# Low Discrepancy Sampling Method on Hyperspheres

@luk036

## Abstract

This notes presents a discussion of the generation of low discrepancy sampling methods for n-dimensional spheres. Low discrepancy sequences are important and have been used in a variety of applications, including numerical integration, optimization, and simulation. This paper addresses the desirable properties of samples over an n-sphere, including uniformity, determinism, and incrementality. Subsequently, the proposed method for generating low discrepancy sequences over higher dimensional sphere is presented, which is based on the van der Corput sequence. We provide a comprehensive account of the algorithm and its implementation. Furthermore, the paper presents the results of numerical experiments conducted to evaluate the performance of the proposed method, including a comparison with randomly generated sequences and other proposed methods such as the Hopf coordinate method and the cylindrical coordinate method.

## Motivation

* Low discrepancy sequences (LDS) play a pivotal role in numerous fields of mathematics, computer science, and engineering. These sequences.
* While low discrepancy sampling methods have been extensively studied in lower dimensions, there is an increasing need for efficient sampling techniques in higher dimensions.

## Advantages over random sampling methods

* Uniform distribution: They provide a uniform distribution of points, which reduces the occurrence of computational errors in a variety of applications.
* Determinism: They provide a deterministic sequence of points, which is essential for reproducibility and consistency in simulations and experiments.
* Incrementality: They allow for the efficient addition of new points to an existing sequence, which is useful in applications where the number of points required to solve a problem is not known in advance.

## Potential Applications

* Robot Motion Planning [@yershova2010generating]: In high-dimensional spaces such as and , Halton sequences provide uniformly distributed point sets, which are suitable for robot path planning and attitude control. This allows for the optimization of computational efficiency and the accuracy of motion trajectories.
* Wireless Communication Coding [@utkovski2006construction]: Spherical coding in MIMO systems employs the dots generated by Halton sequences as codewords to enhance the stability and anti-interference capacity of signal transmission, and to improve the data transmission rate and quality.
* Multivariate Empirical Mode Decomposition [@rehman2010multivariate]: In multivariate empirical mode decomposition, Halton sequences can be used to construct more accurate signal models and improve the accuracy of signal processing.
* Filter Bank Design [@mandic2011filter]: In the context of filter bank design, the utilization of Halton sequences has the potential to facilitate the construction of more precise filter parameters, thereby enhancing the accuracy of signal processing.

# Overview of Low Discrepancy Sequences

## Basic: van der Corput sequence

The van der Corput sequence is a mathematical sequence that is employed to generate a series of evenly distributed numbers between 0 and 1. This function is of particular utility in a multitude of fields, including but not limited to computer graphics and numerical analysis. The van der Corput sequence is a low-discrepancy sequence used to generate uniformly distributed points in the interval [0,1]. It is constructed by means of a specific base (usually prime).

Example:

8 4 2 6 1 9 5 3 7  
----o---o-------o-------o-------o---o---o-------o-------o--------

## Python Implementation

def vdc(k: int, base: int = 2) -> float:  
 res = 0.0  
 denom = 1.0  
 while k != 0:  
 denom \*= base  
 remainder = k % base  
 k //= base  
 res += remainder / denom  
 return res

## VdCorput class

class VdCorput:  
 def \_\_init\_\_(self, base: int = 2) -> None:  
 self.count = 0  
 self.base = base  
  
 def pop(self) -> float:  
 self.count += 1 # ignore 0  
 return vdc(self.count, self.base)  
  
 def reseed(self, seed: int) -> None:  
 self.count = seed

## Halton sequence on

The Halton sequence is a deterministic sequence of points that are distributed uniformly in a multidimensional space. The Halton sequence is named after the mathematicians Halton and Rutishauser, who developed it in the 1960s. The Halton sequence is constructed by combining two or more the van der Corput sequences, which are generated using different prime numbers as the base.

## Python Implementation

from typing import List, Sequence  
  
class Halton:  
 def \_\_init\_\_(self, base: Sequence[int]) -> None:  
 self.vdc0 = VdCorput(base[0])  
 self.vdc1 = VdCorput(base[1])  
  
 def pop(self) -> List[float]:  
 return [self.vdc0.pop(), self.vdc1.pop()]  
  
 def reseed(self, seed: int) -> None: ...

## Halton sequence on

The concept of the Halton sequence can be extended to higher dimensions through the combination of multiple one-dimensional sequences, each generated using a distinct prime number as the base. This approach permits the generation of points that are distributed uniformly in a multidimensional space. There are a multitude of potential applications for quasi-Monte Carlo methods (QMC).

## Python Implementation

class HaltonN:  
 def \_\_init\_\_(self, base: Sequence[int]) -> None:  
 self.vdcs = [VdCorput(b) for b in base]  
  
 def pop(self) -> List[float]:  
 return [vdc.pop() for vdc in self.vdcs]  
  
 def reseed(self, seed: int) -> None:  
 for vdc in self.vdcs:  
 vdc.reseed(seed)

## Unit Circle

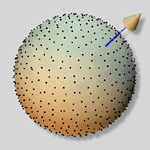
The concept of the Halton sequence can be extended to generate points on a unit circle in a two-dimensional space. The objective of this study is to present a method for generating a sequence of points distributed uniformly around a circle. The objective is to utilize the van der Corput sequence as a foundation for the generation of points on a circle. In order to fulfill its intended function, the Circle class employs the following logical structure:

## Python Implementation

from math import cos, pi, sin, sqrt  
TWO\_PI = 2.0 \* pi  
  
class Circle:  
 def \_\_init\_\_(self, base: int) -> None:  
 self.vdc = VdCorput(base)  
  
 def pop(self) -> List[float]:  
 theta = self.vdc.pop() \* TWO\_PI # map to [0, 2π]  
 return [cos(theta), sin(theta)]

## Unit Sphere

The concept of the Halton sequence can be extended to generate points on the surface of a unit sphere (a sphere with a radius of 1). The objective is to provide a method for generating a sequence of points on a sphere’s surface that are distributed in an even manner. The Sphere class has been applied in computer graphics applications [@wong1997sampling].



image

## How to generate Unit Sphere

To achieve its purpose, the Sphere class employs a sophisticated algorithm that combines one-dimensional and two-dimensional sequences to generate a three-dimensional point. The following is an explanation of the method of operation.

1. The VdCorput generator is employed to generate a value between 0 and 1, which is then transformed to a value between -1 and 1. This value subsequently becomes the z-coordinate, representing the up-down position on the sphere.
2. The radius of the horizontal circle at the specified z-position is then calculated using the cylindrical mapping formula.

* =   
  =
* % map to
* % map to

1. Subsequently, the Circle generator is employed to obtain a point on the aforementioned horizontal circle, thereby providing the x and y coordinates.

## Python Implementation

class Sphere:  
 def \_\_init\_\_(self, base: Sequence[int]) -> None:  
 self.vdc = VdCorput(base[0])  
 self.cirgen = Circle(base[1])  
  
 def pop(self) -> List[float]:  
 cosphi = 2.0 \* self.vdc.pop() - 1.0 # map to [-1, 1]  
 sinphi = sqrt(1.0 - cosphi \* cosphi) # cylindrical mapping  
 [costheta, sintheta] = self.cirgen.pop()  
 return [sinphi \* costheta, sinphi \* sintheta, cosphi]

## and SO(3)

* The concept may be extended to generate points on a four-dimensional sphere (also known as a three-sphere) through
* The application of a mathematical technique known as the Hopf fibration [@mitchell2008sampling; @yershova2010generating].
* The objective of the original paper is to generate optimal deterministic grid point sets for , SO(3).
* It should be noted that the cylindrical mapping method is not applicable in higher dimensions.

## How to generate and SO(3)

In order to achieve its intended purpose, the class employs the use of three van der Corput sequences (implemented by the VdCorput class, which is not shown in this algorithm snippet) to generate values between 0 and 1. These values are then transformed through the application of trigonometric functions and square roots, with the objective of mapping them onto the surface of the 3-sphere.

* Hopf coordinates (cf. [@yershova2010generating])
* is a principal circle bundle over the

Similar to the Halton sequence generation on , we perform the mapping:

* % map to
* % map to for SO(3), or
* % map to for
* % map to

## Python Implementation

class Sphere3Hopf:  
 def \_\_init\_\_(self, base: Sequence[int]) -> None:  
 self.vdc0 = VdCorput(base[0])  
 self.vdc1 = VdCorput(base[1])  
 self.vdc2 = VdCorput(base[2])  
  
 def pop(self) -> List[float]:  
 phi = self.vdc0.pop() \* TWO\_PI # map to [0, 2π]  
 psy = self.vdc1.pop() \* TWO\_PI # map to [0, 2π]  
 vdc = self.vdc2.pop()  
 cos\_eta = sqrt(vdc)  
 sin\_eta = sqrt(1.0 - vdc)  
 return [  
 cos\_eta \* cos(psy),  
 cos\_eta \* sin(psy),  
 sin\_eta \* cos(phi + psy),  
 sin\_eta \* sin(phi + psy),  
 ]

# Our approach

## 3-sphere

This algorithm is part of a Sphere N Generator, which is designed to create points on the surface of spheres in different dimensions. It’s a tool that could be used by people working with 3D graphics or mathematical simulations.

The main input for this algorithm is a list of integers, which are used as bases for generating sequences of numbers. These bases are used to initialize different types of generators that create points on spheres.

The output of this algorithm is a series of lists containing floating-point numbers. Each list represents a point on the surface of a sphere, with the number of elements in the list corresponding to the dimension of the sphere.

To achieve its purpose, the algorithm starts by defining some constants and helper functions that are used in the calculations. These functions (get\_tp\_odd, get\_tp\_even, and get\_tp) create lookup tables for mapping values in different dimensions.

The algorithm then defines two main classes:

1. SphereGen: This is a template class that defines what methods all sphere generators should have.
2. Sphere3: This class generates points on a 4-dimensional sphere. It uses a combination of special sequences (van der Corput and 3-dimensional sphere points) to create 4D points.

The Sphere3 class has methods to generate new points (pop) and to reset the generator with a new starting point (reseed).

## Unit Sphere

* Polar coordinates:
* Spherical surface element:

## Python Implementation

X: np.ndarray = np.linspace(0.0, PI, 300)  
NEG\_COSINE: np.ndarray = -np.cos(X)  
SINE: np.ndarray = np.sin(X)  
F2: np.ndarray = (X + NEG\_COSINE \* SINE) / 2.0  
  
class Sphere3(SphereGen):  
 def \_\_init\_\_(self, base: List[int]) -> None:  
 self.vdc = VdCorput(base[0])  
 self.sphere2 = Sphere(base[1:3])  
  
 def pop(self) -> List[float]:  
 ti = PI \* self.vdc.pop() # map to [0, π]  
 xi = np.interp(ti, F2, X)  
 cosxi = math.cos(xi)  
 sinxi = math.sin(xi)  
 return [sinxi \* s for s in self.sphere2.pop()] + [cosxi]

## n-sphere

This algorithm defines a class called SphereN, which is designed to generate points on the surface of a sphere in any number of dimensions (3 or more). The purpose of this class is to provide a way to create evenly distributed points on a high-dimensional sphere, which can be useful in various mathematical and scientific applications.

The SphereN class takes a list of integers as input when it’s created. These integers are used as bases for generating sequences of numbers that will be transformed into points on the sphere. The main output of this class is a list of floating-point numbers, where each list represents a point on the surface of the sphere.

To achieve its purpose, the SphereN class uses a recursive approach. It creates a sphere in a higher dimension by combining a lower-dimensional sphere with an additional coordinate. For example, to create a 4-dimensional sphere, it uses a 3-dimensional sphere and adds one more coordinate.

The key logic in this algorithm involves transforming numbers from one range to another and using trigonometric functions (sine and cosine) to map these numbers onto the surface of a sphere. It uses a technique called interpolation to smoothly transition between values.

An important aspect of this algorithm is its recursive nature. For dimensions higher than 3, it creates a chain of SphereN objects, each handling one dimension and relying on the next one down for the lower dimensions. This allows it to generate points on spheres of any dimension, limited only by the computer’s memory and processing power.

## Unit N-Sphere

* Polar coordinates:
* Spherical surface element:

## How to Generate the Point Set

* where
* Let = , where .  
  + Note 1: can be defined recursively as:
  + $$ f\_j() =
  + (1/3)( -\cos\theta \sin^2\theta - 2 \cos\theta)  
     (-1/3) \cos\theta (3 - \cos^2\theta)
  + $$
  + Note 2: is a monotonic increasing function in
* Map uniformly to :
* Let
* Define recursively as:

# Numerical Experiments

The objective of this experiment is to serve as a test suite for the evaluation of disparate methods for the generation of points on the surface of a high-dimensional sphere. The experiment is comprised of three principal components: the generation of random points, the calculation of a dispersion measure, and the execution of tests on disparate point generation methods.

The primary objective of this experiment is to evaluate and contrast the quality of point distributions on a sphere, employing both random generation and low-discrepancy sequences (LDS). This is accomplished by first generating a set of points, then constructing a convex hull from those points, and finally calculating a dispersion measure based on the triangles formed by the hull.

* Dispersion roughly measured by the difference of the maximum distance and the minimum distance between every two neighbour points:
* where
* The convex hull is constructed using the scipy.spatial.ConvexHull function.

The experiment does not accept any direct inputs from the user. In lieu of user input, the experiment utilizes predefined parameters, including the number of points to be generated (600) and the dimensionality of the sphere (5D in the random case and 4D in the LDS cases).

The primary outputs are the dispersion measures calculated for each method. Subsequently, the aforementioned measures are subjected to a comparative analysis with the anticipated values inherent to the test functions.

The experiment is designed to achieve its stated objective through a series of steps.

1. Subsequently, a function, designated as “discrep\_2,” is defined which calculates a dispersion measure for a given set of points. This measure is based on the minimum and maximum angles between pairs of points in each simplex (triangle in higher dimensions) of the convex hull.
2. The program includes a function, random\_point\_on\_sphere, which generates a random point on the surface of a sphere in any number of dimensions.
3. The run\_random function generates 600 random points on a 5D sphere, constructs a convex hull, and computes the dispersion measure. To generate random points on , the spherical symmetry of the multidimensional Gaussian density function can be exploited. This results in a normalized vector () that is uniformly distributed over the hypersphere . (Fishman, G. F. (1996))
4. The run\_lds function performs the same operations, but utilizes a provided generator (either SphereN or CylindN) to create the points in the aforementioned manner.
5. Finally, there are three test functions that execute these methods and compare the results to expected values:

* test\_random verifies the random point generation
* test\_sphere\_n validates the SphereN generator
* test\_cylind\_n assesses the CylindN generator

The key logic flow involves generating points, creating a convex hull, and then calculating the dispersion measure. The dispersion measure itself involves finding the minimum and maximum angles between pairs of points in each simplex of the hull.

This experiment is important because it allows comparison between random and deterministic (LDS) methods of generating points on a sphere, which can be crucial in various scientific and mathematical applications where uniform distribution of points is needed.

![Left: our, right: random](data:application/pdf;base64,)

Left: our, right: random

![Result for S^3. Compared with Hopf coordinate method](data:application/pdf;base64,)

Result for . Compared with Hopf coordinate method

![Result for S^3. Compared with cylindrical mapping method.](data:application/pdf;base64,)

Result for . Compared with cylindrical mapping method.

![Result for S^4. Compared with cylindrical mapping method.](data:application/pdf;base64,)

Result for . Compared with cylindrical mapping method.

## Conclusions

This paper provides a comprehensive discussion of low discrepancy sampling methods for n-dimensional spheres. It introduces a proposed methodology for generating low-discrepancy sequences on n-dimensional spheres based on the van der Corput sequence, addressing challenges associated with high-dimensional sampling while maintaining desirable properties. The paper outlines key concepts such as the van der Corput sequence, Halton sequence, and their applications to unit circles, unit spheres, and higher-dimensional spaces. The document provides detailed explanations and implementations of algorithms for generating points on various dimensional spheres, including the use of Hopf fibration for four-dimensional spheres.

## Future Work

The proposed method represents a valuable contribution to the field of low-discrepancy sequences and has the potential to be applied in a variety of applications. The method is both efficient and capable of generating point sets with a high degree of uniformity. The method can be employed in the generation of point sets for a variety of applications, including Monte Carlo simulations, optimization, and machine learning. Moreover, the method is uncomplicated to implement and can be employed by researchers and practitioners alike.

## References