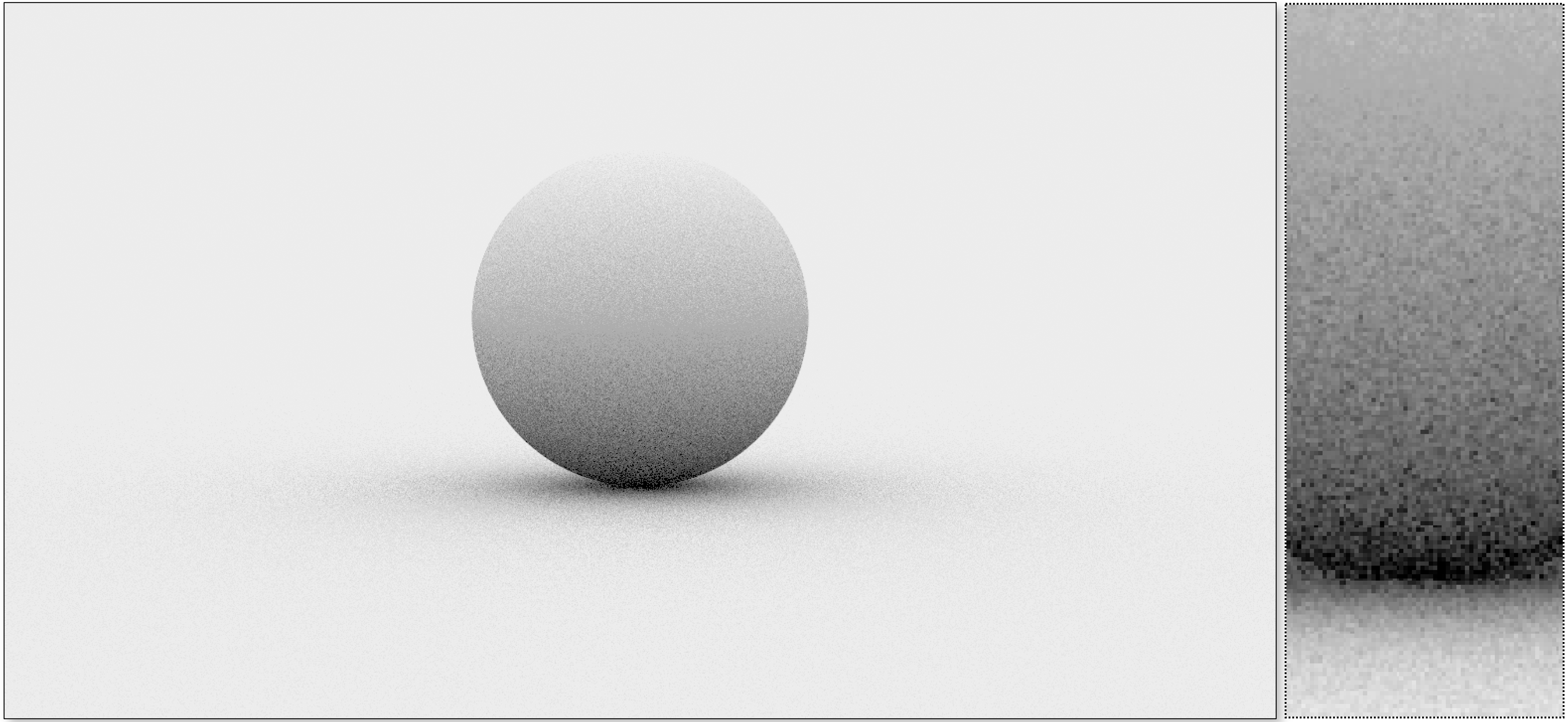
✗ Not progressive

Monte Carlo (16 random samples)





Monte Carlo (16 jittered samples)



Stratifying in Higher Dimensions

Stratification requires O(N^d) samples

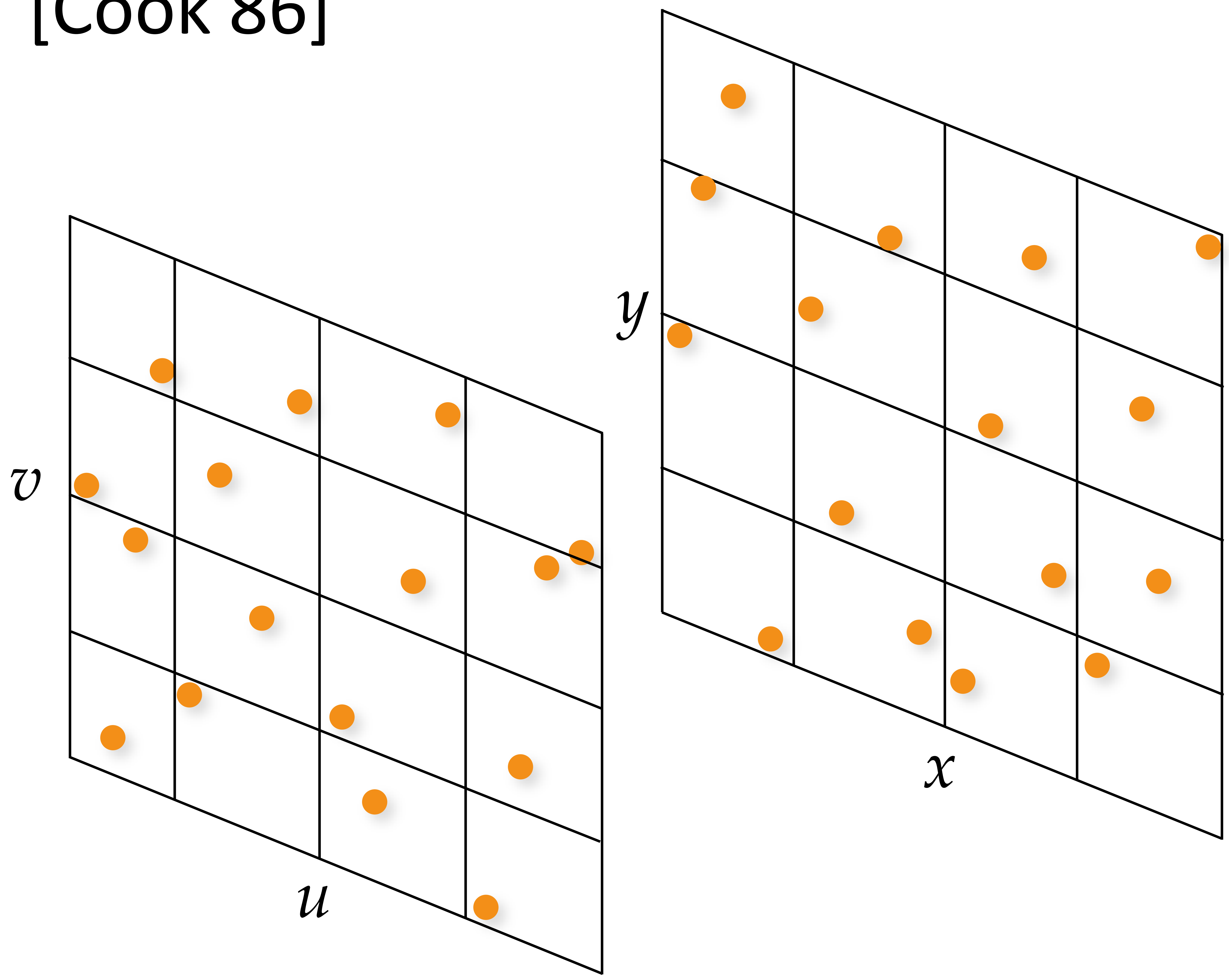
- e.g. pixel (2D) + lens (2D) + time (1D) = 5D
    - splitting 2 times in 5D = 2^5 = 32 samples
    - splitting 3 times in 5D = 3^5 = 243 samples!

Inconvenient for large d

- cannot select sample count with fine granularity

“Padding” 2D points (Uncorrelated Jitter)

[Cook 86]



2D

(x_1, y_1)
(x_2, y_2)
(x_3, y_3)
(x_4, y_4)
$\vdots$

2D

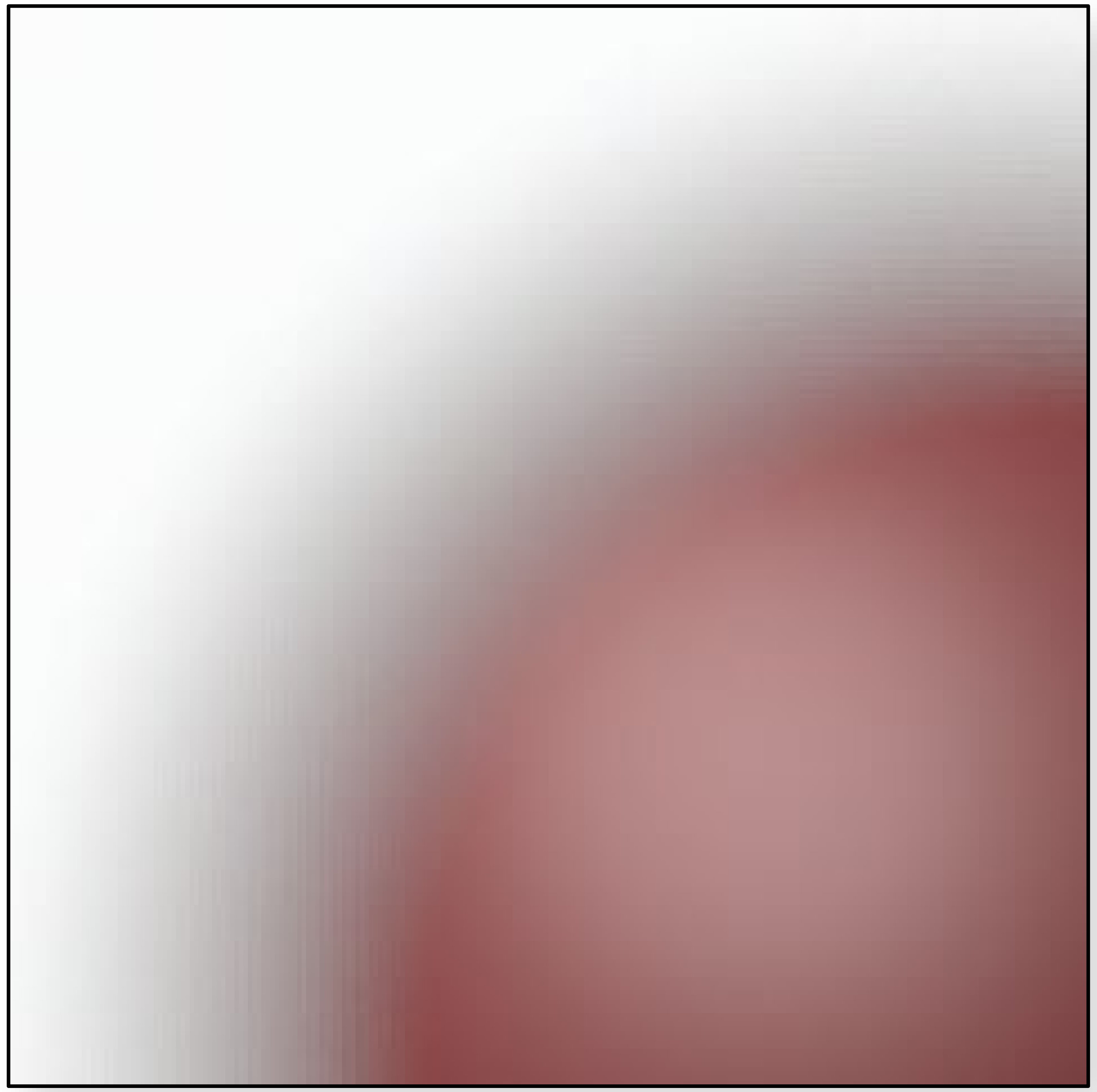
(u_1, v_1)
(u_2, v_2)
(u_3, v_3)
(u_4, v_4)
$\vdots$

4D

(x_1, y_1, u_3, v_3)
(x_2, y_2, u_1, v_1)
(x_3, y_3, u_4, v_4)
(x_4, y_4, u_2, v_2)
$\vdots$

Depth of Field (4D)

Reference



Random Sampling

Uncorrelated Jitter

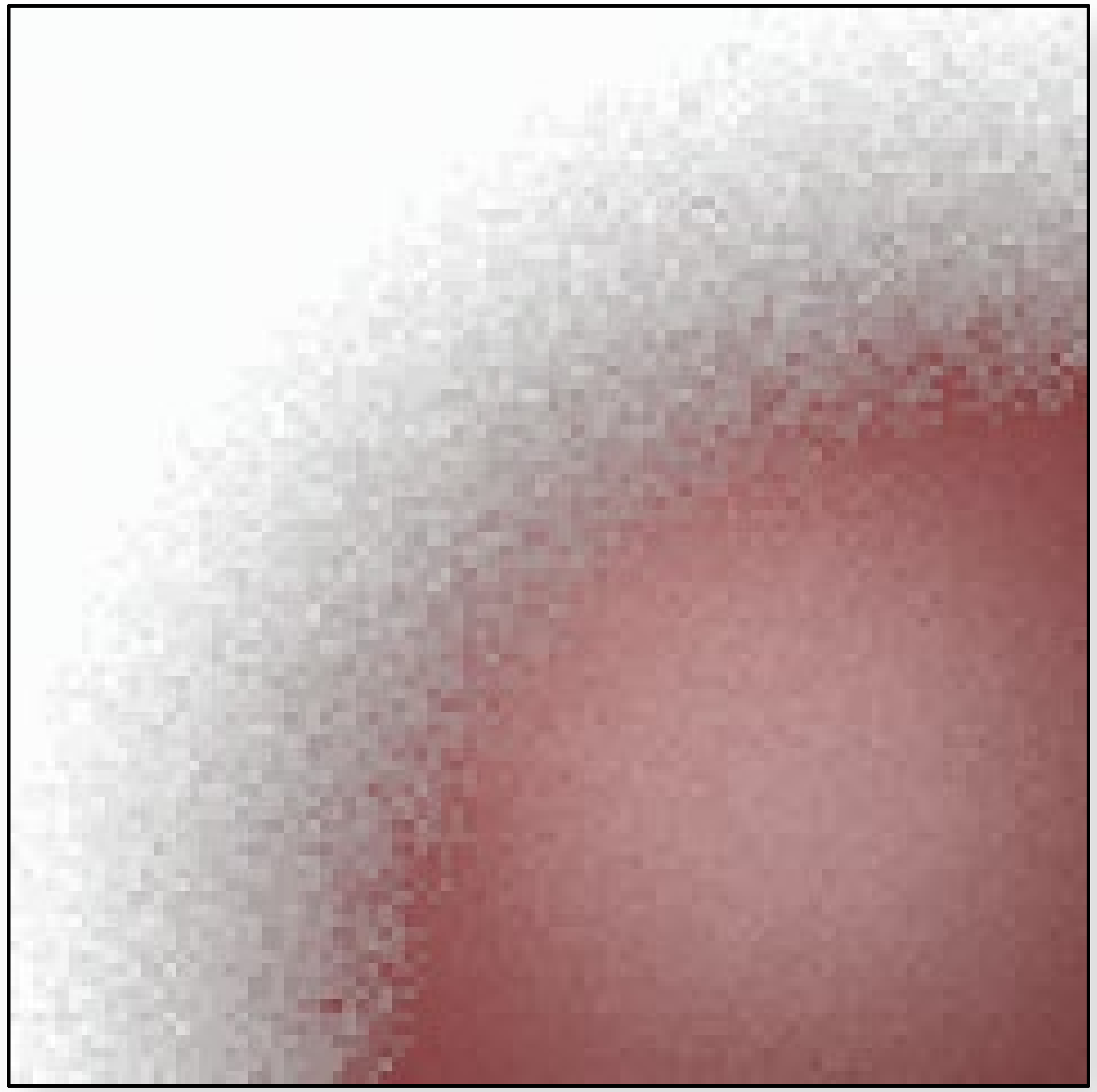


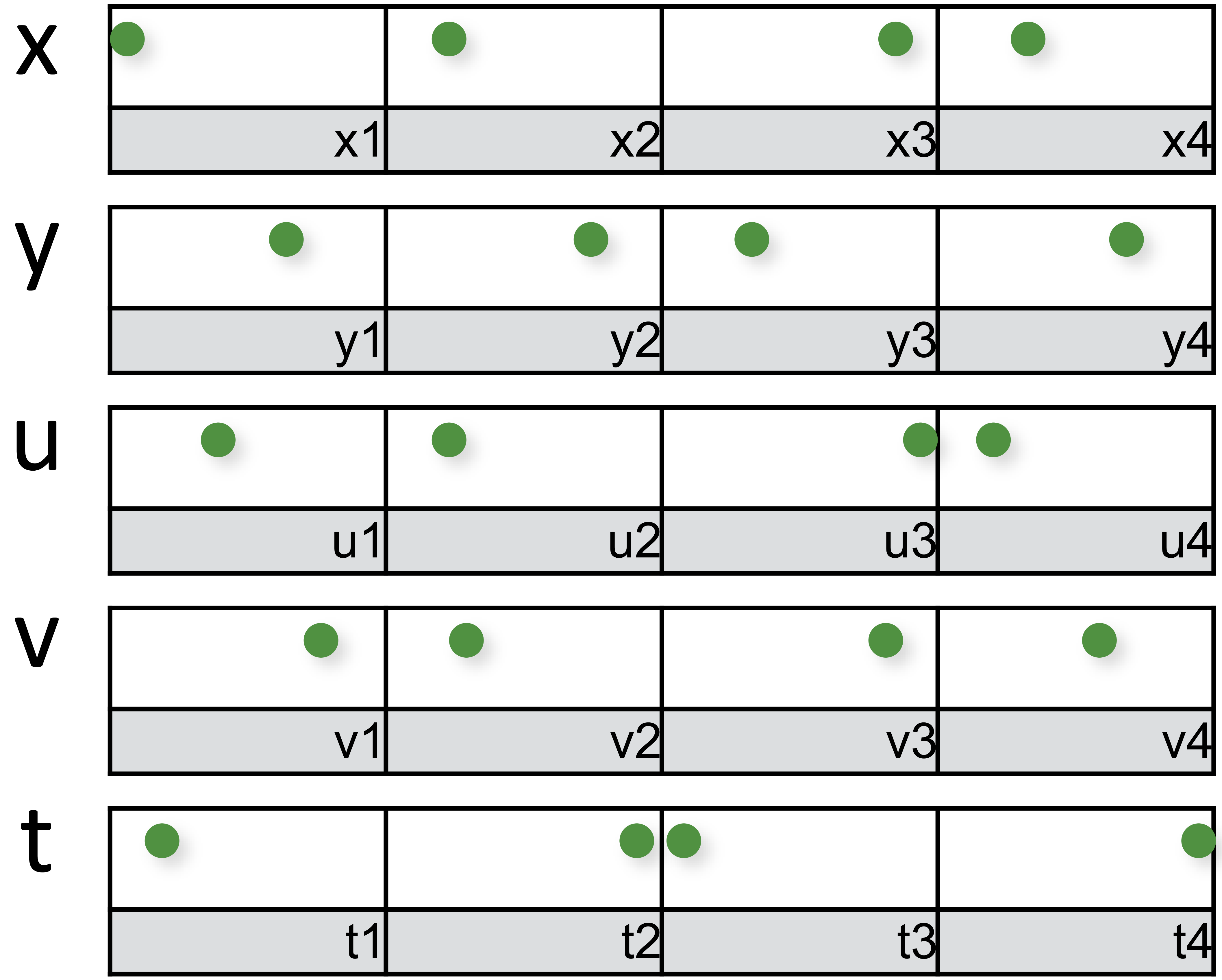
Image source: PBRTe2 [Pharr & Humphreys 2010]

Uncorrelated Jitter ➔ Latin Hypercube

Like uncorrelated jitter, but using 1D point sets

- for 5D: 5 separate 1D jittered point sets

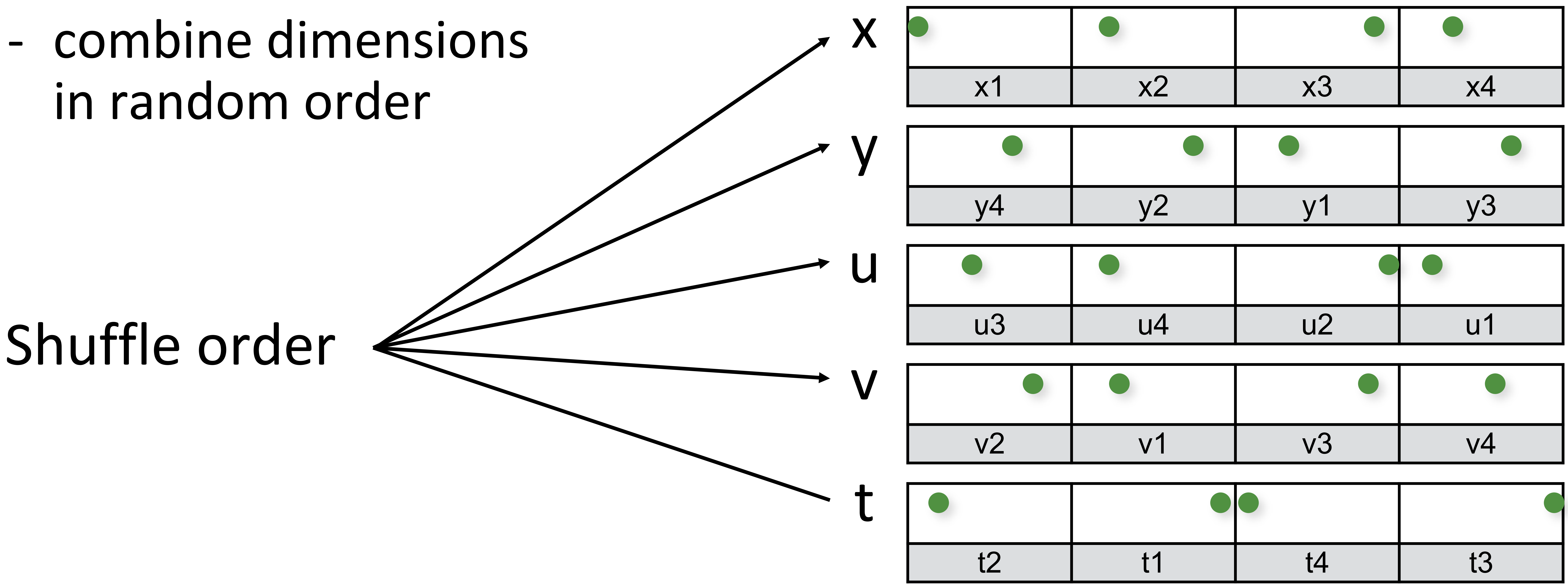
- combine dimensions
  in random order



Uncorrelated Jitter ➔ Latin Hypercube

Like uncorrelated jitter, but using 1D point sets

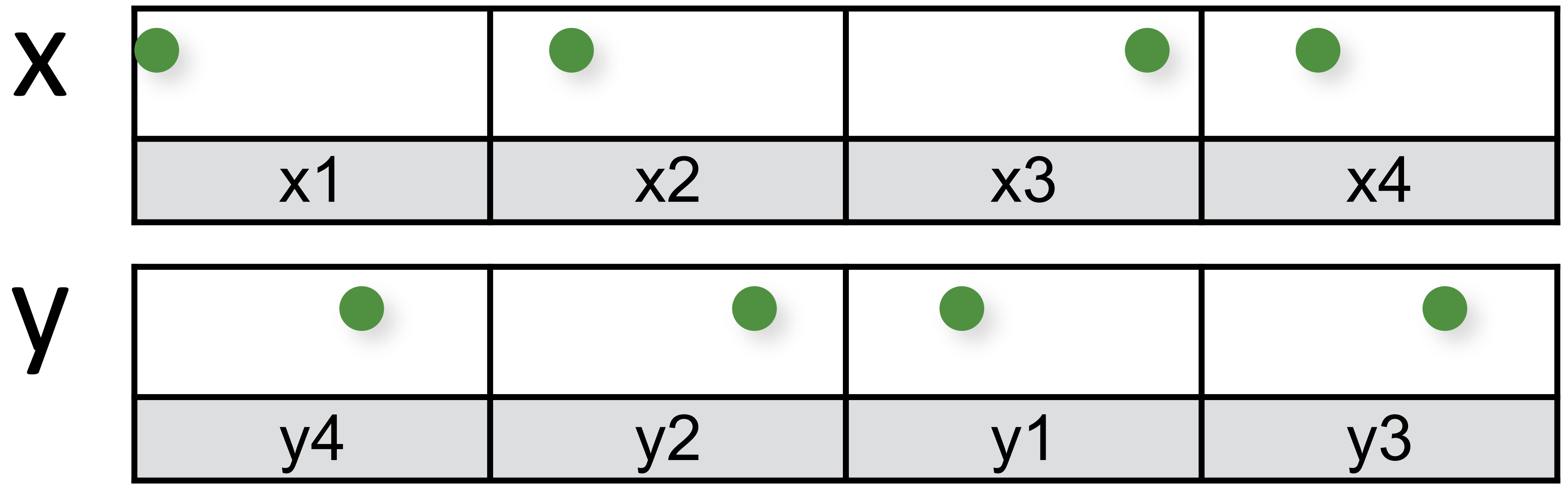
- for 5D: 5 separate 1D jittered point sets
- combine dimensions in random order



N-Rooks = 2D Latin Hypercube [Shirley 91]

Like uncorrelated jitter, but using 1D point sets

- for **2D**: **2** separate 1D jittered point sets
- combine dimensions
  in random order



Latin Hypercube (N-Rooks) Sampling

[Shirley 91]

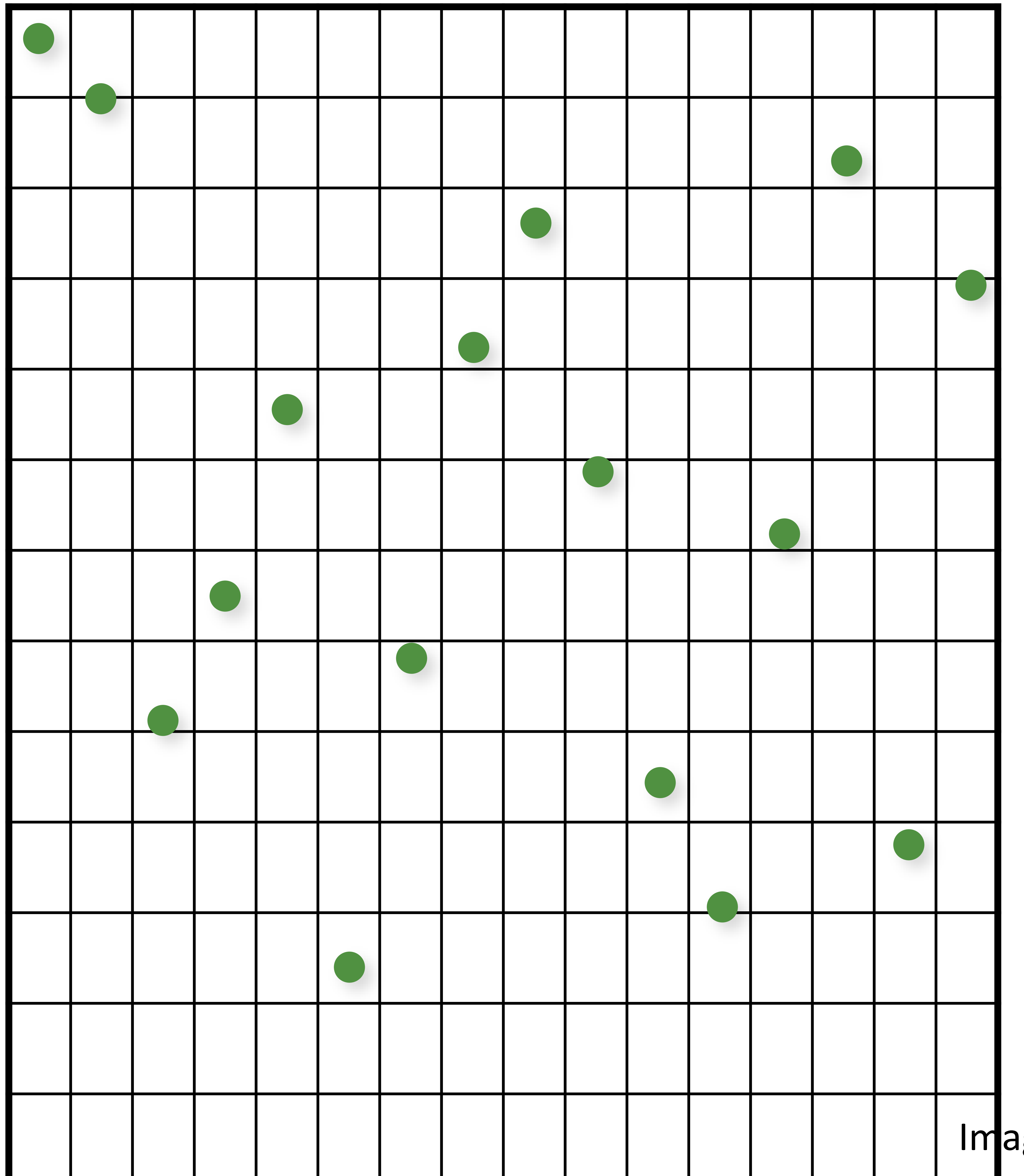
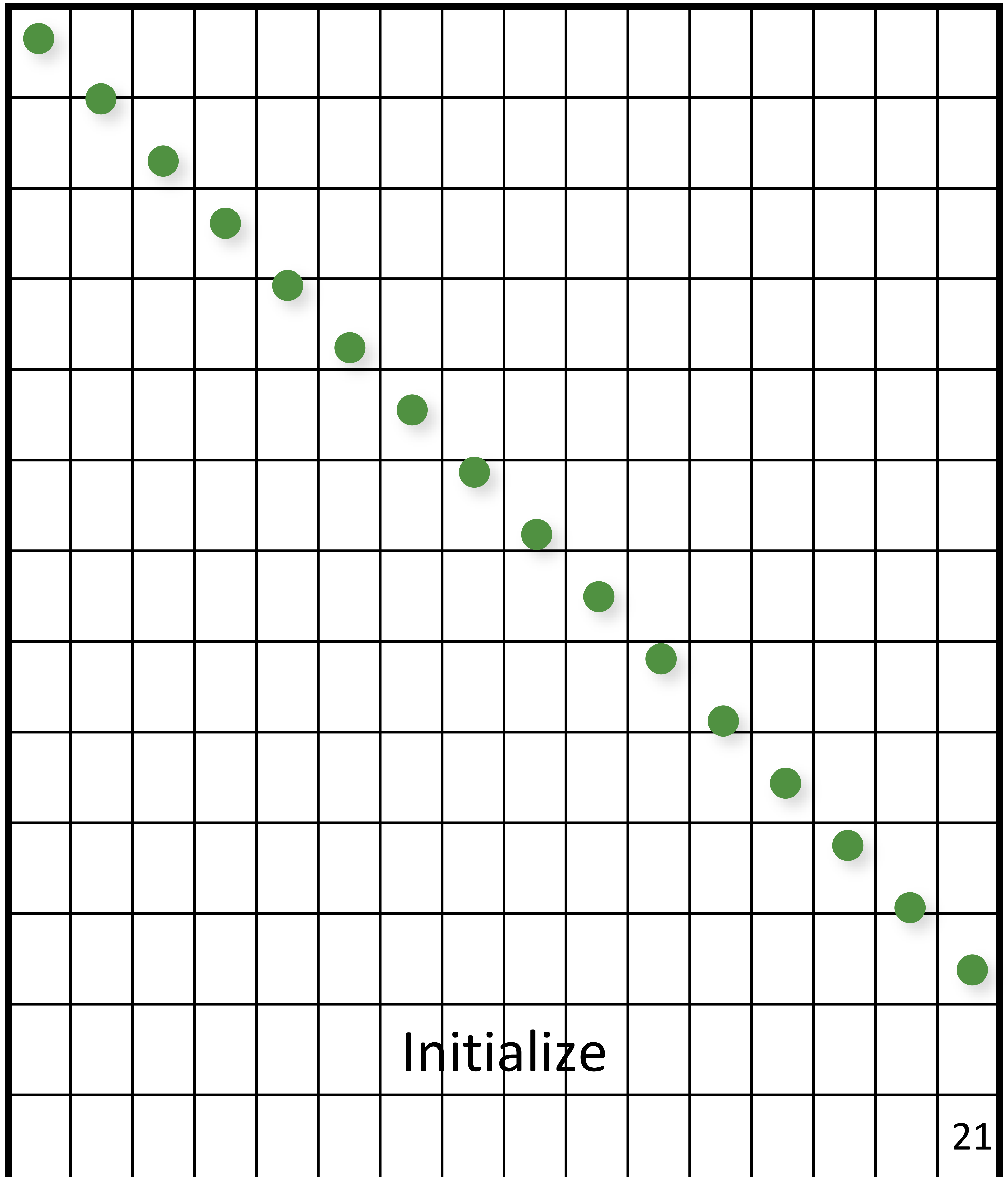


Image source: [Michael Maggs, CC BY-SA 2.5]

Latin Hypercube (N-Rooks) Sampling

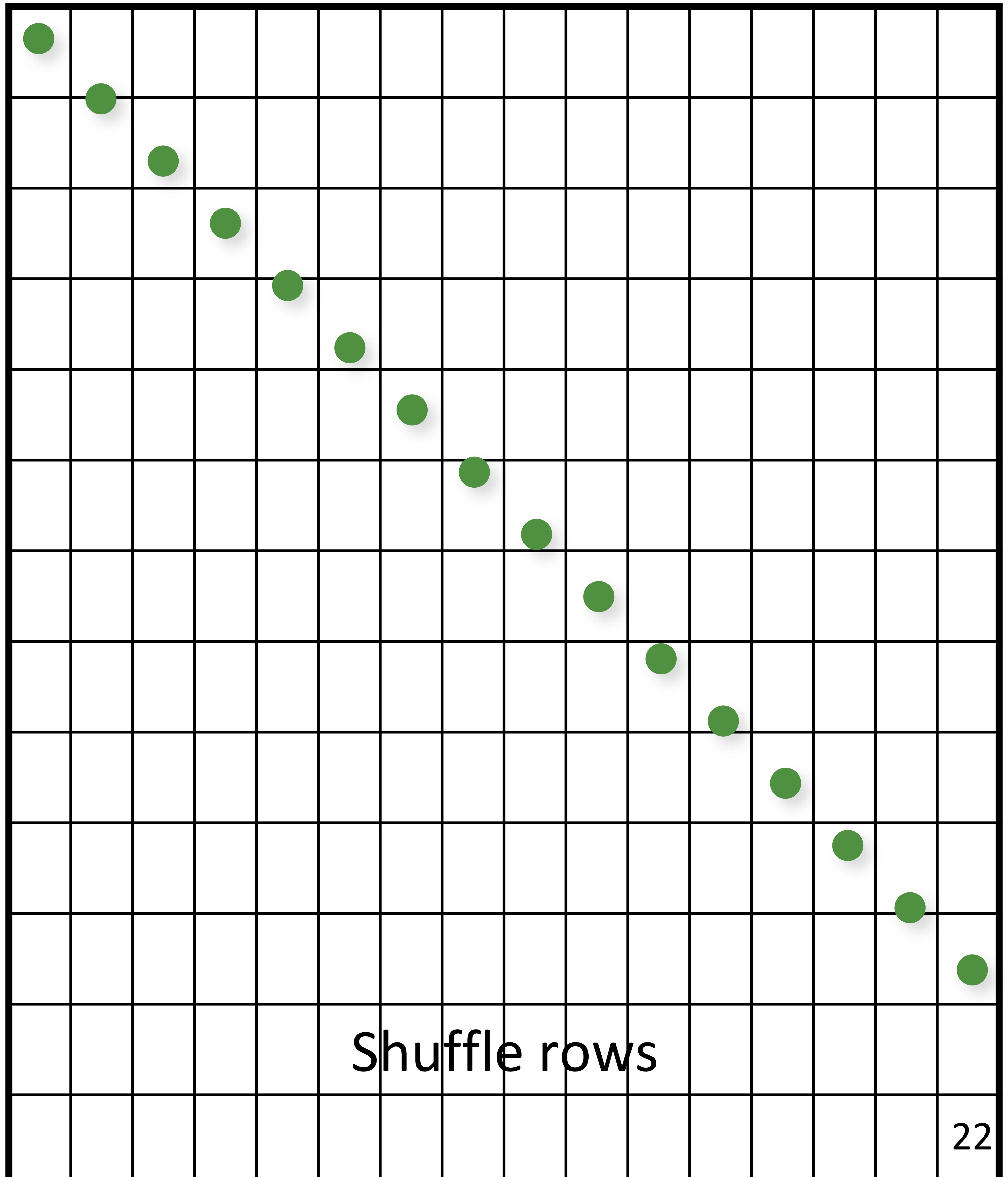
```
// initialize the diagonal
for (uint d = 0; d < numDimensions; d++)
    for (uint i = 0; i < numS; i++)
        samples(d,i) = (i + randf())/numS;
```

```
// shuffle each dimension independently
for (uint d = 0; d < numDimensions; d++)
    shuffle(samples(d,:));
```



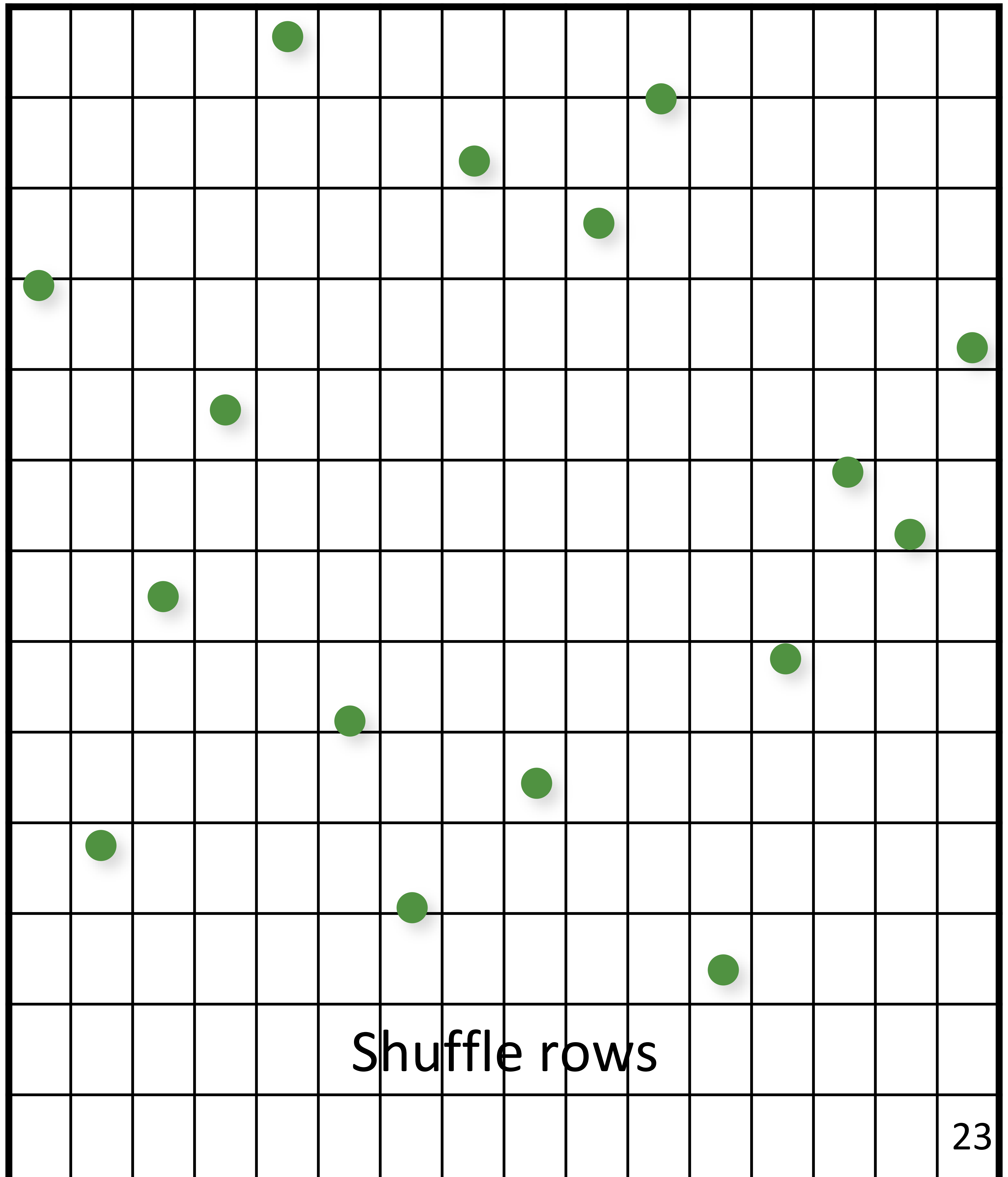
Latin Hypercube (N-Rooks) Sampling

```
// initialize the diagonal
for (uint d = 0; d < numDimensions; d++)
    for (uint i = 0; i < numS; i++)
        samples(d,i) = (i + randf())/numS;

// shuffle each dimension independently
for (uint d = 0; d < numDimensions; d++)
  shuffle(samples(d,:));
```



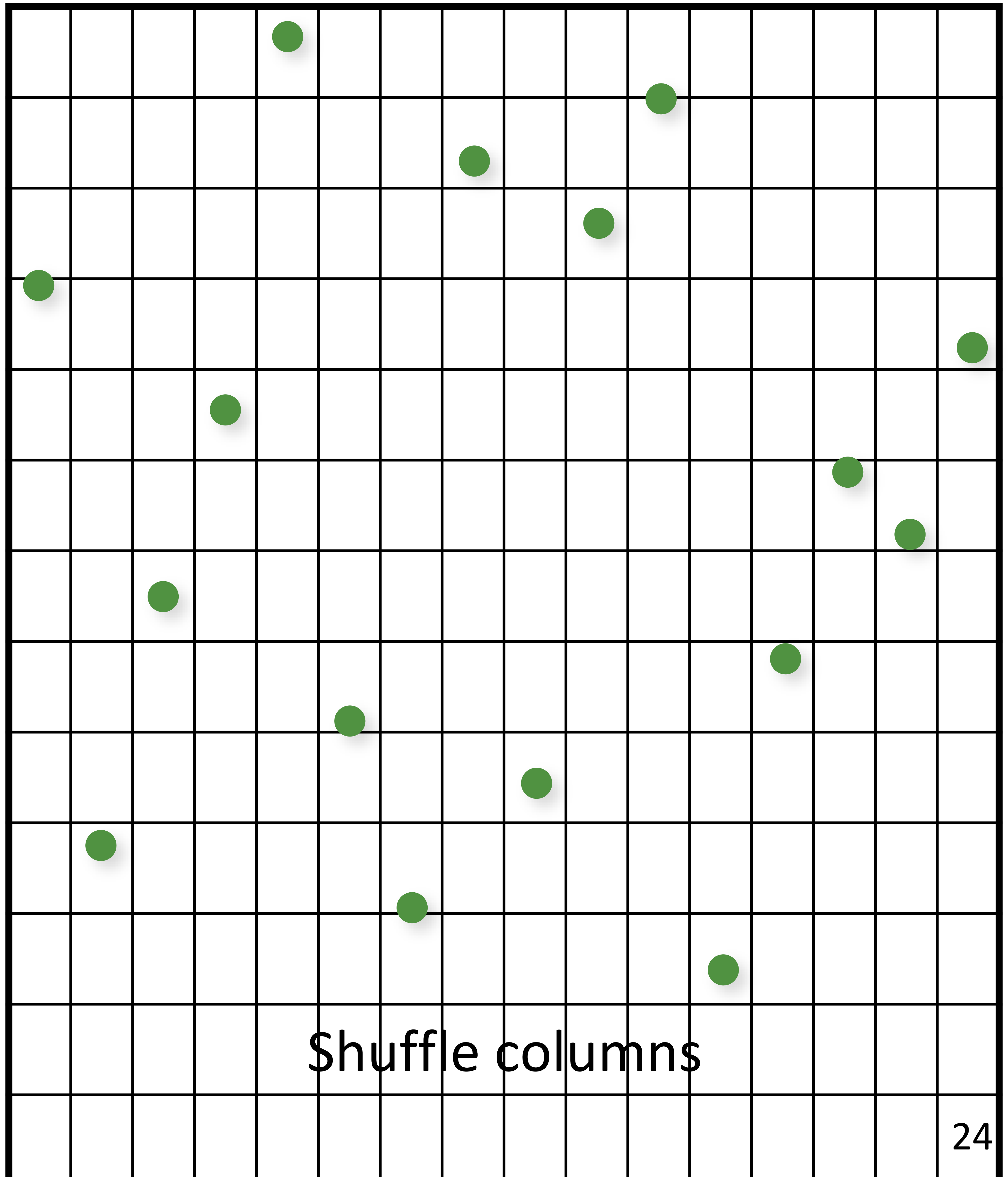
Latin Hypercube (N-Rooks) Sampling

```
// initialize the diagonal
for (uint d = 0; d < numDimensions; d++)
    for (uint i = 0; i < numS; i++)
        samples(d,i) = (i + randf())/numS;

// shuffle each dimension independently
for (uint d = 0; d < numDimensions; d++)
    shuffle(samples(d,:));
```



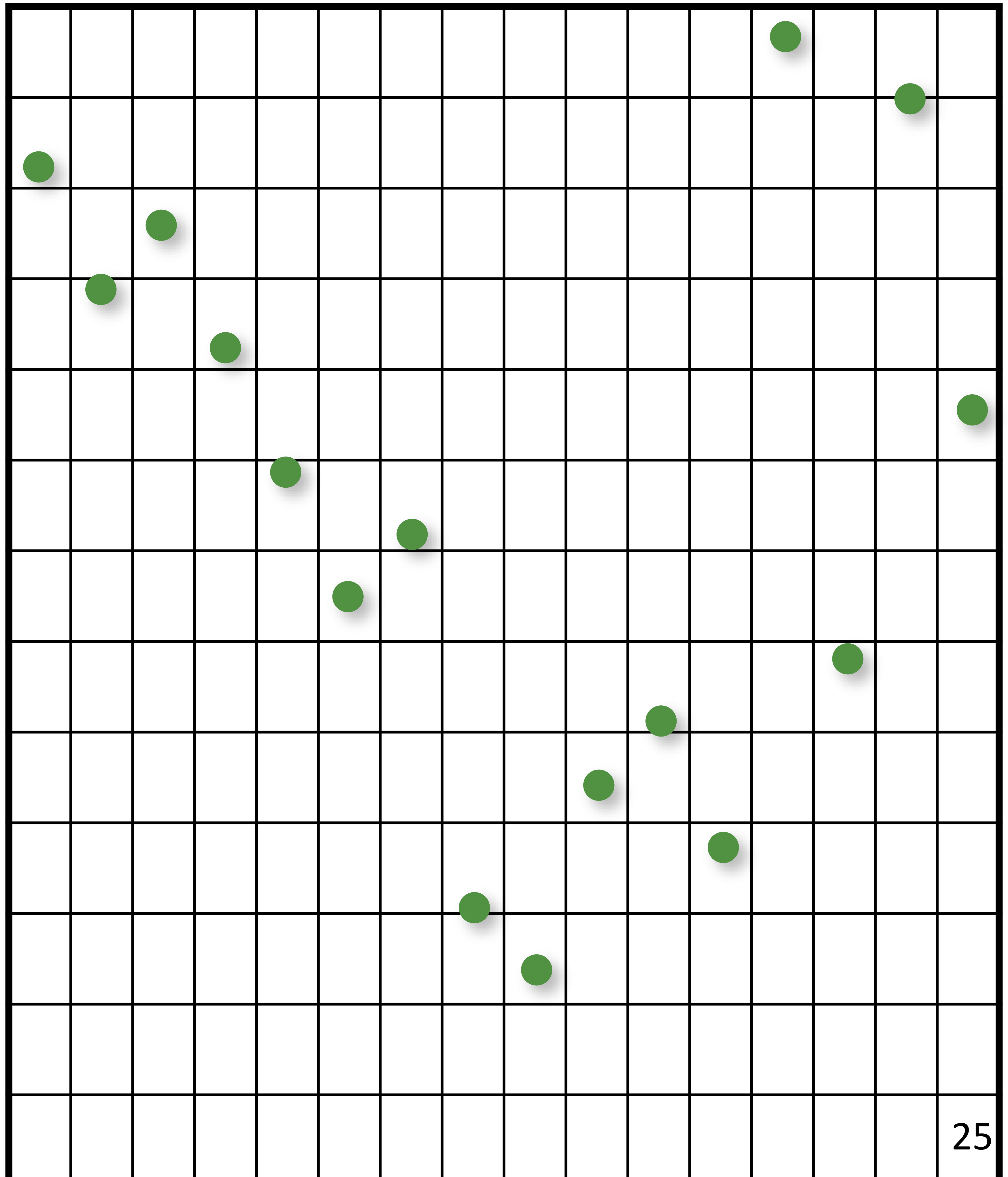
Latin Hypercube (N-Rooks) Sampling

```
// initialize the diagonal
for (uint d = 0; d < numDimensions; d++)
    for (uint i = 0; i < numS; i++)
        samples(d,i) = (i + randf())/numS;

// shuffle each dimension independently
for (uint d = 0; d < numDimensions; d++)
    shuffle(samples(d,:));
```

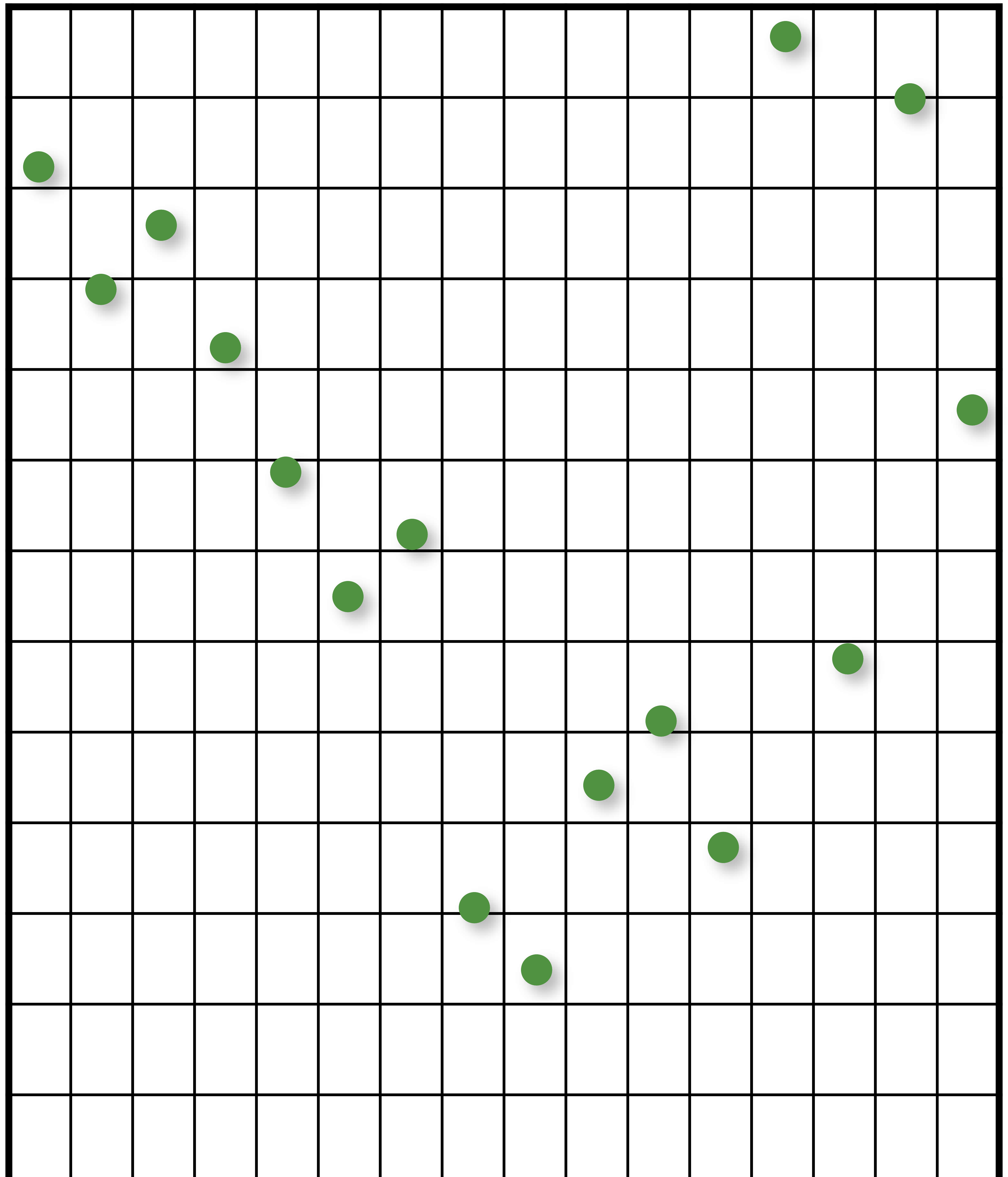


Latin Hypercube (N-Rooks) Sampling

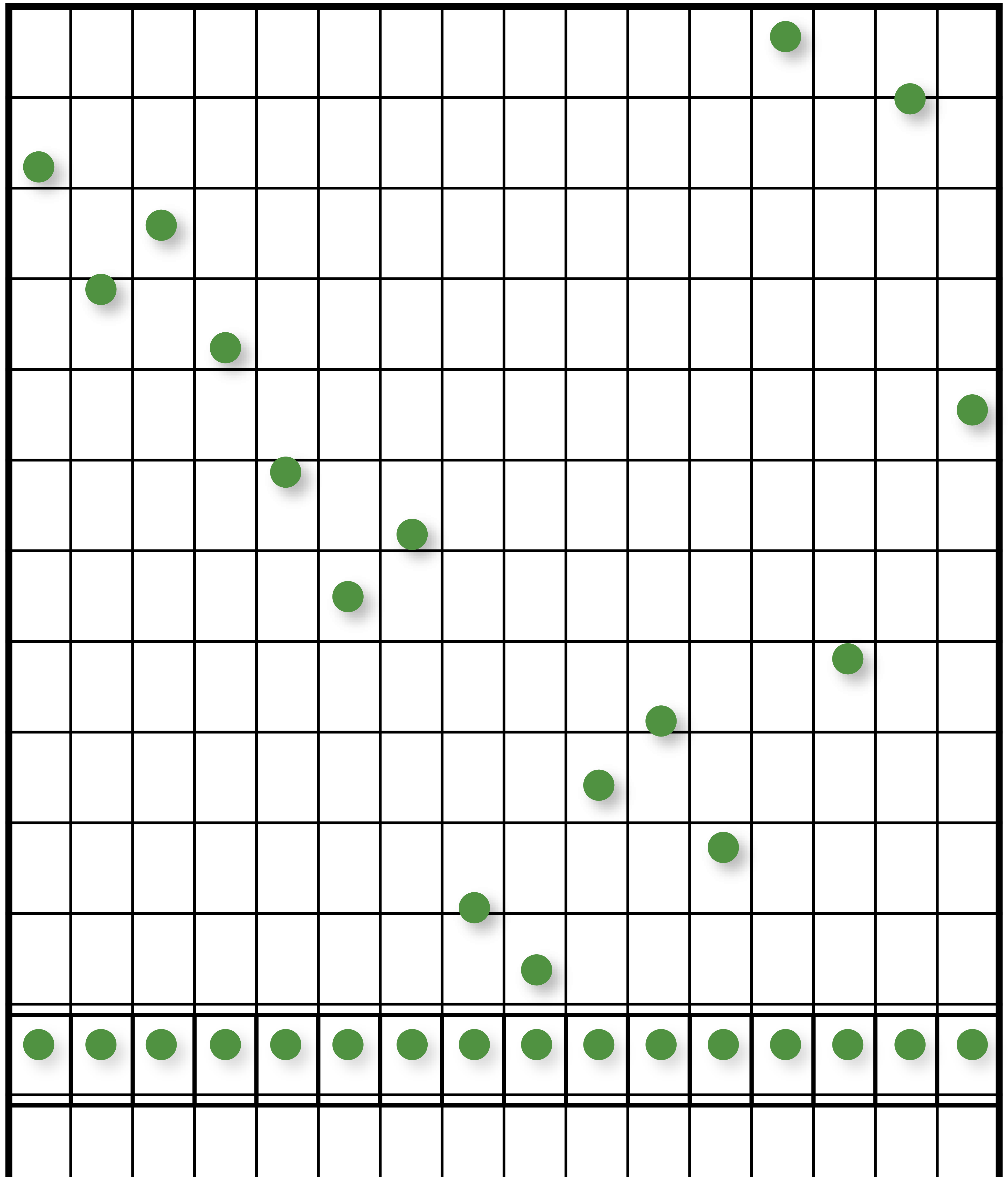
```
// initialize the diagonal
for (uint d = 0; d < numDimensions; d++)
    for (uint i = 0; i < numS; i++)
        samples(d,i) = (i + randf())/numS;

// shuffle each dimension independently
for (uint d = 0; d < numDimensions; d++)
    shuffle(samples(d,:));
```



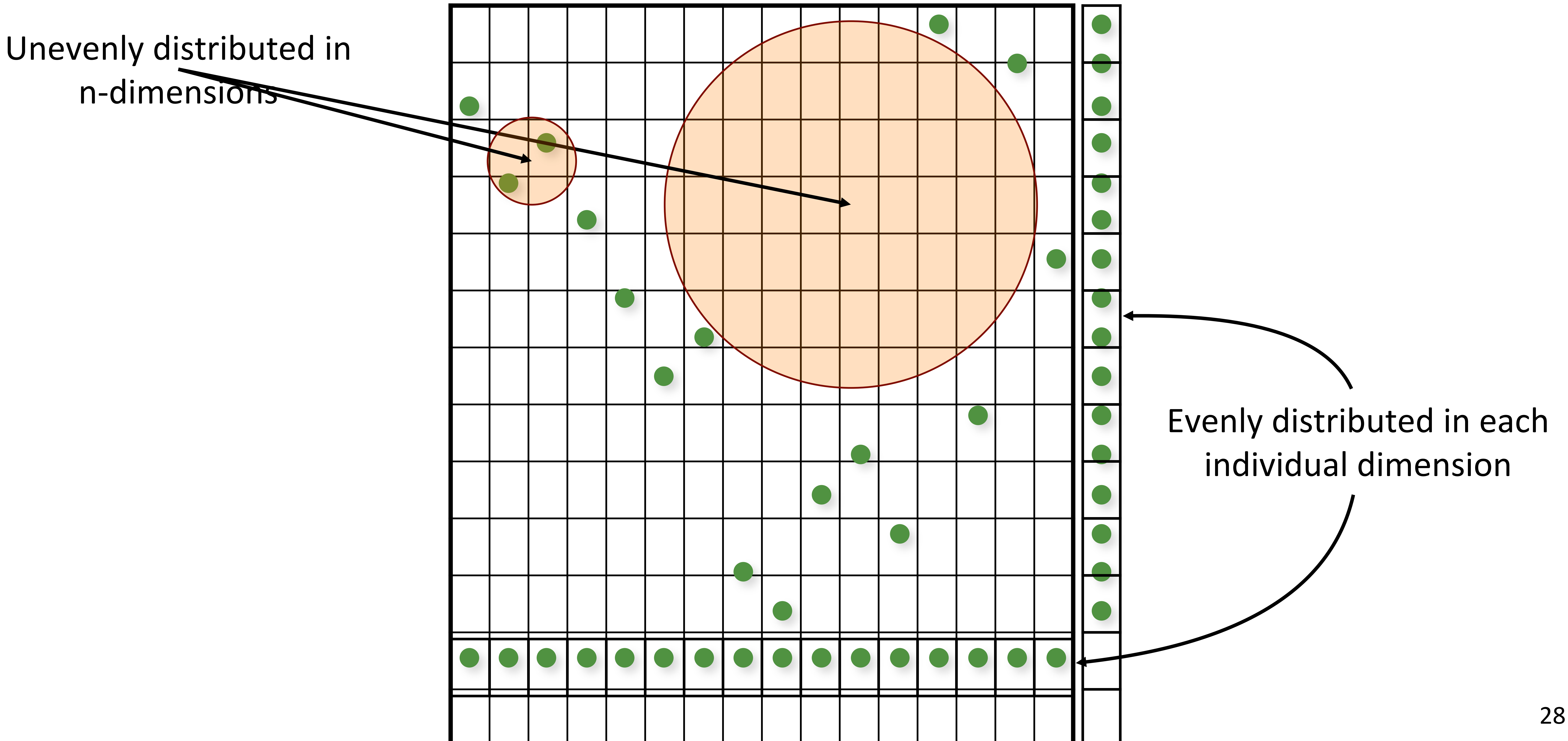
Latin Hypercube (N-Rooks) Sampling



Latin Hypercube (N-Rooks) Sampling



Latin Hypercube (N-Rooks) Sampling

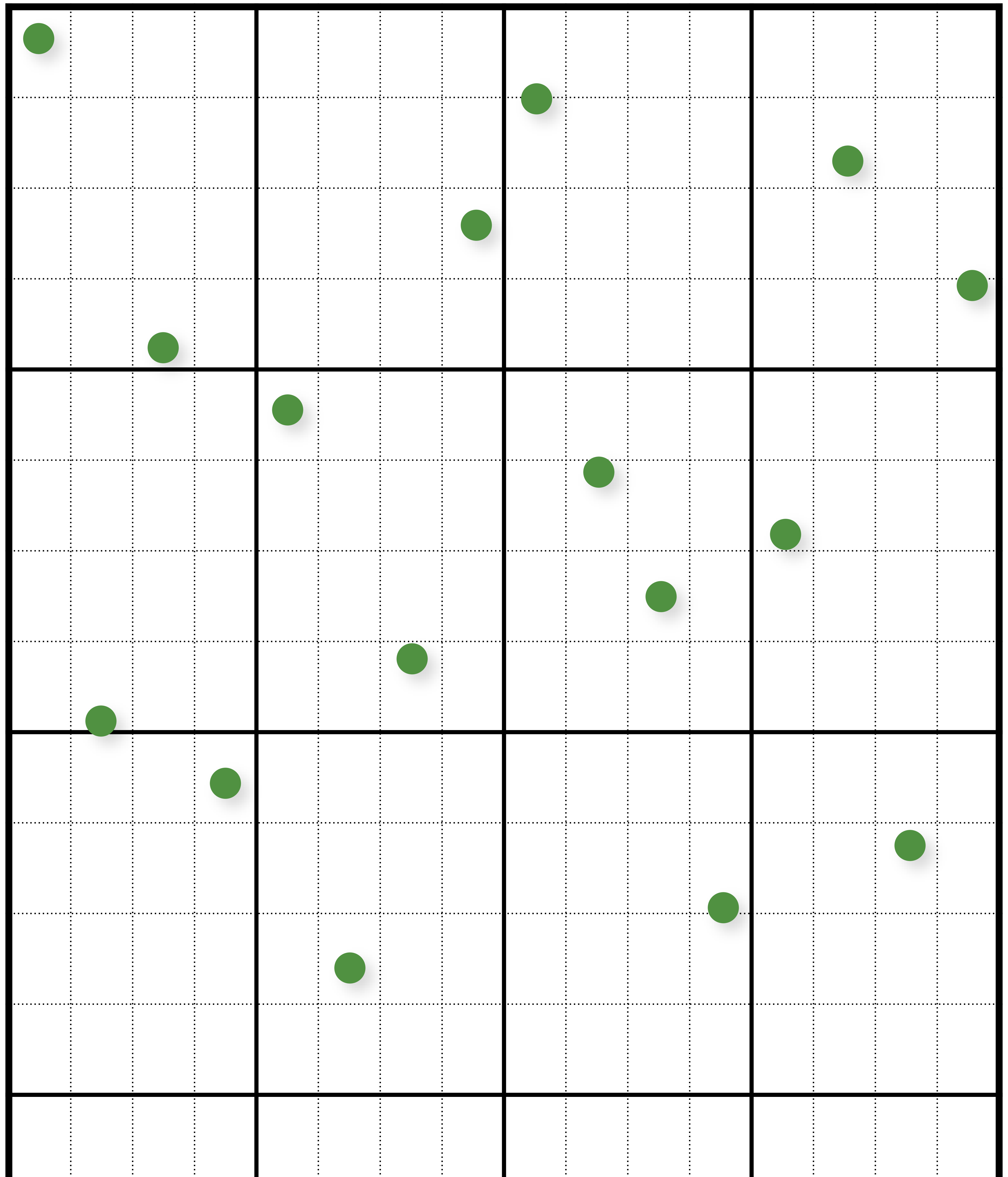


Multi-Jittered Sampling

Kenneth Chiu, Peter Shirley, and Changyaw Wang. “Multi-jittered sampling.” In *Graphics Gems IV*, pp. 370–374. Academic Press, May 1994.

– combine N-Rooks and Jittered stratification constraints

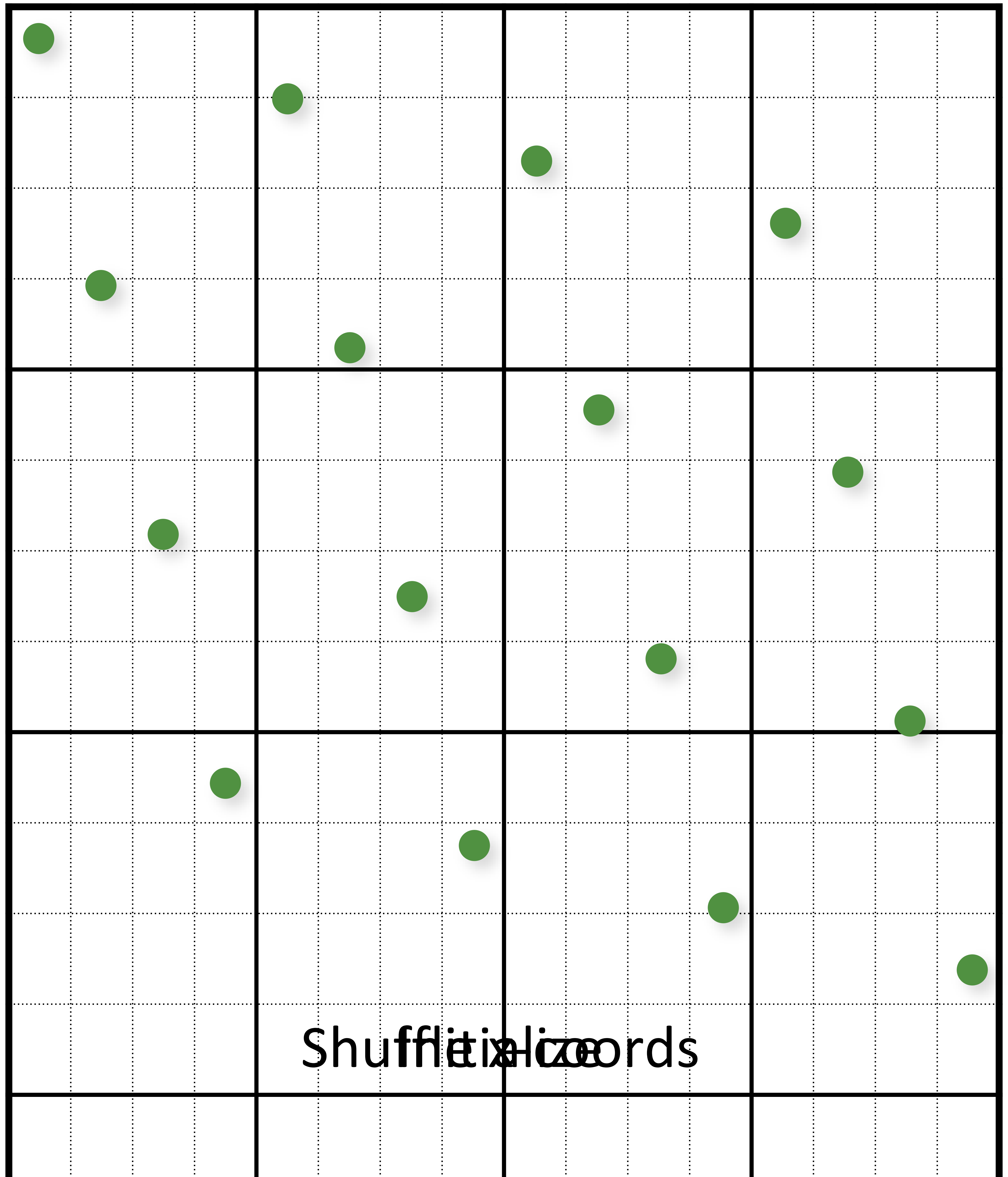
Multi-Jittered Sampling



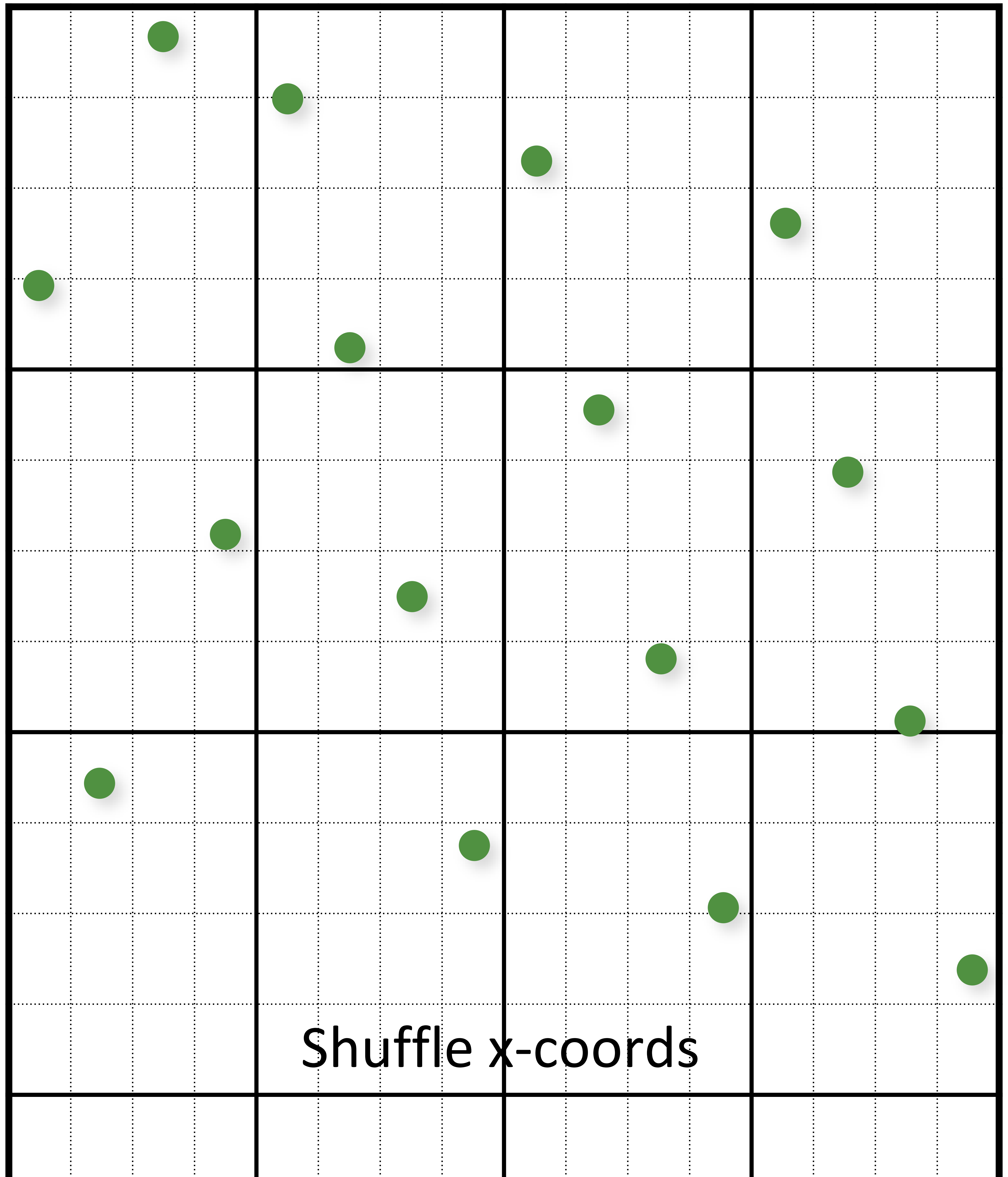
Multi-Jittered Sampling

```
// initialize
float cellSize = 1.0 / (resX*resY);
for (uint i = 0; i < resX; i++)
   for (uint j = 0; j < resY; j++)
   {
      samples(i,j).x = i/resX + (j+randf()) / (resX*resY);
      samples(i,j).y = j/resY + (i+randf()) / (resX*resY);
   }

// shuffle x coordinates within each column of cells
for (uint i = 0; i < resX; i++)
   for (uint j = resY-1; j >= 1; j--)
      swap(samples(i, j).x, samples(i, randi(0, j)).x);

// shuffle y coordinates within each row of cells
for (unsigned j = 0; j < resY; j++)
   for (unsigned i = resX-1; i >= 1; i--)
      swap(samples(i, j).y, samples(randi(0, i), j).y);
```

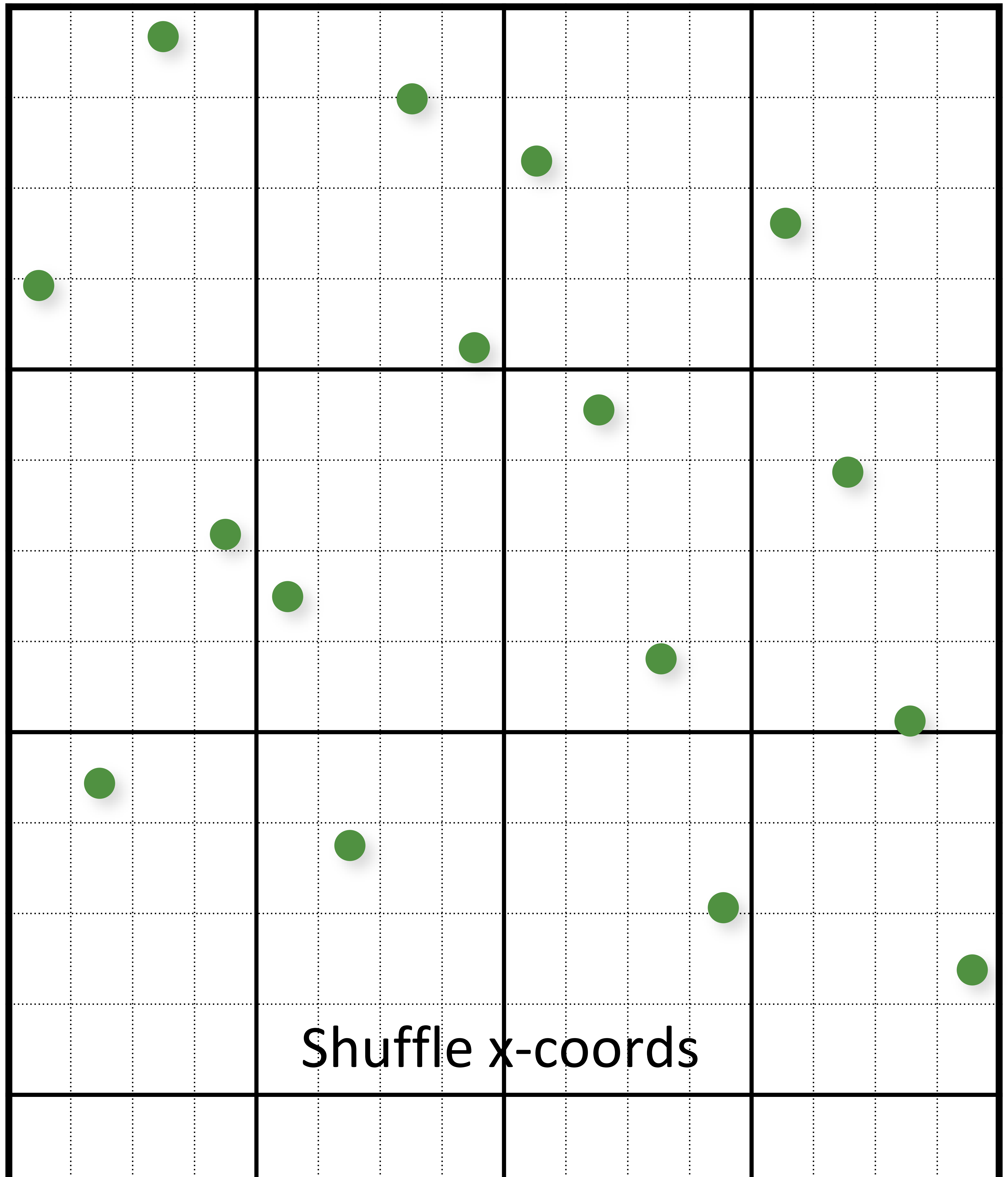
Multi-Jittered Sampling



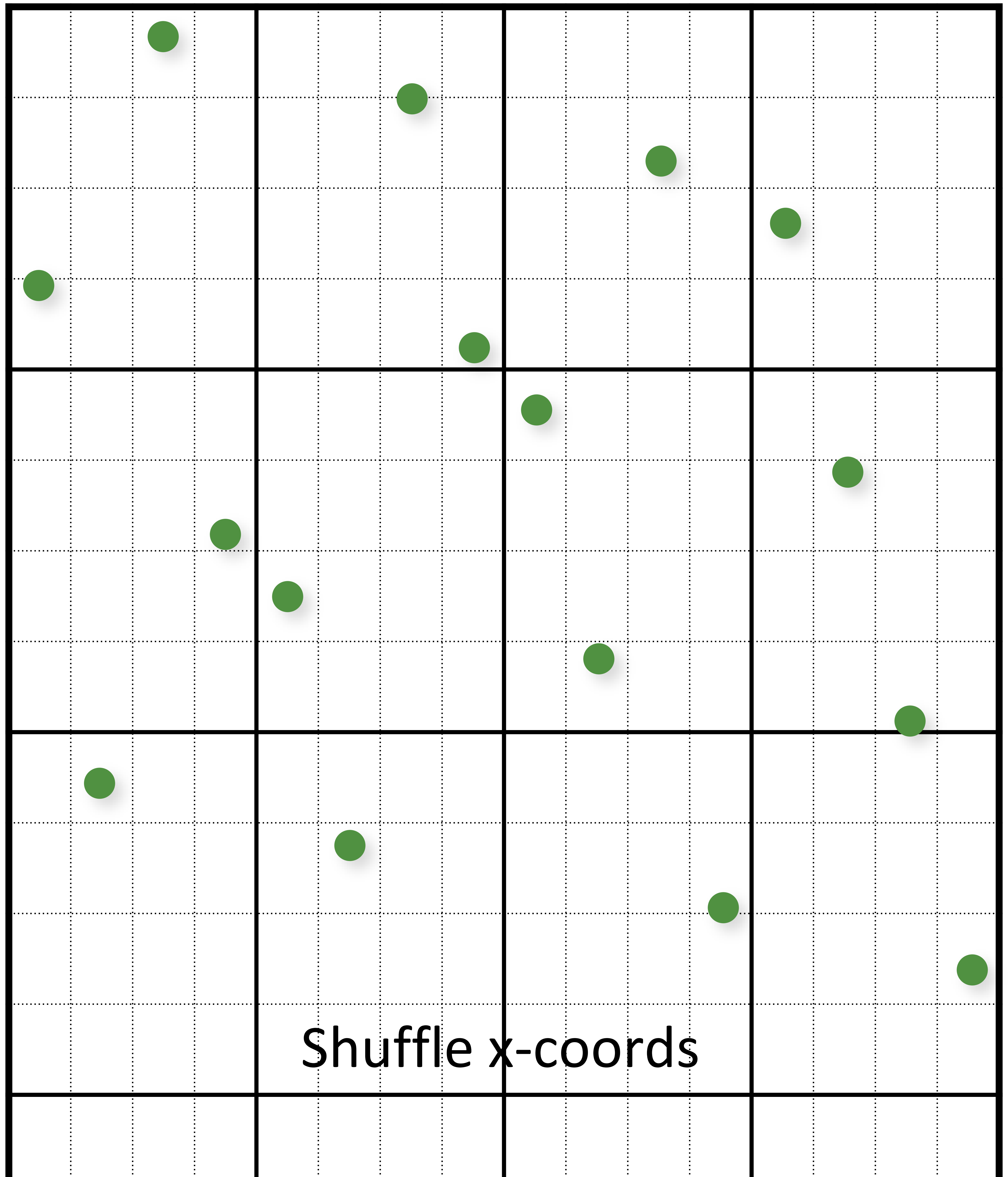
Multi-Jittered Sampling



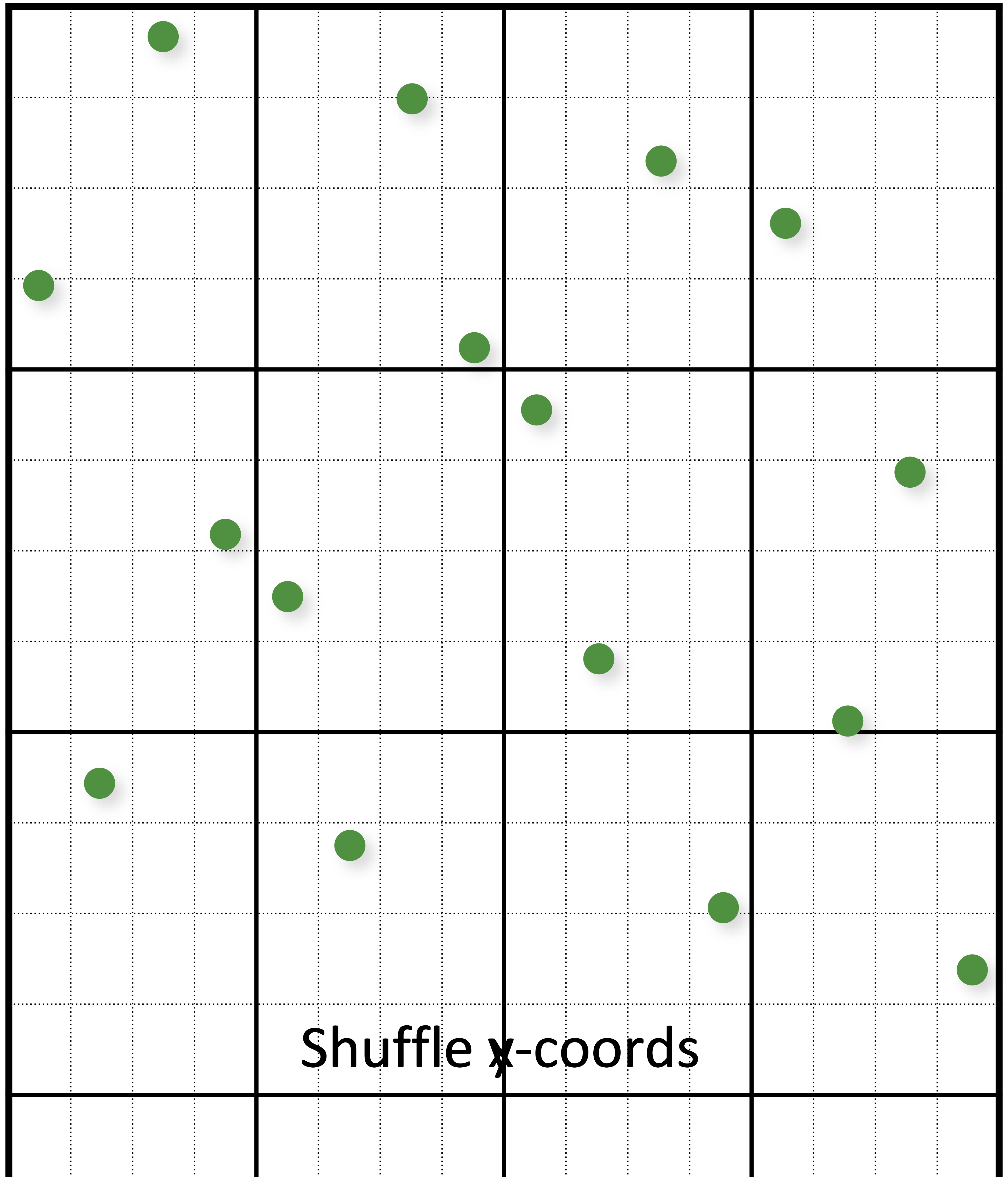
Multi-Jittered Sampling



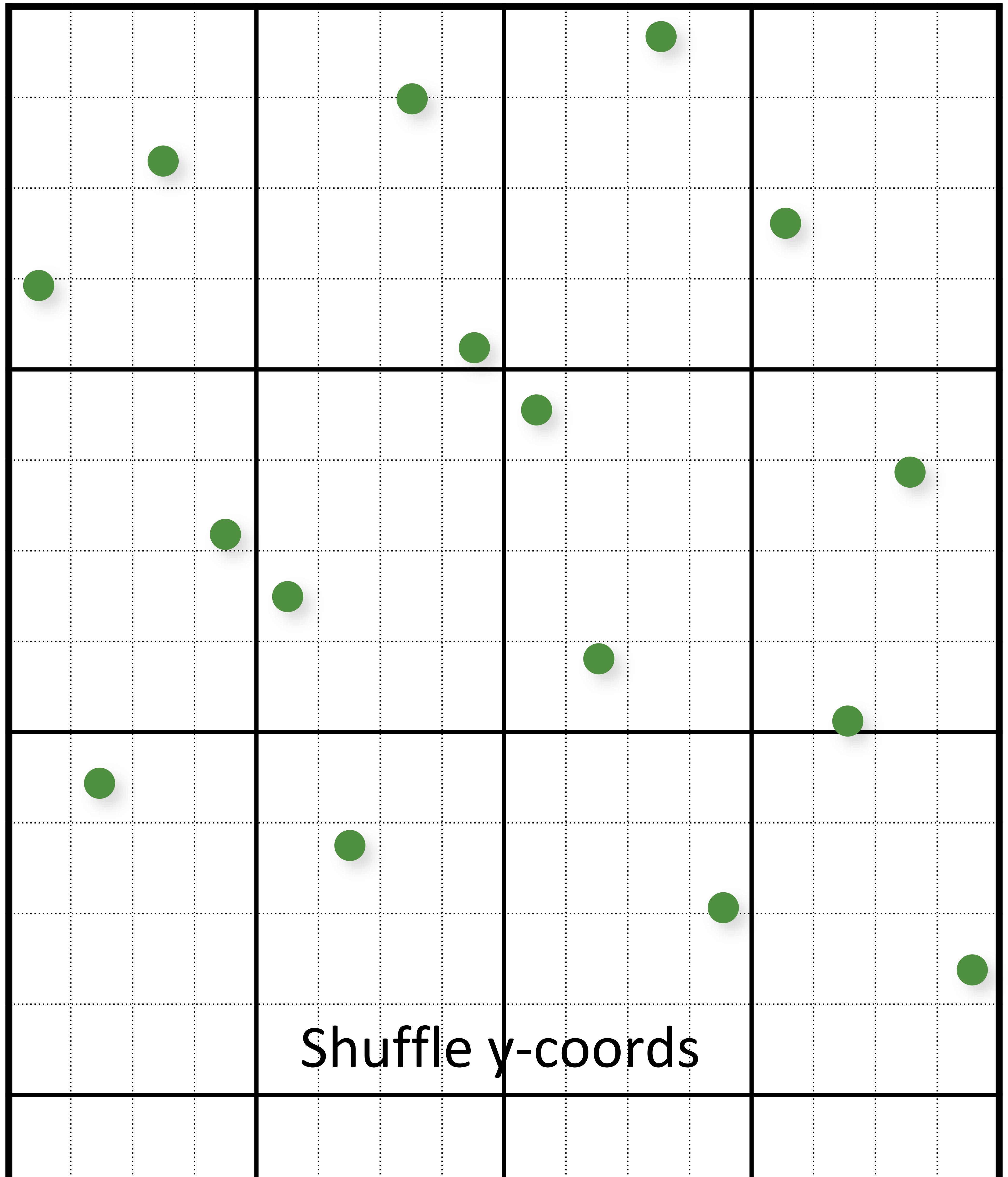
Multi-Jittered Sampling



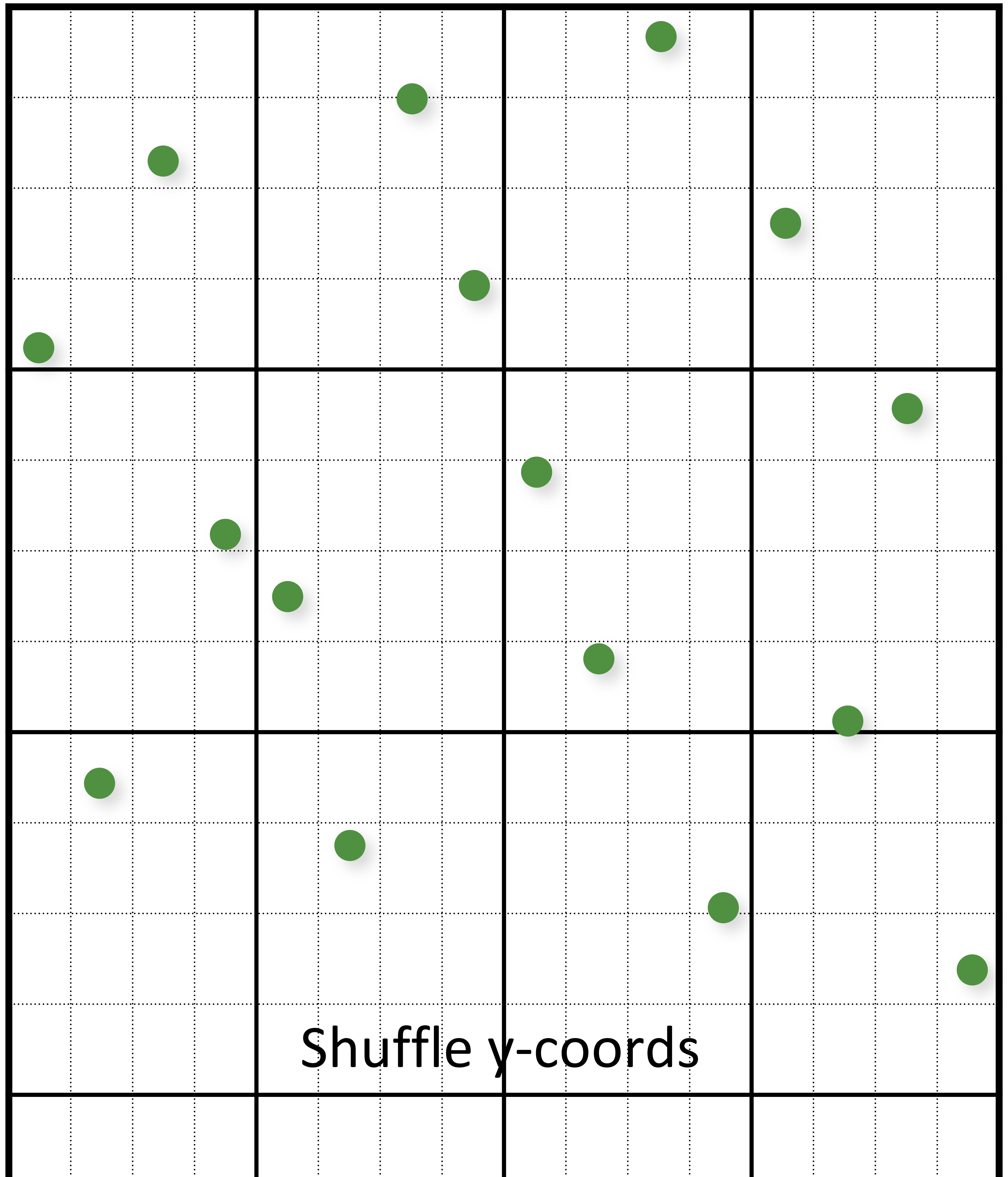
Multi-Jittered Sampling



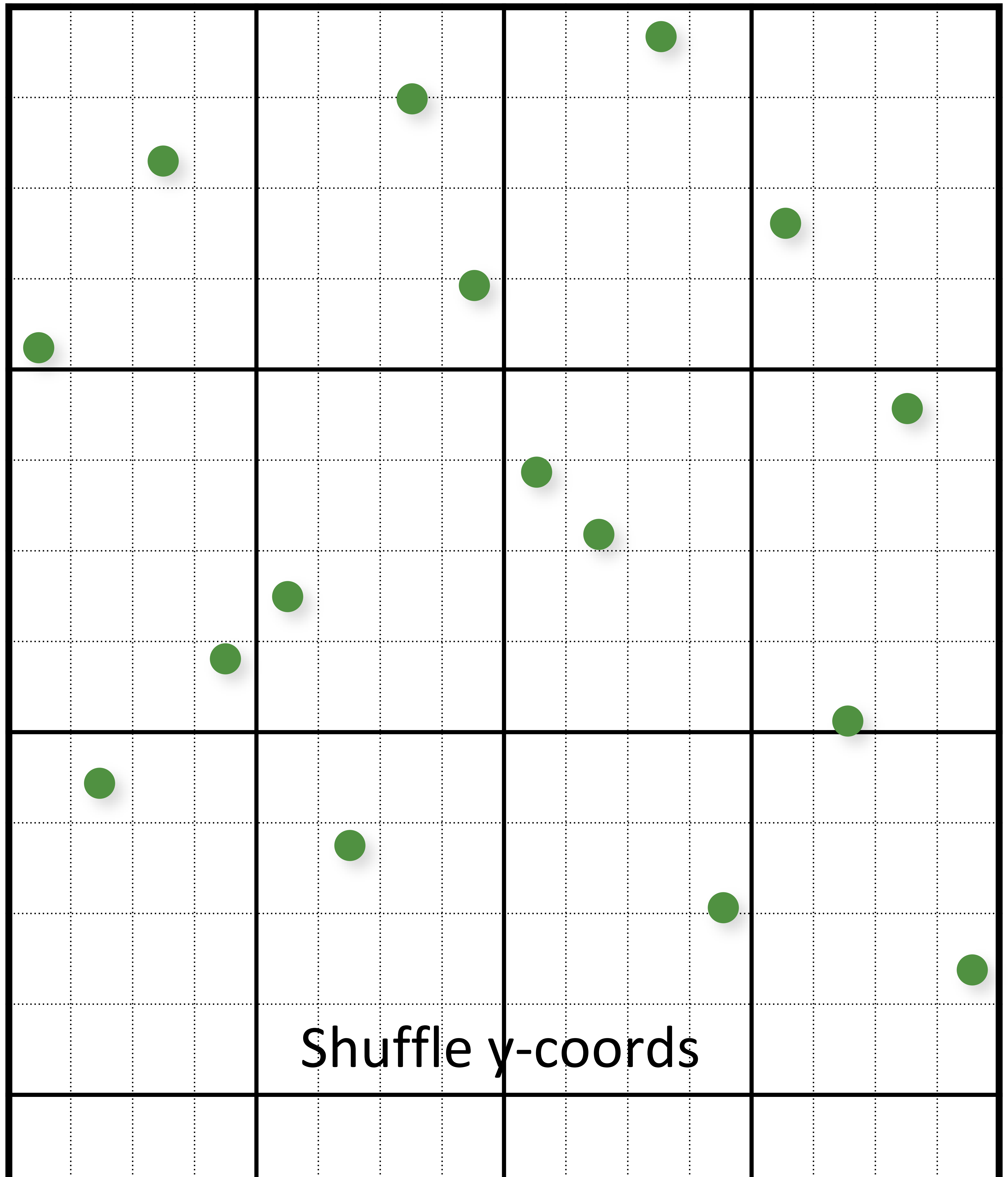
Multi-Jittered Sampling



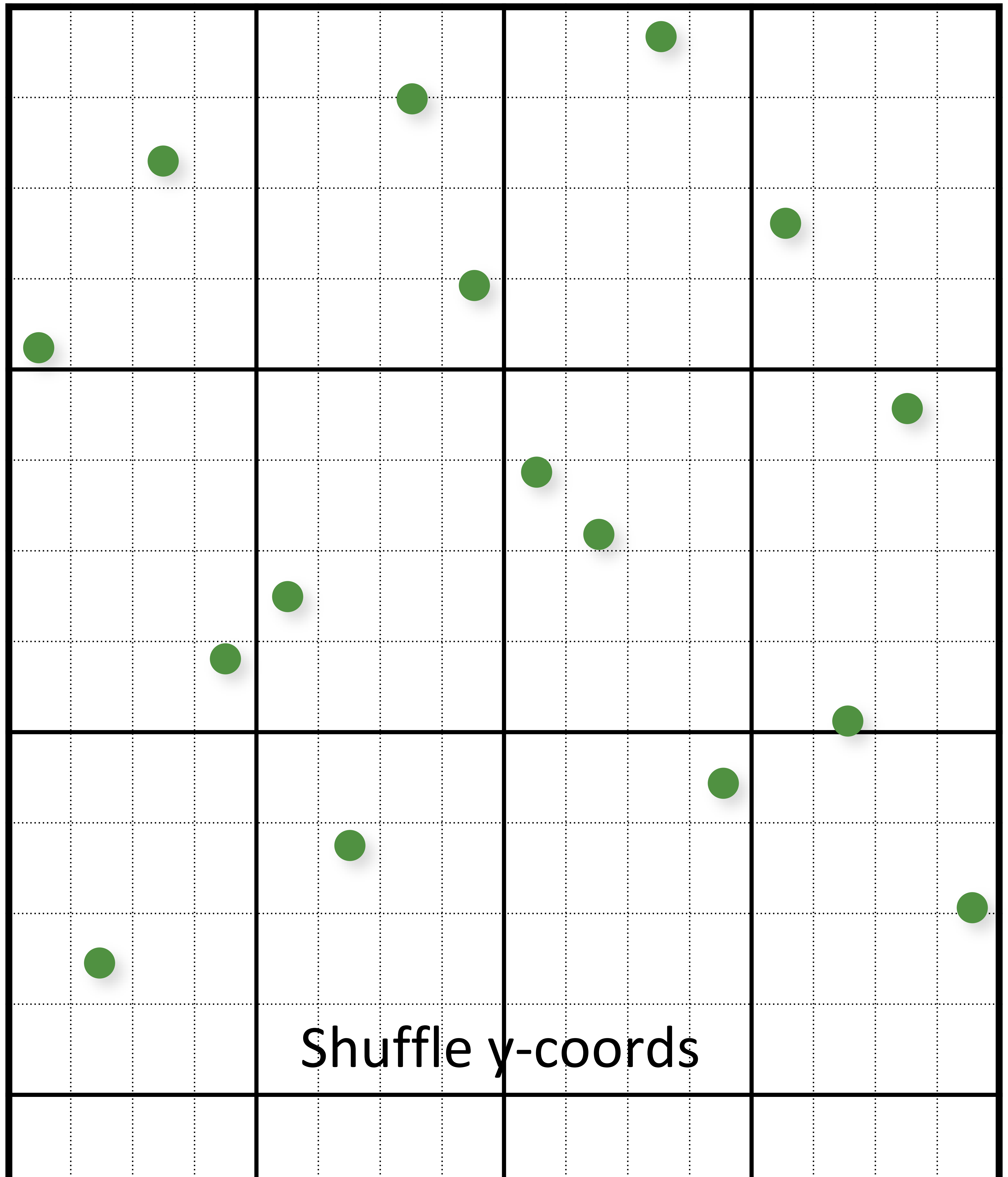
Multi-Jittered Sampling



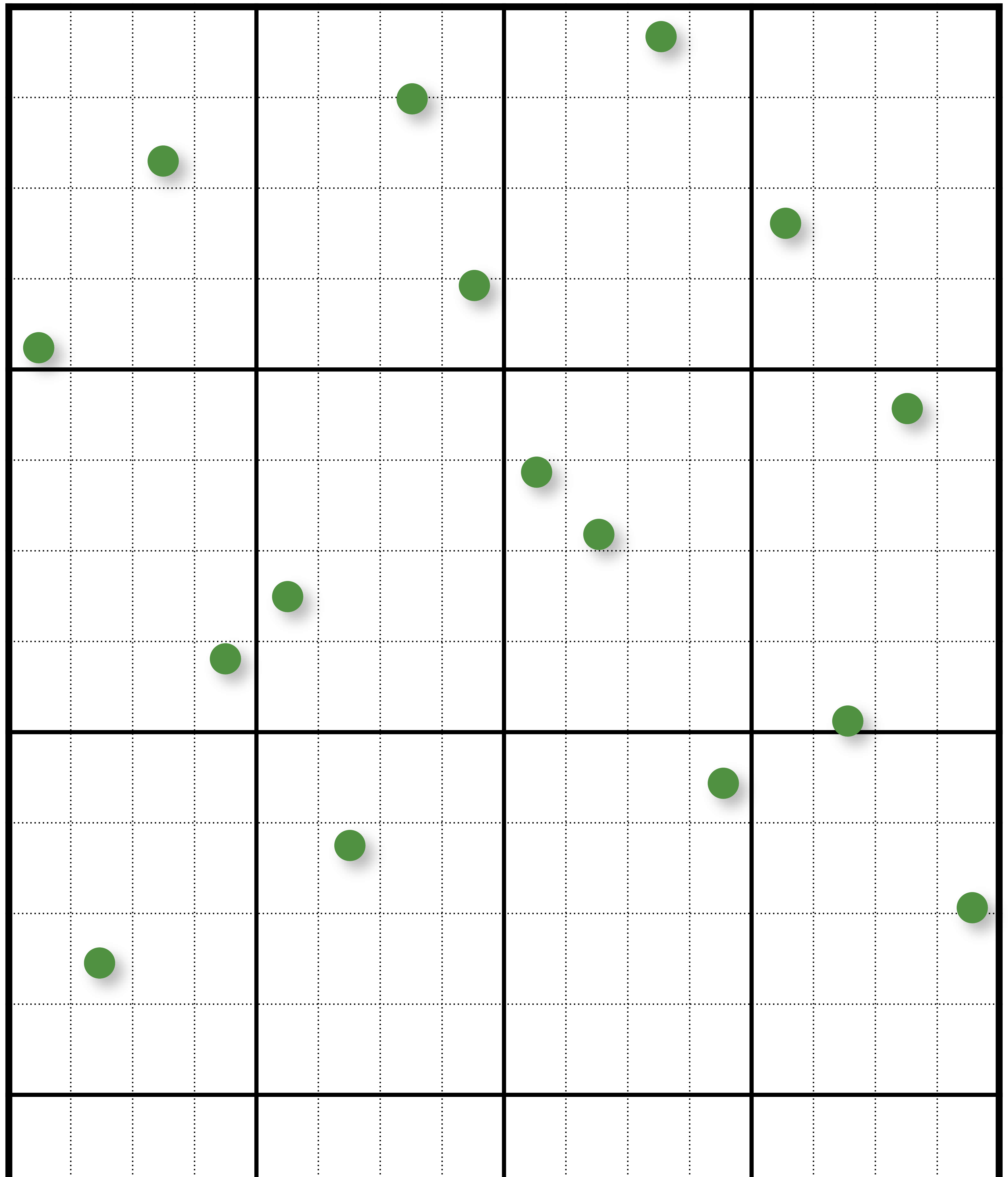
Multi-Jittered Sampling



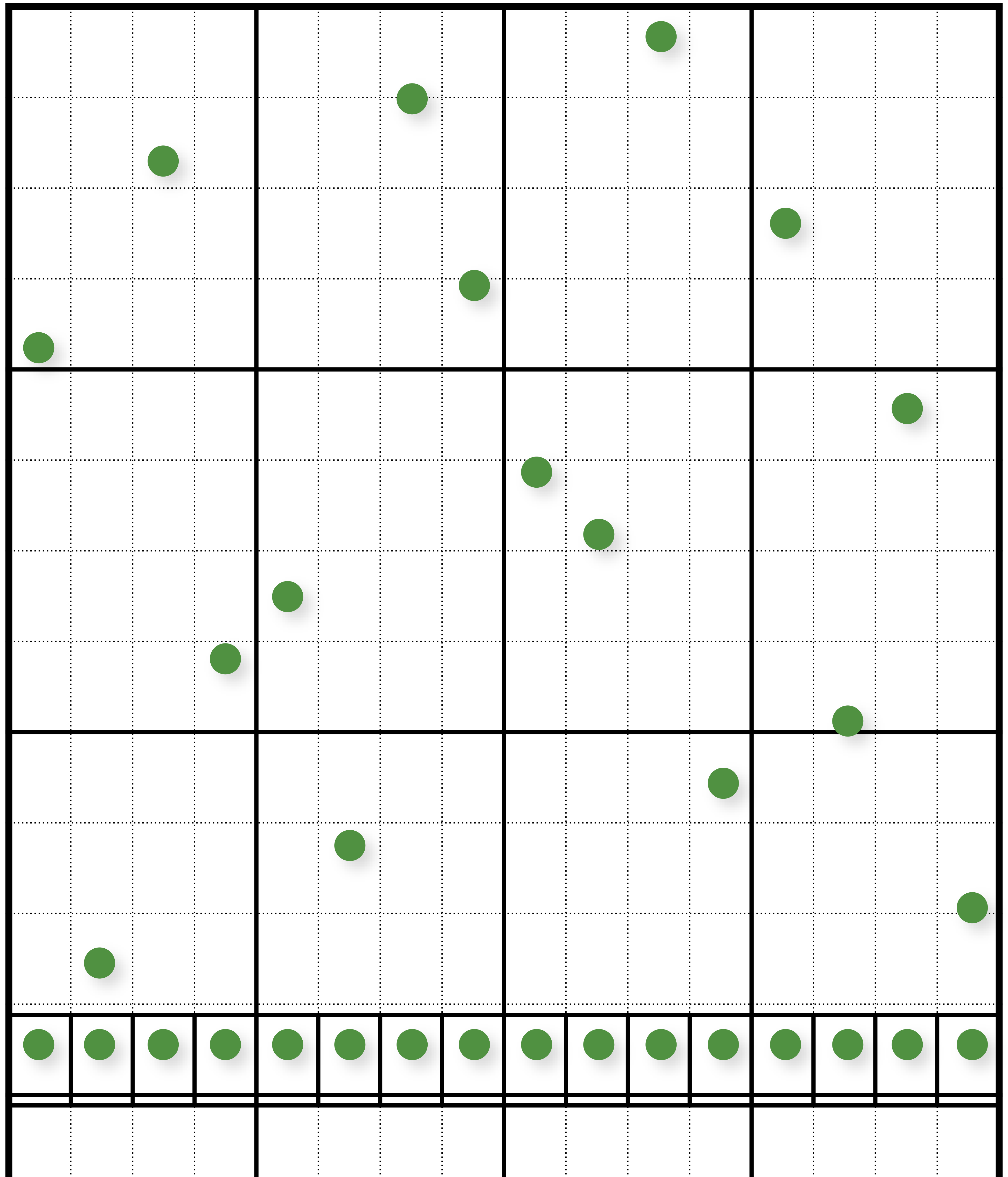
Multi-Jittered Sampling



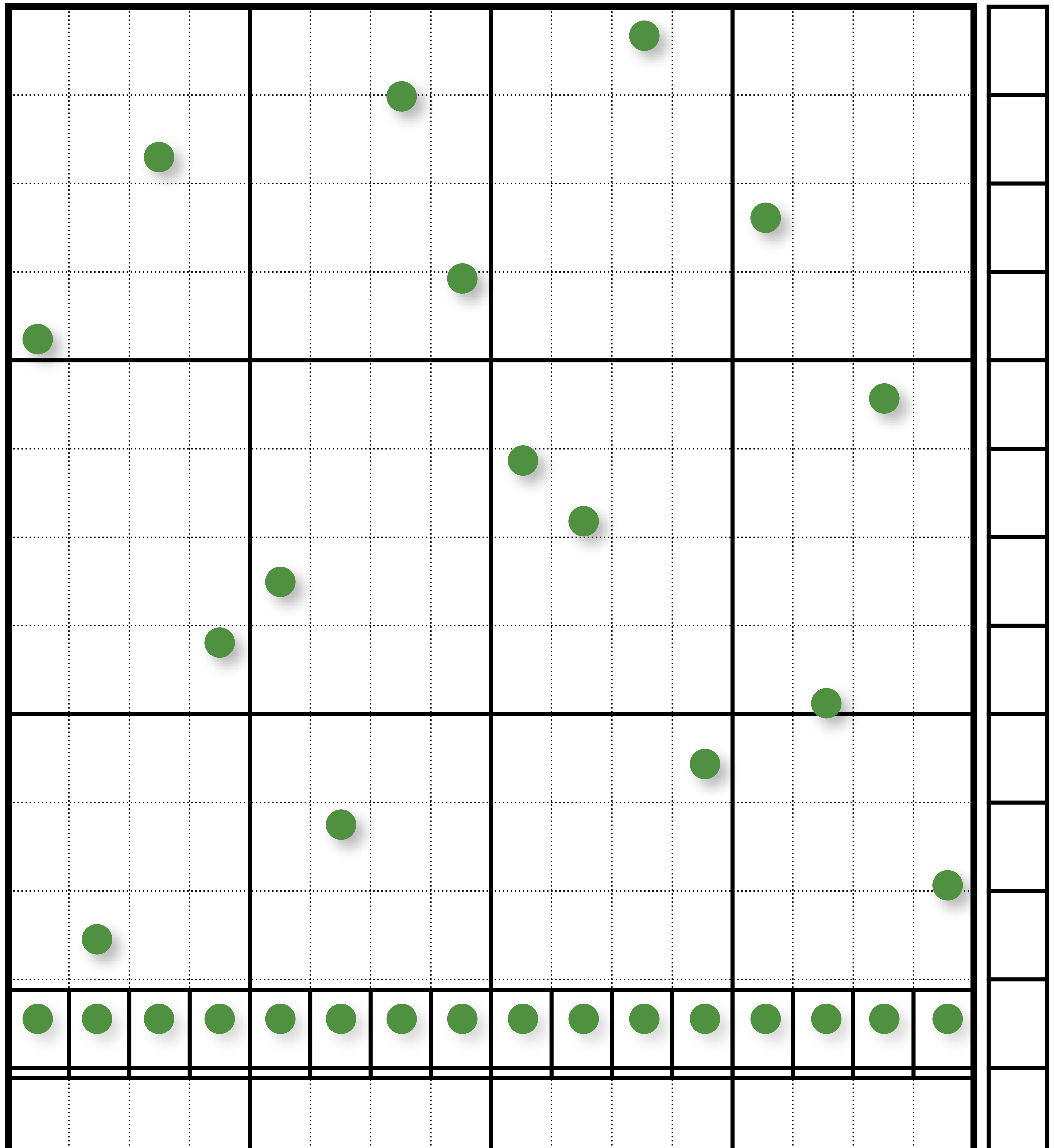
Multi-Jittered Sampling (Projections)



Multi-Jittered Sampling (Projections)

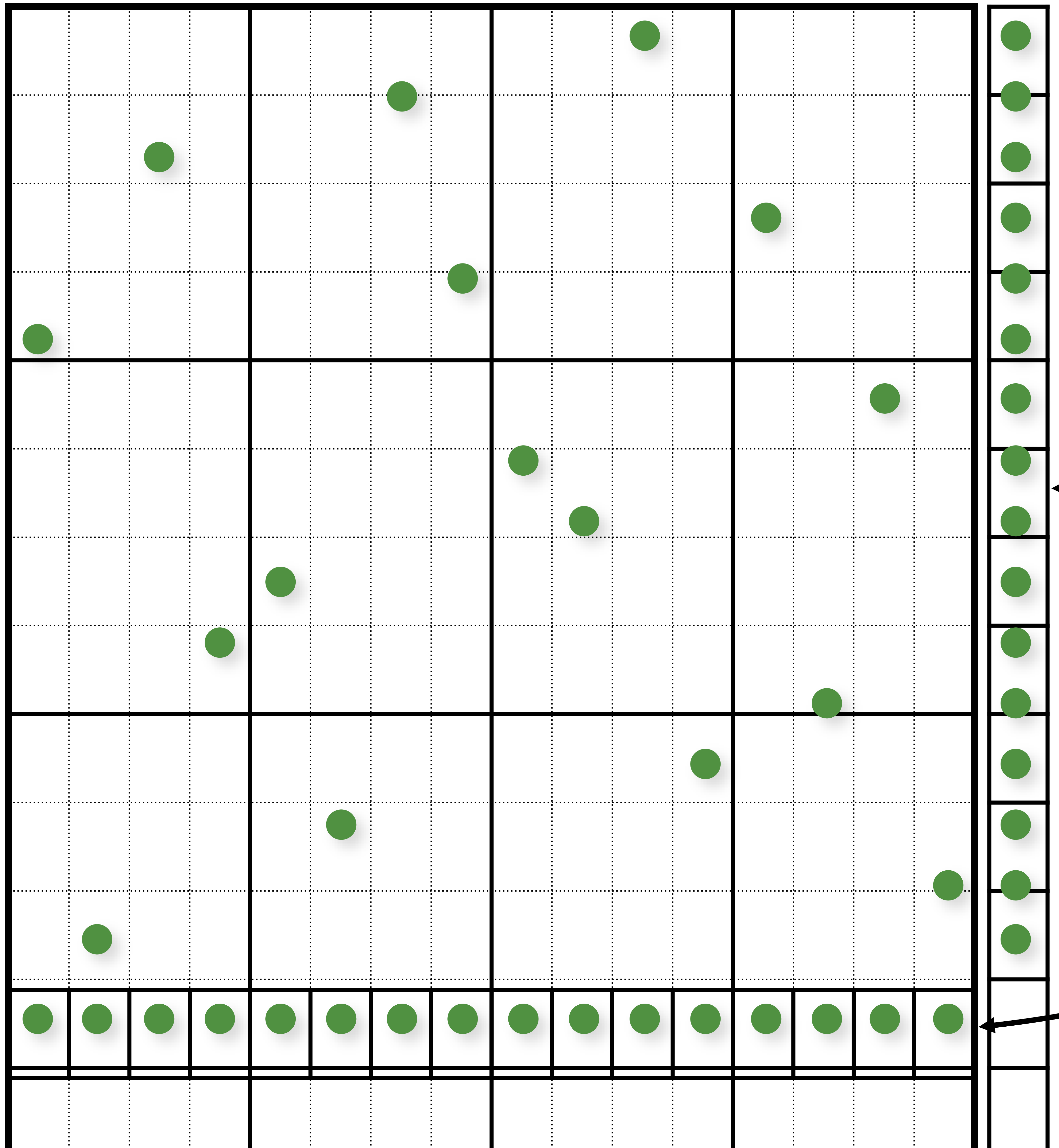


Multi-Jittered Sampling (Projections)



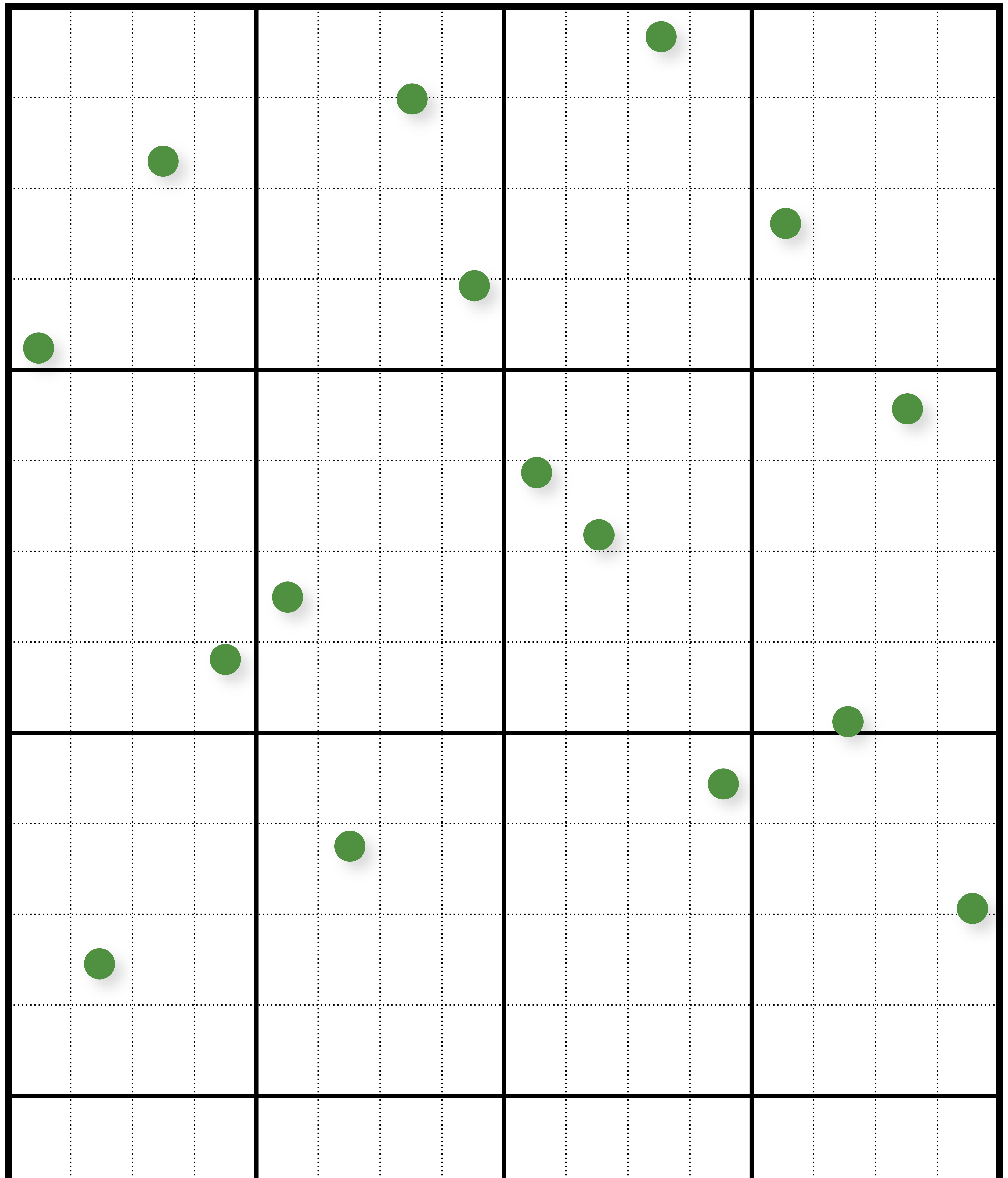
Multi-Jittered Sampling (Projections)

Evenly distributed in 2D!



Evenly distributed in each individual dimension

Multi-Jittered Sampling (Sudoku)



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
9	10	11	12	1	2	3	4	13	14	15	16	5	6	7	8
5	6	7	8	13	14	15	16	1	2	3	4	9	10	11	12
13	14	15	16	9	10	11	12	5	6	7	8	1	2	3	4
3	1	4	2	7	5	8	6	11	9	14	10	15	12	16	13
11	9	14	10	3	1	4	2	15	12	16	13	7	5	8	6
7	5	8	6	15	12	16	13	3	1	4	2	11	9	14	10
15	12	16	13	11	9	14	10	7	5	8	6	3	1	4	2
2	4	1	3	6	8	5	7	10	15	9	11	12	16	13	14
10	15	9	11	2	4	1	3	12	16	13	14	6	8	5	7
6	8	5	7	12	16	13	14	2	4	1	3	10	15	9	11
12	16	13	14	10	15	9	11	6	8	5	7	2	4	1	3
4	3	2	1	8	7	6	5	14	11	10	9	16	13	12	15
14	11	10	9	4	3	2	1	16	13	12	15	8	7	6	5
8	7	6	5	16	13	12	15	4	3	2	1	14	11	10	9
16	13	12	15	14	11	10	9	8	7	6	5	4	3	2	1

[Boulos et al. 2006]

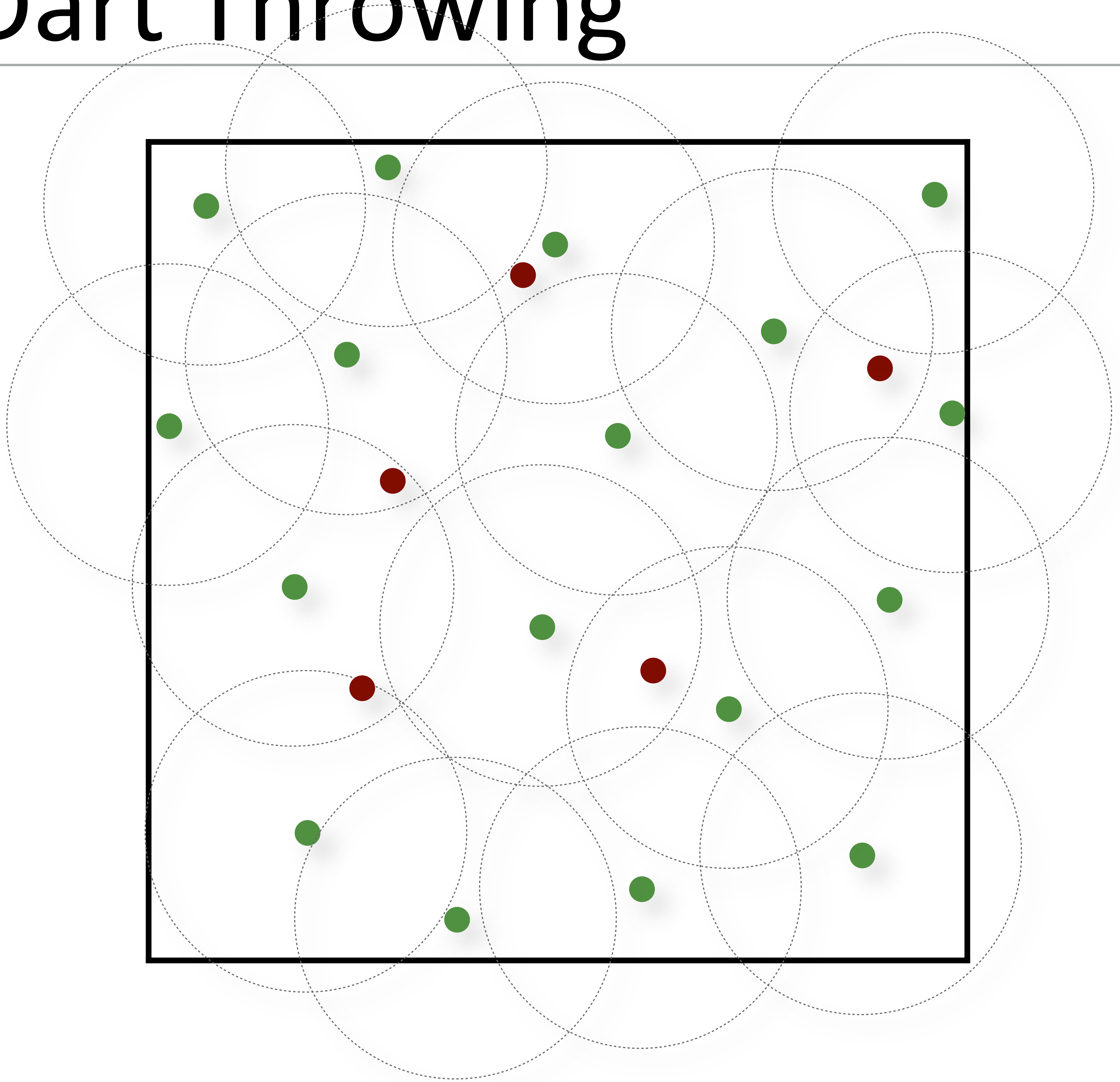
Poisson-Disk/Blue-Noise Sampling

Enforce a minimum distance between points

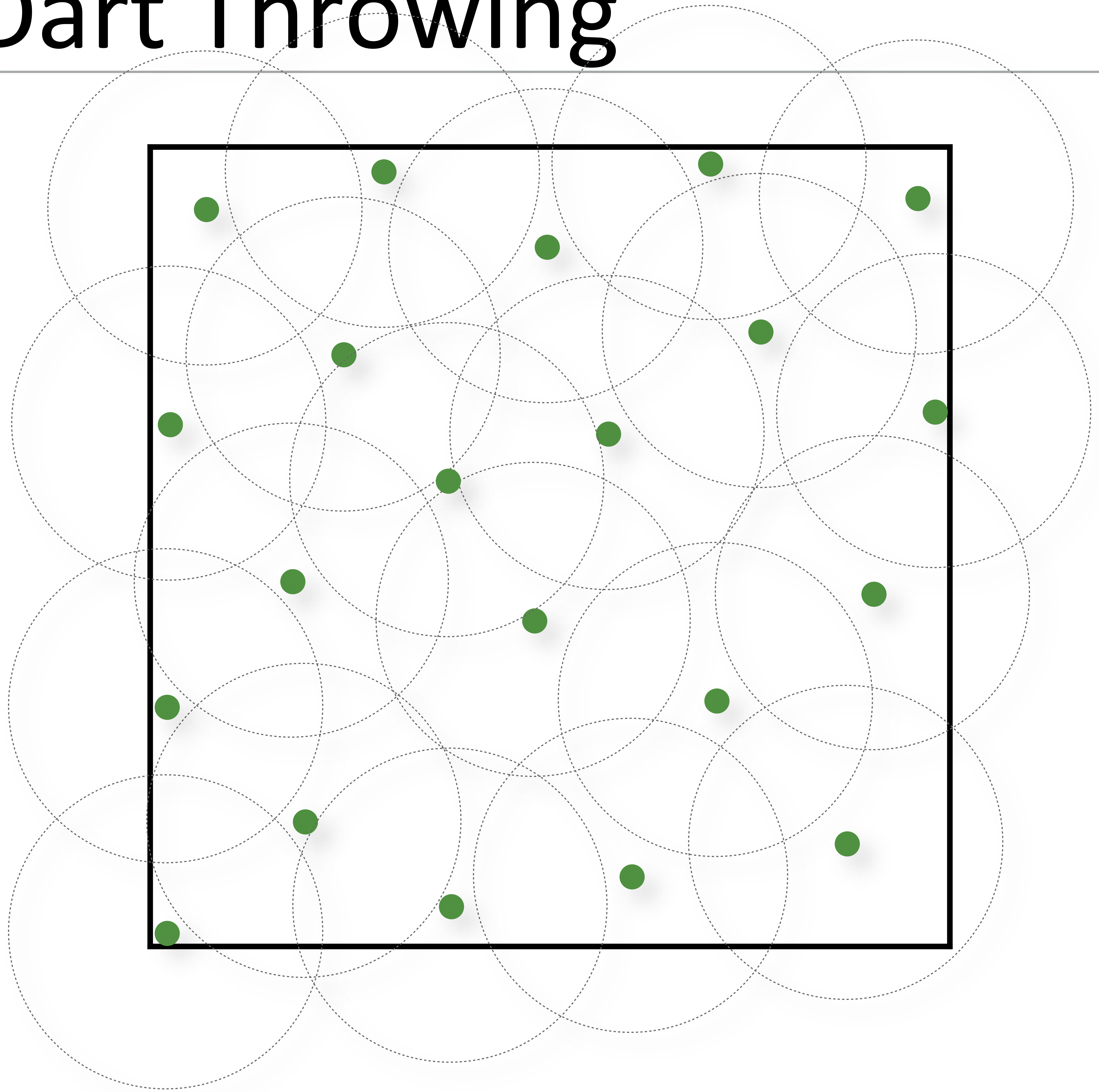
Poisson-Disk Sampling:

- Mark A. Z. Dippé and Erling Henry Wold. “Antialiasing through stochastic sampling.” *ACM SIGGRAPH,* 1985.
- Robert L. Cook. “Stochastic sampling in computer graphics.” *ACM Transactions on Graphics,* 1986.
- Ares Lagae and Philip Dutré. “A comparison of methods for generating Poisson disk distributions.” *Computer Graphics Forum*, 2008.

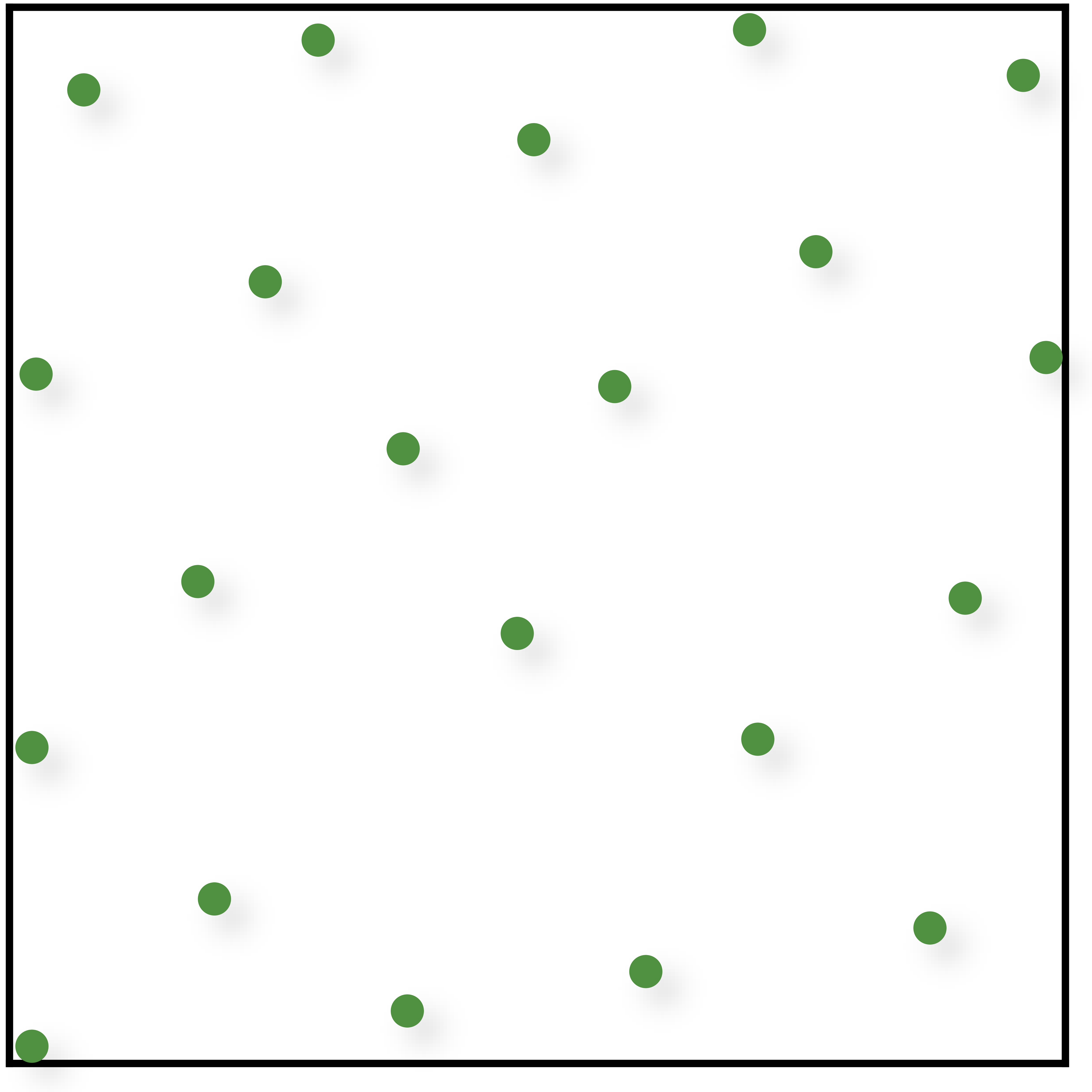
Random Dart Throwing



Random Dart Throwing



Random Dart Throwing



Stratified Sampling

“Best Candidate” Dart Throwing

Blue-Noise Sampling (Relaxation-based)

1. Initialize sample positions (e.g. random)
2. Use an iterative relaxation to move samples away from each other.

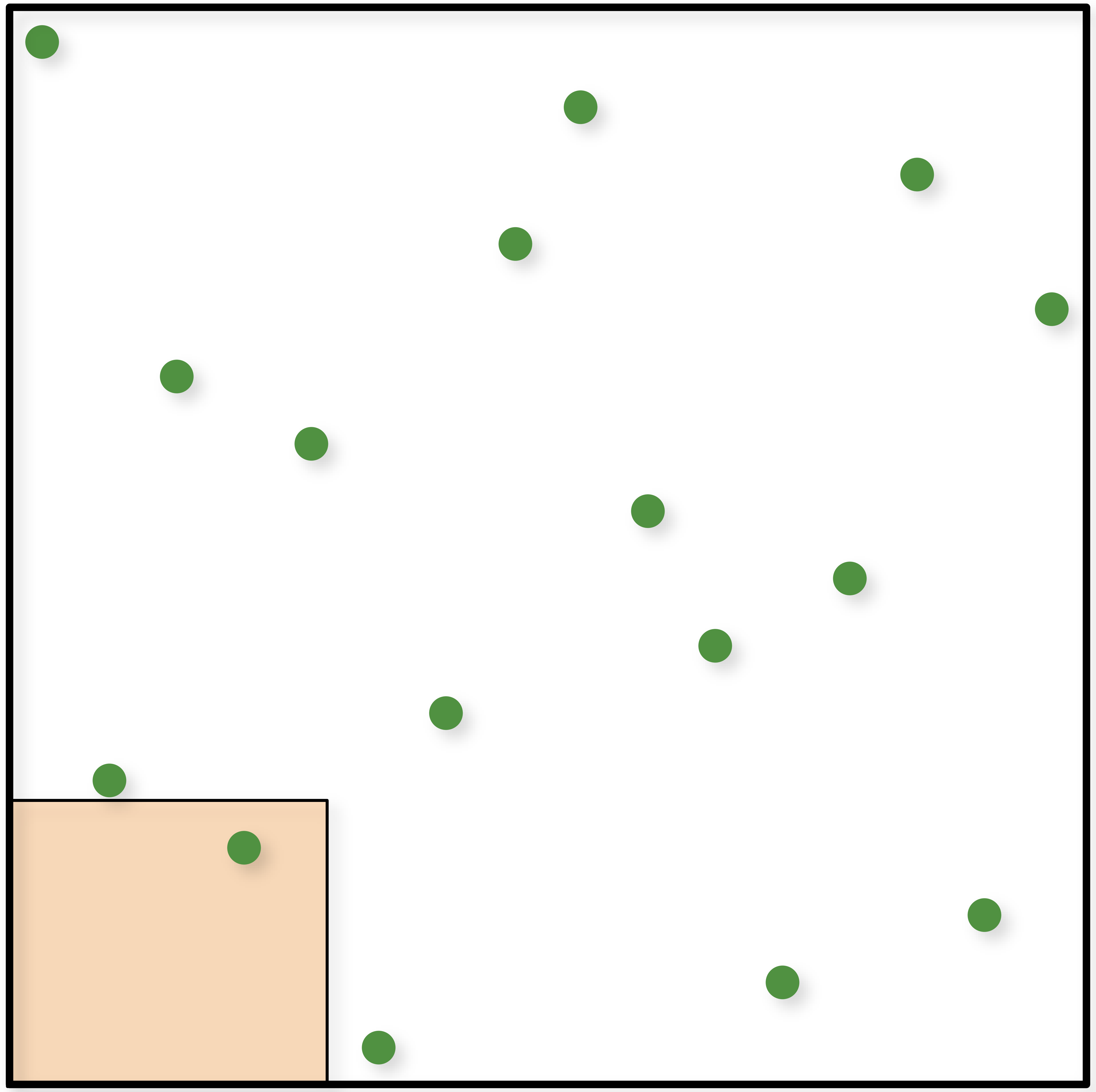
Discrepancy

Previous stratified approaches try to minimize "clumping"

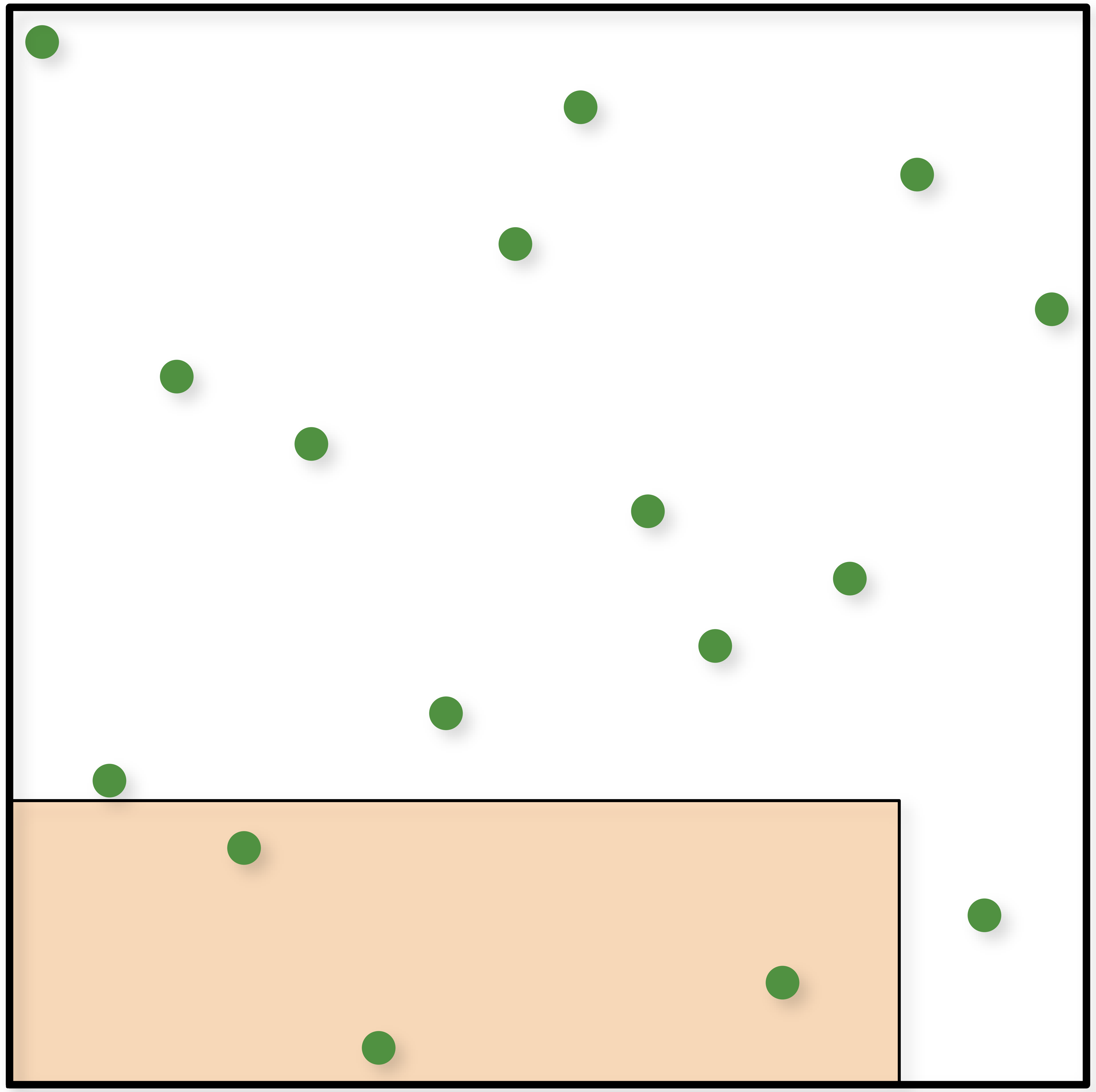
"Discrepancy" is another possible formal definition of clumping: $D^*(x_1,\ldots,x_n)$

- for every possible subregion compute the maximum absolute difference between:
    - fraction of points in the subregion
    - volume of containing subregion

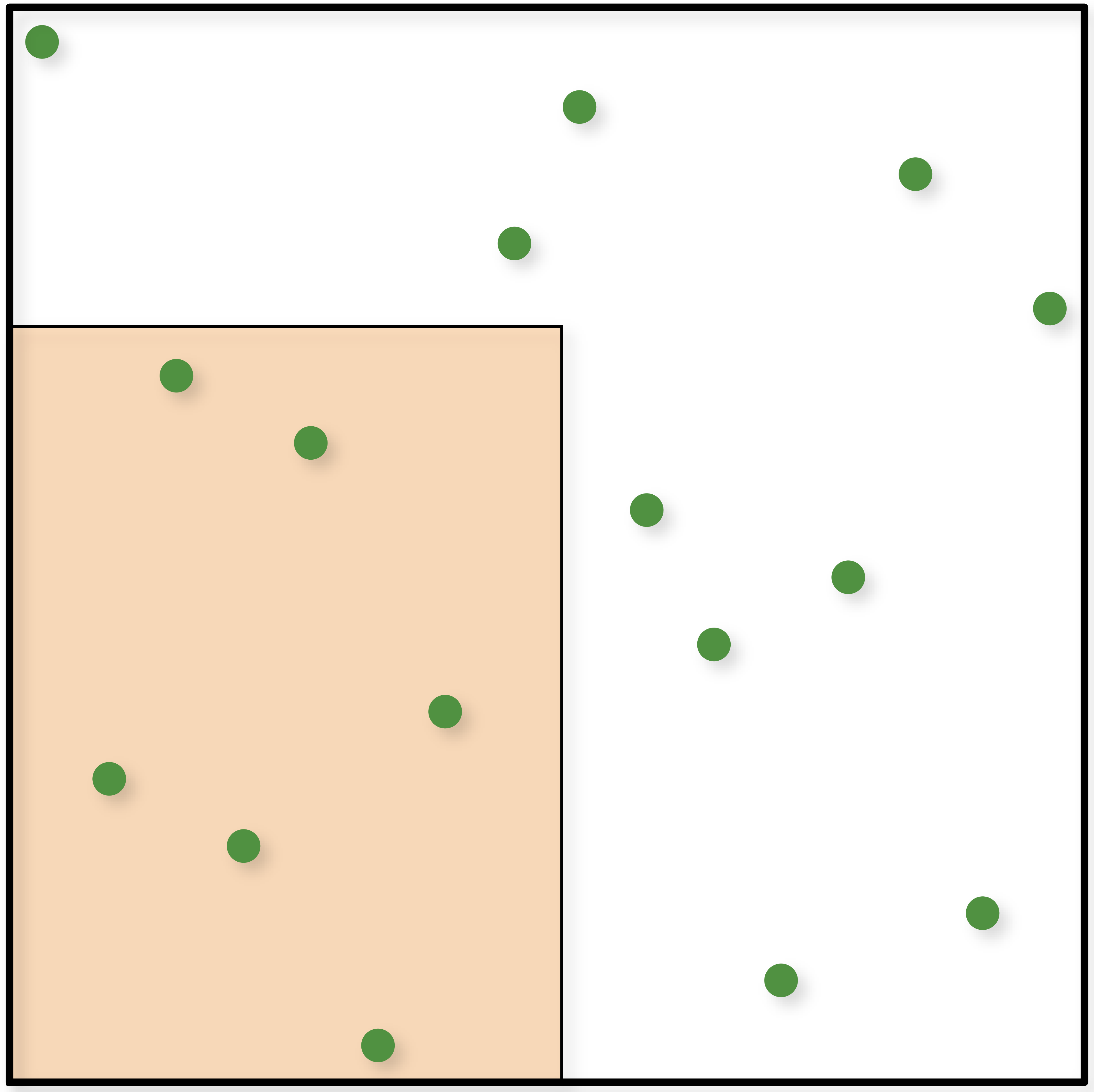
Discrepancy



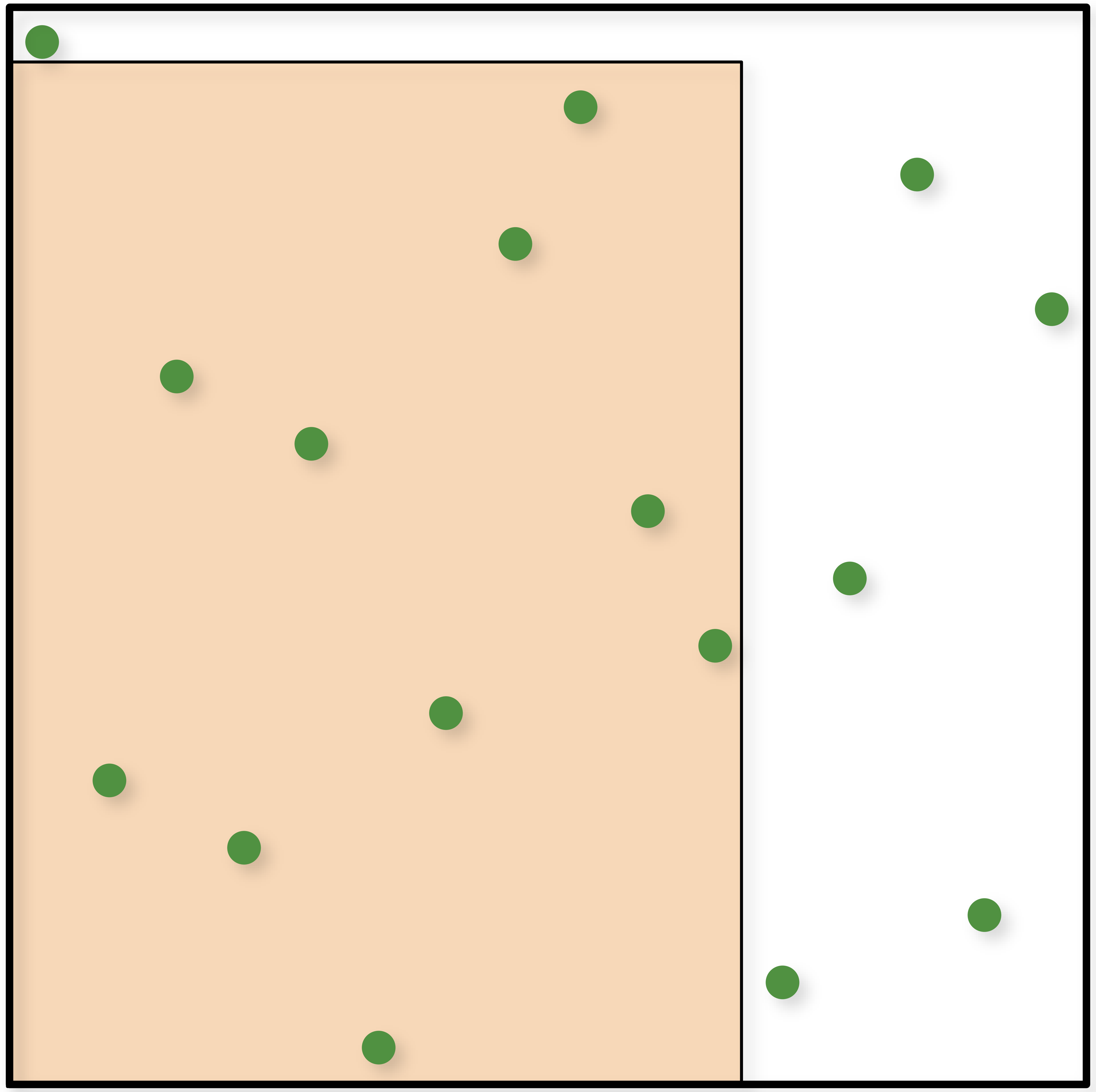
Discrepancy



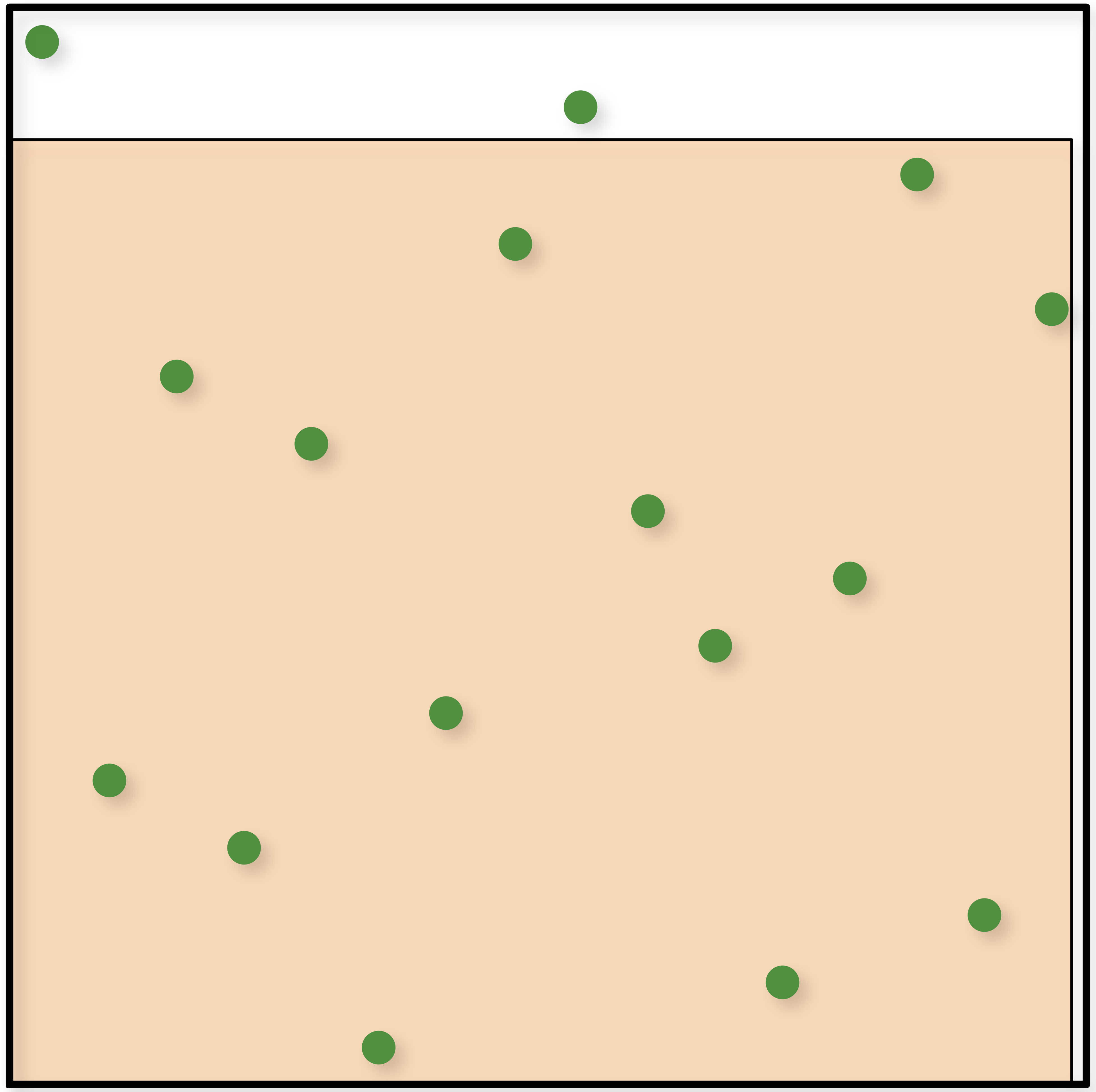
Discrepancy



Discrepancy



Discrepancy



Koksma-Hlawka inequality

$$\left| \frac{1}{n} \sum_{i=1}^{n} f(x_i) - \int f(u)\, \mathrm{d}u \right| \leq V(f) D^*(x_1, \ldots, x_n)$$

Low-Discrepancy Sampling

Deterministic sets of points specially crafted to be evenly distributed (have low discrepancy).

Entire field of study called Quasi-Monte Carlo (QMC)

The Radical Inverse

A positive integer value n can be expressed in a base b with a sequence of digits $d_m ... d_2 d_1$

The radical inverse function Φ_b in base b converts a nonnegative integer n to a floating-point value in [0, 1) by reflecting these digits about the decimal point:

$$\Phi_b(n) = 0.d_1 d_2 \ldots d_m$$

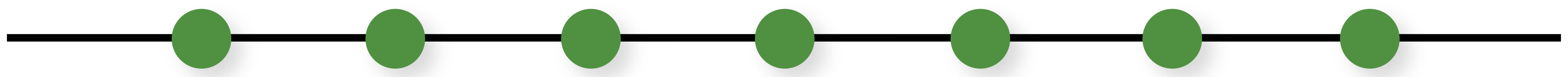
Subsequent points “fall into biggest holes”

The Van der Corput Sequence

Radical Inverse Φ_b in base 2

Subsequent points “fall into biggest holes”

k	Base 2	Φ_b
1	1	.1 = 1/2
2	10	.01 = 1/4
3	11	.11 = 3/4
4	100	.001 = 1/8
5	101	.101 = 5/8
6	110	.011 = 3/8
7	111	.111 = 7/8
...		



The Radical Inverse

```
float radicalInverse(int n, int base, float inv)
{
  float v = 0.0f;
  for (float p = inv; n != 0; p *= inv, n /= base)
    v += (n % base) * p;
  return v;
}

float radicalInverse(int n, int base)
{
  return radicalInverse(n, base, 1.0f / base);
}
```

More efficient version available for base 2

The Radical Inverse (Base 2)

```
float vanDerCorputRIU(uint n)
{
  n = (n << 16) | (n >> 16);
  n = ((n & 0x00ff00ff) << 8) | ((n & 0xff00ff00) >>
8);
  n = ((n & 0x0f0f0f0f) << 4) | ((n & 0xf0f0f0f0) >>
4);
  n = ((n & 0x33333333) << 2) | ((n & 0xcccccccc) >>
2);
  n = ((n & 0x55555555) << 1) | ((n & 0xaaaaaaaa) >>
1);
  return n / float (0x100000000LL);
}
```

Halton and Hammersley Points

Halton: Radical inverse with different base for each dimension:

$$\vec{x}_k = (\Phi_2(k), \Phi_3(k), \Phi_5(k), \ \ldots\ , \Phi_{p_n}(k))$$

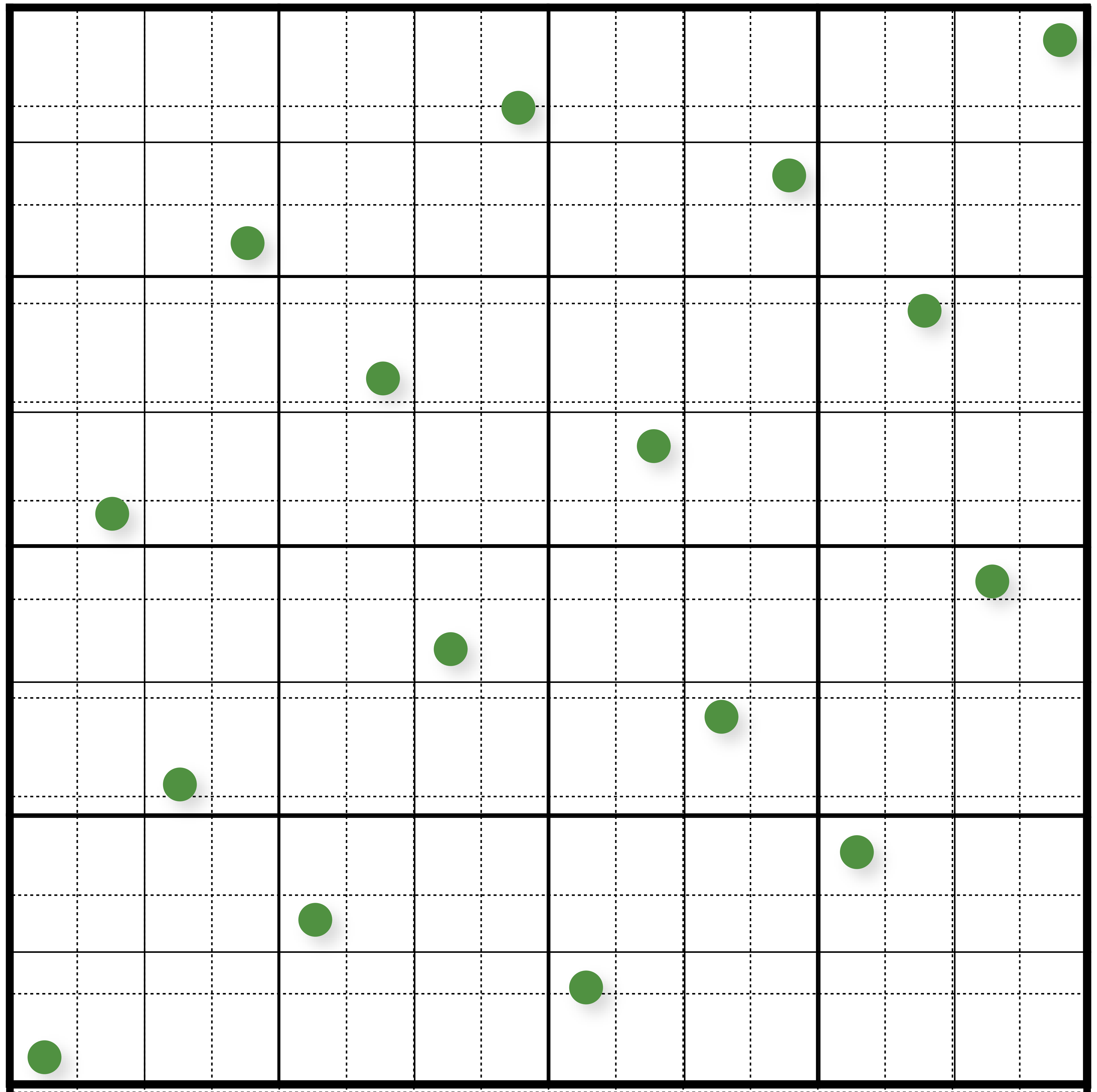
- The bases should all be relatively prime.
- Incremental/progressive generation of samples

Hammersley: Same as Halton, but first dimension is k/N:

$$\vec{x}_k = ({}^{k}/_{N}, \Phi_2(k), \Phi_3(k), \Phi_5(k), \ \ldots\ , \Phi_{p_n}(k))$$

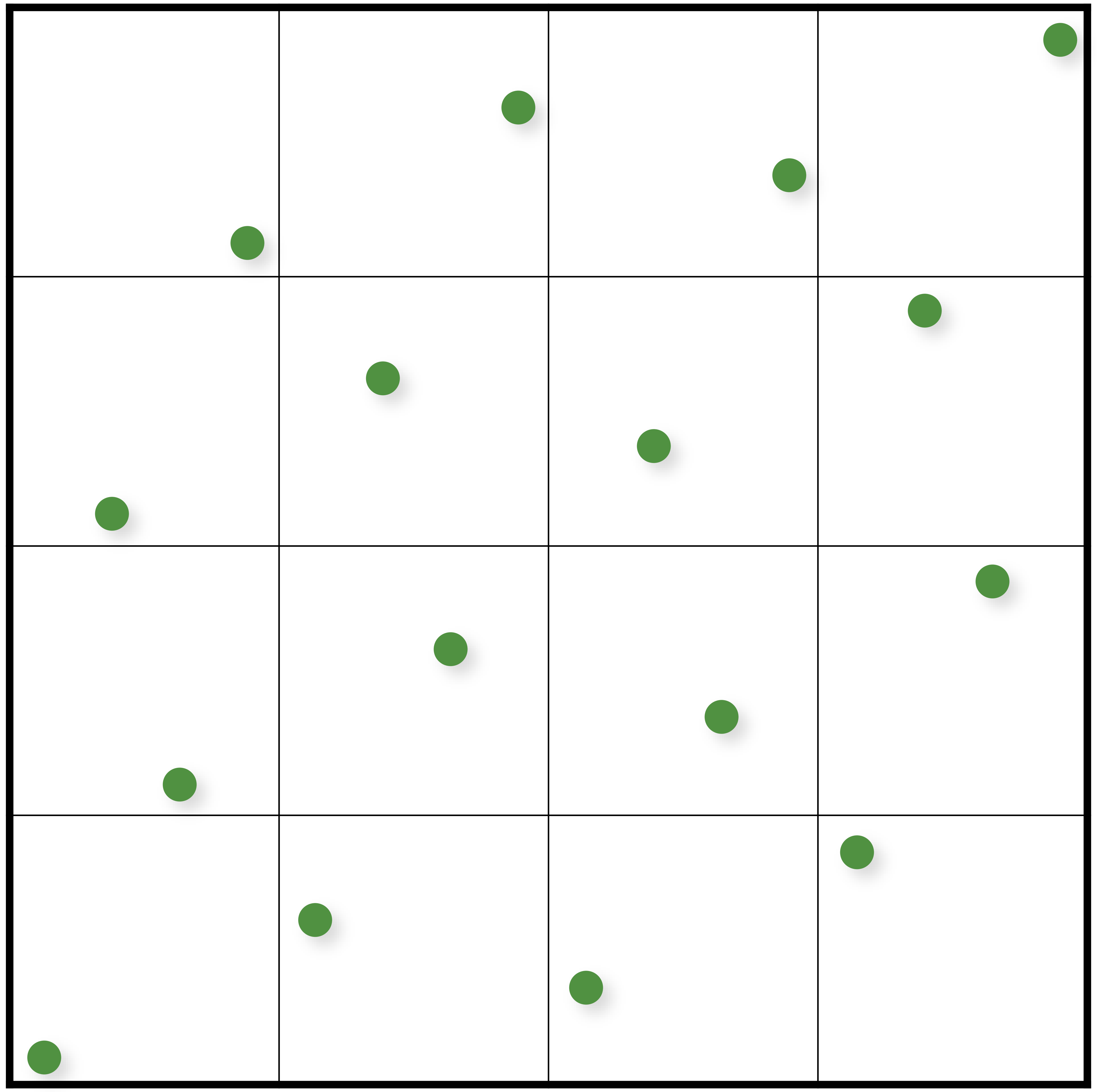
- Not incremental, need to know sample count, N, in advance

The Hammersley Sequence



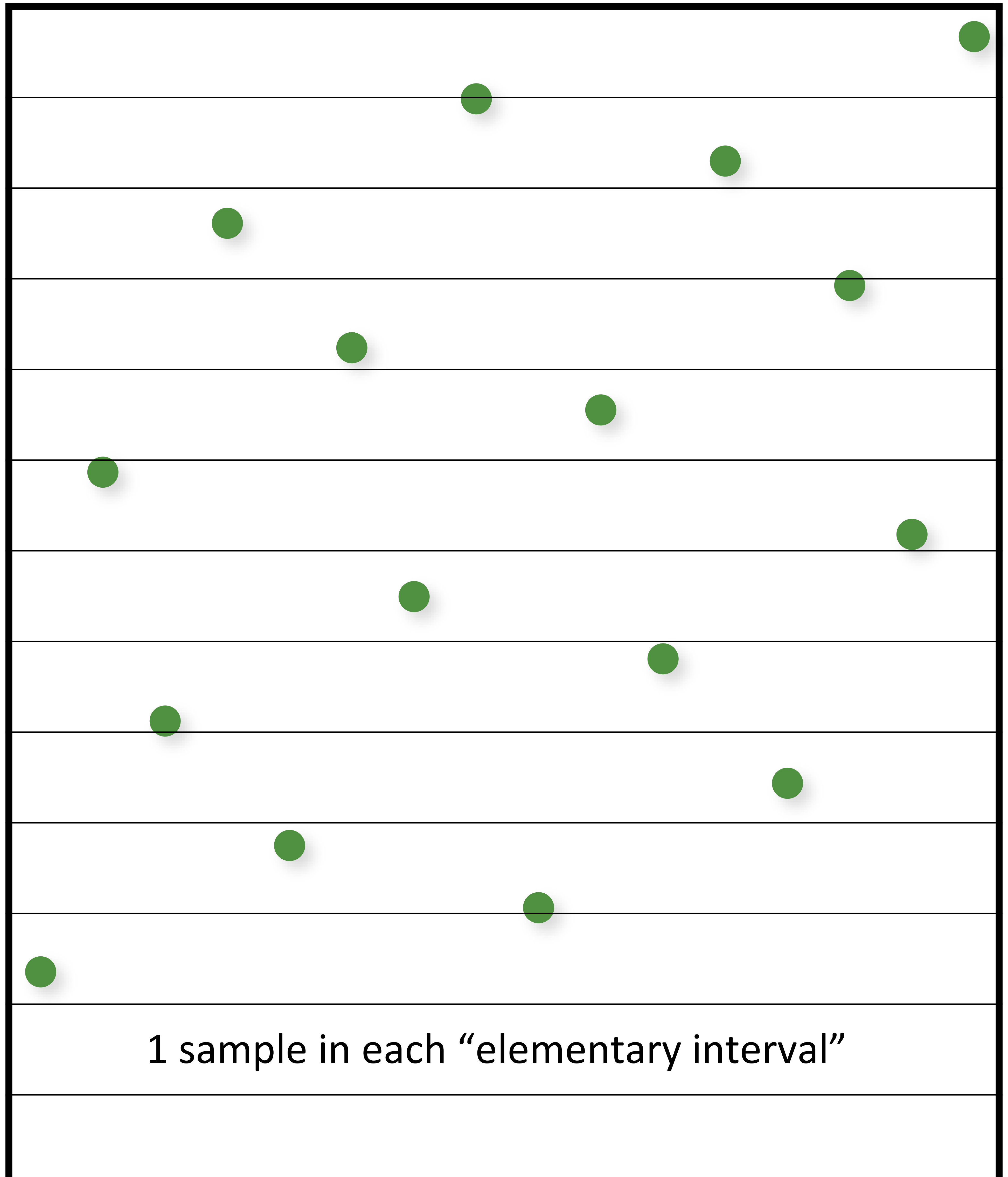
1 sample in each “elementary interval”

The Hammersley Sequence

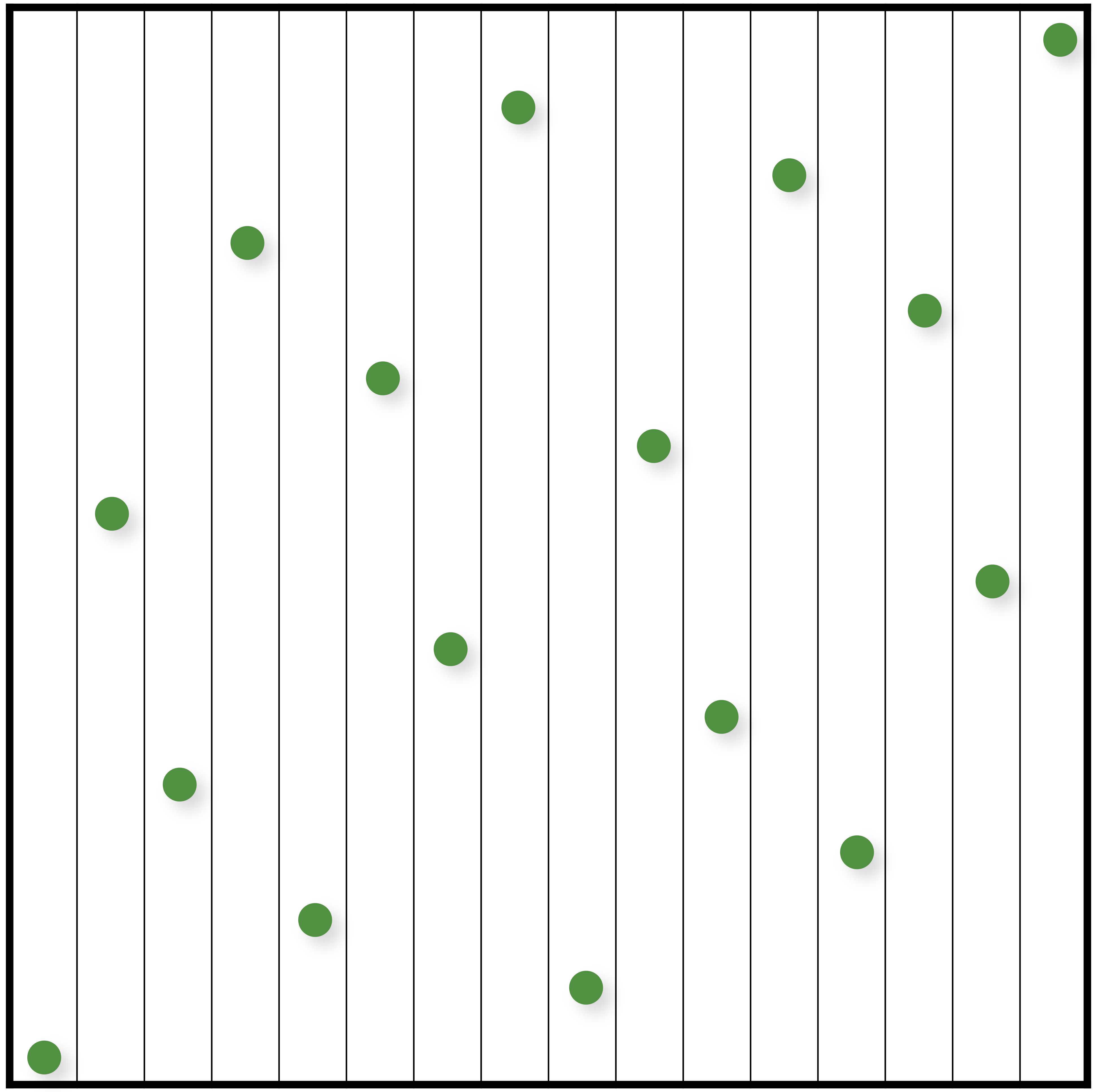


1 sample in each “elementary interval”

The Hammersley Sequence

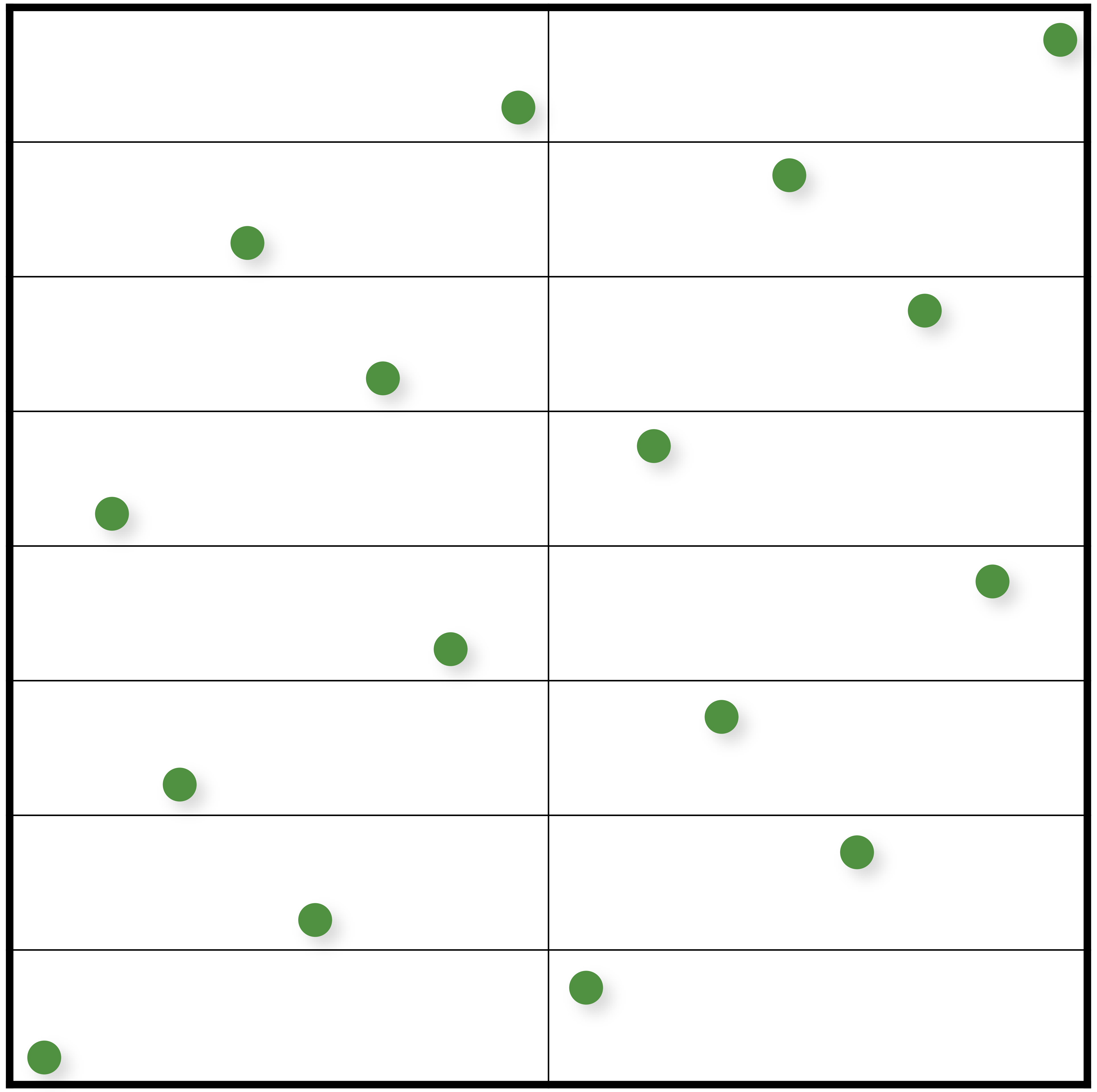


The Hammersley Sequence



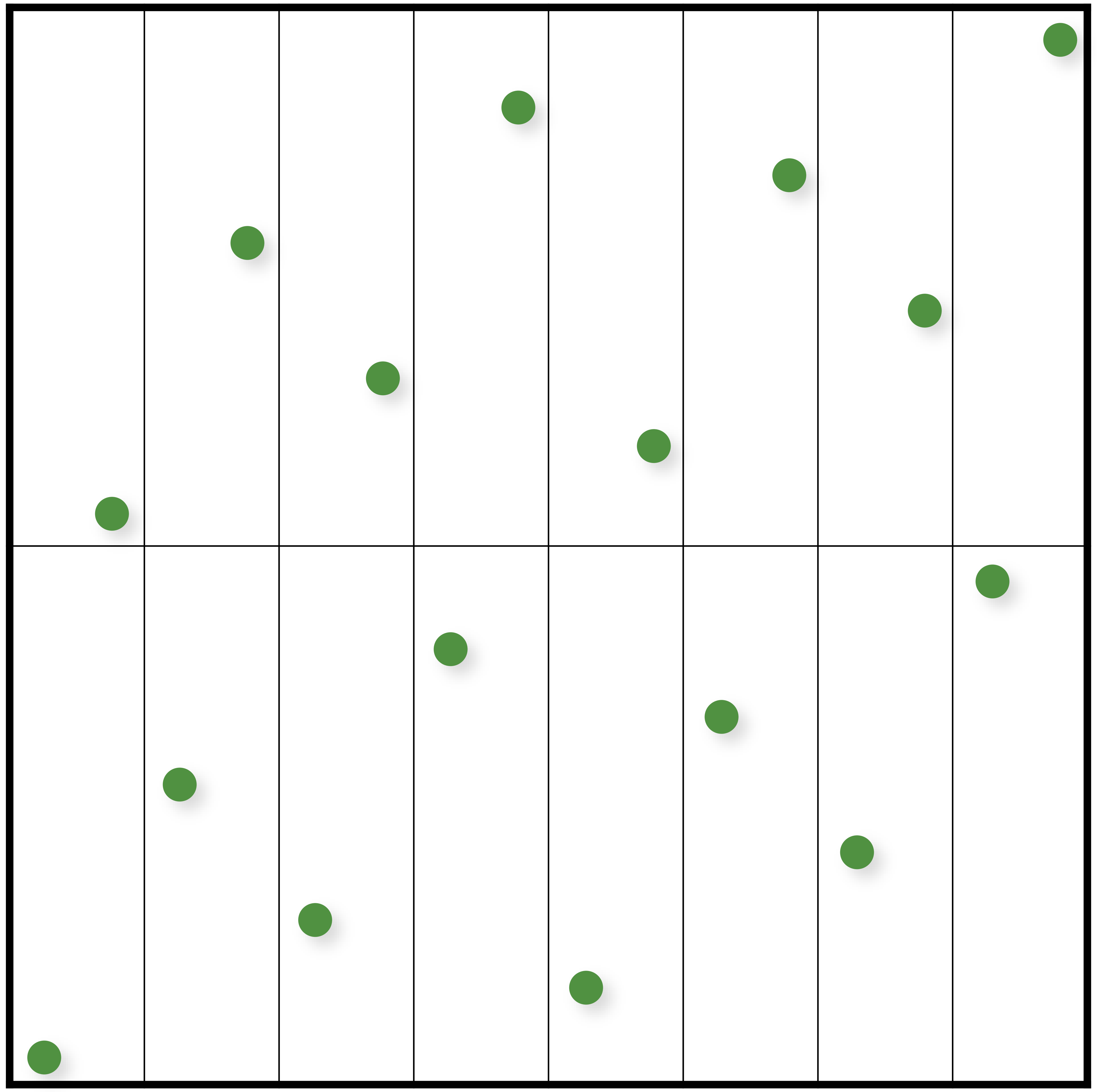
1 sample in each “elementary interval”

The Hammersley Sequence



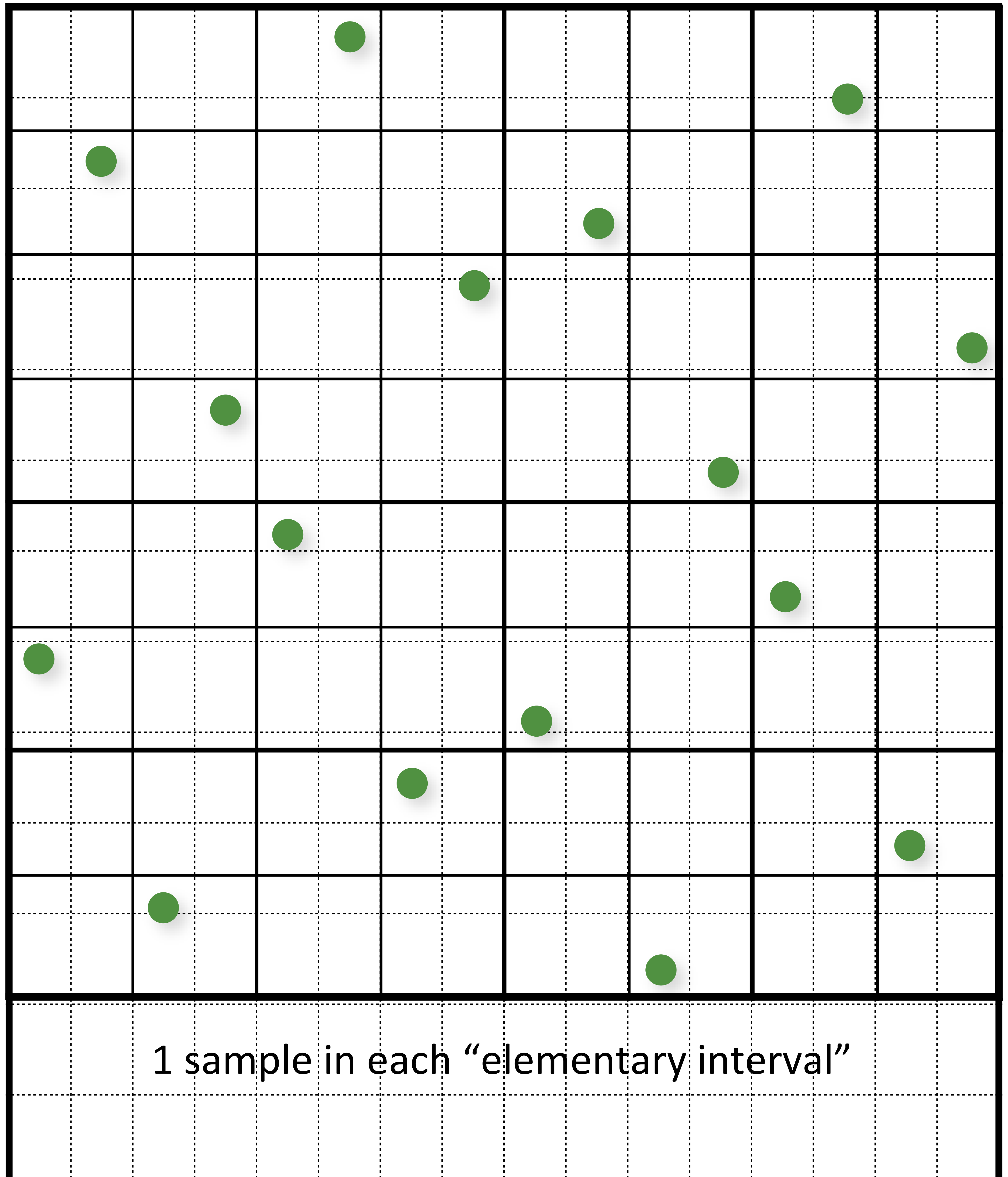
1 sample in each “elementary interval”

The Hammersley Sequence

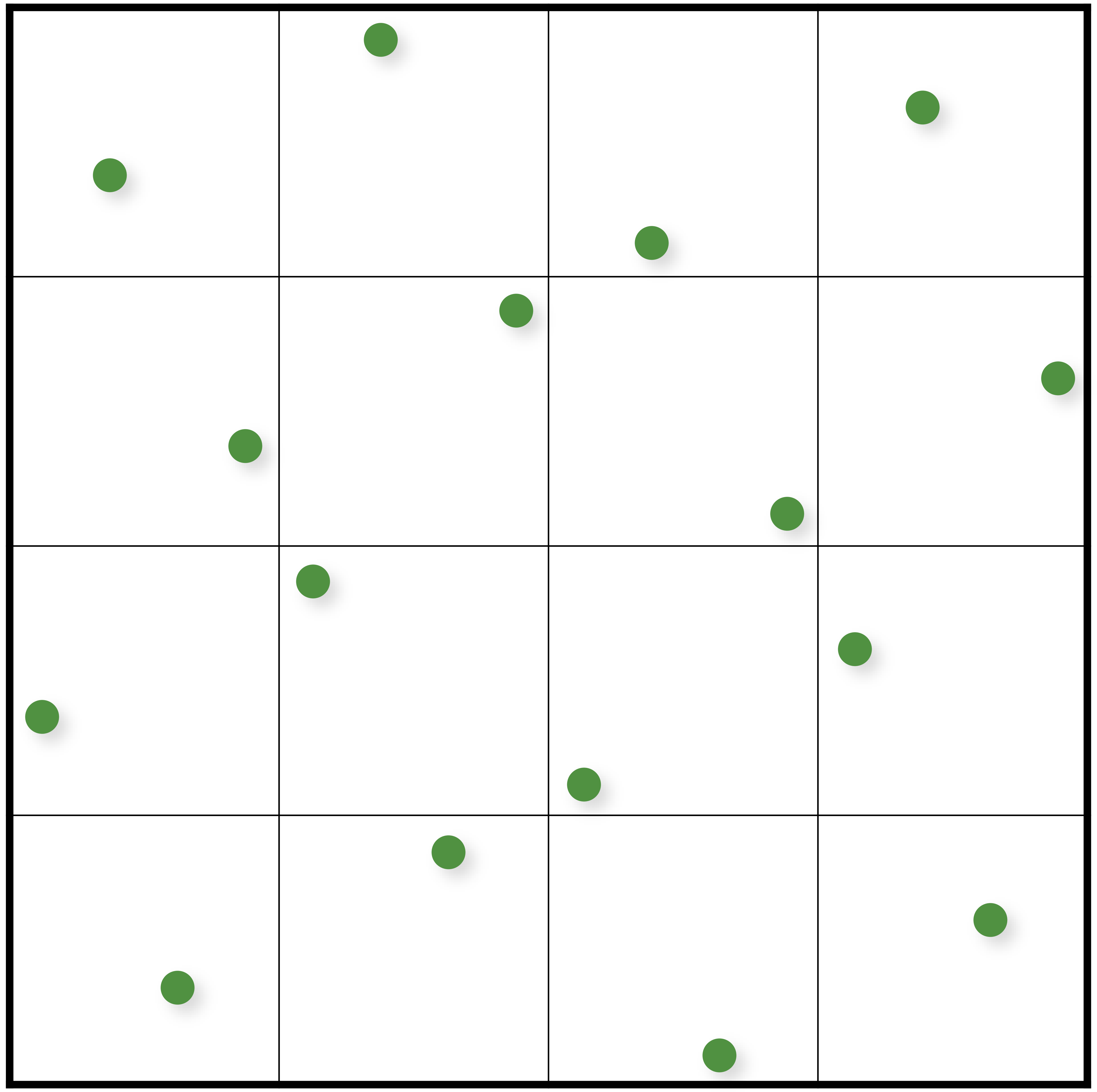


1 sample in each “elementary interval”

(0,2)-Sequences

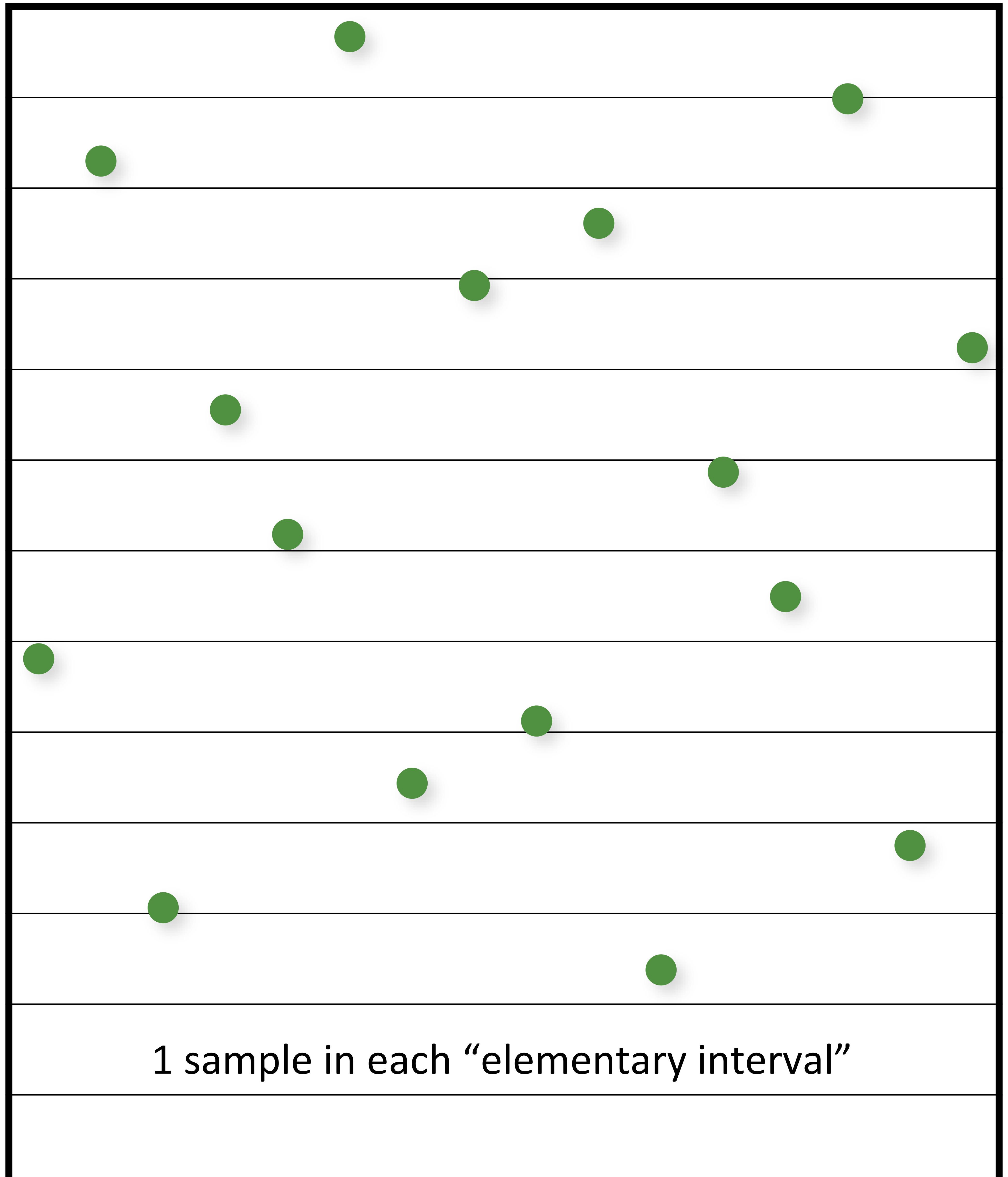


(0,2)-Sequences

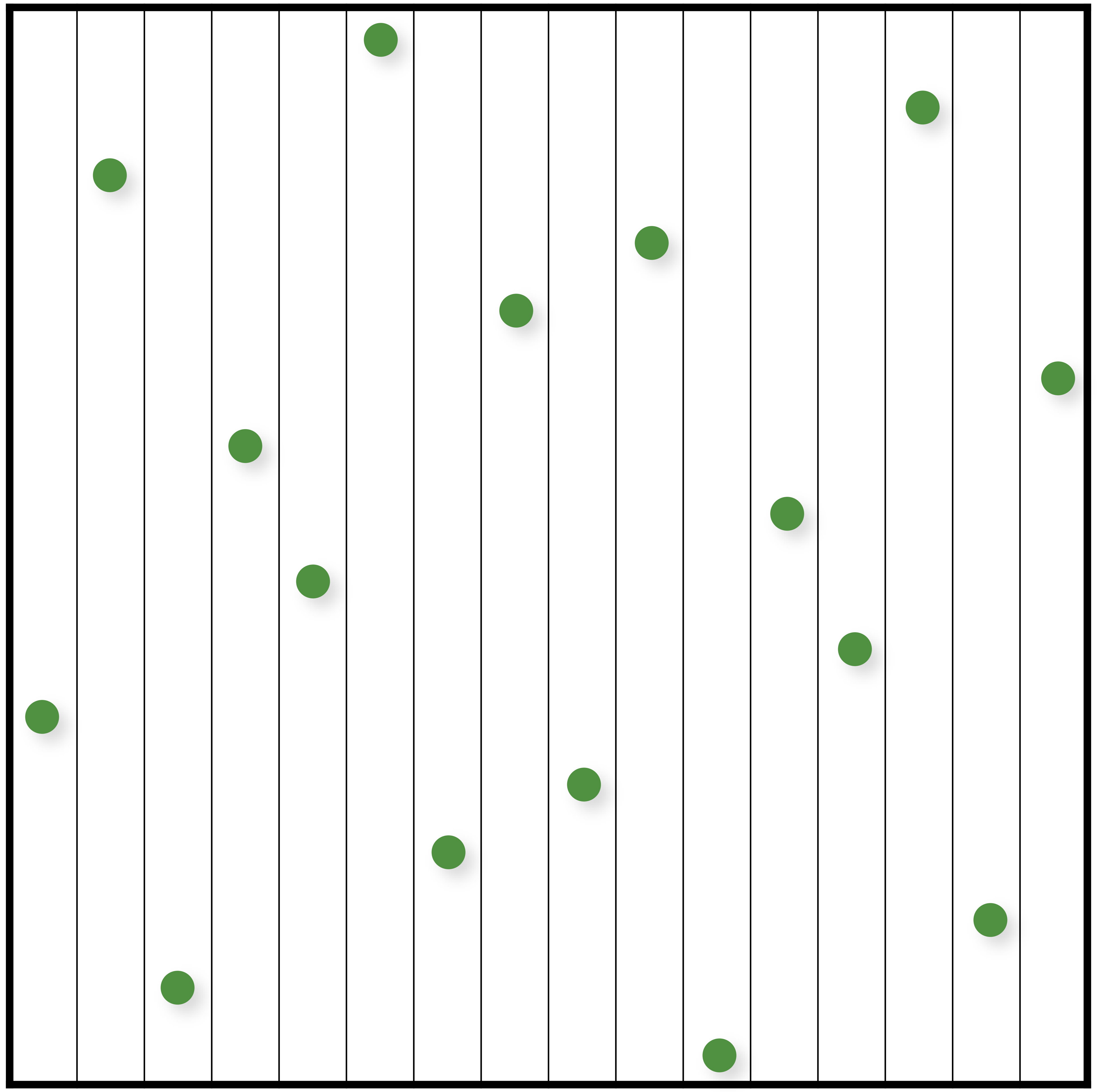


1 sample in each “elementary interval”

(0,2)-Sequences

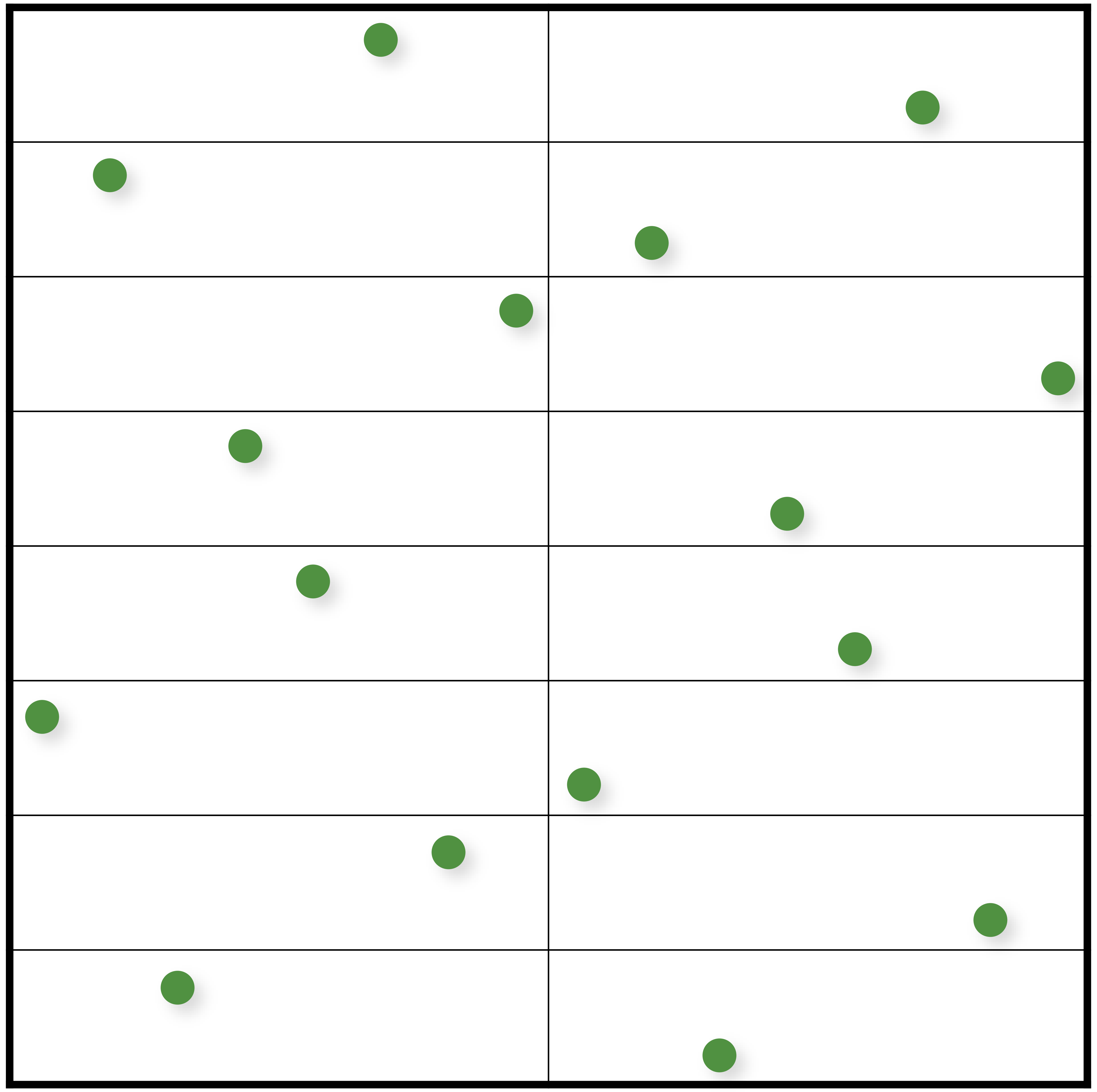


(0,2)-Sequences



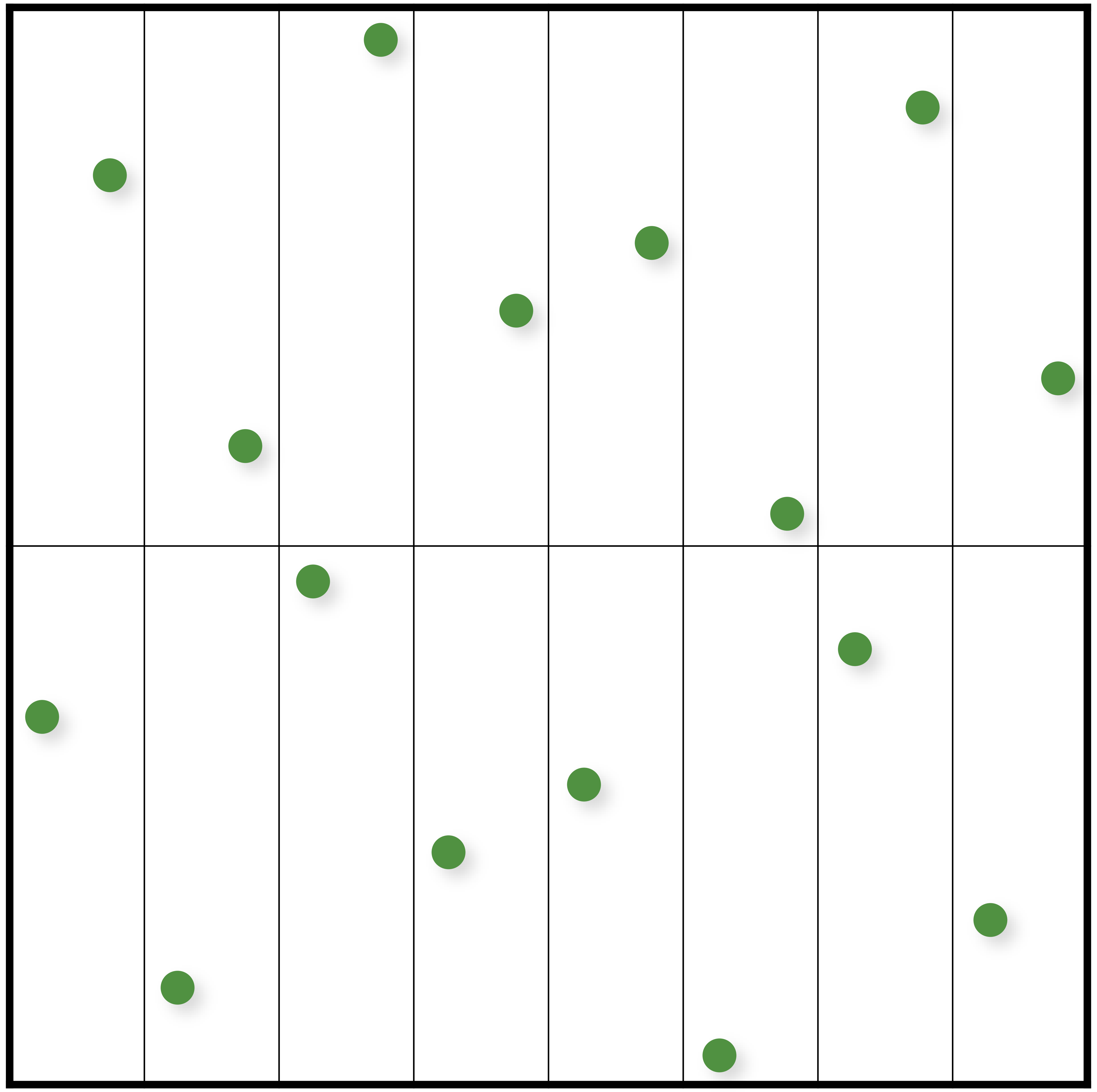
1 sample in each “elementary interval”

(0,2)-Sequences



1 sample in each “elementary interval”

(0,2)-Sequences



1 sample in each “elementary interval”

More info on QMC in Rendering

S. Premoze, A. Keller, and M. Raab.
Advanced (Quasi-) Monte Carlo Methods for Image Synthesis. In SIGGRAPH 2012 courses.

Many more...

Sobol

Faure

Larcher-Pillichshammer

Folded Radical Inverse

(t,s)-sequences & (t,m,s)-nets

Scrambling/randomization

much more...

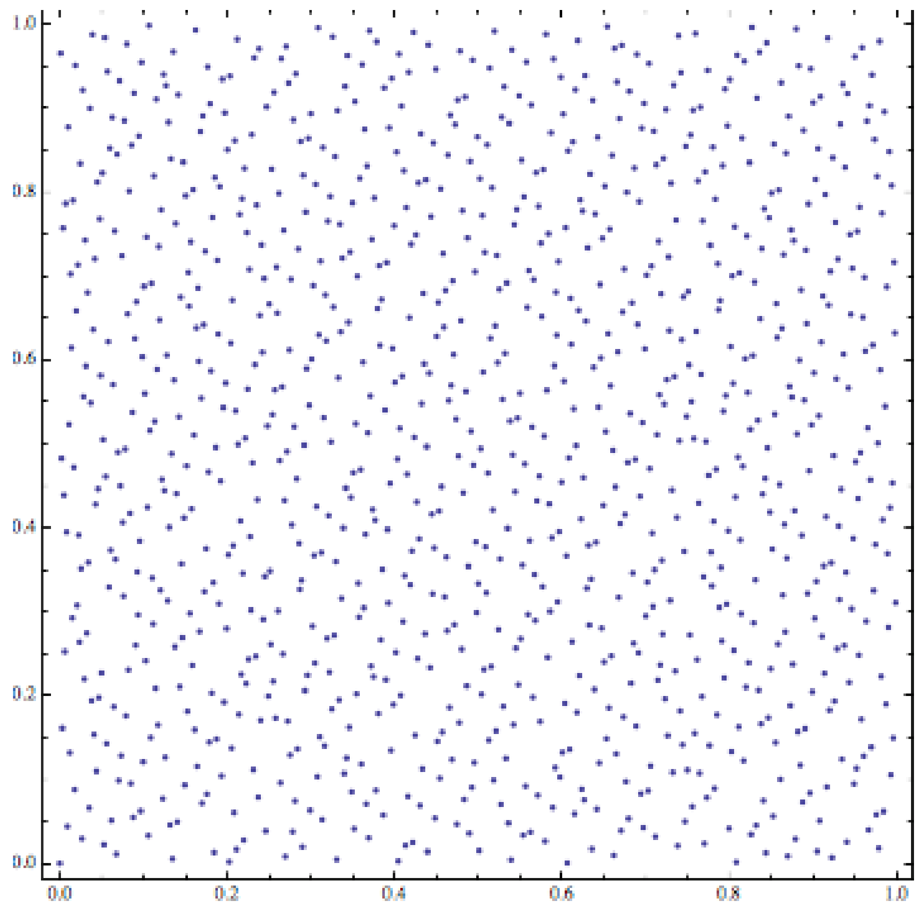
Challenges

LD sequence identical for multiple runs

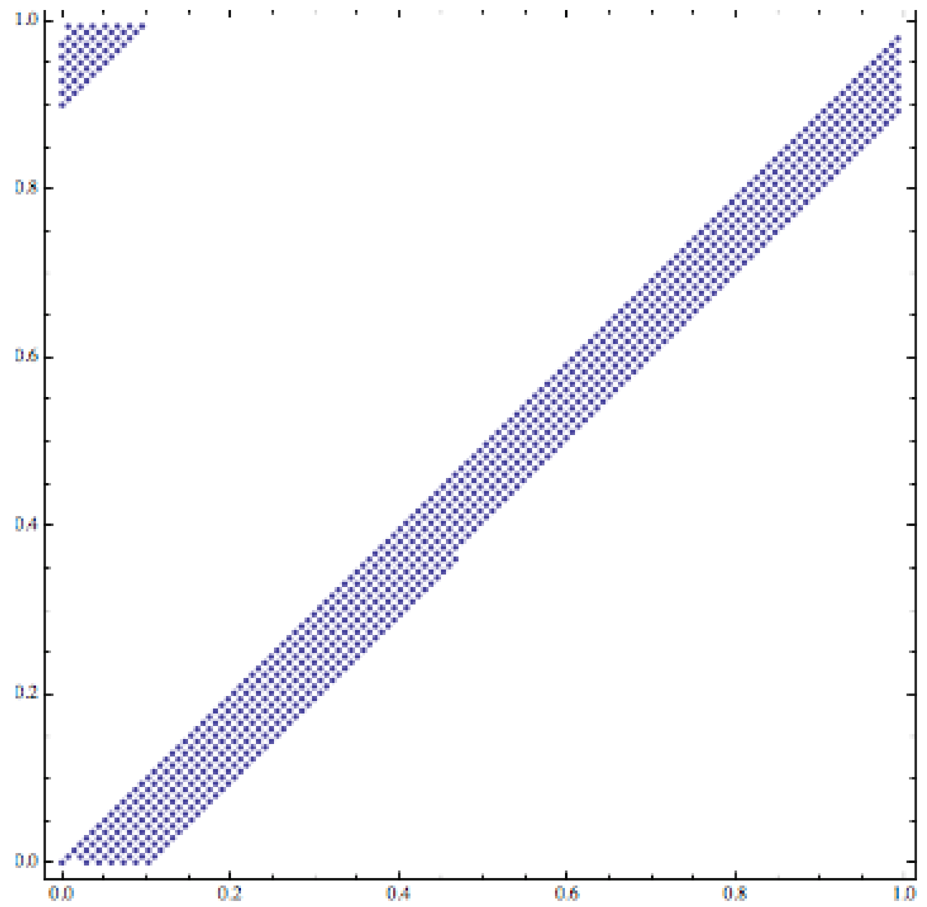
- cannot average independent images!
- no “random” seed

Quality decreases in higher dimensions

Halton Sequence



Dimensions 1 and 2



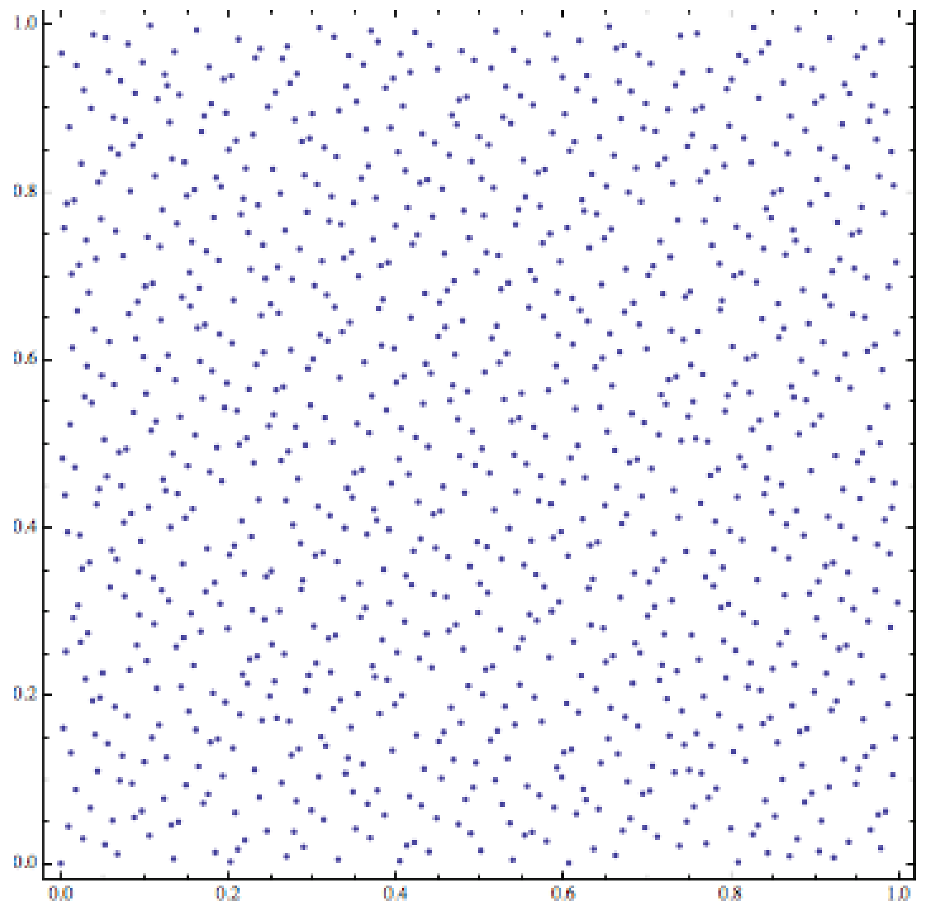
Dimensions 32 and 33

Randomized/Scrambled Sequences

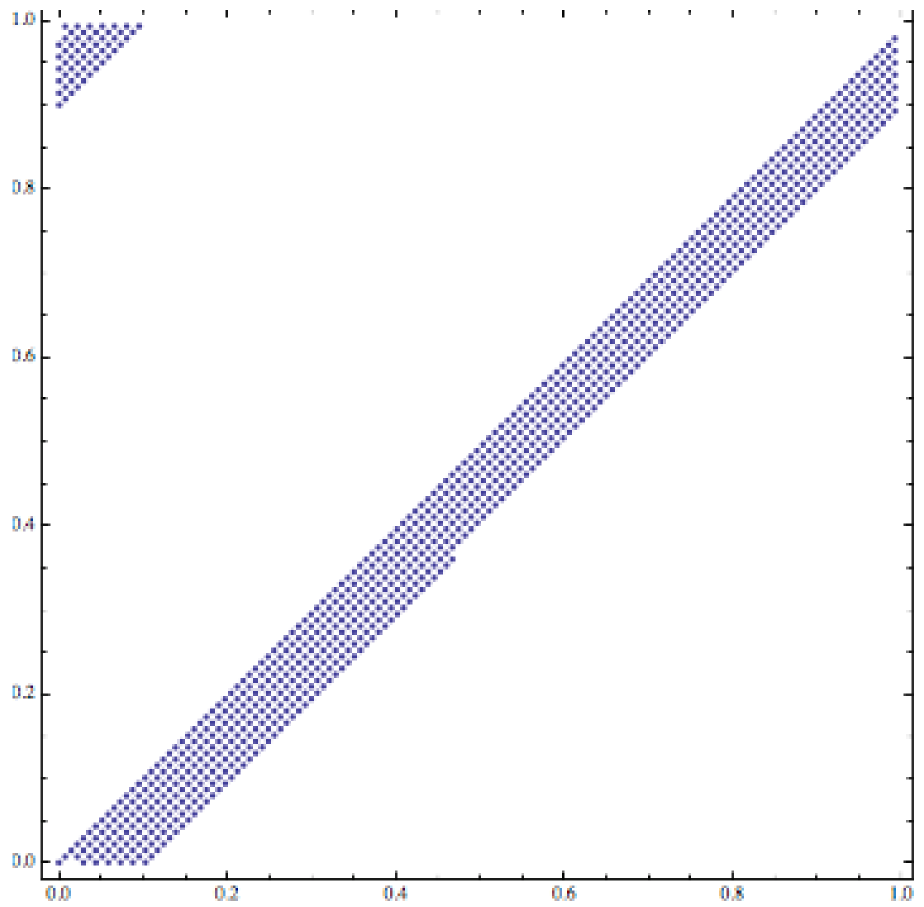
Random permutations: compute a permutation table for the order of the digits and use it when computing the radical inverse

$$\Phi_b(n) = 0.\pi(d_1)\pi(d_2)...\pi(d_m)$$

Without scrambling

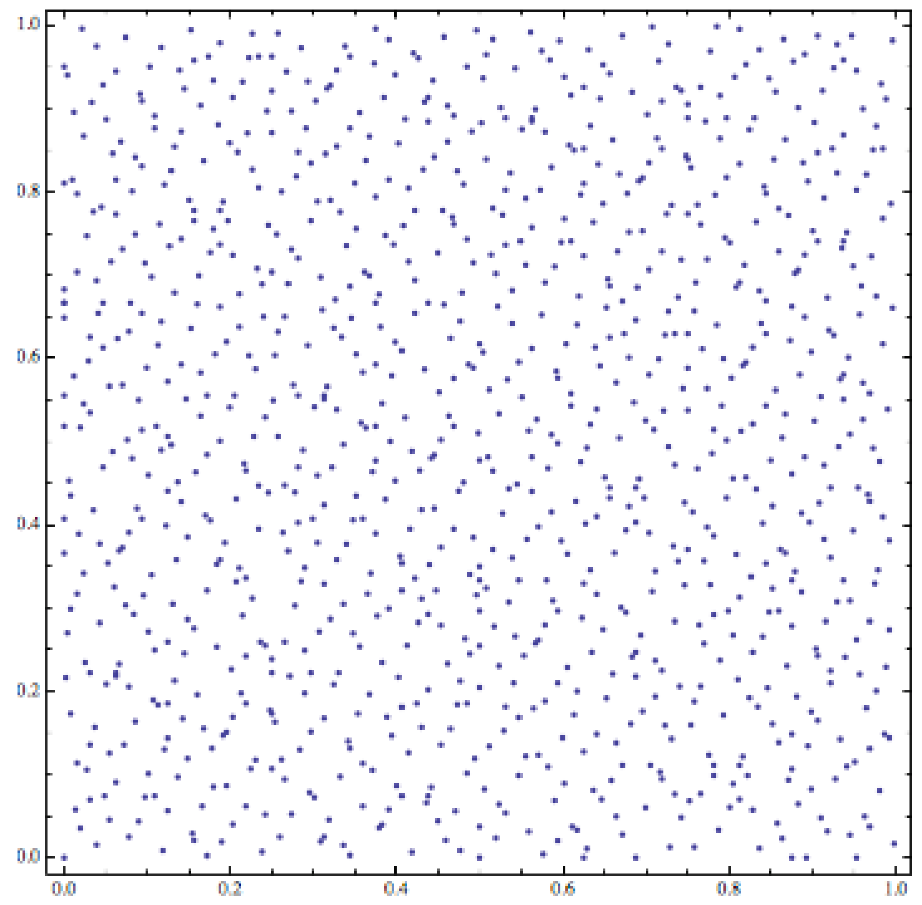


Dimensions 1 and 2

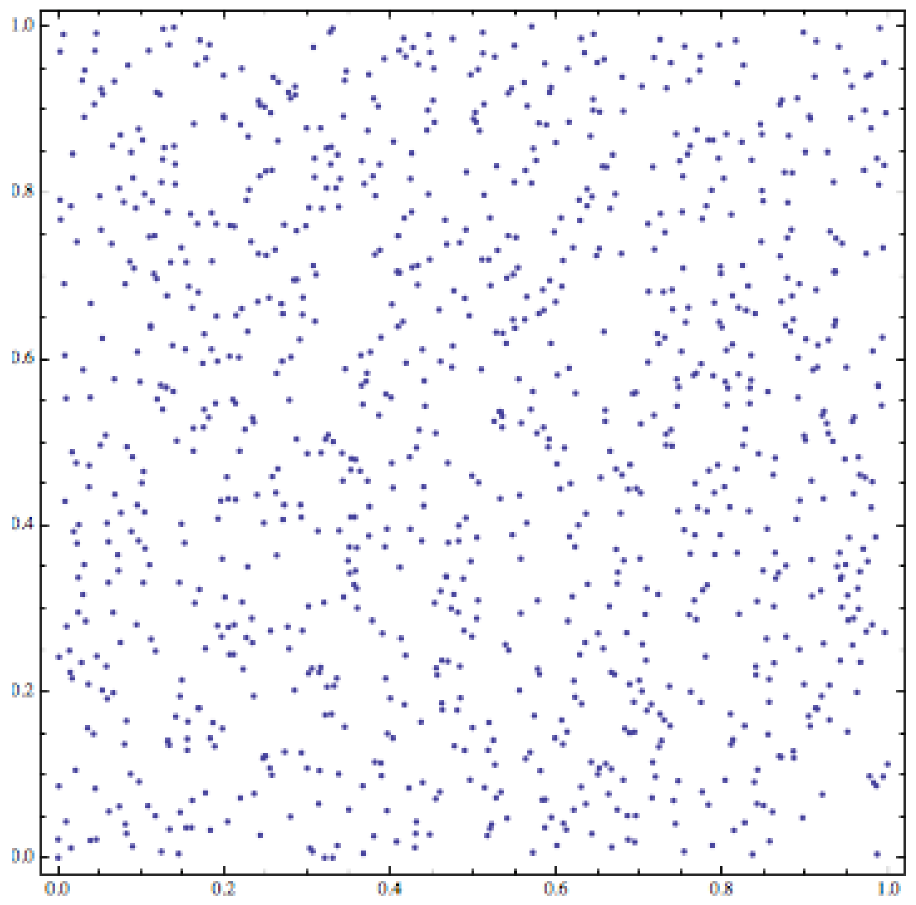


Dimensions 32 and 33

With scrambling



Dimensions 1 and 2



Dimensions 32 and 33

Randomized/Scrambled Sequences

Random permutations: compute a permutation table for the order of the digits and use it when computing the radical inverse

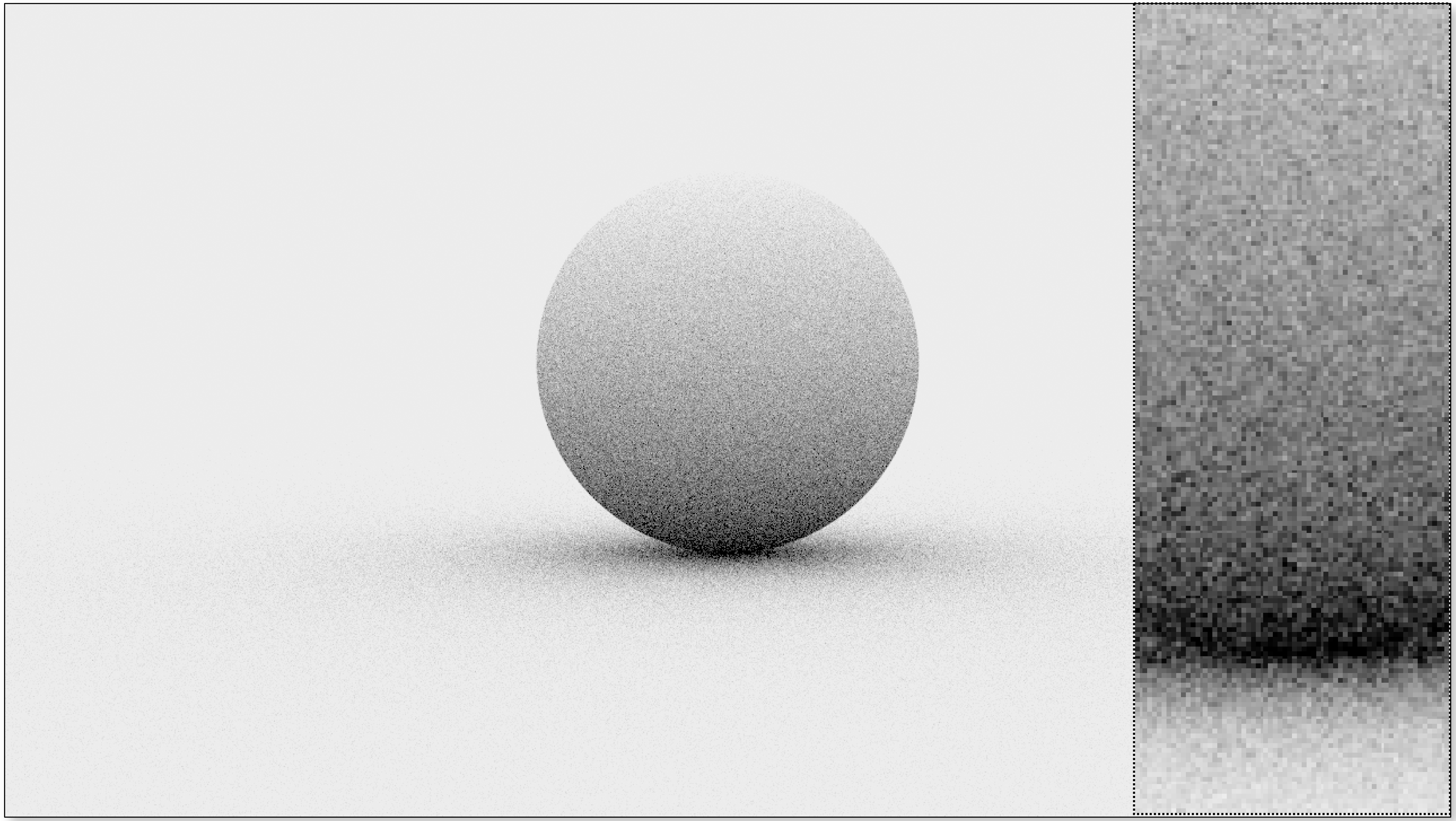
- Can be done very efficiently for base 2 with XOR operation

See PBRe2 Ch7 for details

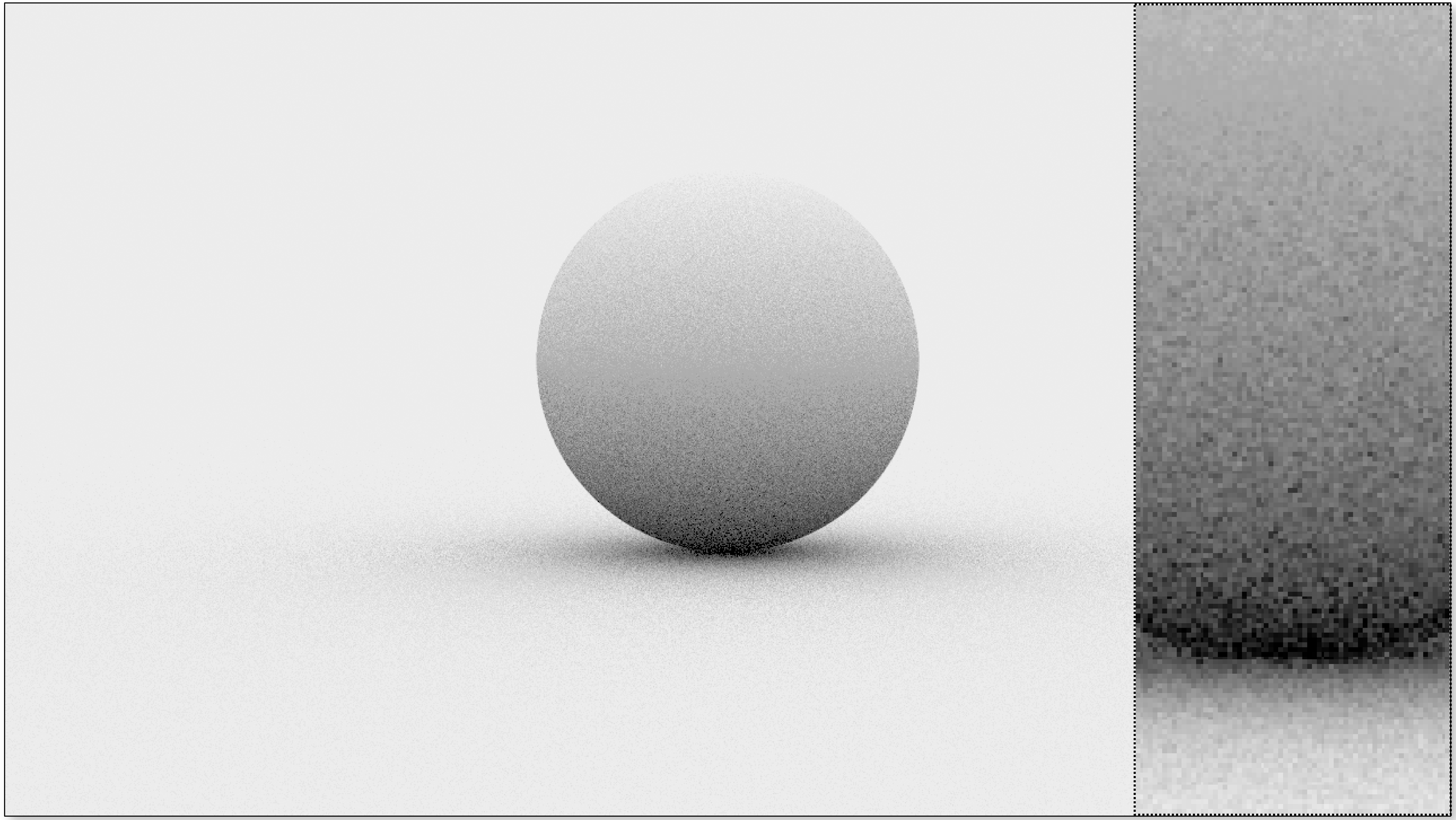
Scrambled Radical Inverse (Base 2)

```
float vanDerCorputRIU(uint n, uint scramble = 0)
{
   n = (n << 16) | (n >> 16);
   n = ((n & 0x00ff00ff) << 8) | ((n & 0xff00ff00) >>
8);
   n = ((n & 0x0f0f0f0f) << 4) | ((n & 0xf0f0f0f0) >>
4);
   n = ((n & 0x33333333) << 2) | ((n & 0xcccccccc) >>
2);
   n = ((n & 0x55555555) << 1) | ((n & 0xaaaaaaaa) >>
1);
   n ^= scramble;
   return n / float (0x100000000LL);
}
```

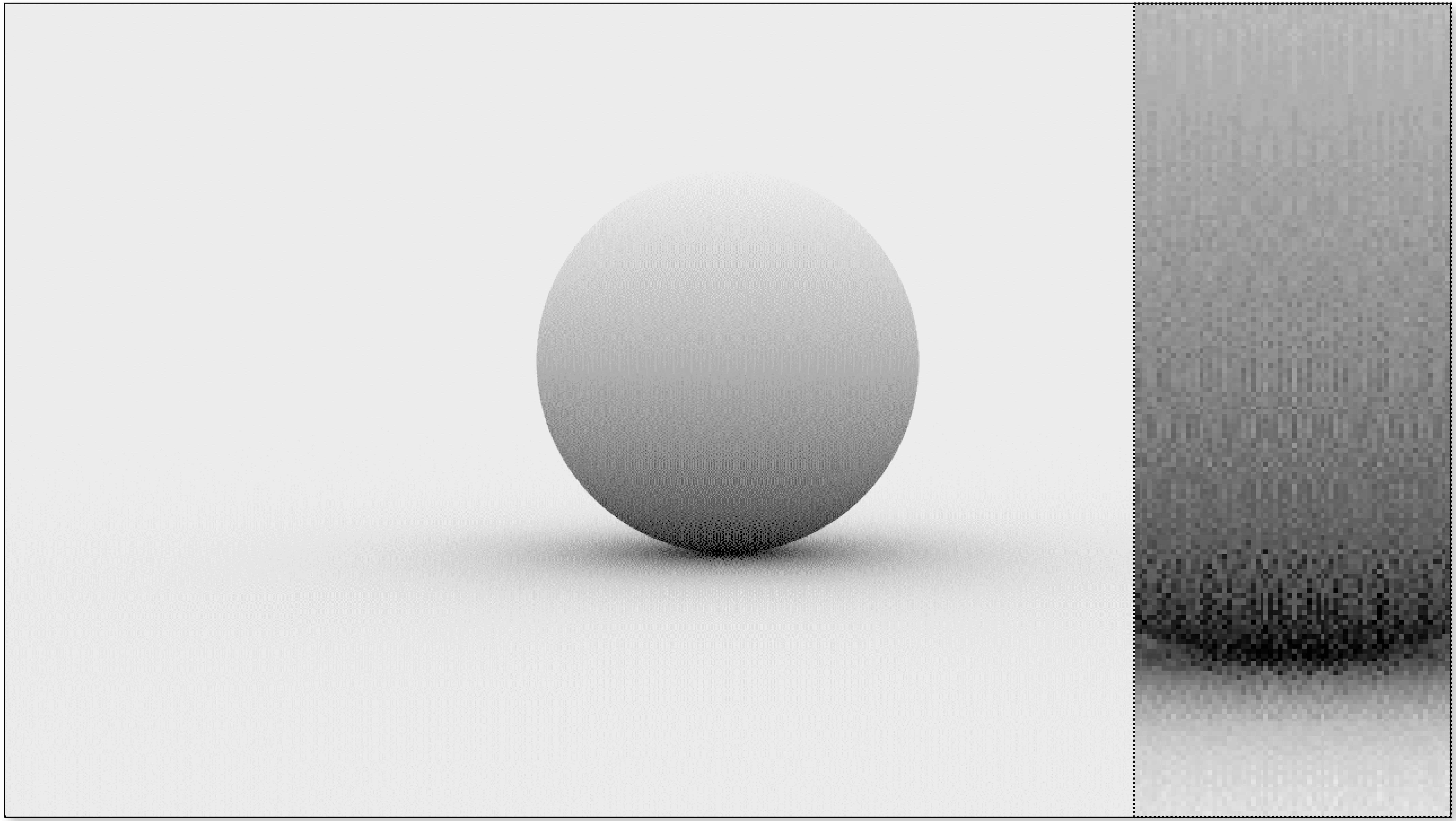
Monte Carlo (16 random samples)



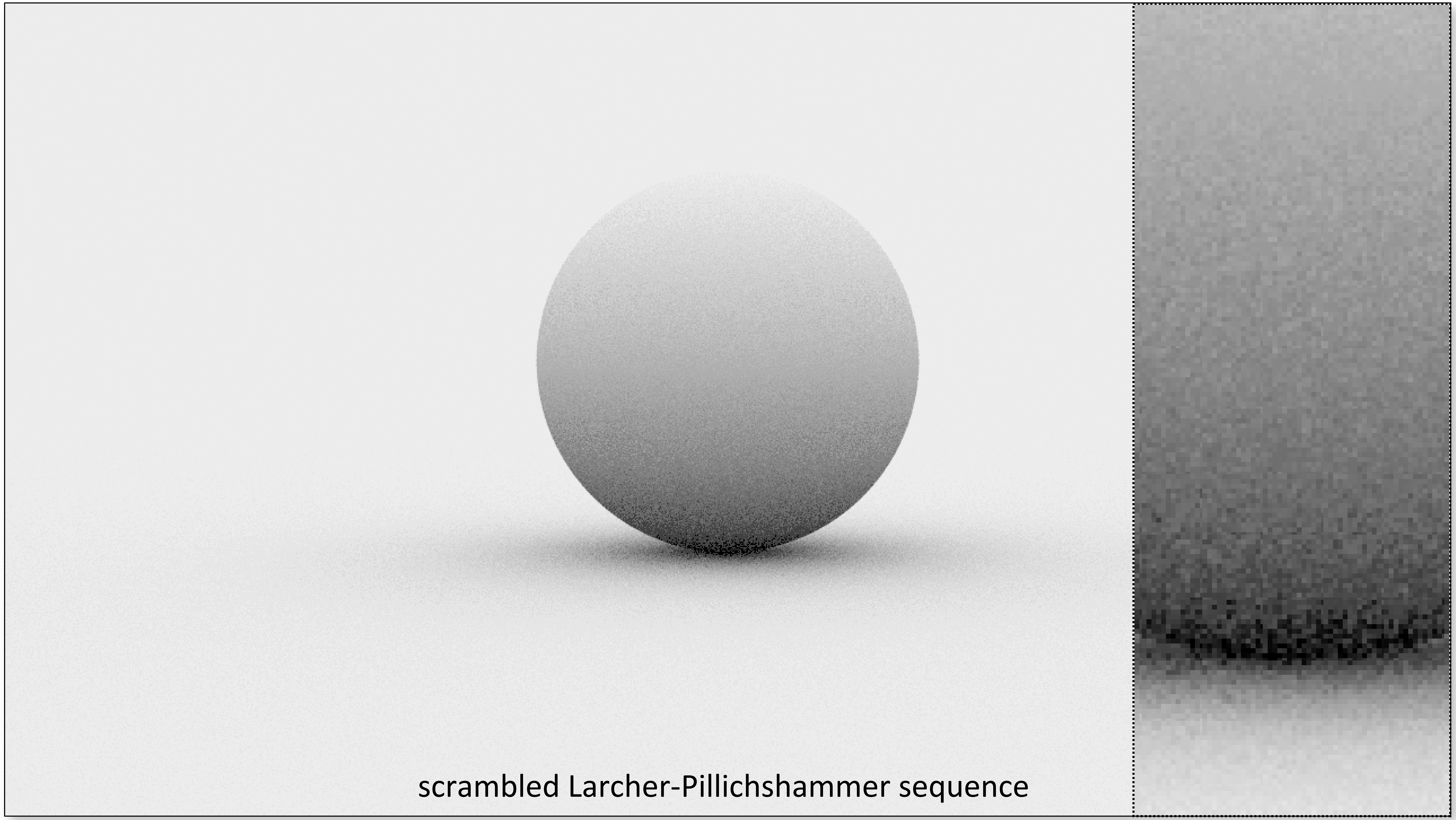
Monte Carlo (16 stratified samples)



Quasi-Monte Carlo (16 Halton samples)



Scrambled Quasi-Monte Carlo



scrambled Larcher-Pillichshammer sequence

Implementation tips

Using QMC can often lead to unintuitive, difficult-to-debug problems.

- Always code up MC algorithms first, using random numbers, to ensure correctness
- Only after confirming correctness, slowly incorporate QMC into the mix

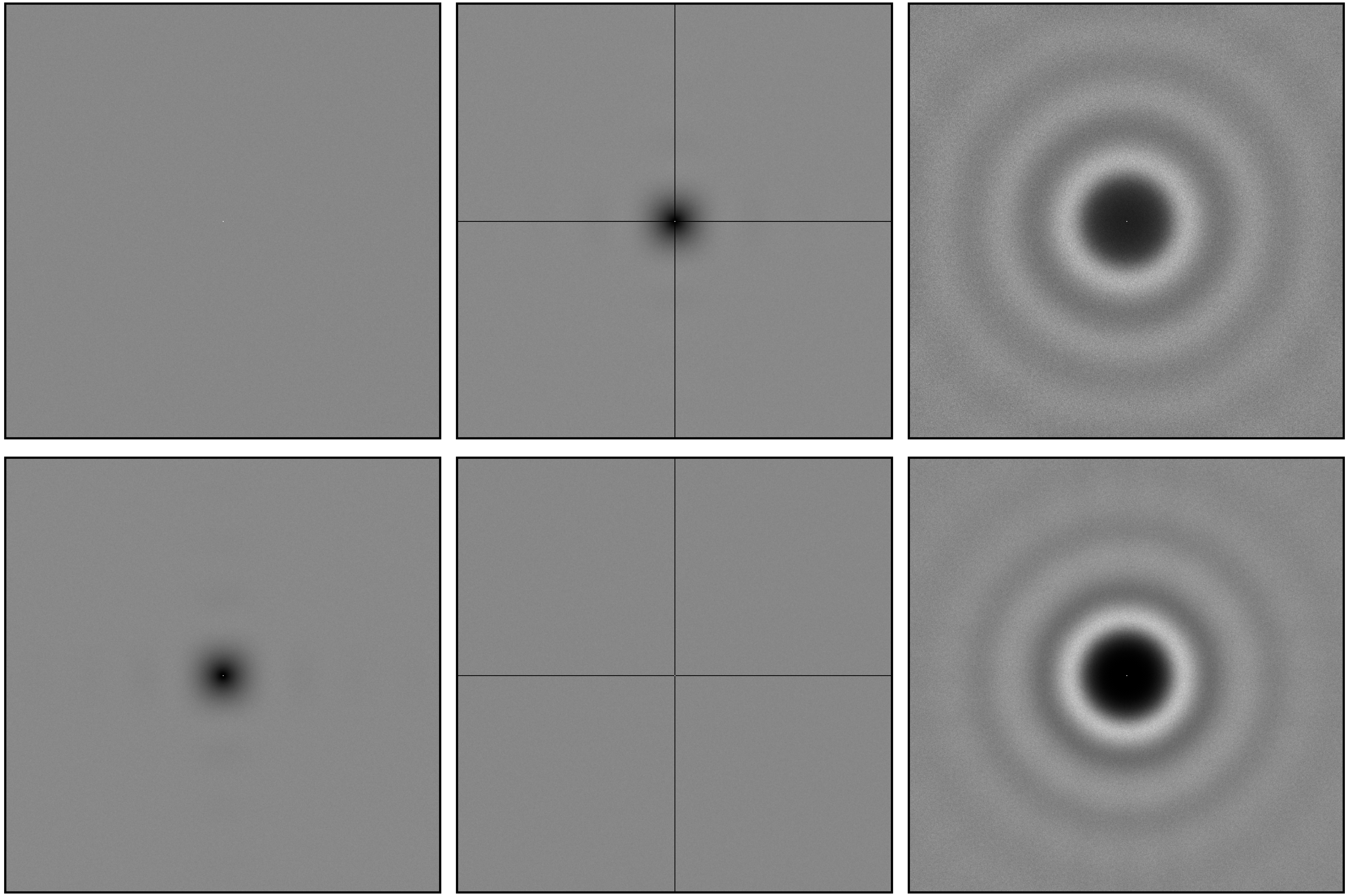
How do you add this to your renderer?

Lots of details in the book

Read about the Sampler interface

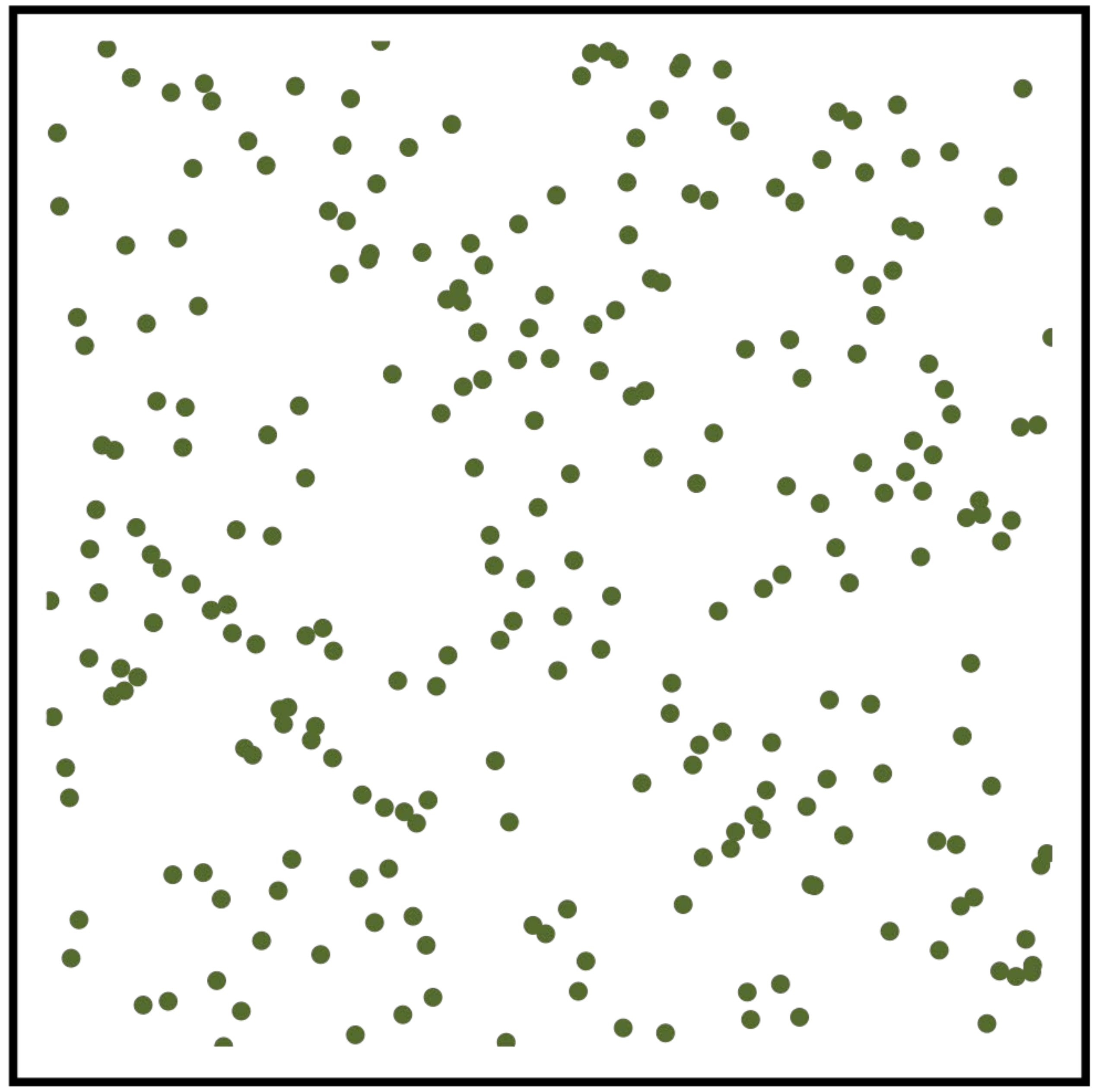
- Basic idea: replace global randf with a Sampler class that produces random (or stratified/quasi-random) numbers
- Also better for multi-threading

How can we predict error from these?

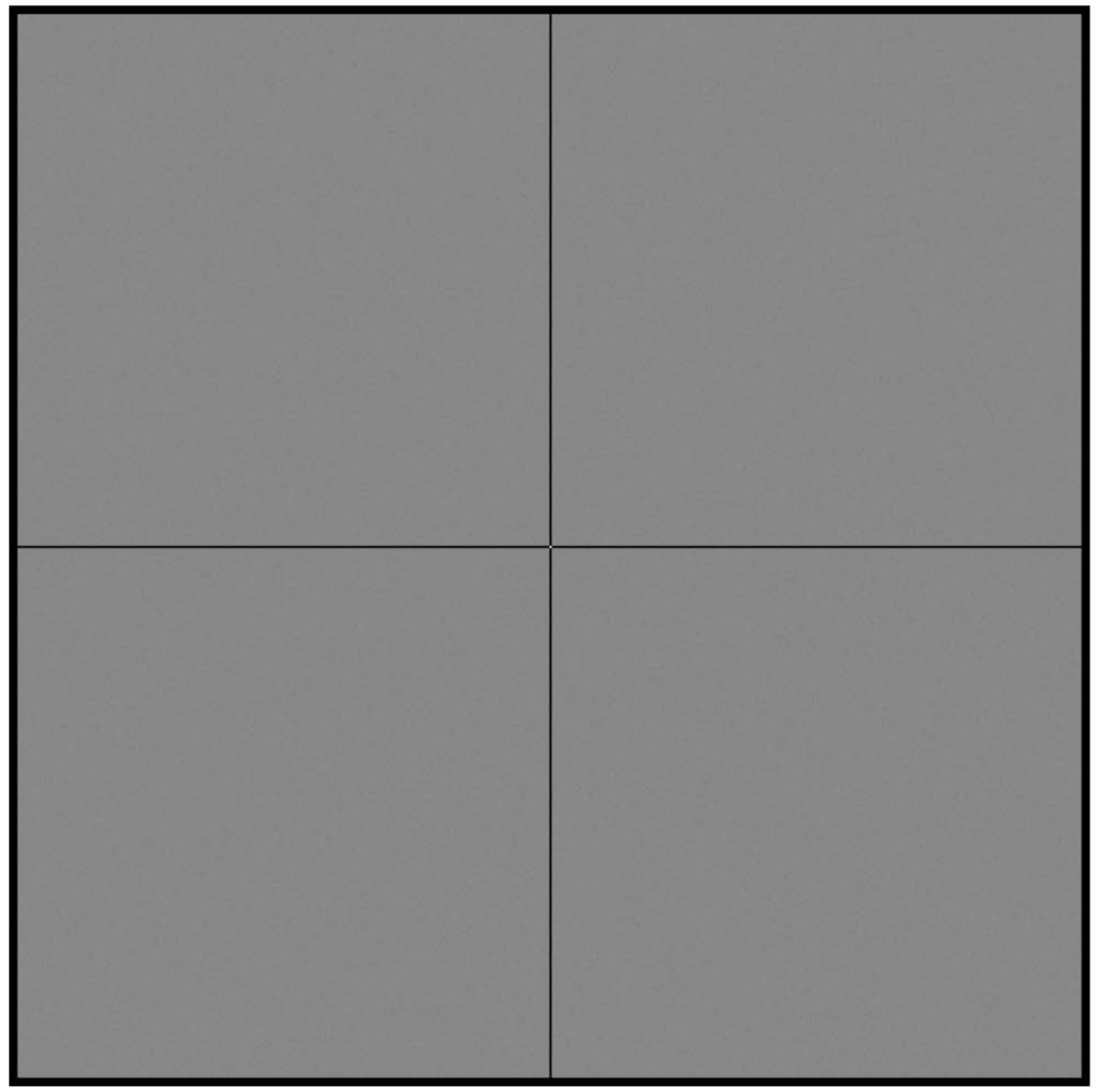


N-Rooks Sampling

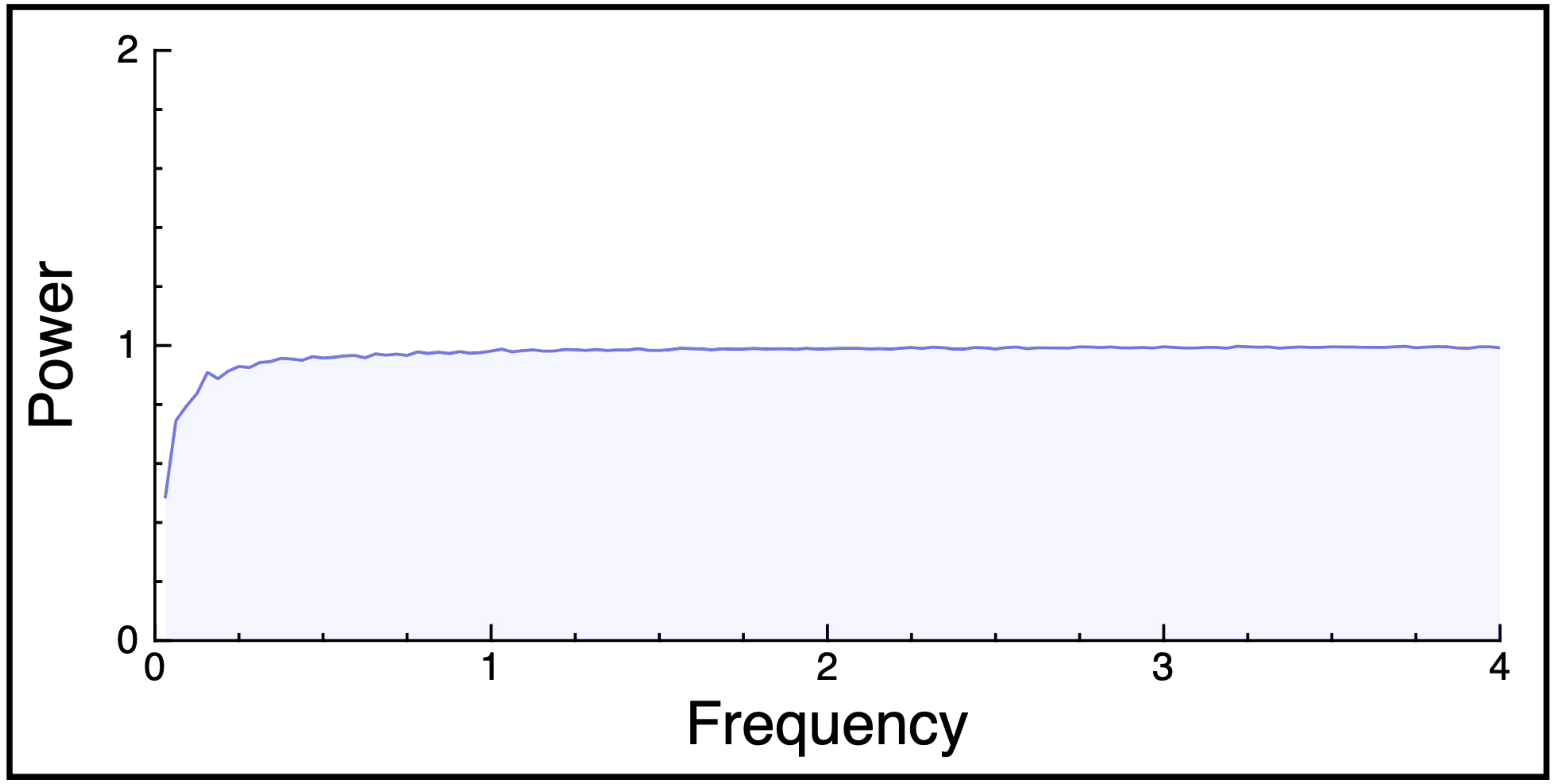
Samples



Expected power spectrum

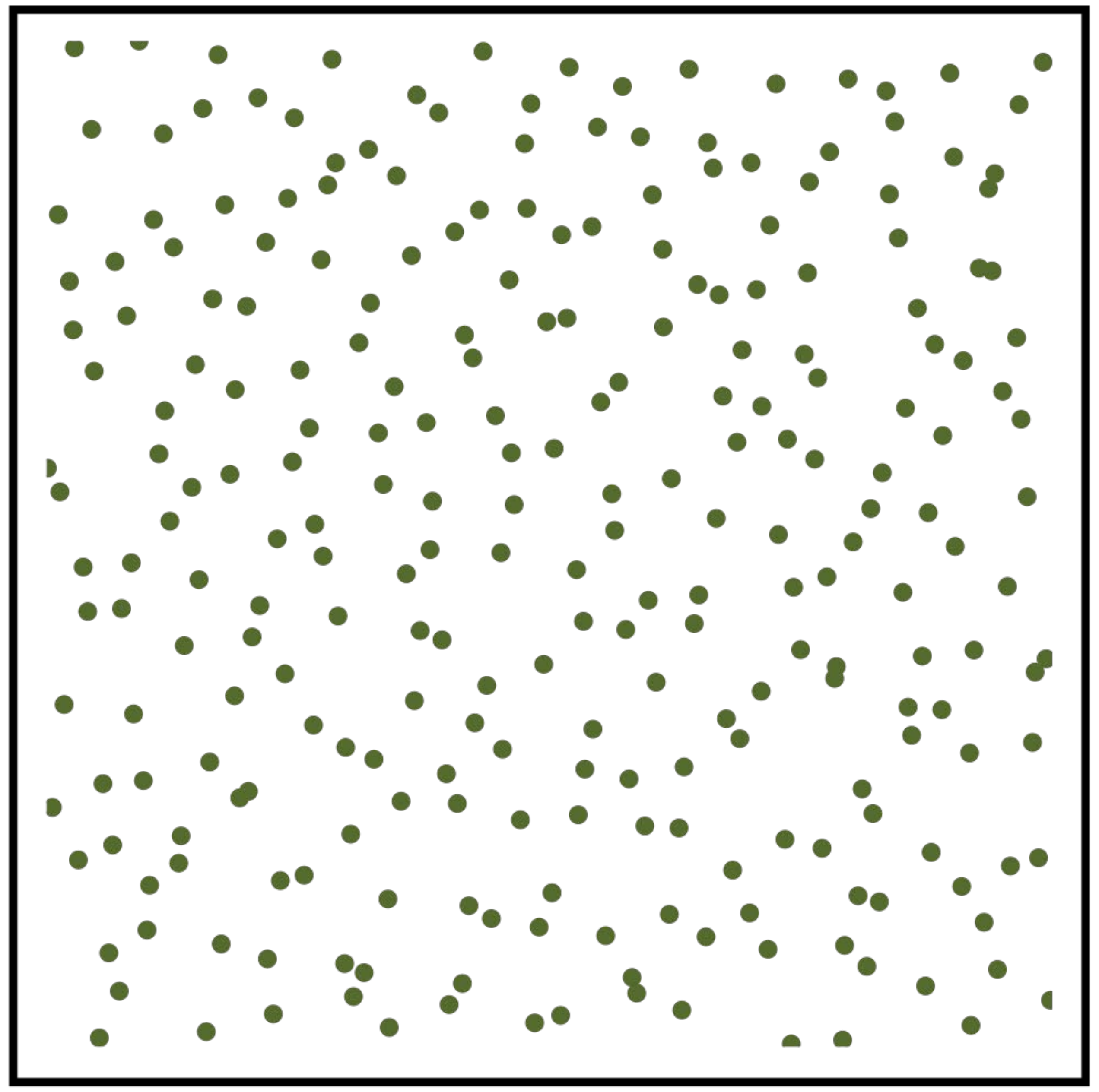


Radial mean

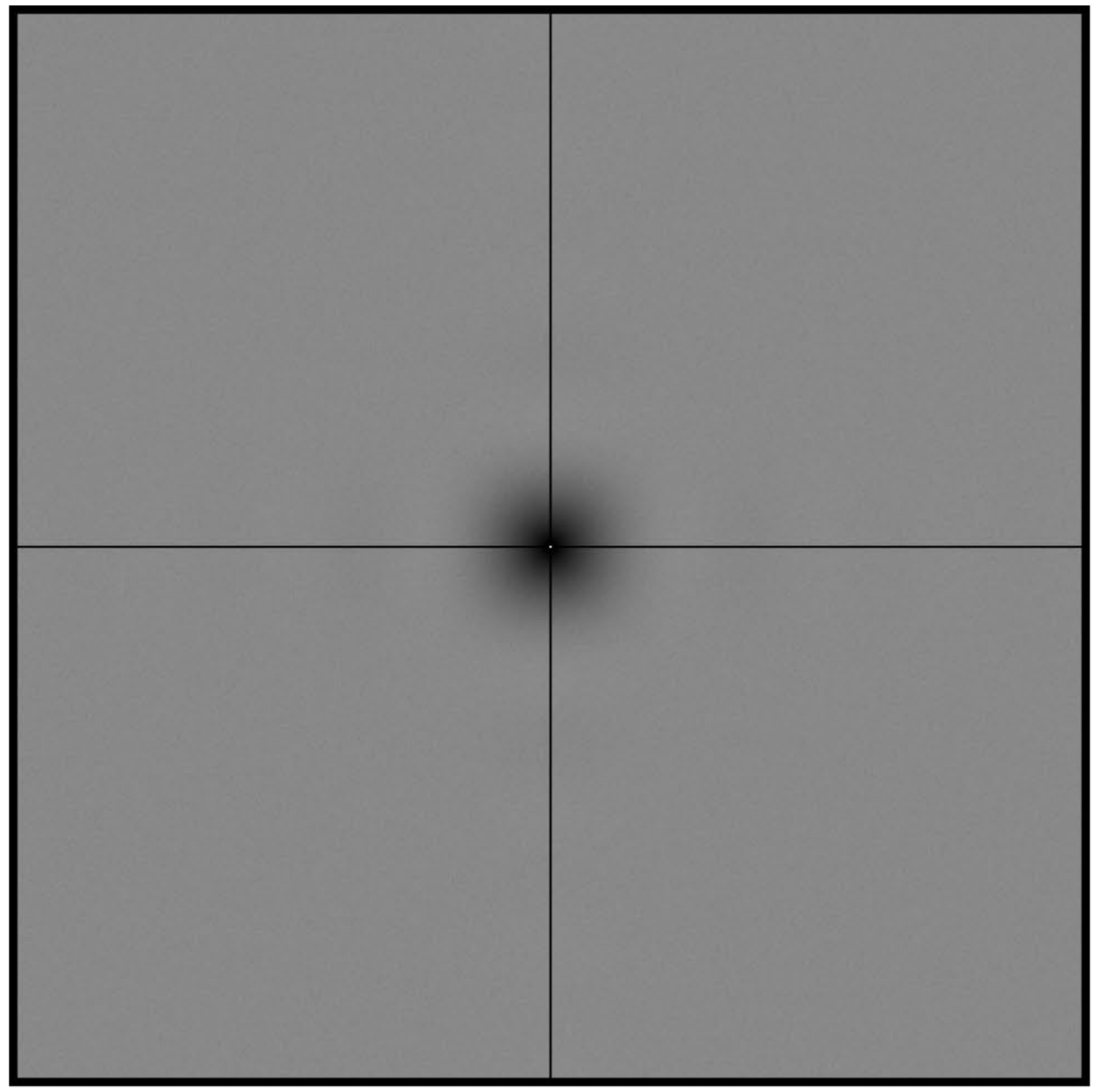


Multi-Jittered Sampling

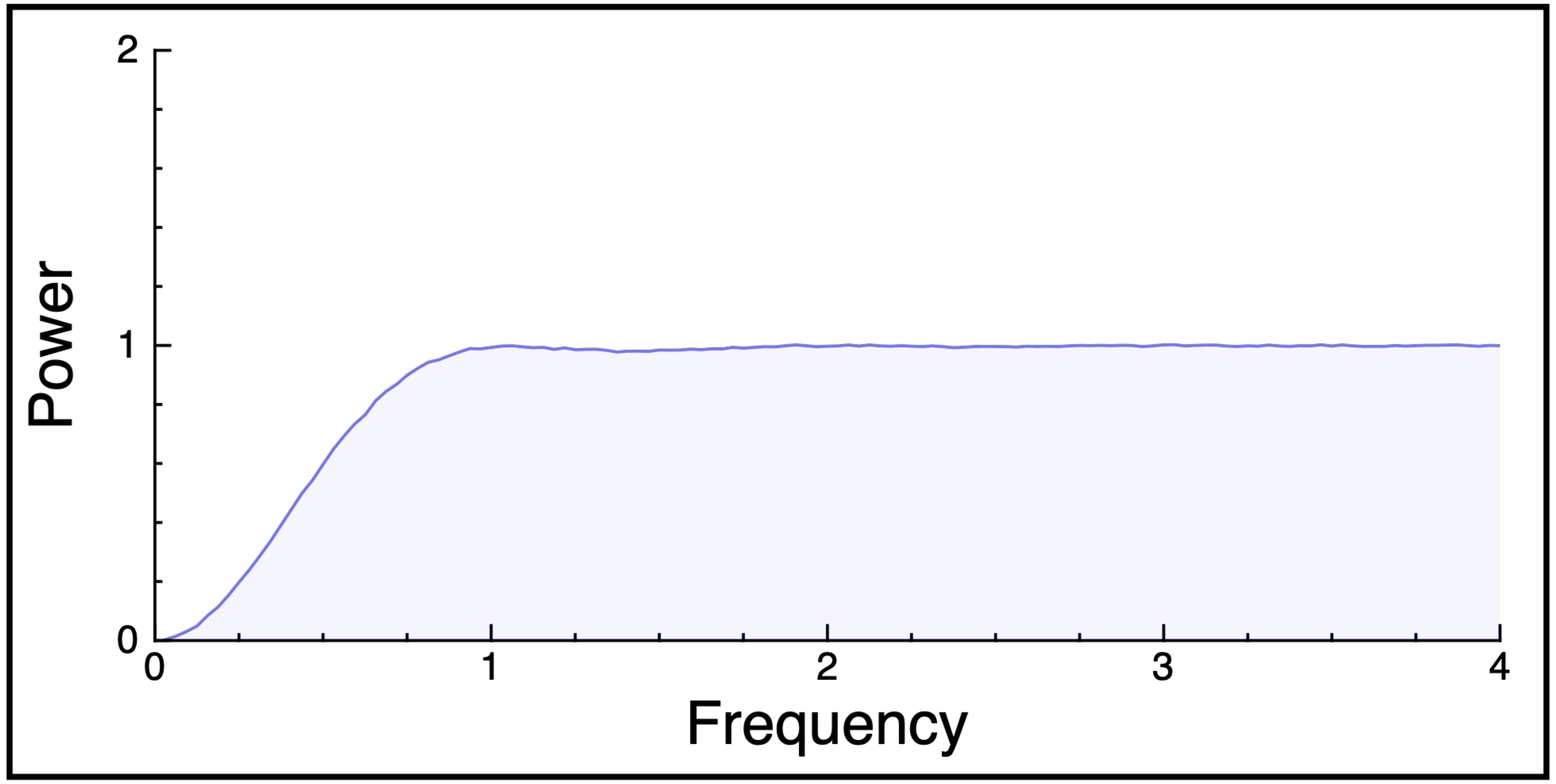
Samples



Expected power spectrum

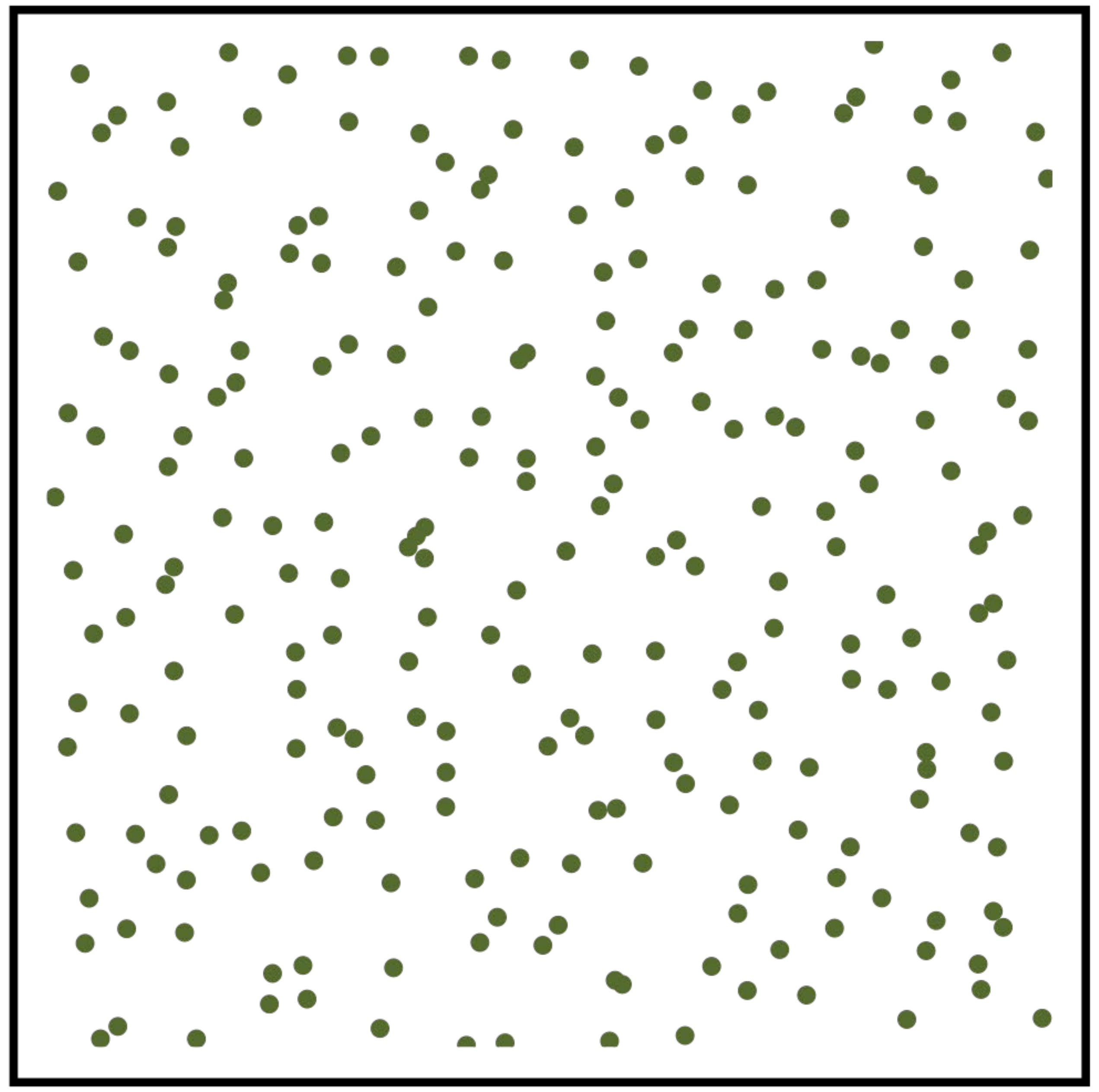


Radial mean

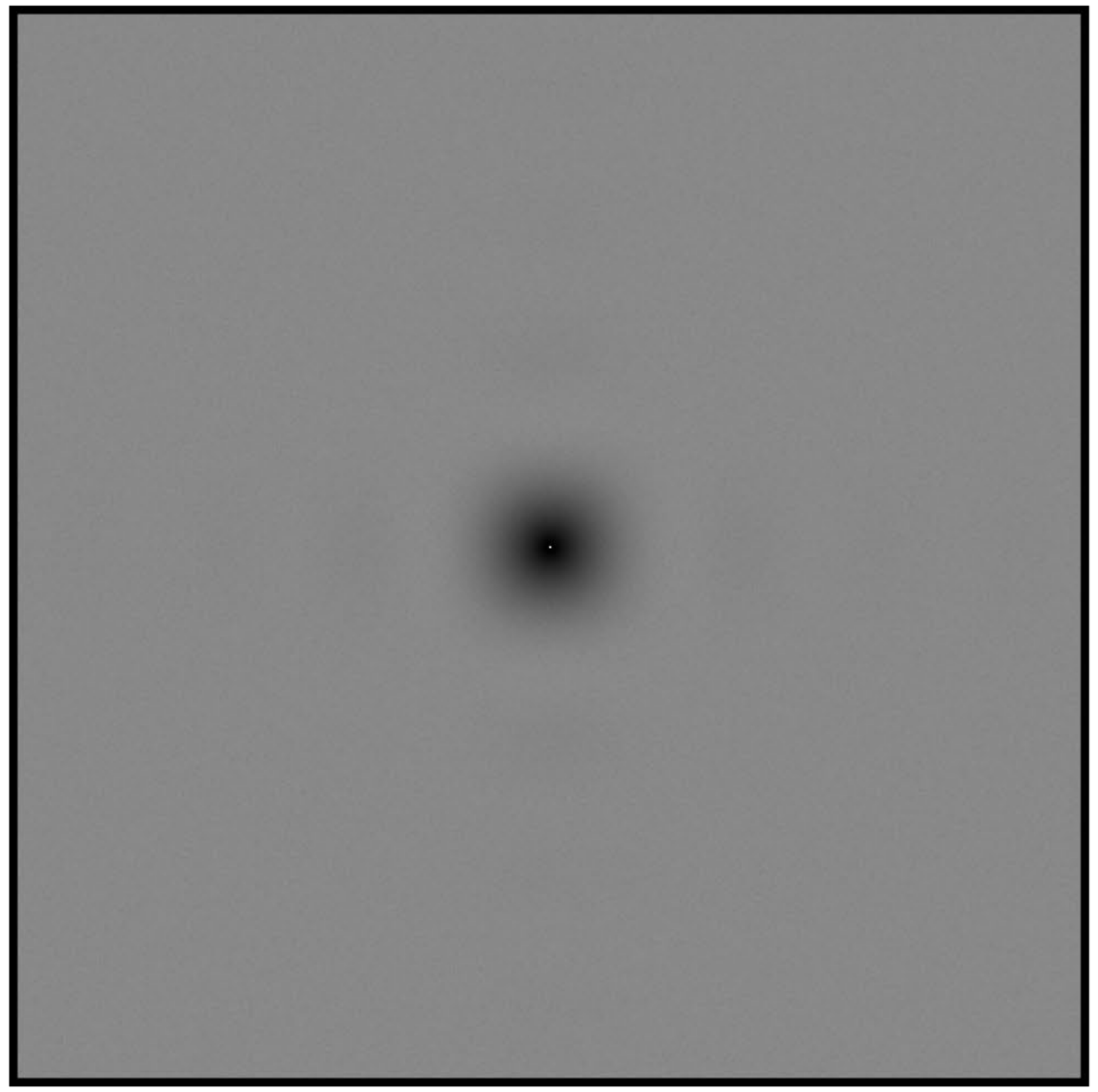


Jittered Sampling

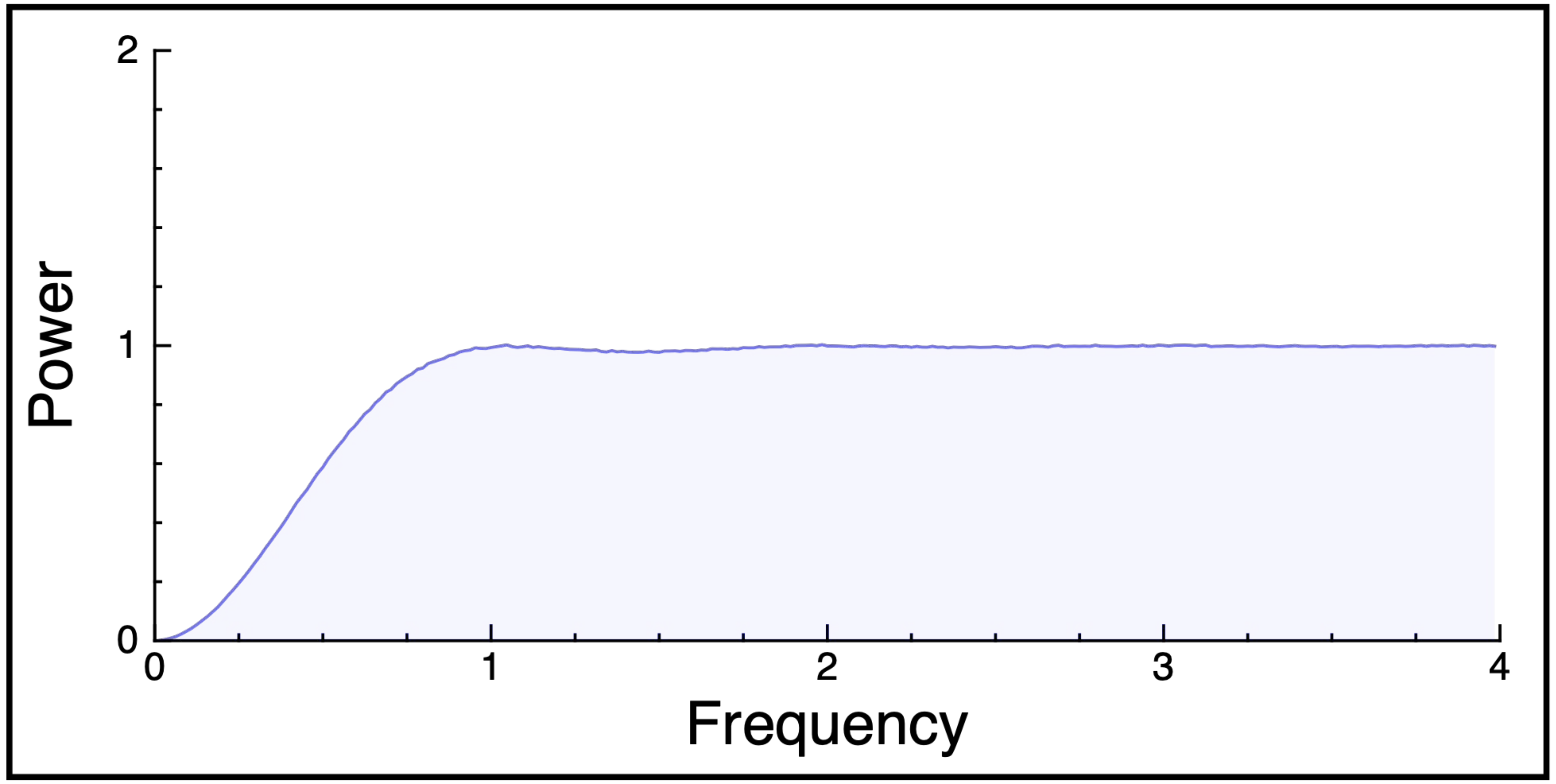
Samples



Expected power spectrum



Radial mean



Poisson Disk Sampling