A Method for Generating 3D Point Clouds with Logarithmic Radial and Uniform Angular Distribution

1. Introduction to Spatial Point Distribution

1.1. Context and Motivation

The generation of precisely distributed point clouds in three-dimensional space is a fundamental task in numerous scientific and engineering disciplines. While uniform random sampling is sufficient for some applications, many physical phenomena and computational models demand non-uniform distributions to accurately represent underlying structures or optimize sampling strategies. For instance, in acoustics and electromagnetics, sensor arrays are often designed with logarithmic spacing to match the natural attenuation of signal strength over distance. In computational fluid dynamics (CFD) and astrophysical simulations, particle seeding may require higher density near a central object, such as a star or a vortex core, with density decreasing logarithmically outwards to model accretion disks or nebulae. Similarly, procedural generation in computer graphics leverages controlled point densities to create realistic natural objects like star clusters or galaxies.
This report addresses the specific problem of generating a set of N points in 3D space that satisfy two distinct criteria:

1. The radial distances of the points from a central origin are logarithmically distributed between a specified inner radius, rin, and an outer radius, rout.
2. For any given radius, the points located on the corresponding spherical surface are distributed as uniformly as possible in angle, ensuring equal-area representation across the sphere.

1.2. The Dual-Component Challenge

The problem can be deconstructed into two orthogonal components: a radial distribution challenge and an angular distribution challenge. The former involves placing points along a line according to a specific density function (logarithmic), while the latter concerns the even

distribution of points on a spherical surface. A powerful and efficient approach to solving the combined problem is to treat these components as independent and combine their solutions. Any point P in 3D Cartesian space can be represented by its position vector p. This vector can be uniquely defined by its magnitude r= // p // and its direction, which is represented by a unit vector u=p/r. The user's request imposes separate constraints on these two elements: r must follow a logarithmic distribution, and u must be uniformly distributed over the unit sphere. This separation of concerns suggests a clear and modular algorithmic strategy. First, a set of N radial magnitudes, R={r1,r2,...,rN}, is generated according to the logarithmic spacing requirement. Concurrently, a set of N unit direction vectors, U={u1,u2,...,uN}, is generated to be uniformly distributed on the surface of a sphere. The final set of points is then constructed by scaling each unit vector ui by its corresponding radial magnitude ri. This modular design not only simplifies the problem conceptually but also lends itself to a highly efficient, vectorized implementation using numerical libraries such as NumPy.

2. The Mathematics of Uniform Angular Distribution

2.1. The Challenge of Even Spacing on a Sphere: The Polar Clustering Problem

Distributing points uniformly on the surface of a sphere is a non-trivial problem. An intuitive but flawed approach is to sample the spherical coordinate angles—azimuth $\theta \in [0,2\pi]$ and inclination (or polar angle) $\phi \in [0,\pi]$—from uniform distributions. This method fails because it does not account for the geometry of the sphere, resulting in a significant oversampling of points near the poles.
The mathematical reason for this failure lies in the formula for a differential surface area element on a sphere, which is given by $dA=\rho 2\sin(\phi)d\phi d\theta$. For a constant radius ρ, the area element is directly proportional to $\sin(\phi)$. When ϕ is near the poles (i.e., $\phi \approx 0$ or $\phi \approx \pi$), $\sin(\phi)$ approaches zero. Consequently, a constant step in the polar angle, $d\phi$, corresponds to a much smaller surface area near the poles than it does near the equator (where $\phi=\pi/2$ and $\sin(\phi)=1$). Uniformly sampling ϕ therefore allocates an equal number of points to regions of unequal area, causing a high-density clustering at the poles.[1]

2.2. The Fibonacci Lattice: A Deterministic and Elegant Solution

An effective and computationally efficient method for achieving a near-uniform distribution is the Fibonacci lattice, also known as the Fibonacci sphere.[1] This deterministic algorithm leverages properties of the golden ratio,
$\Phi=(1+5)/2$, to arrange points in a spiral pattern that covers the sphere with remarkable

evenness. The core idea is to use an angular increment related to the golden angle, which ensures that the spiral path never perfectly repeats, thus filling the surface without creating lines or clusters.[1]
The algorithm can be derived as follows, providing a direct method to generate Cartesian coordinates without intermediate spherical coordinate calculations:

1. Index the points from i=0 to N−1.
2. To achieve an equal-area distribution, the points must be spaced evenly along a central axis. This is accomplished by sampling the cosine of the polar angle, cos(ϕ), uniformly. The z-coordinate of each point on a unit sphere is set to zi=1−2×(i+0.5)/N. This expression generates values for zi that are linearly spaced from just below 1 to just above −1. The addition of an offset of 0.5 to the index i corresponds to using the midpoint rule, a refinement that improves the point packing quality, especially near the poles.[2]
3. For each height zi, the radius of the corresponding horizontal circle on the sphere is calculated as ri=1−zi2.
4. The azimuthal angle θi for each point is determined using an increment based on the golden angle. A common formulation is θi=2πi/Φ2=π×(3−5)×i.[1]
5. Finally, the Cartesian coordinates (xi,yi,zi) for each point on the unit sphere are calculated as:
    - xi=ricos(θi)
    - yi=risin(θi)
    - zi (from step 2)

This formulation is particularly efficient. A standard approach would first compute the spherical coordinates ϕi=arccos(zi) and θi, and then apply the full spherical-to-Cartesian conversion formulas: x=ρsin(ϕ)cos(θ), y=ρsin(ϕ)sin(θ), and z=ρcos(ϕ).[4] However, for a unit sphere where
ρ=1, the term sin(ϕ) is equivalent to 1−cos2(ϕ)=1−z2, which is precisely the radius ri calculated in step 3. The algorithm thus computes xi=sin(ϕi)cos(θi) and yi=sin(ϕi)sin(θi) directly. This bypasses the need for an expensive arccos function call for each point, leading to a more performant and numerically stable implementation.

3. The Principle of Logarithmic Radial Spacing

3.1. Contrasting Linear and Logarithmic Scales

To implement the radial component of the distribution, it is essential to distinguish between linear and logarithmic spacing. Numerical libraries like NumPy provide dedicated functions for both.

- **Linear Spacing:** The numpy.linspace function generates values where the difference

between consecutive elements is constant (vi+1−vi=const). This is appropriate for phenomena that change uniformly over an interval.[7]

- **Logarithmic Spacing:** The numpy.logspace function generates values where the *ratio* of consecutive elements is constant (vi+1/vi=const). This means the values are spaced linearly on a logarithmic scale. This type of spacing is ideal for representing quantities that span multiple orders of magnitude, such as frequencies in signal processing, financial models with compound interest, or astronomical distances.[8] A visual comparison would show
  logspace placing more samples at the lower end of a range and progressively increasing the gap between samples as their value increases.

3.2. Implementing Radial Distribution with numpy.logspace

The numpy.logspace function is the ideal tool for generating the required radial distances. Its signature is numpy.logspace(start, stop, num, base=10.0).[10] A critical detail in using this function correctly is understanding that the
start and stop parameters do not represent the literal start and end values of the output sequence. Instead, they represent the *exponents* of the specified base. The generated sequence starts at base**start and ends at base**stop.[10]
This exponent-based paradigm requires a transformation of the user-provided input radii. If a user specifies an inner_radius and an outer_radius, these values must be converted into their corresponding exponents before being passed to logspace. The logarithm function, being the inverse of exponentiation, performs this conversion. For the default base=10, the correct inputs are:

- start = np.log10(inner_radius)
- stop = np.log10(outer_radius)

The underlying mechanism of numpy.logspace is equivalent to first creating a linearly spaced array of exponents with numpy.linspace(start, stop, num) and then raising the base to the power of these exponents.[10] A robust and user-friendly implementation must perform this logarithmic transformation internally, abstracting this mathematical detail and allowing the user to work with intuitive, literal radius values.

4. Algorithmic Synthesis and Python Implementation

4.1. A Vectorized, High-Performance Approach

By combining the solutions for the angular and radial components, a complete and highly efficient algorithm can be constructed. The use of NumPy for vectorized operations is

paramount to achieving high performance by avoiding explicit loops in Python. The strategy is as follows:

1. **Generate Unit Vectors:** Create an array of shape (N,3) containing N unit vectors (x,y,z) using the Fibonacci sphere algorithm. All calculations are performed on NumPy arrays of size N.
2. **Generate Radii:** Create a 1D array of shape (N,) containing N radial distances using numpy.logspace. This step includes the necessary np.log10 transformation of the input inner_radius and outer_radius.
3. **Scale Vectors:** Scale each unit vector by its corresponding radius. To do this efficiently, the radii array of shape (N,) is reshaped to (N,1). NumPy's broadcasting rules then allow for the element-wise multiplication of the (N,3) unit vector array by the (N,1) radii array, resulting in the final (N,3) point cloud.

4.2. The Complete Python Function: generate_log_spherical_points

The following Python function encapsulates the described methodology, providing a documented, type-hinted, and validated implementation.

Python

```
import numpy as np

def generate_log_spherical_points(
    num_points: int,
    inner_radius: float,
    outer_radius: float
) -> np.ndarray:
    """
    Generates a set of 3D points with logarithmic radial and uniform angular distribution.

    The function creates a point cloud where the distances of points from the origin
    are logarithmically spaced between an inner and outer radius. For any given
    radius, the points on the corresponding spherical shell are distributed
    uniformly using the Fibonacci lattice method, which ensures an equal-area
    distribution.

    Args:
        num_points: The total number of points to generate.
        inner_radius: The minimum distance from the origin. Must be positive.
        outer_radius: The maximum distance from the origin. Must be greater than
                      or equal to inner_radius.
```

```
Returns:
    A NumPy array of shape (num_points, 3) containing the Cartesian
    coordinates (x, y, z) of the generated points.
"""
# Input validation
if not isinstance(num_points, int) or num_points <= 0:
    raise ValueError("num_points must be a positive integer.")
if not (isinstance(inner_radius, (int, float)) and inner_radius > 0):
    raise ValueError("inner_radius must be a positive number.")
if not (isinstance(outer_radius, (int, float)) and outer_radius >= inner_radius):
    raise ValueError("outer_radius must be a positive number and >= inner_radius.")

# --- 1. Generate uniformly distributed unit vectors (Fibonacci Lattice) ---
# Create an array of indices for each point
indices = np.arange(0, num_points, dtype=float) + 0.5

# Uniformly distribute points along the z-axis (cos(phi))
z = 1 - 2 * indices / num_points

# Calculate the radius in the xy-plane for each point
radius_xy = np.sqrt(1 - z**2)

# Calculate the azimuthal angle using the golden angle
golden_angle = np.pi * (3. - np.sqrt(5.))
theta = golden_angle * indices

# Convert to Cartesian coordinates for the unit sphere
x = radius_xy * np.cos(theta)
y = radius_xy * np.sin(theta)

# Stack into an (N, 3) array of unit vectors
unit_vectors = np.stack([x, y, z], axis=1)

# --- 2. Generate logarithmically spaced radii ---
# The start and stop parameters for logspace are exponents
start_exp = np.log10(inner_radius)
stop_exp = np.log10(outer_radius)
radii = np.logspace(start_exp, stop_exp, num_points)

# --- 3. Scale unit vectors by radii using broadcasting ---
# Reshape radii to (N, 1) to broadcast with (N, 3) unit_vectors
points = unit_vectors * radii[:, np.newaxis]
```

```
    return points
```

4.3. Code Dissection and Performance Analysis

The implementation is composed of three main parts, each leveraging vectorized NumPy operations:

- **Angular Distribution:** The lines indices = np.arange(...), z = 1 -..., radius_xy = np.sqrt(...), theta = golden_angle * indices, and the subsequent np.cos and np.sin calls all operate on entire arrays at once. This section generates the unit vectors representing the uniform angular distribution.[1]
- **Radial Distribution:** The line radii = np.logspace(...) efficiently generates the logarithmically spaced radial magnitudes by correctly converting the input radii to their base-10 logarithms.[8]
- **Combination:** The final multiplication, points = unit_vectors * radii[:, np.newaxis], is where NumPy's broadcasting mechanism shines. It scales each 3D unit vector by its corresponding scalar radius without any explicit loops, producing the final point cloud.

The computational complexity of this algorithm is determined by the NumPy operations, all of which are performed on arrays of size num_points. Each operation has a time complexity proportional to the number of elements, making the overall algorithm's time complexity O(N). Similarly, the space complexity is O(N) to store the intermediate and final arrays. This makes the solution highly efficient and scalable to large numbers of points.

5. Validation and Visualization

5.1. Visual Verification of the Point Cloud

A qualitative assessment of the generated point cloud can be performed through visualization. Using a 3D plotting library such as Matplotlib's mplot3d toolkit [3], a scatter plot of the points can be generated. Inspection from different viewpoints can confirm the desired properties:

- **General View:** An arbitrary view should show a spherical volume of points with no obvious lines, clusters, or patterns, confirming the uniformity of the angular distribution.
- **Cross-Sectional View:** A slice through the origin (e.g., viewing along the x-axis to see the yz-plane) will reveal the radial distribution. The density of points should be visibly highest near the inner_radius and should decrease as the distance from the origin

increases towards the outer_radius, clearly demonstrating the logarithmic spacing.

5.2. Quantitative Analysis of Distribution Properties

For a more rigorous validation, the statistical properties of the distribution can be analyzed.

- **Radial Distribution Test:** The logarithmic nature of the radial spacing can be confirmed numerically. First, the radial distance of each generated point is calculated using r=x2+y2+z2, or more efficiently, r = np.linalg.norm(points, axis=1). Then, the base-10 logarithm of these radii is computed. A histogram of these logarithmic values should be approximately uniform, confirming that the points are spaced linearly in the logarithmic domain, which is the definition of a logarithmic distribution.
- **Angular Distribution Test:** The uniformity of the angular distribution can be assessed by analyzing nearest-neighbor distances. By selecting a subset of points located within a thin spherical shell (e.g., points with radii between ravg−ϵ and ravg+ϵ), the influence of the radial distribution is minimized. For each point in this subset, the distance to its nearest neighbor on the shell can be calculated. A histogram of these minimum distances should exhibit a single, narrow peak. A broad distribution or the presence of multiple peaks would indicate non-uniformity or clustering, whereas a sharp peak confirms that the points are spaced evenly.

6. Advanced Considerations and Applications

6.1. Comparison with Alternative Algorithms

The Fibonacci lattice was chosen for its excellent balance of quality, performance, and implementation simplicity. However, other algorithms exist for distributing points on a sphere, each with its own trade-offs.

- **Electrostatic Repulsion:** This method treats each point as an electron constrained to a spherical surface and simulates their repulsion until they reach a stable, minimum-energy configuration.[1] This often produces near-optimal distributions (in the sense of maximizing the minimum distance between any two points) but is computationally intensive, with each simulation step typically having a complexity of O(N2).
- **Hypercube Rejection:** This probabilistic method involves generating points uniformly within a cube that encloses the sphere, rejecting any points that fall outside the sphere, and then normalizing the remaining vectors to unit length.[1] While simple to conceptualize, it is inefficient as a large fraction of the generated points are discarded, and the quality of the final distribution is not guaranteed.

The Fibonacci lattice is a deterministic, O(N) approximation. While its solutions do not coincide with the provably optimal arrangements for small N (such as the vertices of Platonic solids for N=4,6,8,12,20), its quality is exceptionally high for most practical purposes, especially for larger N.[2] The trade-off between computational cost and marginal gains in uniformity overwhelmingly favors the Fibonacci lattice in a vast majority of applications.

Algorithm	Type	Complexity	Uniformity Quality	Implementation
Fibonacci Lattice	Deterministic	O(N)	Excellent (Equal-Area)	Simple
Electrostatic Repulsion	Iterative/Simulative	O(N2) per step	Near-Optimal (Energy Min.)	Complex
Hypercube Rejection	Probabilistic	Variable, Inefficient	Stochastic	Simple

6.2. Potential Extensions and Modifications

The presented methodology can be extended in several ways. The number of angular points could be varied for different radial shells, for instance, by increasing the number of points on outer shells to maintain a more constant spatial volume density. The core principles could also be adapted for other geometries, such as ellipsoids or tori, by modifying the scaling and coordinate transformation steps. Furthermore, the Fibonacci lattice concept itself can be generalized to higher dimensions to distribute points on hyperspheres, a feature supported by some specialized libraries.[12]

6.3. Case Studies in Scientific Computing

The utility of the provided function is best illustrated through specific case studies:

- **Acoustics:** For designing a spherical microphone array for sound source localization, points can be generated to place microphones. Logarithmic spacing is ideal for capturing a wide spectrum of frequencies, as human perception of frequency is also logarithmic.
- **Computer Graphics:** In the procedural generation of a nebula, this function can be used to place stars or dust particles. The logarithmic distribution creates a dense, bright core that naturally fades into the sparse outer regions.
- **Physics Simulation:** When initializing a simulation of an expanding supernova remnant, the initial particle positions can be set using this function to reflect a density profile that is highest near the point of explosion and decreases with distance.

7. Conclusion

This report has detailed a robust and efficient method for generating a 3D point cloud with a logarithmic radial distribution and a uniform angular distribution. The solution is based on a powerful two-part methodology that decouples the problem into its radial and angular components. For the angular distribution, the deterministic Fibonacci lattice algorithm provides a high-quality, equal-area arrangement of points on a sphere with optimal O(N) complexity. For the radial distribution, the numpy.logspace function is used, with careful attention paid to its exponent-based parameterization to ensure a correct implementation. The final synthesized algorithm, presented as a complete Python function, is high-performance due to its fully vectorized nature, mathematically sound, and easy to use. The analysis and validation techniques provided confirm that the generated points adhere to the specified constraints. This solution represents a state-of-the-art, practical tool for a common and important problem in computational science, engineering, and computer graphics.

Works cited

1. Evenly distributing n points on a sphere - Stack Overflow, accessed August 15, 2025, https://stackoverflow.com/questions/9600801/evenly-distributing-n-points-on-a-sphere
2. Evenly distributing points on a sphere - Extreme Learning, accessed August 15, 2025, https://extremelearning.com.au/evenly-distributing-points-on-a-sphere/
3. Create equidistant points on the surface of a sphere using Fibonacci sphere algorithm - GitHub Gist, accessed August 15, 2025, https://gist.github.com/Seanmatthews/a51ac697db1a4f58a6bca7996d75f68c
4. 5.7: Cylindrical and Spherical Coordinates - Mathematics LibreTexts, accessed August 15, 2025, https://math.libretexts.org/Courses/Mount_Royal_University/Calculus_for_Scientists_II/7%3A_Vector_Spaces/5.7%3A_Cylindrical_and_Spherical_Coordinates
5. math.libretexts.org, accessed August 15, 2025, https://math.libretexts.org/Courses/Mount_Royal_University/Calculus_for_Scientists_II/7%3A_Vector_Spaces/5.7%3A_Cylindrical_and_Spherical_Coordinates#:~:text=To%20convert%20a%20point%20from,y%2Bz2).
6. Spherical coordinates - Math Insight, accessed August 15, 2025, https://mathinsight.org/spherical_coordinates
7. numpy.linspace — NumPy v2.4.dev0 Manual, accessed August 15, 2025, https://numpy.org/devdocs/reference/generated/numpy.linspace.html
8. numpy.logspace with Examples - Medium, accessed August 15, 2025, https://medium.com/@amit25173/numpy-logspace-with-examples-87aed47b4728
9. Python Numpy logspace() - Generate Logarithmic Spaced Array - Vultr Docs,

accessed August 15, 2025, https://docs.vultr.com/python/third-party/numpy/logspace
10. numpy.logspace — NumPy v2.4.dev0 Manual, accessed August 15, 2025, https://numpy.org/devdocs/reference/generated/numpy.logspace.html
11. NumPy logspace() - Programiz, accessed August 15, 2025, https://www.programiz.com/python-programming/numpy/methods/logspace
12. erikbrinkman/fibonacci_lattice: python project for generating fibonacci lattices - GitHub, accessed August 15, 2025, https://github.com/erikbrinkman/fibonacci_lattice