Coding Example: Blue Noise Sampling using Bridson method

In this lesson, we will try to do the blue noise sampling using the Bridson method. First we will discuss the step-by-step approach and then implement it in code.

We'll cover the following

Bridson method

If the vectorization of the previous method poses no real difficulty, the speed improvement is not so good and the quality remains low and dependent on the k parameter. The higher, the better since it basically governs how hard to try to insert a new sample. But, when there is already a large number of accepted samples, only chance allows us to find a position to insert a new sample. We could increase the k value but this would make the method even slower without any guarantee in quality. It's time to think out-of-the-box and luckily enough, Robert Bridson did that for us and proposed a simple yet efficient method:

Step 0:

Initialize an n-dimensional background grid for storing samples and accelerating spatial searches. We pick the cell size to be bounded by $\frac{r}{\sqrt{n}}$, so that each grid cell will contain at most one sample, and thus the grid can be implemented as a simple n-dimensional array of integers: the default `-1` indicates no sample, a non-negative integer gives the index of the sample located in a cell.

Step 1:

Select the initial sample, `x_0`, randomly chosen uniformly from the domain. Insert it into the background grid, and initialize the "active list" (an array of sample indices) with this index (zero).

Step 2:

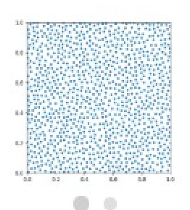
While the active list is not empty, choose a random index from it (say i). Generate up to k points chosen uniformly from the spherical annulus between radius r and $2r$ around x_i. For each point in turn, check if it is within distance r of existing samples (using the background grid to only test nearby samples). If a point is adequately far from existing samples, emit it as the next sample and add it to the active list. If after k attempts no such point is found, instead remove i from the active list.

Implementation

The implementation poses no real problem. Note that not only is this method fast, but it also offers a better quality (more samples) than the DART method even with a high k parameter. Here's the complete `Bridson` implementation:

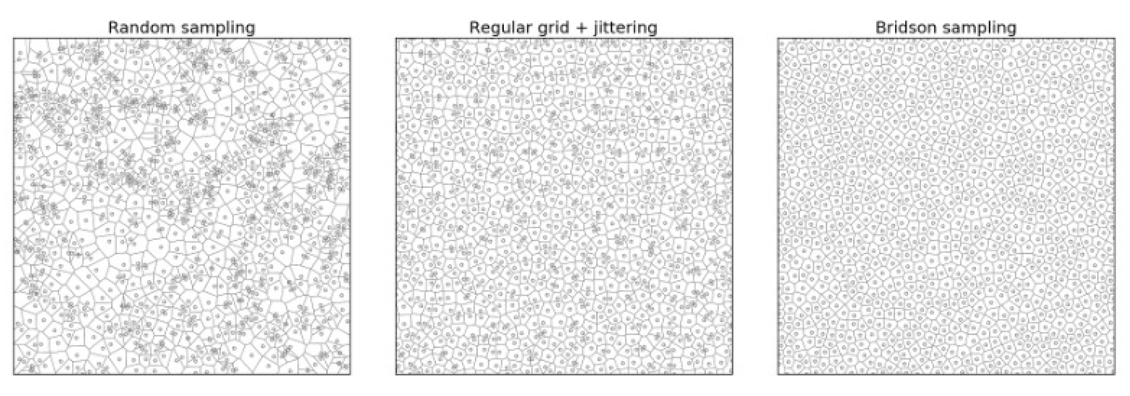
```python
# -----------------------------------------------------------------------------
# From Numpy to Python
# Copyright (2017) Nicolas P. Rougier - BSD license
# More information at https://github.com/rougier/numpy-book
# -----------------------------------------------------------------------------
import numpy as np
import matplotlib.pyplot as plt


def Bridson_sampling(width=1.0, height=1.0, radius=0.025, k=30):
    # References: Fast Poisson Disk Sampling in Arbitrary Dimensions
    #             Robert Bridson, SIGGRAPH, 2007
    def squared_distance(p0, p1):
        return (p0[0]-p1[0])**2 + (p0[1]-p1[1])**2

    def random_point_around(p, k=1):
        # WARNING: This is not uniform around p but we can live with it
        R = np.random.uniform(radius, 2*radius, k)
        T = np.random.uniform(0, 2*np.pi, k)
        P = np.empty((k, 2))
        P[:, 0] = p[0]+R*np.sin(T)
        P[:, 1] = p[1]+R*np.cos(T)
        return P

    def in_limits(p):
        return 0 <= p[0] < width and 0 <= p[1] < height

    def neighborhood(shape, index, n=2):
        row, col = index
        row0, row1 = max(row-n, 0), min(row+n+1, shape[0])
        col0, col1 = max(col-n, 0), min(col+n+1, shape[1])
        I = np.dstack(np.mgrid[row0:row1, col0:col1])
        I = I.reshape(I.size//2, 2).tolist()
        I.remove([row, col])
        return I

    def in_neighborhood(p):
        i, j = int(p[0]/cellsize), int(p[1]/cellsize)
        if M[i, j]:
            return True
        for (i, j) in N[(i, j)]:
            if M[i, j] and squared_distance(p, P[i, j]) < squared_radius:
                return True
        return False

    def add_point(p):
        points.append(p)
        i, j = int(p[0]/cellsize), int(p[1]/cellsize)
        P[i, j], M[i, j] = p, True

    # Here `2` corresponds to the number of dimension
    cellsize = radius/np.sqrt(2)
    rows = int(np.ceil(width/cellsize))
    cols = int(np.ceil(height/cellsize))

    # Squared radius because we'll compare squared distance
    squared_radius = radius*radius

    # Positions cells
    P = np.zeros((rows, cols, 2), dtype=np.float32)
    M = np.zeros((rows, cols), dtype=bool)

    # Cache generation for neighborhood
    N = {}
    for i in range(rows):
        for j in range(cols):
            N[(i, j)] = neighborhood(M.shape, (i, j), 2)

    points = []
    add_point((np.random.uniform(width), np.random.uniform(height)))
    while len(points):
        i = np.random.randint(len(points))
        p = points[i]
        del points[i]
        Q = random_point_around(p, k)
        for q in Q:
            if in_limits(q) and not in_neighborhood(q):
                add_point(q)
    return P[M]


if __name__ == '__main__':

    plt.figure()
    plt.subplot(1, 1, 1, aspect=1)

    points = Bridson_sampling()
    X = [x for (x, y) in points]
    Y = [y for (x, y) in points]
    plt.scatter(X, Y, s=10)
    plt.xlim(0, 1)
    plt.ylim(0, 1)
    plt.savefig("output/BridsonSampling.png")
    plt.show()
```

RUN SAVE RESET



Random vs. Regular vs. Bridson Sampling

The image below shows the pattern of three samplings, i.e., Random Sampling, Regular Sampling and Bridson Sampling.



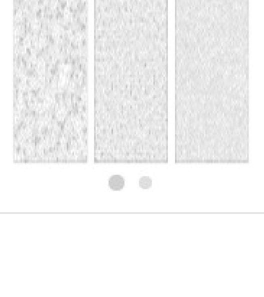
Comparison of uniform, grid-jittered and Bridson sampling.

Here's the code to plot all three samplings:

main.py
voronoi.py

```python
# -----------------------------------------------------------------------------
# From Numpy to Python
# Copyright (2017) Nicolas P. Rougier - BSD license
# More information at https://github.com/rougier/numpy-book
# -----------------------------------------------------------------------------
import numpy as np

def Bridson_sampling(width=1.0, height=1.0, radius=0.025, k=30):
    # References: Fast Poisson Disk Sampling in Arbitrary Dimensions
    #             Robert Bridson, SIGGRAPH, 2007
    def squared_distance(p0, p1):
        return (p0[0]-p1[0])**2 + (p0[1]-p1[1])**2

    def random_point_around(p, k=1):
        # WARNING: This is not uniform around p but we can live with it
        R = np.random.uniform(radius, 2*radius, k)
        T = np.random.uniform(0, 2*np.pi, k)
        P = np.empty((k, 2))
        P[:, 0] = p[0]+R*np.sin(T)
        P[:, 1] = p[1]+R*np.cos(T)
        return P

    def in_limits(p):
        return 0 <= p[0] < width and 0 <= p[1] < height

    def neighborhood(shape, index, n=2):
        row, col = index
        row0, row1 = max(row-n, 0), min(row+n+1, shape[0])
        col0, col1 = max(col-n, 0), min(col+n+1, shape[1])
        I = np.dstack(np.mgrid[row0:row1, col0:col1])
        I = I.reshape(I.size//2, 2).tolist()
        I.remove([row, col])
        return I

    def in_neighborhood(p):
        i, j = int(p[0]/cellsize), int(p[1]/cellsize)
        if M[i, j]:
            return True
        for (i, j) in N[(i, j)]:
            if M[i, j] and squared_distance(p, P[i, j]) < squared_radius:
                return True
        return False

    def add_point(p):
        points.append(p)
        i, j = int(p[0]/cellsize), int(p[1]/cellsize)
        P[i, j], M[i, j] = p, True

    # Here `2` corresponds to the number of dimension
    cellsize = radius/np.sqrt(2)
    rows = int(np.ceil(width/cellsize))
    cols = int(np.ceil(height/cellsize))

    # Squared radius because we'll compare squared distance
    squared_radius = radius*radius

    # Positions cells
    P = np.zeros((rows, cols, 2), dtype=np.float32)
    M = np.zeros((rows, cols), dtype=bool)

    # Cache generation for neighborhood
    N = {}
    for i in range(rows):
        for j in range(cols):
            N[(i, j)] = neighborhood(M.shape, (i, j), 2)

    points = []
    add_point((np.random.uniform(width), np.random.uniform(height)))
    while len(points):
        i = np.random.randint(len(points))
        p = points[i]
        del points[i]
        Q = random_point_around(p, k)
        for q in Q:
            if in_limits(q) and not in_neighborhood(q):
                add_point(q)
    return P[M]


def draw_voronoi(ax, X, Y):
    from voronoi import voronoi
    from matplotlib.path import Path
    from matplotlib.patches import PathPatch
    cells, triangles, circles = voronoi(X, Y)
    for i, cell in enumerate(cells):
        codes = [Path.MOVETO] \
                + [Path.LINETO] * (len(cell)-2) \
                + [Path.CLOSEPOLY]
        path = Path(cell, codes)
        patch = PathPatch(path,
                          facecolor="none", edgecolor="0.5", linewidth=0.5)
        ax.add_patch(patch)


# -----------------------------------------------------------------------------
if __name__ == '__main__':
    import matplotlib.pyplot as plt

    # Benchmark
    # from tools import print_timeit
    # print_timeit("poisson_disk_sample()", globals())

    fig = plt.figure(figsize=(18, 6))

    ax = plt.subplot(1, 3, 1, aspect=1)
    n = 1000
    X = np.random.uniform(0, 1, n)
    Y = np.random.uniform(0, 1, n)
    ax.scatter(X, Y, s=10, facecolor='w', edgecolor='0.5')
    ax.set_xlim(0, 1), ax.set_ylim(0, 1)
    ax.set_xticks([]), ax.set_yticks([])
    ax.set_title("Random sampling", fontsize=18)
    draw_voronoi(ax, X, Y)

    ax = plt.subplot(1, 3, 2, aspect=1)
    n = 32
    X, Y = np.meshgrid(np.linspace(0, 1, n), np.linspace(0, 1, n))
    X += 0.45*np.random.uniform(-1/n, 1/n, (n, n))
    Y += 0.45*np.random.uniform(-1/n, 1/n, (n, n))
    ax.scatter(X, Y, s=10, facecolor='w', edgecolor='0.5')
    ax.set_xlim(0, 1), ax.set_ylim(0,  1)
    ax.set_xticks([]), ax.set_yticks([])
    ax.set_title("Regular grid + jittering", fontsize=18)
    draw_voronoi(ax, X.ravel(), Y.ravel())

    ax = plt.subplot(1, 3, 3, aspect=1)
    P = Bridson_sampling(width=1.0, height=1.0, radius=0.025, k=30)
    plt.scatter(P[:, 0], P[:, 1], s=10, facecolor='w', edgecolor='0.5')
    ax.set_xlim(0, 1), ax.set_ylim(0, 1)
    ax.set_xticks([]), ax.set_yticks([])
    ax.set_title("Bridson sampling", fontsize=18)
    draw_voronoi(ax, P[:, 0], P[:, 1])

    plt.tight_layout(pad=2.5)
    plt.savefig("output/sampling.png")
    plt.show()
```

RUN SAVE RESET



Further Readings

- Visualizing Algorithms, Mike Bostock, 2014.
- Stippling and Blue Noise, Jose Esteve, 2012.
- Poisson Disk Sampling, Herman Tulleken, 2009.
- Fast Poisson Disk Sampling in Arbitrary Dimensions, Robert Bridson, SIGGRAPH, 2007.