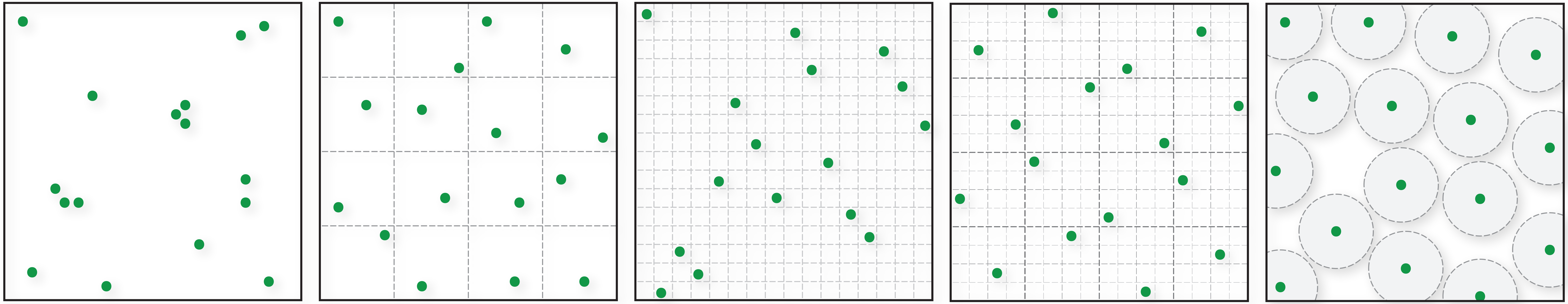
Popular Sampling Patterns

Fourier Analysis of Numerical Integration in Monte Carlo Rendering



Wojciech Jarosz

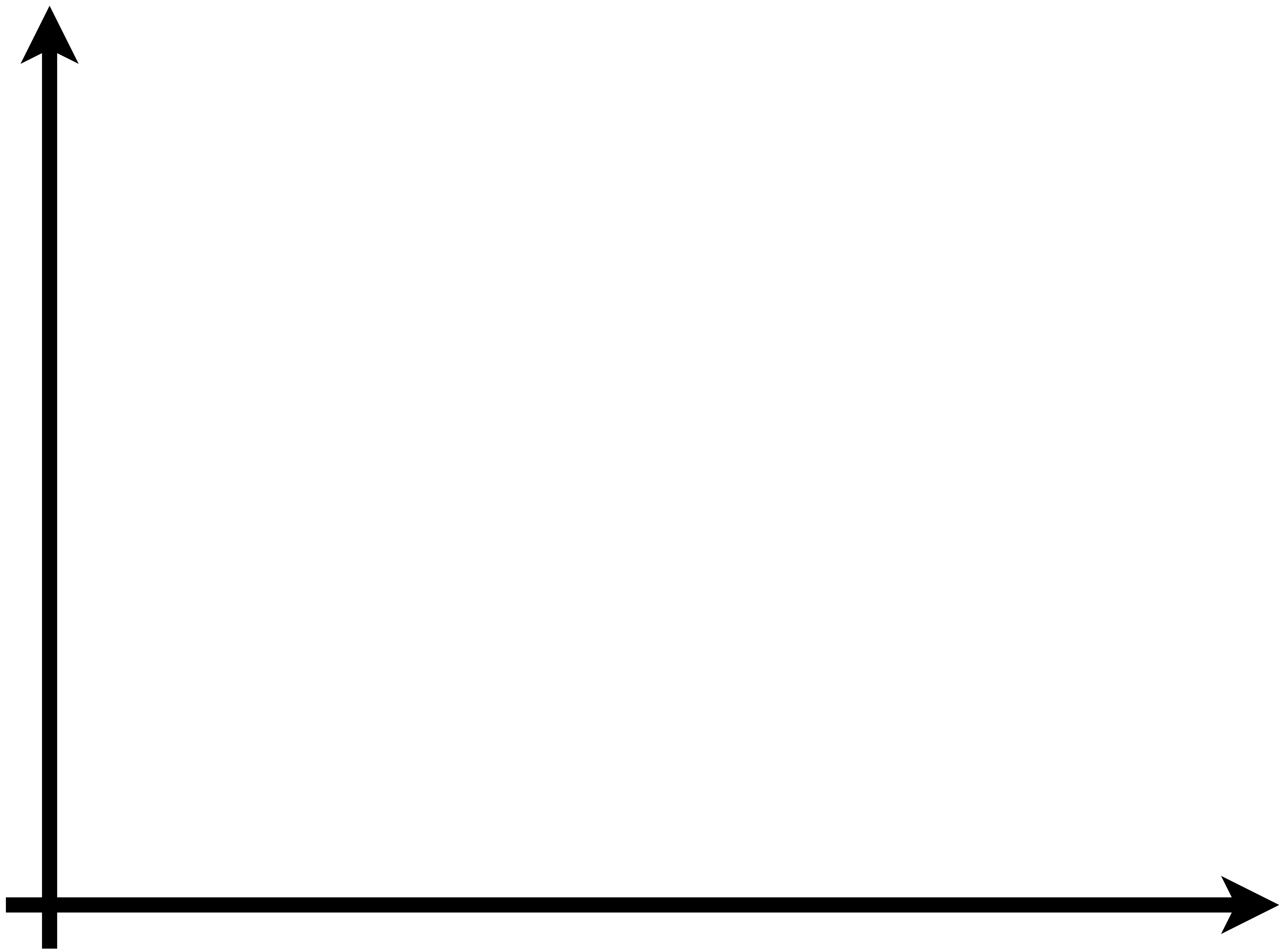
wjarosz@dartmouth.edu

DARTMOUTH
VISUAL COMPUTING LAB

Render the Possibilities
SIGGRAPH2016

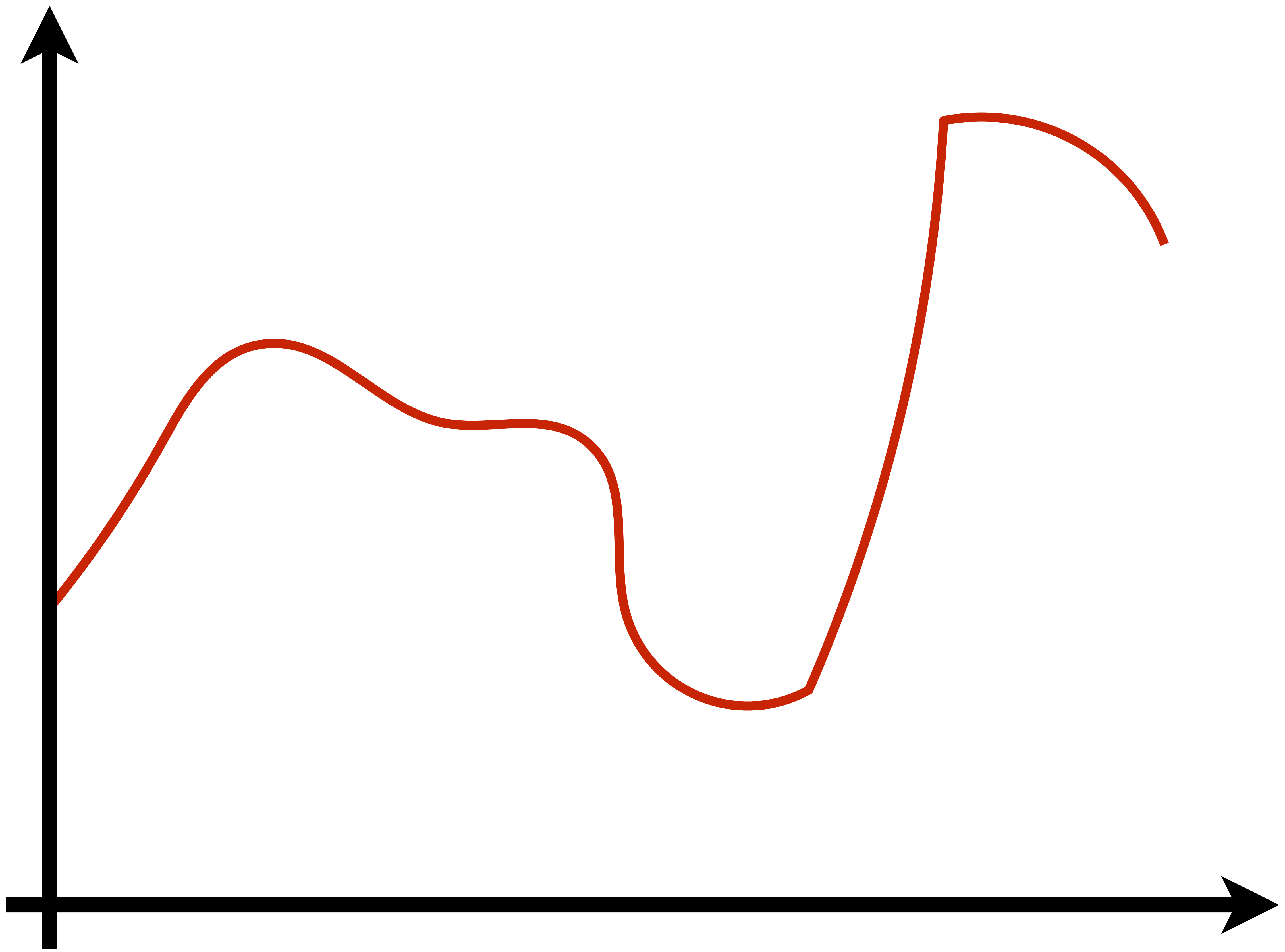
Recall: Monte Carlo Integration

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



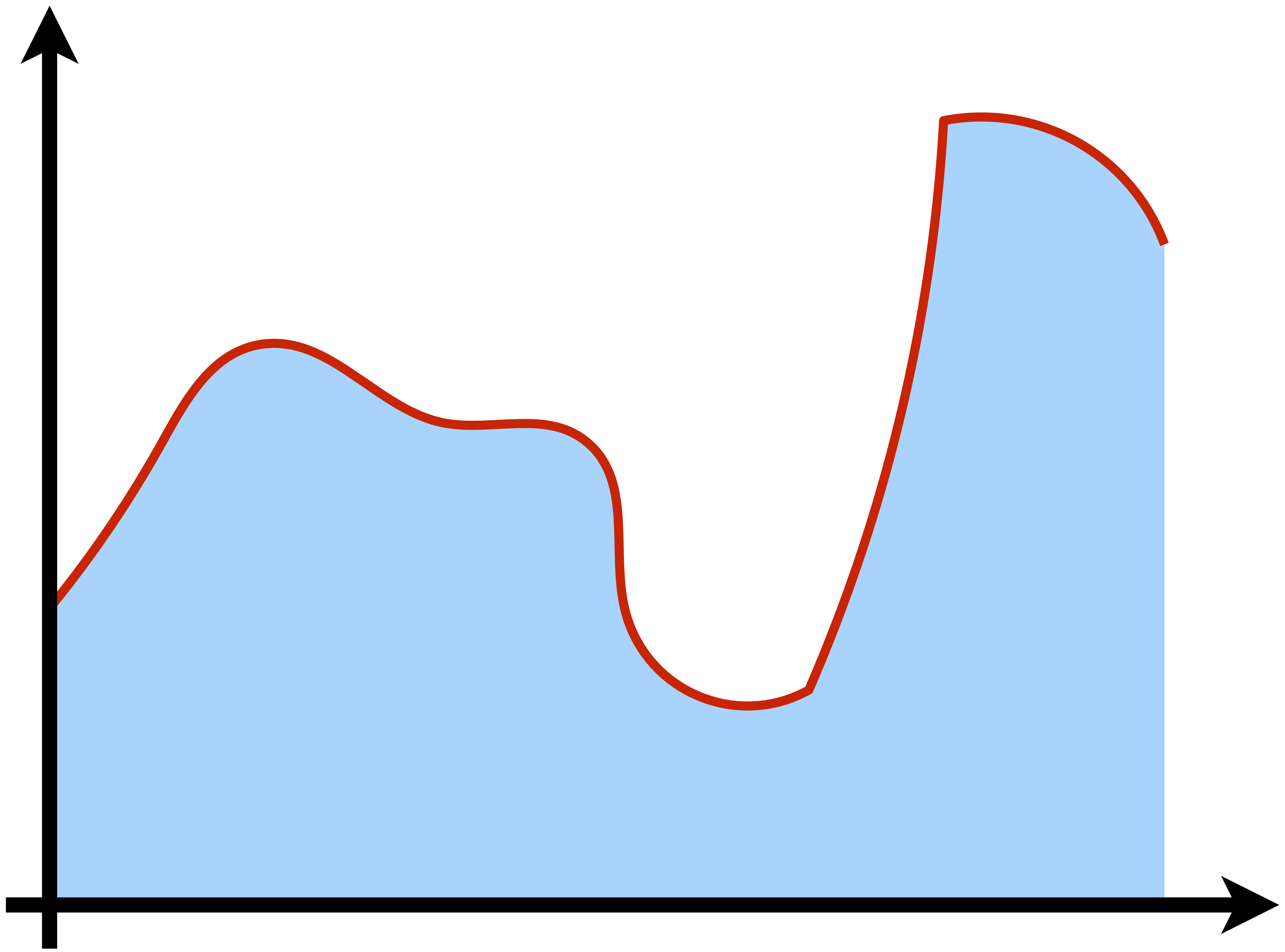
Recall: Monte Carlo Integration

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



Recall: Monte Carlo Integration

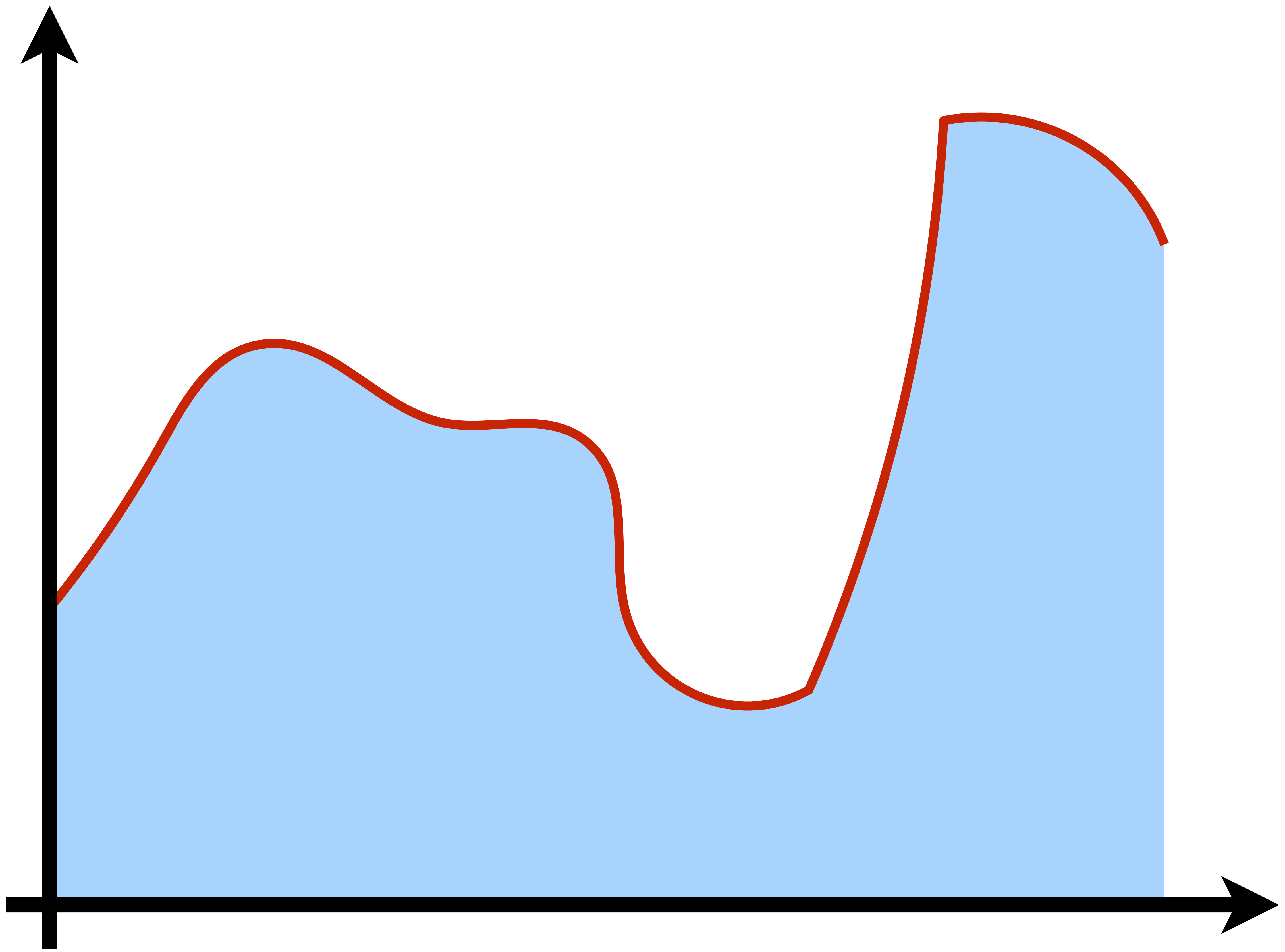
$$I = \int_D f(x)\, \mathrm{d}x$$



Recall: Monte Carlo Integration

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

$$\approx \int_D f(x)\,\mathbf{S}(x)\,\mathrm{d}x$$

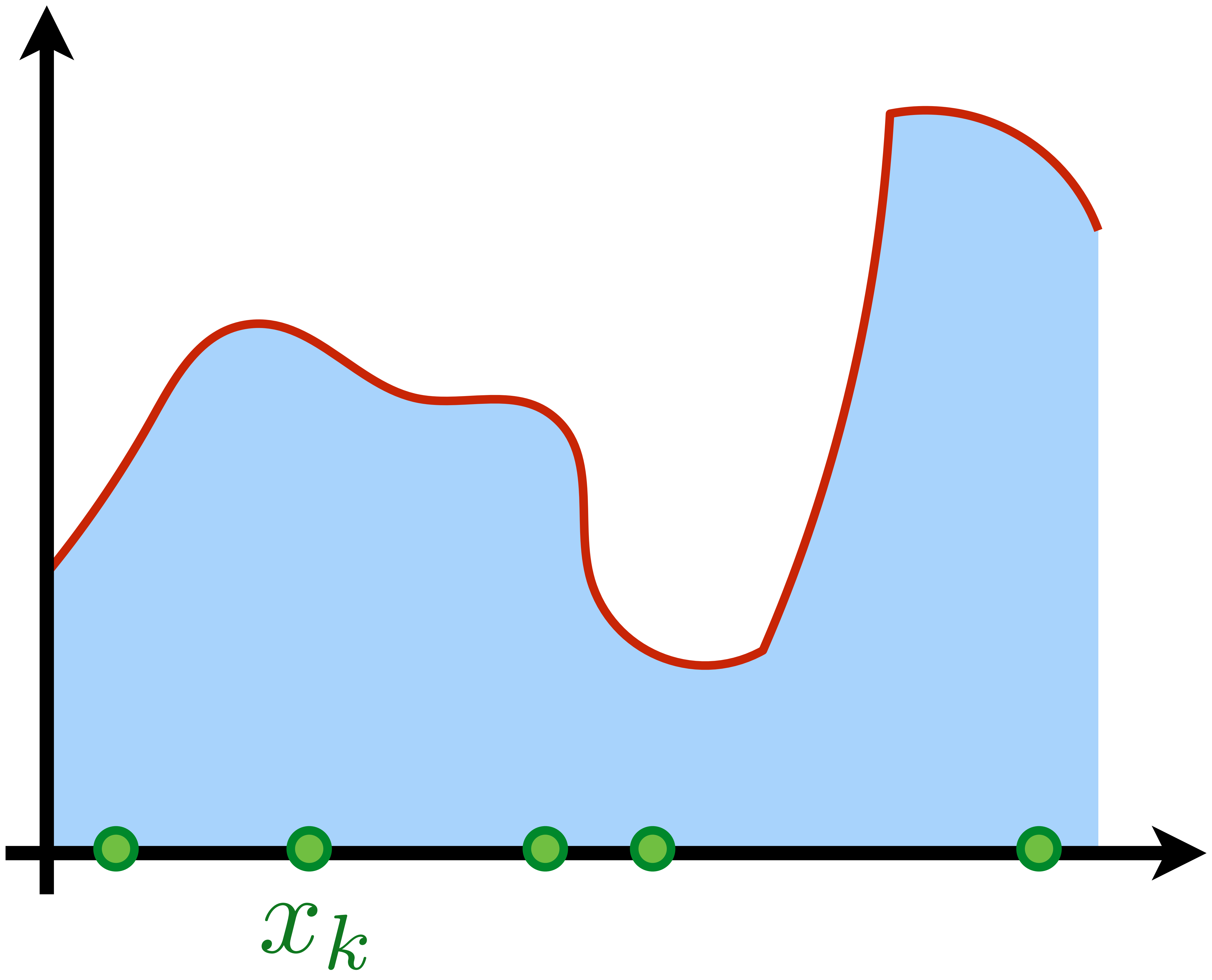


Recall: Monte Carlo Integration

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

$$\approx \int_D f(x)\, \mathbf{S}(x)\, \mathrm{d}x$$

$$\mathbf{S}(x) = \frac{1}{N} \sum_{k=1}^{N} \delta(x - x_k)$$

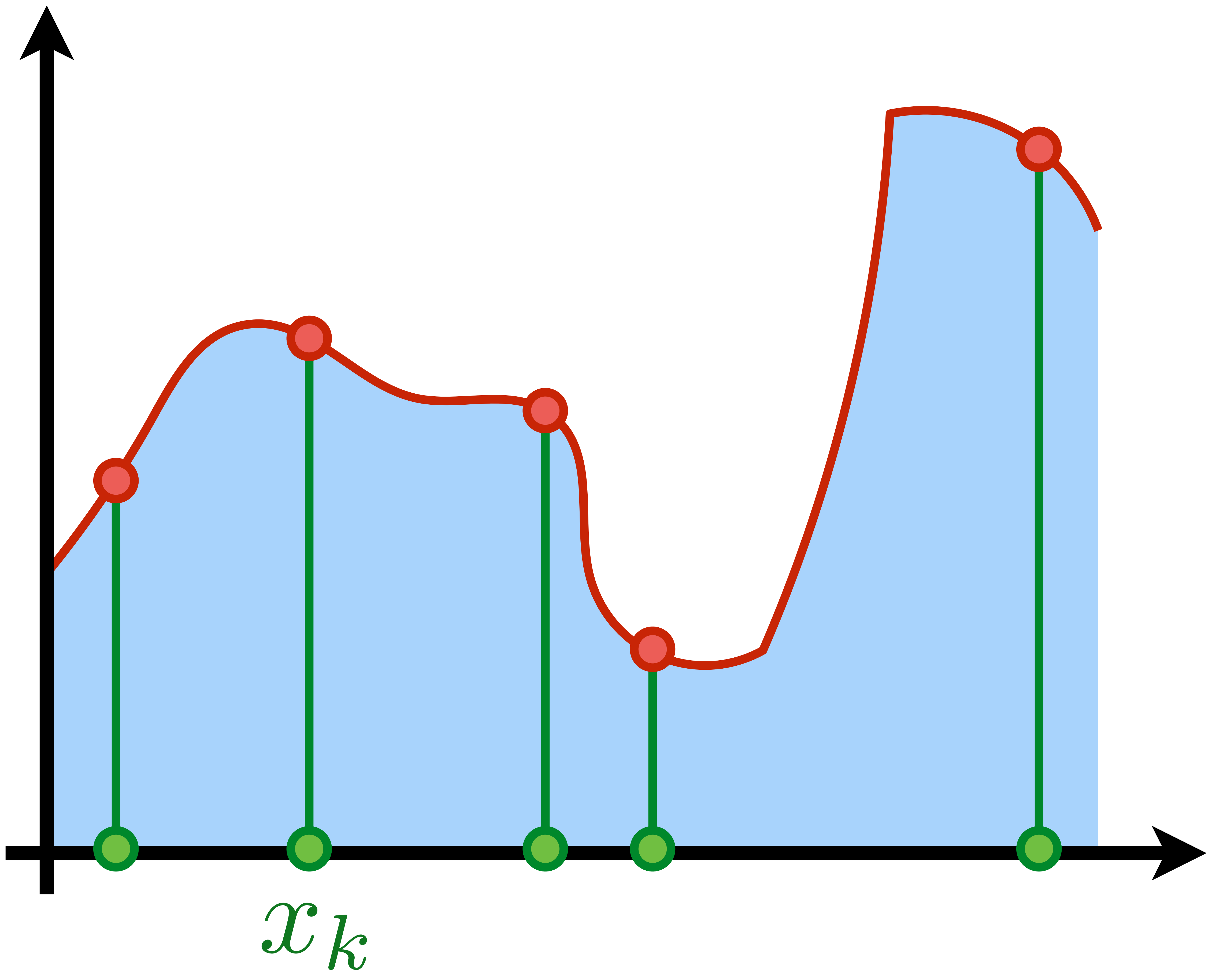


Recall: Monte Carlo Integration

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

$$\approx \int_D f(x)\, \mathbf{S}(x)\, \mathrm{d}x$$

$$\mathbf{S}(x) = \frac{1}{N} \sum_{k=1}^{N} \delta(x - x_k)$$



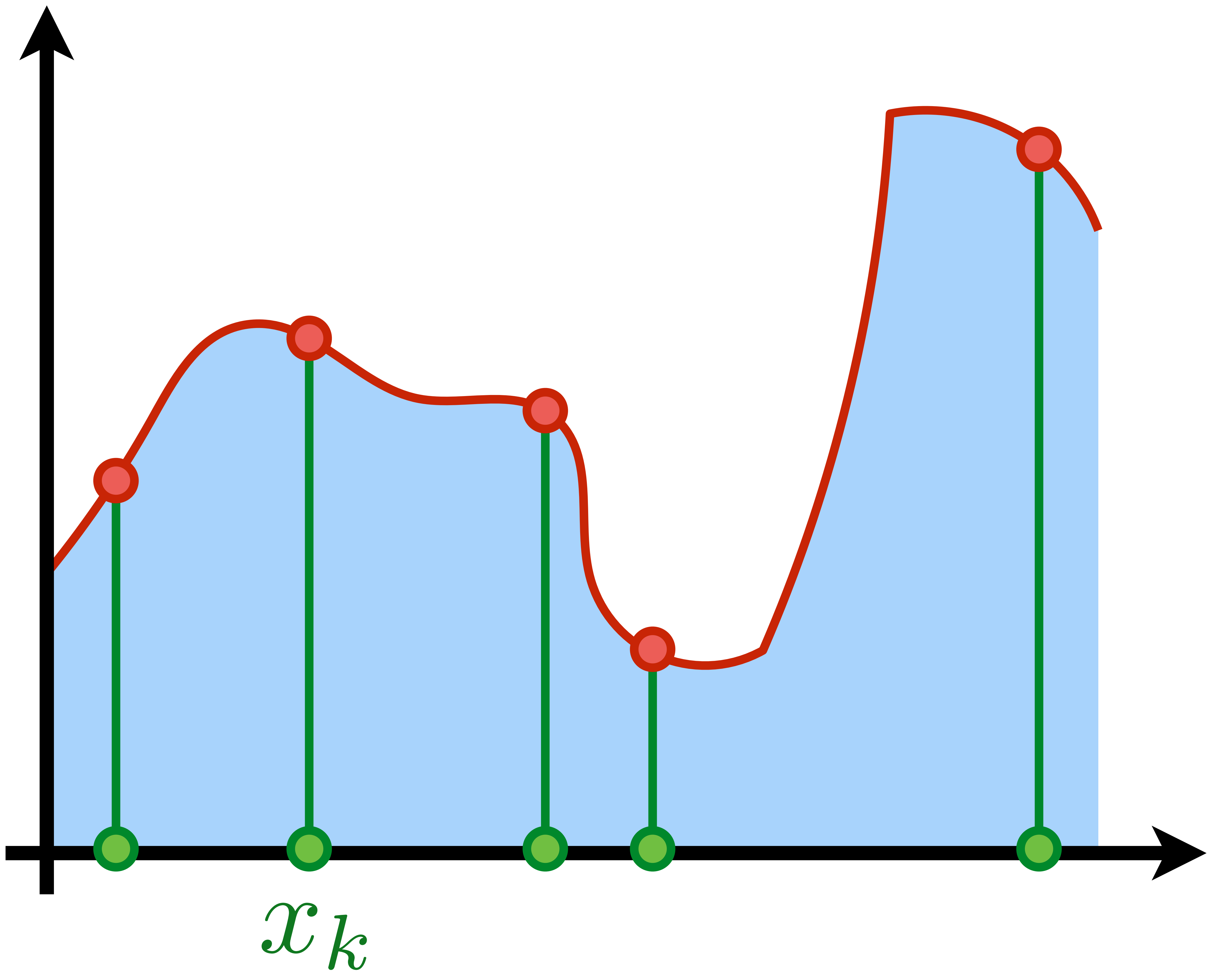
Recall: Monte Carlo Integration

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

$$\approx \int_D f(x)\,\mathbf{S}(x)\,\mathrm{d}x$$

$$\mathbf{S}(x) = \frac{1}{N}\sum_{k=1}^{N} \delta(x - x_k)$$

How to generate the locations x_k?

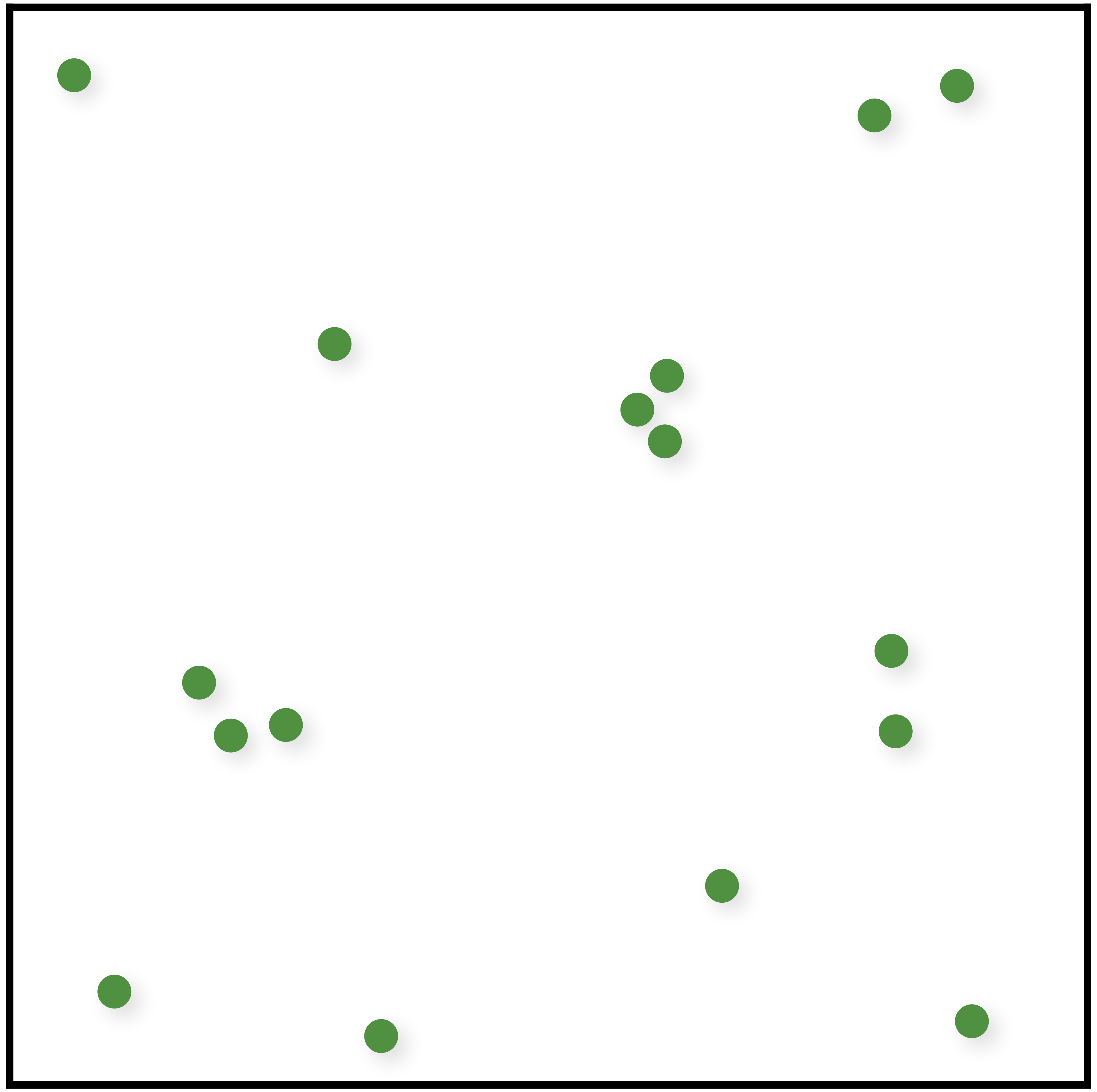


Independent Random Sampling

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

Independent Random Sampling

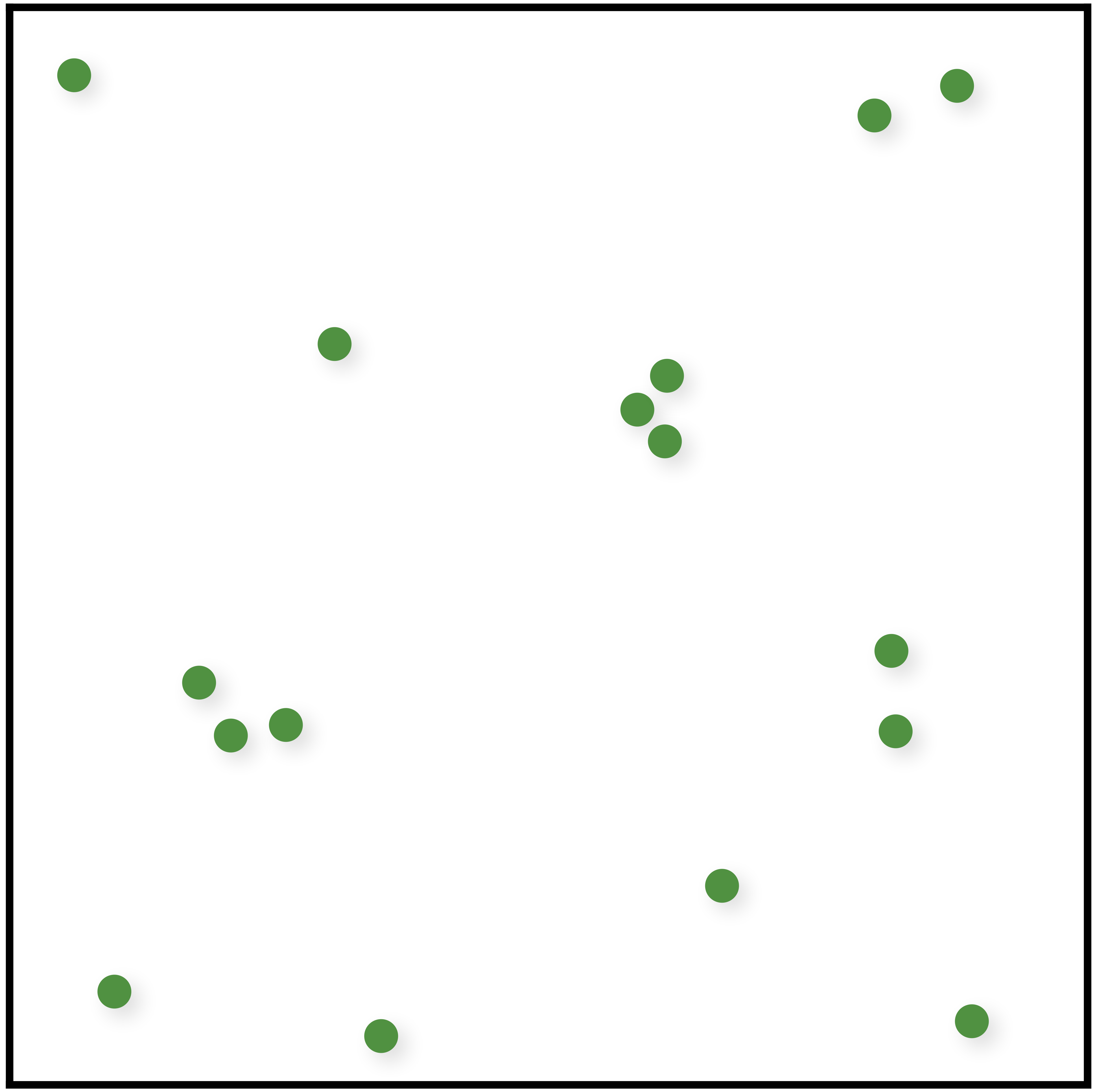
```
for (int k = 0; k < num; k++)
{
    samples(k).x = randf();
    samples(k).y = randf();
}
```



Independent Random Sampling

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

✔ Trivially extends to higher dimensions

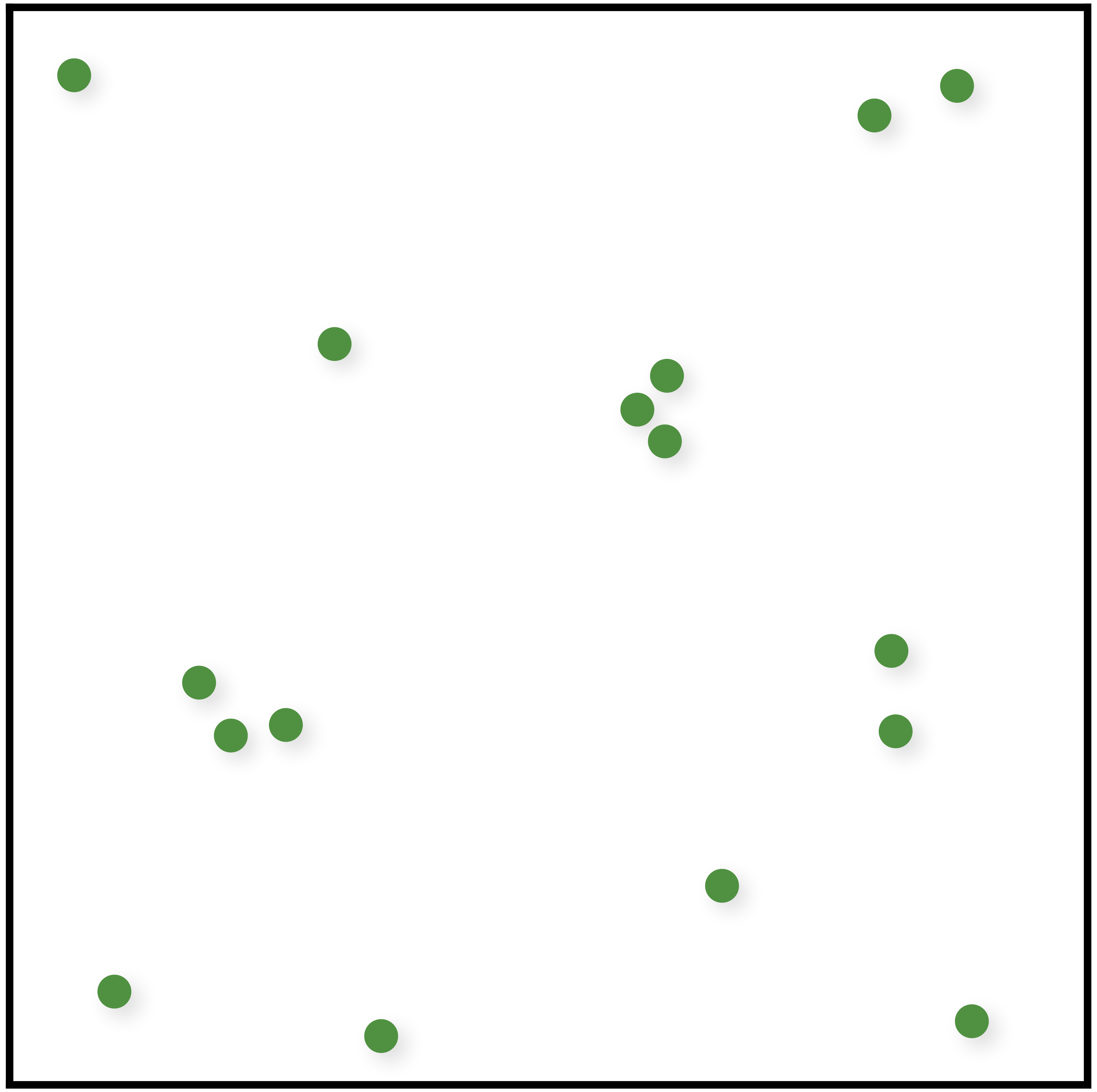


Independent Random Sampling

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

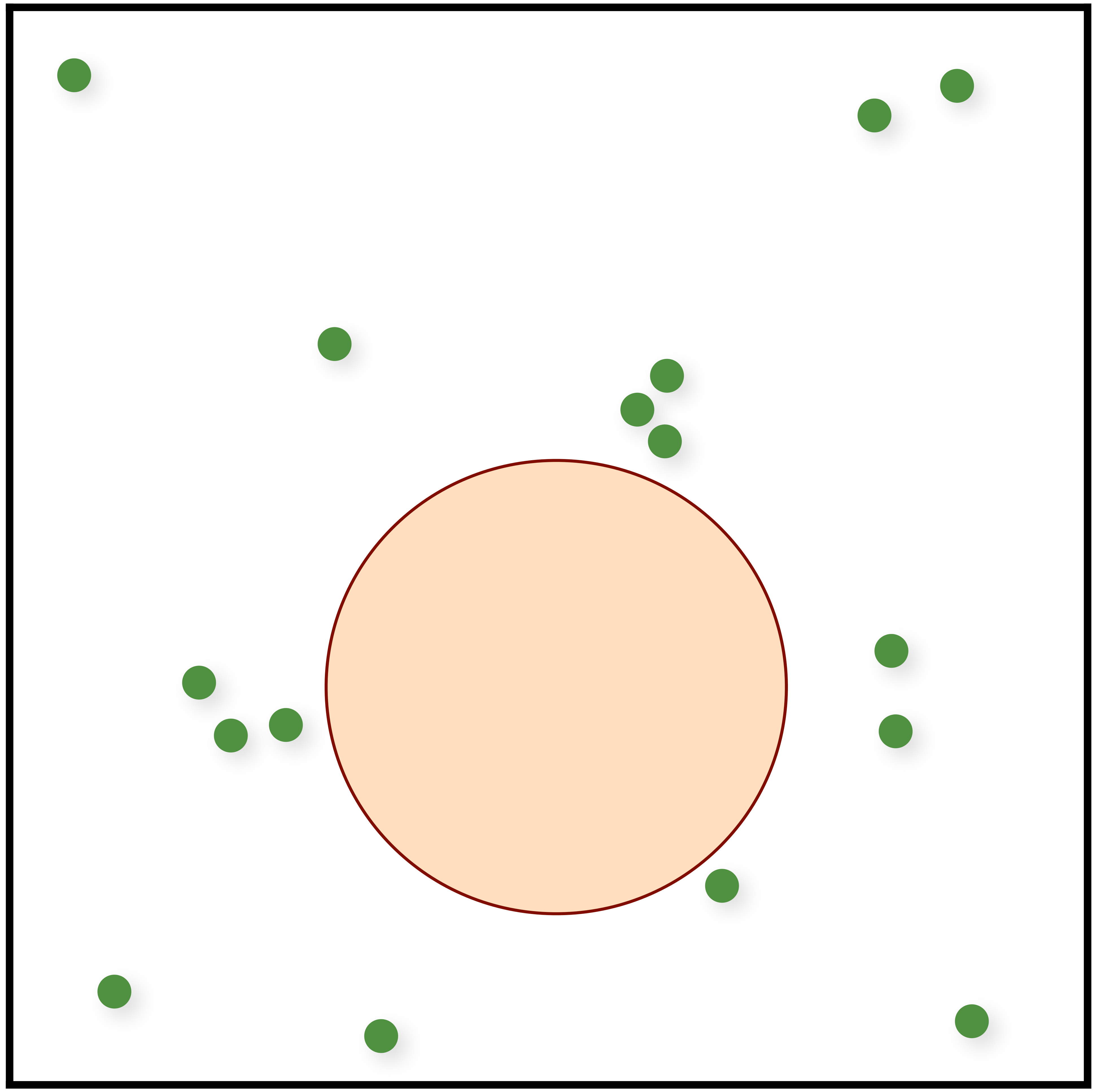
✔ Trivially extends to higher dimensions

✔ Trivially progressive and memory-less



Independent Random Sampling

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



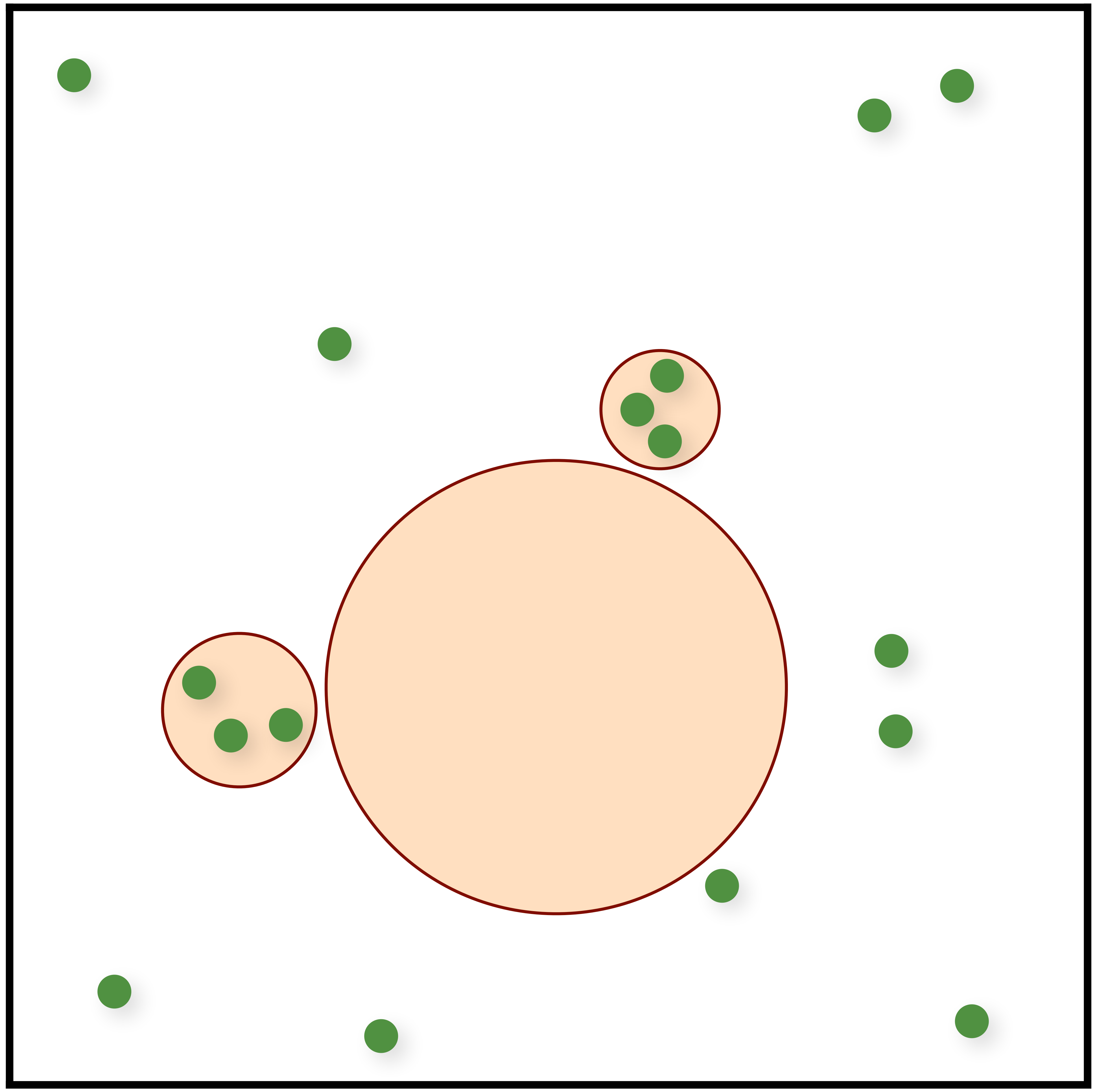
✔ Trivially extends to higher dimensions

✔ Trivially progressive and memory-less

✘ Big gaps

Independent Random Sampling

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



✔ Trivially extends to higher dimensions

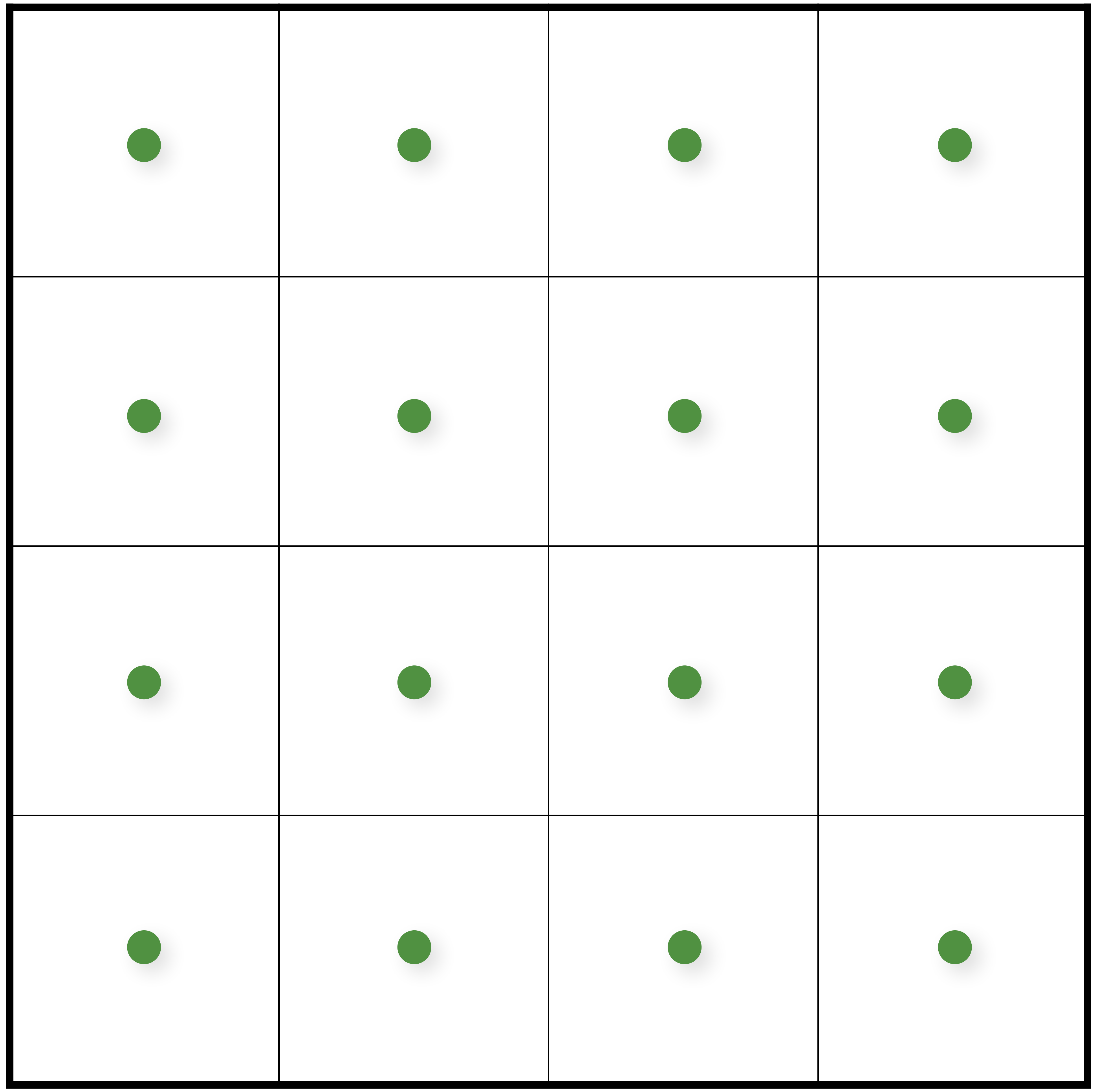
✔ Trivially progressive and memory-less

✘ Big gaps

✘ Clumping

Regular Sampling

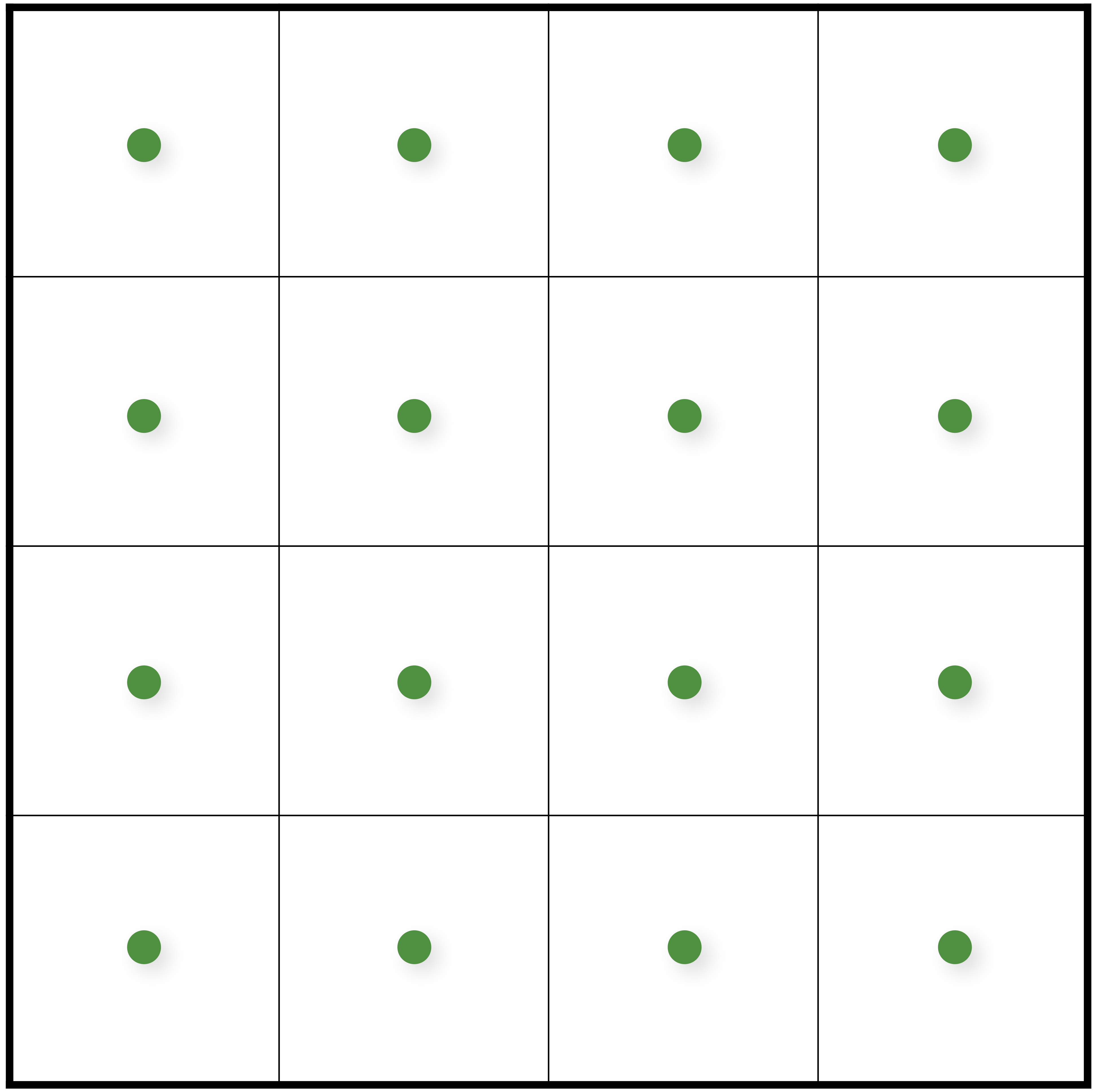
```
for (uint i = 0; i < numX; i++)
    for (uint j = 0; j < numY; j++)
    {
        samples(i,j).x = (i + 0.5)/numX;
        samples(i,j).y = (j + 0.5)/numY;
    }
```



✔ Extends to higher dimensions, but…

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;
    }
```

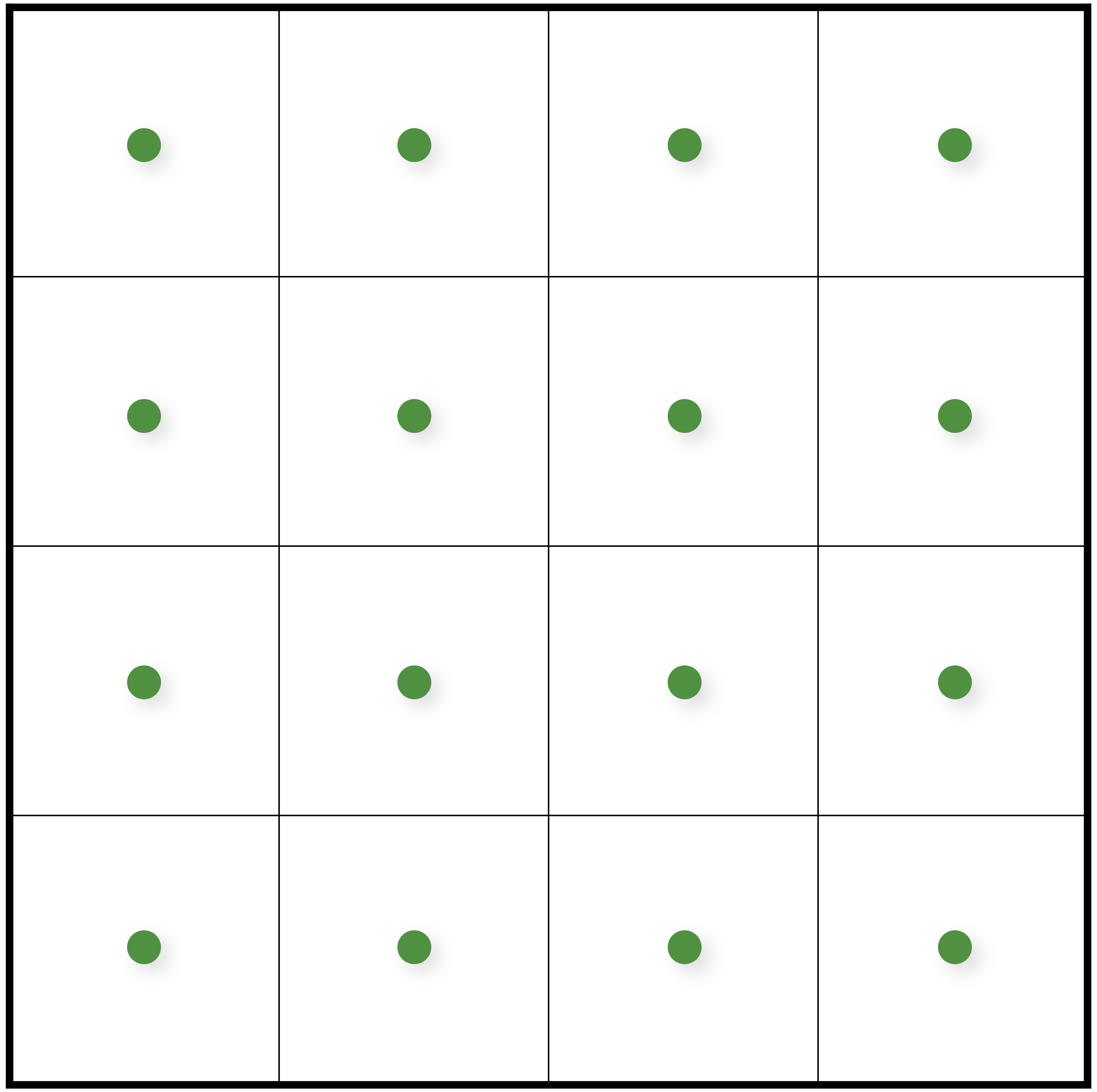


✔ Extends to higher dimensions, but…

✘ Curse of dimensionality

Regular Sampling

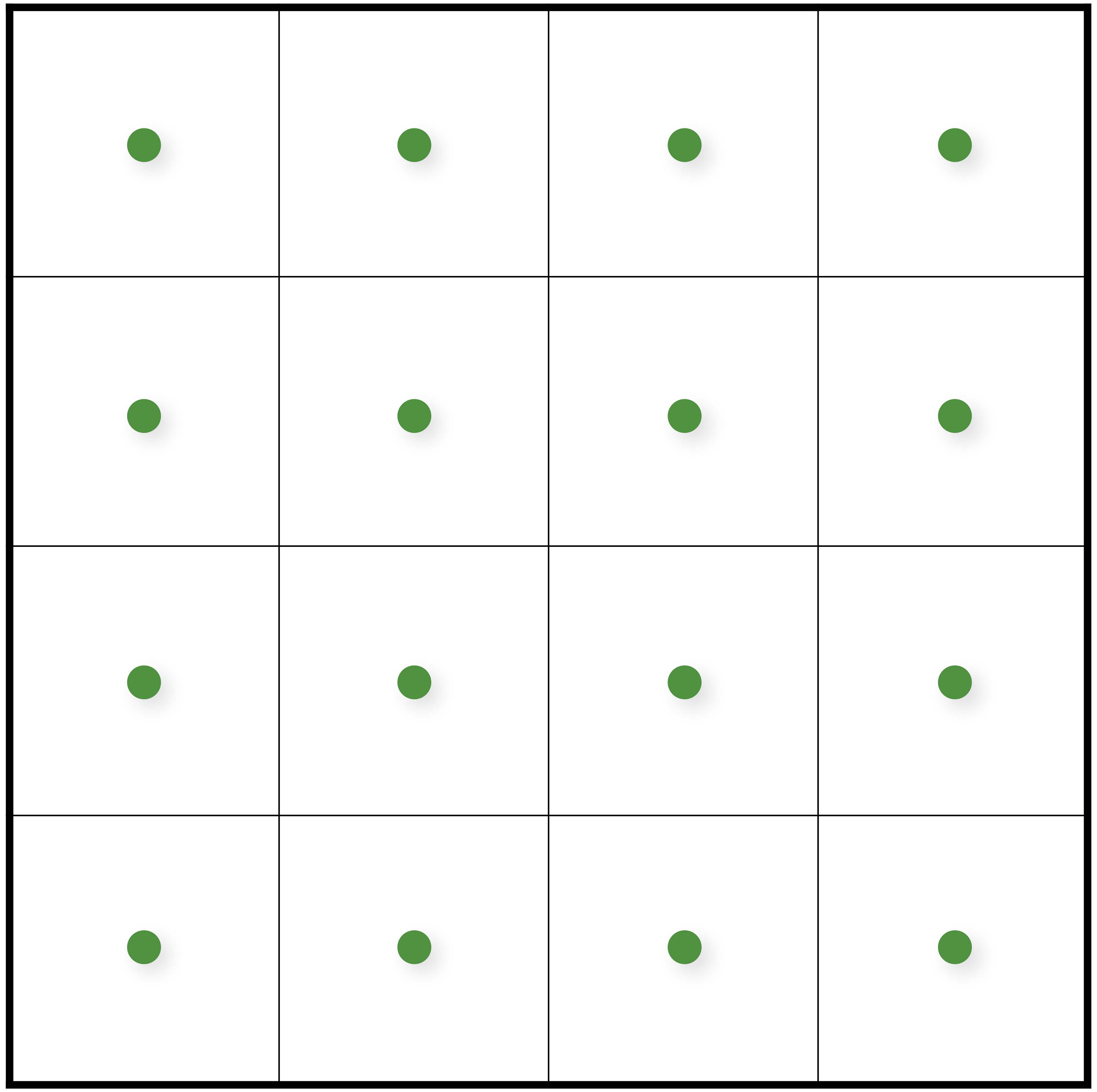
```
for (uint i = 0; i < numX; i++)
    for (uint j = 0; j < numY; j++)
    {
        samples(i,j).x = (i + 0.5)/numX;
        samples(i,j).y = (j + 0.5)/numY;
    }
```



- ✔ Extends to higher dimensions, but…
- ✘ Curse of dimensionality
- ✘ Aliasing

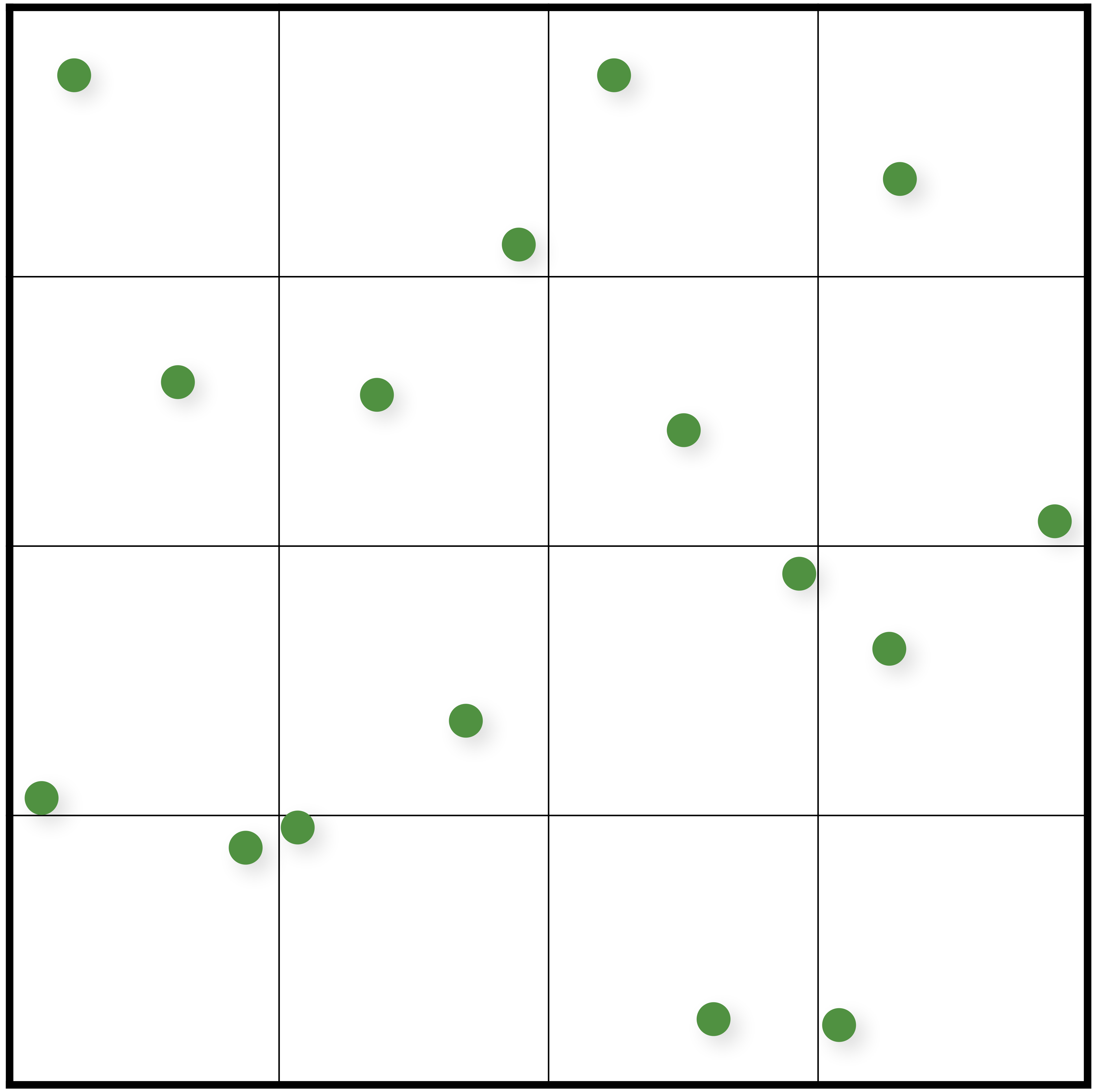
Regular Sampling

```
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;
    }
```



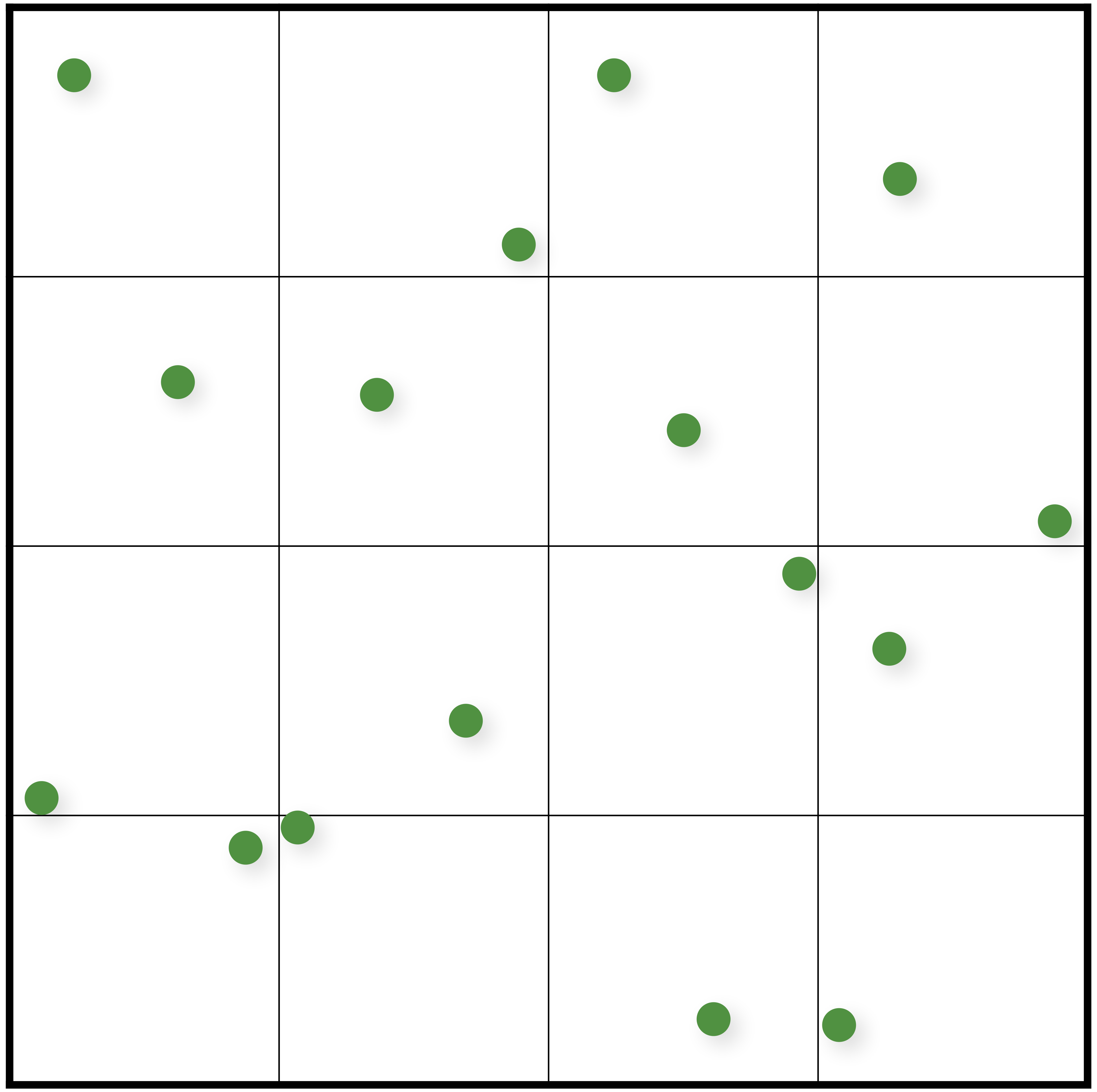
Jittered/Stratified Sampling

```
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;
    }
```



Jittered/Stratified Sampling

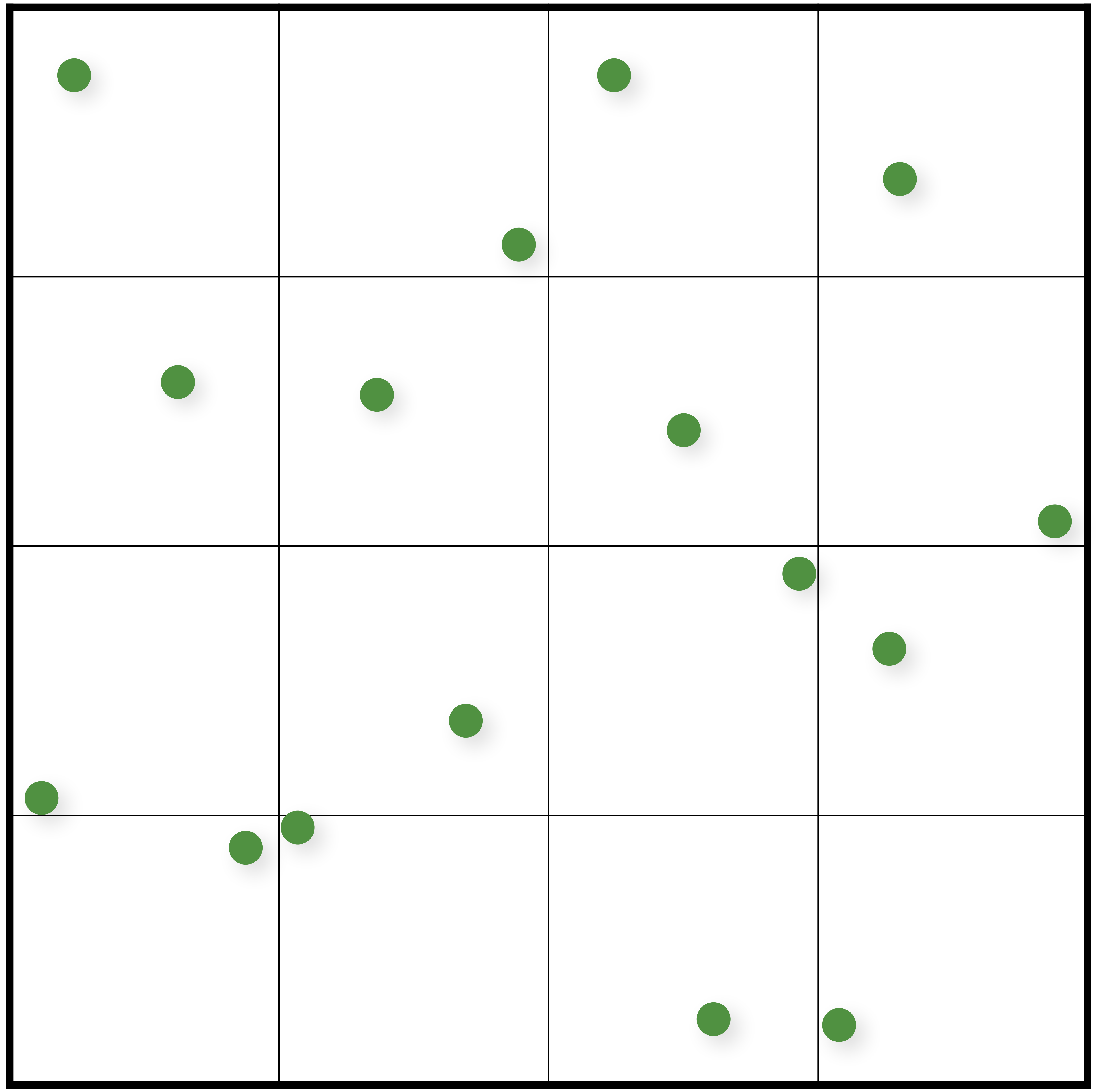
```
for (uint i = 0; i < numX; i++)
    for (uint j = 0; j < numY; j++)
    {
        samples(i,j).x = (i + randf())/numX;
        samples(i,j).y = (j + randf())/numY;
    }
```



✔ Provably cannot increase variance

Jittered/Stratified Sampling

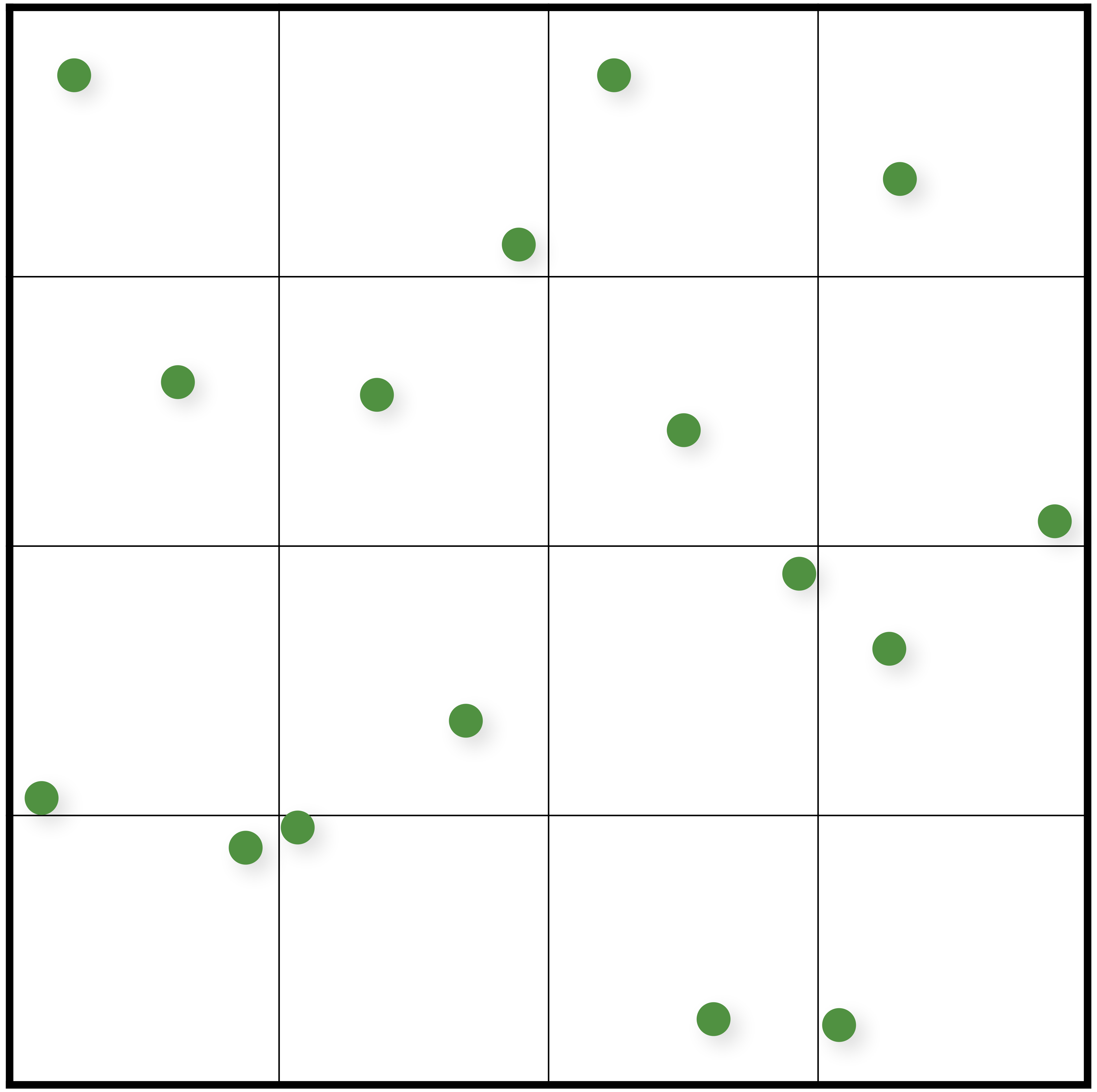
```
for (uint i = 0; i < numX; i++)
    for (uint j = 0; j < numY; j++)
    {
        samples(i,j).x = (i + randf())/numX;
        samples(i,j).y = (j + randf())/numY;
    }
```



- ✔ Provably cannot increase variance
- ✔ Extends to higher dimensions, but…

Jittered/Stratified Sampling

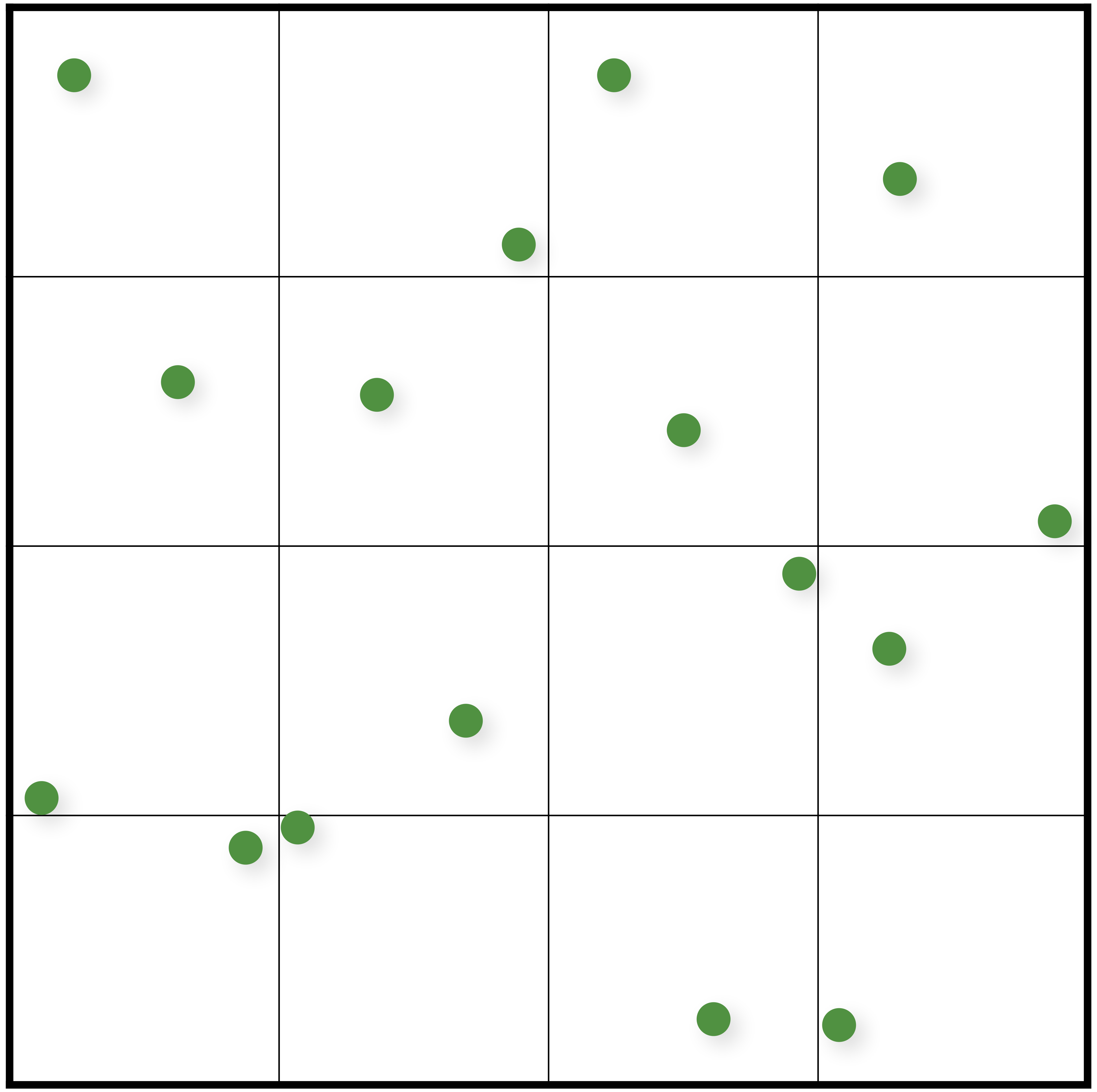
```
for (uint i = 0; i < numX; i++)
    for (uint j = 0; j < numY; j++)
    {
        samples(i,j).x = (i + randf())/numX;
        samples(i,j).y = (j + randf())/numY;
    }
```



- ✔ Provably cannot increase variance
- ✔ Extends to higher dimensions, but…
- ✘ Curse of dimensionality

Jittered/Stratified Sampling

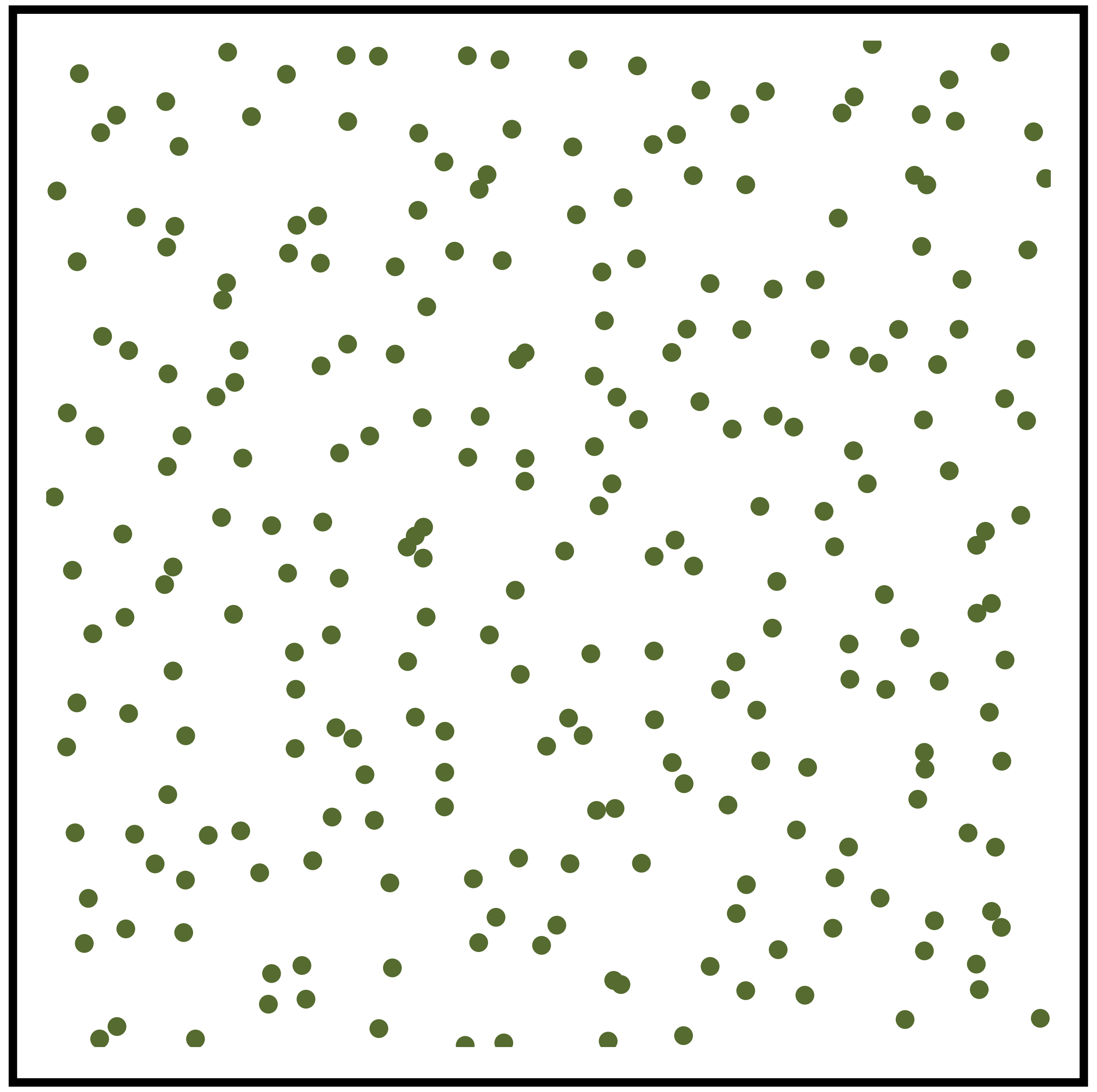
```
for (uint i = 0; i < numX; i++)
    for (uint j = 0; j < numY; j++)
    {
        samples(i,j).x = (i + randf())/numX;
        samples(i,j).y = (j + randf())/numY;
    }
```



- ✔ Provably cannot increase variance
- ✔ Extends to higher dimensions, but…
- ✘ Curse of dimensionality
- ✘ Not progressive

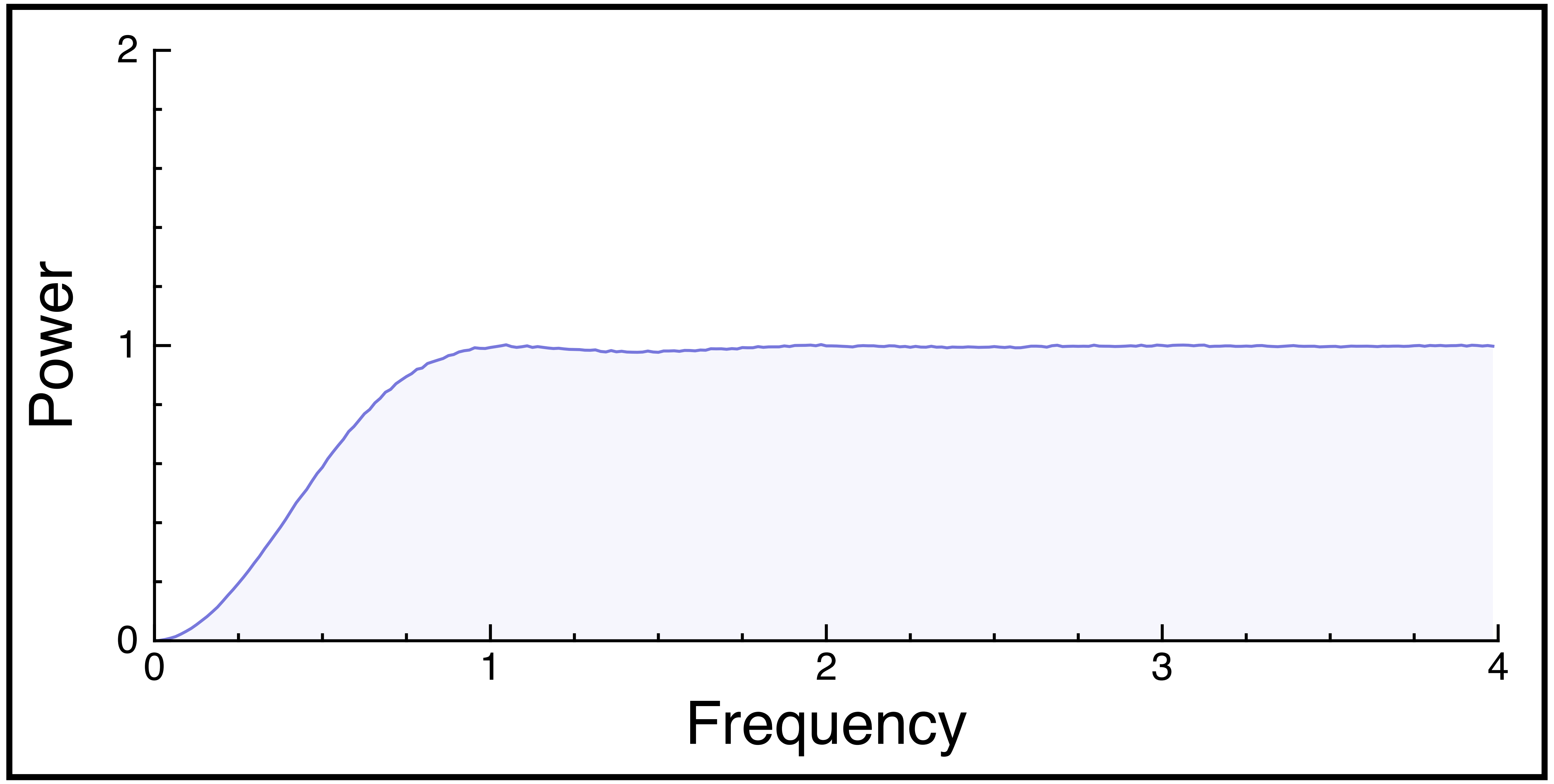
Jittered Sampling

Samples



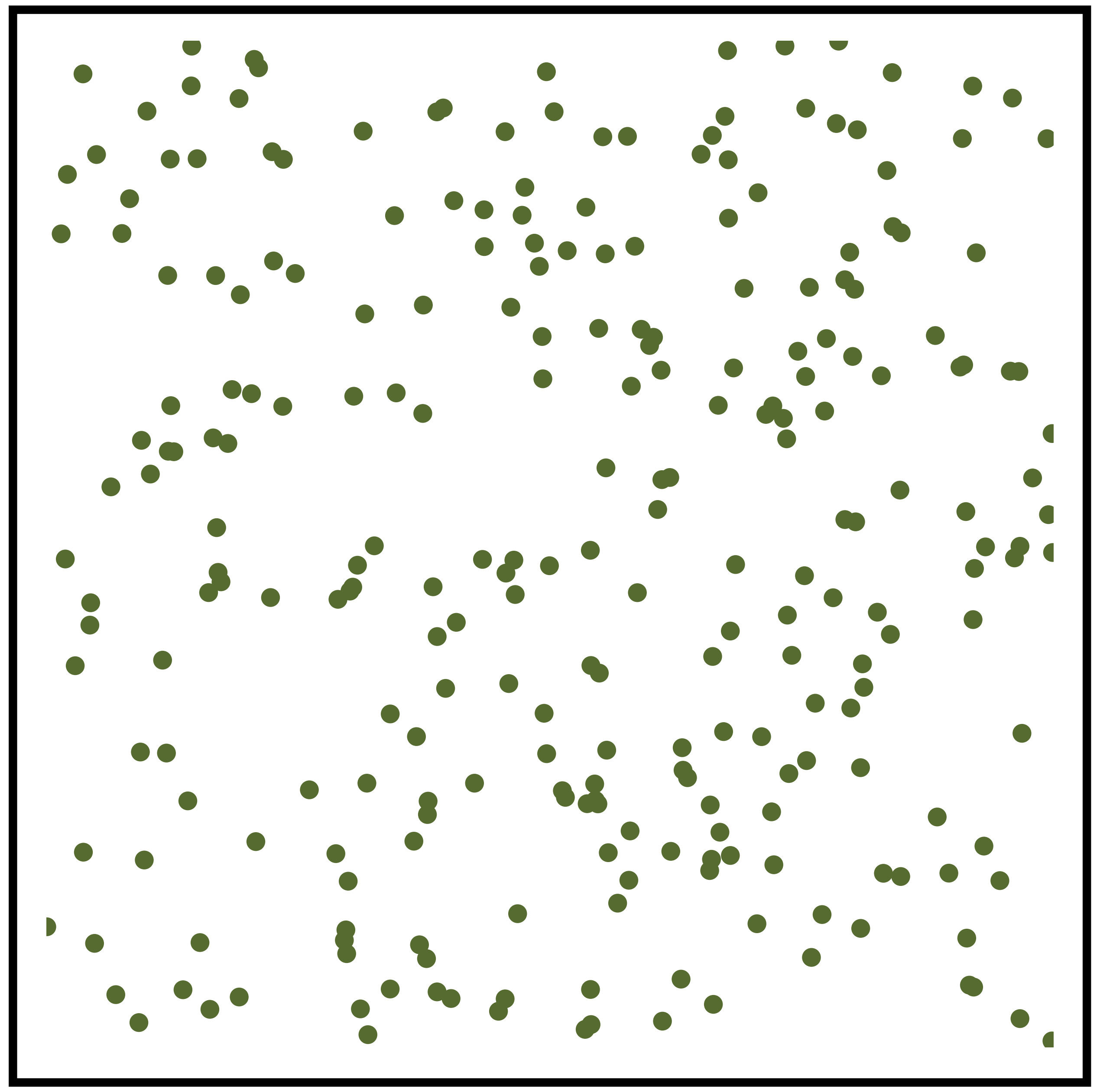
Expected power spectrum

Radial mean



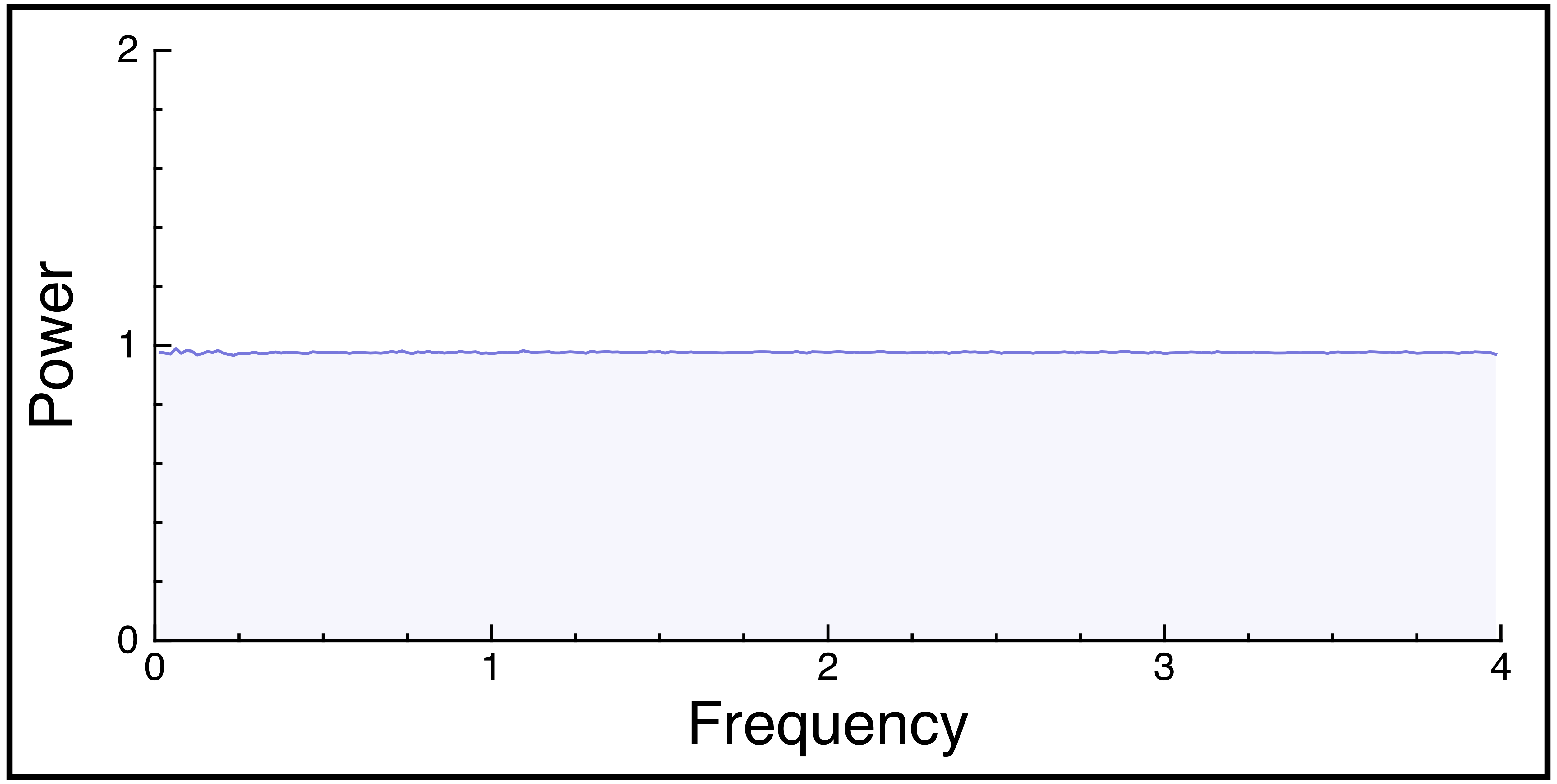
Independent Random Sampling

Samples



Expected power spectrum

Radial mean



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

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!

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

Uncorrelated Jitter [Cook et al. 84]

Uncorrelated Jitter [Cook et al. 84]

Compute stratified samples in sub-dimensions

Uncorrelated Jitter [Cook et al. 84]

Compute stratified samples in sub-dimensions

- 2D jittered (x,y) for pixel

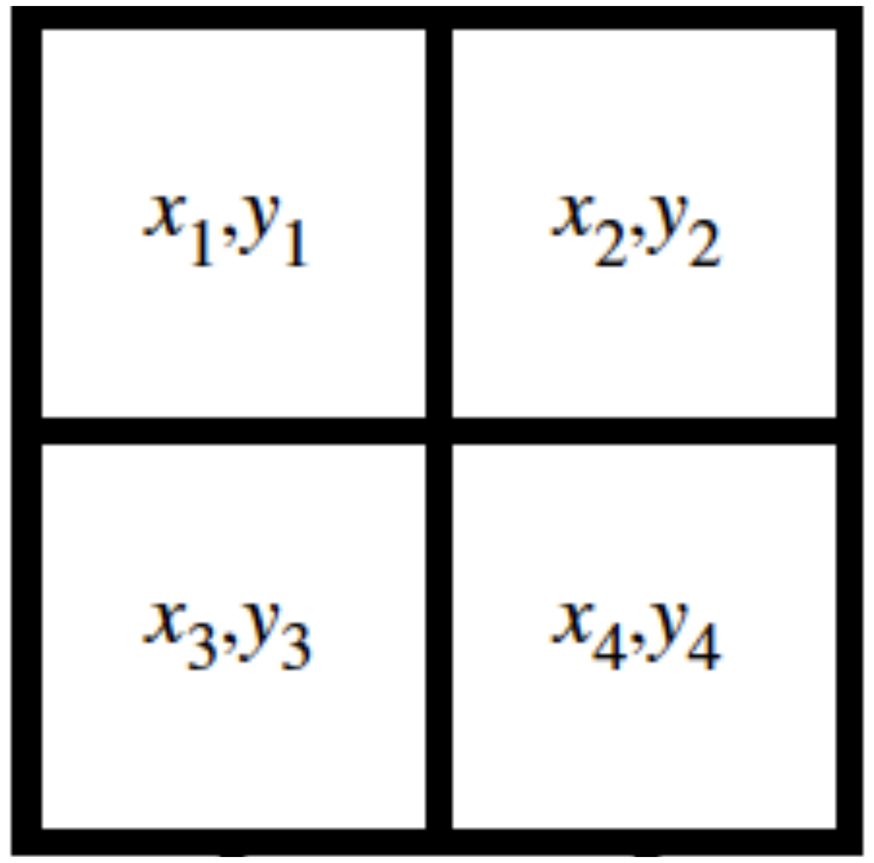
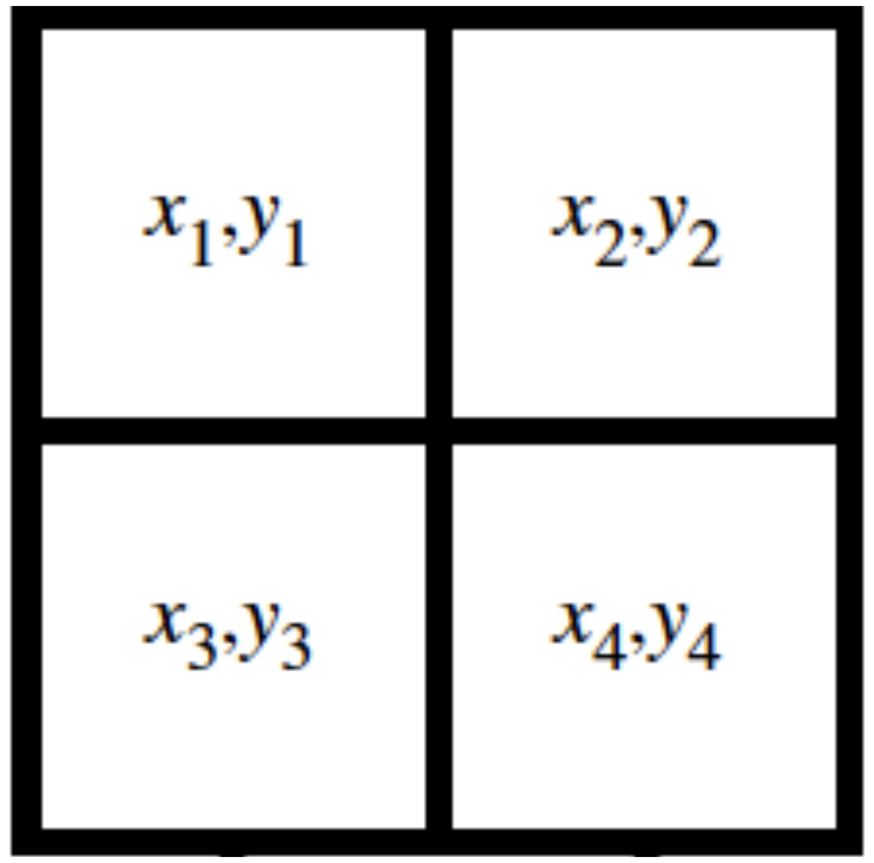


Image source: PBRTe2 [Pharr & Humphreys 2010]

Uncorrelated Jitter [Cook et al. 84]

Compute stratified samples in sub-dimensions

- 2D jittered (x,y) for pixel
- 2D jittered (u,v) for lens



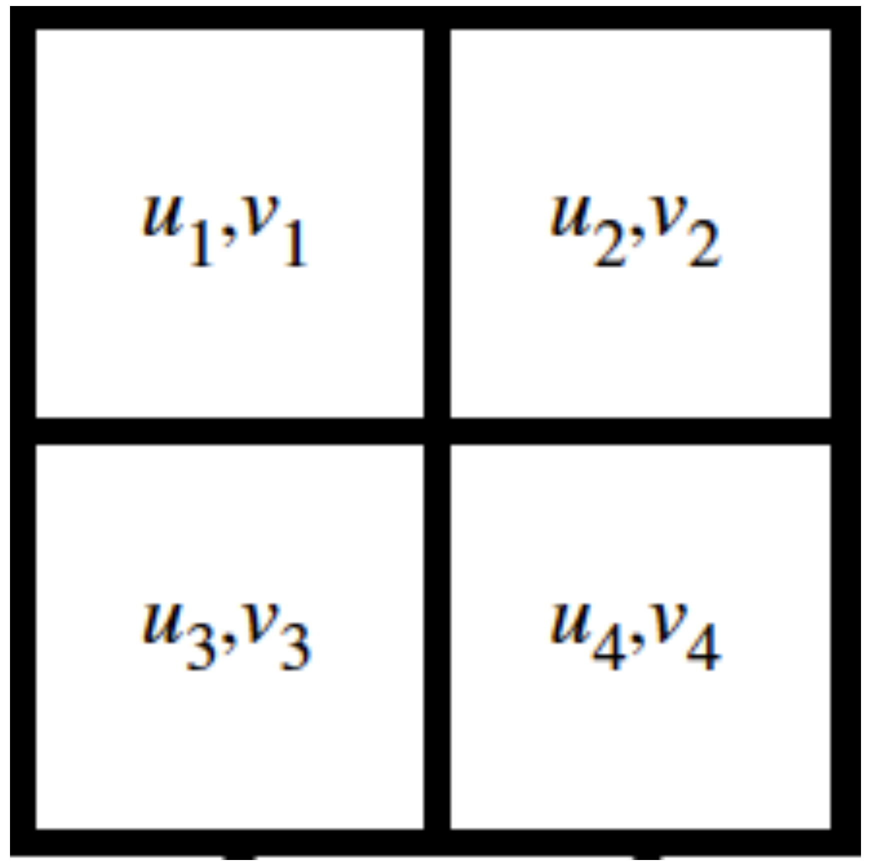


Image source: PBRTe2 [Pharr & Humphreys 2010]

Uncorrelated Jitter [Cook et al. 84]

Compute stratified samples in sub-dimensions

- 2D jittered (x,y) for pixel
- 2D jittered (u,v) for lens
- 1D jittered (t) for time

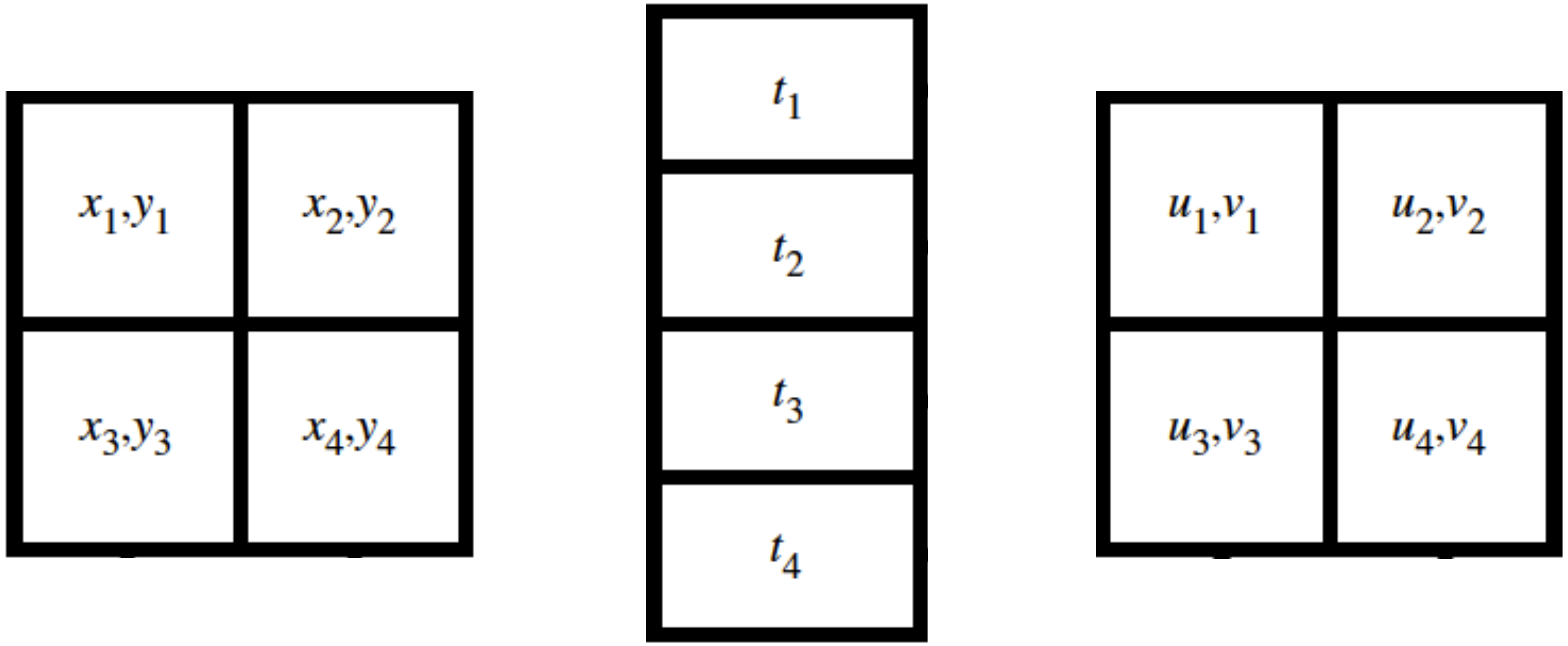


Image source: PBRTe2 [Pharr & Humphreys 2010]

Uncorrelated Jitter [Cook et al. 84]

Compute stratified samples in sub-dimensions

- 2D jittered (x,y) for pixel
- 2D jittered (u,v) for lens
- 1D jittered (t) for time
- combine dimensions in random order

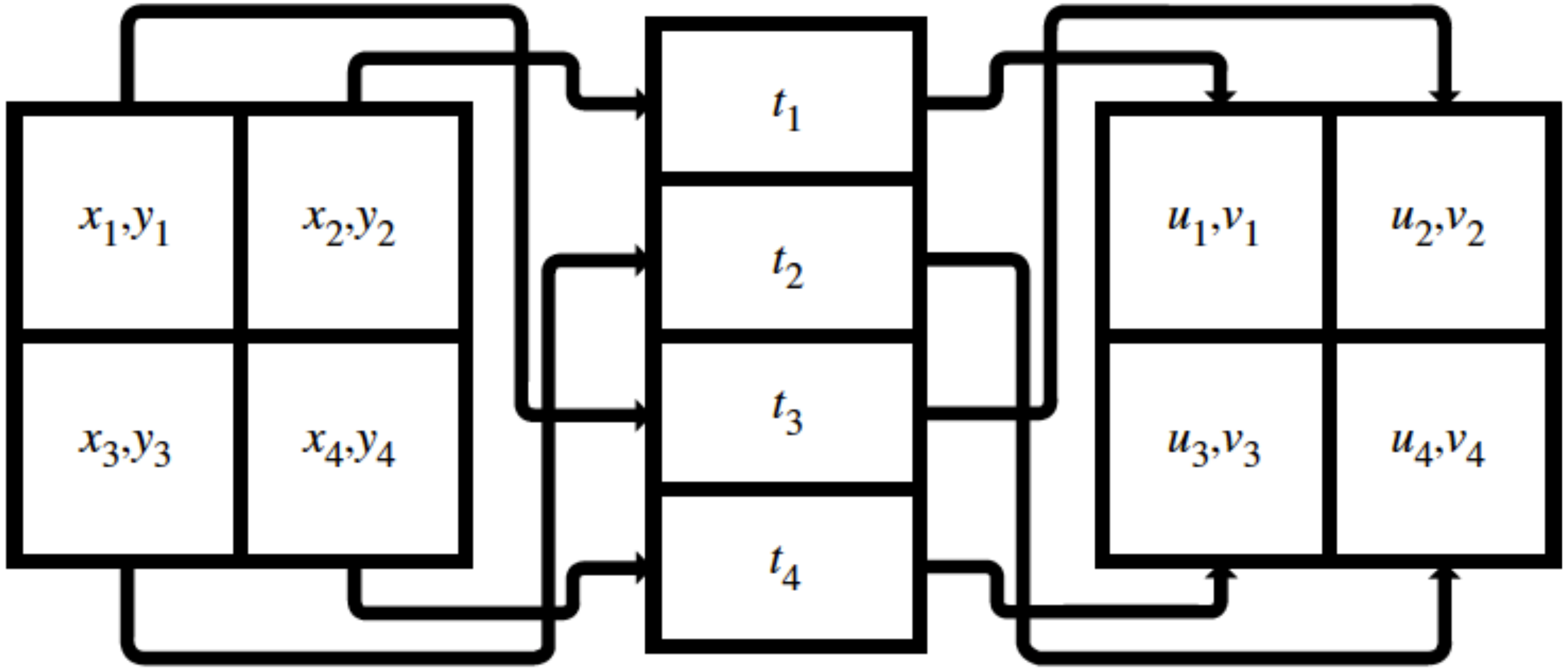
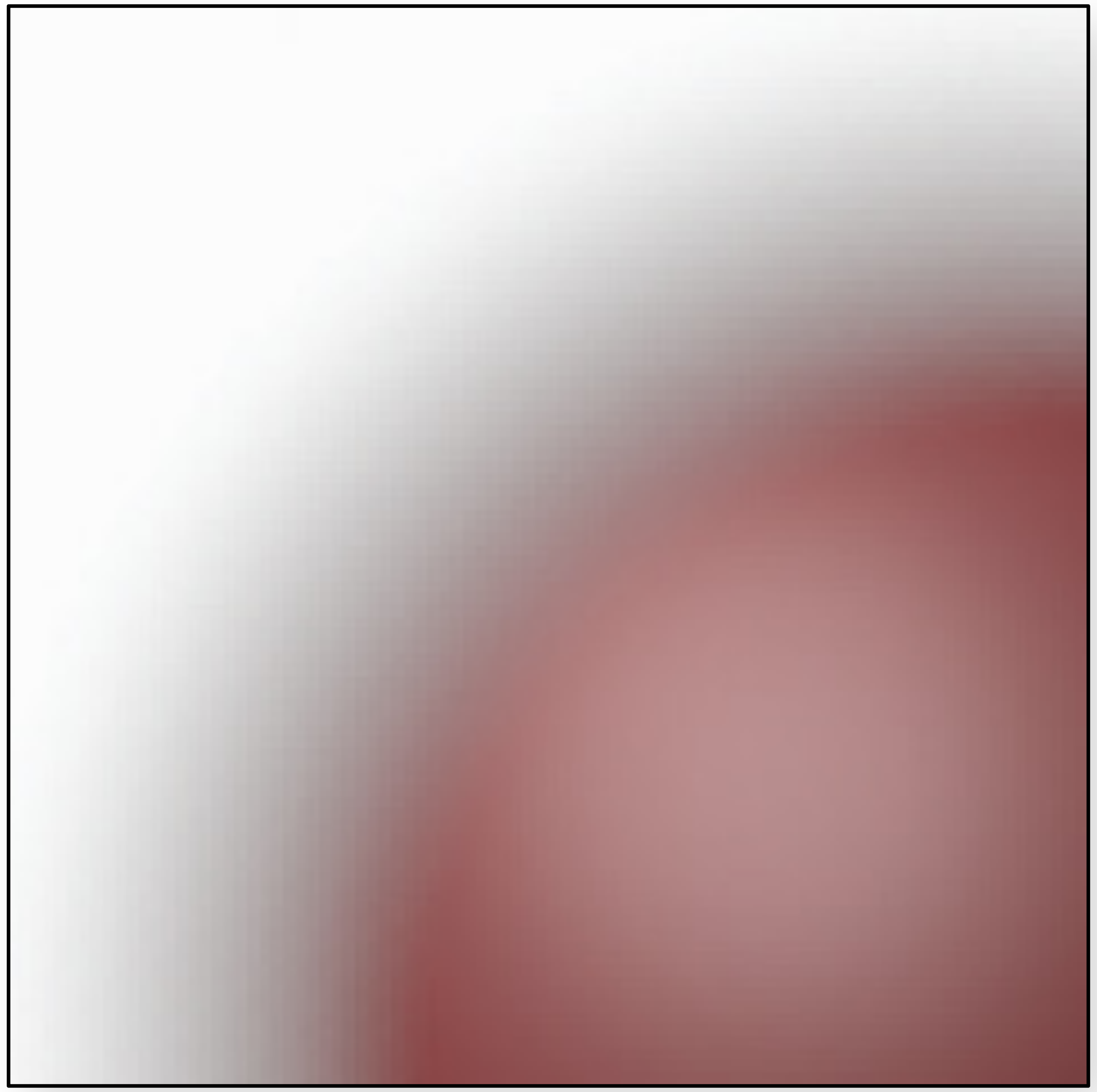


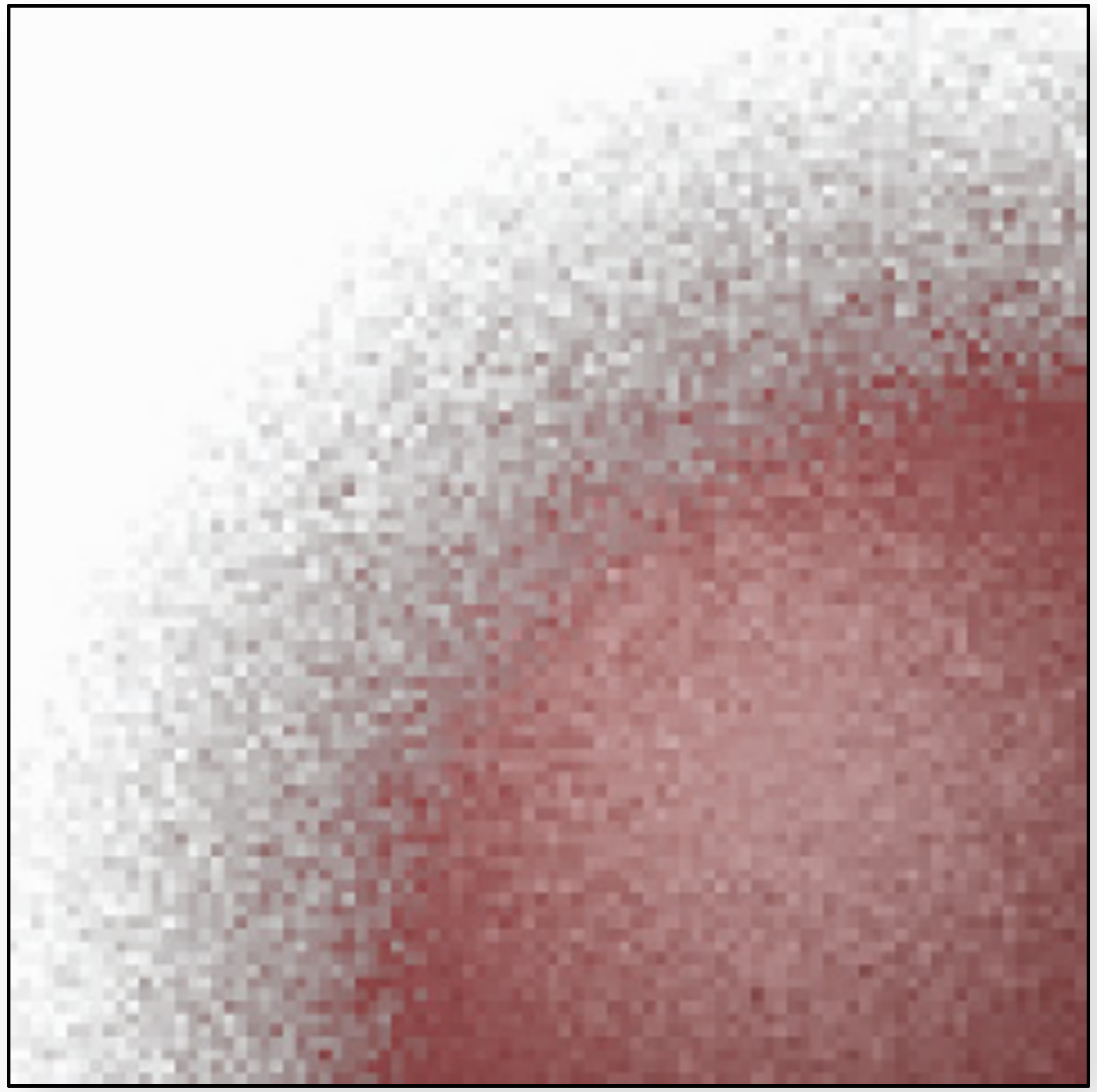
Image source: PBRTe2 [Pharr & Humphreys 2010]

Depth of Field (4D)

Reference



Random Sampling



Uncorrelated Jitter

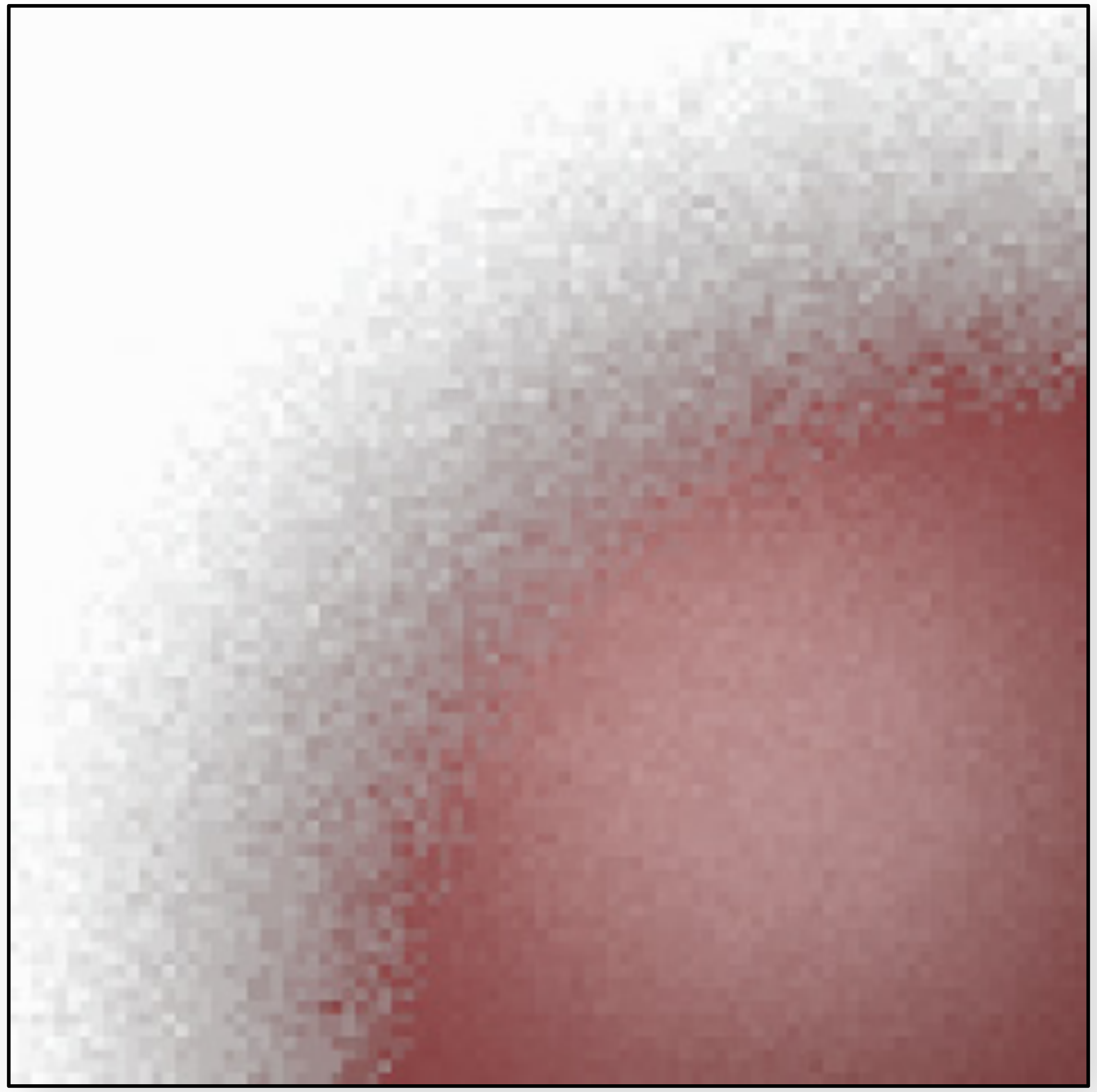


Image source: PBRTe2 [Pharr & Humphreys 2010]

Latin Hypercube (N-Rooks) Sampling

[Shirley 91]

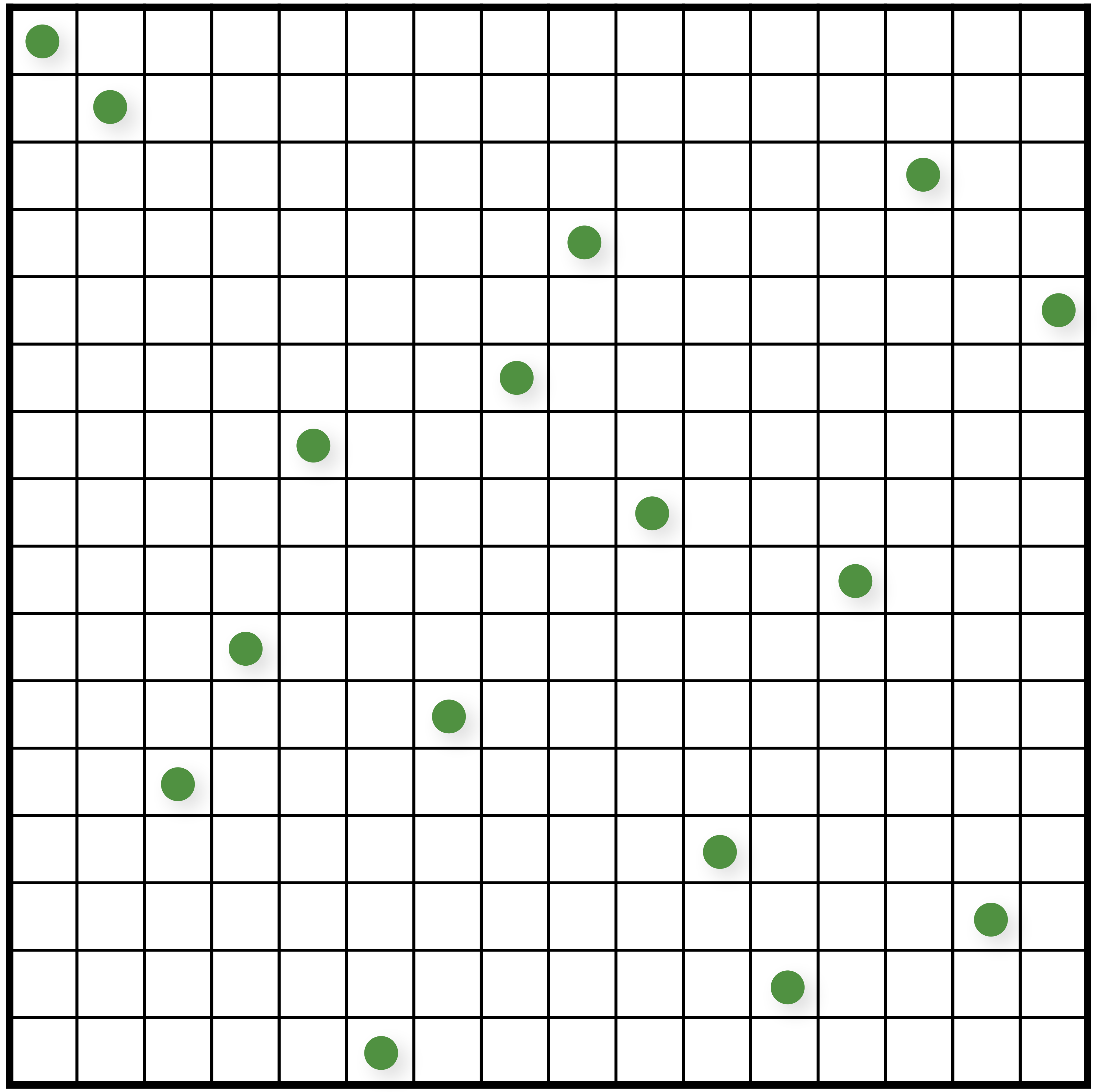
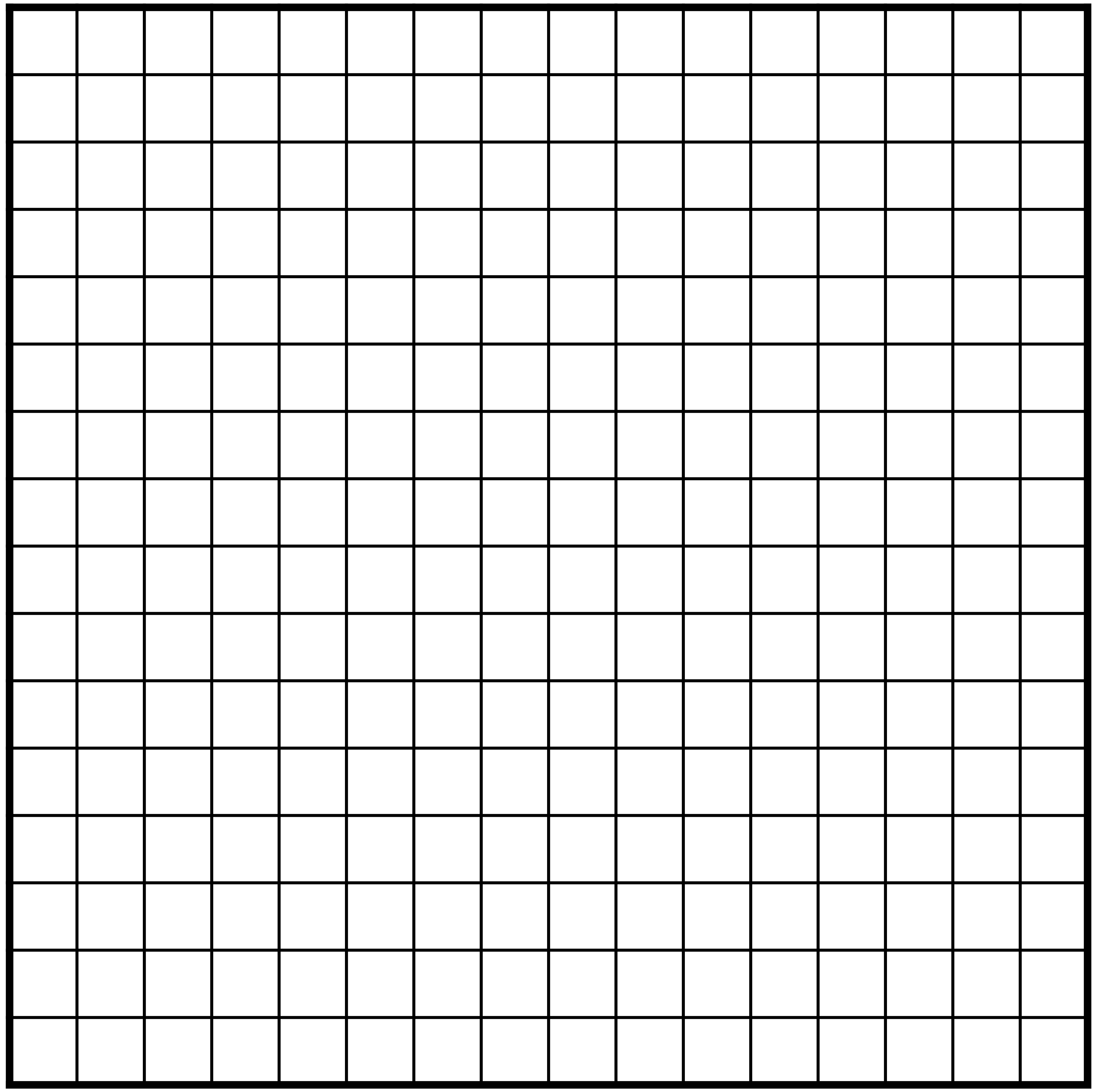


Image source: Michael Maggs, CC BY-SA 2.5

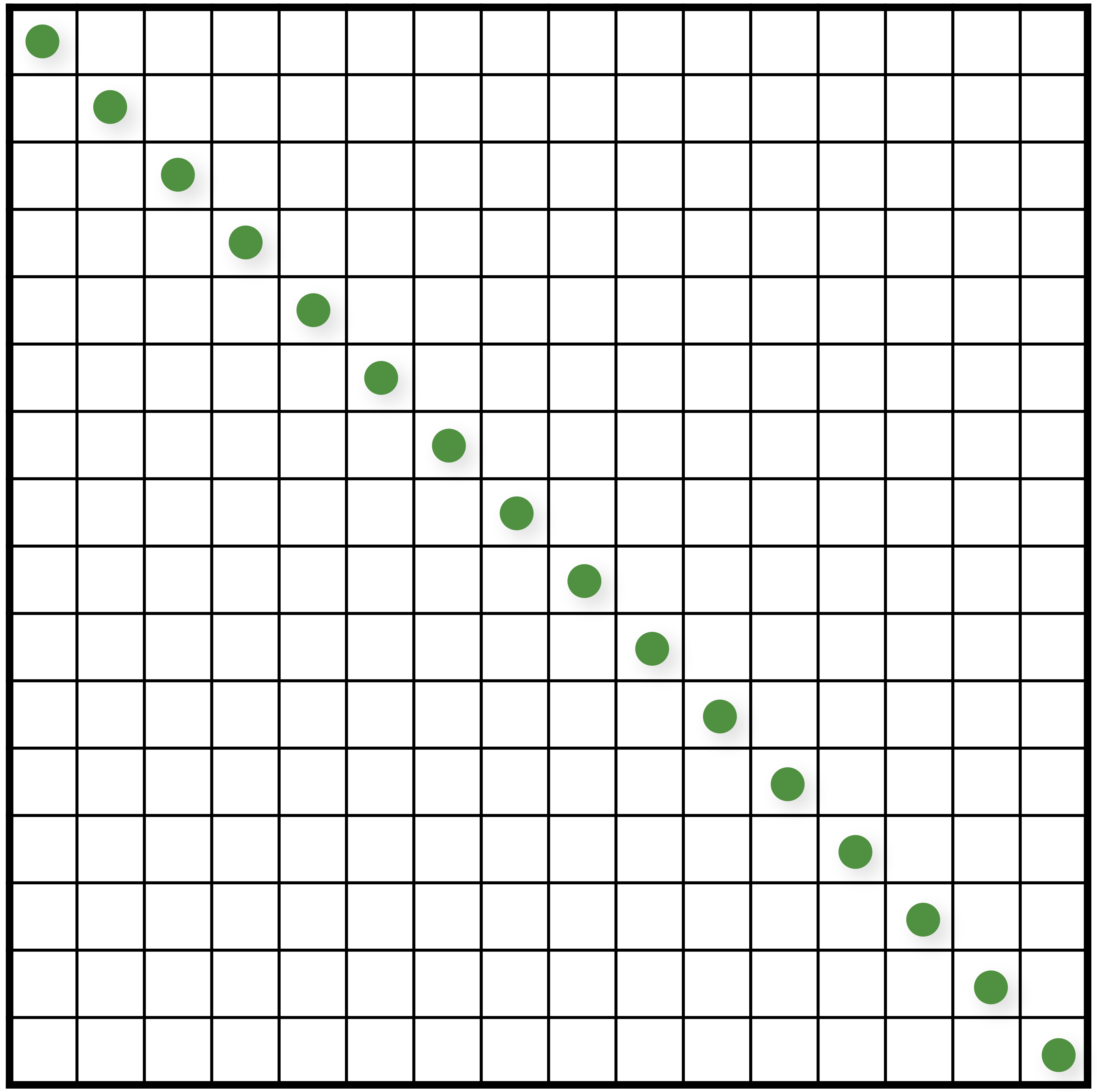
Latin Hypercube (N-Rooks) Sampling

```
// 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,:));
```

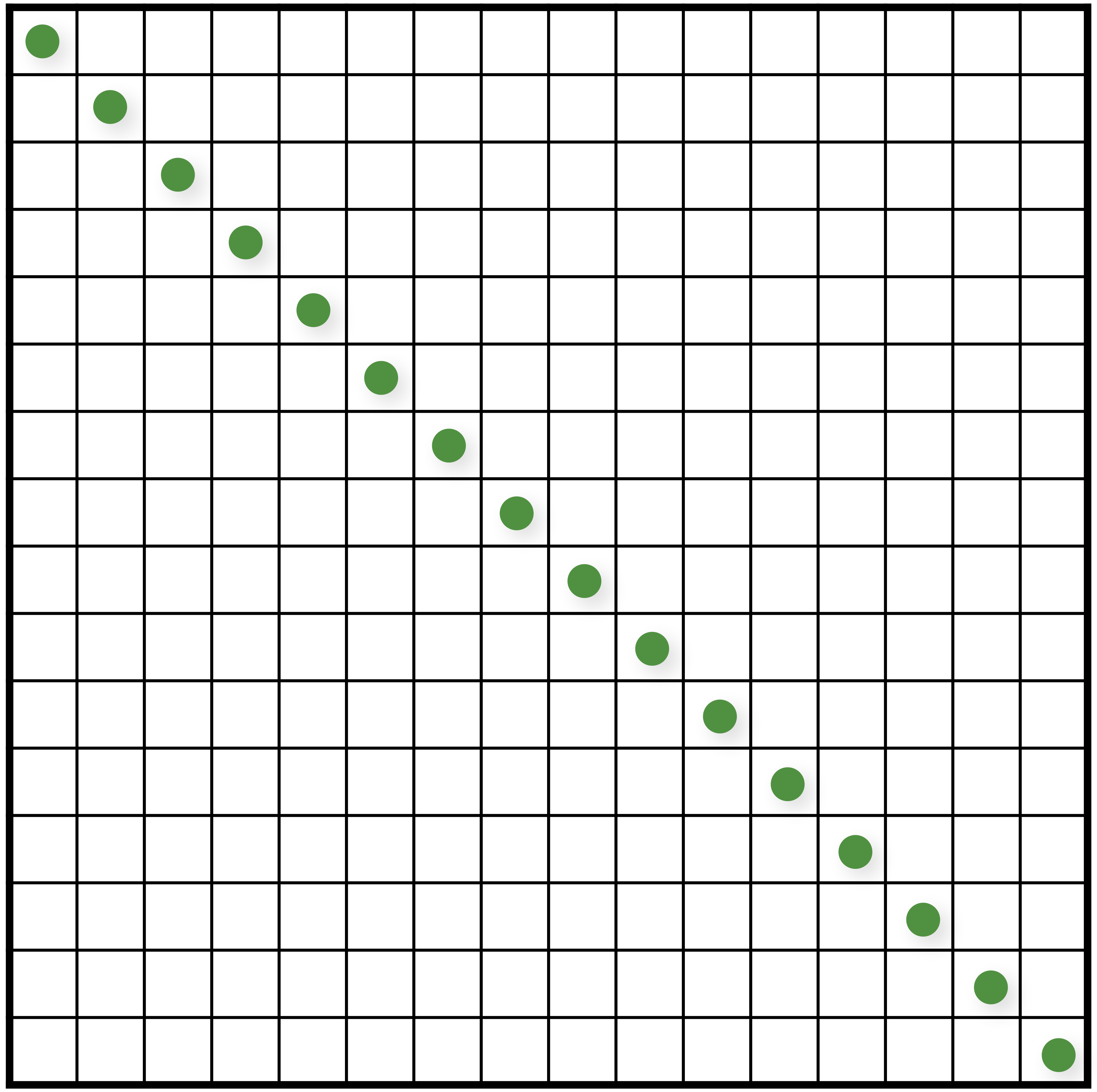


Initialize

Latin Hypercube (N-Rooks) Sampling

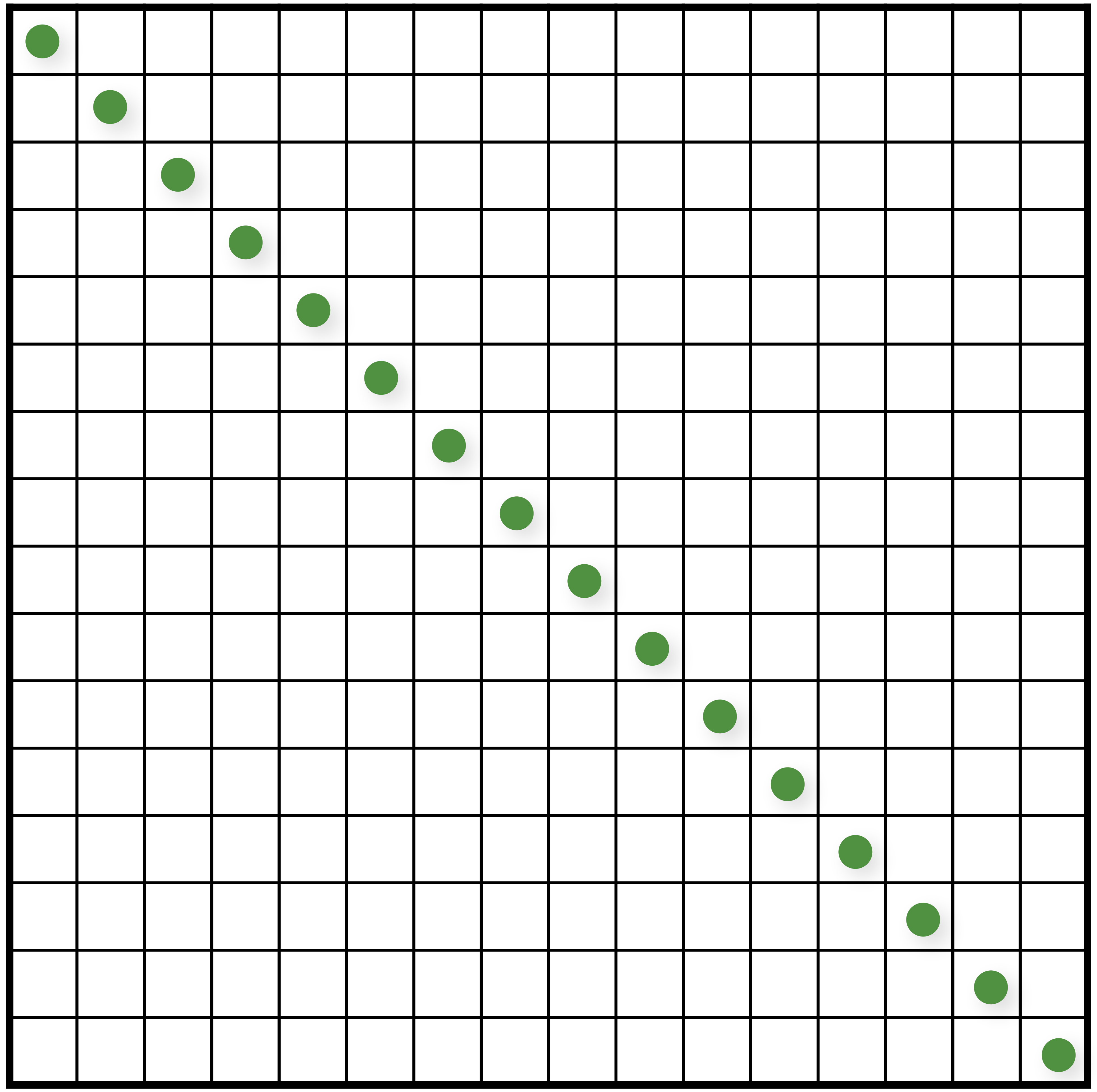
```
// initialize the diagonal
for (uint d = 0; d < numDimensions; d++)
    for (uint i = 0; i < numS; i++)
        samples(d,i) = (i + randf())/numS;
```

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



Latin Hypercube (N-Rooks) Sampling

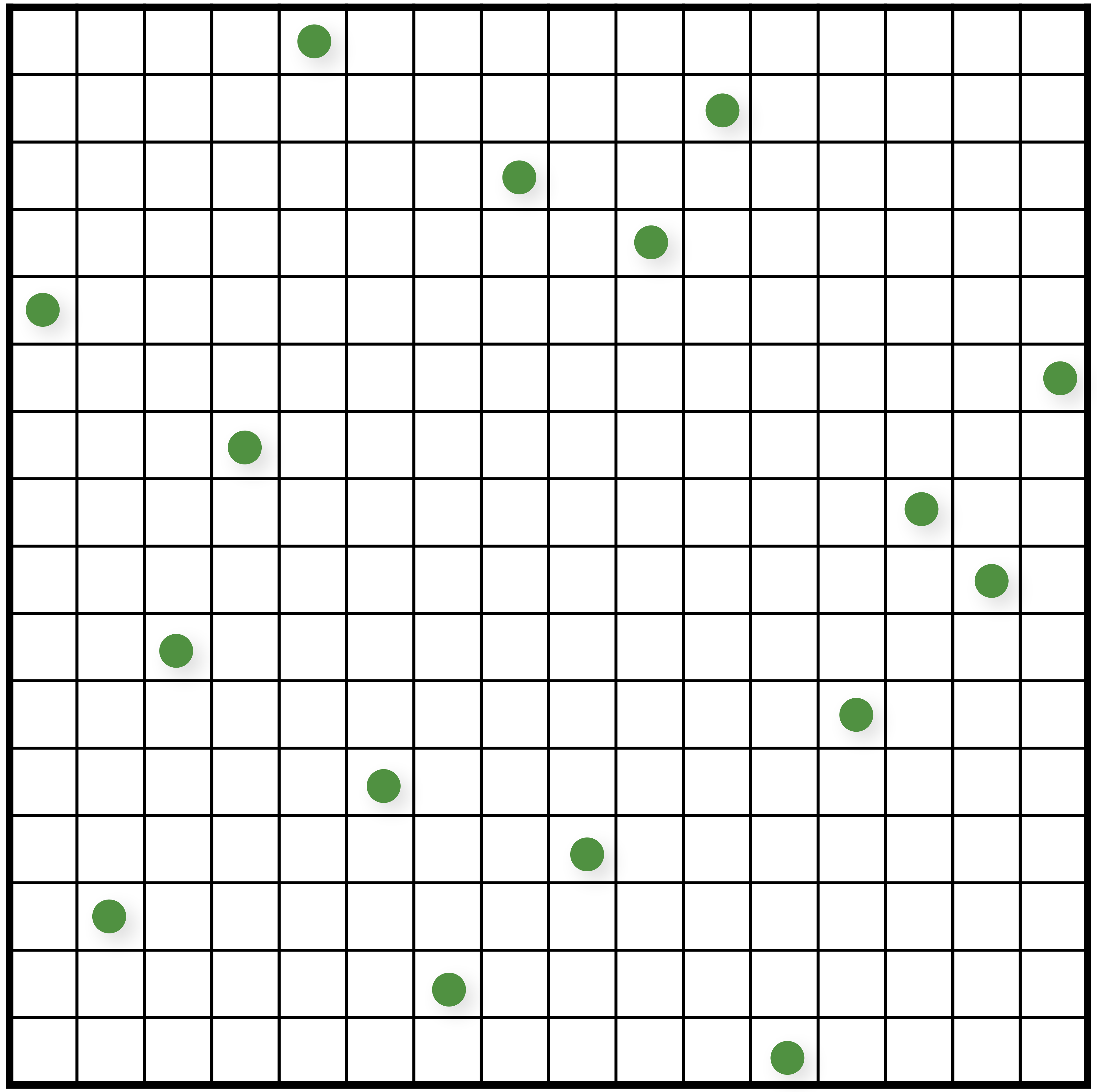
```
// initialize the diagonal
for (uint d = 0; d < numDimensions; d++)
    for (uint i = 0; i < numS; i++)
        samples(d,i) = (i + randf())/numS;

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



Shuffle rows

Latin Hypercube (N-Rooks) Sampling

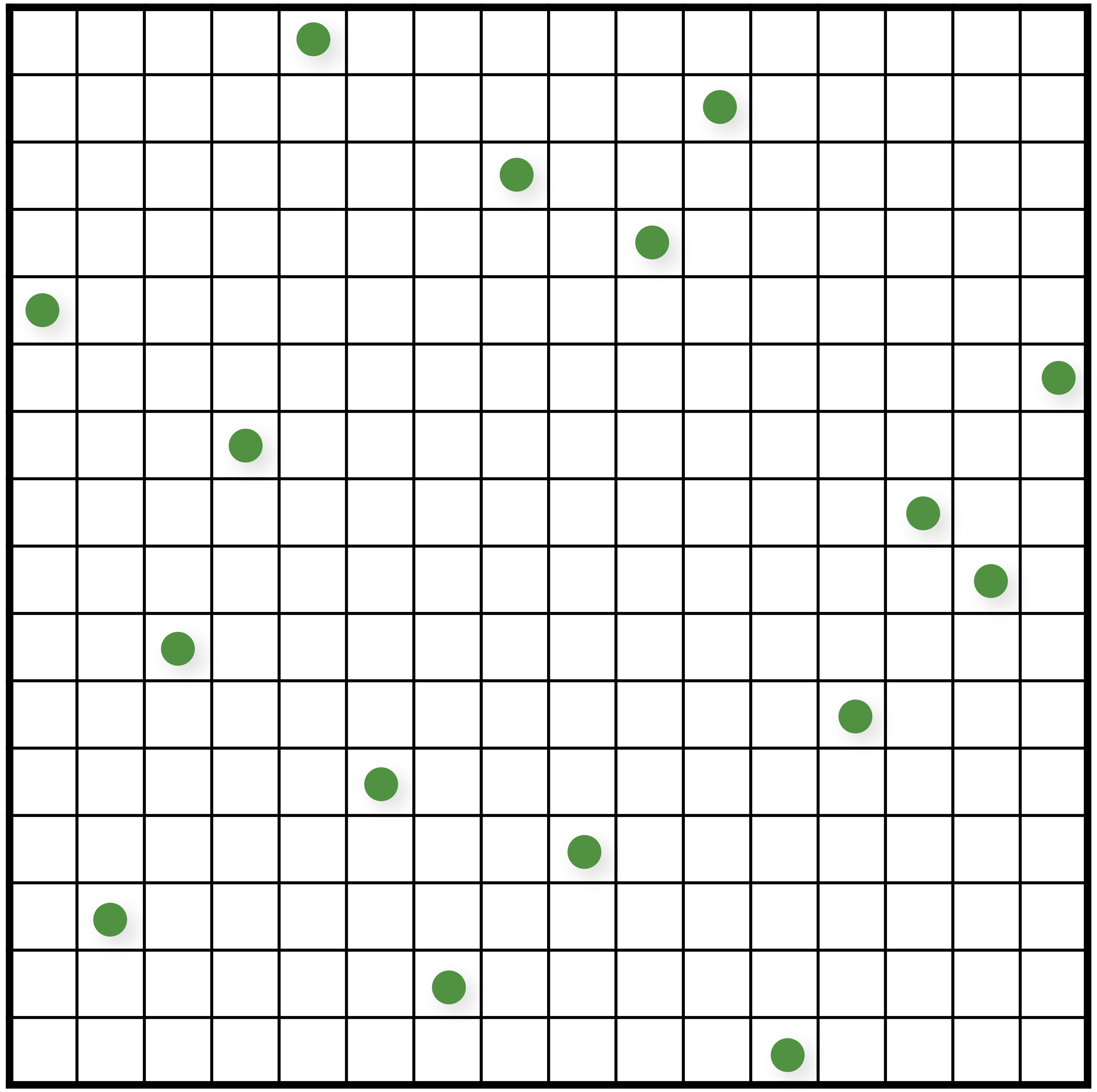
```
// initialize the diagonal
for (uint d = 0; d < numDimensions; d++)
    for (uint i = 0; i < numS; i++)
        samples(d,i) = (i + randf())/numS;

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



Shuffle rows

Latin Hypercube (N-Rooks) Sampling

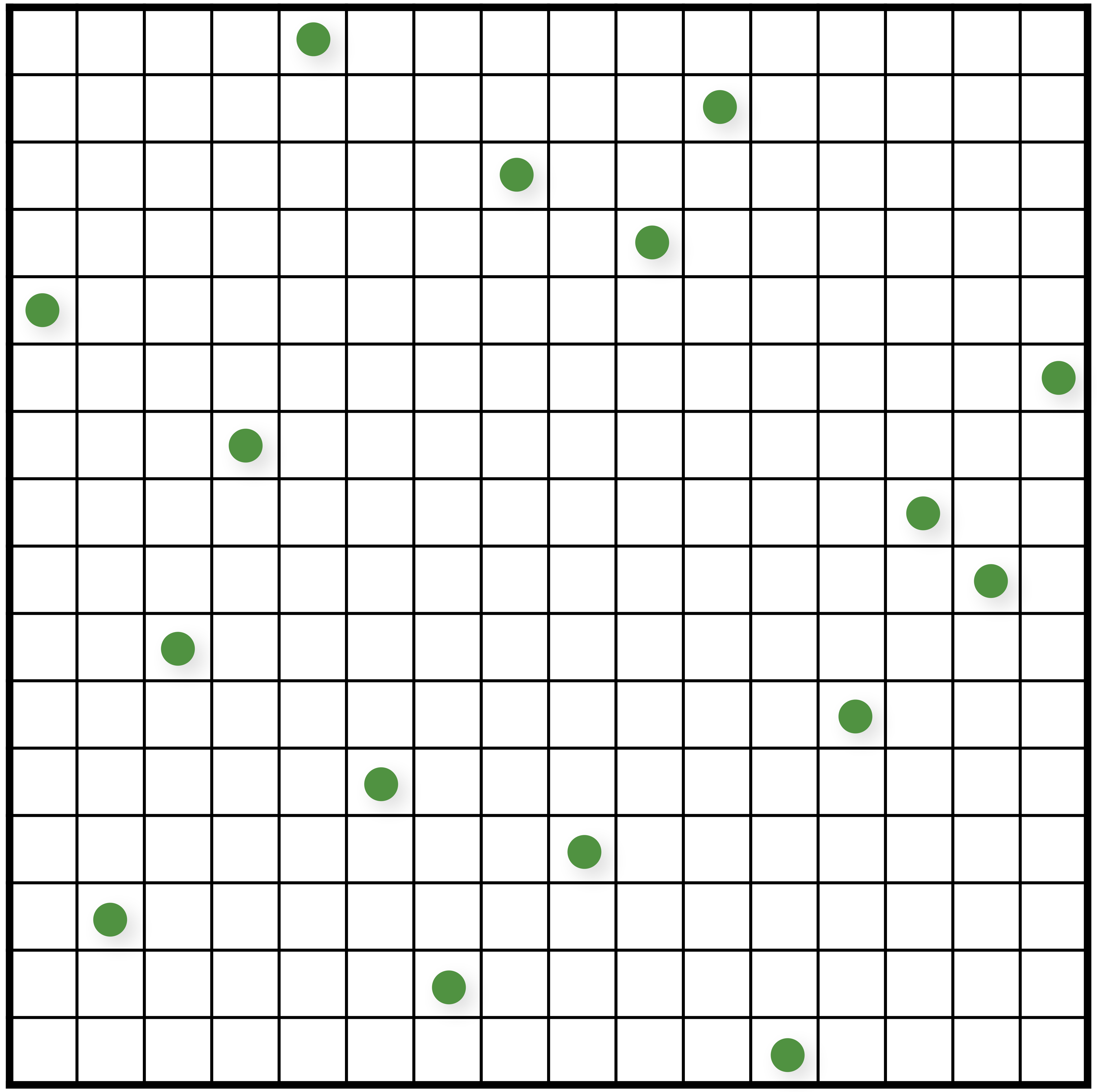
```
// initialize the diagonal
for (uint d = 0; d < numDimensions; d++)
    for (uint i = 0; i < numS; i++)
        samples(d,i) = (i + randf())/numS;

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



Shuffle rows

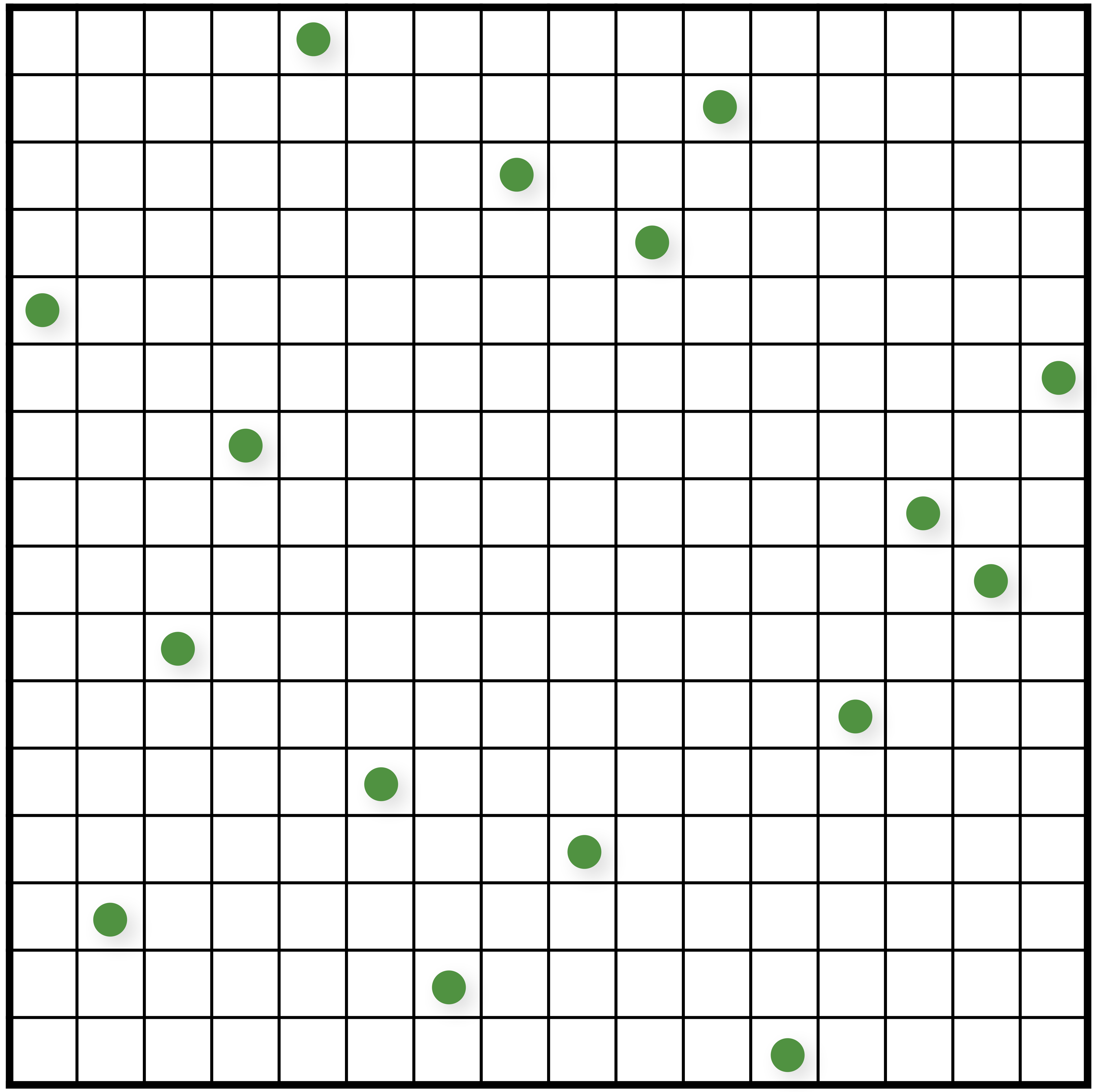
Latin Hypercube (N-Rooks) Sampling

```
// 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

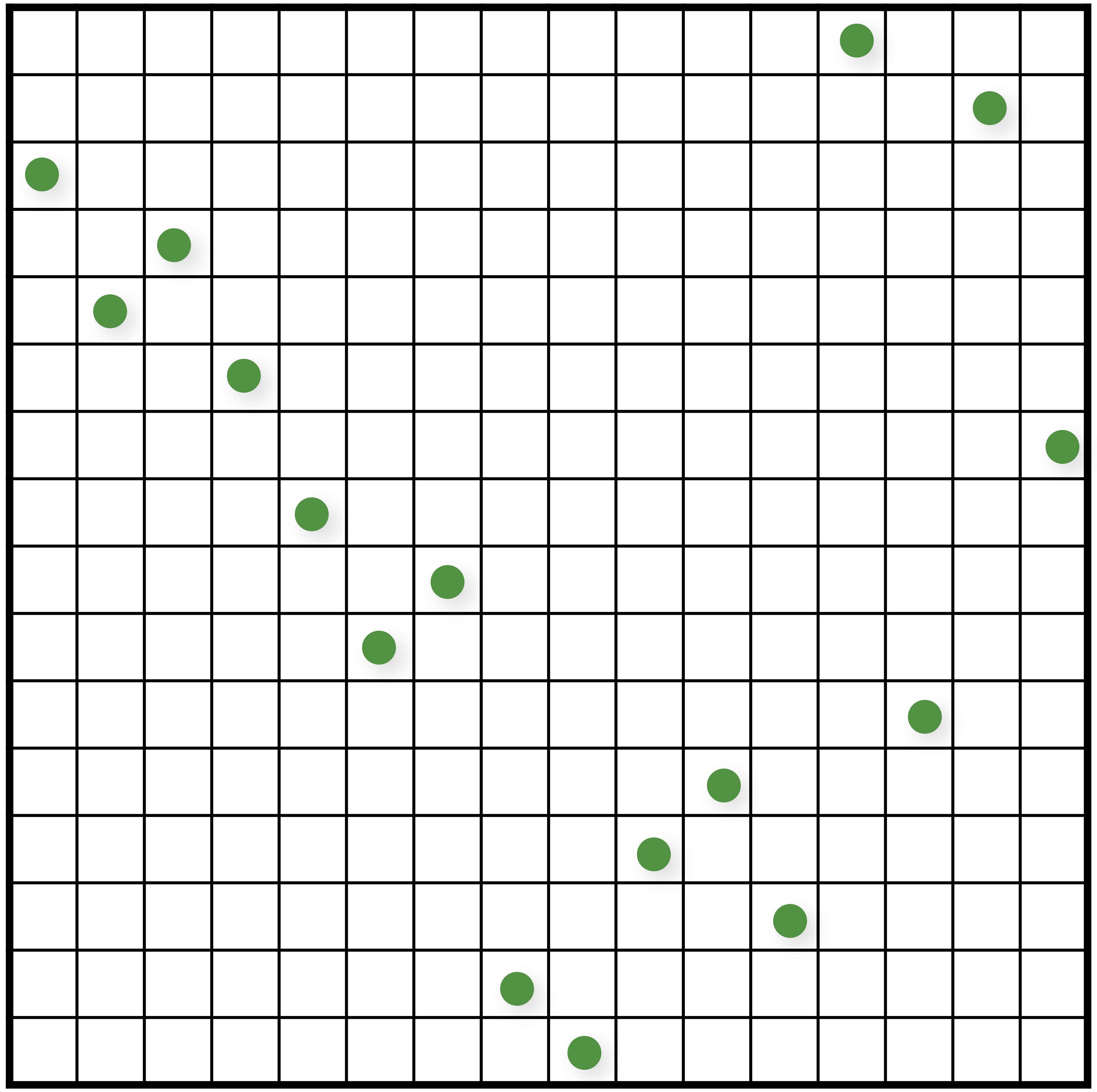
```
// initialize the diagonal
for (uint d = 0; d < numDimensions; d++)
    for (uint i = 0; i < numS; i++)
        samples(d,i) = (i + randf())/numS;

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



Shuffle columns

Latin Hypercube (N-Rooks) Sampling

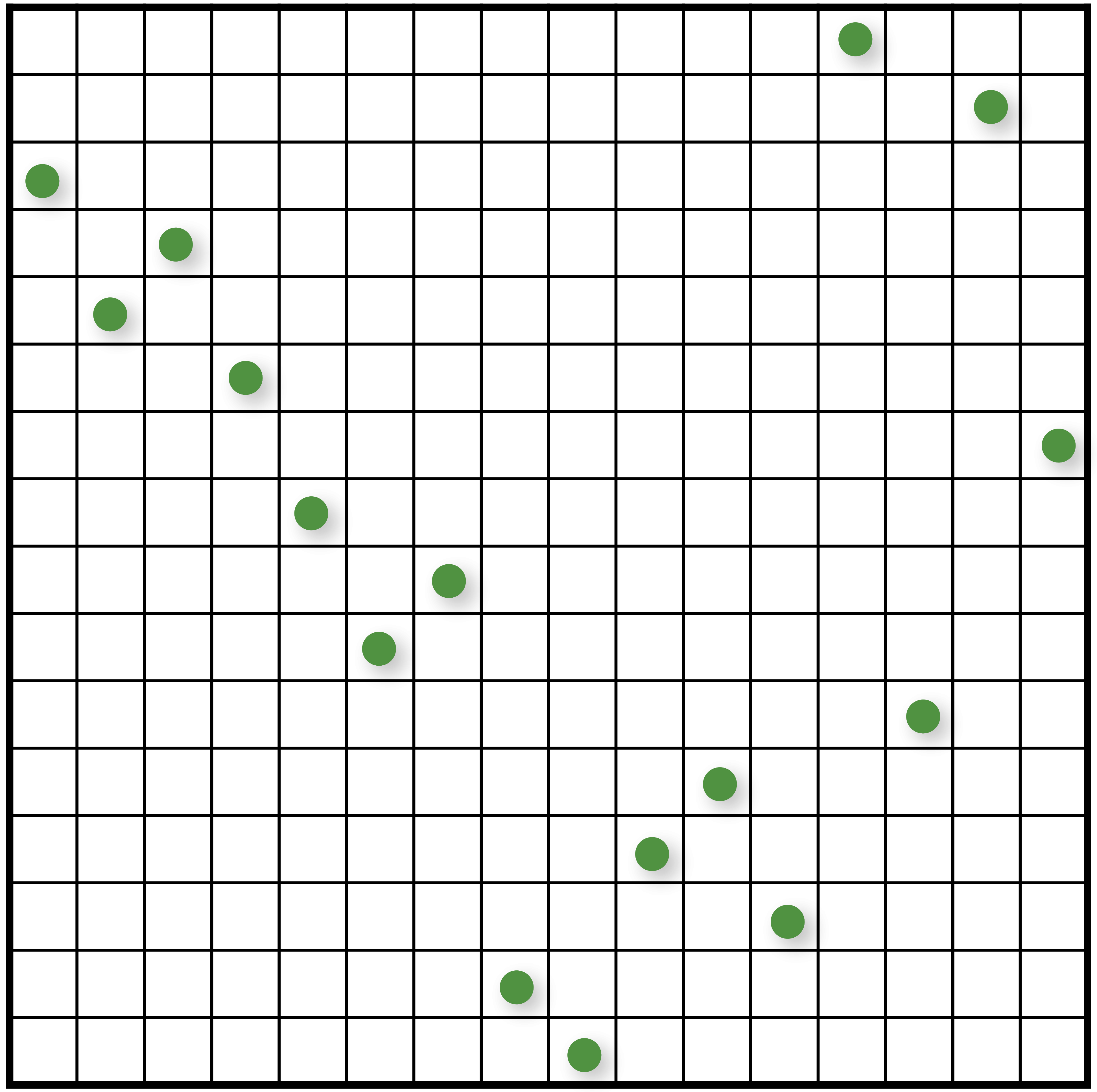
```
// initialize the diagonal
for (uint d = 0; d < numDimensions; d++)
    for (uint i = 0; i < numS; i++)
        samples(d,i) = (i + randf())/numS;

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



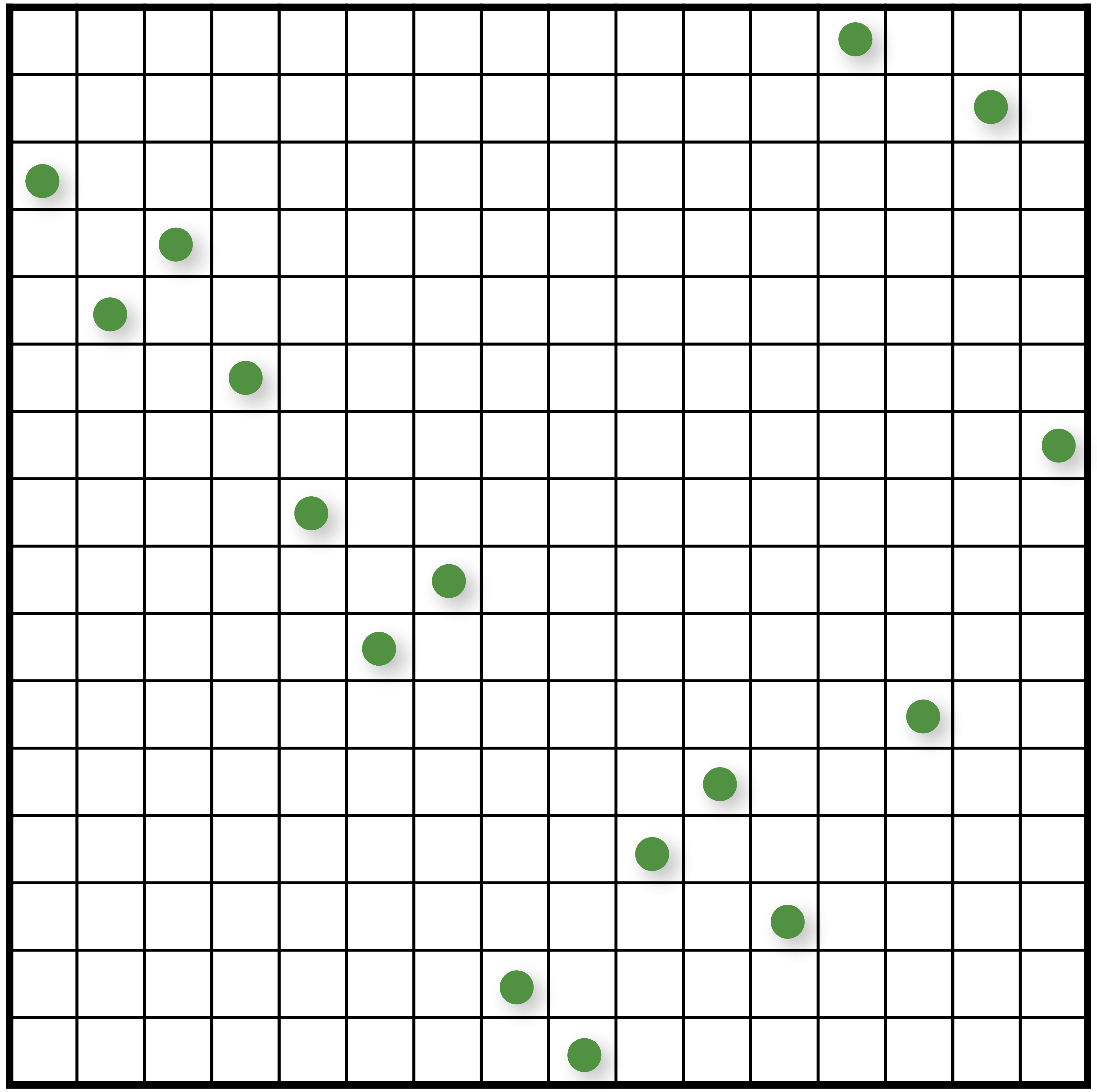
Shuffle columns

Latin Hypercube (N-Rooks) Sampling

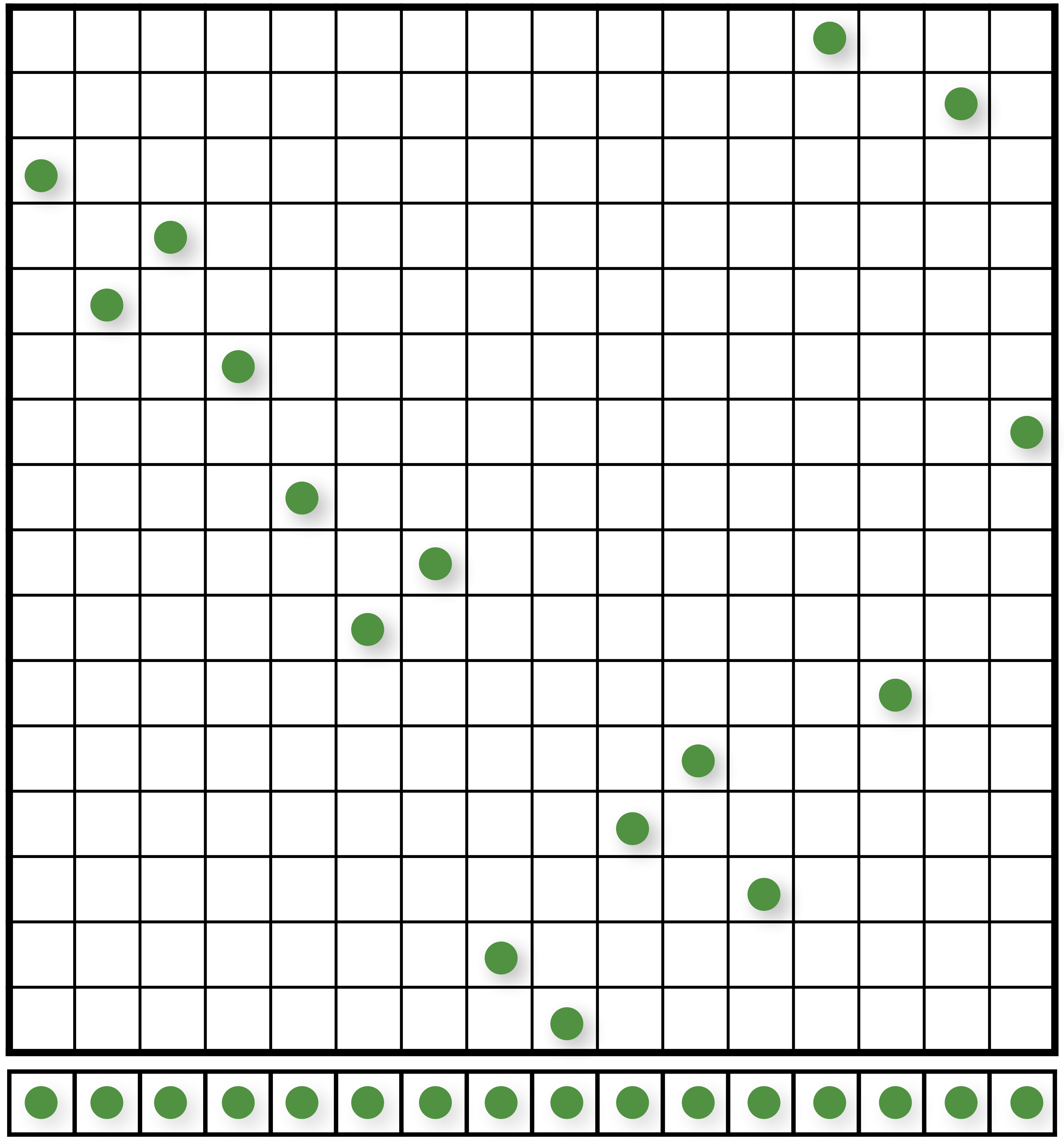
```
// initialize the diagonal
for (uint d = 0; d < numDimensions; d++)
    for (uint i = 0; i < numS; i++)
        samples(d,i) = (i + randf())/numS;

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



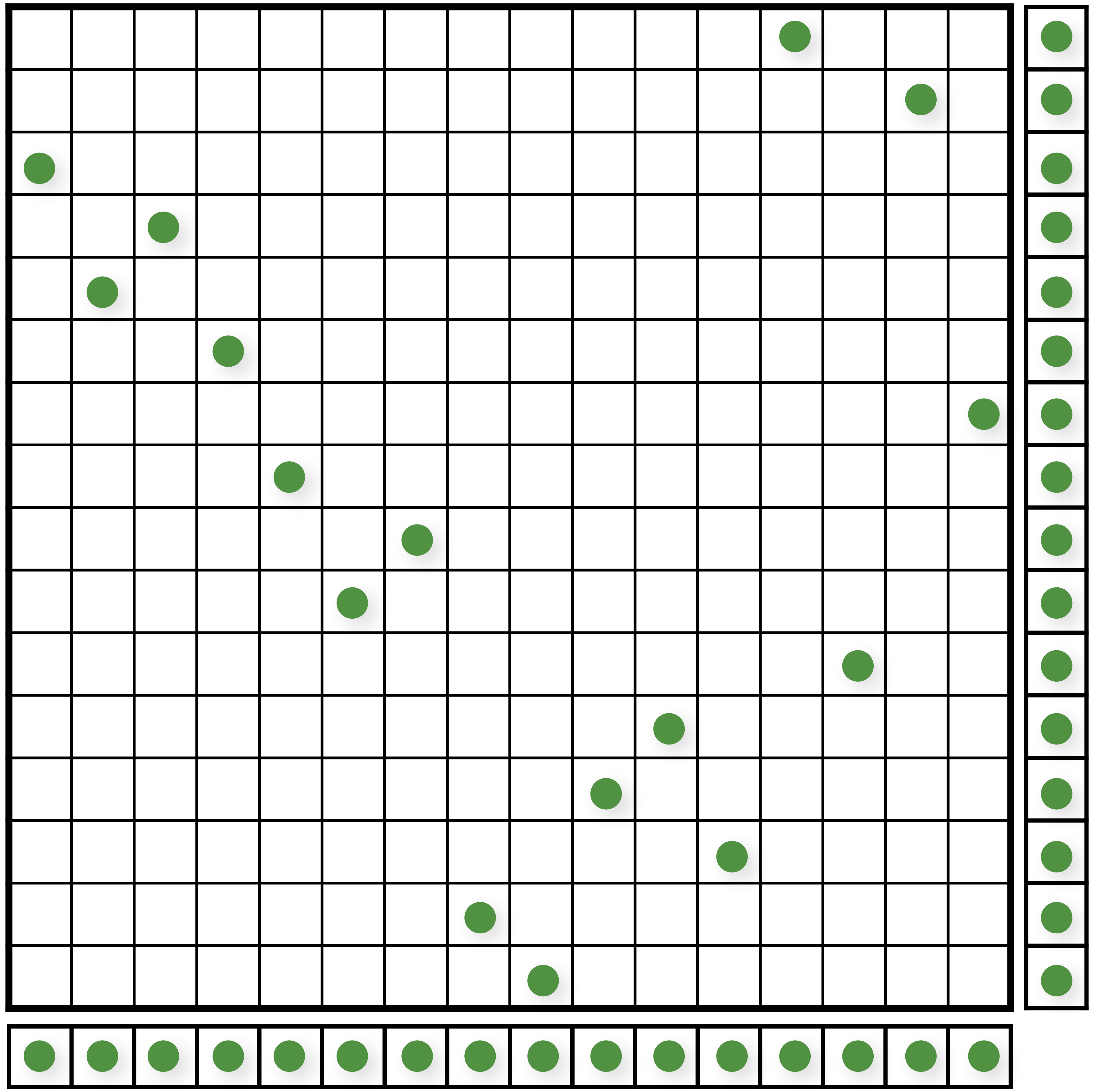
Latin Hypercube (N-Rooks) Sampling



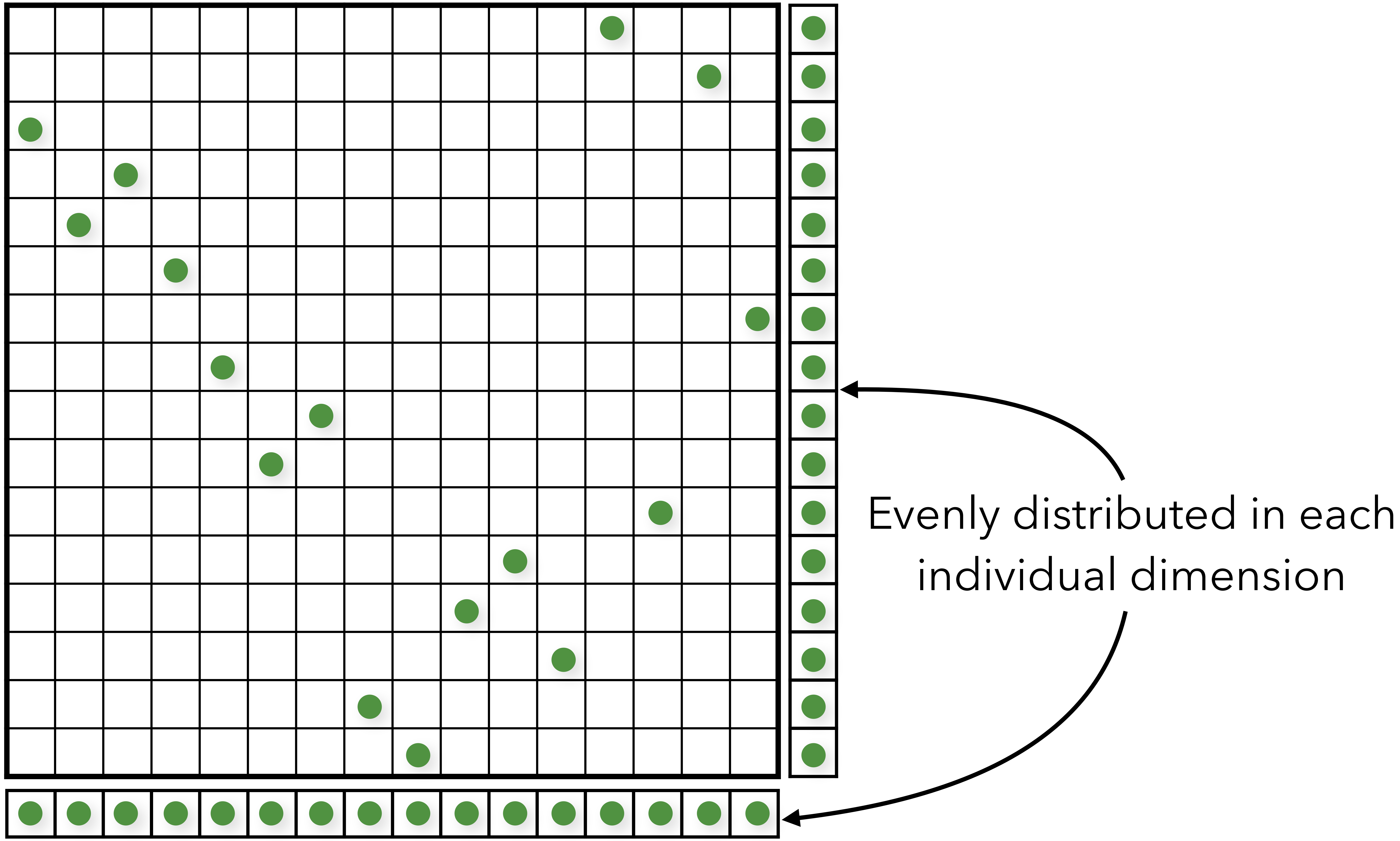
Latin Hypercube (N-Rooks) Sampling



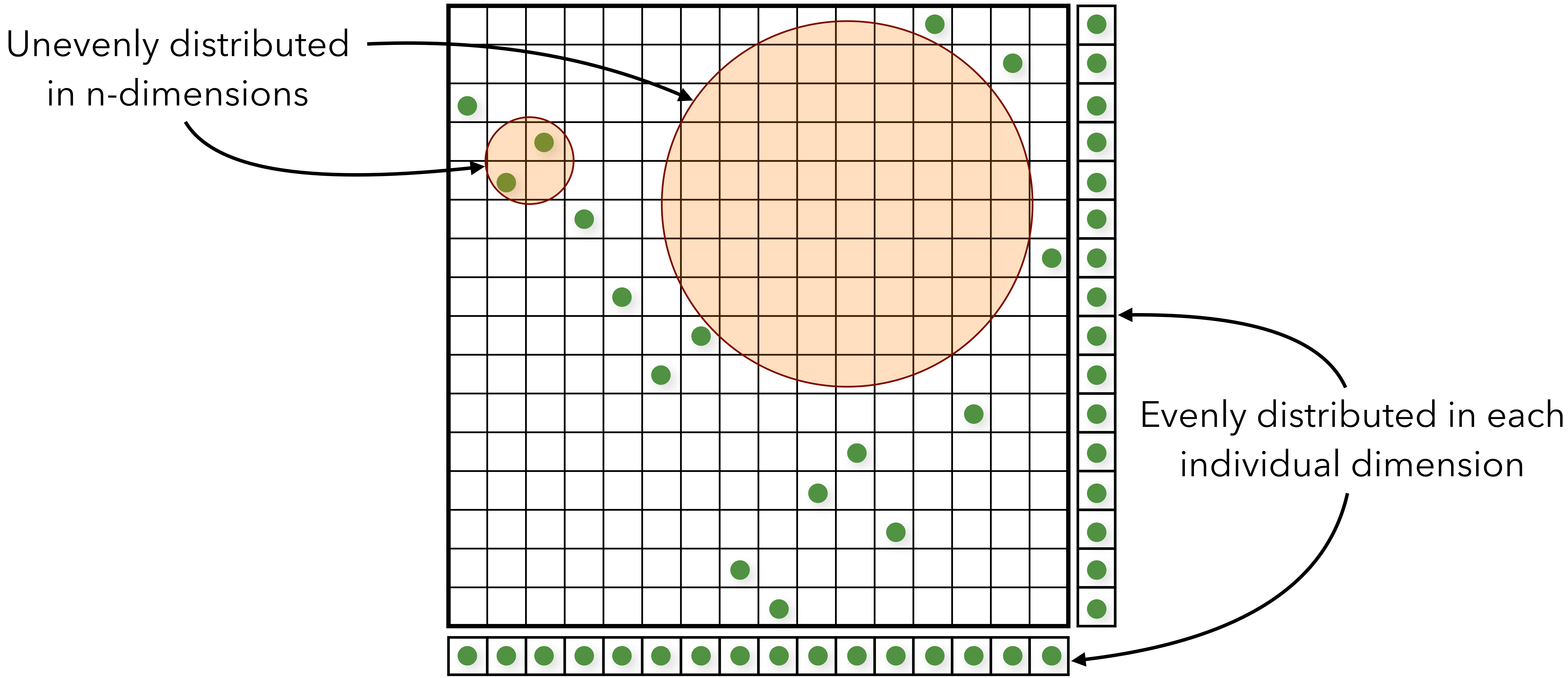
Latin Hypercube (N-Rooks) Sampling



Latin Hypercube (N-Rooks) Sampling

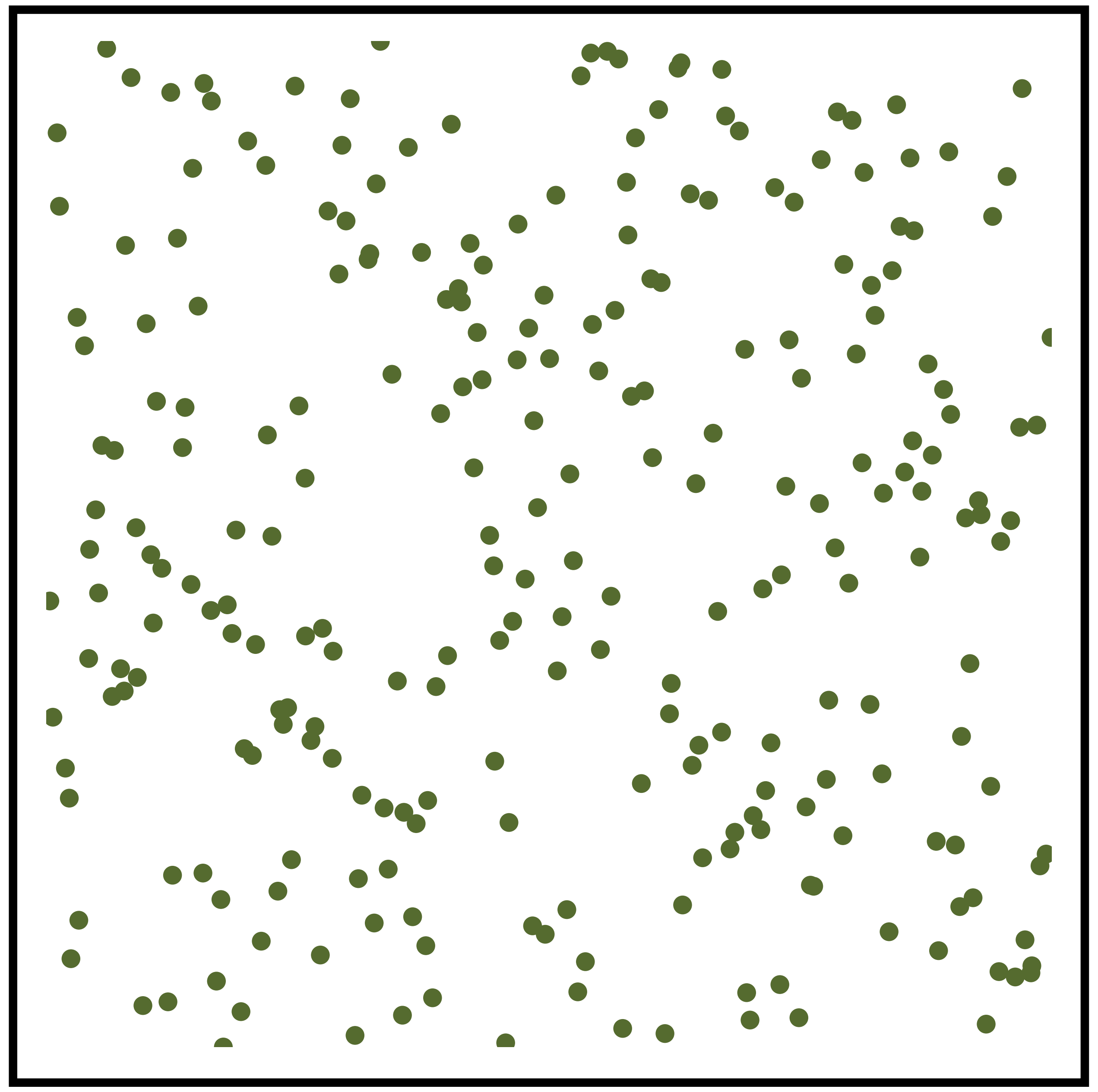


Latin Hypercube (N-Rooks) Sampling



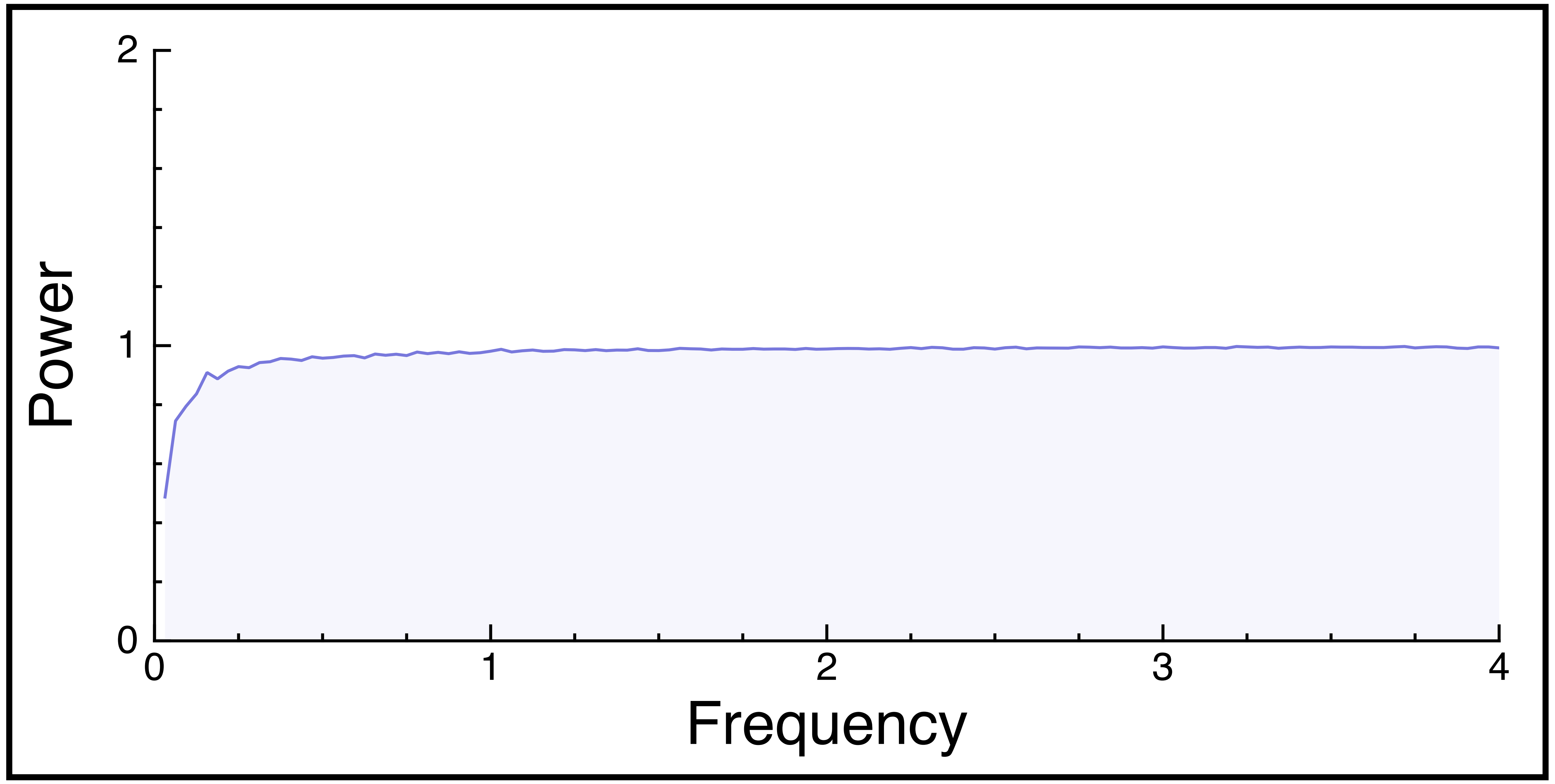
N-Rooks Sampling

Samples



Expected power spectrum

Radial mean

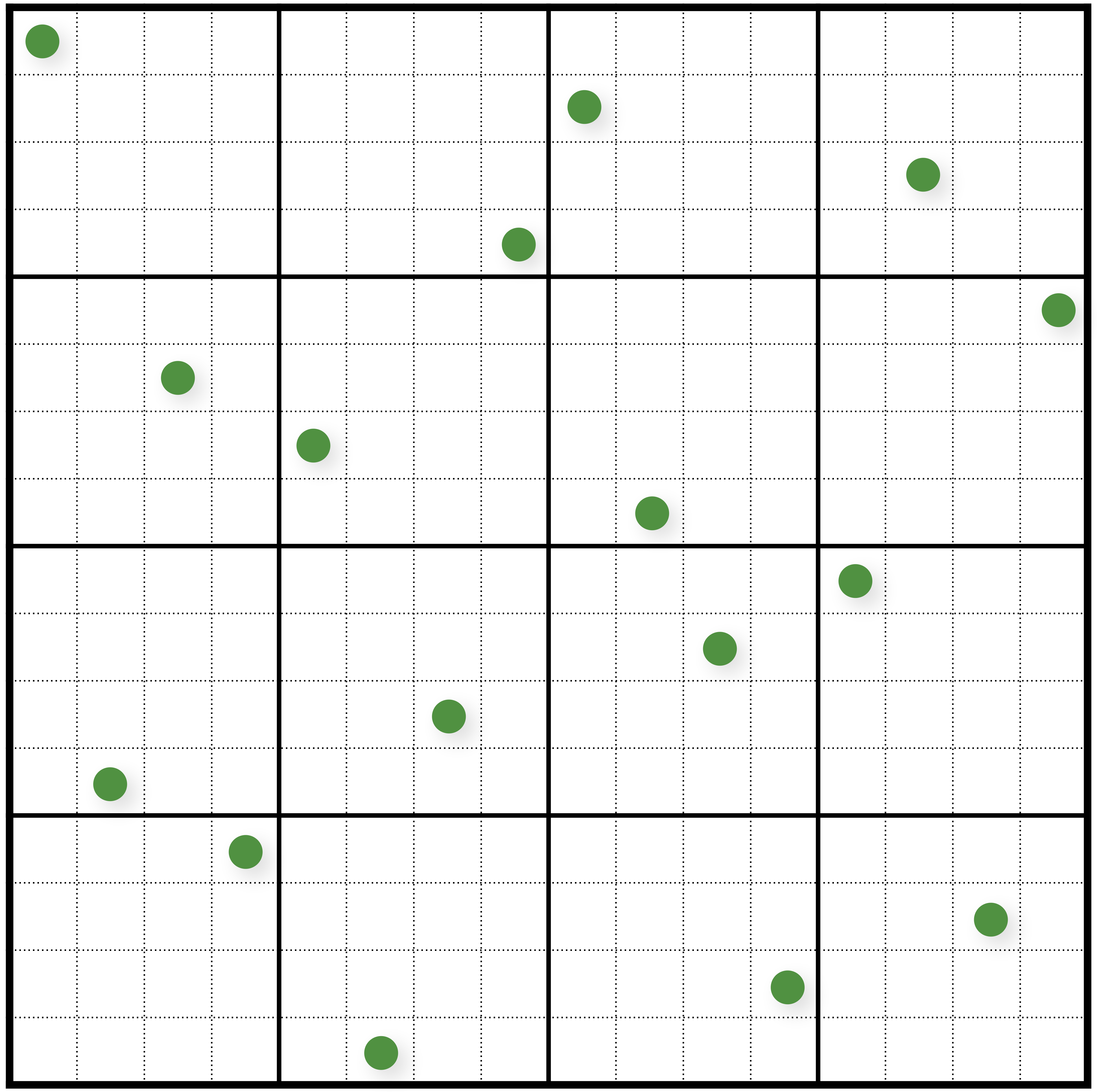


Multi-Jittered Sampling

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

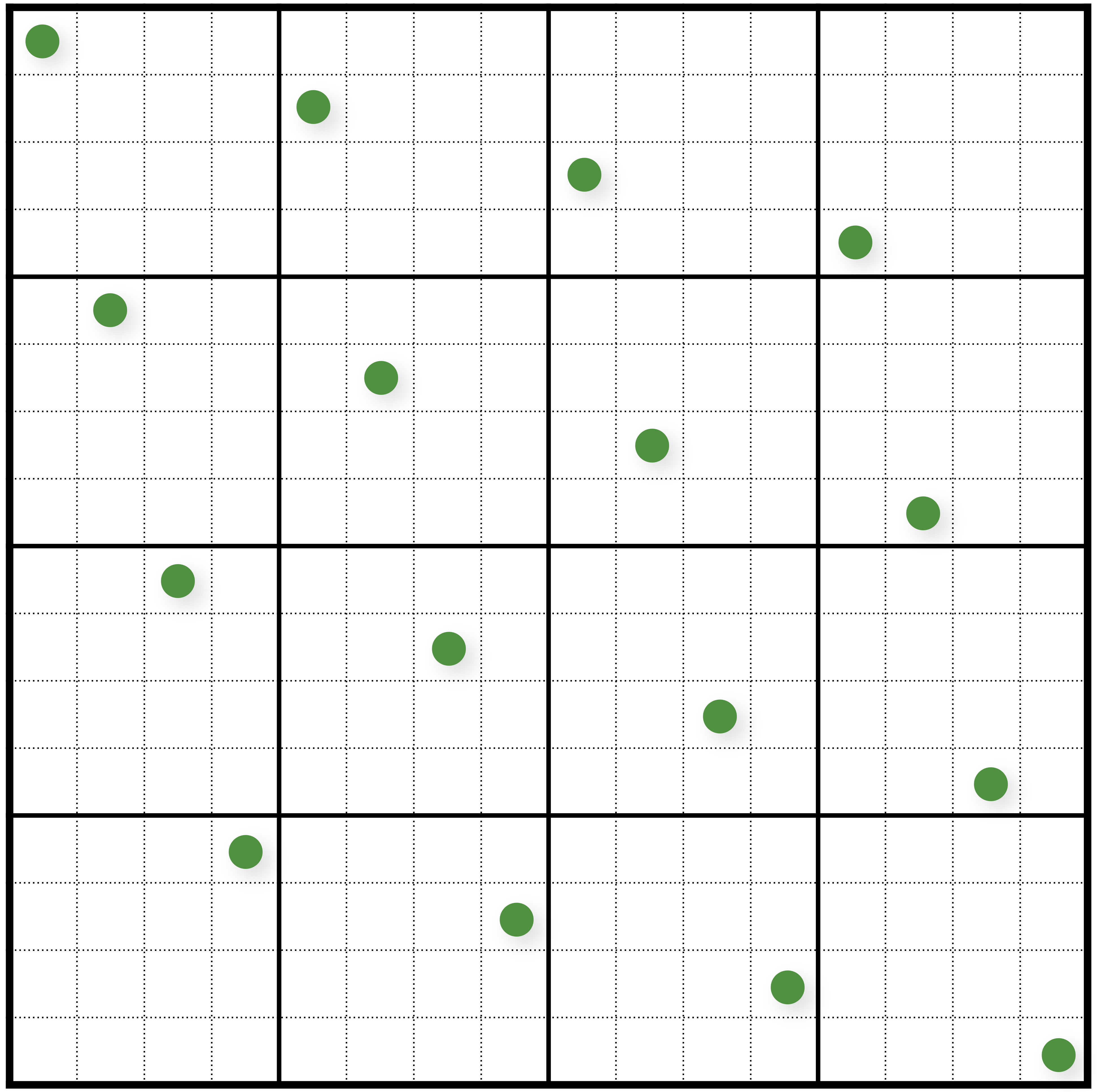
Multi-Jittered Sampling



Multi-Jittered Sampling

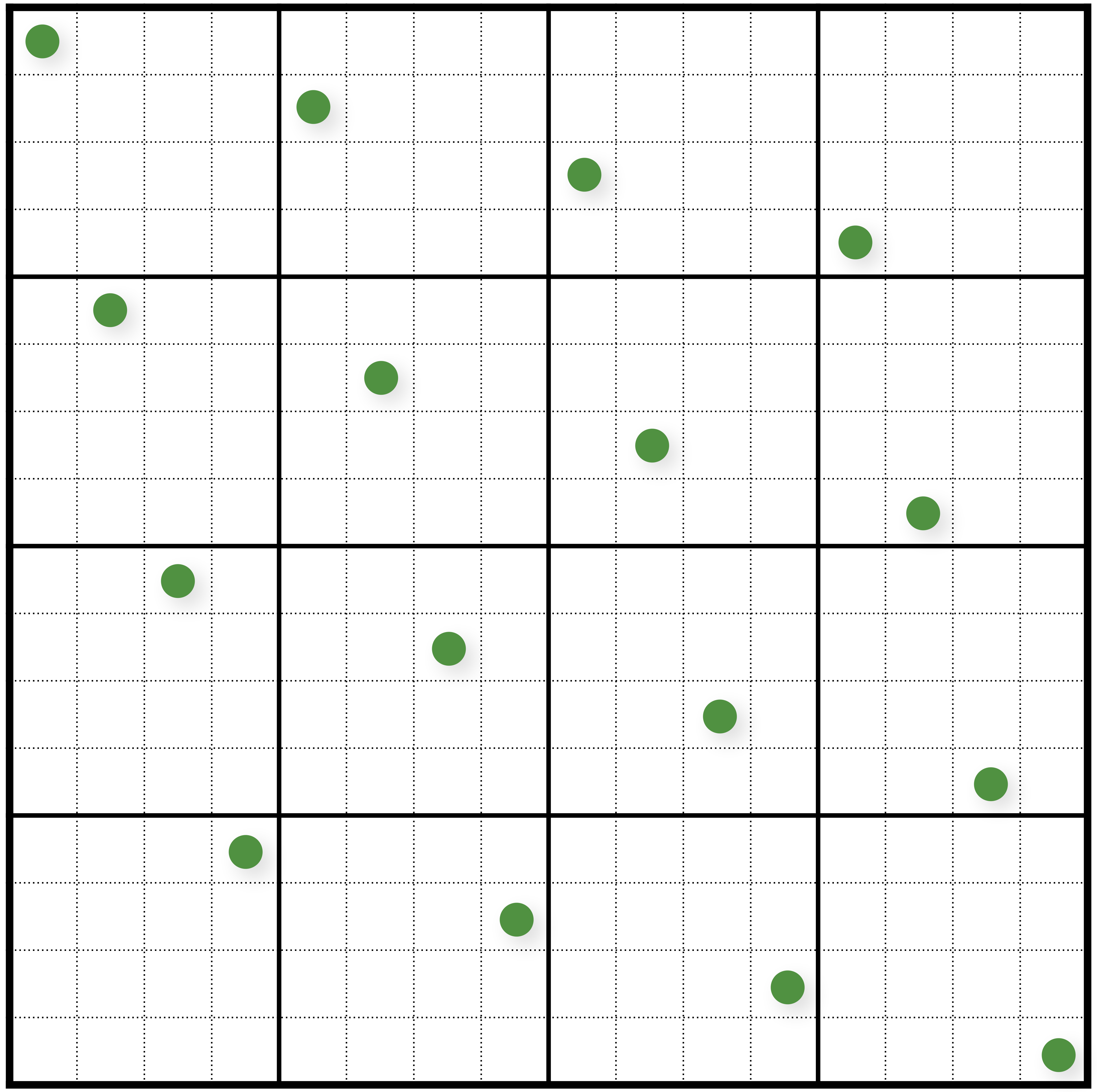
```
// initialize
float cellSize = 1.0 / (resX*resY);
for (uint i = 0; i < resX; i++)
    for (uint j = 0; j < resY; j++)
    {
        samples(i,j).x = i/resX + (j+randf()) / (resX*resY);
        samples(i,j).y = j/resY + (i+randf()) / (resX*resY);
    }

// 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



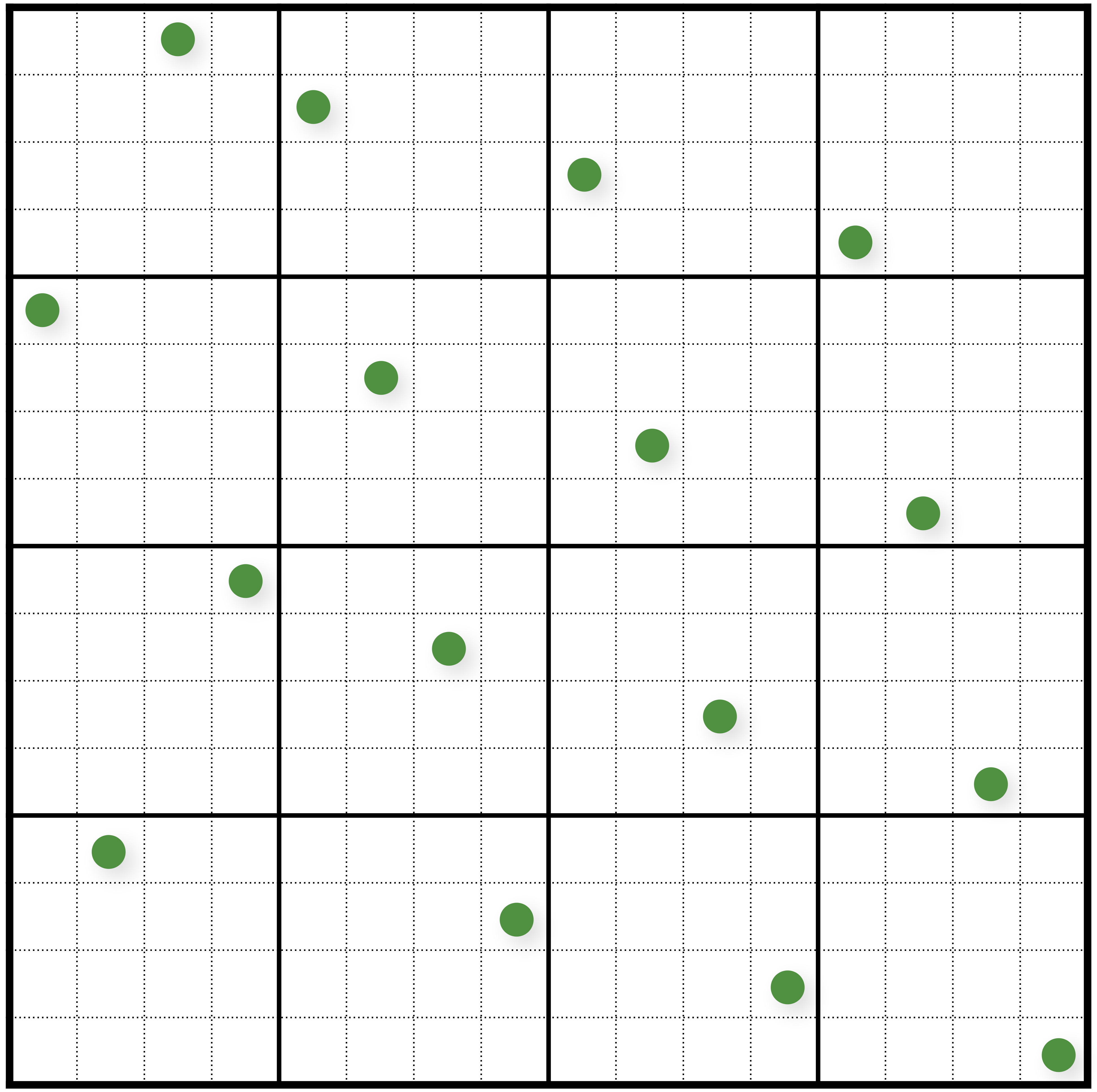
Initialize

Multi-Jittered Sampling



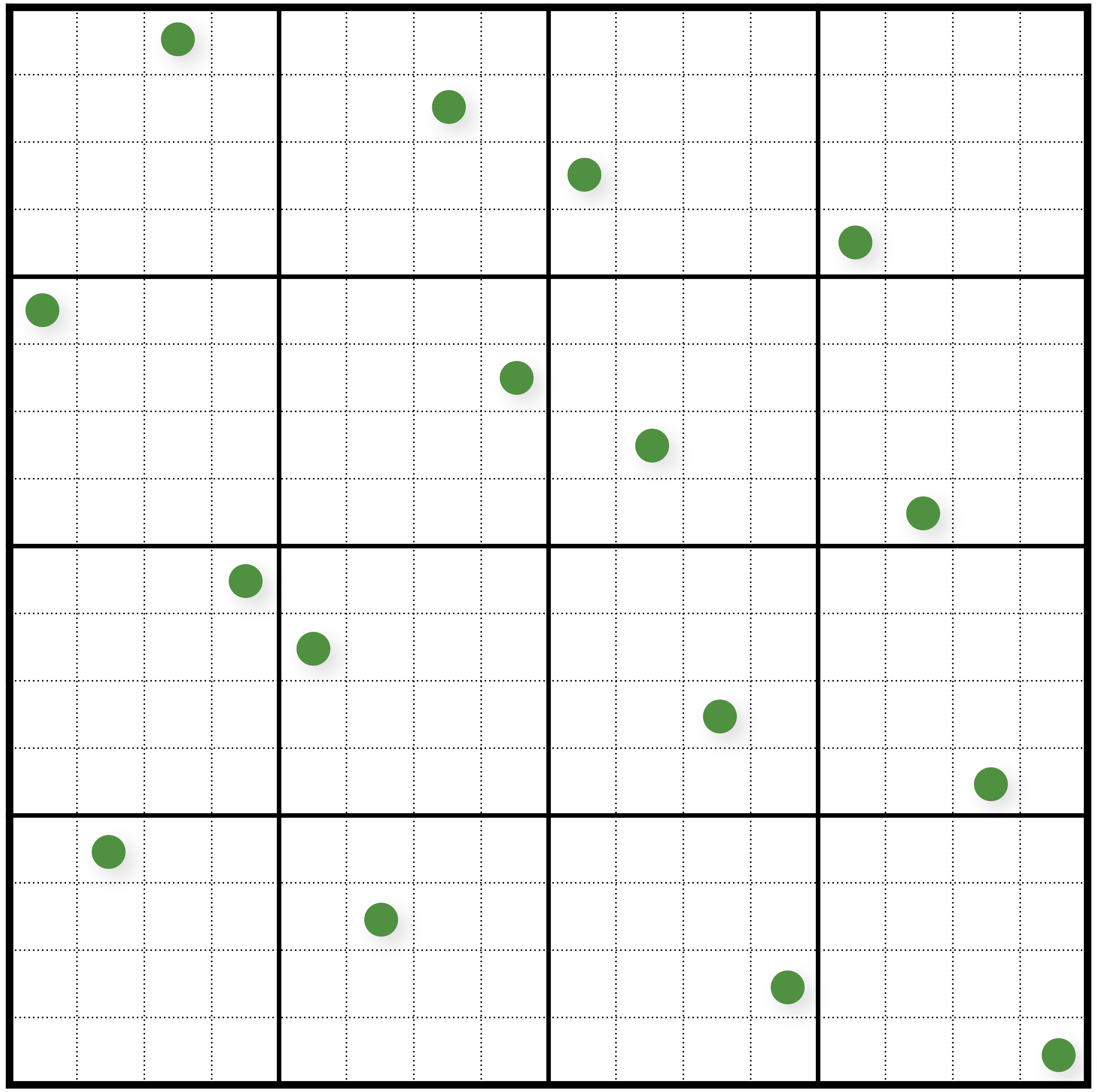
Shuffle x-coords

Multi-Jittered Sampling



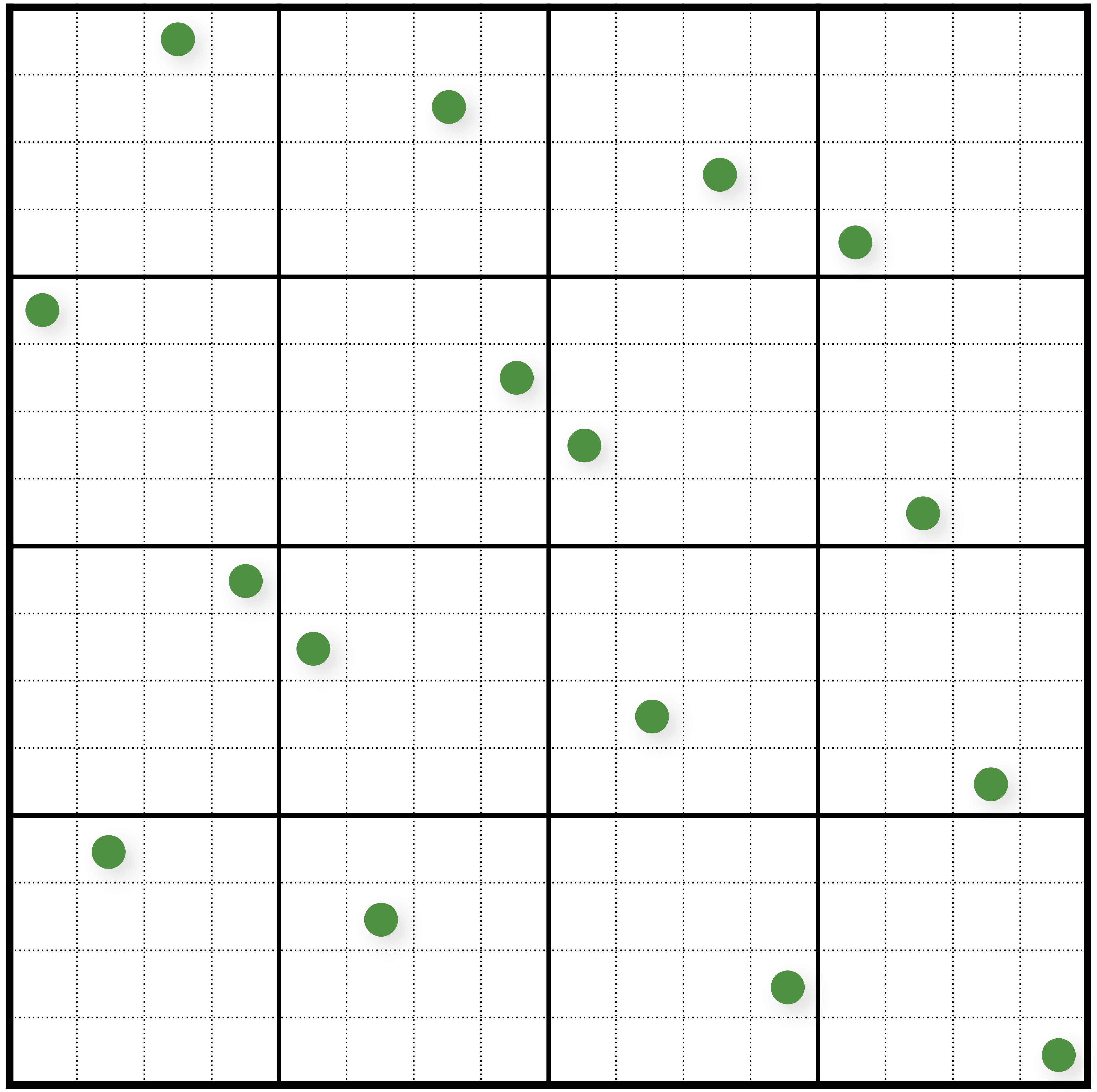
Shuffle x-coords

Multi-Jittered Sampling



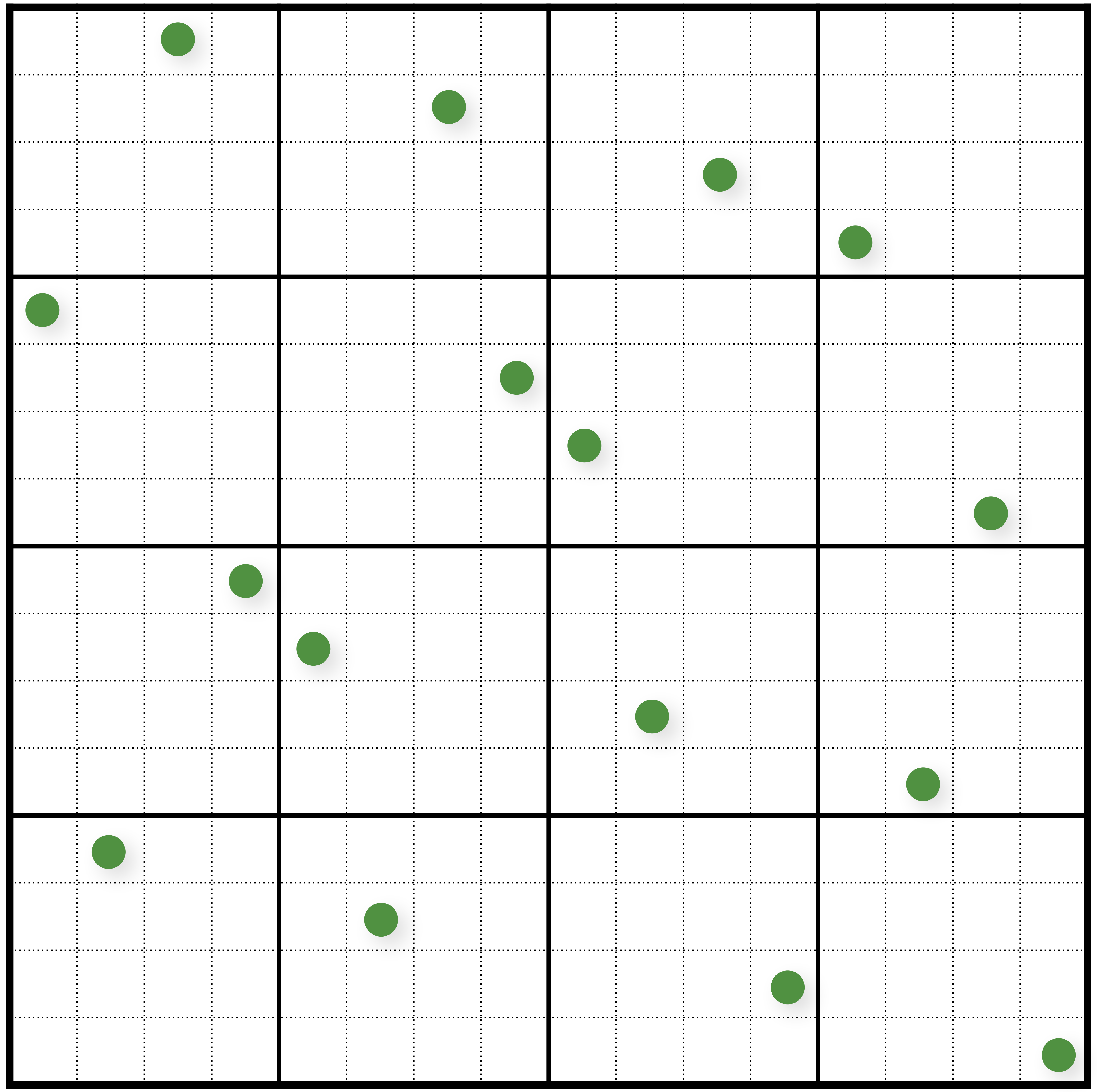
Shuffle x-coords

Multi-Jittered Sampling



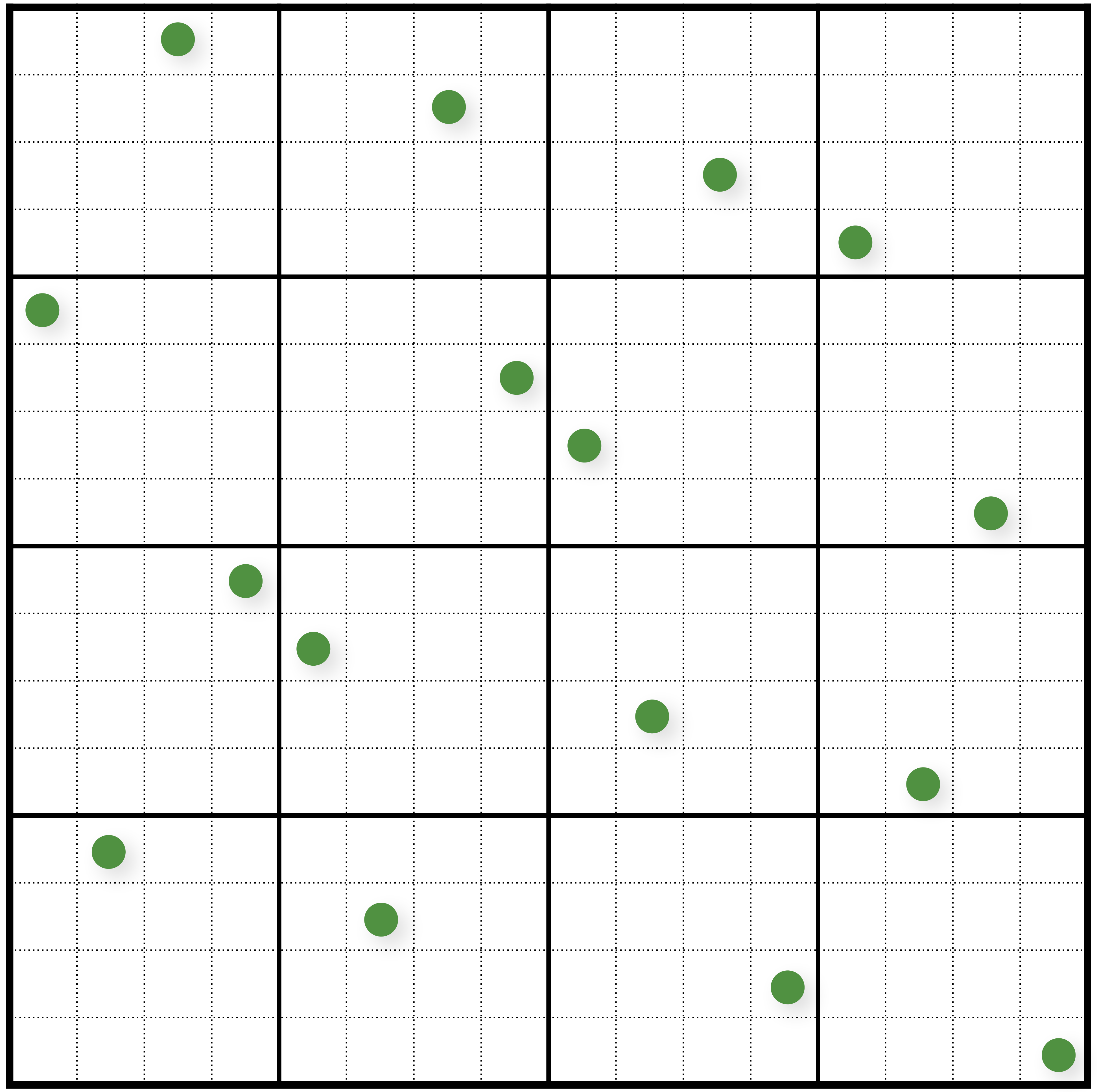
Shuffle x-coords

Multi-Jittered Sampling

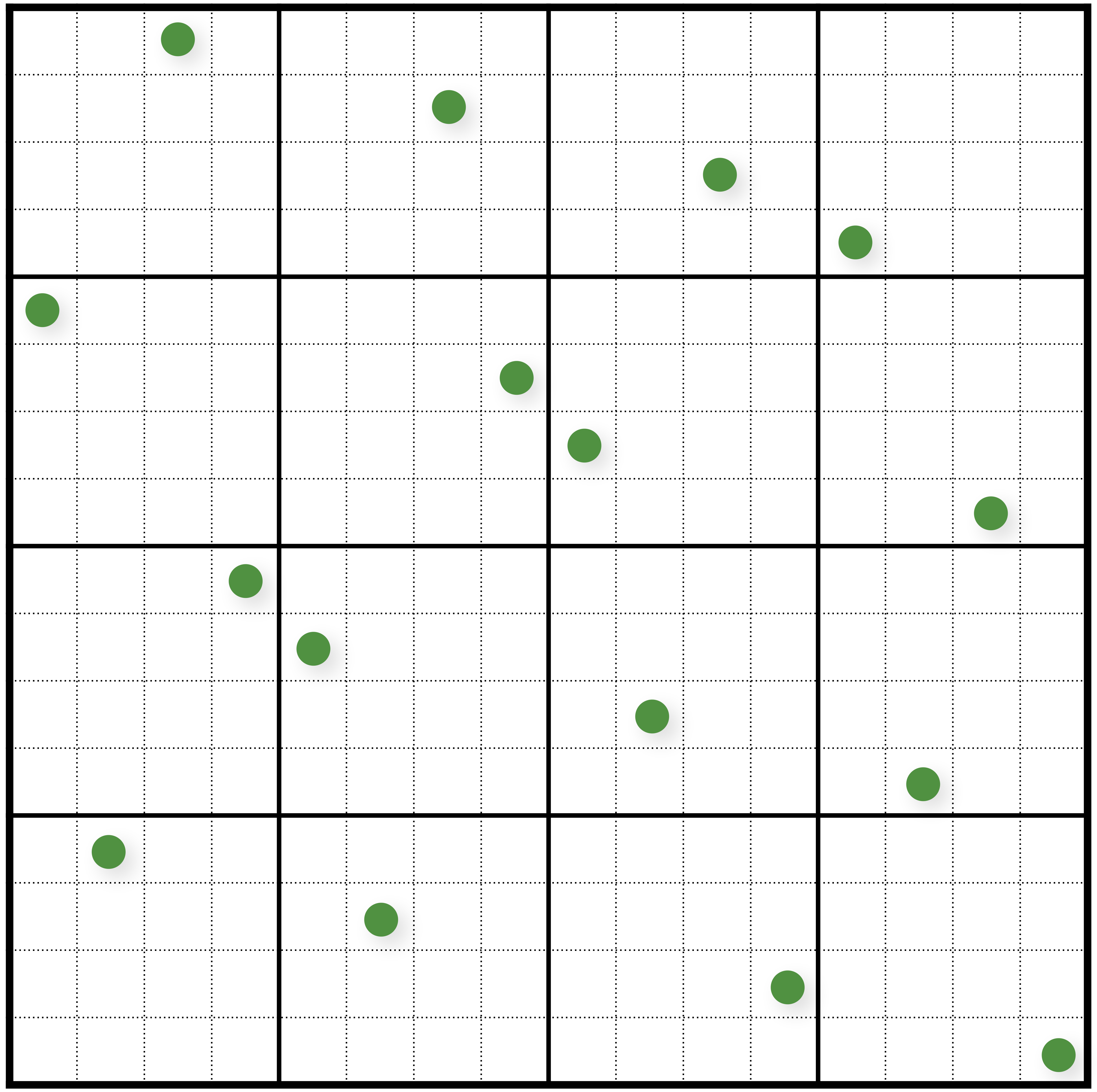


Shuffle x-coords

Multi-Jittered Sampling

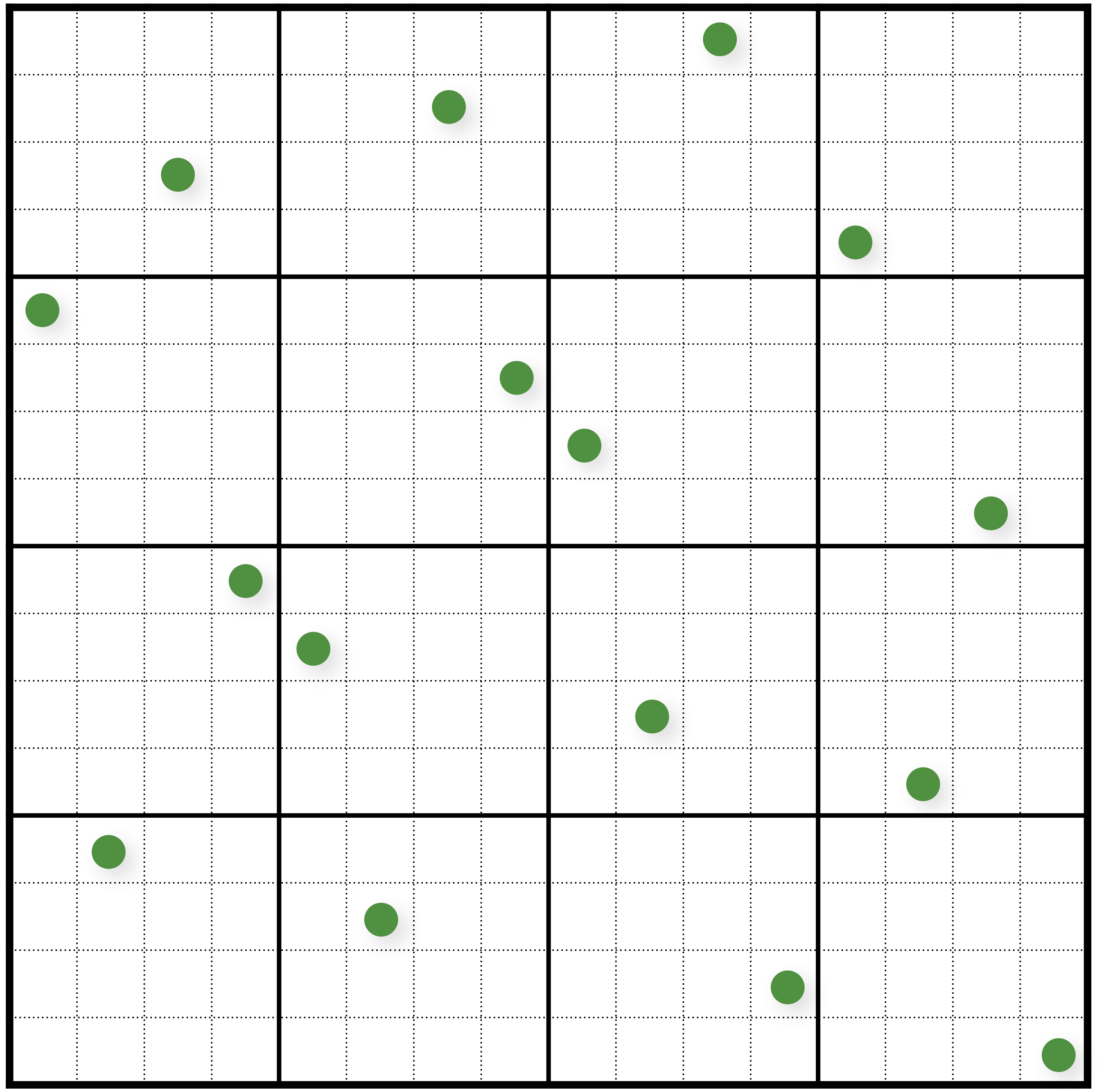


Multi-Jittered Sampling



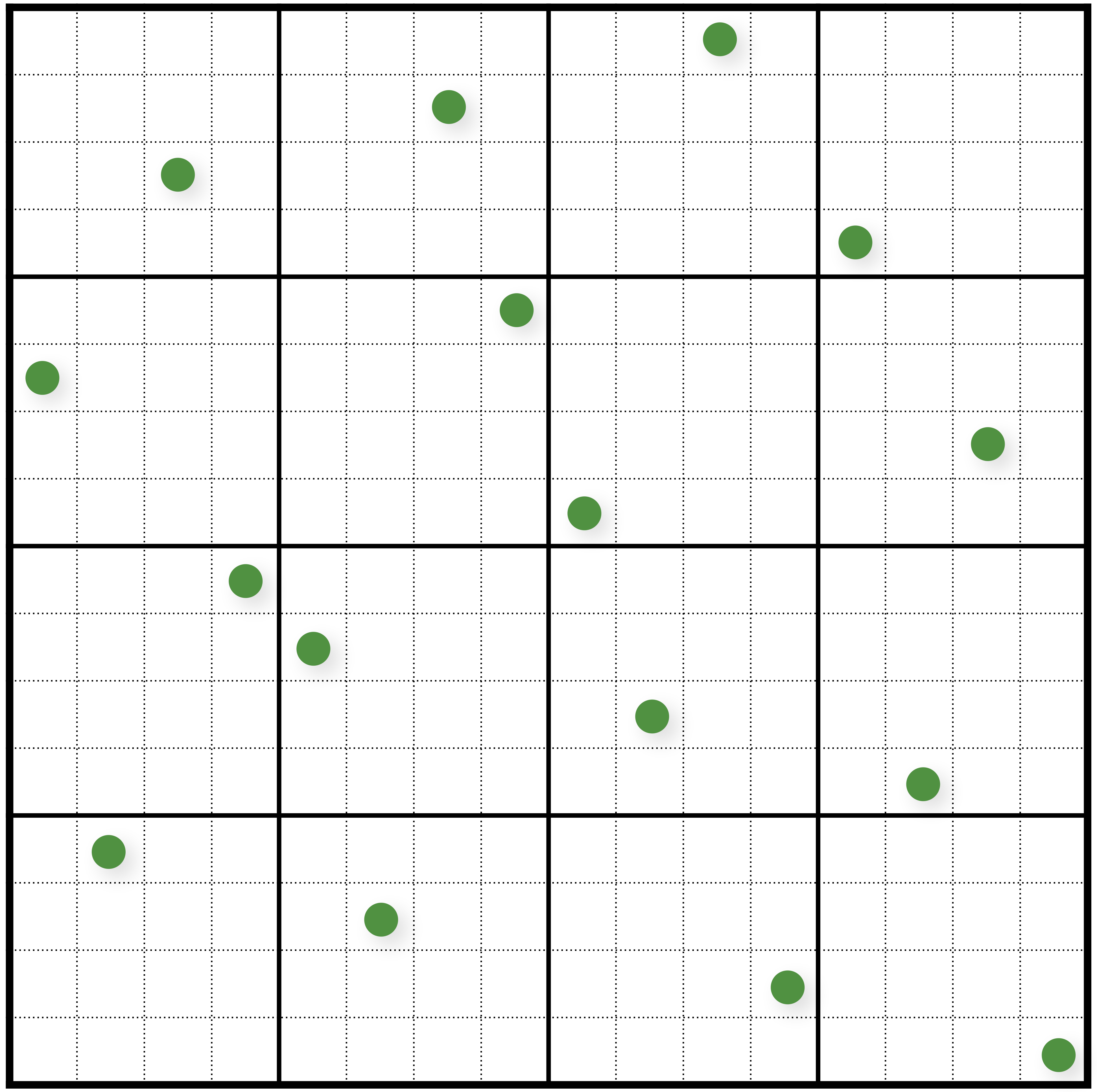
Shuffle y-coords

Multi-Jittered Sampling



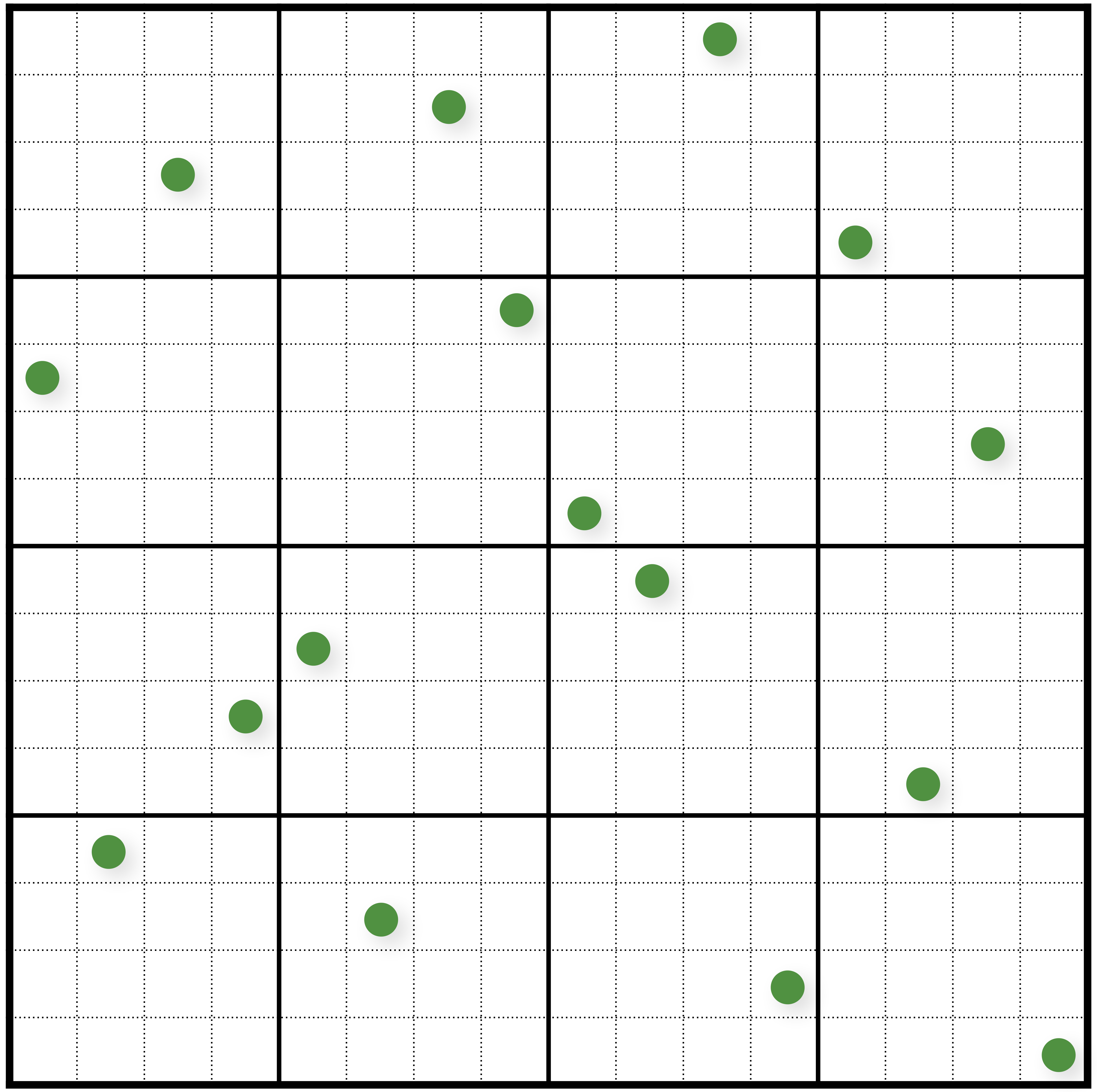
Shuffle y-coords

Multi-Jittered Sampling



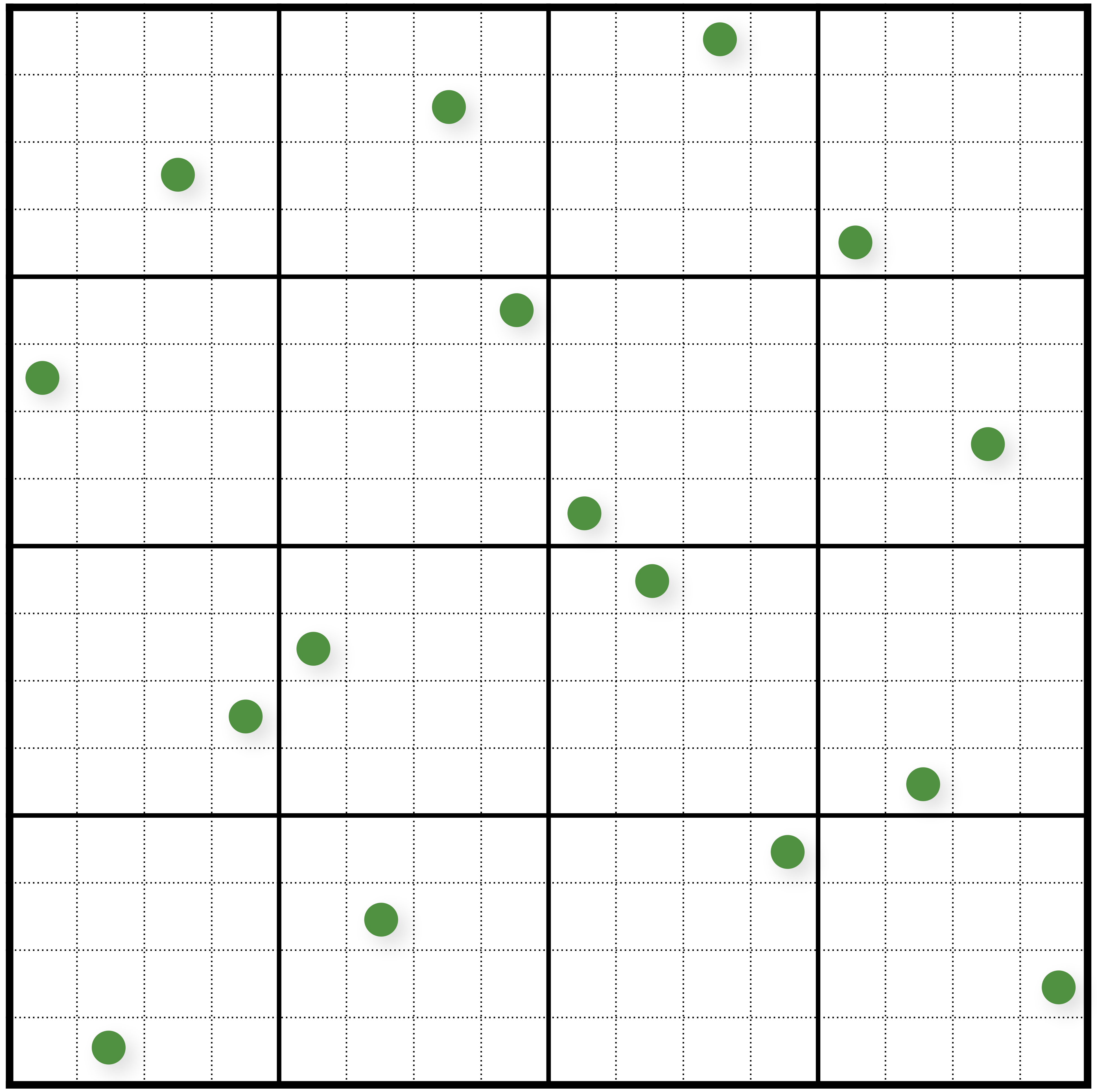
Shuffle y-coords

Multi-Jittered Sampling



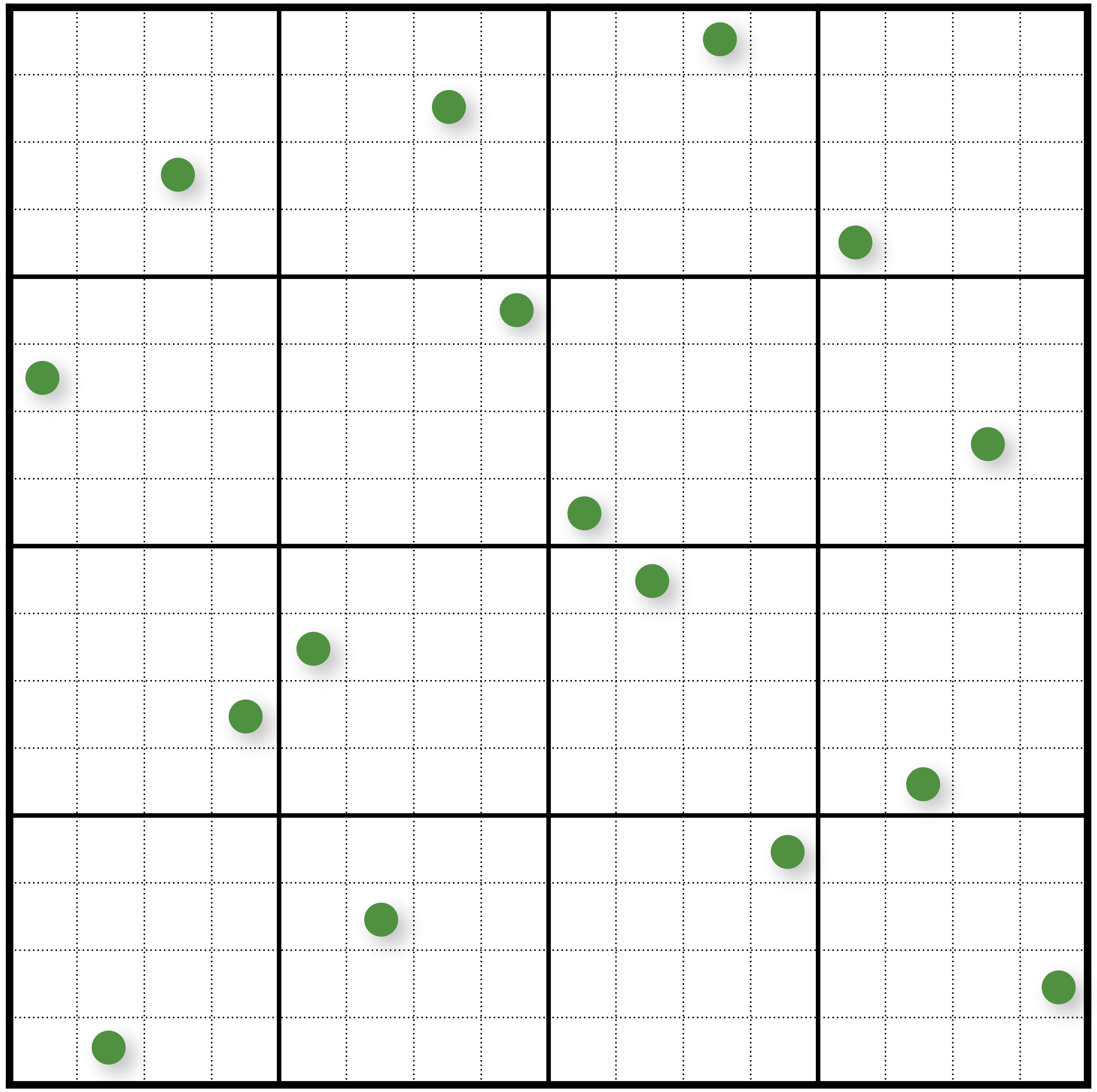
Shuffle y-coords

Multi-Jittered Sampling

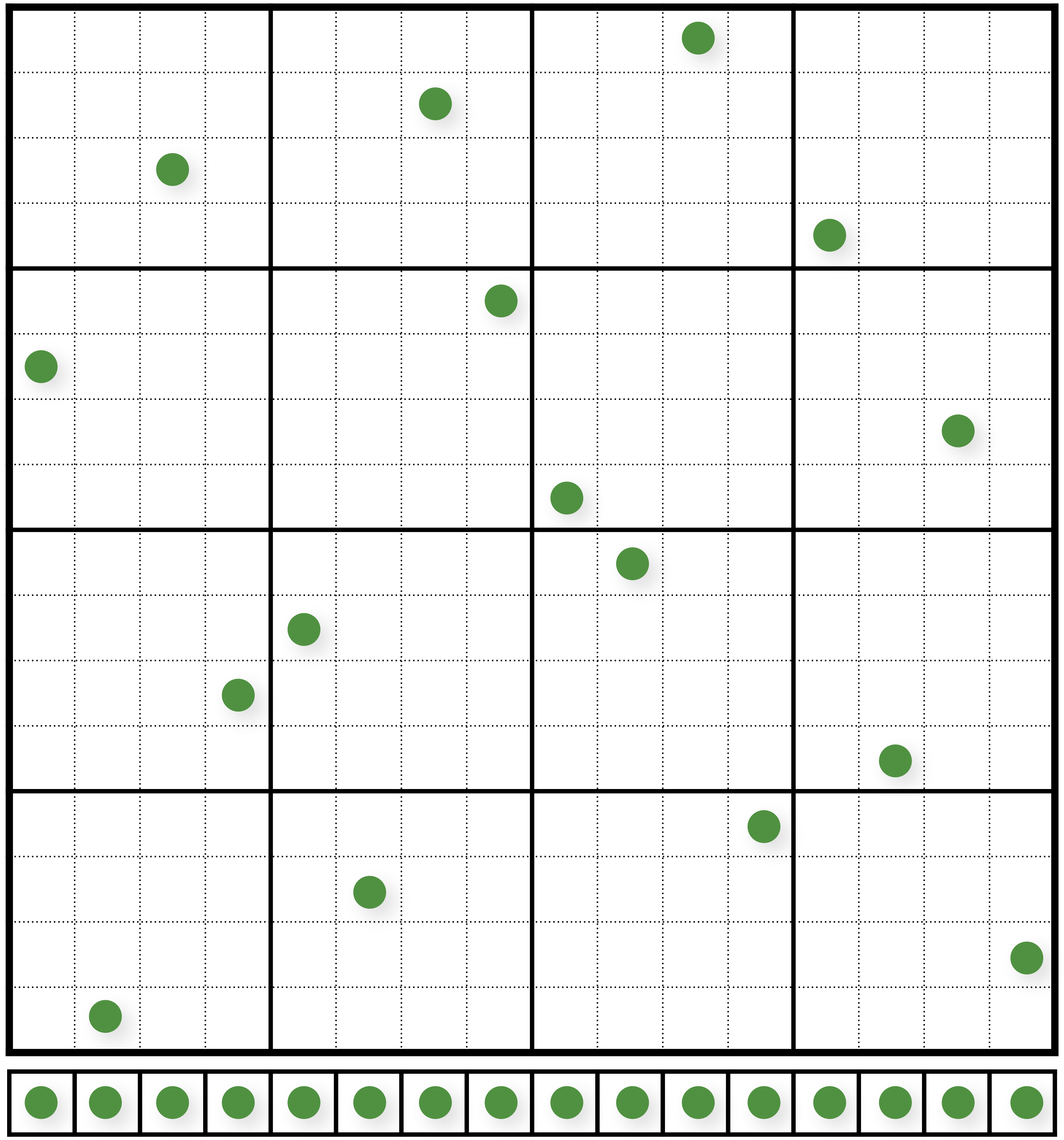


Shuffle y-coords

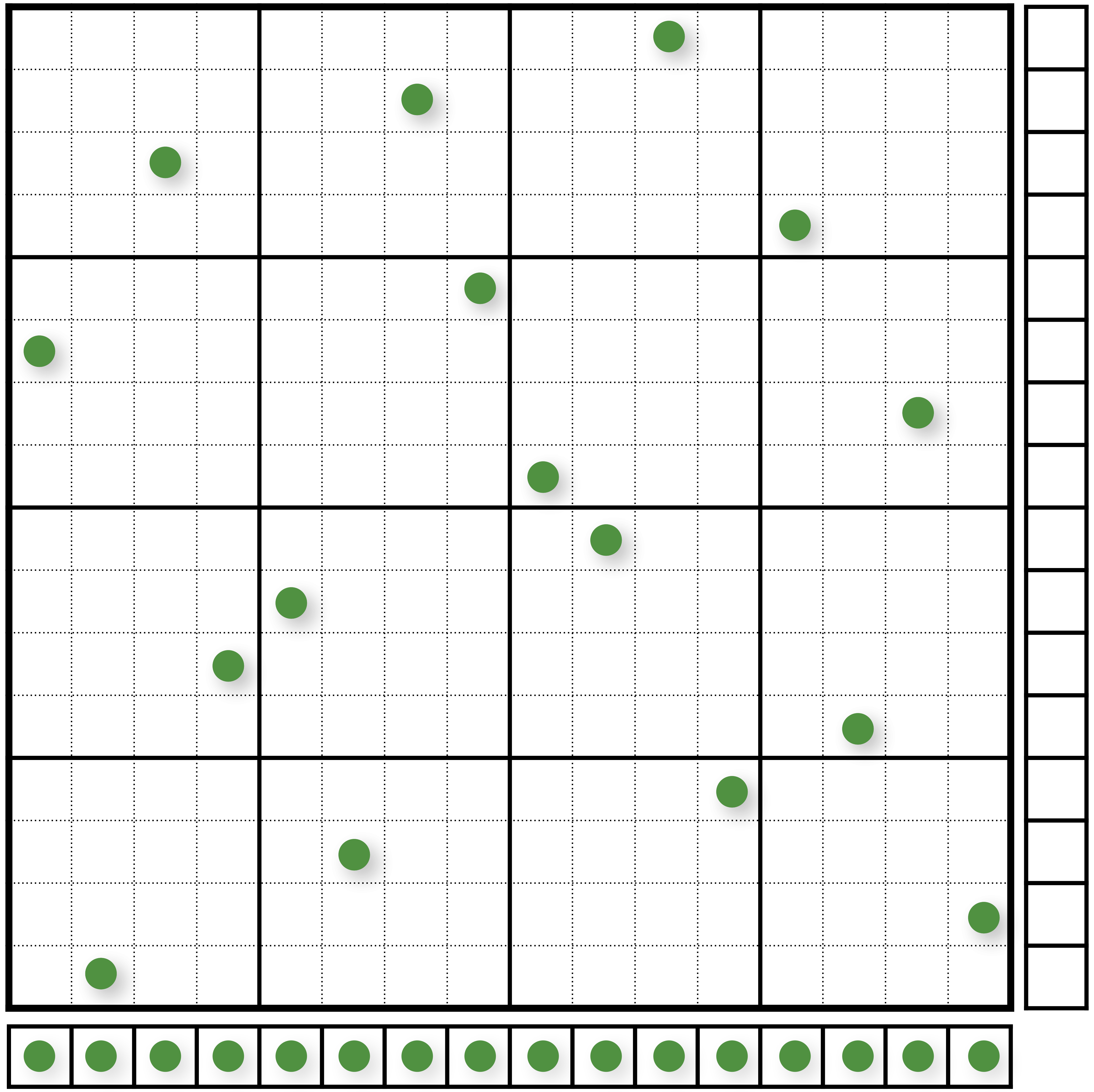
Multi-Jittered Sampling (Projections)



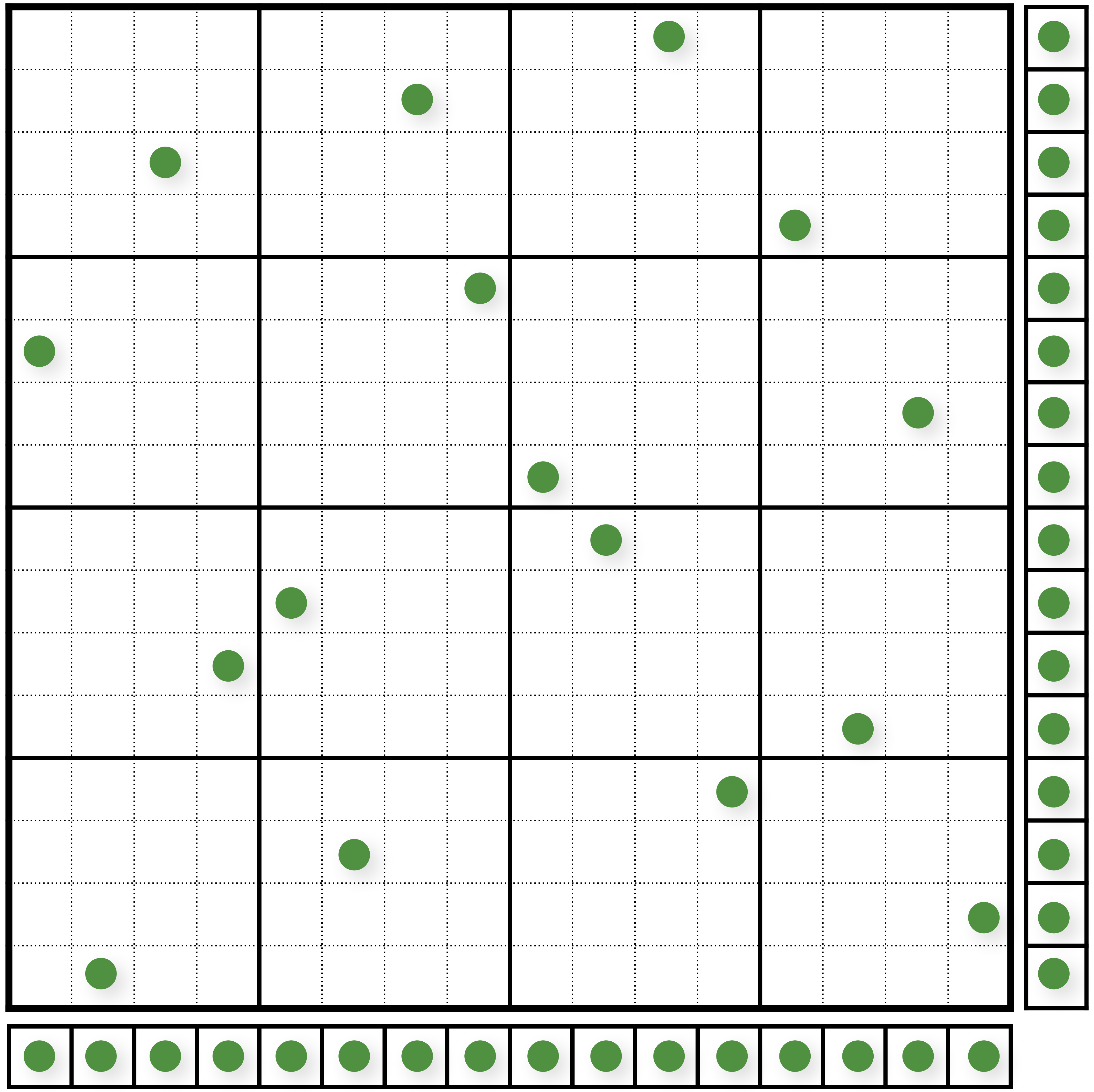
Multi-Jittered Sampling (Projections)



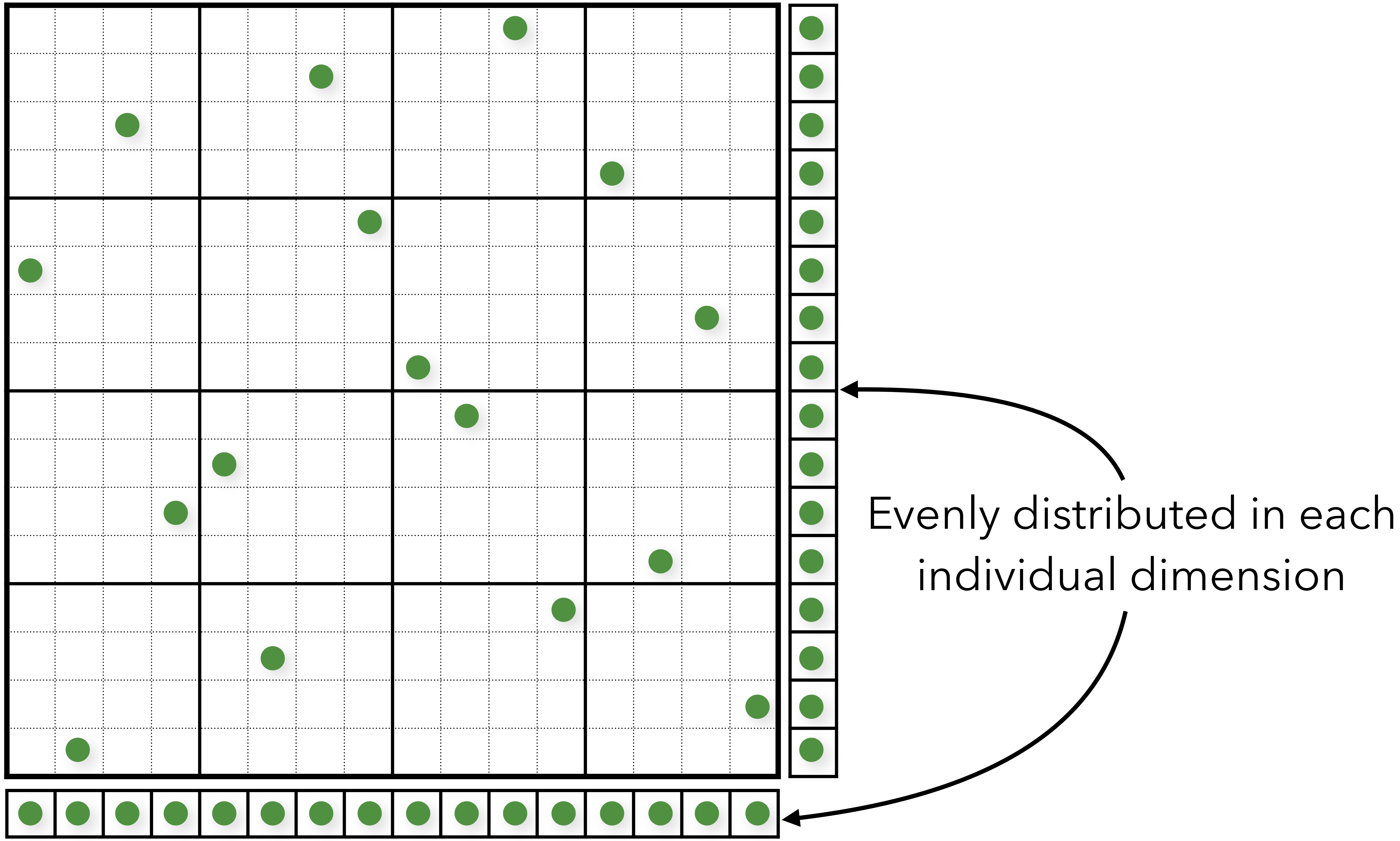
Multi-Jittered Sampling (Projections)



Multi-Jittered Sampling (Projections)

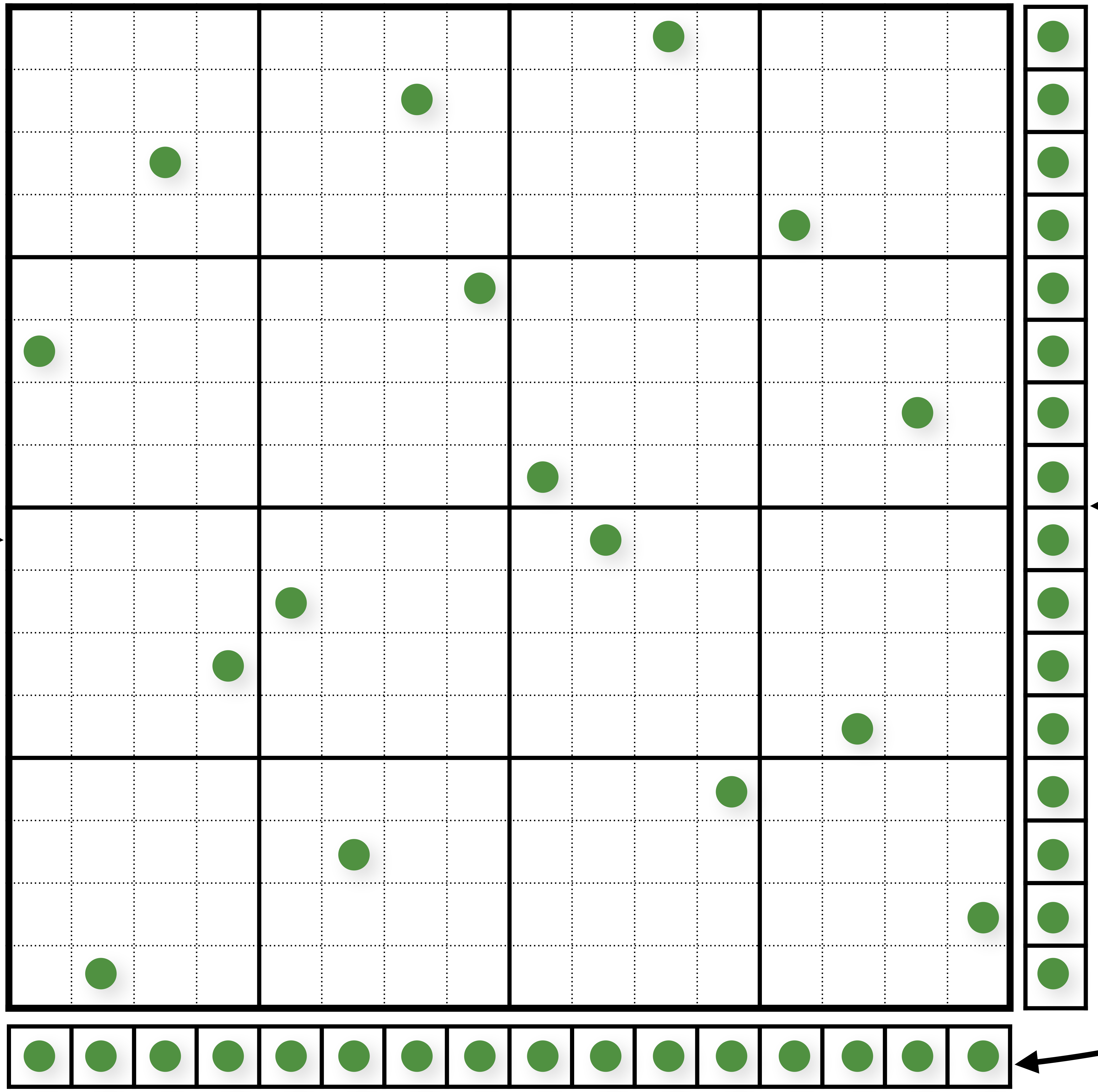


Multi-Jittered Sampling (Projections)



Multi-Jittered Sampling (Projections)

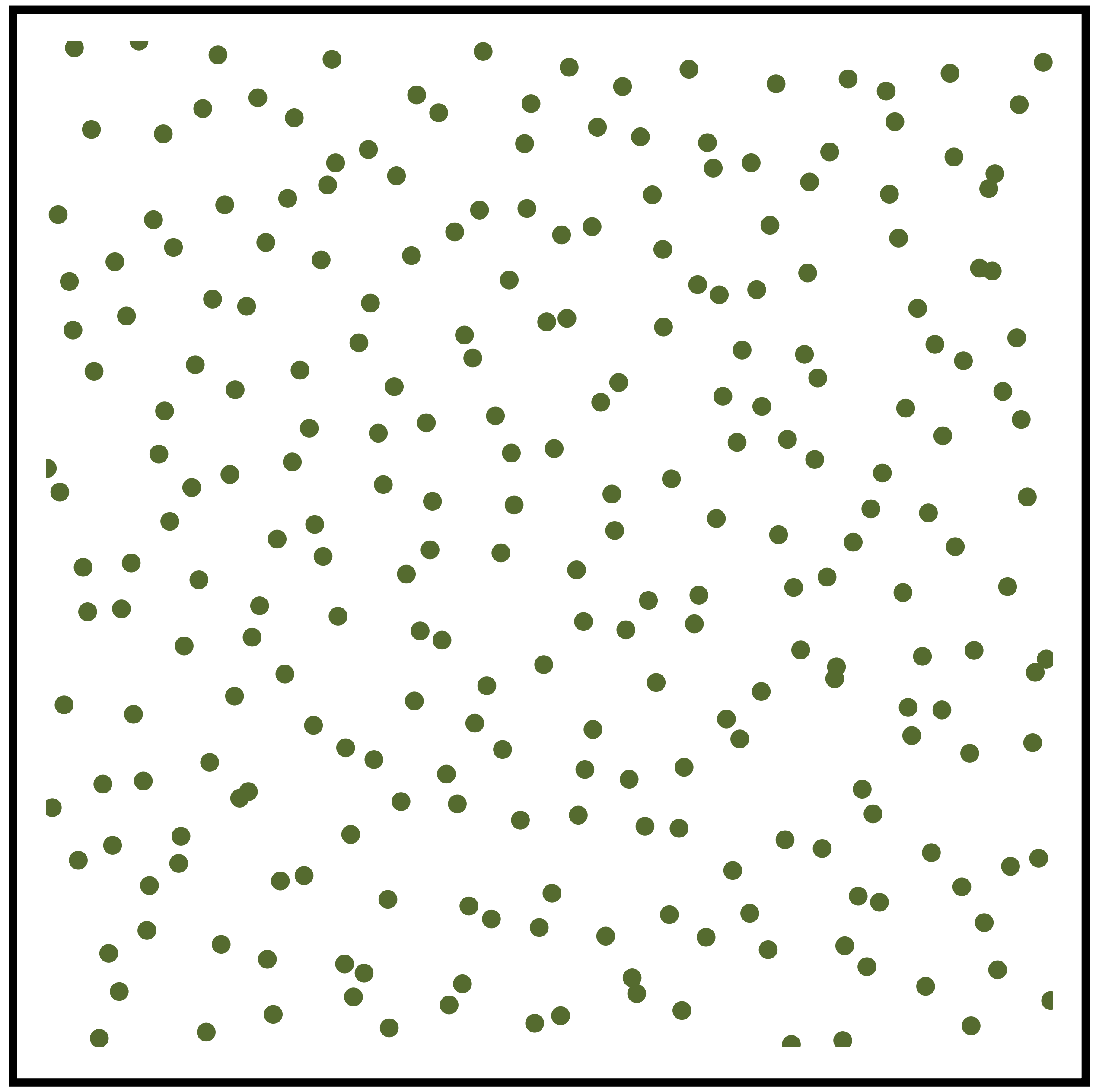
Evenly distributed in 2D!



Evenly distributed in each individual dimension

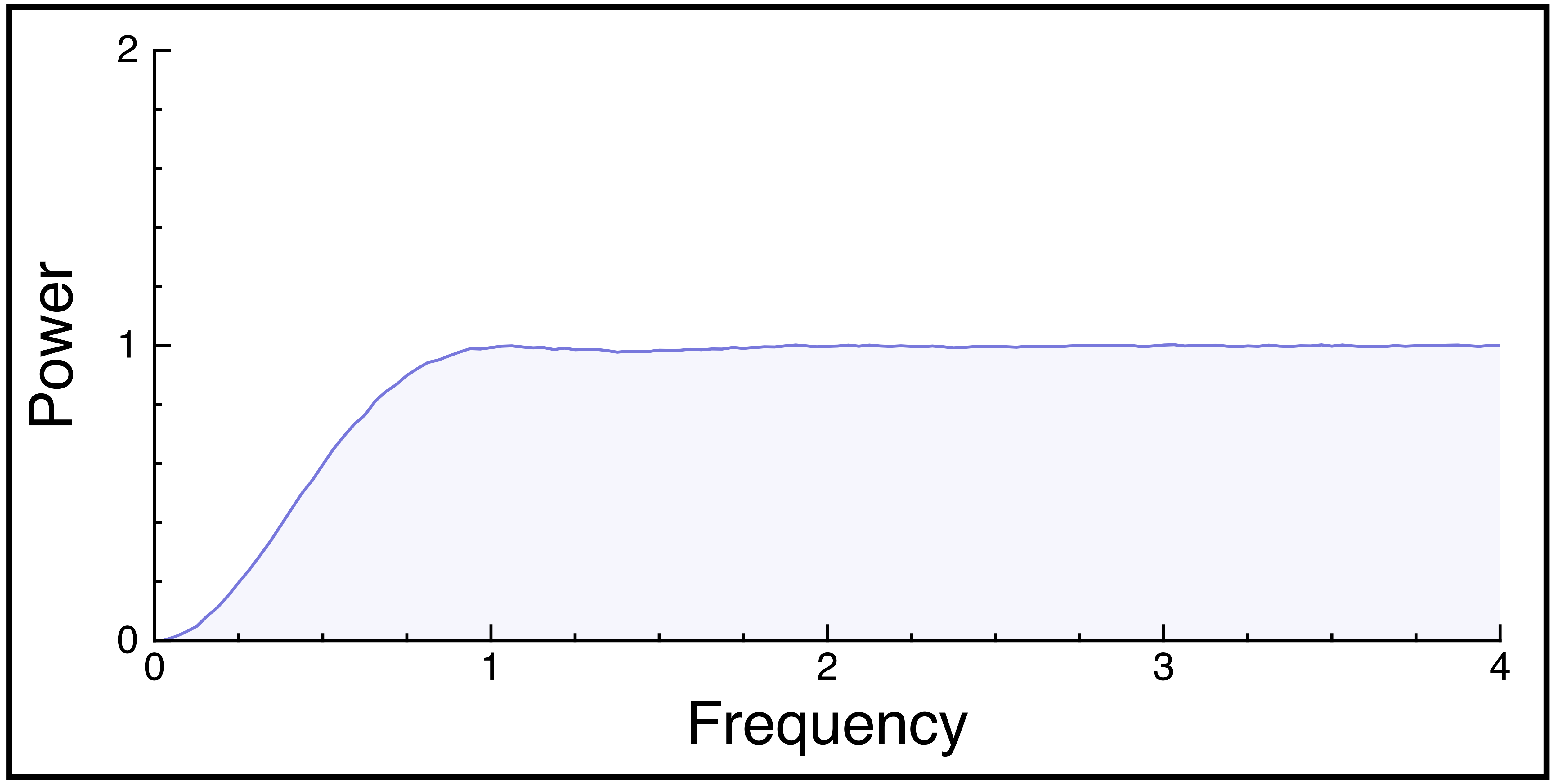
Multi-Jittered Sampling

Samples



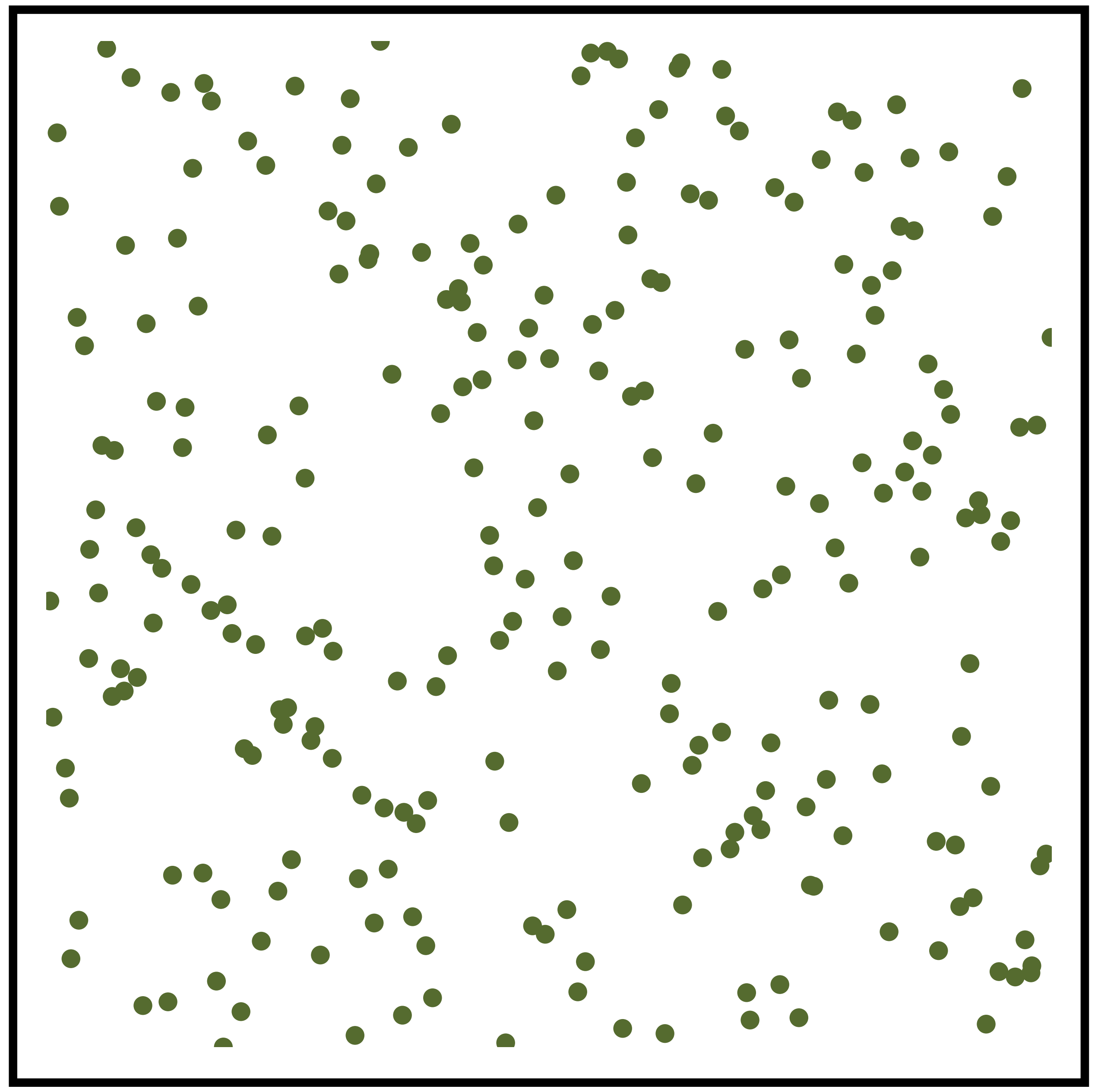
Expected power spectrum

Radial mean



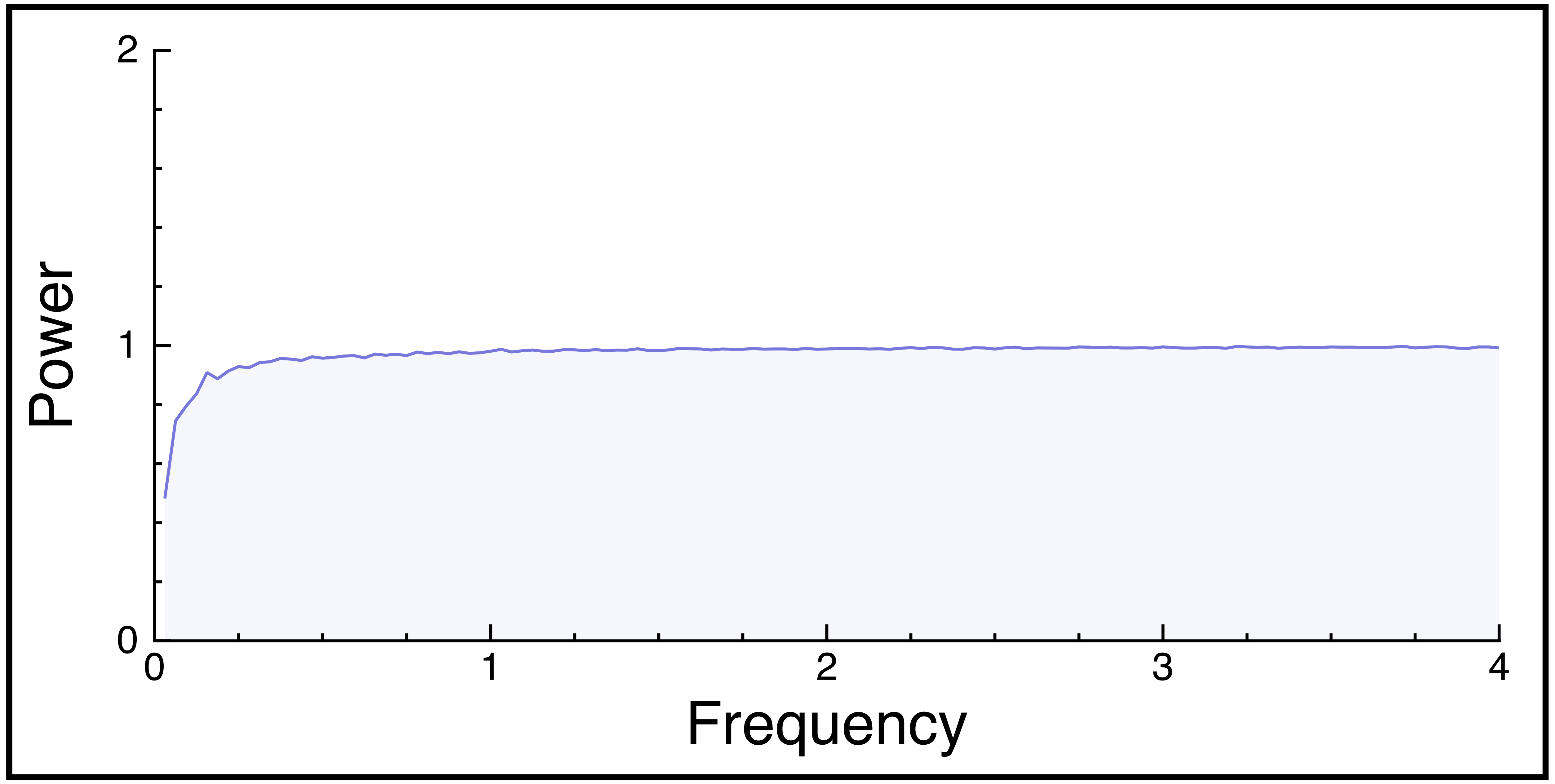
N-Rooks Sampling

Samples



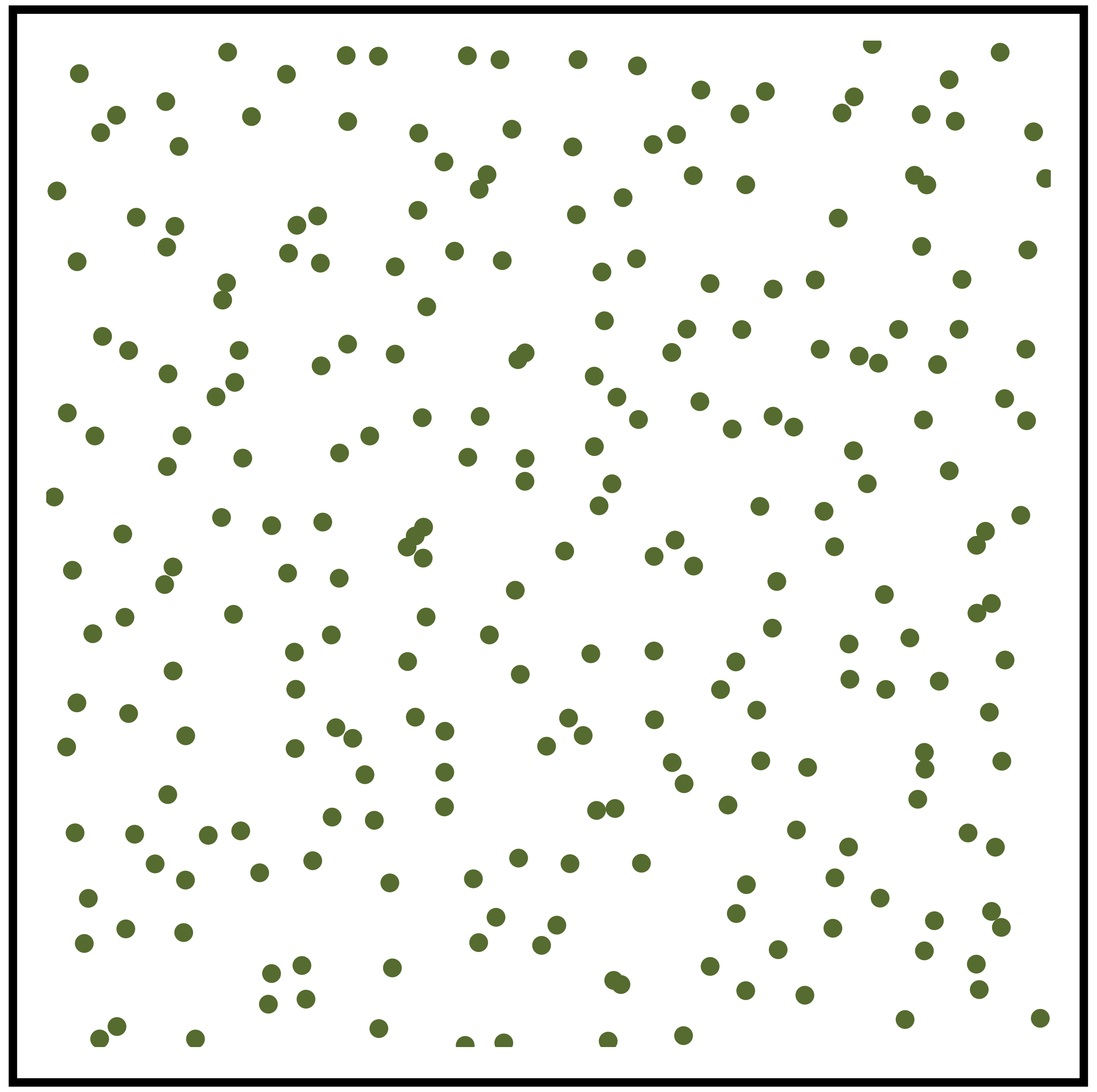
Expected power spectrum

Radial mean



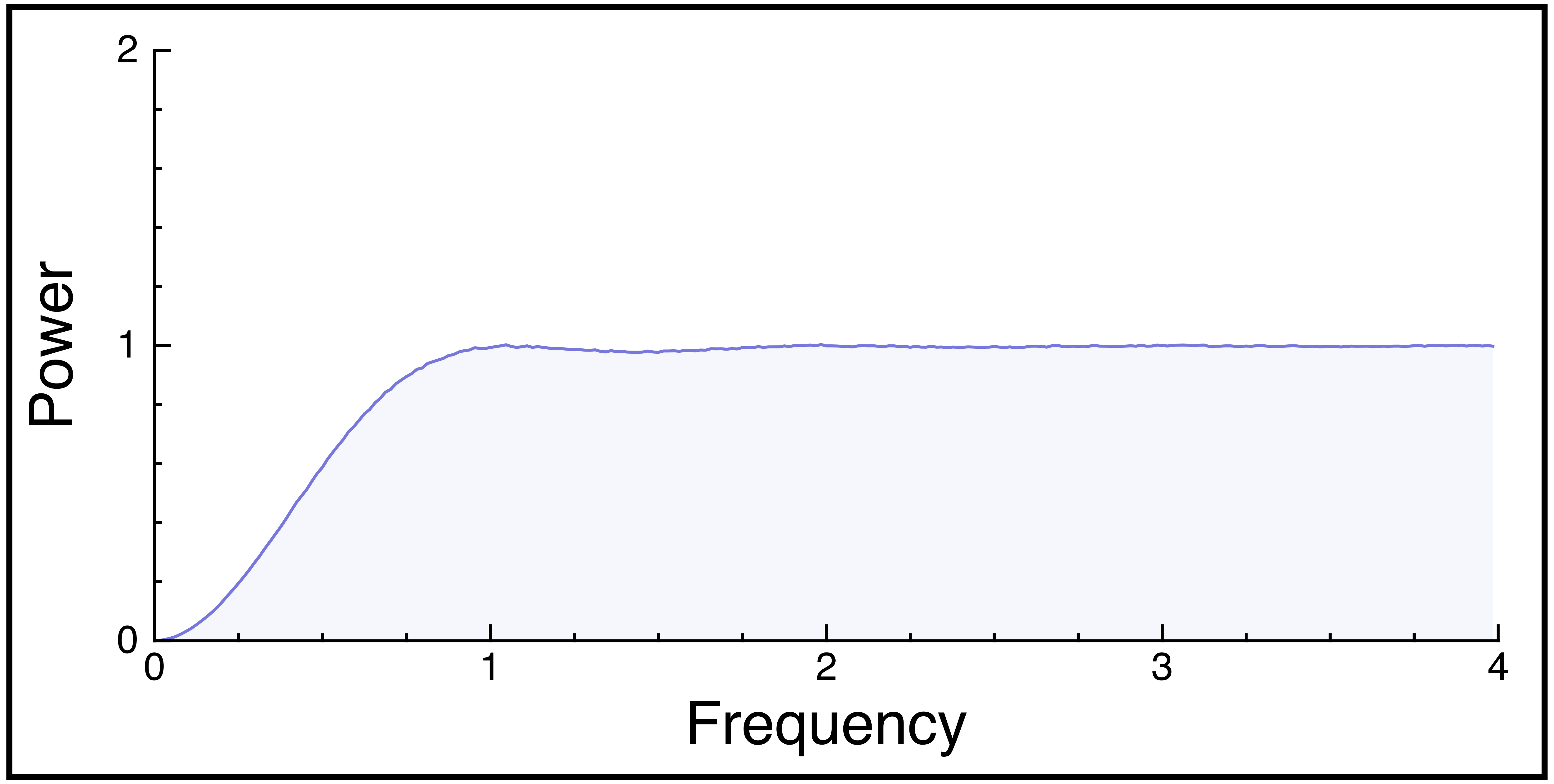
Jittered Sampling

Samples



Expected power spectrum

Radial mean



Poisson-Disk/Blue-Noise Sampling

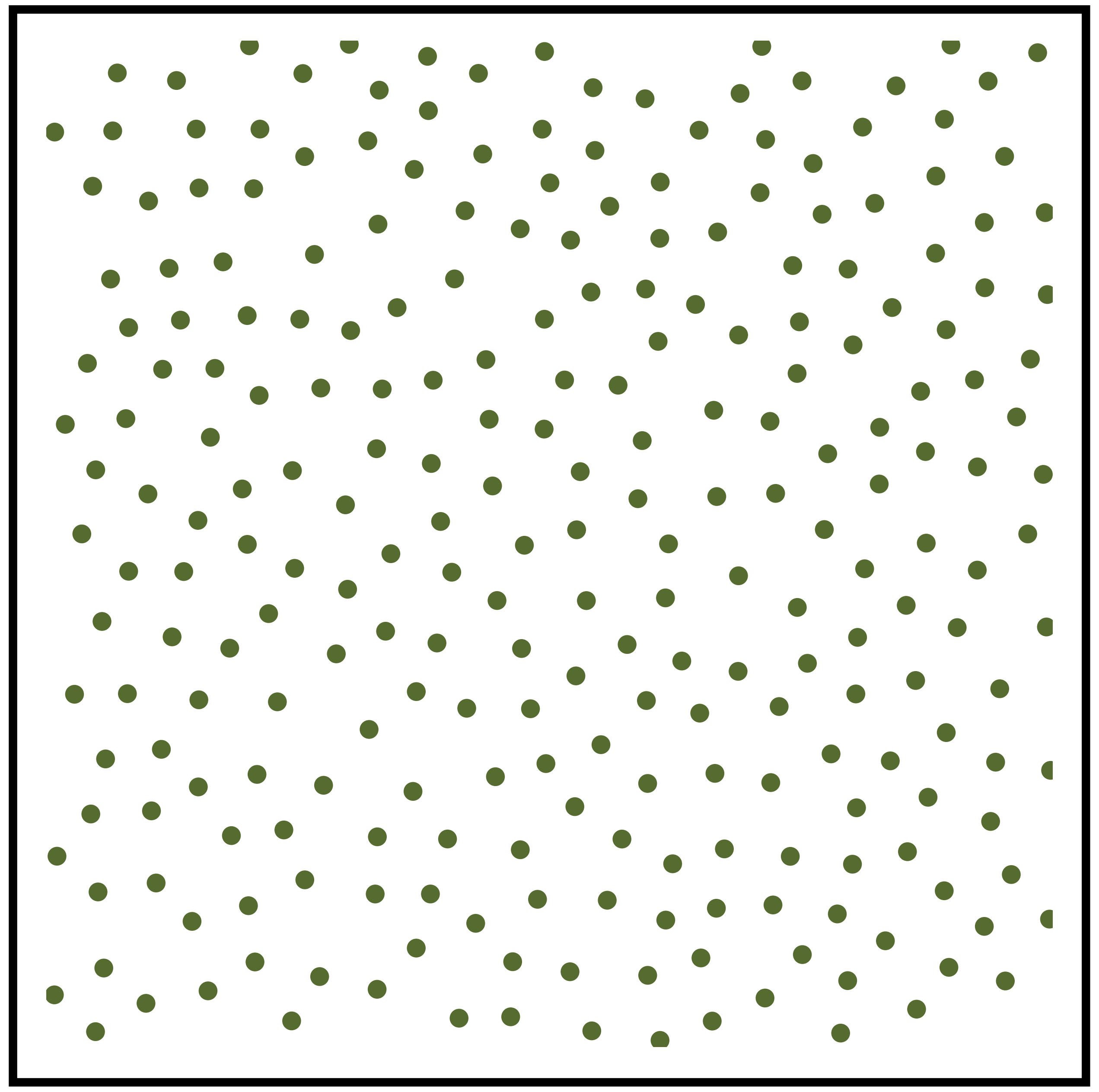
Enforce a minimum distance between points

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.

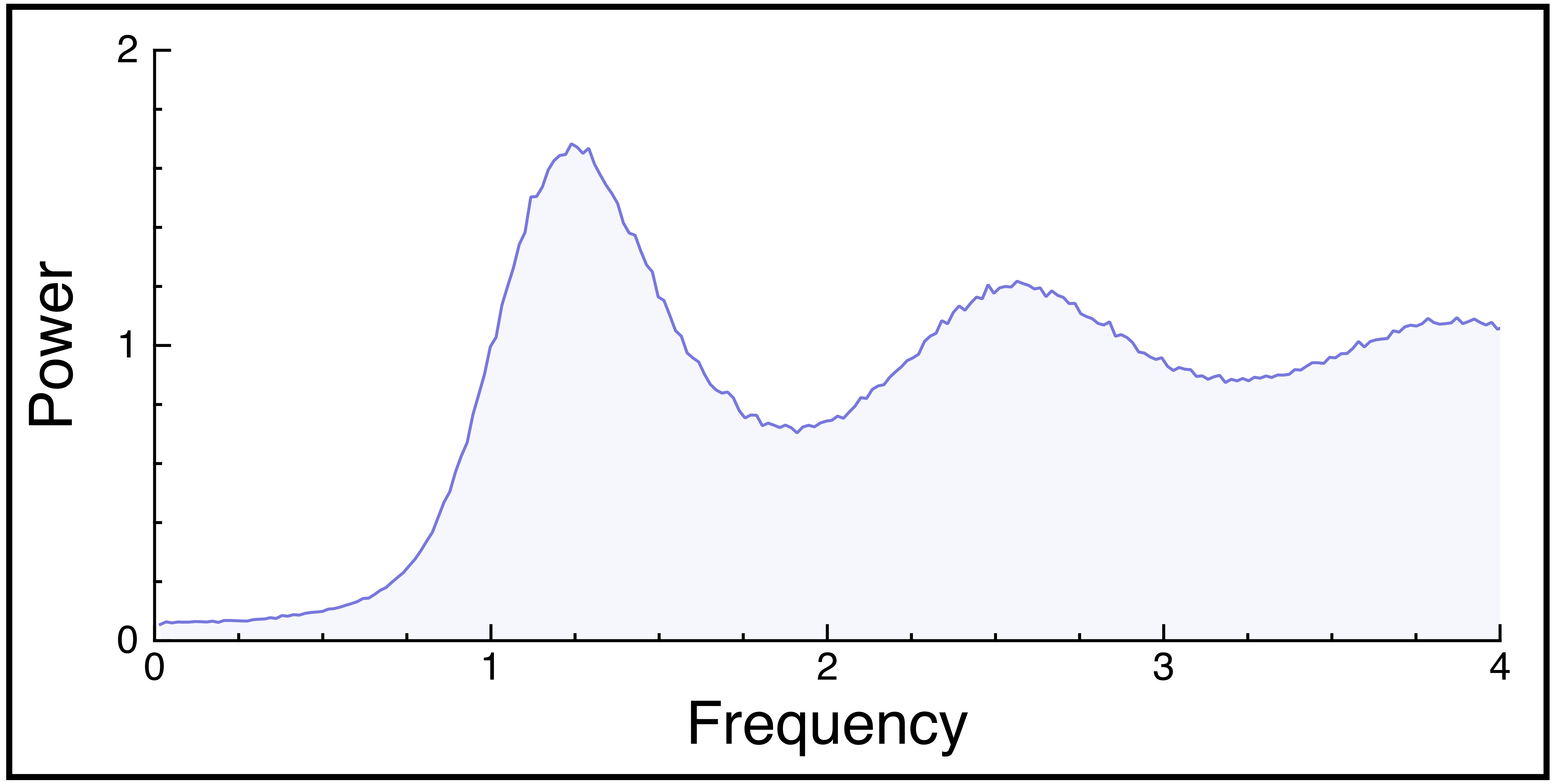
Poisson Disk Sampling

Samples



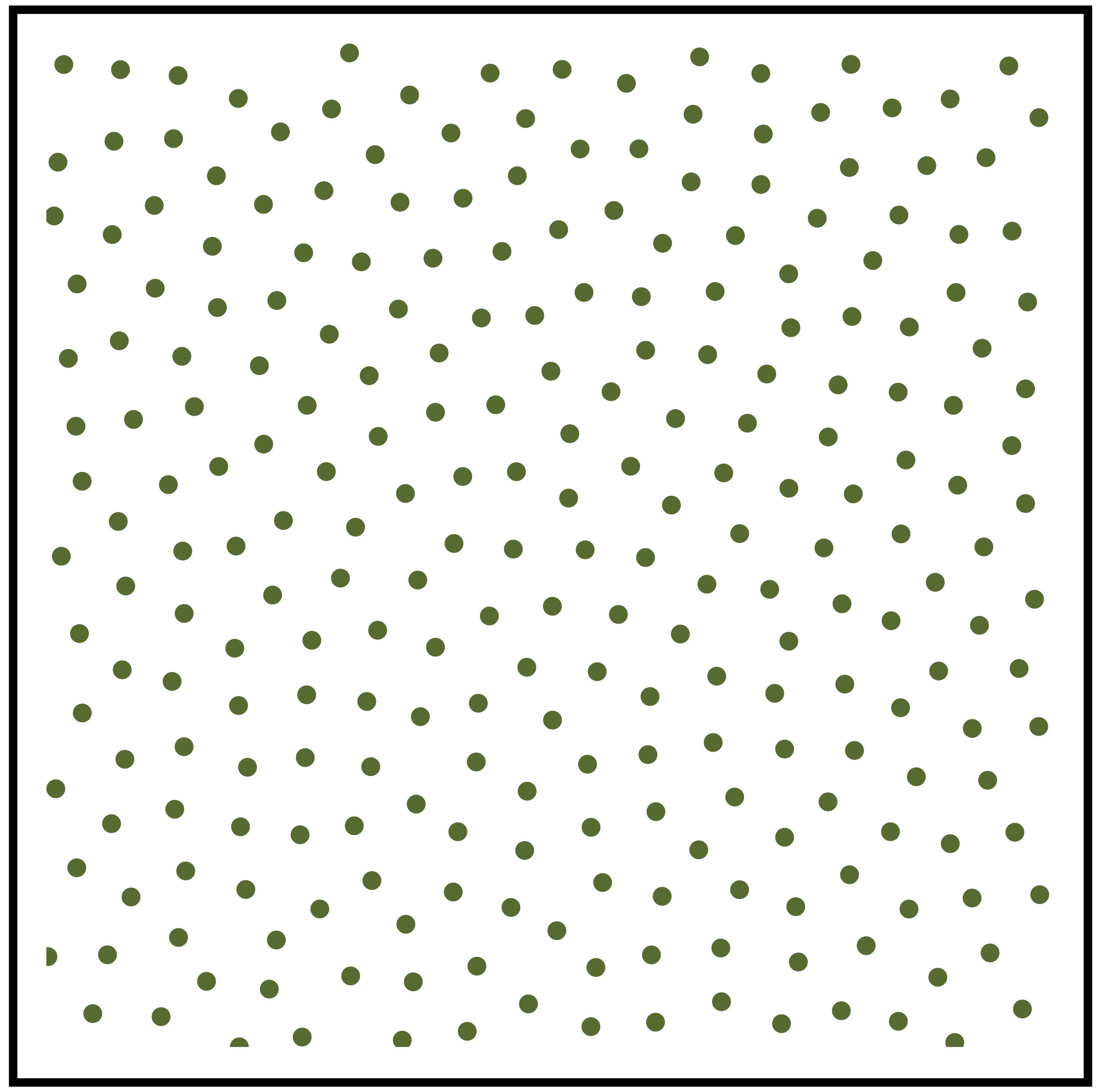
Expected power spectrum

Radial mean



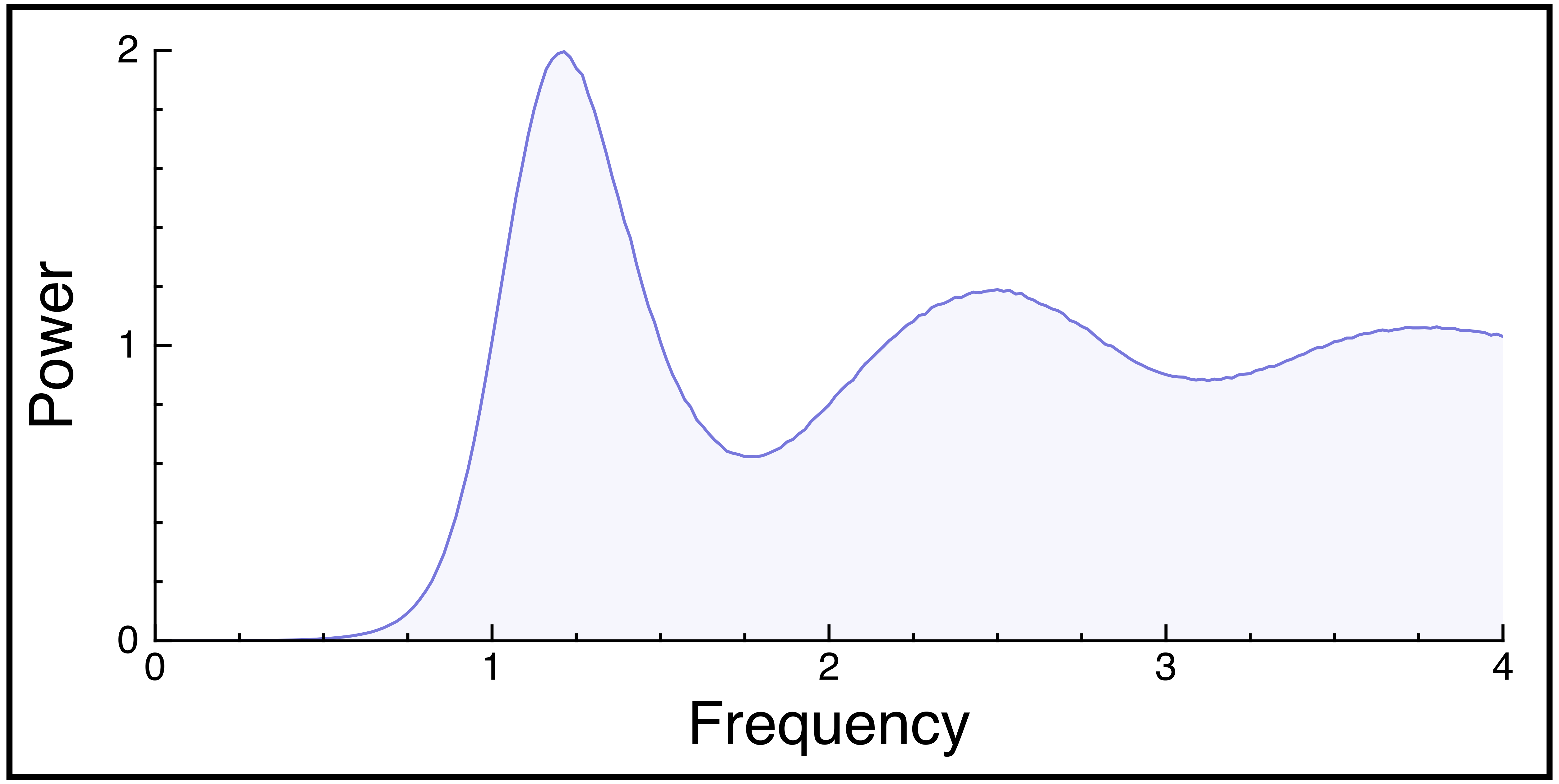
CCVT Sampling [Balzer et al. 2009]

Samples



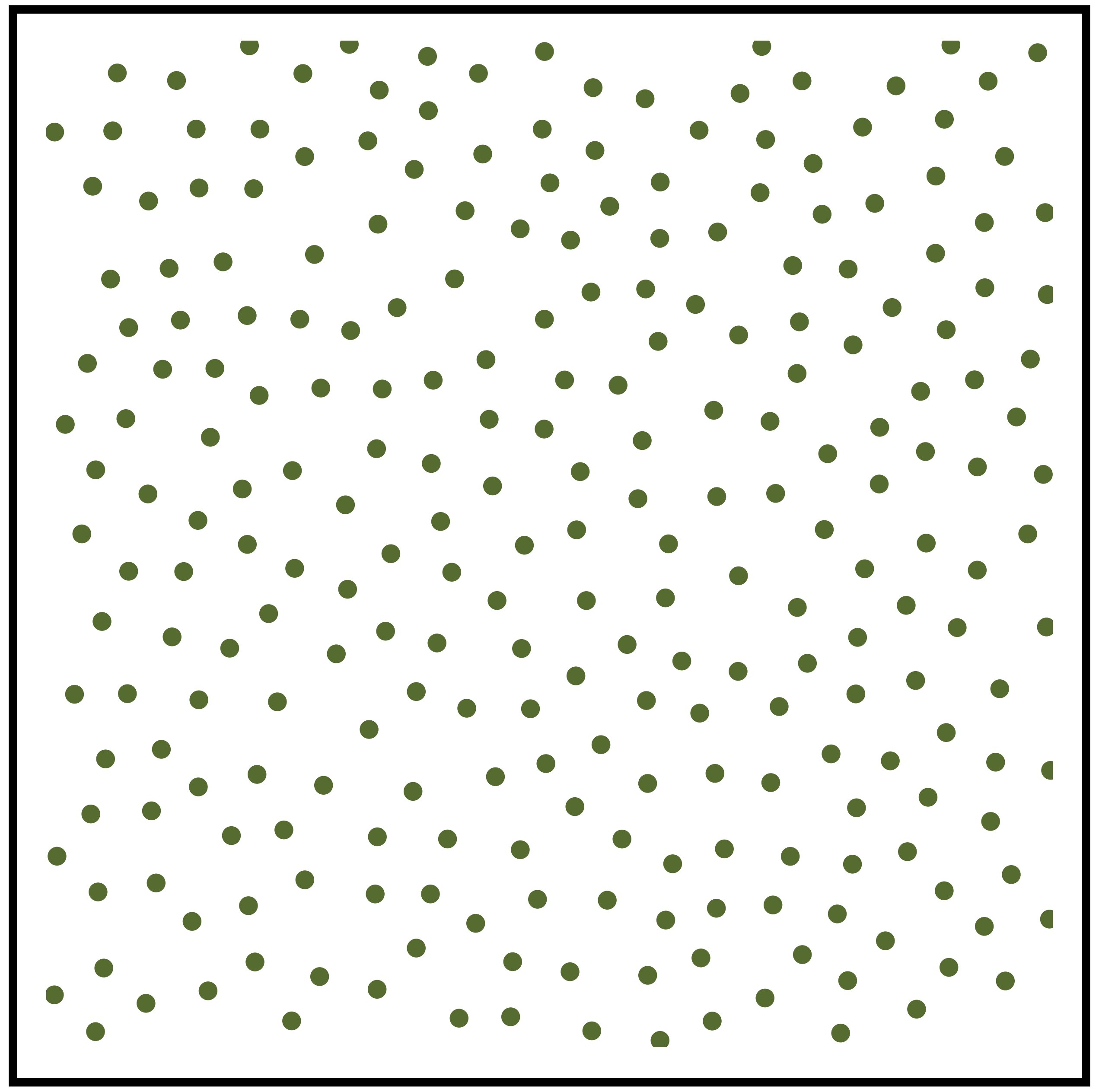
Expected power spectrum

Radial mean



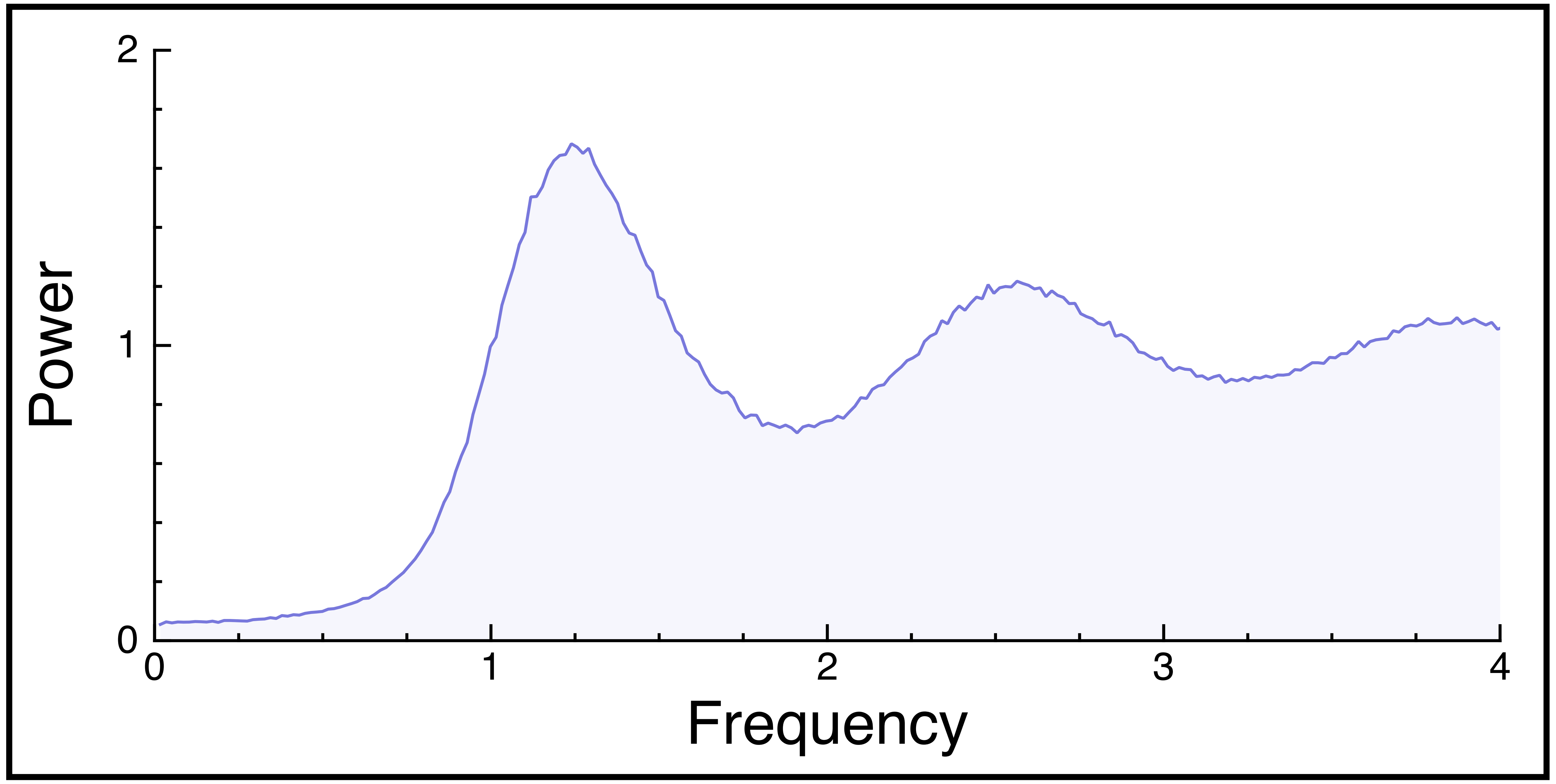
Poisson Disk Sampling

Samples



Expected power spectrum

Radial mean



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 Van der Corput Sequence

Radical Inverse Φ_b in base 2

Subsequent points "fall into biggest holes"

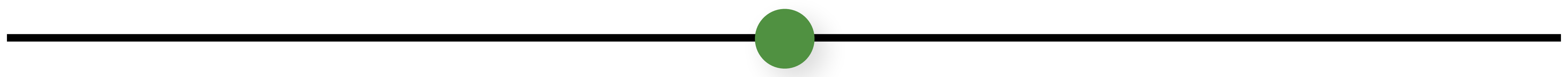
k	Base 2	Φ_b

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

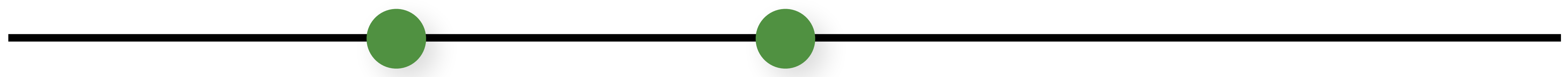


The Van der Corput Sequence

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

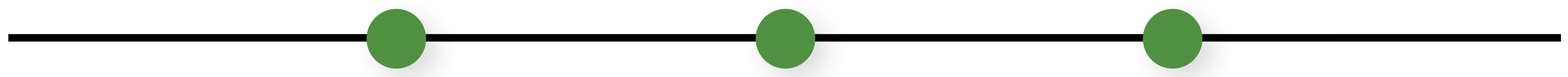


The Van der Corput Sequence

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

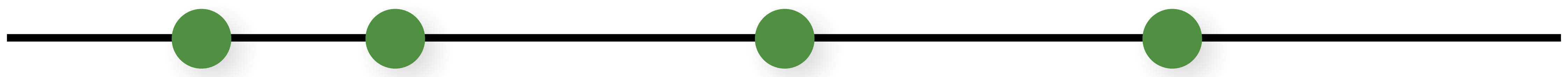


The Van der Corput Sequence

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

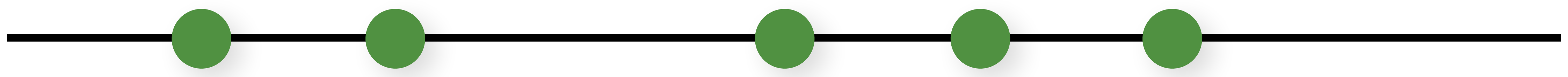


The Van der Corput Sequence

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

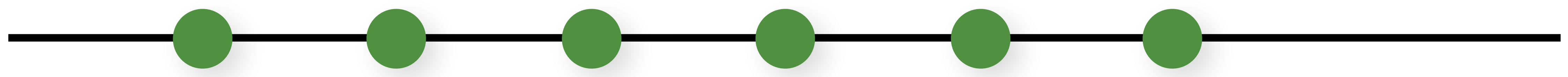


The Van der Corput Sequence

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

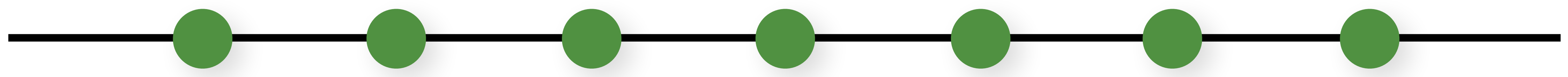


The Van der Corput Sequence

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

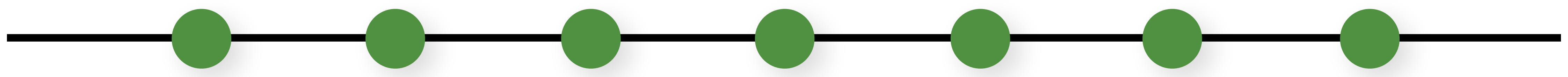


The Van der Corput Sequence

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
...		



Halton and Hammersley Points

Halton: Radical inverse with different base for each dimension:

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

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.

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

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:

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

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.

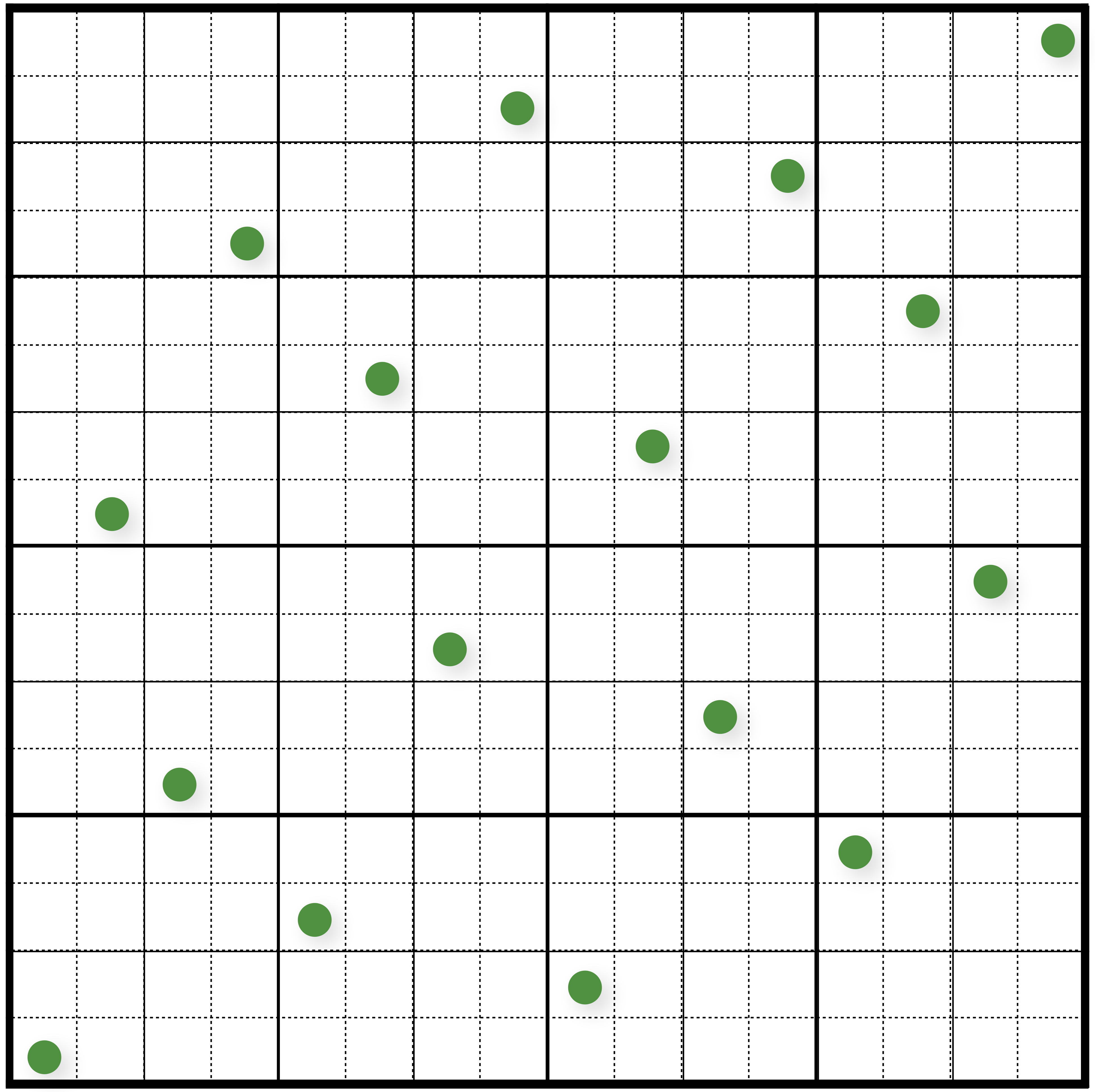
- Incremental/progressive generation of samples

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

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

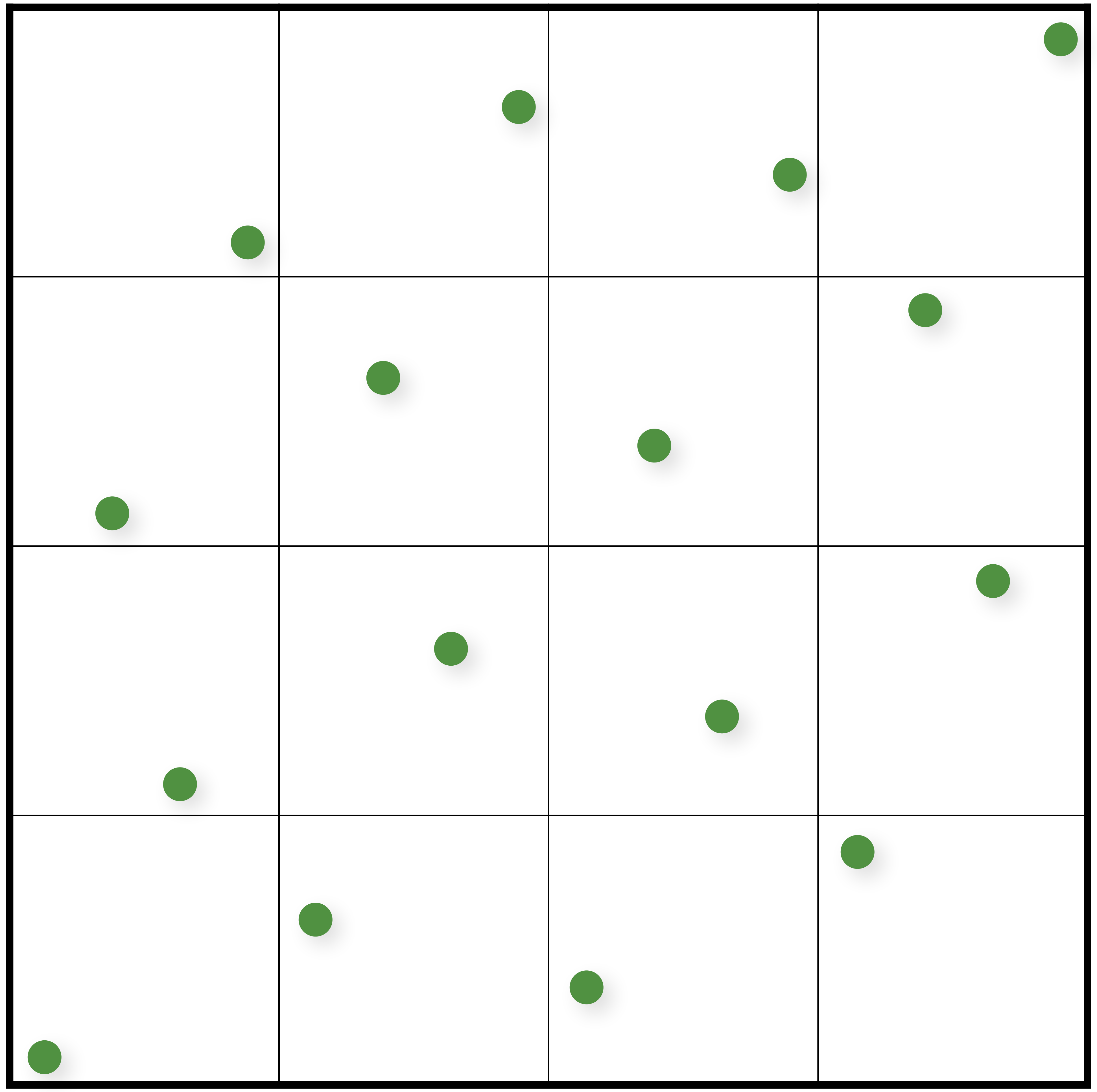
- Not incremental, need to know sample count, N, in advance

The Hammersley Sequence



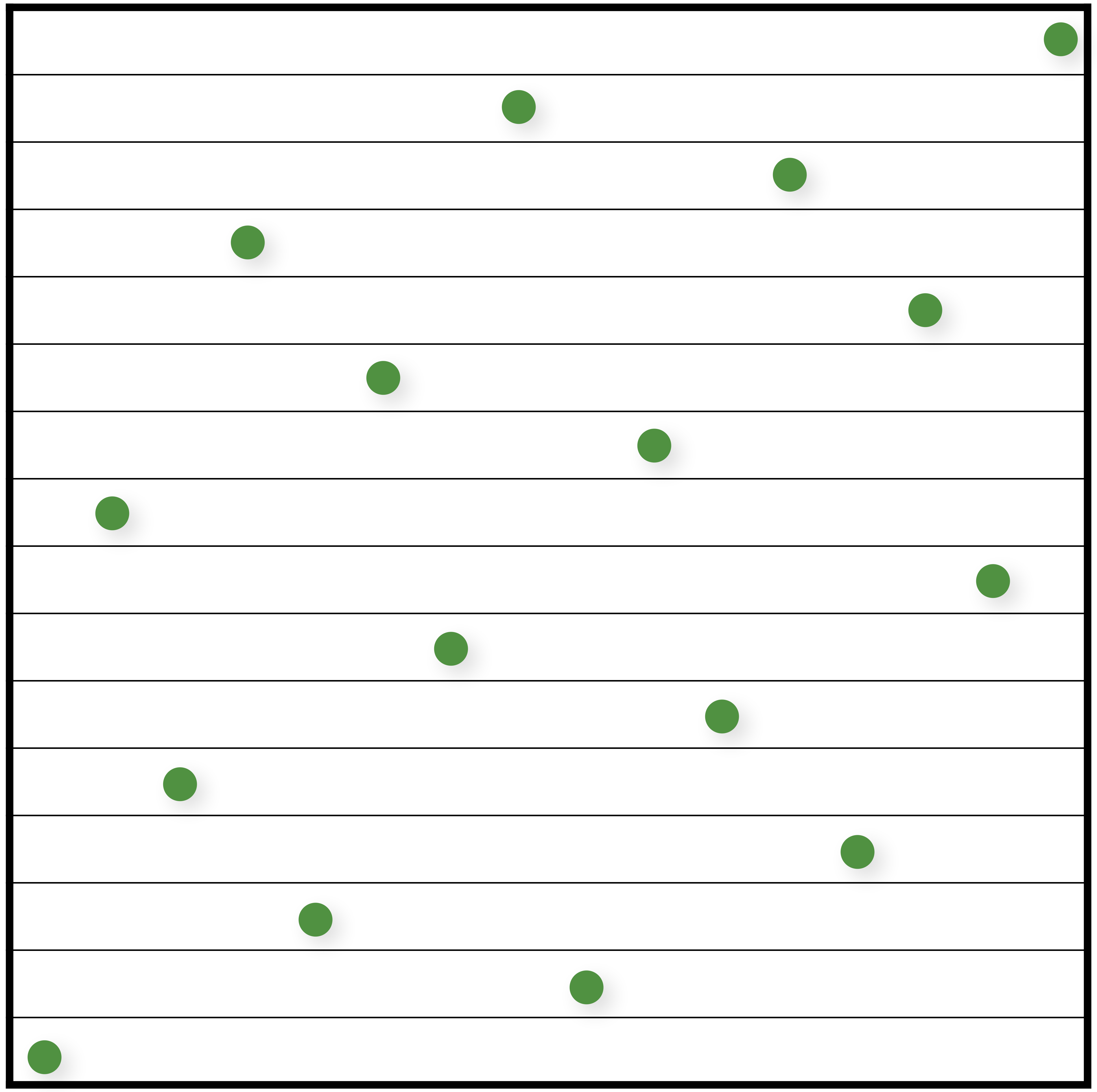
1 sample in each “elementary interval”

The Hammersley Sequence



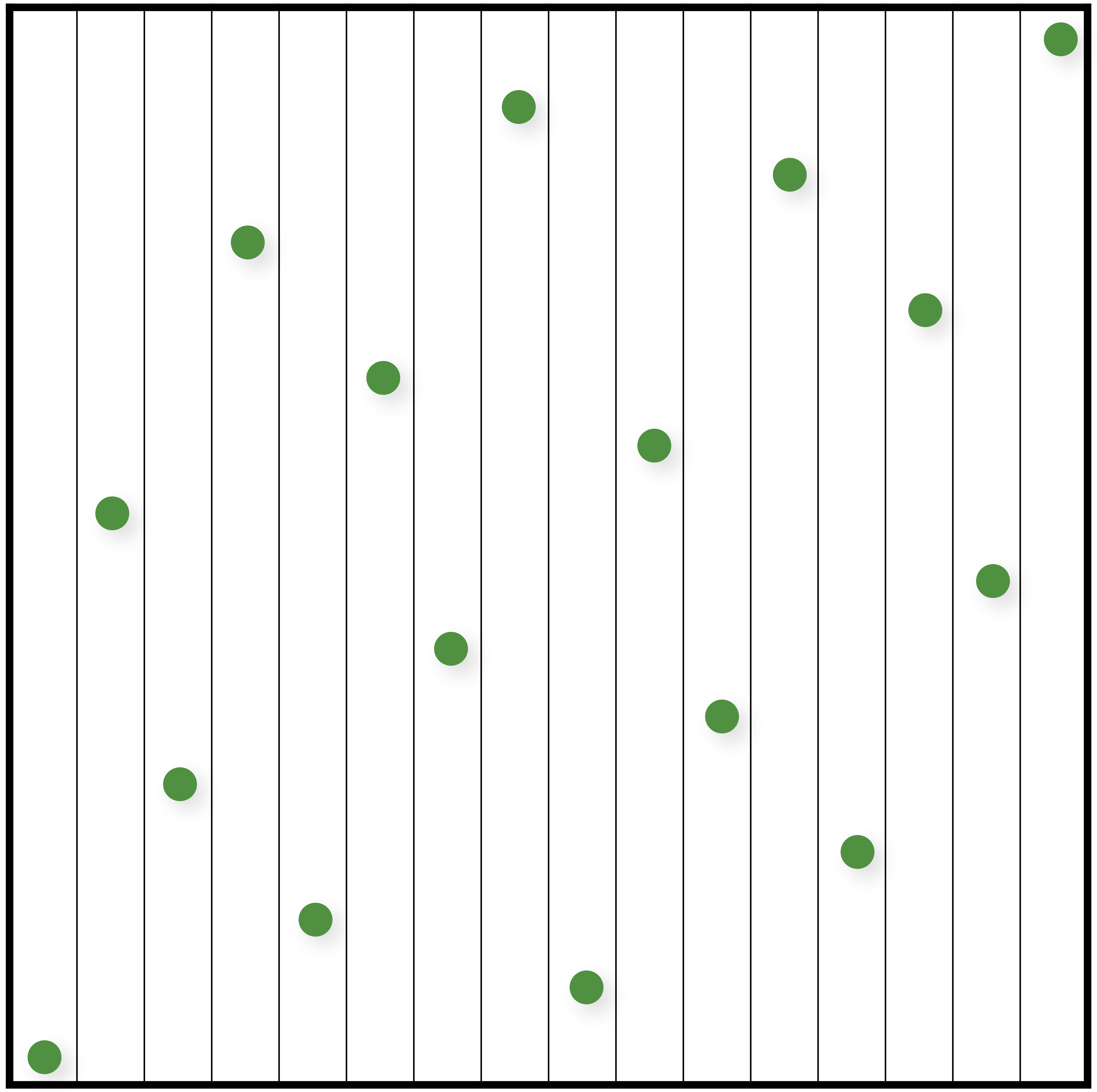
1 sample in each "elementary interval"

The Hammersley Sequence



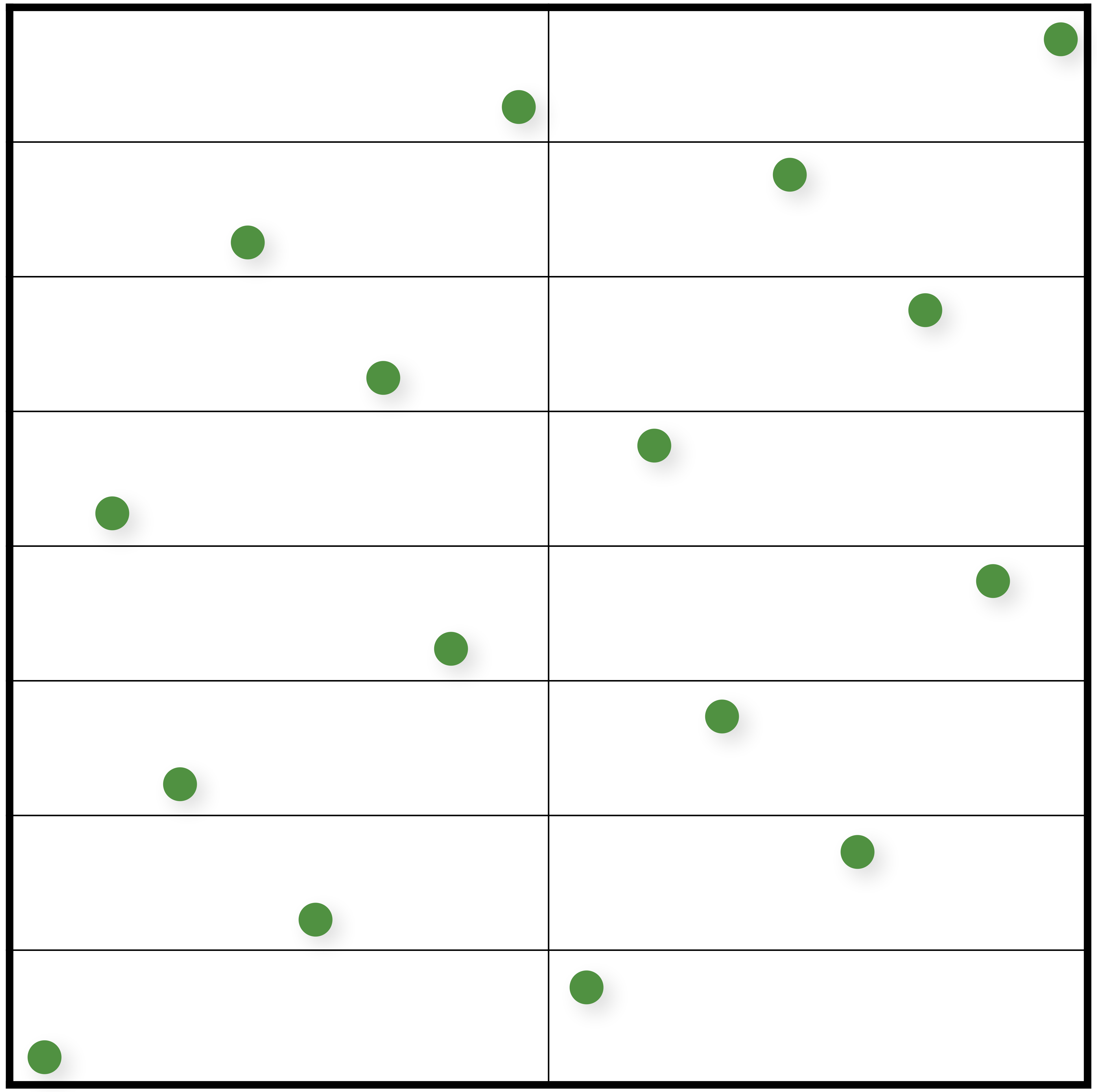
1 sample in each "elementary interval"

The Hammersley Sequence



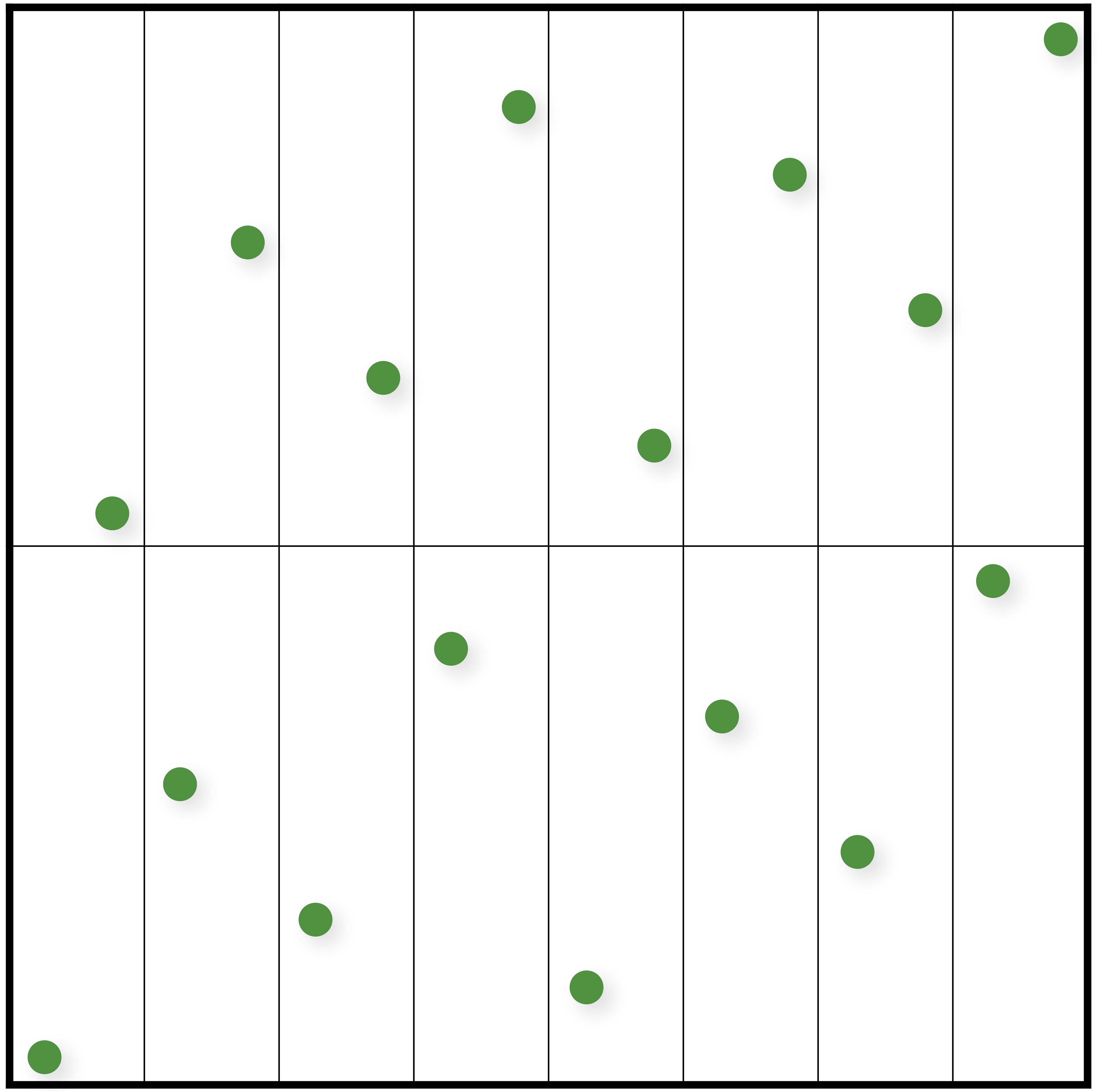
1 sample in each “elementary interval”

The Hammersley Sequence



1 sample in each "elementary interval"

The Hammersley Sequence



1 sample in each “elementary interval”

Monte Carlo (16 random samples)

Monte Carlo (16 jittered samples)

Scrambled Low-Discrepancy Sampling

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.