Progressive Multi-Jittered Sample Sequences: Supplemental Materials

Per Christensen      Andrew Kensler      Charlie Kilpatrick
Pixar Animation Studios

Abstract

1. Pseudo-code
In this section we list pseudocode for generating progressive jittered, progressive multi-jittered, and progressive multi-jittered (0,2) sample points on the unit square.

1.1. Progressive jittered
Pseudocode for generating a sequence of M progressive jittered sample points. Main function:

```
procedure GENERATE_PMJ(M)
    // Generate first sample point at random position
    samples[0] ← (rdn(), rdn())
    N ← 1
    while N < M do
        // Generate next 3N sample points
        extendSequence(N)
        N ← 4*N
    end while
end procedure
```

Generate next 3N sample points (N ... 4N):

```
procedure EXTEND_SEQUENCE(N)
    n ← √N
    // Loop over N old samples and generate 3 new samples for each old sample
    for s ← 0 . . . N-1 do
        // Determine sub-quadrant of existing sample point
        oldpt ← samples[s]
        i ← ⌊n * oldpt.x⌋
        j ← ⌊n * oldpt.y⌋
        xhalf ← 2.0 * (n * oldpt.x - i)
        yhalf ← 2.0 * (n * oldpt.y - j)
        // First select the diagonally opposite sub-quadrant
        xhalf ← 1-xhalf
        yhalf ← 1-yhalf
        samples[3*N+s] ← generateSamplePoint(i, j, xhalf, yhalf, n)
        // Then randomly select one of the two remaining sub-quadrants
        if rdn() > 0.5 then
            xhalf ← 1-xhalf
        else
            yhalf ← 1-yhalf
        end if
        samples[2*N+s] ← generateSamplePoint(i, j, xhalf, yhalf, n)
        // And finally select the last sub-quadrant
        xhalf ← 1-xhalf
        yhalf ← 1-yhalf
        samples[1*N+s] ← generateSamplePoint(i, j, xhalf, yhalf, n)
    end for
end procedure
```

Generate a sample point:

```
function GENERATE_SAMPLE_POINT(i, j, xhalf, yhalf, n)
    pt.x ← (i + 0.5 * (xhalf + rdn())) / n
    pt.y ← (j + 0.5 * (yhalf + rdn())) / n
    return pt
end function
```

In this pseudocode, M, N, n, i, j, and s are (unsigned) integers, xhalf and yhalf are 0 or 1, and all other variables are (double-precision) floating-point numbers. rdn() is a function that returns pseudo-random numbers between 0 and 1 (for example drand48() or a table lookup), and ⌊x⌋ is the floor function (simple float-to-int conversion in C-like languages).

1.2. Progressive multi-jittered (with blue noise)
Pseudocode for generating a sequence of M progressive multi-jittered sample points (with blue noise). We have to generate points in consecutive order for best candidates, so split the extendSequence() function into two. Main function:

```
procedure GENERATE_PMJ(M)
    // Generate first sample point at random position
    samples[0] ← (rdn(), rdn())
    N ← 1
    while N < M do
        // Generate next 3N sample points
        extendSequence(N)
        N ← 4*N
    end while
end procedure
```
Generate next \( N \) sample points (for \( N \) being an even power of two):

**procedure**

\[
\text{EXTEND} \quad n \leftarrow \sqrt[N]{4} N
\]

// Mark already occupied 1D strata so we can avoid them
markOccupiedStrata\((N)\)
// Loop over \( N \) old samples and generate 1 new sample for each
for \( s \leftarrow 0 \ldots N-1 \) do
   oldpt \( \leftarrow \) samples\([s]\)
   i \( \leftarrow \lfloor n \ast \text{oldpt.x} \rfloor \)
   j \( \leftarrow \lfloor n \ast \text{oldpt.y} \rfloor \)
   xhalf \( \leftarrow \lfloor 2.0 \ast (n \ast \text{oldpt.x} - i) \rfloor \)
   yhalf \( \leftarrow \lfloor 2.0 \ast (n \ast \text{oldpt.y} - j) \rfloor \)
   // Select the diagonally opposite subquadrant
   xhalf \( \leftarrow 1-xhalf \)
   yhalf \( \leftarrow 1-yhalf \)
   // Generate a sample point
   generateSamplePoint\((i, j, xhalf, yhalf, n, N)\)
end for

Generate next \( N \) sample points (for \( N \) being an odd power of two):

**procedure**

\[
\text{EXTENDSEQUENCEODD}(N) \quad n \leftarrow \sqrt[N]{N/2}
\]

// Mark already occupied 1D strata so we can avoid them
markOccupiedStrata\((N)\)
// Loop over \( N/2 \) old samples and generate 2 new samples for each
for \( s \leftarrow 0 \ldots N/2 - 1 \) do
   oldpt \( \leftarrow \) samples\([s]\)
   i \( \leftarrow \lfloor n \ast \text{oldpt.x} \rfloor \)
   j \( \leftarrow \lfloor n \ast \text{oldpt.y} \rfloor \)
   xhalf \( \leftarrow \lfloor 2.0 \ast (n \ast \text{oldpt.x} - i) \rfloor \)
   yhalf \( \leftarrow \lfloor 2.0 \ast (n \ast \text{oldpt.y} - j) \rfloor \)
   // Randomly select one of the two remaining subquadrants
   if \( \text{rnd()} > 0.5 \) then
      xhalf \( \leftarrow 1-xhalf \)
   else
      yhalf \( \leftarrow 1-yhalf \)
   end if
   // Generate a sample point
   generateSamplePoint\((i, j, xhalf, yhalf, n, N)\)
end for

Generate a sample point by choosing best of 10 candidate points:

**procedure**

\[
\text{GENERATESAMPLEPOINT}(i, j, xhalf, yhalf, n, N) \quad NN \leftarrow 2 * N
\]

bestDist \( \leftarrow 0.0 \)
numCand \( \leftarrow 10 \) // number of candidate points
// Generate candidate points and pick the best
for \( t \leftarrow 1 \ldots \text{numCand} \) do
   // Generate candidate sample \( x \) coord
   candpt.x \( \leftarrow (i + 0.5 \ast (\text{xhalf} + \text{rnd}())) / n \)
xstratum \( \leftarrow [NN \ast \text{candpt.x}]\)
   // Generate candidate sample \( y \) coord
   candpt.y \( \leftarrow (j + 0.5 \ast (\text{yhalf} + \text{rnd}())) / n \)
ystratum \( \leftarrow [NN \ast \text{candpt.y}]\)
   // Evaluate distance between candidate point and existing samples
   d \( \leftarrow \text{minDist}() \)
   // Keep candidate point if it has higher dist than best so far
   if \( d > \text{bestDist} \) then
      bestDist \( \leftarrow d \)
      pt \( \leftarrow \text{candpt} \)
   end if
end for

Mark all occupied 1D strata:

**procedure**

\[
\text{MARKOCCUPIEDSTRATA}(N) \quad NN \leftarrow 2 * N
\]
occupied1Dx\([0 \ldots NN-1]\) \( \leftarrow \) false // init array
occupied1Dy\([0 \ldots NN-1]\) \( \leftarrow \) false // init array
for \( s \leftarrow 0 \ldots N-1 \) do
   xstratum \( \leftarrow [NN \ast \text{samples}[s].x]\)
ystratum \( \leftarrow [NN \ast \text{samples}[s].y]\)
occupied1Dx\([\text{xstratum}]\) \( \leftarrow \) true
occupied1Dy\([\text{ystratum}]\) \( \leftarrow \) true
end for

Generate next \( 3N \) sample points:

**procedure**

\[
\text{GENERATESEQUENCE}(2 * N, \text{samples}, \text{numSamples})
\]
extendSequenceOdd\((2 * N, \text{samples}, \text{numSamples})\)
// Loop over \( 2N \) old samples and generate 2 new samples for each
for \( s \leftarrow 0 \ldots 2N-1 \) do
   // Assign new sample point
   occupied1Dx\([0 \ldots 2N-1]\) \( \leftarrow \) \(\text{false}\) // init array
   occupied1Dy\([0 \ldots 2N-1]\) \( \leftarrow \) \(\text{false}\)
   // Mark 1D strata as occupied
   occupied1Dx\([0 \ldots 2N-1]\) \( \leftarrow \) \(\text{true}\)
end for

Generate next \( 4N \) sample points:

**procedure**

\[
\text{GENERATESEQUENCE}(4 * N, \text{samples}, \text{numSamples})
\]
extendSequenceEven\((4 * N, \text{samples}, \text{numSamples})\)
// Loop over \( N \) old samples and generate 1 new sample for each
for \( s \leftarrow 0 \ldots N-1 \) do
   // Assign new sample point
   occupied1Dx\([0 \ldots N-1]\) \( \leftarrow \) \(\text{false}\)
   occupied1Dy\([0 \ldots N-1]\) \( \leftarrow \) \(\text{false}\)
   // Mark 1D strata as occupied
   occupied1Dx\([0 \ldots N-1]\) \( \leftarrow \) \(\text{true}\)
end for

The optional procedure to classify already occupied coarse sub-quads as even or odd diagonals, i.e. \( '/' \) or \( ' \backslash '\):

Here, \text{occupied1Dx}[\] and \text{occupied1Dy}[\] are arrays of Booleans, while \text{xhalves}[\] and \text{yhalves}[\] are arrays of 0 or 1 values. The function \text{minDist}() computes the distance to the nearest existing point – please refer to \cite{MF92,DH06} for efficient progressive algorithms to calculate this. Setting \text{numCand} to 1 and skipping the call to \text{minDist}() will generate pmj samples without blue noise.

The optional procedure to classify already occupied coarse sub-quads as even or odd diagonals, i.e. \( '/' \) or \( ' \backslash '\):
Here evens and evens are Booleans, and evenx\[\] and eveny\[\] is a two-dimensional array of Booleans.

As mentioned in the main paper, the sample distribution between powers of two can be improved by selecting one of the two remaining subquadrants in a more balanced manner; the balanced choices can be done in ox-plowing, a.k.a. boustrophedonic, order. Here we use a mix of pseudocode and C notation for compactness:

```plaintext
function SELECTSUBQUADRANTS(n)
    choiceBalanceX[0 . . . n-1] ← 0 // array init
    choiceBalanceY[0 . . . n-1] ← 0 // array init
    up ← false
    // Visit quadrants in up/down "ox-plowing" (boustrophedonic) order
    for i ← 0 . . . n-1 do
        up ← not up
        for jj ← 0 . . . n-1 do
            j ← up ? jj : n-jj-1
            last ← (j = n-abs(choiceBalanceX[i])) and n > 1
            evendiag ← evenx[jj]
            // If last entry in a column: enforce x balance
            if choiceBalanceY[jj] != 0 and not last then
                neg ← choiceBalanceY[jj] < 0 // more y lows than highs
                // Do opposite y choice than previous column
                subquadchoicesX[i][j] ← neg ? 1 : 0
                subquadchoicesX[i][j] ← evendiag xor neg ? 1 : 0
                choiceBalanceX[i] ← evendiag xor neg ? 1 : -1
            else if choiceBalanceX[i] = 0 then
                neg ← choiceBalanceX[i] < 0 // more x lows than highs
                // Do opposite y choice than previous column
                subquadchoicesY[i][j] ← neg ? 1 : -1
                choiceBalanceY[i] ← evendiag xor neg ? 1 : -1
            else // even balance in both x and y
                // Randomly select one of the two subquadrants
                xhalf ← (rnd() > 0.5)
                yhalf ← evendiag ? 1 - xhalf : xhalf
                subquadchoicesX[i][j] ← xhalf
                subquadchoicesY[i][j] ← yhalf
                choiceBalanceX[i] ← xhalf ? 1 : -1
                choiceBalanceY[i] ← yhalf ? 1 : -1
        end if
    end for
end procedure
```

The results are two-dimensional integer arrays subquadchoicesX[] and subquadchoicesY[]. The function returns true if it succeeded in finding fully balanced choices. (If not, one can try again.)

1.3. Progressive multi-jittered (0,2)

Pseudocode for generating a sequence of progressive multi-jittered (0,2) sample points. Mostly the same as for pmj, but the markOccupiedStrata() procedure now works on strata of all rectangular shapes (elementary intervals):

```plaintext
procedure markOccupiedStrata(N)
    NN ← 2^n
    // Init occupiedStrata 2D array
    occupiedStrata[0 . . . log2(NN)][0 . . . NN-1] ← false
    // Loop over samples and mark occupied strata
    for s ← 0 . . . N-1 do
        markOccupiedStrata1(samples[s], NN)
    end for
end procedure
```

Procedure that marks all strata that point pt is in as occupied:

```plaintext
procedure markOccupiedStrata1(pt, NN)
    shape ← 0; xdivs ← NN; ydivs ← 1
    // Loop over strata shapes and mark occupied strata
    repeat
        xstratum ← [xdivs * pt.x]
        ystratum ← [ydivs * pt.y]
        occupiedStrata[shape][ystratum*xdivs+xstratum] ← true
    until xdivs = 0
end procedure
```

The generateSamplePoint() procedure (without best candidates) now rejects samples that are not stratified in all elementary intervals. This results in a (0,2) sequence:

```plaintext
procedure generateSamplePoint1(g, j, xhalf, yhalf, n, N)
    NN ← 2^n
    // Generate x and y until sample is accepted as a (0,2) sample
    repeat
        pt.x ← (i + 0.5 * (xhalf + rnd())) / n
        pt.y ← (j + 0.5 * (yhalf + rnd())) / n
    until not isOccupied(pt, NN)
    // Mark strata that this new sample occupy
    markOccupiedStrata1(pt, NN)
    // Assign new sample point
    samples[numSamples] ← pt
    numSamples ← numSamples+1
end procedure
```

Function to check strata of all shapes (elementary intervals) to see if point pt is in an occupied stratum:

```plaintext
function isOccupied(pt, NN)
    shape ← 0; xdivs ← NN; ydivs ← 1
    // Loop over strata shapes and check if stratum is occupied
    repeat
        xstratum ← [xdivs * pt.x]
        ystratum ← [ydivs * pt.y]
        if occupiedStrata[shape][ystratum*xdivs+xstratum] then
            return true // stratum is already occupied
        end if
    end repeat
end function
```

© 2018 The Author(s)
Computer Graphics Forum © 2018 The Eurographics Association and John Wiley & Sons Ltd.
2. Zone plate images with maximum error metric

Rms error is the typical measure for image comparisons, but maximum error might be more meaningful in settings where we’re trying to eliminate fireflies. Here we repeat the zone plate tests in section 8.1 of the main paper using maximum error.

Binary zone plate. Figure 1 (top) shows that the maximum error for rotated and xor-scrambled Sobol’ sequences is nearly as horrible and erratic as for sampling the triangle function. Curiously, Ahmed’s ART sequence has slightly lower maximum error than the other sequences in this test – here the combination of blue noise and stratification seems to pay off. Pmj02 and Owen-scrambled Sobol’ are on a tied second place.

Smooth zone plate. Figure 1 (bottom) shows very similar results to rms error, although the maximum error for pmj02 and Owen-scrambled Sobol’ converges a bit slower than rms error: roughly \(O(N^{-0.85})\) at powers of two.

![Binary zoneplate: pixel sampling max error](image)

![Smooth zoneplate: pixel sampling max error](image)

Figure 1: Maximum error for pixel sampling of binary and smooth zone plate images. (25–2500 samples per pixel.)

3. Pixel sampling with Gaussian filter

The tests in Section 8 of the main paper used a \(1 \times 1\) box pixel filter. Here we repeat the test of textured teapots (Figure 18 in the main paper) using a pixel filter more typical of production rendering: a (truncated) Gaussian filter covering \(2 \times 2\) pixels. We use pixel filter importance sampling [ESG06, Pur86], i.e., we map the pixel sample positions from the unit square with a cdf determined by the Gaussian function. Without these additional tests, it is not a priori obvious whether this mapping will warp the sample domain in such a way that the stratifications in the original square domain will be less efficient.

Figure 2 (top) shows error plots for pixels with discontinuities due to object edges. Here all sample sequences have the same convergence rate as for the box pixel filter (see Figure 19 in the main paper): random and best candidates converge as \(O(N^{-0.5})\) and all other sequences converge as roughly \(O(N^{-0.75})\).

Figure 2 (bottom) shows that the convergence rates in smooth pixels are a bit reduced: roughly \(O(N^{-0.9})\) for most sequences and \(O(N^{-1.3})\) for pmj02 and Owen-scrambled Sobol’. Nevertheless, these results show that \((0,2)\) stratification is still much better than lesser stratifications, even when the samples are warped.

![Textured teapots: pixel sampling rms error](image)

![Textured groundplane: pixel sampling rms error](image)

Figure 2: rms error for pixel sampling with Gaussian pixel filter. Top: teapot edges. Bottom: smooth ground plane.

4. Rectangular area light source

Now we repeat the area light experiments in Section 9 of the main paper with a rectangular light source rather than the square light. The size of the rectangle is \(4 \times 0.25\) and it illuminates the same two teapots on a ground plane. The long skinny rectangular light source emphasizes 1D distribution, so we expect the difference between pj and pmj (and pmj02) to be larger than for the square light source.

© 2018 The Author(s)

Computer Graphics Forum © 2018 The Eurographics Association and John Wiley & Sons Ltd.
Figure 3 (top) shows error plots for pixels in the penumbra region. Here all sample sequences have the same convergence rate as for the square light source: random and best candidates converge as $O(N^{-0.5})$ and all other sequences converge as $O(N^{-0.1})$. The error for Ahmed’s ART sequence and $pj$ is significantly higher than for the sequences that have 1D stratification. (This is similar to the test of the step function.)

Figure 3 (bottom) shows error plots for the smooth, fully illuminated image region. Random and best candidates converge as $O(N^{-0.3})$, most other sample sequences converge as $O(N^{-1.1})$, and $pmj02$ and Owen-scrambled Sobol’ converge as roughly $O(N^{-1.5})$. The Ahmed ART sequence is again suffering due to poor 1D stratification.

We have performed similar tests for a disk area light source in a separate technical report [Chr18]. With an appropriate sampling strategy, we can obtain convergence rates that are almost as good.

5. Comparing sets with sequences

We found it interesting to compare sample sets and sequences (even though sets are unsuited for incremental rendering and adaptive sampling). Table 1 shows error for sampling a 2D Gaussian function with 256, 1024, and 4096 samples using various progressive sample sequences and non-progressive sample sets. Average of 10000 trials each.

For a different comparison of sample sets and sequences, we tested pixel sampling of the checkerboard texture in Figure 16 of the main paper. Figure 4 shows the convergence of multi-jittered sets with 100, 400, and 1600 samples compared to a few progressive sequences. The sets have slow initial convergence due to their lack of progressive properties.

6. On discrepancy

Shirley introduced discrepancy for evaluating sample sets to computer graphics in a very influential paper [Shi91]. However, many authors – Shirley included – have found discrepancy to be a misleading measure of the quality of sample sets [Mit92, Gla95, PH10, Mit92, Gla95, PH10].

Table 1: Error for sampling of a 2D Gaussian function with 256, 1024, and 4096 samples using various progressive sample sequences and non-progressive sample sets. Average of 10000 trials each.

<table>
<thead>
<tr>
<th>sequence/set</th>
<th>256</th>
<th>1024</th>
<th>4096</th>
</tr>
</thead>
<tbody>
<tr>
<td>random</td>
<td>0.010774</td>
<td>0.005388</td>
<td>0.001731</td>
</tr>
<tr>
<td>best cand</td>
<td>0.003542</td>
<td>0.001676</td>
<td>0.000819</td>
</tr>
<tr>
<td>Perrier rot</td>
<td>0.000870</td>
<td>0.000666</td>
<td>0.000863</td>
</tr>
<tr>
<td>Ahmed</td>
<td>0.001100</td>
<td>0.000227</td>
<td>0.000711</td>
</tr>
<tr>
<td>Halton rot</td>
<td>0.001147</td>
<td>0.000337</td>
<td>0.000996</td>
</tr>
<tr>
<td>Halton scr</td>
<td>0.000953</td>
<td>0.000254</td>
<td>0.000866</td>
</tr>
<tr>
<td>Sobol rot</td>
<td>0.000971</td>
<td>0.000238</td>
<td>0.000865</td>
</tr>
<tr>
<td>Sobol xor</td>
<td>0.000621</td>
<td>0.000154</td>
<td>0.000338</td>
</tr>
<tr>
<td>Sobol owen</td>
<td>0.000064</td>
<td>0.000008</td>
<td>0.000001</td>
</tr>
<tr>
<td>$pj$</td>
<td>0.000670</td>
<td>0.000166</td>
<td>0.000424</td>
</tr>
<tr>
<td>pmj</td>
<td>0.000191</td>
<td>0.000046</td>
<td>0.000011</td>
</tr>
<tr>
<td>$pmj02$</td>
<td>0.000064</td>
<td>0.000009</td>
<td>0.000001</td>
</tr>
<tr>
<td>uniform jittered</td>
<td>0.009851</td>
<td>0.004959</td>
<td>0.002458</td>
</tr>
<tr>
<td>jittered</td>
<td>0.000663</td>
<td>0.000167</td>
<td>0.000424</td>
</tr>
<tr>
<td>multi-jittered</td>
<td>0.000184</td>
<td>0.000044</td>
<td>0.000111</td>
</tr>
<tr>
<td>cmj</td>
<td>0.000670</td>
<td>0.000231</td>
<td>0.000880</td>
</tr>
<tr>
<td>Hammersley rot</td>
<td>0.000768</td>
<td>0.000199</td>
<td>0.000551</td>
</tr>
<tr>
<td>Hammersley scr</td>
<td>0.000712</td>
<td>0.000183</td>
<td>0.000408</td>
</tr>
<tr>
<td>Larcher-Pil rot</td>
<td>0.000771</td>
<td>0.000199</td>
<td>0.000552</td>
</tr>
<tr>
<td>Larcher-Pil scr</td>
<td>0.000622</td>
<td>0.000158</td>
<td>0.000339</td>
</tr>
<tr>
<td>Grünschloß rot</td>
<td>0.000763</td>
<td>0.000200</td>
<td>0.000552</td>
</tr>
<tr>
<td>Rank-1 lattice rot</td>
<td>0.000747</td>
<td>0.000195</td>
<td>0.000494</td>
</tr>
</tbody>
</table>
In this section we investigate whether discrepancy can be used to evaluate and compare sample sequences in a meaningful way. (The term “low-discrepancy sequences” implies that the sequence with lowest discrepancy is best. It is not.)

### 6.1. Star discrepancy

Zaremba’s star discrepancy $D^*$ [Zar68] is the maximum difference between the area of an axis-aligned anchored rectangle and the fraction of samples (on the unit square) that fall within it. Star discrepancy is a common discrepancy measure for low-discrepancy patterns [Nie92], and is often used to compare the quality of sample sets – mostly due to its ease of computation and because theoretical convergence bounds can be derived using this measure (see, e.g., Grünschloß and Keller [GK08]).

#### Table 2: Star discrepancy $D^*$ for 64, 256, and 1024 samples from various progressive sample sequences.

<table>
<thead>
<tr>
<th>sequence</th>
<th>64</th>
<th>256</th>
<th>1024</th>
</tr>
</thead>
<tbody>
<tr>
<td>random</td>
<td>0.1501</td>
<td>0.0780</td>
<td>0.0374</td>
</tr>
<tr>
<td>best cand</td>
<td>0.0692</td>
<td>0.0317</td>
<td>0.0148</td>
</tr>
<tr>
<td>Perrier *</td>
<td>0.0614</td>
<td>0.0139</td>
<td>0.0086</td>
</tr>
<tr>
<td>Ahmed</td>
<td>0.0741</td>
<td>0.0276</td>
<td>0.0123</td>
</tr>
<tr>
<td>Halton *</td>
<td>0.0519</td>
<td>0.0232</td>
<td>0.0067</td>
</tr>
<tr>
<td>Halton scr</td>
<td>0.0527</td>
<td>0.0178</td>
<td>0.0056</td>
</tr>
<tr>
<td>Sobol *</td>
<td>0.0536</td>
<td>0.0123</td>
<td>0.0043</td>
</tr>
<tr>
<td>Sobol xor</td>
<td>0.0409</td>
<td>0.0124</td>
<td>0.0035</td>
</tr>
<tr>
<td>Sobol owen</td>
<td>0.0420</td>
<td>0.0129</td>
<td>0.0037</td>
</tr>
<tr>
<td>pj</td>
<td>0.0833</td>
<td>0.0335</td>
<td>0.0134</td>
</tr>
<tr>
<td>pjbn</td>
<td>0.0756</td>
<td>0.0304</td>
<td>0.0123</td>
</tr>
<tr>
<td>pmj</td>
<td>0.0529</td>
<td>0.0202</td>
<td>0.0078</td>
</tr>
<tr>
<td>pmjbn</td>
<td>0.0498</td>
<td>0.0193</td>
<td>0.0071</td>
</tr>
<tr>
<td>pmj02</td>
<td>0.0417</td>
<td>0.0128</td>
<td>0.0037</td>
</tr>
</tbody>
</table>

Table 2 shows star discrepancy $D^*$ for 64, 256, and 1024 sample points from various sample sequences. Discrepancies for the stochastic or randomized sample sequences are computed as the average of 100 trials; for deterministic sample sequences (marked with *) only one trial. The table shows that $D^*$ of pmj is better than random, best candidates, Perrier LDBN, and Ahmed ART, but worse than Halton and Sobol’, and that pmj02 is on par with the Owen-scrambled Sobol’ sequence, but that xor-scrambled Sobol’ is slightly better than both. This ordering does not correspond to the results we observed in Sections 7 to 9 in the main paper, where we found that Owen-scrambling is far better than xor-scrambling and rotation.

### 6.2. Arbitrary-edge discrepancy

Dobkin et al. [DM93, DEM96] introduced an alternative discrepancy measure, the maximum arbitrary-edge discrepancy $D_{ae}$. Like $D^*$ it measures the maximum difference between an area and the fraction of samples within it, but it considers all straight edges through the unit square – not just axis-aligned boxes. This corresponds to the maximum error we would see, for example, in a pixel with a sharp (object or texture) straight edge in it. (This discrepancy measure is isotropic, i.e. does not focus on just one or two directions, but not as general as the isotropic discrepancy $J$ of Kuipers and Niederreiter [KN74].)

#### Table 3: Arbitrary-edge discrepancy $D_{ae}$ for 64, 256, and 1024 samples from various progressive sample sequences.

<table>
<thead>
<tr>
<th>sequence</th>
<th>64</th>
<th>256</th>
<th>1024</th>
</tr>
</thead>
<tbody>
<tr>
<td>random</td>
<td>0.1548</td>
<td>0.0794</td>
<td>0.0396</td>
</tr>
<tr>
<td>best cand</td>
<td>0.0798</td>
<td>0.0358</td>
<td>0.0160</td>
</tr>
<tr>
<td>Perrier *</td>
<td>0.0722</td>
<td>0.0267</td>
<td>0.0111</td>
</tr>
<tr>
<td>Ahmed</td>
<td>0.0687</td>
<td>0.0238</td>
<td>0.0108</td>
</tr>
<tr>
<td>Halton</td>
<td>0.1089</td>
<td>0.0367</td>
<td>0.0139</td>
</tr>
<tr>
<td>Halton scr</td>
<td>0.0814</td>
<td>0.0345</td>
<td>0.0134</td>
</tr>
<tr>
<td>Sobol *</td>
<td>0.1112</td>
<td>0.0367</td>
<td>0.0263</td>
</tr>
<tr>
<td>Sobol xor</td>
<td>0.1046</td>
<td>0.0366</td>
<td>0.0262</td>
</tr>
<tr>
<td>Sobol owen</td>
<td>0.0705</td>
<td>0.0282</td>
<td>0.0108</td>
</tr>
<tr>
<td>pj</td>
<td>0.0742</td>
<td>0.0295</td>
<td>0.0116</td>
</tr>
<tr>
<td>pjbn</td>
<td>0.0702</td>
<td>0.0282</td>
<td>0.0110</td>
</tr>
<tr>
<td>pmj</td>
<td>0.0733</td>
<td>0.0292</td>
<td>0.0114</td>
</tr>
<tr>
<td>pmjbn</td>
<td>0.0676</td>
<td>0.0277</td>
<td>0.0107</td>
</tr>
<tr>
<td>pmj02</td>
<td>0.0702</td>
<td>0.0278</td>
<td>0.0109</td>
</tr>
</tbody>
</table>

Table 3 shows $D_{ae}$ for the progressive sequences. For non-deterministic sequences the discrepancy value is again computed as the average of 100 samples (trials). With this discrepancy measure, Owen-scrambled Sobol’ and all five pj/pmj/pmj02 sequences are found to be far better than Halton and the other Sobol’ sequences. The canonical Sobol’ sequence is correctly “penalized” for its alignment of sample points along diagonals and for clumped samples, and xor-scrambling helps only a little. Ahmed ART is best or among the best for 64 and 256 samples, while for 1024 samples, Ahmed ART, Owen-scrambled Sobol’, pmjbn, and pmj02 are tied for best. This discrepancy measure does not capture the fact that Ahmed ART sequences have poor 1D stratification.

### 6.3. Discussion

To summarize: according to the star discrepancy, xor-scrambled Sobol’ sequences are slightly better than Owen-scrambled Sobol’ sequences and pmj02 sequences; but according to the arbitrary-edge discrepancy, pmj/pmj/pmj02 sequences and Owen-scrambled
Sobol’ sequences are superior to Halton and the other Sobol’ sequences, and Ahmed ART sequences are as good as the best. The arbitrary-edge results are closest to what we found in the more realistic tests in Sections 7–9, but even that discrepancy measure does not account for the important facts that some sequences give significantly lower error when sampling smooth functions and that 1D projections are important.

Our conclusion from these results is that selecting one sequence over another based solely on discrepancy is misguided! – just as it is the case for sample sets. Instead, more realistic tests such as those we have presented in Sections 7 through 9 in the main paper and in this supplemental material must be performed.

References