Hyperspherical Pseudorandom Vectors

Technical Report, GSI Technology

Brian Beckman
Technology Fellow
October, 2023
(Adapted from my work previously done at Amazon Robotics)

Abstract

We need uniformly distributed pseudorandom vectors in $d \geq 2$ dimensions for simulation, estimation, control, machine learning, optimization, search, test generation, and more. With vectors rooted at the origin, *uniformly distributed* means *with a constant likelihood of generating a vector endpoint anywhere inside a d-ball*. The number of dimensions needed can be very large, corresponding to the number of degrees of freedom in a physical system or the number of weights in a neural network. Rejection sampling in the d-cube is not feasible for even moderate numbers of dimensions because almost all vector endpoints are in the corners. For example, in twenty-seven dimensions, only one in a trillion vector endpoints in the d-cube is in the d-ball.

We demonstrate two computational methods that scale efficiently with d for generating pseudorandom vectors uniformly distributed on the unit d-sphere (surface of the unit hypersphere) and in the unit d-ball (interior of the unit hypersphere). Though no method here novel, descriptions available to me are unsatisfactory. The contributions of this note are proofs, executable specifications of algorithms, and numerical demonstrations.

DEFINITIONS

The unit d-ball is the interior of a hypersphere: the set of all real d-vectors rooted at the origin, $(x_1, x_2, \ldots, x_d) \in \mathbb{R}^d$, such that $x_1^2 + x_2^2 + \cdots + x_d^2 < 1$. The unit d-sphere is the *boundary* of the d-ball, that is, the set of all real d-vectors such that $x_1^2 + x_2^2 + \cdots + x_d^2 = 1$.

The unit d-cube is the set of all real d-vectors such that each coordinate is absolutely bounded by unity, inclusively, that is, where $-1 \leq x_{i \in [1..d]} \leq 1$. The unit d-ball, d-sphere, and d-cube are also said to have radius 1, diameter 2.

We do not distinguish between the mathematical procedure of *sampling* a distribution and the computational procedure of *generating pseudorandoms* according to a distribution. Likewise, we do not distinguish between mathematical true randoms and computational pseudorandoms.

Generating randoms uniformly means the same as *generating uniform randoms*. *Generating randoms symmetrically* means the same as *generating symmetric randoms*.

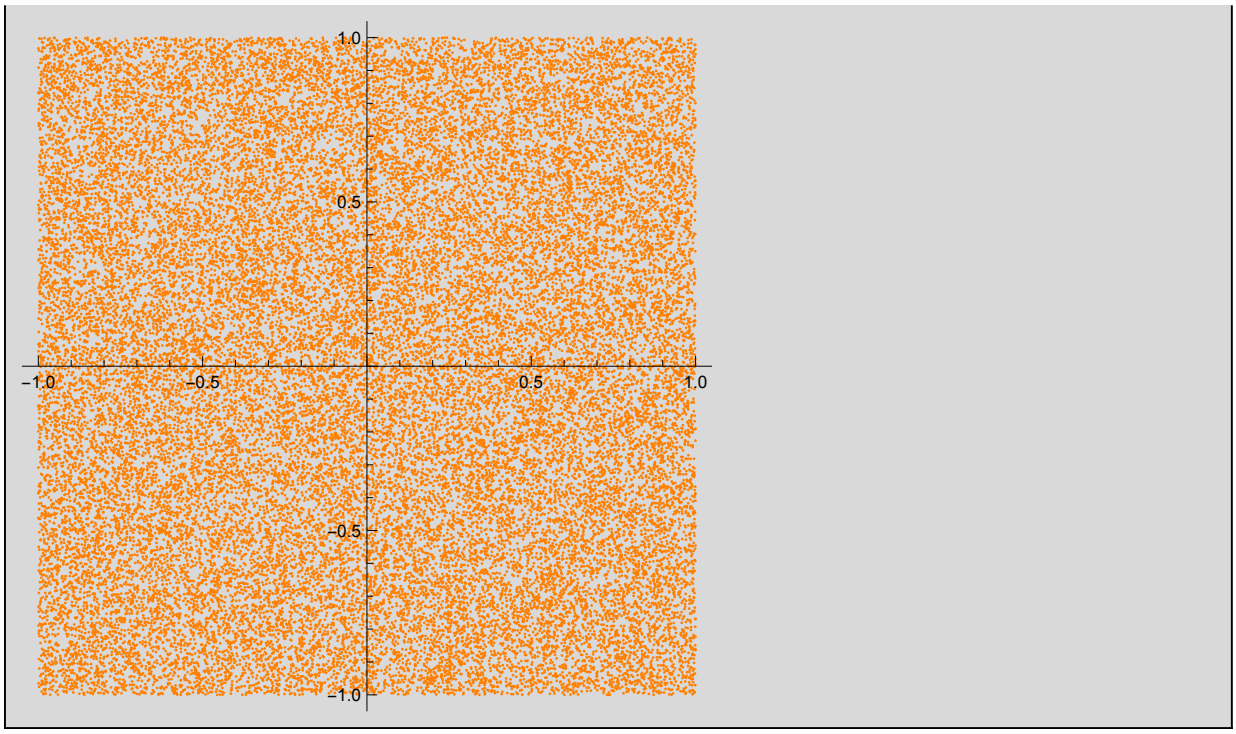
UNIFORM RANDOMS IN THE D-CUBE

Given an efficient method for generating uniform random scalars in the real interval [–1 .. × 1] — in the 1-cube — it is easy and efficient to generate rectangularly symmetric, uniform randoms in the *d*-cube: generate randoms for each of the *d* coordinate components independently.

Can we generate spherically symmetric uniform randoms starting with rectangularly symmetric, uniform randoms? Yes, but it takes some care to do it efficiently. Rejection methods — generating rectangularly symmetric, uniform randoms and then throwing out those that are not in the *d*-ball — are crippled: only one in a trillion endpoints in the 27-cube is also in the 27-ball; the ratio gets worse exponentially as *d* increases.

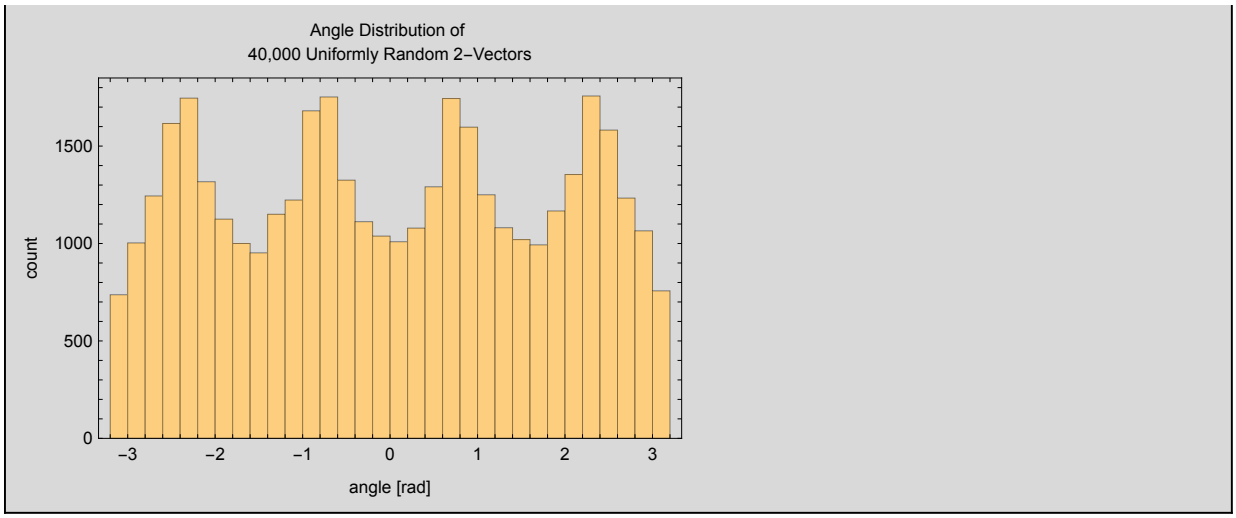
Consider points (vector endpoints) uniformly distributed in the unit 2-cube, i.e., the unit square. The resulting (*x*, *y*) pairs are not uniform with angle, therefore not spherically symmetric. The following diagrams illustrate: here are points uniformly distributed in the 2-cube:

Out[4]=



There are too many points in the corners, at ArcTan[1, 1] = 0.7854, ArcTan[–1, 1] = 2.3562 and their negatives:

Out[5]=



REJECTION METHOD: IMPRACTICAL

We could reject, i.e., remove points in the unit d-cube that are not in the unit d-ball. Those would be points with radial component — Euclidean norm — greater than 1. However, rejection does not scale. As d grows, the number of corners goes up as 2^d and almost all points are in the corners, outside the d-ball. To see this this, consider the ratio of the volume of a unit d-ball to the volume of a unit d-cube.

What is the volume of a unit d-cube? Answer: 2^d. What is the volume of a unit d-ball? (see https://en.wikipedia.org/wiki/Volume_of_an_n-ball)

Answer for integer d:

In[6]:=
```
ClearAll[vBall];
vBall[d_, r_ : 1] := Which[Mod[d, 2] == 0, With[{k = d / 2}, π^k / k! r^d],
  Mod[d, 2] == 1, With[{k = (d - 1) / 2}, 2 k! (4 π)^k / d! r^d]];
Table[<|"dimension" → d, "ball volume" → vBall[d], "cube volume" → 2^d,
   "cube/ball ratio" → N[2^d / vBall[d]]|>, {d, 1, 10}] // MatrixForm
```

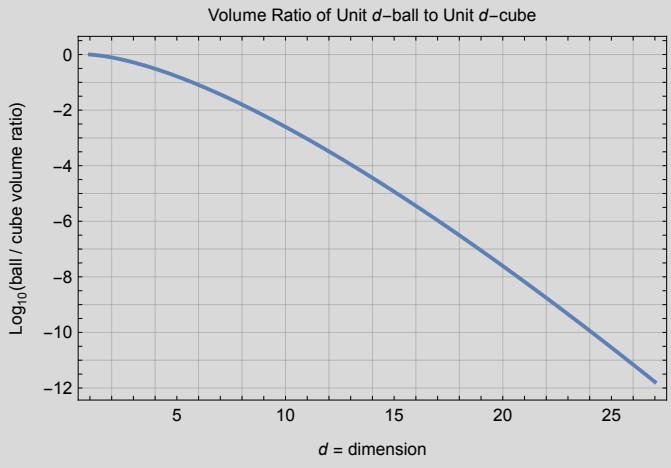
Out[8]//MatrixForm=

$$\begin{pmatrix} \langle|\text{dimension}\to 1,\ \text{ball volume}\to 2,\ \text{cube volume}\to 2,\ \text{cube/ball ratio}\to 1.|\rangle \\ \langle|\text{dimension}\to 2,\ \text{ball volume}\to \pi,\ \text{cube volume}\to 4,\ \text{cube/ball ratio}\to 1.27324|\rangle \\ \langle|\text{dimension}\to 3,\ \text{ball volume}\to \frac{4\pi}{3},\ \text{cube volume}\to 8,\ \text{cube/ball ratio}\to 1.90986|\rangle \\ \langle|\text{dimension}\to 4,\ \text{ball volume}\to \frac{\pi^2}{2},\ \text{cube volume}\to 16,\ \text{cube/ball ratio}\to 3.24228|\rangle \\ \langle|\text{dimension}\to 5,\ \text{ball volume}\to \frac{8\pi^2}{15},\ \text{cube volume}\to 32,\ \text{cube/ball ratio}\to 6.07927|\rangle \\ \langle|\text{dimension}\to 6,\ \text{ball volume}\to \frac{\pi^3}{6},\ \text{cube volume}\to 64,\ \text{cube/ball ratio}\to 12.3846|\rangle \\ \langle|\text{dimension}\to 7,\ \text{ball volume}\to \frac{16\pi^3}{105},\ \text{cube volume}\to 128,\ \text{cube/ball ratio}\to 27.0913|\rangle \\ \langle|\text{dimension}\to 8,\ \text{ball volume}\to \frac{\pi^4}{24},\ \text{cube volume}\to 256,\ \text{cube/ball ratio}\to 63.0742|\rangle \\ \langle|\text{dimension}\to 9,\ \text{ball volume}\to \frac{32\pi^4}{945},\ \text{cube volume}\to 512,\ \text{cube/ball ratio}\to 155.222|\rangle \\ \langle|\text{dimension}\to 10,\ \text{ball volume}\to \frac{\pi^5}{120},\ \text{cube volume}\to 1024,\ \text{cube/ball ratio}\to 401.543|\rangle \end{pmatrix}$$

for arbitrary real dimensions:

In[9]:=
```
ClearAll[vBall];
vBall[d_, r_ : 1] := π^(d/2) r^d / Gamma[1 + d / 2];
```

Out[11]=



The ratio of ball-to-cube volume in 27 dimensions is one trillionth, so the rejection method generates a trillion times more points than it keeps. Rejection is impractical for even moderately large *d*.

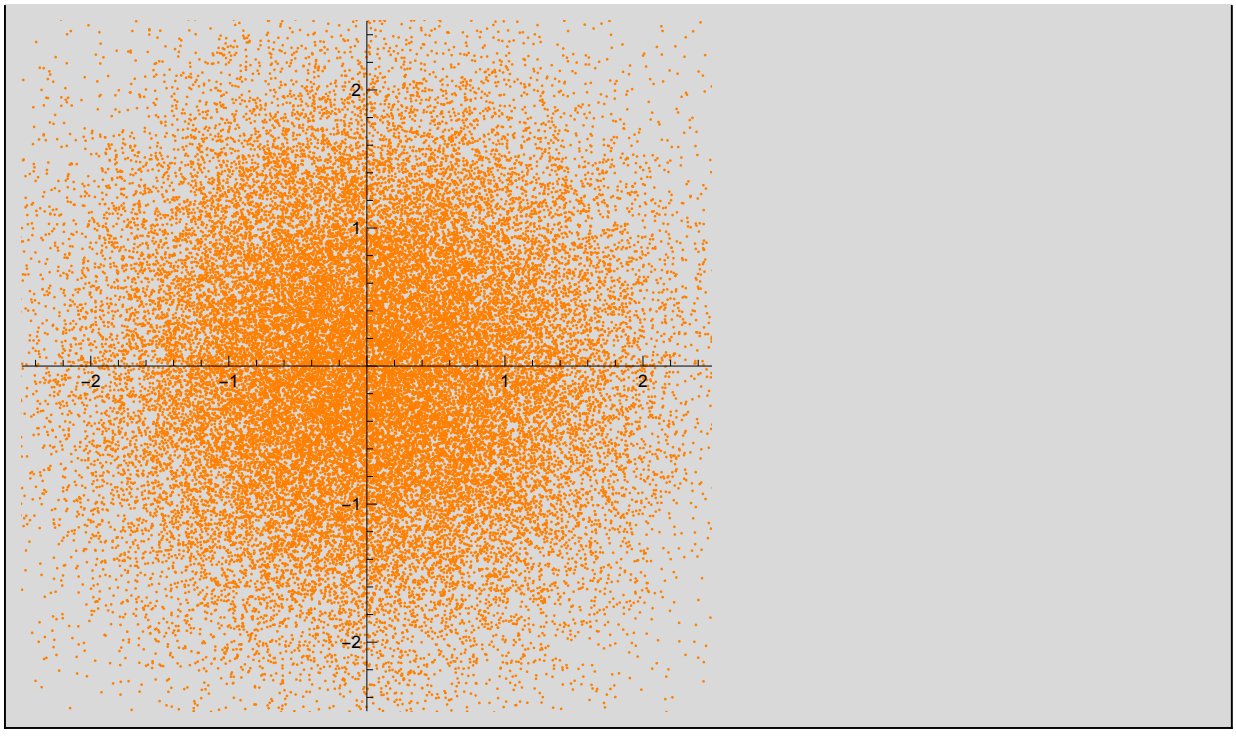
RADIAL SYMMETRY

Observe that a multivariate Gaussian or *multinormal* distribution is radially symmetric. Sampling x from $\frac{e^{-\frac{x^2}{2}}}{\sqrt{2\pi}}$, and y similarly, then noticing that x and y are independent (*the probability of their occurring simultaneously is the product of their individual probabilities*), we get $\frac{e^{-\frac{x^2}{2}}}{\sqrt{2\pi}} \times \frac{e^{-\frac{y^2}{2}}}{\sqrt{2\pi}} = \frac{e^{-\frac{1}{2}(x^2+y^2)}}{2\pi}$. A similar construction extends to any number of dimensions.

The multinormal distribution is radially symmetric because it depends only on the radius squared, $x^2 + y^2$, not on the azimuth angle. But it is not radially *uniform*, as the following figure demonstrates:

In[12]:=
```
show2D$[RandomVariate[
  MultinormalDistribution[IdentityMatrix[2]], 40 000], 2.5]
```

Out[12]=

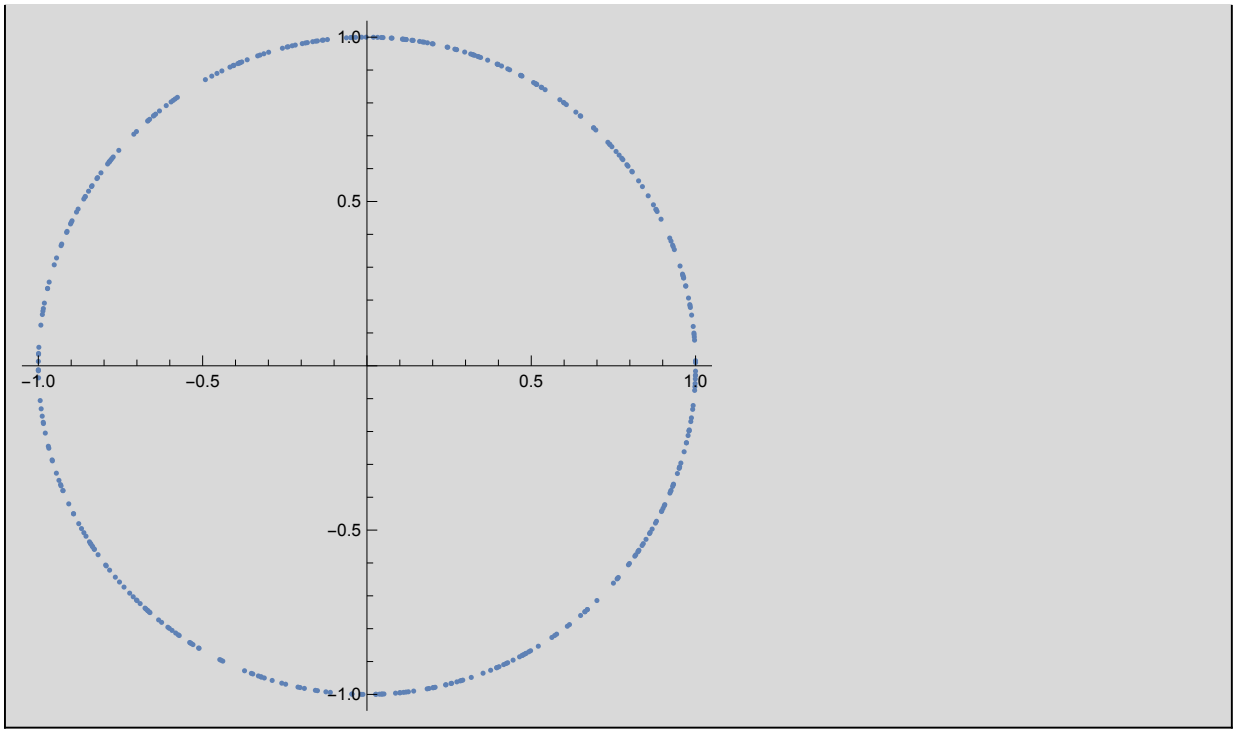


RADIAL UNIFORMITY

Due to radial symmetry, normalizing d-vectors from the d-ball (interior) produces d-vectors uniform on the d-sphere (surface). For example, the following points are uniformly distributed on the 2-sphere:

In[13]:=
```
With[{ϒ = 1.05},
 ListPlot[Normalize /@
   RandomVariate[
    MultinormalDistribution[IdentityMatrix[2]], 400],
  PlotRange → {{-ϒ, ϒ}, {-ϒ, ϒ}}, AspectRatio → 1]]
```

Out[13]=



FROM SYMMETRY TO UNIFORMITY

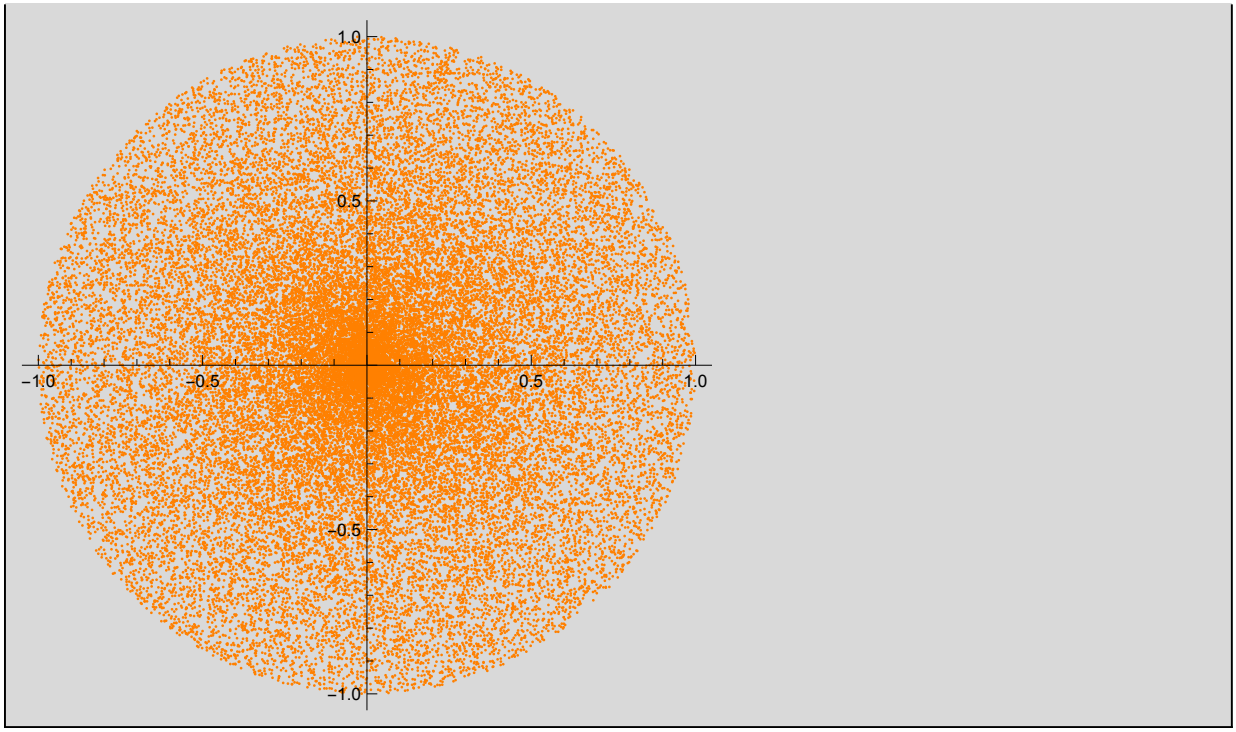
Now that we can generate uniform randoms on the *d*-sphere, the boundary of the *d*-ball, can we fill the interior uniformly? We show below two ways: a geometrical construction and a statistical construction.

GEOMETRICAL CONSTRUCTION

Reduce the lengths of vectors radially, pulling their end-points toward the center, preserving radial symmetry. But how to achieve spherical uniformity? Shrinking vectors — pulling them inward — by a uniform random scalar factor in [0, 1] will not work. The results will be too sparse in outer zones and too dense in inner zones:

In[14]:=
```
show2D$@With[{count = 40 000}, With[{pointsOnSphere =
      Normalize /@
       RandomVariate[
        MultinormalDistribution[IdentityMatrix[2]], count],
     scalar =
      RandomVariate[UniformDistribution[{0, 1}], count]},
    MapThread[Times, {pointsOnSphere, scalar}]]]
```

Out[14]=



If points were uniformly distributed with radius r, the volume density would be proportional to $r^{-d} = 1/r^d$, approaching infinity as $r \to 0$. If points were uniformly distributed in $r^{1/d}$, then the volume density would be constant: that is what we want. The following is code we re-use in the rest of this note.

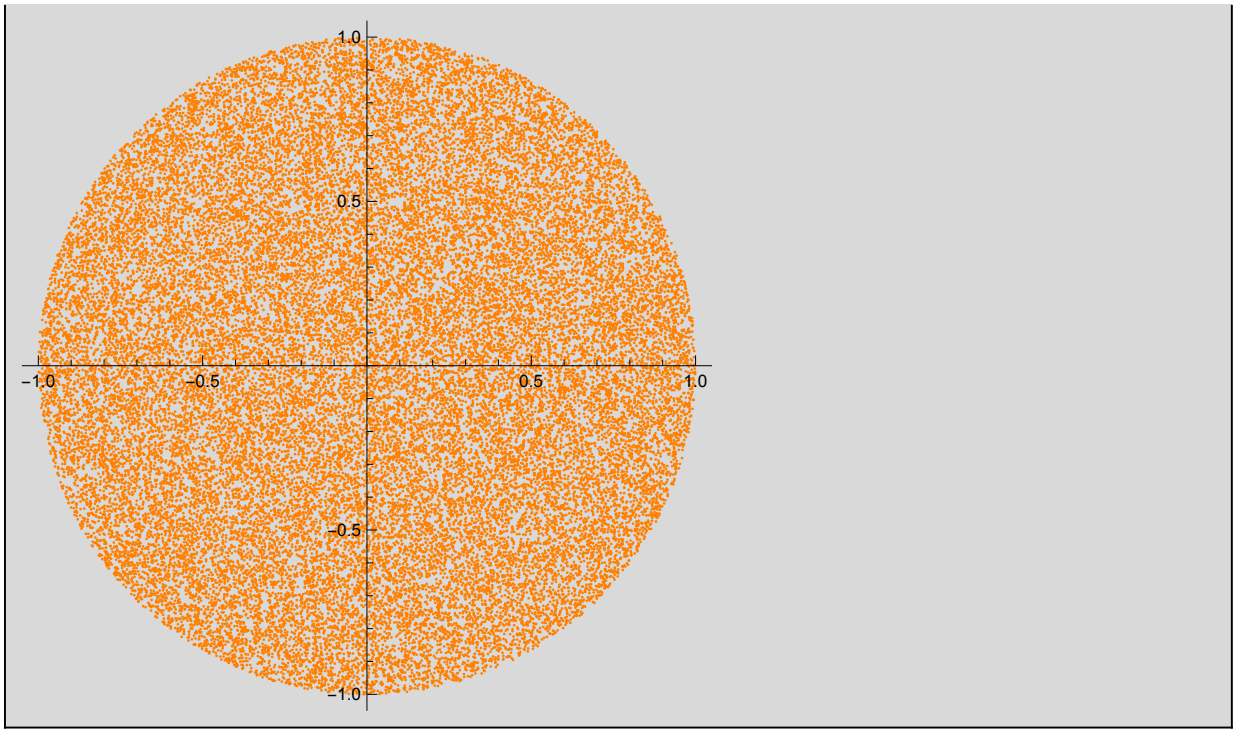
sampleDBall

In[15]:=
```
ClearAll[sampleDBall];
sampleDBall[d_, n_] :=
  With[{
    pointsOnSphere =
     Normalize /@
      RandomVariate[
       MultinormalDistribution[IdentityMatrix[d]], n],
    scalar =
     Power[
      RandomVariate[UniformDistribution[], n],
      1 / d]},
   MapThread[Times, {pointsOnSphere, scalar}]];
```

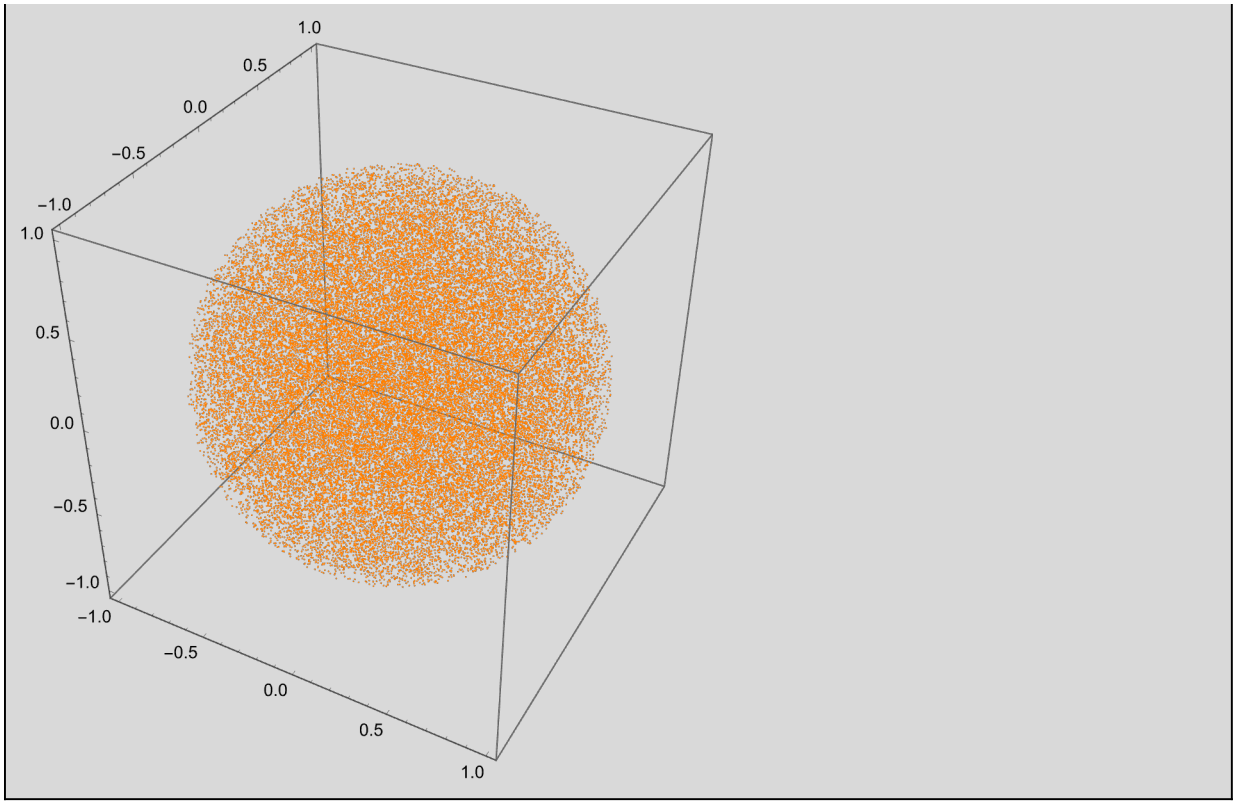
The following plots illustrate points uniformly distributed in radius in two and three dimensions, and thus uniformly distributed in the 2-ball and in the 3-ball, respectively:

In[17]:=
```
show2D$[sampleDBall[2, 40 000]]
```

Out[17]=



In[18]:=
```
show3D$[sampleDBall[3, 40 000]]
```
Out[18]=



UNIFORMITY STATISTICS

We exhibit two ways to collect statistics on a sample: *by shell* and *by cubie*. From the statistics, we can calculate average point density (per unit *d*-volume) and check that it is uniform. The shell method counts points that lie within *d*-shells of constant volume. The cubie method counts points in elementary *d*-cubes of constant volume. The shell method is robust and practical because it is easy to construct shells of accurately and finitely constant volume. The cubie method is obvious and intuitive, but inaccurate and impractical for dimensions more than three. Accurate statistics require a large number of small cubies, but practical statistics requires a small number of large cubies. Large cubies cross the boundary of a ball too often — a discretization artifact. Cubies that cross the boundaries of the ball have deficient counts: below the mean. The distribution of cubie counts quickly becomes skewed to the small tail as *d* increases. Skewed statistics don't fit a uniform distribution — a Gaussian — and fail a Chi-squared test for no decent reason. The rest of this section demonstrates.

BY SHELL

The volume of a *d*-ball of radius ρ is proportional to ρ^d. Incrementing ρ^d by integer multiples of a constant Δv produces a sequence of shells of equal volumes (constant Δv is proportional to constant Δ-

volume). Thus, we may index shells by integer multiples of Δv: shell number Ceiling$[\rho^d, \Delta v]$ contains the point p with Euclidean norm ρ.

In a large sample, we expect the mean number of vectors with endpoints in each shell to be drawn from the same, uniform distribution with constant mean μ and standard deviation $\sqrt{\mu}$. The Chi-squared test for a uniform distribution demands that the standard deviation of endpoint counts in each shell be approximately equal to $\sqrt{\mu}$. That observation gives us a pragmatic, "eyeball" test for uniformity. If μ is not constant or if the standard deviation is unreasonably far from $\sqrt{\mu}$, we likely have a non-uniform source distribution.

The following are numerical statistics across 100 shells of equal volume for 1 000 000 spherically uniform randoms in each of two, three, six, sixteen, and 276 dimensions. The expected mean number of points in each shell is $10\,000 = 1\,000\,000/100$, and the expected standard deviations of the counts is $\sqrt{10\,000} = 100$. The empirical standard deviations agree reasonably well with the expected.

A rigorous Chi-squared test would give the probability that each shell is not drawn from the same uniform distribution, but the agreement between empirical and expected standard deviations is good enough that a rigorous Chi-squared test does not seem necessary.

SAMPLED SHELL COUNTS

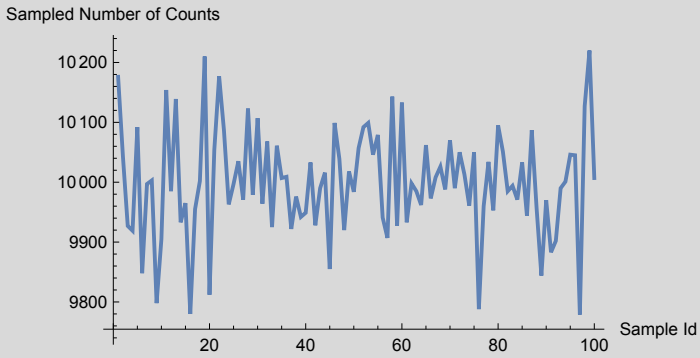
In[19]:=

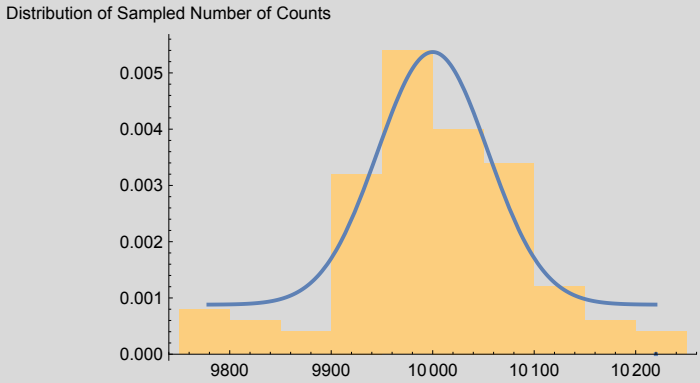
```
ClearAll[pairwise];
pairwise[fn_, list_List] :=
  MapThread[fn, {Drop[list, -1], Rest[list]}];

ClearAll[pointsInShell2];
pointsInShell2[points_, d_, dr_] :=
  GroupBy[points, p ↦ Ceiling[Norm[p]^d, dr]];

ClearAll[sampledShellCounts2];
sampledShellCounts2[deltaR_ : 0.01, d_ : 2, n_ : 4000, sampler_ : sampleDBall] :=
  With[{points = sampler[d, n]},
   With[{countsAssoc = Length /@ pointsInShell2[points, d, deltaR]},
    Values[countsAssoc]]];

ClearAll[countsStats];
countsStats[counts_] :=
  With[{𝒟 = counts},
   With[{
     tot = Plus @@ 𝒟,
     μ = Mean@𝒟,
     σ = N@StandardDeviation@𝒟,
     min = Min[𝒟],
     max = Max[𝒟],
     P = Quiet@FindDistribution[𝒟]},
    Column@{ListLinePlot[𝒟, PlotRange → Full, ImageSize → Medium,
       AxesLabel → {"Sample Id", "Sampled Number of Counts"}],
      Show[
       Histogram[𝒟, Automatic, "PDF", PlotRange → Full,
        AxesLabel → "Distribution of Sampled Number of Counts"],
       Plot[PDF[P, c], {c, min, max}], ImageSize → Medium],
      ⟨|"actual == expected μ" → N@μ,
       "expected stddev" → N@Sqrt[μ], "n samples" → Length[counts],
       "actual stddev" → σ, "total" → tot, "min" → min, "max" → max|⟩}]];
```

TWO DIMENSIONS

In[27]:=
```
countsStats[sampledShellCounts2[0.01, 2, 1 000 000]]
```

Out[27]=



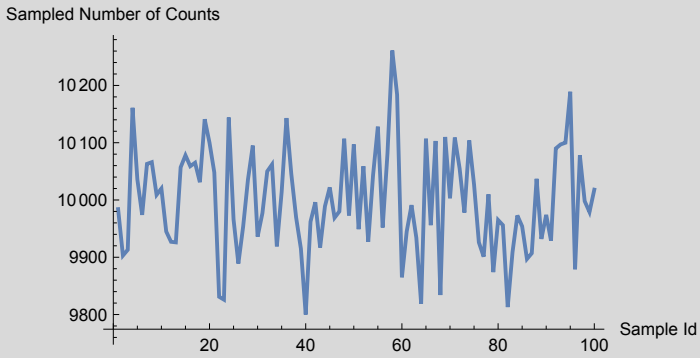


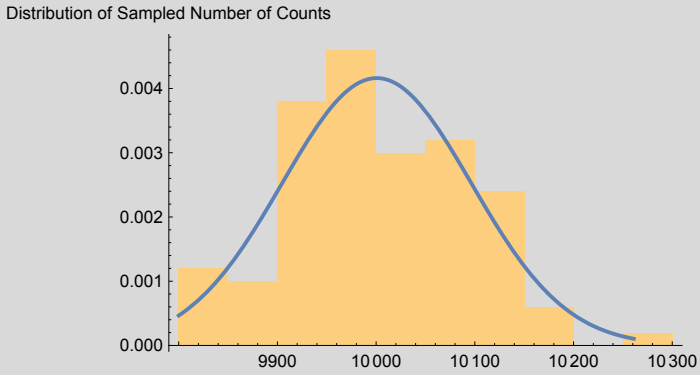

```
⟨|actual == expected μ → 10 000., expected stddev → 100., n samples → 100,
 actual stddev → 90.0325, total → 1 000 000, min → 9779, max → 10 220|⟩
```

THREE DIMENSIONS

In[28]:=
```
countsStats[sampledShellCounts2[0.01, 3, 1 000 000]]
```

Out[28]=



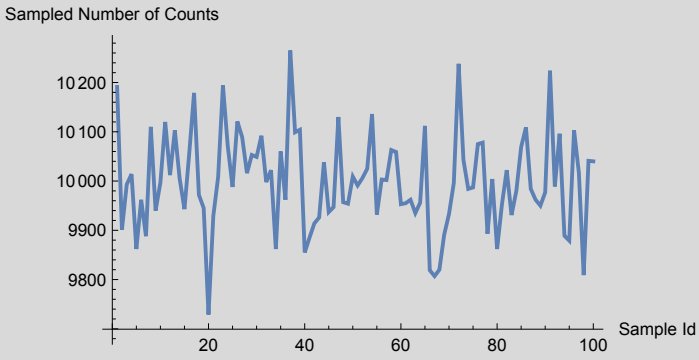


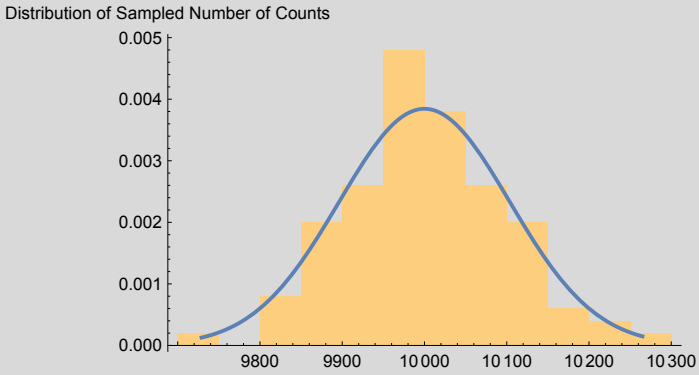
⟨|actual == expected μ → 10 000., expected stddev → 100., n samples → 100,
actual stddev → 91.5347, total → 1 000 000, min → 9800, max → 10 261|⟩

SIX DIMENSIONS

In[29]:=
```
countsStats[sampledShellCounts2[0.01, 6, 1 000 000]]
```

Out[29]=



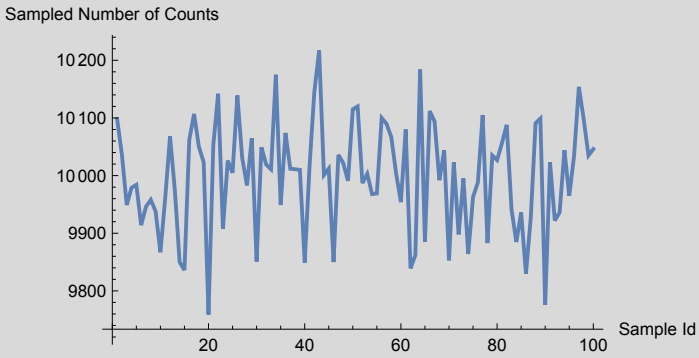


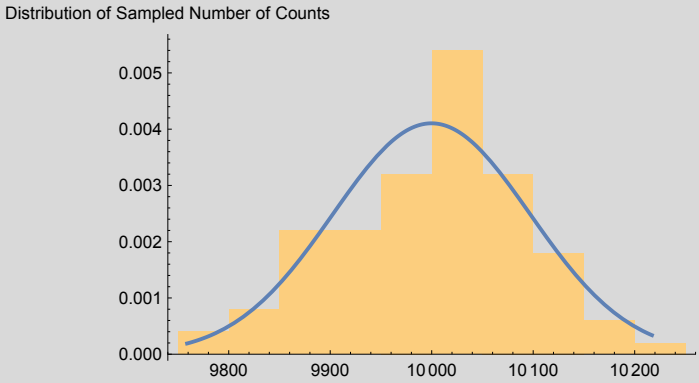
⟨|actual == expected μ → 10 000., expected stddev → 100., n samples → 100,
actual stddev → 98.1884, total → 1 000 000, min → 9729, max → 10 265|⟩

SIXTEEN DIMENSIONS

In[30]:=
```
countsStats[sampledShellCounts2[0.01, 16, 1 000 000]]
```

Out[30]=



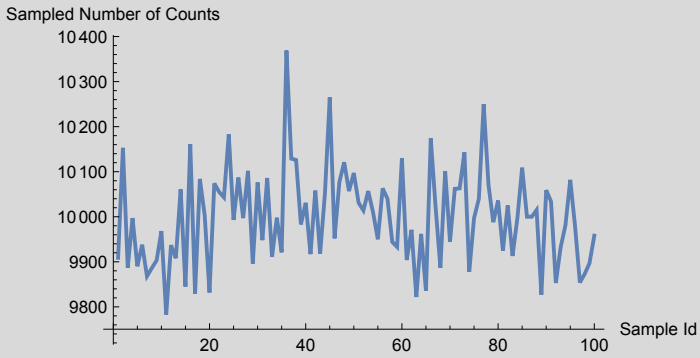


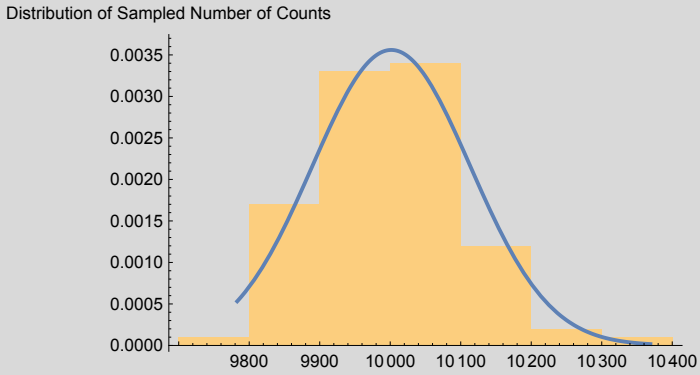

```
⟨|actual == expected μ → 10 000., expected stddev → 100., n samples → 100,
 actual stddev → 93.1243, total → 1 000 000, min → 9759, max → 10 217|⟩
```

276 DIMENSIONS

In[31]:=
```
countsStats[sampledShellCounts2[0.01, 276, 1 000 000]]
```

Out[31]=






```
⟨|actual == expected μ → 10 000., expected stddev → 100., n samples → 100,
 actual stddev → 105.353, total → 1 000 000, min → 9783, max → 10 369|⟩
```

BY CUBIE

Statistics by cubie ignore geometry and count points in an evenly-spaced Cartesian grid of cubies. Statistics by shell are much faster and more informative because they honor geometry. Statistics by cubie do not scale to large *d*, where almost all the volume contains empty cubies. Accurate statistics require an astronomically impractical number of small cubies. However, statistics by cubie are instructive for small *d*, illustrating indexing by hashing cubie coordinates in a Mathematica Association, similar to a Python dictionary or to a Clojure hashmap.

Let's do some cubie statistics. First, find the cubie boundaries near a point *x*. Boundary tuples are the keys of the hash table. Every point within a certain cubie hashes to the same key.

In[32]:=
```
ClearAll[cellBoundaryTuple];
cellBoundaryTuple[x_, δ_] :=
  With[{f = Floor[x, δ], c = Ceiling[x, δ]},
   {f, c}];
```

Make a function of an existing hash table and a point, closed over a given cubie size δ. Locate the cubie of the point in the hash table and increment the cubie's counts.

In[34]:=
```
ClearAll[makeCubifer];
makeCubifer[δ_] :=
  Function[{dict, point},
   Module[{temp = dict},
    With[{key = cellBoundaryTuple[point, δ]},
     With[{val = temp[key]},
      With[{test = Head[val]},
       If[Missing === test,
        AssociateTo[temp, key → 1],
        temp[key] ++]]]];
    temp]];
```

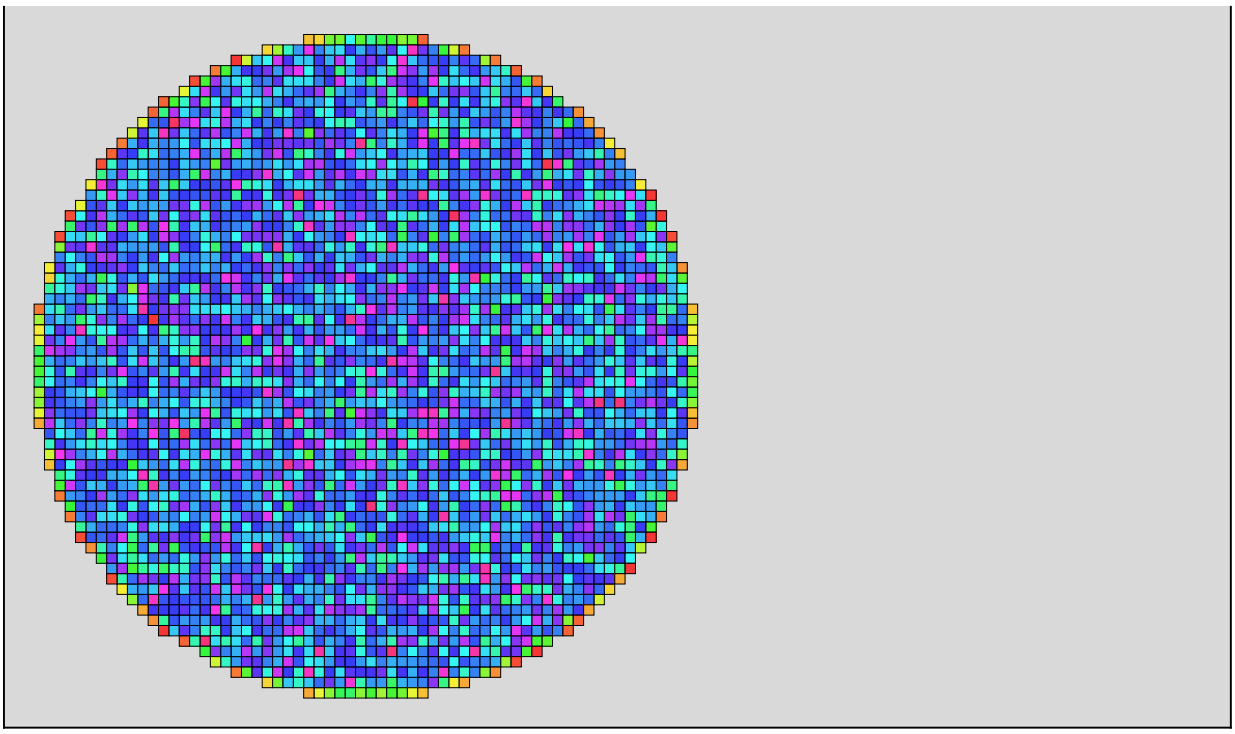
Visualize cubie densities — red for small counts, indigo for large counts.

In[36]:=
```
ClearAll[cubiePolygon];
Module[{},
  cubiePolygon[{xyLeft_, xyRight_}, value_, maxVal_, minVal_] :=
   With[{xLeft = xyLeft[[1]], xRight = xyRight[[1]],
     yBottom = xyLeft[[2]], yTop = xyRight[[2]]},
    {Hue[(value - minVal)/(maxVal - minVal)], Polygon[{{xLeft, yBottom}, {xRight, yBottom},
       {xRight, yTop}, {xLeft, yTop}}]}]];
ClearAll[illustrateCubifers];
illustrateCubifers[δR_ : 0.1, d_ : 2, n_ : 4000] :=
 With[{sample = sampleDBall[d, n]},
  With[{kvs = Fold[makeCubifer[δR], <||>, sample]},
   With[{keys = Keys[kvs], values = Values[kvs]},
    With[{cubieArea = Times @@ (keys[[1, 2]] - keys[[1, 1]])},
     With[{density = values / cubieArea},
      Print[<|"values" → Short[values], "sum" → Plus @@ values,
        "number of cubies" → Length[keys], "cubie area" → cubieArea,
        "point densities per cubie" → Short[density],
        "avg point density" → Mean[density],
        "stddev point density" → StandardDeviation[density]|>];
      With[{vx = Max[values], vn = Min[values], len = Length[values]},
       MapThread[cubiePolygon, {keys, values,
         ConstantArray[vx, len],
         ConstantArray[vn, len]}]]]]]]]
```

In[40]:=
```
Graphics[
 {Opacity[0.75], EdgeForm[Black], illustrateCubifers[0.001^(1/2), 2, 100 000]}]
```

```
<|values → {25, 28, 24, 33, 31, 42, 24, 32, <<3232>>, 1, 1, 2, 1, 3, 1, 1},
 sum → 100 000, number of cubies → 3247, cubie area → 0.001,
 point densities per cubie → {25 000., <<3245>>, 1000.},
 avg point density → 30 797.7, stddev point density → 7177.66|>
```

Out[40]=



In[41]:=
```
ClearAll[sampledCubieCounts];
sampledCubieCounts[δR_ : 0.1, d_ : 2, n_ : 4000] :=
  Module[{volumeRatio = N[vBall[d] / 2^d]},
   Module[{pointDensityInTheBall = N[n / vBall[d]]},
    Print[<|"Total number of cubies, δR^-d" → δR^-d,
      "d-ball volume / d-cube volume" → volumeRatio,
      "number of cubies in the ball" → δR^-d volumeRatio,
      "Theoretical point density in the ball" → pointDensityInTheBall,
      "Theoretical overall mean points per cubie" →
       pointDensityInTheBall / δR^-d|>];
    Module[{cubieHash = Fold[makeCubifer[δR], <||>, sampleDBall[d, n]]},
     Print[<|"Number of hash keys" → Length@Keys[cubieHash],
       "sum of counts in all cubies" → Plus @@ Values[cubieHash]|>];
     Values[cubieHash]]];
```

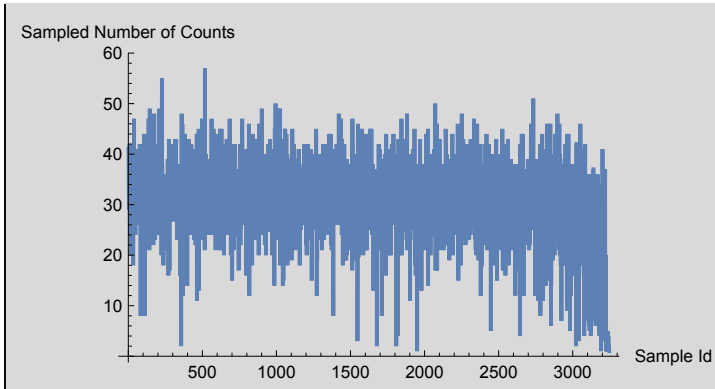
In the following demonstrations, we consider $n = 100\,000$ vector endpoints, restricting the sizes of cubies to the d-th root of 0.001 so that the total number of cubies is 1000; any more and the simulations are uncomfortably slow. The number of hash keys is much larger than the real-number fraction of cubies inside the ball because hash keys include cubies with deficient counts — cubies crossing the boundary. The proportion of cubies with deficient counts grows quickly with d because the relative size of cubies to the size of the ball grows quickly with d. This is a discretization effect. Also notice that Mathematica cannot find a smooth fit to the statistics even for small d. Unlike shell statistics, the plots show no smooth curve because Mathematica's *FindDistribution* fails. That's why we wrap it in *Quiet* in the code for *countsStats*.

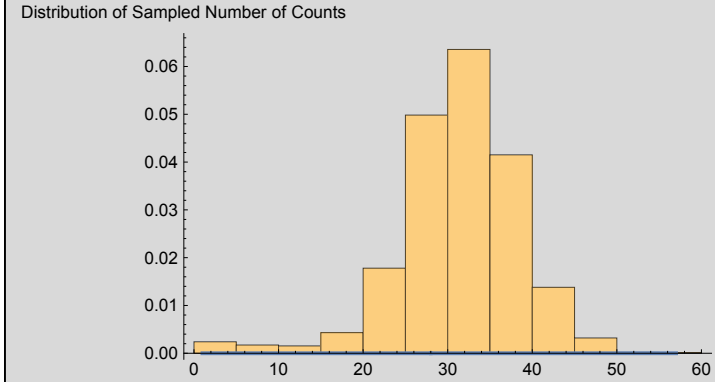
TWO DIMENSIONS

In[43]:=
```
countsStats[sampledCubieCounts[0.001^(1/2), 2, 100 000]]
```

```
⟨|Total number of cubies, δR^-d → 1000., d-ball volume / d-cube volume → 0.785398,
 number of cubies in the ball → 785.398, Theoretical point density in the ball → 31 831.,
 Theoretical overall mean points per cubie → 31.831|⟩

⟨|Number of hash keys → 3247, sum of counts in all cubies → 100 000|⟩
```

Out[43]=





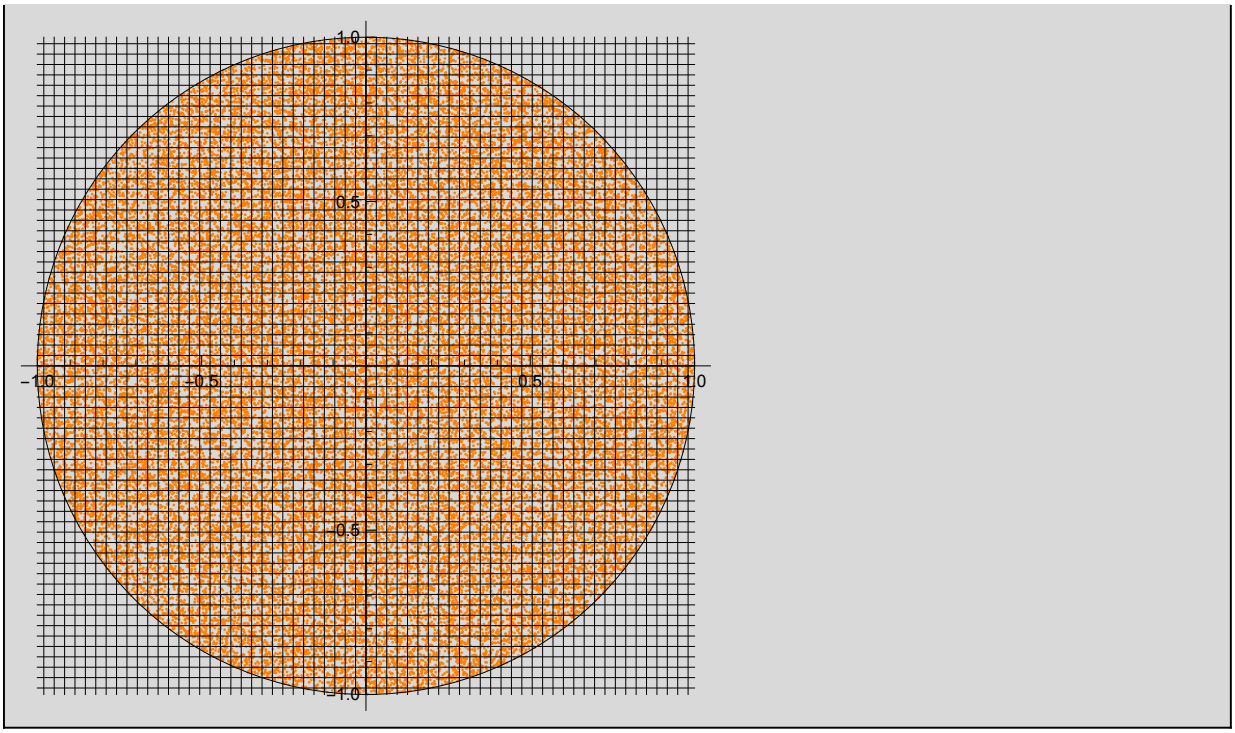

```
⟨|actual == expected μ → 30.7977, expected stddev → 5.54956,
 n samples → 3247, actual stddev → 7.23426, total → 100 000, min → 1, max → 57|⟩
```

There are already, even in two dimensions, a noticeable number of deficiently populated cubies (the tail to the left of the peak above). These are the cubies crossing the circle and in the corners outside the

circle.

In[44]:=
```
Show[show2D$[sampleDBall[2, 40 000]], Graphics[Circle[]],
 Graphics@Line[{{#, -1}, {#, 1}}] & /@ Range[0, 1, 0.001^(1/2)],
 Graphics@Line[{{-1, #}, {1, #}}] & /@ Range[0, 1, 0.001^(1/2)],
 Graphics@Line[{{#, -1}, {#, 1}}] & /@ Range[0, -1, -0.001^(1/2)],
 Graphics@Line[{{-1, #}, {1, #}}] & /@ Range[0, -1, -0.001^(1/2)]]
```

Out[44]=

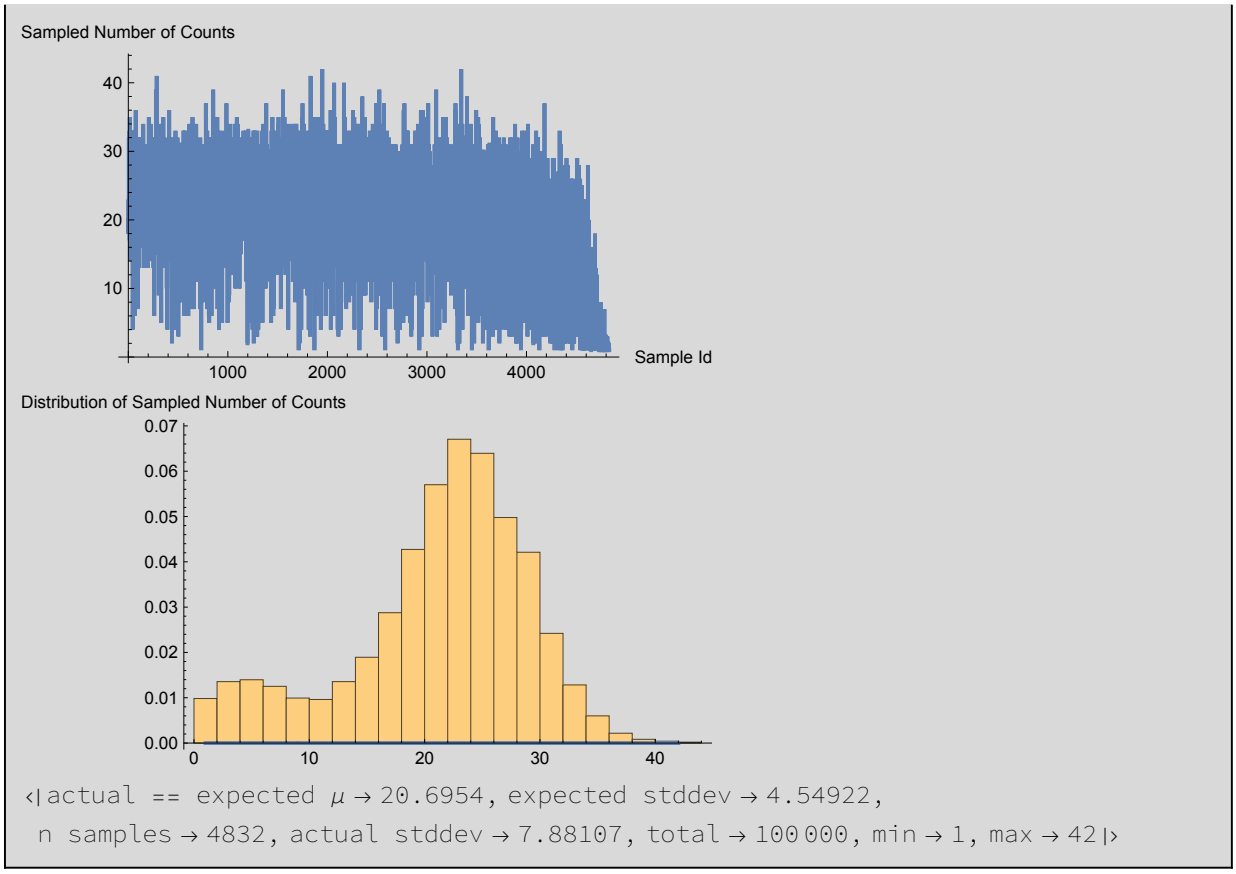


THREE DIMENSIONS

In[45]:=
```
countsStats[sampledCubieCounts[0.001^(1/3), 3, 100 000]]
```

```
⟨|Total number of cubies, δR^-d → 1000., d-ball volume / d-cube volume → 0.523599,
 number of cubies in the ball → 523.599, Theoretical point density in the ball → 23 873.2,
 Theoretical overall mean points per cubie → 23.8732|⟩

⟨|Number of hash keys → 4832, sum of counts in all cubies → 100 000|⟩
```

Out[45]=




```
⟨|actual == expected μ → 20.6954, expected stddev → 4.54922,
 n samples → 4832, actual stddev → 7.88107, total → 100 000, min → 1, max → 42|⟩
```

In three dimensions, the number of deficient cubies is larger than “noticeable.”

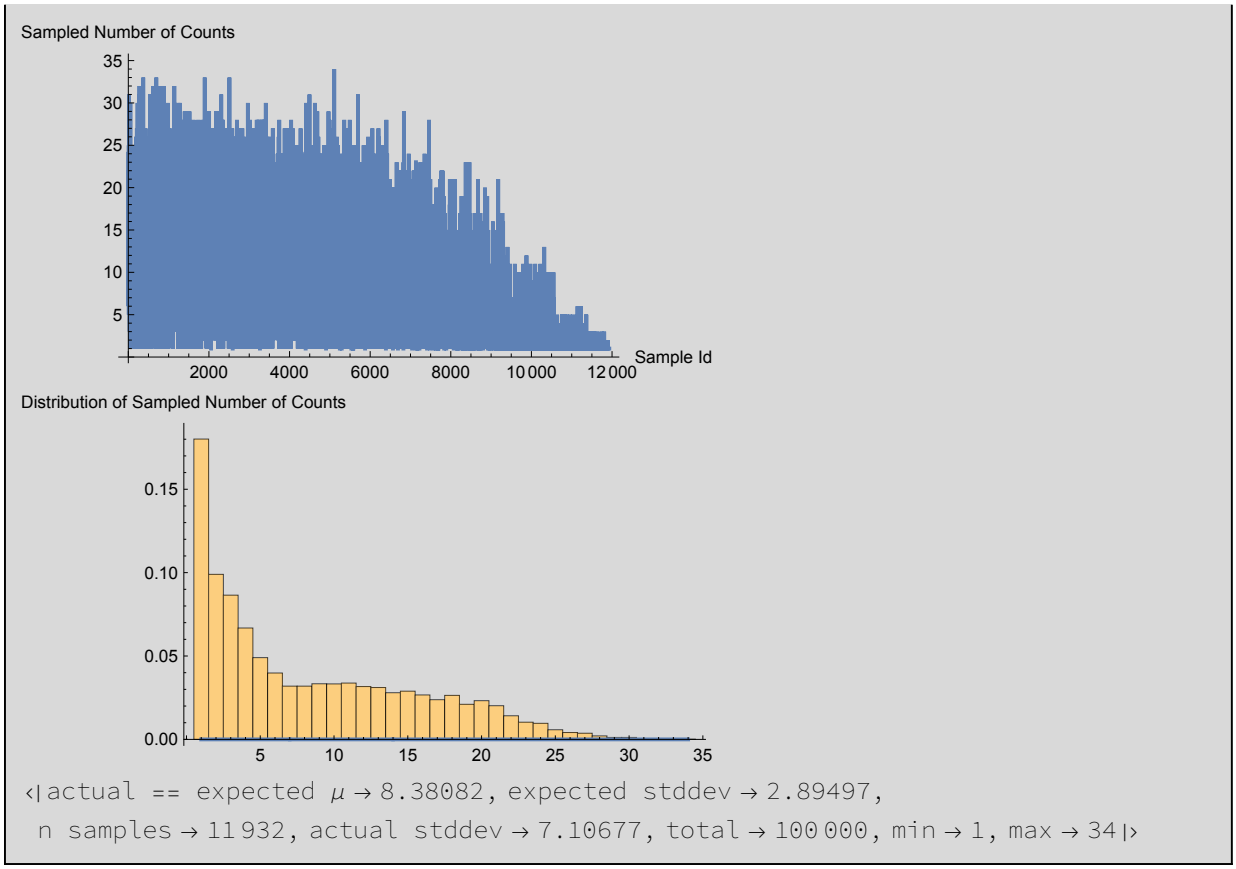
SIX DIMENSIONS

In[46]:=

```
countsStats[sampledCubieCounts[0.001^(1/6), 6, 100 000]]
```

```
⟨|Total number of cubies, δR^-d → 1000., d-ball volume / d-cube volume → 0.0807455,
 number of cubies in the ball → 80.7455, Theoretical point density in the ball → 19 350.9,
 Theoretical overall mean points per cubie → 19.3509|⟩

⟨|Number of hash keys → 11 932, sum of counts in all cubies → 100 000|⟩
```

Out[46]=




```
⟨|actual == expected μ → 8.38082, expected stddev → 2.89497,
 n samples → 11 932, actual stddev → 7.10677, total → 100 000, min → 1, max → 34|⟩
```

In six dimensions, the deficient cubies dominate the statistics.

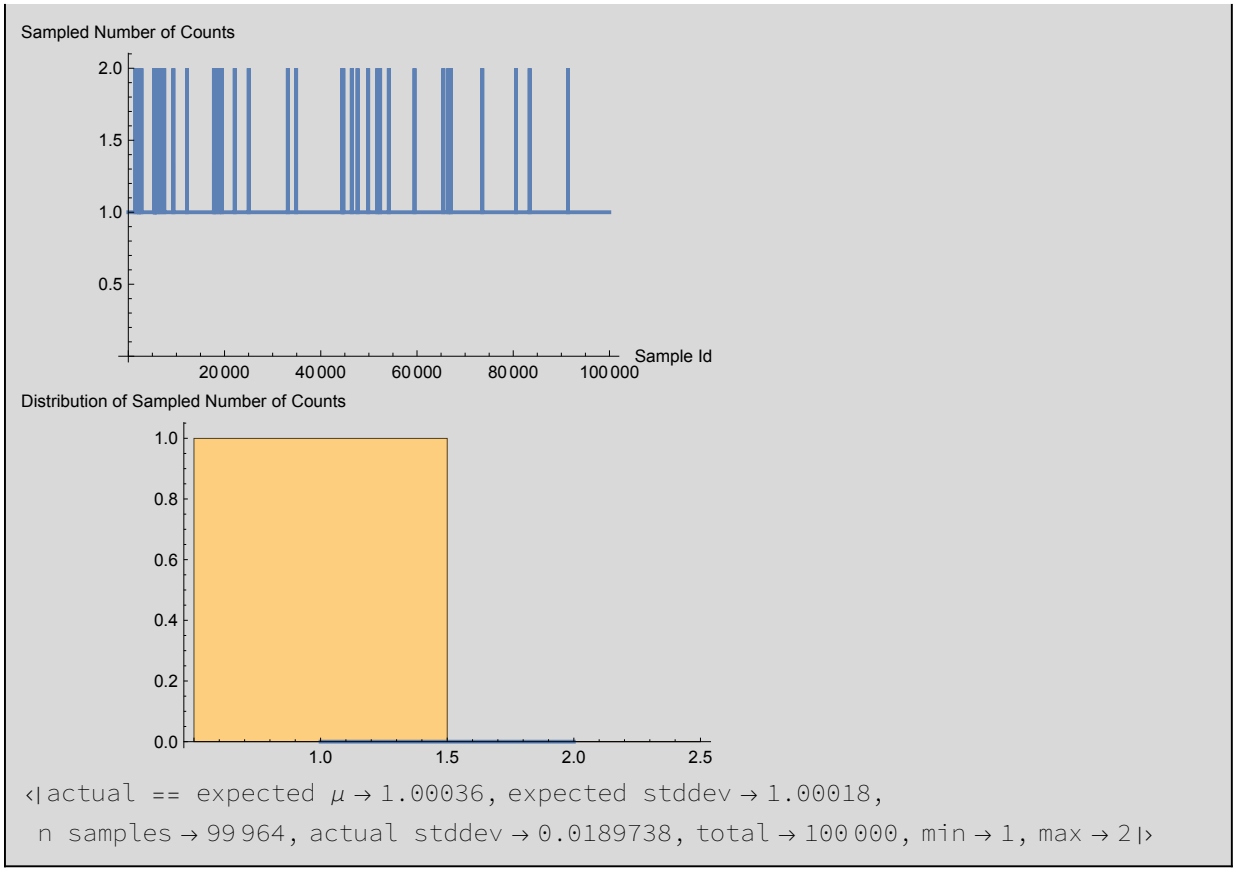
TWENTY-SEVEN DIMENSIONS

In[47]:=

```
countsStats[sampledCubieCounts[0.001^(1/27), 27, 100 000]]
```

```
⟨|Total number of cubies, δR^-d → 1000., d-ball volume / d-cube volume → 1.66053 × 10^-12,
 number of cubies in the ball → 1.66053 × 10^-9,
 Theoretical point density in the ball → 4.48688 × 10^8,
 Theoretical overall mean points per cubie → 448 688.|⟩

⟨|Number of hash keys → 99 964, sum of counts in all cubies → 100 000|⟩
```

Out[47]=




```
⟨|actual == expected μ → 1.00036, expected stddev → 1.00018,
 n samples → 99 964, actual stddev → 0.0189738, total → 100 000, min → 1, max → 2|⟩
```

In twenty-seven dimensions, virtually all the cubies are deficient.

There is no point in examining larger dimensions. To find cubies with populations greater than 1 (statistically), we must reduce the sizes of cubies, and thus increase their number and the computation time exponentially with d. It is not worth the computation time. We already have the shell method for verifying uniformity.

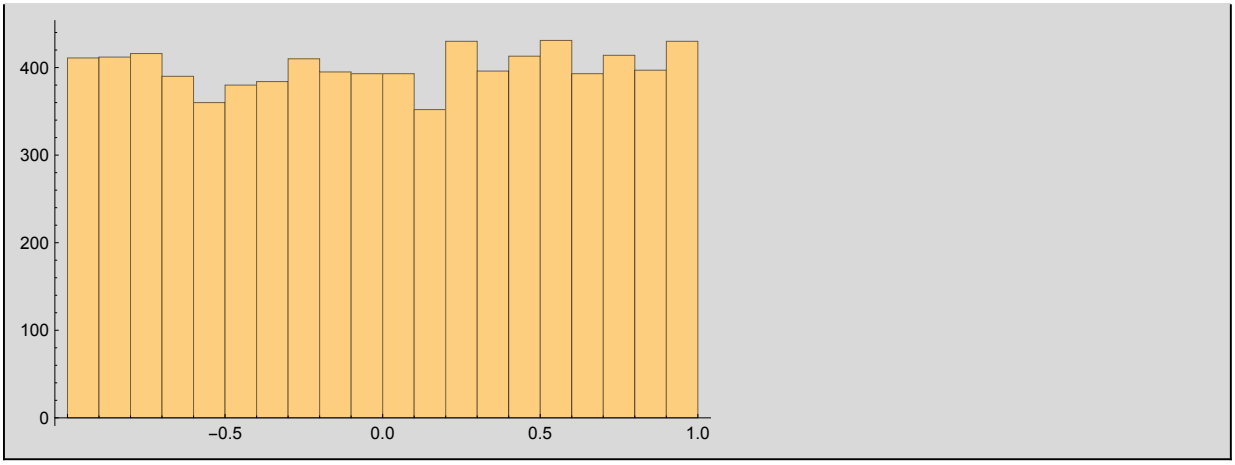
STATISTICAL CONSTRUCTION

Alternatively to the geometrical constructions above, sample uniformly on the $(d+2)$-sphere (not in the $(d+2)$-ball), then delete any two coordinates of each point, deterministically or at random. We already know how to sample uniformly on the $(d+2)$-sphere: normalized multinormals. The following code illustrates generating uniforms in the 1-ball from uniforms on the 3-sphere by dropping two, randomly chosen coordinates:

In[68]:=
```
ClearAll[sampleDBall2];
sampleDBall2[d_, n_, e_ : 2] :=
  RandomSample[#, e] & /@ (* random rejection *)
   Normalize /@ (* uniform on the d-sphere *)
    RandomVariate[
     MultinormalDistribution[IdentityMatrix[d + e]], n];
```

In[50]:=
```
Histogram[Flatten[sampleDBall2[1, 4000, 2]]]
```

Out[50]=



Why does this work? Why reject two coordinates at random, instead of one or three or seventeen?

First, realize that dropping one coordinate at random is the same as dropping the same coordinate from every point, at least for calculating the distribution of points, because all coordinates are sampled from the same distribution. Henceforth, we always drop the last two coordinates, *z* and *y*.

THEOREM 1

The *d*-volume density of points is constant after projecting out *z* and *y* from a distribution constant per unit (*d* + 2)-hyperspherical area.

PROOF

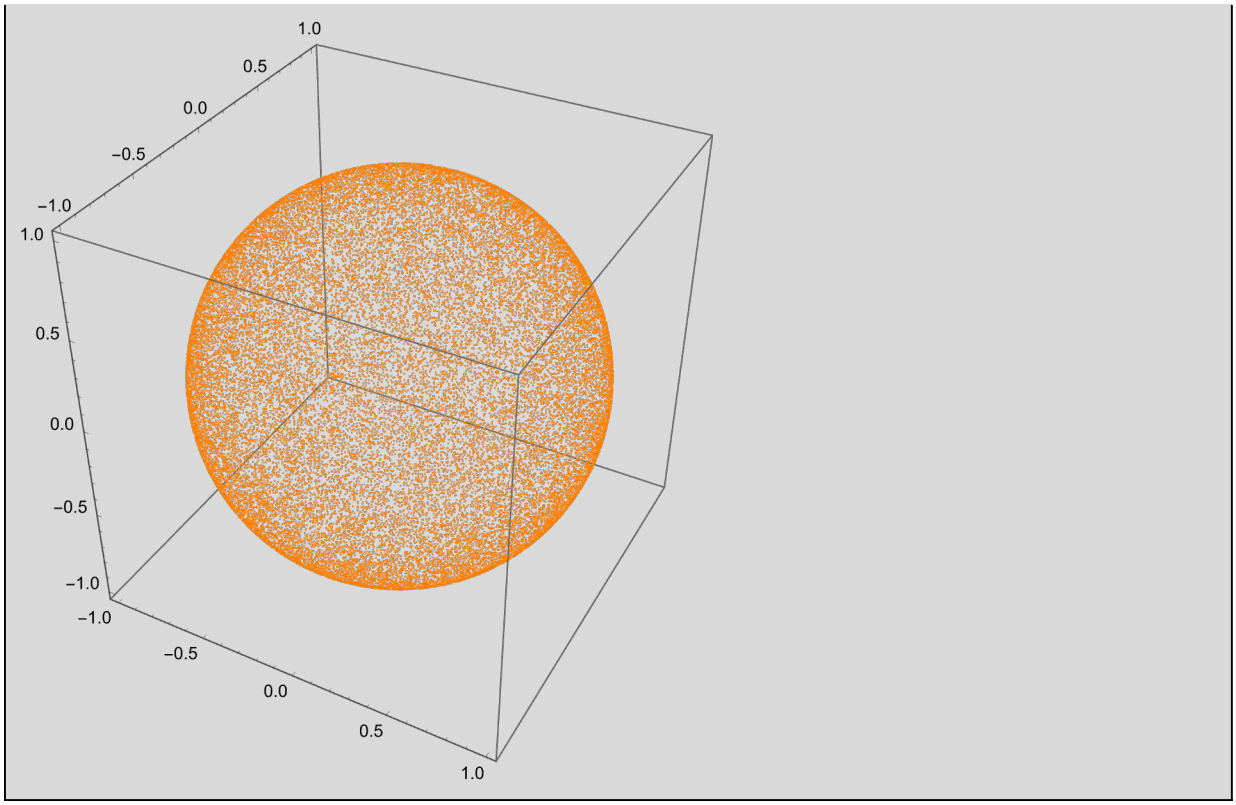
We prove the theorem first for three dimensions, then argue by extension to dimensions greater than three.

Plan: In 3D, produce a 2D distribution of points by dropping *z*. Next, drop the *y* coordinate by integrating it to produce a 1D density. Prove the resulting 1D density of points per unit of the *x* axis is independent of *x*.

As before, generate spherically uniform 3-vectors of unit length — on the 3-sphere — by sampling the 3-dimensional multinormal distribution, then normalizing the vectors:

In[51]:=
```
show3D$[threeDPoints$ =
  Normalize /@
   RandomVariate[
    MultinormalDistribution[IdentityMatrix[3]],
    40 000]]
```

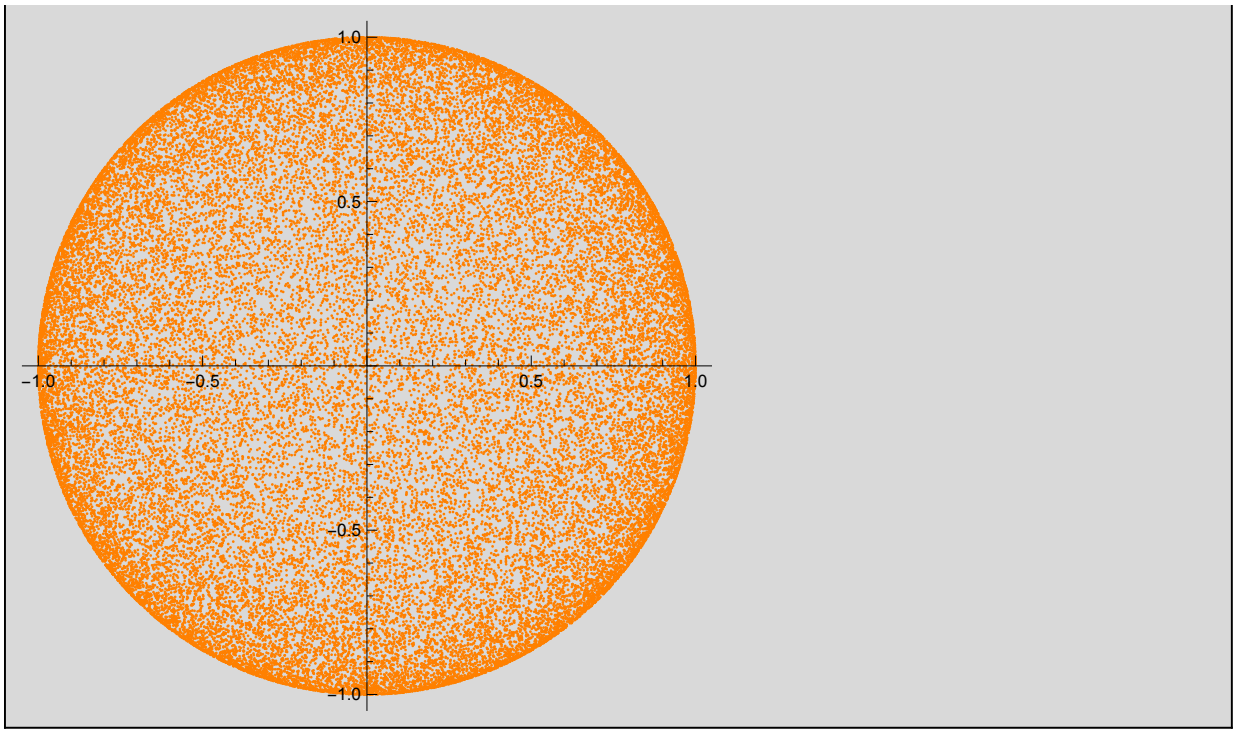
Out[51]=



Drop the *z* (last) coordinate of each point to produce a radially symmetric but non-uniform distribution in the 2-ball:

In[52]:=
```
show2D$[(twoDPoints$ = Drop[#, -1] & /@ threeDPoints$)]
```

Out[52]=



What is the 2D distribution at this step?

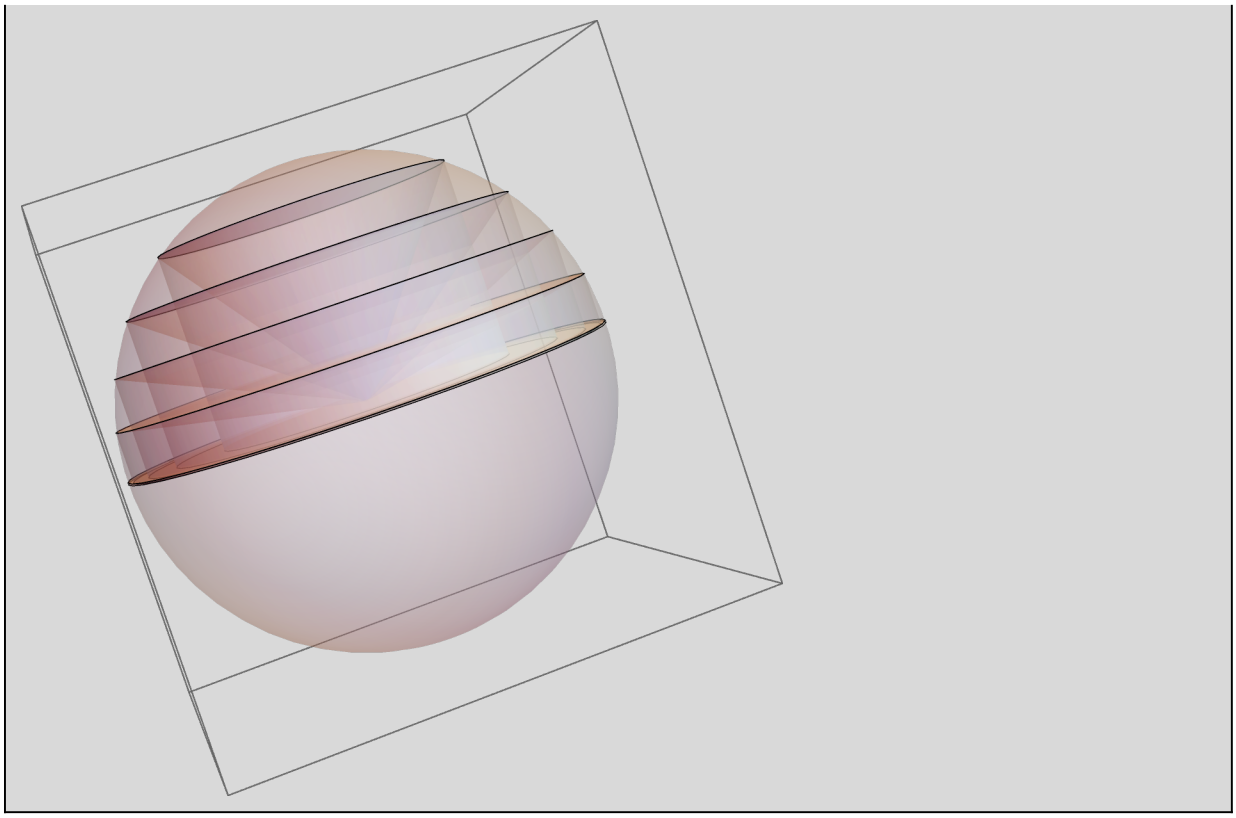
LEMMA 1

Proof of LEMMA 1:

Consider curved belts (or *spherical segments*) of equal *z*-height *h* on the surface of the unit 3-sphere surrounding the *z* axis. Each such belt has equal surface area $2\,\pi\,h$ by Archimedes' Hat-Box Theorem (also see Apostol, T., Mnatsakanian, M. (2004). *A fresh look at the method of Archimedes*. Am. Math. Monthly 111: 496–508. doi.org/10.2307/4145068, for much more.)

The following illustrates five belts in 3D:

Out[53]=



PROOF OF THE HAT-BOX THEOREM

The next diagram shows a longitudinal cross-section of the 3-ball with belts of equal z-height.

A belt of z-height h corresponds to two elevation angles, $\theta_1 < \theta_2$, such that $\sin\theta_2 - \sin\theta_1 = h$.

An differential element of area on the surface, in co-spherical coordinates, (θ (elevation), ϕ (azimuth)), via elevation (co-latitude) angles θ rather than the usual latitude angles, is $\cos\theta\, d\phi\, d\theta$. The area of a belt is

In[54]:=

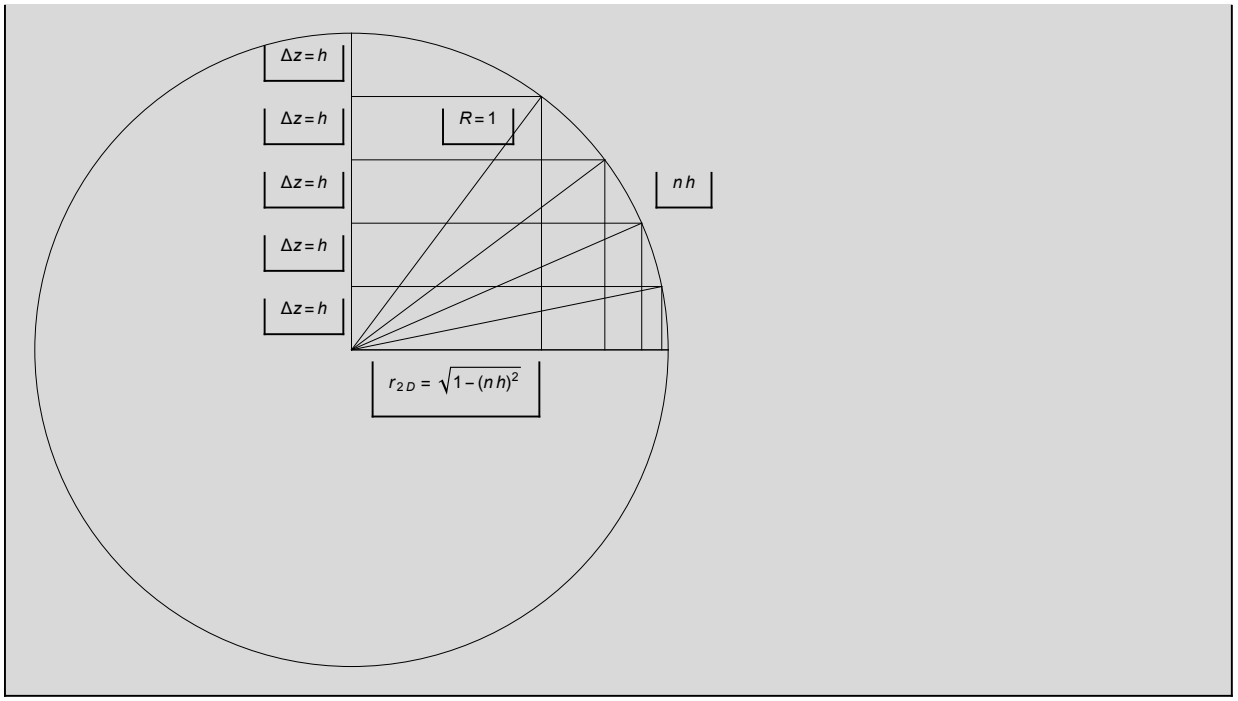
$$\int_0^{2\pi}\int_{\theta_1}^{\theta_2} \text{Cos}[\theta]\, d\theta\, d\phi$$

Out[54]=

$$2\pi\,(-\text{Sin}[\theta_1] + \text{Sin}[\theta_2])$$

which is $2\pi h$.

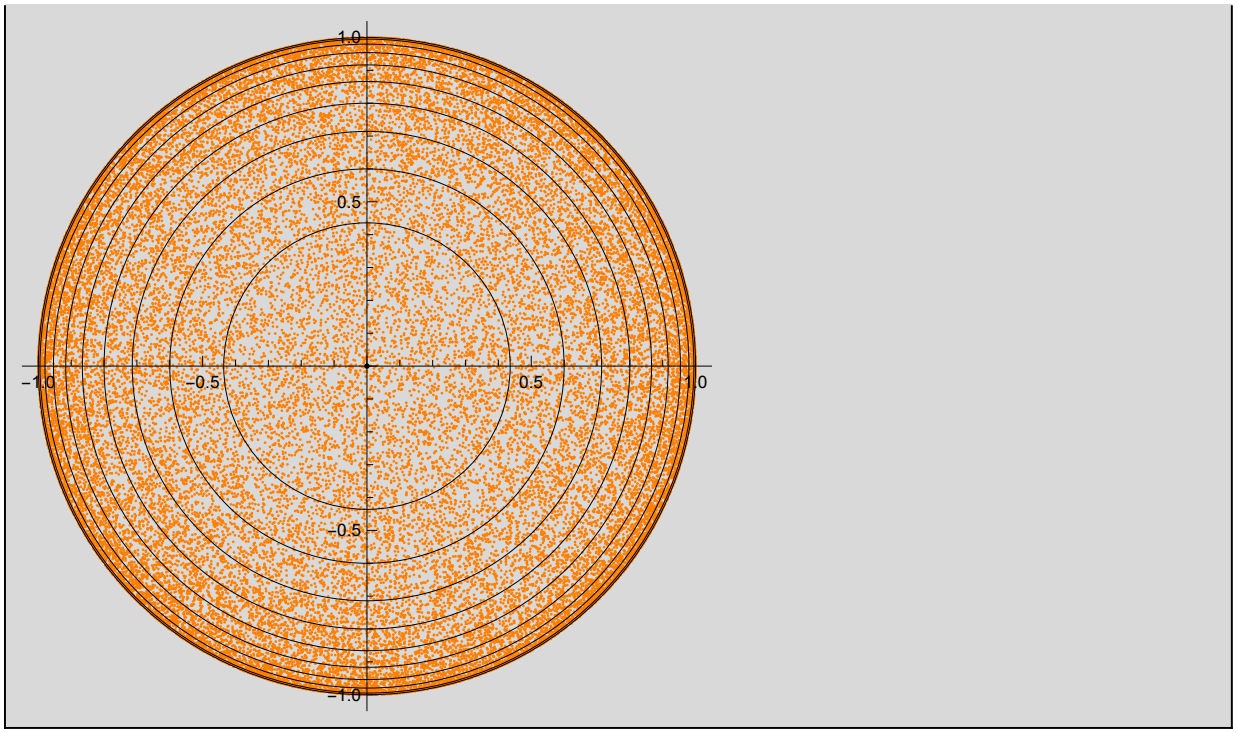
Out[55]=



QED Hat-Box Theorem ■

Back to lemma 1. The projected belts form *annuli*, or rings, in 2D. Consider the following projection of belts of z-height $h = 0.1$ onto the 2D plane, along with the sample points, with their z coordinates dropped:

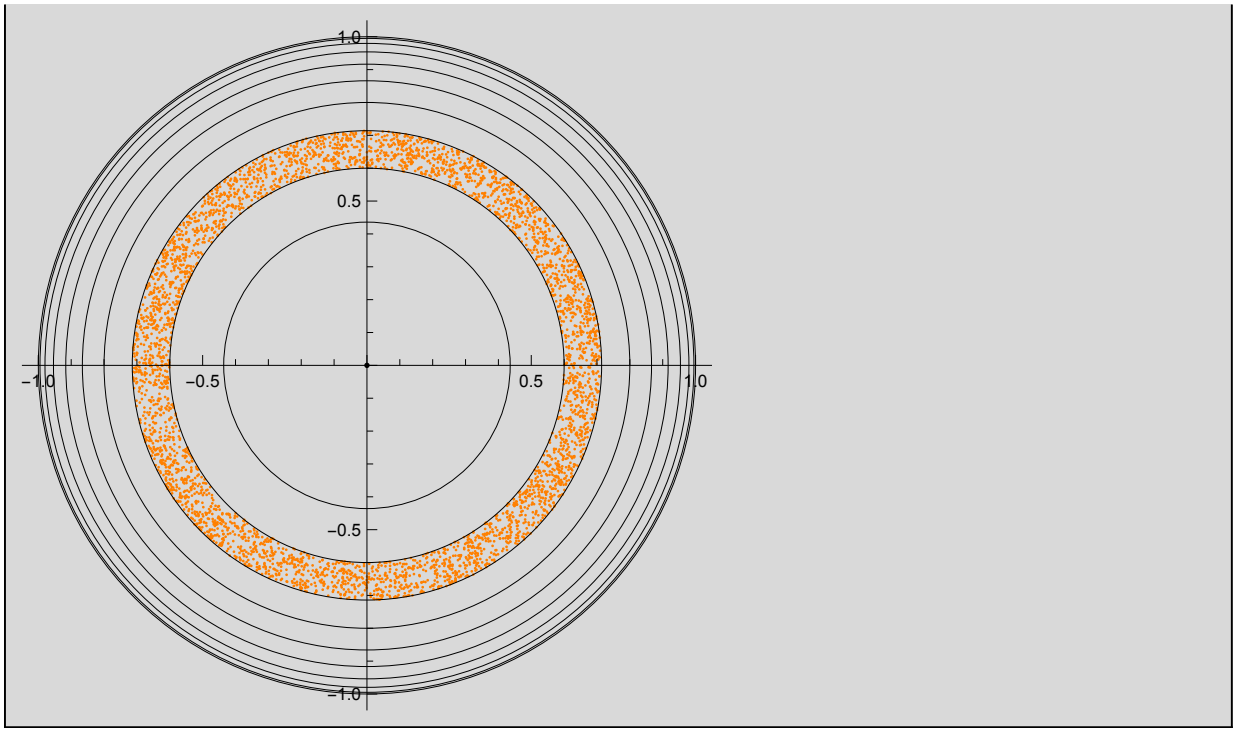
Out[58]=



The following illustrates a grouping function that isolates points in each annulus and shows a selection

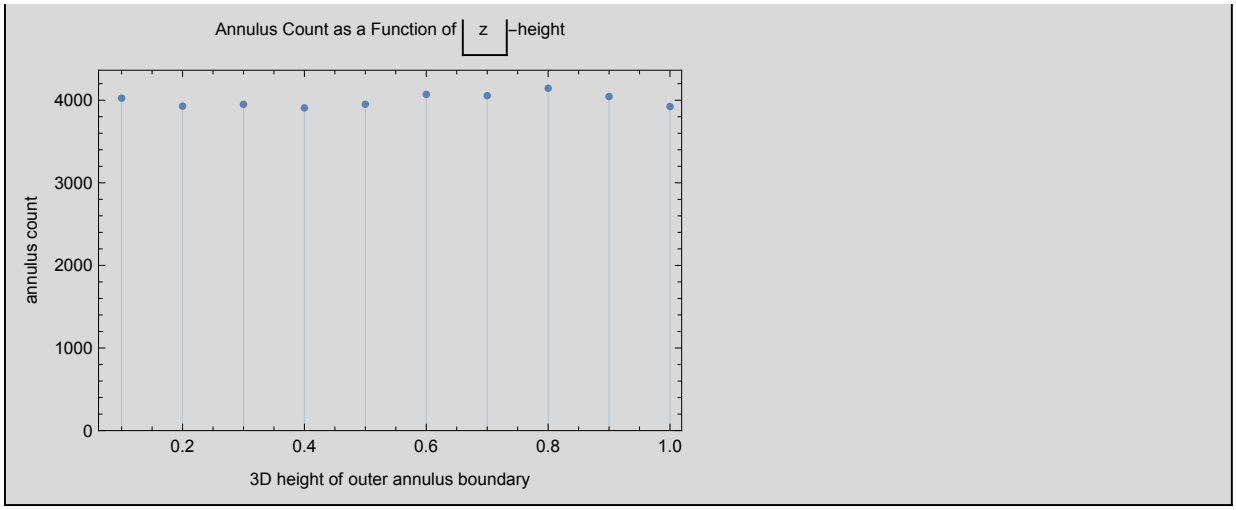
of one such group, the one whose maximum *z*-height is 0.8:

Out[61]=



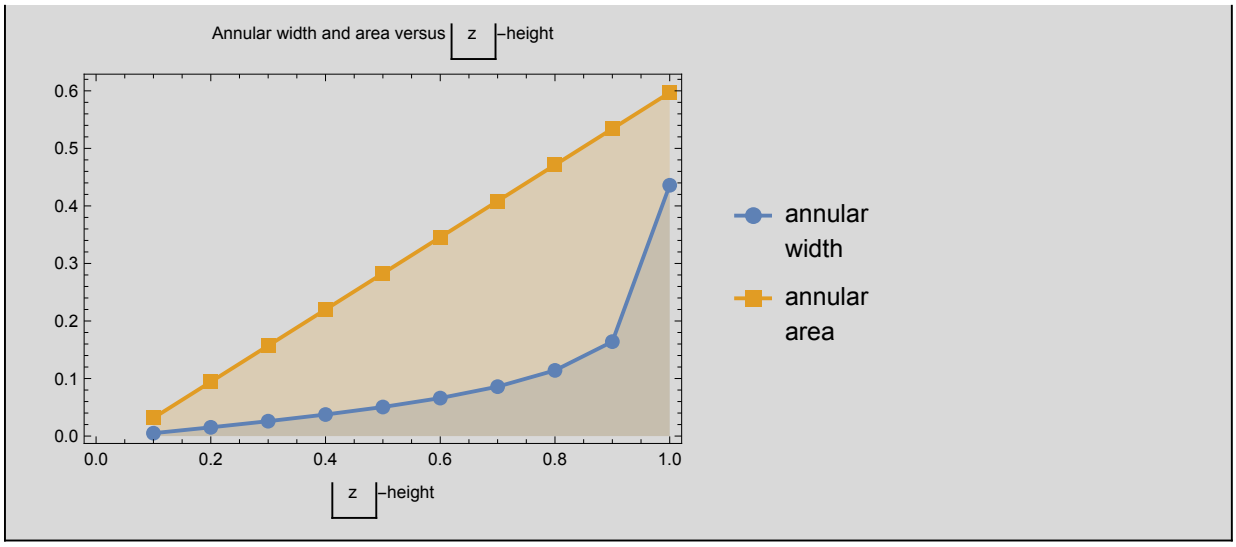
The following illustrates that all annuli have, statistically, the same numbers of points.

Out[62]=



The following illustrates the 2D widths and 2D areas of the annuli as a function of *z*-height:

Out[63]=

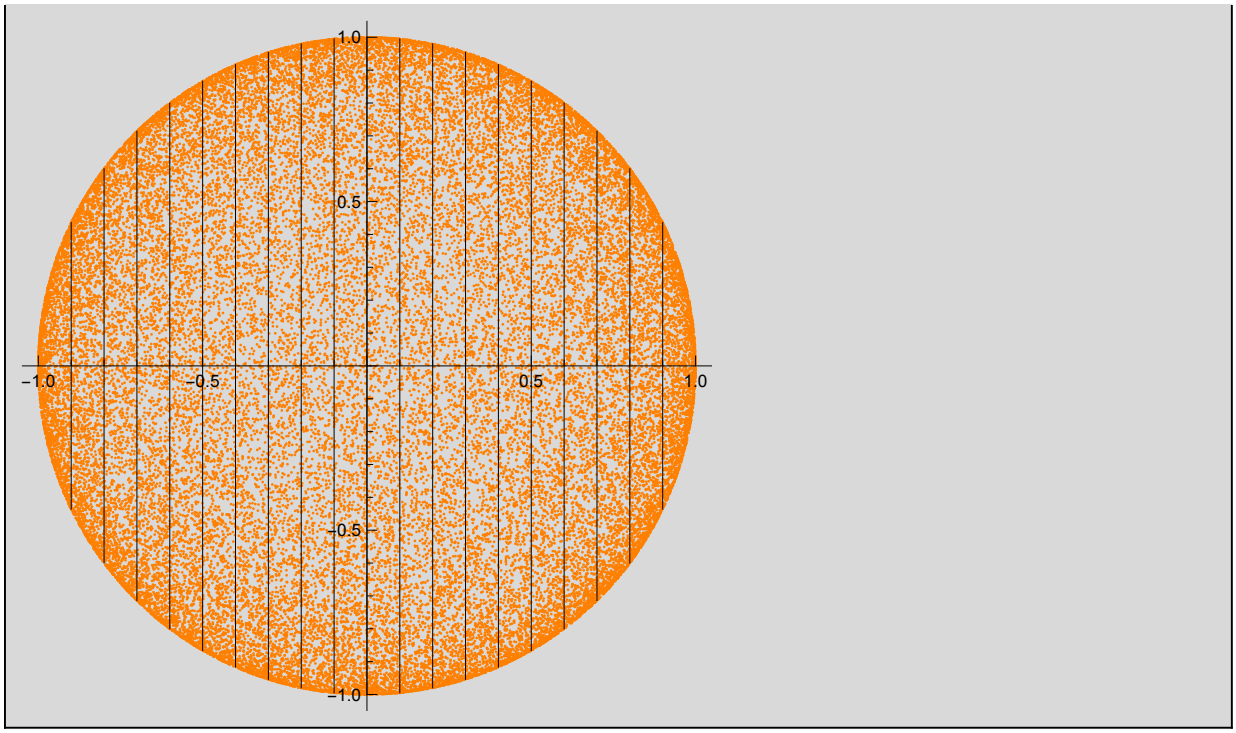


Annular area is linear with *z*-height, and *z*-height varies as $\sqrt{1-r^2}$. Point count per annular area is constant with *z*-height, so 2D point density against 2D *r* varies as $1\Big/\sqrt{1-r^2}$. We use that fact in the integral below.

QED Lemma 1 ■

Slice the circle vertically into bands of equal width Δx and *y*-height $\sqrt{1-x^2}$:

Out[64]=



Eliminate *y* by integrating the density over a particular band at position *x* and invoking lemma 1 for the density $1\Big/\sqrt{1-r^2} = 1\Big/\sqrt{1-y^2-x^2}$:

In[65]:=
$$\int_{-\sqrt{1-x^2}}^{\sqrt{1-x^2}} \frac{1}{\sqrt{1-y^2-x^2}}\, dy$$

Out[65]=
$$\pi \text{ if } -1 < \text{Re}[x] < 1 \&\& \text{Im}[x] == 0$$

independent of x.

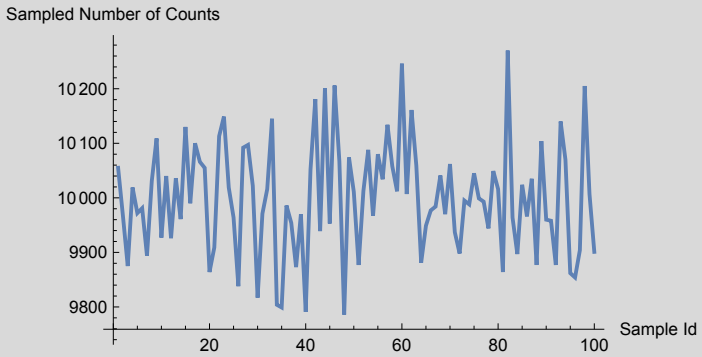
Therefore each band has the same density.

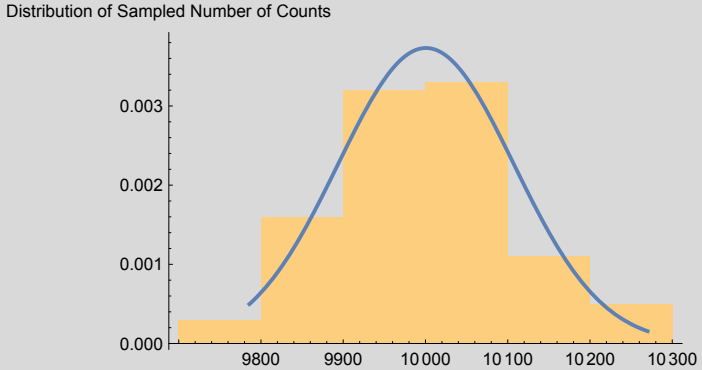
There follow statistics for sampling 1,000,000 points according to this scheme of rejecting two coordinates for two dimensions and for 276 dimensions. As before, with a uniform distribution, we expect the standard deviations of counts to vary as the square root, 100, of the mean of the 10 000 points in each annulus — the “eyeball” Chi-squared test for a uniform distribution.

TWO DIMENSIONS

In[66]:=
```
countsStats[sampledShellCounts2[0.01, 2, 1 000 000, sampleDBall2]]
```

Out[66]=



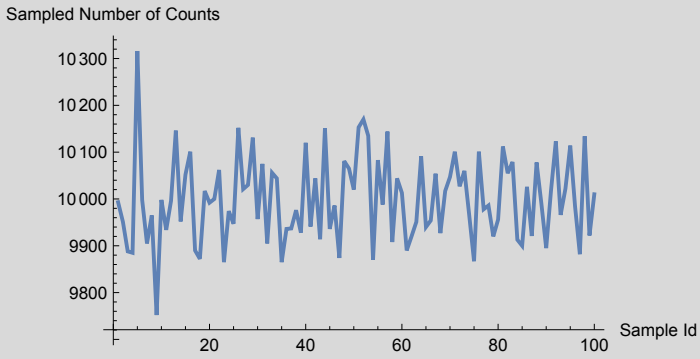


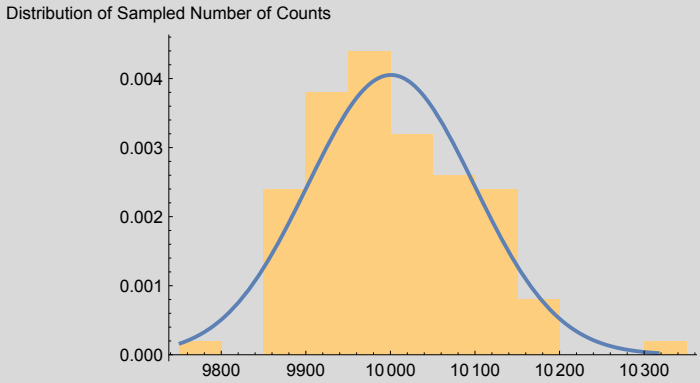

```
⟨|actual == expected μ → 10 000., expected stddev → 100., n samples → 100,
 actual stddev → 102.821, total → 1 000 000, min → 9786, max → 10 270|⟩
```

276 DIMENSIONS

In[67]:=
```
countsStats[sampledShellCounts2[0.01, 276, 1 000 000]]
```

Out[67]=






```
⟨|actual == expected μ → 10 000., expected stddev → 100., n samples → 100,
 actual stddev → 91.4368, total → 1 000 000, min → 9752, max → 10 316|⟩
```

To extend the theorem to more dimensions, let x^2 stand for the Euclidean norm of all dimensions other than the last two. The integral still does not depend on x.

QED Theorem 1 ■