Integration

- General problem: integrate a given function over an n-dimensional domain $[0, 1]^n$
  - Remap dimensions with other ranges to $[0, 1]$

- Monte Carlo integration
  - Approximate this integral with a sum with a set of (pseudo)-random points $x_i$ drawn from a PDF $p(x_i)$:
    \[
    \int_{[0, 1]^n} f(x) \, dx \approx \frac{1}{N} \sum_{i=1}^{N} \frac{f(x_i)}{p(x_i)}
    \]

- Quasi-Monte Carlo integration
  - Replace pseudo-random points with low-discrepancy points
Remapping From Sample Distributions

\[ 0.2 \quad 0.4 \quad 0.6 \quad 0.8 \quad 1.0 \]

\[ 0.2 \quad 0.4 \quad 0.6 \quad 0.8 \quad 1.0 \]
Four 2D Point Sets

Which one will generally be most effective for integration? Why?
Low Discrepancy vs. Monkey Eye

Monkey Eye Cone Distribution

Low Discrepancy Point Set
Good Sample Patterns Really Matter

Section of Blurry Bigguy Image, 16 samples per pixel
Discrepancy

- One way to measure the quality of a set of sample points
- Given a point set, consider all boxes with one corner at the origin
- Define $n(x,y)$ as the number of points inside the box with upper corner $(x,y)$
- The discrepancy is the difference between the estimated area $n(x,y)/N$ and the actual area
Discrepancy

- The maximum discrepancy over all possible boxes gives a measure of how well distributed the points are.
  - We’re generally interested in the asymptotic behavior of the discrepancy for arbitrary numbers of samples $N$.

- A number of constructions have been developed to generate low-discrepancy point sets algorithmically...
The Radical Inverse

- Consider the digits of a number $n$, expressed in base $b$

$$n = \sum_{i=1}^{\infty} d_i b^{i-1}$$

- e.g. for $n = 6$ in base 2, $n=110_2$, and

$$d_1 = 0, d_2 = 1, d_3 = 1, d_i = 0$$

- The radical inverse mirrors the digits around the decimal:

$$\Phi_2(6) = 0.011_2 = 0 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdot 2^{-3} = 0.375$$

$$\Phi_b(n) = \sum_{i=1}^{\infty} d_i b^{-i}$$
### Low Discrepancy in 1D: van der Corput

<table>
<thead>
<tr>
<th>n</th>
<th>$\Phi_2(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0.5</td>
</tr>
<tr>
<td>2</td>
<td>0.25</td>
</tr>
<tr>
<td>3</td>
<td>0.75</td>
</tr>
<tr>
<td>4</td>
<td>0.125</td>
</tr>
<tr>
<td>5</td>
<td>0.625</td>
</tr>
<tr>
<td>6</td>
<td>0.375</td>
</tr>
<tr>
<td>7</td>
<td>0.875</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
## Low Discrepancy in 1D: van der Corput

<table>
<thead>
<tr>
<th>n</th>
<th>$\Phi_2(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0.5</td>
</tr>
<tr>
<td>2</td>
<td>0.25</td>
</tr>
<tr>
<td>3</td>
<td>0.75</td>
</tr>
<tr>
<td>4</td>
<td>0.125</td>
</tr>
<tr>
<td>5</td>
<td>0.625</td>
</tr>
<tr>
<td>6</td>
<td>0.375</td>
</tr>
<tr>
<td>7</td>
<td>0.875</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Low Discrepancy in 1D: van der Corput

<table>
<thead>
<tr>
<th>$n$</th>
<th>$\Phi_2(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0.5</td>
</tr>
<tr>
<td>2</td>
<td>0.25</td>
</tr>
<tr>
<td>3</td>
<td>0.75</td>
</tr>
<tr>
<td>4</td>
<td>0.125</td>
</tr>
<tr>
<td>5</td>
<td>0.625</td>
</tr>
<tr>
<td>6</td>
<td>0.375</td>
</tr>
<tr>
<td>7</td>
<td>0.875</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
### Low Discrepancy in 1D: van der Corput

<table>
<thead>
<tr>
<th>$n$</th>
<th>$\Phi_2(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0.5</td>
</tr>
<tr>
<td>2</td>
<td>0.25</td>
</tr>
<tr>
<td>3</td>
<td>0.75</td>
</tr>
<tr>
<td>4</td>
<td>0.125</td>
</tr>
<tr>
<td>5</td>
<td>0.625</td>
</tr>
<tr>
<td>6</td>
<td>0.375</td>
</tr>
<tr>
<td>7</td>
<td>0.875</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Diagram showing points on a line for $n = 0, 1, 2, 3, 4, 5, 6, 7$.
# Low Discrepancy in 1D: van der Corput

<table>
<thead>
<tr>
<th>n</th>
<th>$\Phi_2(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0.5</td>
</tr>
<tr>
<td>2</td>
<td>0.25</td>
</tr>
<tr>
<td>3</td>
<td>0.75</td>
</tr>
<tr>
<td>4</td>
<td>0.125</td>
</tr>
<tr>
<td>5</td>
<td>0.625</td>
</tr>
<tr>
<td>6</td>
<td>0.375</td>
</tr>
<tr>
<td>7</td>
<td>0.875</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Implementing a Base 2 Radical Inverse

\[ \Phi_2(n) = \sum_{i=1}^{32} d_i 2^{-i} \]

float RadicalInverse2(uint32_t v) {
    float ret = 0.f;
    for (int i = 1; i <= 32; ++i) {
        int digit = (v & (1 << (i+1))) != 0;
        ret += digit * powf(2.f, -i);
    }
    return ret;
}

Correct, but terrible.
What all is wrong with this code?
Efficient Base 2 Radical Inverse

- Recall the definition of the digits of a number in base $b$:

$$n = \sum_{i=1}^{\infty} d_i b^{(i-1)}$$

- Thanks to computer binary representation, these digits are easily extracted:

```c
int DigitBase2(uint32_t n, int i) {
  return (n & (1 << (i-1))) ? 1 : 0;
}
```

- In arbitrary dimensions, we’re not so lucky:

```c
int DigitBaseB(uint32_t n, int i, int b) {
  n /= ipow(b, i-1);
  return n % b;
}
```
Efficient Base 2 Radical Inverse

- Assume a fixed number of bits (say 32): \( \Phi_b(n) = \sum_{i=1}^{32} d_i b^{-i} \)

- We have the sum: \( d_1 2^{-1} + d_2 2^{-2} + \cdots + d_{32} 2^{-32} \)

- Pull out a factor of \( 2^{-32} \): \( 2^{-32}(d_1 2^{31} + d_2 2^{30} + \cdots + d_{32}) \)

- Can also express this in terms of bit shifts: \( 2^{-32}((d_1 \ll 31) + (d_2 \ll 30) + \cdots + d_{32}) \)
Efficient Base 2 Radical Inverse

\[ 2^{-32}((d_1 \ll 31) + (d_2 \ll 30) + \cdots + d_{32}) \]

- We have the digits already in the bits of \( n \)

\[
n = \sum_{i=1}^{\infty} d_i b^{(i-1)}
\]

- So
  - Reverse the bits
  - Multiply by \( 2^{-32} \)
Reversing The Bits in an Integer

```c
uint32_t ReverseBits(uint32_t 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;
}
```
Reversing The Bits in an Integer

```
uint32_t ReverseBits(uint32_t 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;
}
```
uint32_t ReverseBits(uint32_t 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 & 0xc3cccccc) >> 2);
    n = ((n & 0x55555555) << 1) | ((n & 0xaaaaaaaa) >> 1);
    return n;
}

Reversing The Bits in an Integer

32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1

16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17

16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17

8 7 6 5 4 3 2 1 16 15 14 13 12 11 10 9 24 23 22 21 20 19 18 17 32 31 30 29 28 27 26 25

8 7 6 5 4 3 2 1 16 15 14 13 12 11 10 9 24 23 22 21 20 19 18 17 32 31 30 29 28 27 26 25

8 7 6 5 4 3 2 1 16 15 14 13 12 11 10 9 24 23 22 21 20 19 18 17 32 31 30 29 28 27 26 25

8 7 6 5 4 3 2 1 16 15 14 13 12 11 10 9 24 23 22 21 20 19 18 17 32 31 30 29 28 27 26 25

8 7 6 5 4 3 2 1 16 15 14 13 12 11 10 9 24 23 22 21 20 19 18 17 32 31 30 29 28 27 26 25

8 7 6 5 4 3 2 1 16 15 14 13 12 11 10 9 24 23 22 21 20 19 18 17 32 31 30 29 28 27 26 25

8 7 6 5 4 3 2 1 16 15 14 13 12 11 10 9 24 23 22 21 20 19 18 17 32 31 30 29 28 27 26 25

8 7 6 5 4 3 2 1 16 15 14 13 12 11 10 9 24 23 22 21 20 19 18 17 32 31 30 29 28 27 26 25

8 7 6 5 4 3 2 1 16 15 14 13 12 11 10 9 24 23 22 21 20 19 18 17 32 31 30 29 28 27 26 25

8 7 6 5 4 3 2 1 16 15 14 13 12 11 10 9 24 23 22 21 20 19 18 17 32 31 30 29 28 27 26 25
uint32_t ReverseBits(uint32_t 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;
}
uint32_t ReverseBits(uint32_t 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;
}
Reversing The Bits in an Integer

```c
uint32_t ReverseBits(uint32_t 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 RadicalInverse2(uint32_t v) {
    v = ReverseBits(v);
    const float Inv2To32 = 1.f / (1ull << 32);
    return v * Inv2To32;
}
Halton Points

- Low-discrepancy point set for any number of dimensions
- Can generate points incrementally without knowing the total number needed
  - There are better choices for 1-3D and if the # needed is known

\[(\Phi_2(n), \Phi_3(n))\]

Defined by \((\Phi_{b_1}(n), \Phi_{b_2}(n), \Phi_{b_3}(n), \ldots)\), where the bases for each of the dimensions are relatively prime
Halton Points

- One caution: 2D projections of higher bases may not be great
- The overall pattern in remains low-discrepancy over all dimensions, though

\[(\Phi_{23}(n), \Phi_{29}(n))\]
Hammersley Points

- If the number of points, $N$, is known in advance, set one dimension to $n/N$

  \[(n/N, \Phi_{b_1}(n), \Phi_{b_2}(n), \ldots)\]

- Slightly lower discrepancy than Halton
Efficient Radical Inverses for bases $b \neq 2$

- Break the sum into two components:

$$\Phi_b(n) = \sum_{i=1}^{\infty} d_i b^{-i} = \sum_{i=T+1}^{\infty} d_i b^{-i} + \sum_{i=1}^{T} d_i b^{-i}$$

- If generating successive values of $\Phi_b(n)$, then this value changes only every $b^T$ times.

- This depends only on the $b^T$ low order bits of $n$; can tabularize.
Generator Matrices

- Given a base $b$ and a matrix $C$, define:

$$c(n) = (b^{-1}, b^{-2}, \ldots, b^{-m}) C \begin{pmatrix} d_1 \\ d_2 \\ \vdots \\ d_m \end{pmatrix}$$

- where $d_i$ are the base-$b$ digits of $n$
- and arithmetic is done over the ring $\mathbb{Z}_b$
- For our purposes, just do everything “mod $b$”

- This generates a set of $b^m$ points
- Appropriately-chosen $C$ matrices generate various low-discrepancy point sets
Generator Matrices

- We’ll focus only on $b=2$, which allows particularly efficient implementation

\[ c(n) = (2^{-1}, 2^{-2}, \ldots, 2^{-m}) C \begin{pmatrix} d_1 \\ d_2 \\ \vdots \\ d_m \end{pmatrix} \]
Generator Matrices

- How do we do this multiplication efficiently?
  - Consider e.g. $m=32$ for regular 32-bit integers...

\[
\begin{pmatrix}
  c_{1,1} & c_{1,2} & \cdots & c_{1,m} \\
  c_{2,1} & \ddots & & \vdots \\
  \vdots & & \ddots & \vdots \\
  c_{m,1} & c_{m,2} & \cdots & c_{m,m}
\end{pmatrix}
\begin{pmatrix}
  d_1 \\
  d_2 \\
  \vdots \\
  d_n
\end{pmatrix}
= d_1
\begin{pmatrix}
  c_{1,1} \\
  c_{2,1} \\
  \vdots \\
  c_{m,1}
\end{pmatrix}
+ d_2
\begin{pmatrix}
  c_{1,2} \\
  c_{2,2} \\
  \vdots \\
  c_{m,2}
\end{pmatrix}
+ \cdots d_m
\begin{pmatrix}
  c_{1,m} \\
  c_{2,m} \\
  \vdots \\
  c_{m,m}
\end{pmatrix}
\]
Generator Matrices

- Recall that we’re doing all of this arithmetic mod 2
  - All values are either 0 or 1...

\[
d_1 \left( \begin{array}{c} c_{1,1} \\ c_{2,1} \\ \vdots \\ c_{m,1} \end{array} \right) + d_2 \left( \begin{array}{c} c_{1,2} \\ c_{2,2} \\ \vdots \\ c_{m,2} \end{array} \right) + \cdots d_m \left( \begin{array}{c} c_{1,m} \\ c_{2,m} \\ \vdots \\ c_{m,m} \end{array} \right)
\]

- What logical operations are + and *, mod 2, equivalent to?

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>*</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Efficient Evaluation of Generator Matrices

- ADD (XOR) together matrix columns if corresponding digit is one

```c
// C[] holds columns of the generator matrix
float GenMatA(uint32_t n, const uint32_t C[32]) {
  uint32_t bits = 0;
  for (int i = 0; i < 32; ++i) {
    if (n & (1 << i) != 0)
      bits ^= C[i];  // 32 ADDs, mod 2
  }
  const float Inv2To32 = 1.f / (1ull << 32);
  return ReverseBits(bits) * Inv2To32;
}
```
Efficient Evaluation of Generator Matrices

- **ADD (XOR) together matrix columns if corresponding digit is one**

- **Better: stop once n is zero**

```c
// C[] holds columns of the generator matrix
float GenMatA(uint32_t n, const uint32_t C[32]) {
  uint32_t bits = 0;
  for (int i = 0; i < 32; ++i) {
    if (n & (1 << i) != 0)
      bits ^= C[i];  // 32 ADDs, mod 2
  }
  const float Inv2To32 = 1.f / (1ull << 32);
  return ReverseBits(bits) * Inv2To32;
}

float GenMatB(uint32_t n, const uint32_t C[32]) {
  uint32_t bits = 0, i = 0;
  while (n != 0) {
    if (n & 1)
      bits ^= C[i];
    n >>= 1;
    ++i;
  }
  const float Inv2To32 = 1.f / (1ull << 32);
  return ReverseBits(bits) * Inv2To32;
}
```
Efficient Evaluation of Generator Matrices

Better: avoid the bit reverse by processing columns in reverse order

```c
float GenMatB(uint32_t n, const uint32_t C[32]) {
    uint32_t bits = 0, i = 31;
    while (n != 0) {
        if (n & 1)
            bits ^= C[i];
        n >>= 1;
        --i;
    }
    const float Inv2To32 = 1.f / (1ull << 32);
    return bits * Inv2To32;
}
```
Even Faster Evaluation with Grey Codes

- Grey codes: permutation of integers within blocks of size $2^n$ such that adjacent values only differ in a single bit.

- Very simple to compute:

```c
int GreyCode(int v) {
    return v ^ (v >> 1);
}
```

- Basic approach: generate $c(n)$ in Grey code order, incrementally based on $c(n-1)$.

```
<table>
<thead>
<tr>
<th>n</th>
<th>binary</th>
<th>Grey code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>110</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
<td>111</td>
</tr>
<tr>
<td>6</td>
<td>110</td>
<td>101</td>
</tr>
<tr>
<td>7</td>
<td>111</td>
<td>100</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>1100</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
```
Incremental Evaluation with Grey Codes

- If \( g() \) is the Grey code function, and we have \( c(g(n-1)) \), what is \( c(g(n)) \)?

  - We know that \( g(n-1) \) and \( g(n) \) differ in a single bit (call it \( b \))

\[
c(g(n)) = c(g(n - 1)) + \begin{pmatrix} c_b,1 \\ c_b,2 \\ \vdots \\ c_b,m \end{pmatrix}
\]

Or

\[
c(g(n)) = c(g(n - 1)) - \begin{pmatrix} c_b,1 \\ c_b,2 \\ \vdots \\ c_b,m \end{pmatrix}
\]
Good News In Base 2

- Both + and - are equivalent to XOR in $\mathbb{Z}_2$

<table>
<thead>
<tr>
<th>+ 0 1</th>
<th>- 0 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1</td>
<td>0 0 1</td>
</tr>
<tr>
<td>1 1 0</td>
<td>1 1 0</td>
</tr>
</tbody>
</table>

$c(g(n)) = c(g(n - 1)) \pm \begin{pmatrix} c_{b,1} \\ c_{b,2} \\ \vdots \\ c_{b,m} \end{pmatrix}$

next = prev ^ C[changedBit]
Even Faster Evaluation with Grey Codes

<table>
<thead>
<tr>
<th>n</th>
<th>binary</th>
<th>Grey code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>110</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
<td>111</td>
</tr>
<tr>
<td>6</td>
<td>110</td>
<td>101</td>
</tr>
<tr>
<td>7</td>
<td>111</td>
<td>100</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>1100</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Which Grey Code

<table>
<thead>
<tr>
<th>Bit # Changed?</th>
</tr>
</thead>
<tbody>
<tr>
<td>n/a</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

The number of trailing 0s in the binary representation of n

Now if we can quickly find the number of trailing zeros in the
Even Faster Evaluation with Grey Codes

- Final implementation is super efficient
- Most ISAs have an instruction to count trailing zeros
- Compiler intrinsics:
  - `__builtin_ctz()` in gcc/clang
  - `_BitScanForward` in MSVC

```c
uint32_t CIncremental(uint32_t n, uint32_t prev, const uint32_t C[32]) {
    int changedBit = CountTrailingZeros(n);
    return prev ^ C[31-changedBit];
}
```

```assembly
bsfl %edi, %eax
xorl $31, %eax
xorl (%rdx,%rax,4), %esi
```
Sobol’ Point Sets

- Sobol’ first showed how to find generator matrices for low-discrepancy point sets in base 2
  - Can scale low-discrepancy samples in 1000s of dimensions
32 2D Sobol’ Points
Stratification Over Elementary Intervals (1x64)
Stratification Over Elementary Intervals (2x32)
Stratification Over Elementary Intervals (4x8)
Stratification Over Elementary Intervals (8x4)
Stratification Over Elementary Intervals (16x2)
Stratification Over Elementary Intervals (32x1)
(0,2)-sequences

- In addition to satisfying general stratification properties, power-of-two length subsequences are well-distributed with respect to each other
  - Helpful for adaptive sampling, for example
(0,2)-sequences

- In addition to satisfying general stratification properties, power-of-two length subsequences are well-distributed with respect to each other
  - Helpful for adaptive sampling, for example
(0,2)-sequences

- In addition to satisfying general stratification properties, power-of-two length subsequences are well-distributed with respect to each other.
  - Helpful for adaptive sampling, for example
(0,2)-sequences

- In addition to satisfying general stratification properties, power-of-two length subsequences are well-distributed with respect to each other
- Helpful for adaptive sampling, for example
Maximized Minimum Distance

- Some problems with low discrepancy as a quality measure
  - Anisotropic: rotating the points changes discrepancy
  - Not shift-invariant: similarly for translating them
- In general, can have low discrepancy yet still have points clumped together
- Can we find low-discrepancy point sets that also maximize minimum distance between points?
Maximized Minimum Distance

- Grünschloß and Keller: exhaustive search over generator matrices

New patterns are still stratified over elementary intervals
Same LD Samples Used For Lighting At All Pixels
Randomized LD Samples Used For Lighting At All Pixels
Cranley-Patterson Rotations

- A way to generate independent random instances of a point set
- Choose a random offset \((dx, dy) \in [0,1]^2\)
- Then, offset each point by \((dx, dy)\), wrapping around if the result is > 1

\[ (0.1, 0.075) \]
Cranley-Patterson Rotations

A way to generate independent random instances of a point set
- Choose a random offset \((dx, dy) \in [0, 1]^2\)
- Then, offset each point by \((dx, dy)\), wrapping around if the result is \(> 1\)

\(\vec{r} = (0.1, 0.075)\)
Cranley-Patterson Rotations

- A way to generate independent random instances of a point set
  - Choose a random offset \((dx, dy) \in [0,1]^2\)
  - Then, offset each point by \((dx, dy)\), wrapping around if the result is > 1

\[
\leftarrow = (0.1, 0.075)
\]
Rendering with Low-Discrepancy Point Sets

- What is the integration domain?
  - One pixel at a time?
    - Generate independent point set within each pixel area
    - Randomize with e.g. Cranley-Patterson
    - Simple to implement, parallelizes well

May have poor sample coverage near pixel edges

\[
\int f(x, y, t, u, v) \, dx \, dy \, dt \, du \, dv
\]
Rendering with Low-Discrepancy Point Sets

- Use the entire image as the domain?
  - Generate a single point set with (# pixels * # pixel samples) points
  - Parallelization is trickier: which threads use which points?

- Middle ground: break image into tiles of pixels, independently generate point sets within tiles
### Padding For Higher Dimensions

- General stratification properties \((t,s)\)-sequences are fantastic
  - But there are no \((t,s)\)-sequences for \(\geq 3\) dimensions

- Can just use e.g. Sobol’ point sets when have more dimensions
  - These points are still low discrepancy at least!

- Or, can use padding. e.g. for 4D:
  - Generate two sets of high-quality 2D points
  - Each 4D sample is formed by random assignment to 2 2D samples:

\[
(p_0, p_1, p_2, p_3)
\]
Various Sampling Options for MB+Defocus

- Random
  RMS 1.85%

- Padded (0,2)-nets
  RMS 1.44%

- Halton
  RMS 1.74%

- Sobol'
  RMS 1.38%
Resources

- http://gruenschloss.org/
- Sobol’ Sequence Generator
- Hacker’s Delight, Henry Warren