Eurographics Symposium on Rendering 2018
T. Hachisuka and W. Jakob
(Guest Editors)

Volume 37 (2018), Number 4

Progressive Multi-Jittered Sample Sequences: Supplemental Materials

Per Christensen Andrew Kensler Charlie Kilpatrick

Pixar Animation Studios

Abstract

Supplemental material for the paper "Progressive multi-jittered sample sequences": Pseudocode for sample generation. Test of zone plate images with maximum error metric. Test of the sample sequences for pixel sampling with a Gaussian pixel filter and for sampling rectangular area lights. Comparing sample sequences with sample sets. A discussion of discrepancy.

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 GENERATEPJ(M)
    // Generate first sample point at random position
    samples[0] ← (rnd(), rnd())
    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 EXTENDSEQUENCE(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[N+s] ← generateSamplePoint(i, j, xhalf, yhalf, n)
        // Then randomly select one of the two remaining sub-quadrants
        if rnd() > 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[3*N+s] ← generateSamplePoint(i, j, xhalf, yhalf, n)
    end for
end procedure
```

Generate a sample point:

```
function GENERATESAMPLEPOINT(i, j, xhalf, yhalf, n)
    pt.x ← (i + 0.5 * (xhalf + rnd())) / n
    pt.y ← (j + 0.5 * (yhalf + rnd())) / 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. rnd() is a function that returns pseudo-random numbers between 0 and 1 (for example drand48() or a table lookup), and $\lfloor x \rfloor$ 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 extend-Sequence() function into two. Main function:

```
procedure GENERATEPMJ(M)
    // Generate first sample point at random position
    samples[0] ← (rnd(), rnd())
    N ← 1
    while N < M do
```

```
        // Generate next 3N sample points
        extendSequenceEven(N, samples, numSamples) // N even pow2
        extendSequenceOdd(2*N, samples, numSamples)//2N odd pow2
        N ← 4*N
    end while
end procedure
```

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

```
procedure EXTENDSEQUENCEEVEN(N)
    n ← √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 ← 0 ... N-1 do
        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)⌋
        // Select the diagonally opposite subquadrant
        xhalf ← 1-xhalf
        yhalf ← 1-yhalf
        // Generate a sample point
        generateSamplePoint(i, j, xhalf, yhalf, n, N)
    end for
end procedure
```

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

```
procedure EXTENDSEQUENCEODD(N)
    n ← √(N/2)
    // Mark already occupied 1D strata so we can avoid them
    markOccupiedStrata(N)
    // (Optionally:
    // 1) Classify occupied sub-pixels: odd or even diagonal
    // 2) Pre-select well-balanced subquadrants here for better
    // sample distribution between powers of two samples)
    // Loop over N/2 old samples and generate 2 new samples for each
    // – one at a time to keep the order consecutive (for "greedy"
    // best candidates)
    // Select one of the two remaining subquadrants
    for s ← 0 ... N/2-1 do
        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)⌋
        // Randomly select one of the two remaining subquadrants
        // (Or optionally use the well-balanced subquads chosen above)
        if rnd() > 0.5 then
            xhalf ← 1-xhalf
        else
            yhalf ← 1-yhalf
        end if
        xhalves[s] ← xhalf
        yhalves[s] ← yhalf
        // Generate a sample point
        generateSamplePoint(i, j, xhalf, yhalf, n, N)
    end for
    // And finally fill in the last subquadrants
    for s ← 0 ... N/2-1 do
        oldpt ← samples[s]
        i ← ⌊n * oldpt.x⌋
        j ← ⌊n * oldpt.y⌋
        xhalf ← 1-xhalves[s]
        yhalf ← 1-yhalves[s]
        // Generate a sample point
        generateSamplePoint(i, j, xhalf, yhalf, n, N)
    end for
end procedure
```

Mark all occupied 1D strata:

```
procedure MARKOCCUPIEDSTRATA(N)
    NN ← 2*N
    occupied1Dx[0 ... NN-1] ← false // init array
    occupied1Dy[0 ... NN-1] ← false // init array
    for s ← 0 ... N-1 do
        xstratum ← ⌊NN * samples[s].x⌋
        ystratum ← ⌊NN * samples[s].y⌋
        occupied1Dx[xstratum] ← true
        occupied1Dy[ystratum] ← true
    end for
end procedure
```

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

```
procedure GENERATESAMPLEPOINT(i, j, xhalf, yhalf, n, N)
    NN ← 2*N
    bestDist ← 0.0
    numCand ← 10 // number of candidate points
    // Generate candidate points and pick the best
    for t ← 1 ... numCand do
        // Generate candidate sample x coord
        repeat
            candpt.x ← (i + 0.5 * (xhalf + rnd())) / n
            xstratum ← ⌊NN * candpt.x⌋
        until not occupied1Dx[xstratum]
        // Generate candidate sample y coord
        repeat
            candpt.y ← (j + 0.5 * (yhalf + rnd())) / n
            ystratum ← ⌊NN * candpt.y⌋
        until not occupied1Dy[ystratum]
        // Evaluate distance between candidate point and existing samples
        d ← minDist(candpt)
        // Keep candidate point if it has higher dist than best so far
        if d > bestDist then
            bestDist ← d; pt ← candpt
        end if
    end for
    // Mark 1D strata as occupied
    xstratum ← ⌊NN * pt.x⌋
    ystratum ← ⌊NN * pt.y⌋
    occupied1Dx[xstratum] ← true
    occupied1Dy[ystratum] ← true
    // Assign new sample point
    samples[numSamples] ← pt
    numSamples ← numSamples+1
end procedure
```

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

The optional procedure to classify already occupied coarse subquads as even or odd diagonals, ie. '/' or '\':

```
procedure CLASSIFYSUBQUADS(n, N)
    nn = 2*n
    for s ← 0 ... N/2-1 do
        xstratum ← ⌊nn * samples[s].x⌋
        ystratum ← ⌊nn * samples[s].y⌋
        evenx ← xstratum mod 2
        eveny ← ystratum mod 2
        evendiags[ystratum/2][xstratum/2] ← (evenx = eveny)
    end for
end procedure
```

Here evenx and eveny are Booleans, and evendiags[][] 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:

```
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 ← (jj = n-abs(choiceBalanceX[i])) and n > 1
            evendiag ← evendiags[j][i]
            // If last entry in a column: enforce x balance
            if choiceBalanceY[j] ≠ 0 and not last then
                neg ← choiceBalanceY[j] < 0 // more y lows than highs
                // Do opposite y choice than previous column
                subquadchoicesY[j][i] ← neg ? 1 : 0
                subquadchoicesX[j][i]] ← evendiag xor neg ? 1 : 0
                choiceBalanceY[j] += neg ? 1 : -1
                choiceBalanceX[i] += evendiag xor neg ? 1 : -1
            else if choiceBalanceX[i] ≠ 0 then
                neg ← choiceBalanceX[i] < 0 // more x lows than highs
                subquadchoicesX[j][i] ← neg ? 1 : 0
                subquadchoicesY[j][i] ← evendiag xor neg ? 1 : 0
                choiceBalanceX[i] += neg ? 1 : -1
                choiceBalanceY[j] += 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[j][i] ← xhalf
                subquadchoicesY[j][i] ← yhalf
                choiceBalanceX[i] += xhalf ? 1 : -1
                choiceBalanceY[j] += yhalf ? 1 : -1
            end if
        end for
    end for
    if n = 1 then
        return true // fine even though not balanced
    end if
    for i ← 0 ... n-1 do
        if choiceBalanceY[i] ≠ 0 then
            return false
        end if
    end for
    return true // all is balanced
end function
```

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) :

```
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:

```
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
        shape ← shape+1; xdivs ← xdivs/2; ydivs ← ydivs*2
    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.

```
procedure GENERATESAMPLEPOINT(i, 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:

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

```
    end if
    shape ← shape+1; xdivs ← xdivs/2; ydivs ← ydivs*2
  until xdivs = 0
  return false // ok: sample pt is not in any occupied stratum
end function
```

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 $O(N^{-1.35})$ at powers of two.

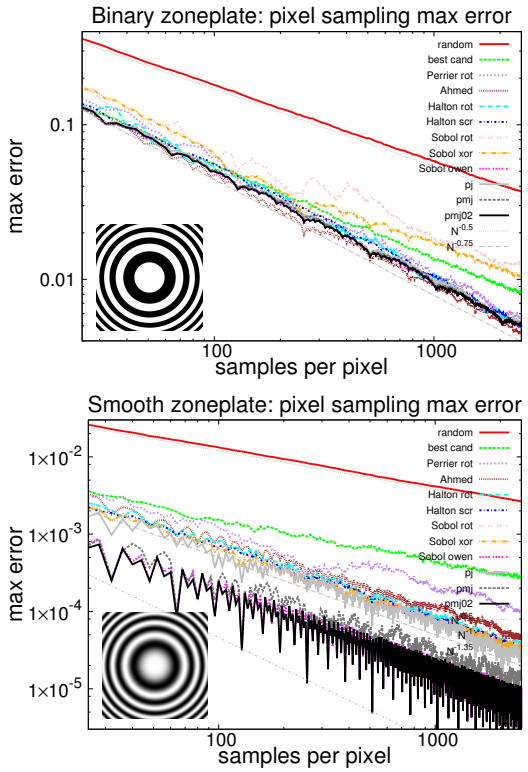

Figure 1: *Maximum error for pixel sampling of binary and smooth zone plate images. (25–2500 samples per pixel.)*

3. Pixel sampling with Gaussian filter

The tests in Section 8 of the main paper used a 1×1 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 2×2 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.

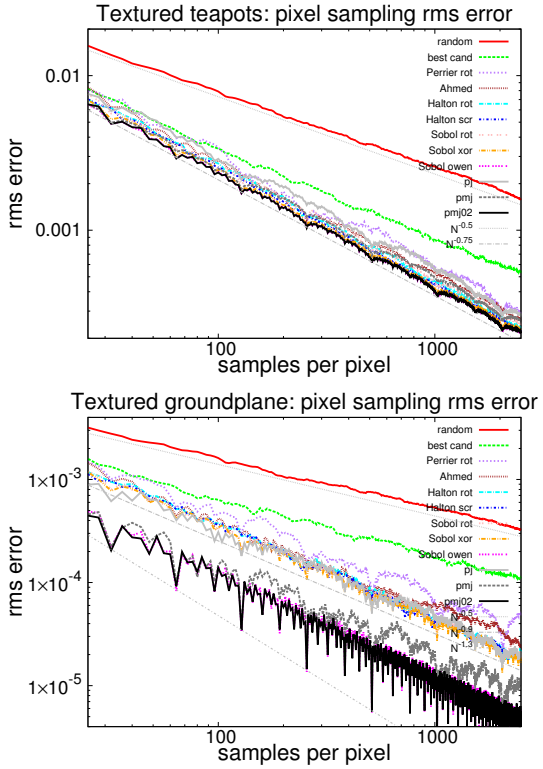

Figure 2: *rms error for pixel sampling with Gaussian pixel filter. Top: teapot edges. Bottom: smooth ground plane.*

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 4×0.25 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.

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 $O(N^{-0.75})$. 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 $O(N^{-0.5})$, most other sample sequences converge as $O(N^{-1})$, 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.

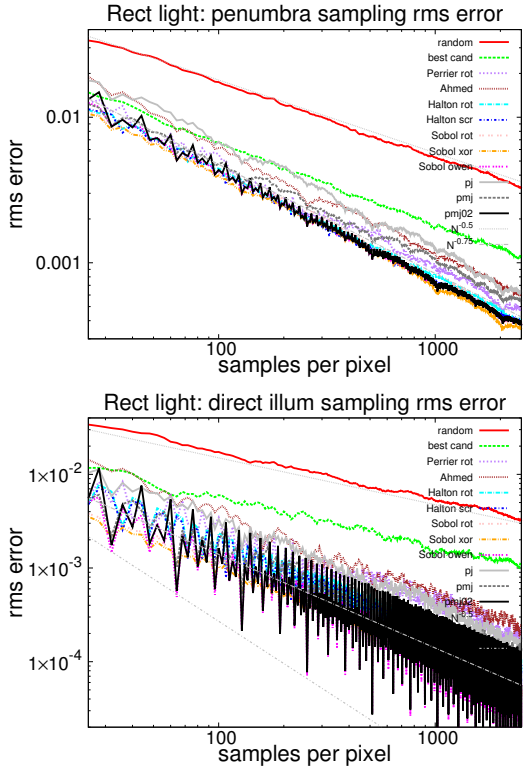

Figure 3: *rms error for 25–2500 samples per pixel of a rectangular light source. Top: penumbra region. Bottom: fully illuminated region.*

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 using the usual progressive sample sequences as well as various sample sets: uniform jittered, jittered, multi-jittered, correlated multi-jittered (cmj), Hammersley, Larcher-Pillichshammer, maximized-minimum distance (t,m,s) nets [GK08], and CBC rank-1 lattices [LM12]. Observations:

- Uniform jittered sets are very bad – they have nearly as high error as uniform random.
- Multi-jittered sets have lower error than uniform jittered, jittered, cmj, and most quasi-random sequences.
- Cmj sets have unexpectedly high error; their correlation is actually bad for sampling monotonic functions like the Gaussian (or a sharp edge through a pixel).
- Progressive jittered (pj) sequences have error that matches jittered sets – as it should since the sample numbers in the table are powers of two. Similarly, progressive multi-jittered (pmj) sequences have error that matches multi-jittered sets.
- Pmj02 and Owen-scrambled Sobol' sequences have much lower error than even the best sample sets when the number of samples is a power of two. (This is a noteworthy result.)

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.*

sequence/set	256	1024	4096
random	0.010774	0.005388	0.002731
best cand	0.003542	0.001676	0.000819
Perrier rot	0.000870	0.000666	0.000063
Ahmed	0.001100	0.000227	0.000071
Halton rot	0.001147	0.000337	0.000096
Halton scr	0.000953	0.000254	0.000066
Sobol rot	0.000971	0.000238	0.000065
Sobol xor	0.000621	0.000154	0.000038
Sobol owen	**0.000064**	**0.000008**	**0.000001**
pj	0.000670	0.000166	0.000042
pmj	0.000191	0.000046	0.000011
pmj02	**0.000064**	**0.000009**	**0.000001**
uniform jittered	0.009851	0.004959	0.002458
jittered	0.000663	0.000167	0.000042
multi-jittered	0.000184	0.000044	0.000011
cmj	0.000670	0.000231	0.000080
Hammersley rot	0.000768	0.000199	0.000051
Hammersley scr	0.000712	0.000183	0.000048
Larcher-Pil rot	0.000771	0.000199	0.000052
Larcher-Pil scr	0.000622	0.000158	0.000039
Grünschloß rot	0.000763	0.000200	0.000052
Rank-1 lattice rot	0.000747	0.000195	0.000049

For a different comparison of sample sets and sequences, we tested pixel sampling of the checkered teapots image 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,

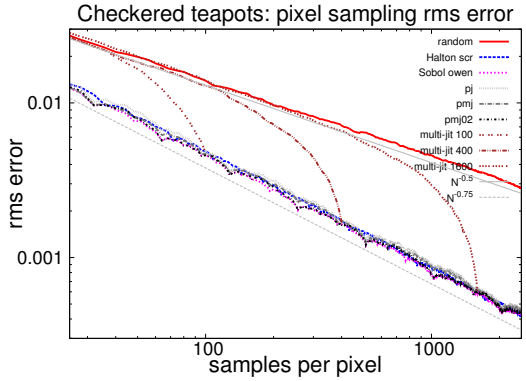

Figure 4: *rms error for pixel sampling of the checkered teapots image sampled with progressive sample sequences and multi-jittered sample sets. The sets have slow initial convergence.*

ÖS98]. 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.*

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

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.*

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

Table 3 shows D_{ae} for the progressive sequences. For non-deterministic sequences the discrepancy value is again computed as the average of 100 sequences (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, pm/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

[Chr18] CHRISTENSEN P.: *Progressive sampling strategies for disk light sources*. Tech. Rep. 18-02, Pixar Animation Studios, 2018. 5

[DEM96] DOBKIN D., EPPSTEIN D., MITCHELL D.: Computing the discrepancy with applications to supersampling patterns. *ACM Transactions on Graphics 15*, 4 (1996), 354–376. 6

[DH06] DUNBAR D., HUMPHREYS G.: A spatial data structure for fast Poisson-disk sample generation. *ACM Transactions on Graphics (Proc. SIGGRAPH) 25*, 3 (2006), 503–508. 2

[DM93] DOBKIN D., MITCHELL D.: Random-edge discrepancy of supersampling patterns. *Proc. Graphics Interface* (1993), 62–69. 6

[ESG06] ERNST M., STAMMINGER M., GREINER G.: Filter importance sampling. In *Proc. IEEE Symposium on Interactive Ray Tracing* (2006), pp. 125–132. 4

[GK08] GRÜNSCHLOSS L., KELLER A.: (t,m,s)-nets and maximized minimum distance, part ii. In *Proc. Monte Carlo and Quasi-Monte Carlo Methods* (2008). 5, 6

[Gla95] GLASSNER A.: *Principles of Digital Image Synthesis*. Morgan Kaufmann, 1995. 5

[KN74] KUIPERS L., NIEDERREITER H.: *Uniform Distribution of Sequences*. Dover Publications, 1974. 6

[LM12] L'ECUYER P., MUNGER D.: Algorithm 958: Lattice builder: A general software tool for constructing rank-1 lattice rules. *ACM Transactions of Mathematical Software 42*, 2 (2012). 5

[MF92] MCCOOL M., FIUME E.: Hierarchical Poisson disk sampling distributions. In *Proc. Graphics Interface* (1992), pp. 94–105. 2

[Mit92] MITCHELL D.: Ray tracing and irregularities in distribution. *Proc. Eurographics Workshop on Rendering* (1992), 61–69. 5

[Nie92] NIEDERREITER H.: *Random Number Generation and Quasi-Monte Carlo Methods*. SIAM, 1992. 6

[ÖS98] ÖKTEN G., SHAH M.: *Random and deterministic digit-scrambling of the Halton sequence*. Tech. rep., Florida State University, 1998. 5

[PH10] PHARR M., HUMPHREYS G.: *Physically Based Rendering: From Theory To Implementation*, 2nd ed. Morgan Kaufmann, 2010. 5

[Pur86] PURGATHOFER W.: A statistical method for adaptive stochastic sampling. In *Proc. Eurographics* (1986), pp. 145–152. 4

[Shi91] SHIRLEY P.: Discrepancy as a quality measure for sample distributions. *Proc. Eurographics* (1991), 183–193. 5

[Zar68] ZAREMBA S.: The mathematical basis of Monte Carlo and quasi-Monte Carlo methods. *SIAM Review 10*, 3 (1968), 303–314. 6