Fast Python sampler for the von Mises Fisher distribution

Carlos Pinzón, Kangsoo Jung

Fast Python sampler of the von Mises Fisher distribution

Carlos Pinzón
Inria, LIX

Kangsoo Jung
Inria, LIX

Abstract

This paper implements a method for sampling from the d-dimensional Von Mises Fisher distribution using NumPy, focusing on speed and readability. The complexity of the algorithm is $O(nd)$ for n samples, which is theoretically optimal taking into account that nd is the output size.

1 Introduction

The von Mises Fisher distribution (VMF), named after Richard von Mises [vM81] and Ronald Fisher [Fis53], is a probability distribution that is used in directional statistics and privacy mechanisms. The domain of the VMF distribution in d dimensions is the $d-1$-dimensional sphere $S^{d-1} \subseteq \mathbb{R}^d$, where $d \in \{1, 2, ...\}$ is a parameter. This distribution generalizes the von Mises distribution (VM) which corresponds to the case $d = 2$ and a circumference as domain, or equivalently, the interval $[-\pi, \pi]$.

Since the case $d = 1$ corresponds to a degraded sphere that collapses into the points -1 and 1, we will always assume a fixed number of dimensions $d \geq 2$. The case $d = 1$ will be discussed later.

The VMF distribution appears in many applications that are constrained to unit spheres because it is the result of conditioning an isotropic gaussian distribution to the unit sphere. When the gaussian is centered at the origin, the uniform distribution is recovered, and as the center is moved and the isotropic variance altered, the shape of the distribution varies. For given parameters of mean direction $\mu \in S^{d-1}$ and concentration $\kappa \in (0, \infty)$, the density function of the VMF at each point $x \in S^{d-1}$ is given by

$$p(x \mid \mu, \kappa) \propto_x \exp(\kappa \mu \cdot x),$$

where the $\cdot$ denotes the dot product, i.e. the standard inner product in d-dimensions, and the symbol $\propto_x$ denotes proportionality with the same factor for all x. In other words, the missing factor c depends only on the dimensionality d and the concentration κ.

When $d = 2$, the domain is S^1, i.e. the circumference of radius 1 in $\mathbb{R}^2$, and letting θ_0 and θ be the angles that define μ and x respectively, the density function is given by $p(\theta \mid \theta_0, \kappa) \propto_\theta \exp(\kappa \cos(\theta_0 - \theta))$, because $\mu \cdot x =$

$|\mu|\,|x|\cos(\theta_0 - \theta) = \cos(\theta_0 - \theta)$. This formula coincides with that of the VM distribution in the form that it is typically presented, with its parameter varying in $[-\pi, \pi]$.

There exist implementations for generating samples that follow a VM distribution in the main scientific Python packages SciPy [Sci23] and NumPy [Num23], but they do not implement the VMF distribution in general (for $d \neq 2$). In this paper, we propose a fast sample generator of the VMF distribution that runs in $\Theta(d)$ for every sample generated, which is optimal, since d is also to the size of the output.

2 Related work

There are two existing Python implementations for sampling from the VMF distribution [ten23, Whi23]. Before introducing the main difference between our algorithm and these implementations, let us describe in detail the principles they follow.

∘ **Principle 1 (rotation).** Due to the rotational symmetry, it suffices to have a sampler that handles only $\mu = \hat{e}_d = (0, 0, ..., 0, 1)^\top$, because the resulting samples can be rotated afterwards towards the true μ. This rotation is carried out using a rotation matrix, which can be computed using Givens rotations and Householder reflections [Hou58]. We present below an example of the actual implementation of this rotation, copied from a Python library [Bru22], whose essence is derived from an earlier paper [CA20].

◦ **Principle 2 (decomposition).** The sampling problem can be decomposed into computing the random angular deviation $\theta \in [0, \pi]$ between the output x and μ, and the random direction z in which the angle occurs.

Figure 2.1, taken from Weggenmann and Kerschbaum [WK21], depicts the decomposition principle for the case $d = 3$. The figure shows the angle θ between μ and x, and all the remaining orientation information, is captured in the $(d-1)$-dimensional unitary vector ξ. Since μ is fixed to be $\mu = \hat{e}_d = (0, 0, 1)$, we can embed $\xi = (\xi_1, \xi_2)$ in 3 dimensions as $z = (\xi_1, \xi_2, 0)$, so that it is perpendicular to μ. The problem of sampling from VMF can therefore be reduced to sampling θ and ξ separately, and letting the output (assuming $\mu = \hat{e}_d$) be $x = \cos(\theta)\mu + \sin(\theta)z$.

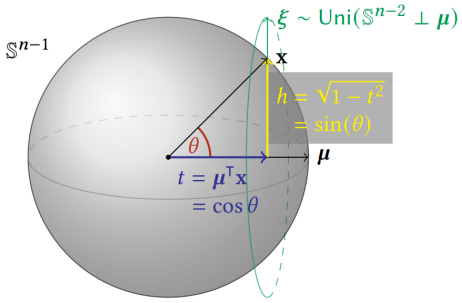

Figure 2.1: Decomposition into θ and ξ.

◦ **Principle 3 (Muller).** Sampling ξ can be done using the algorithm by Muller [Mul59]. More precisely, if $Y_1, ..., Y_{d-1} \sim \mathcal{N}(0, 1)$ are independent, and we let $Y = (Y_1, ..., Y_{d-1})^\top$, it is known that the unitary vector $\xi :- Y/|Y|$ is uniformly distributed on the $(d-2)$-sphere. For instance, when $d = 3$, ξ is a 2-dimensional vector taken from the 1-sphere (the circumference), and when $d = 2$, ξ is a 1-dimensional vector taken from the 0-sphere (the set $\{-1, 1\}$).

◦ **Principle 4 (Ulrich-Woods).** Sampling θ can be done efficiently using a rejection method with an envelope proposed by Ulrich [Ulr84], whose pseudocode is later presented by Woods [Woo94]. The distribution of the angle $\theta \in [0, \pi]$ is given by $p(\theta \mid \mu, \kappa) \propto_\theta (\sin\theta)^{d-3} \exp(\kappa \cos\theta)$. Let us put aside the trigonometric functions and focus on $t \overset{\text{def}}{:=} \cos\theta \in [-1, 1]$, whose distribution is given by

$$p(t \mid \mu, \kappa) \propto_t (1 - t^2)^{\frac{d-3}{2}} \exp(\kappa t). \tag{2.1}$$

An algorithm that generates samples from this distribution is shown and derived below.

procedure AngleGenerator(d, κ)
  $r_0 := t_0 := \sqrt{1 + \left(\frac{d-1}{2\kappa}\right)^2} - \frac{d-1}{2\kappa}$ ▷ $r_0 = t_0 \in (0, 1)$
  while True **do**
    $t \sim \mathrm{Beta}\left(\frac{d-1}{2}, \frac{d-1}{2}\right)$
    $t \leftarrow 2t + 1$
    $t \leftarrow \frac{r_0 + t}{1 + r_0 t}$
    Break the loop with probability $\frac{\exp(\kappa t)(1 - r_0 t)^{(d-1)}}{\exp(\kappa t_0)(1 - r_0 t_0)^{(d-1)}}$
  return t ▷ The output $t = \cos\theta$ follows Eq. 2.1

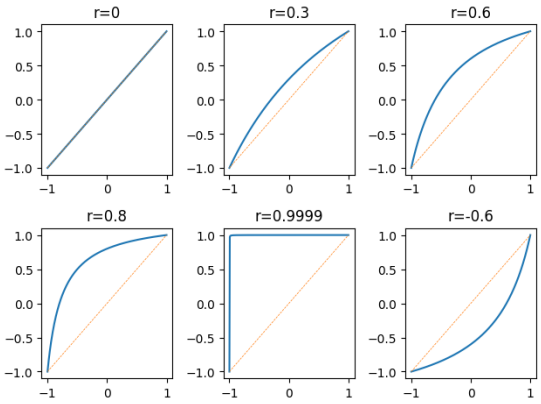

Figure 3.1: Mapping $t \mapsto \frac{r+t}{1+rt}$

3 Algorithm for the angle

In this section we explain the angle generator presented in the previous section. The reader might notice that this algorithm is not literally the one presented in [Woo94], but they are equivalent. Our added value is to explain explicitly the derivation of the algorithm, using a rich family of continuous transformations in $(-1, 1)$.

As a first approximation for Equation 2.1, if we ignore the exponential term, the equation corresponds to a Beta distribution shifted and scaled to $[-1, 1]$. Concretely, if $t_1 \sim \text{Beta}(\alpha, \beta)$, its density is $f_1(t) \propto_t t^{\alpha-1}(1-t)^{\beta-1}$, and if we let $t_2 := 2t_1 - 1$, its density can be obtained using the theory of change of variables as $f_2(t) = f_1(\frac{t+1}{2})(\frac{d}{dt}\frac{t+1}{2}) \propto_t t^{\alpha-1}(1-t)^{\beta-1}$. So, for $\alpha = \beta = \frac{d-1}{2}$, we have that t_2 follows $f_2(t) \propto_t (1-t^2)^{\frac{d-3}{2}}$.

In order to take the exponential factor into account, we will use an additional transformation for t_2. Since the exponential factor continuously gives more weight to inputs towards the right end $(+1)$, it is natural to consider mappings that shift the input continuously and bijectively, like those shown in Figure 3.1. One such type of candidate mappings is the family $\{h_r : r \in (-1, 1)\}$ of linear fractional transformations given by

$$t \mapsto h_r(t) := \frac{r+t}{1+rt}, \quad r \in (-1, 1)$$

This family has several mathematically rich properties. For instance, the inverse of h_r is h_{-r}, and if we compose h_{r_1} with h_{r_2}, we obtain again a function of the type h_r, where $r = h_{k_1}(k_2) = h_{k_2}(k_1)$.

Let, therefore, $t_3 := h_r(t_2)$ with unset parameter r, as it will be tuned later to maximize the acceptance rate. The density for t_3 is $f_3(r, t) = f_2(h_{-r}(t))\left(\frac{d}{dt}h_{-r}(t)\right)$ and can be simplified to

$$f_3(r,t) \propto_{t,r} \quad (1-r^2)^{\frac{d-1}{2}}(1-t^2)^{\frac{d-3}{2}}(1-rt)^{1-d} \tag{3.1}$$

$$\propto_t \quad (1-t^2)^{\frac{d-3}{2}}(1-rt)^{1-d}. \tag{3.2}$$

We will now develop the rejection sampling method for $p(t) := p(t \mid \mu, \kappa)$ (Eq. 2.1) using the envelope $f_3(r,t)$ with r tuned to maximize the acceptance rate. The theory of rejection sampling theory tells that a sample $t_3 \sim f_3(r,t)$ should be accepted with probability $\frac{p(t_3)}{M\, f_3(r,t_3)}$ where $M := \max_{t\in[-1,1]} \frac{p(t)}{f_3(r,t)}$. Thus, the acceptance rate is given by

$$\max_r \mathbb{E}_{t\sim f_3(r,t)}[\frac{p(t)}{M\, f_3(r,t)}] = \max_r \frac{1}{M} \int_{-1}^{+1} f_3(r,t) \frac{p(t)}{M\, f_3(r,t)} dt = \max_r \frac{1}{M}.$$

Equivalently, if $q(t)$ is any function proportional on t to $\frac{p(t)}{f_3(r,t)}$, then the sample t_3 should be accepted with probability $\frac{q(t)}{q(t_0)}$, where $t_0 := \arg\max_{t\in[-1,1]} q(t)$, and the acceptance rate maximization is equivalent to the optimization

$$\max_r \frac{1}{M} \equiv \min_r M \equiv \min_r q(t_0) = \min_r \max_t q(t) \equiv \min_r \max_t \log q(t).$$

Using Equations 3.1 and 2.1, we get $\log q(t) = \kappa t + (d-1)\log(1-rt) - \frac{d-1}{2}\log(1-r^2)$. For a fixed r, it can easily be seen that this is maximized in $t \in (-1,1)$ at $t_0 := 1 + \frac{d-1}{k}$, and $\log q(t_0)$ is minimized in $r \in (-1,1)$ at $r_0 := \sqrt{1 + \left(\frac{d-1}{2\kappa}\right)^2} - \frac{d-1}{2\kappa}$. Coincidentally, when substituting r_0 into the formula of t_0, it occurs that $r_0 = t_0$.

4 Algorithm

We propose to avoid the rotation step and to work entirely in the d-dimensional space. This means that the Muller method needs to be adjusted to produce vectors uniformly in $\{z \in S^{d-1} : z \perp \mu\}$ instead of $\{\xi \in S^{d-2}\}$. The Ulrich-Woods method for sampling θ, however, is left untouched, although it is vectorized to increase speed. The implementation is shown below.

```
import numpy as np

def random_VMF(mu, kappa, size=None):
    """
    Von Mises-Fisher distribution sampler with
    mean direction mu and concentration kappa.
    Source: https://hal.science/hal-04004568
    """
    # parse input parameters
    n = 1 if size is None else np.product(size)
    shape = () if size is None else tuple(np.ravel(size))
    mu = np.asarray(mu)
    mu = mu / np.linalg.norm(mu)
    (d,) = mu.shape
    # z component: radial samples perpendicular to mu
    z = np.random.normal(0, 1, (n, d))
    z /= np.linalg.norm(z, axis=1, keepdims=True)
    z = z - (z @ mu[:, None]) * mu[None, :]
    z /= np.linalg.norm(z, axis=1, keepdims=True)
```

```
    # sample angles (in cos and sin form)
    cos = random_VMF_angle(d, kappa, n)
    sin = np.sqrt(1 - cos**2)
    # combine angles with the z component
    x = z * sin[:, None] + cos[:, None] * mu[None, :]
    return x.reshape((*shape, d))


def random_VMF_angle(d: int, kappa: float, n: int):
    """
    Generate n iid samples t with density function given by
    p(t) = someConstant * (1-t**2)**((d-3)/2) * exp(kappa*t)
    """
    alpha = (d - 1) / 2
    t0 = r0 = np.sqrt(1 + (alpha / k) ** 2) - alpha / k
    log_t0 = k * t0 + (d - 1) * np.log(1 - r0 * t0)
    found = 0
    out = []
    while found < n:
        m = min(n, int((n - found) * 1.5))
        t = np.random.beta(alpha, alpha, m)
        t = 2 * t - 1
        t = (r0 + t) / (1 + r0 * t)
        log_acc = k * t + (d - 1) * np.log(1 - r0 * t) - log_t0
        t = t[np.random.random(m) < np.exp(log_acc)]
        out.append(t)
        found += len(out[-1])
    return np.concatenate(out)[:n]
```

The function `random_VMF` takes as input parameters a vector μ of shape `(d,)`, a floating point number κ, and an optional shape `size`, that follows the `numpy` ecosystem standards. More precisely, when `size` is not given, the function generates a single d-dimensional output vector; when it is an integer n, it generates n vectors and the output has shape (n, d); and when it is a shape tuple, say $(1, 100, 3)$, the output will have shape $(1, 100, 3, d)$.

The function `random_VMF_angle` is used by `random_VMF`. This function generates angles θ that follow Equation (2.1) and returns their cosine. It uses the Ulrich-Woods [Woo94] method with vectorization to accelerate the process. The number of loops executed by this algorithm (hence also the complexity) is stochastic, but it typically is 2 and rarely exceeds 4. This occurs because, the acceptance rate of the Ulrich-Woods method is always above 65.9% (worst case), and is higher for large d or small κ. As a heuristic to increase speed, we take n samples on the first loop, out which a fraction is possibly rejected, and for the next iteration, if there are n' more samples needed, we take $n' + 50\%$. This extra overhead increases significantly the chances of the algorithm of needing only one extra round, at the expense of possibly computing more samples than needed. The increase of 50% is not done in the first round to account for the case when the acceptance rate is high (large d or small κ), in which most of the extra samples in the first round will be unused. In the worst case in which the first round rejects more than $100\% - 65.9\% = 34\%$, the next round will use a sample size of around $65.9\% \times 150\% \approx 100\%$ of n. We acknowledge that our empirically fast heuristic can be replaced with an even faster more refined

heuristic that computes the optimal increase of samples, but probably at the cost of reducing simplicity and readability.

5 Experiments

We experimentally show that our implementation is faster while producing the same outputs as [ten23, Whi23], which are the existing sampling implementations from VMF distributions.

Table 1: Execution time comparison

		d=2	d=3	d=5	d=50
κ=5	[Whi23]	10.28ms	9.01ms	8.54ms	7.75ms
	[ten23]	68.53ms	8.55ms	41.84ms	27.75ms
	[Our]	0.60ms	0.48ms	0.55ms	1.11ms
κ=50	[Whi23]	10.39ms	10.03ms	10.21ms	9.74ms
	[ten23]	60.76ms	8.85ms	52.30ms	50.96ms
	[Our]	0.61ms	0.58ms	0.65ms	1.29ms

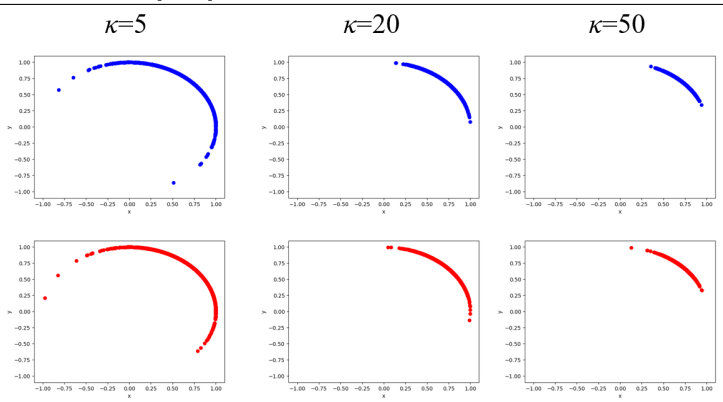

Figure 5.1: The figures in the first row are the outputs of sampling in 2-dimension from [Whi23] and the figures in the second row are the outputs of sampling in 2-dimension with our implementation.

Table 1 compares the execution time comparison of our implementation with the existing implementations [ten23, Whi23] when generating 1,000 samples. We compared the results while increasing the number of dimensions to 2, 3, 5, and 50 when kappa was 5 and 50, respectively. As can be seen from the results, the proposed implementation is more than 10 times faster than the existing implementations. The experimental environment is Intel Core i5-9400H CPU 2.50GHZ with 16GB ram and the OS is Windows.

Figure 5.2 shows the empirical histograms of the function `random_VMF_angle` compared with the theoretical distribution given by Equation 2.1. The picture, which was generated with the Python code below, illustrates clearly that the algorithm is faithful to the theoretical density curve.

```python
import numpy as np
import matplotlib.pyplot as plt

fig, Ax = plt.subplots(2, 3)
N = 100000
x = np.linspace(-1, 1, 1000)[1:-1]
dk = [(3, 3), (4, 1), (2, 5), (2, 1), (50, 1), (50, 150)]
for (d, k), ax in zip(dk, np.ravel(Ax)):
    X = random_VMF_angle(d, k, N)
    ax.hist(X, density=True, bins="auto")
    y = (1 - x**2) ** ((d - 3) / 2) * np.exp(k * x)
    y /= np.trapz(y=y, x=x)
    ax.plot(x, y)
    ax.set_title(f"(d,k)={(d,k)}")
plt.tight_layout()
plt.show()
```

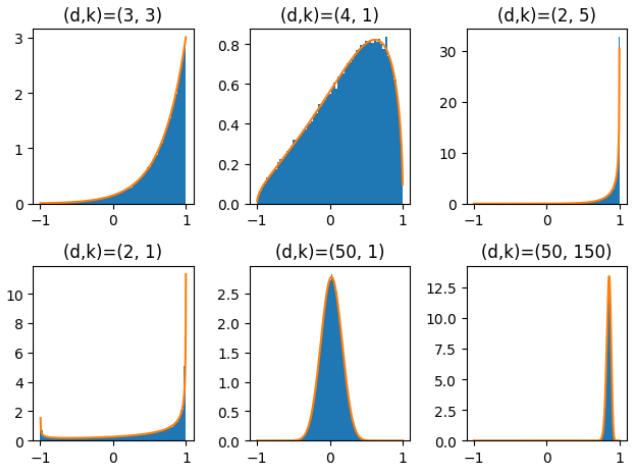

Figure 5.2: Empirical histograms of 100 000 generated samples (blue) vs theoretical density $p(t \mid \mu, \kappa)$ (orange) for different values of d and κ.

Furthermore, Figures 5.1, 5.3, and 5.4 visualize samples generated from [Whi23] and the proposed implementation. It can be concluded, visually, the distributions appear to be the indeed the same when the dimension is 2, 3, and 4, respectively.

6 Conclusion

Directional statistics are used in various fields such as animal navigation, tracking, image analysis, and applications in which the direction is more important than the magnitude, though, it has received less attention than statistics in Euclidean space. However, if the data are normalized to the unit norm and lie on the surface of S^{d-1}, directional statistics are more appropriate than Euclidean statistics.

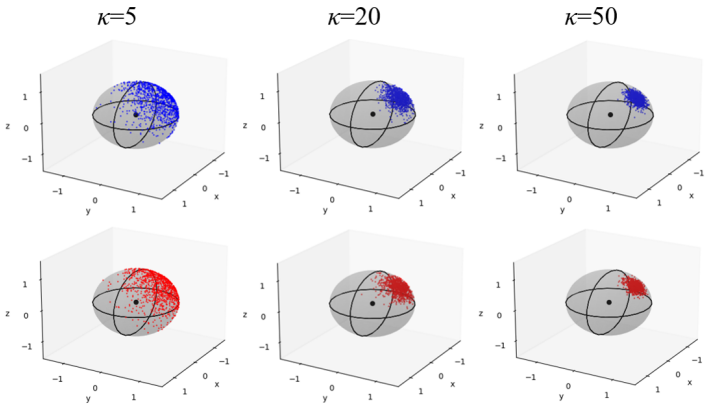

Figure 5.3: The figures in the first row are the outputs of sampling in 3-dimension from [Whi23] and the figures in the second row are the outputs of sampling in 3-dimension with our implementation.

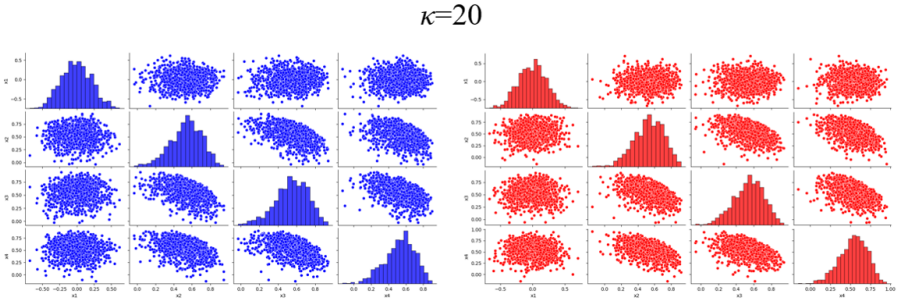

Figure 5.4: The left figures are the outputs of sampling in 4-dimension from [Whi23] and the right figures are the outputs of sampling in 4-dimension with our implementation.

The von Mises Fisher distribution (VMF) has been used as a probability distribution in directional statistics, but its implementation is not being provided as python libraries. We intend to contribute to the field where directional statics are needed by proposing an implementation that produces the same outputs as existing implementations but is more than 10 times faster.

Acknowledgements

This work was supported by Catuscia Palamidessi and the European Research Council (ERC) project HYPATIA under the European Union's Horizon 2020 research and innovation programme. Grant agreement n. 835294.

We also thank Niko Brummer for his advice during the development of this article.

References

[Bru22] Niko Brummer. Toroidal-psda github repository. https://github.com/bsxfan/Toroidal-PSDA/blob/cb44db3f57a1bd95aad492ff8f94b4c8dd747cca/tpsda/vmf_sampler.py#L17, 2022.

[CA20] Nicola De Cao and Wilker Aziz. The power spherical distribution, 2020.

[Fis53] Ronald Aylmer Fisher. Dispersion on a sphere. *Proceedings of the Royal Society of London. Series A. Mathematical and Physical Sciences*, 217(1130):295–305, 1953.

[Hou58] Alston S Householder. Unitary triangularization of a nonsymmetric matrix. *Journal of the ACM (JACM)*, 5(4):339–342, 1958.

[Mul59] Mervin E Muller. A note on a method for generating points uniformly on n-dimensional spheres. *Communications of the ACM*, 2(4):19–20, 1959.

[Num23] Numpy.org. Von mises. https://numpy.org/doc/stable/reference/random/generated/numpy.random.vonmises.html, 2023. Accessed: 2023-02-16.

[Sci23] Scipy.org. Von mises. https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.vonmises.html, 2023. Accessed: 2023-02-16.

[ten23] tensorflow.org. Von mises fisher. https://github.com/tensorflow/probability/blob/v0.19.0/tensorflow_probability/python/distributions/von_mises_fisher.py#L44-L519, 2023. Accessed: 2023-02-16.

[Ulr84] Gary Ulrich. Computer generation of distributions on the m-sphere. *Journal of the Royal Statistical Society: Series C (Applied Statistics)*, 33(2):158–163, 1984.

[vM81] Richard von Mises. Uber die" ganzzahligkeit" der atomgewicht und verwandte fragen. *Physikal. Z.*, 19:490–500, 1981.

[Whi23] Daniel Whittenbury. Von mises. `https://dlwhittenbury.github.io/ds-2-sampling-and-visualising-the-von-mises-fisher-distribution-in-p-dimensions.html`, 2023. Accessed: 2023-02-16.

[WK21] Benjamin Weggenmann and Florian Kerschbaum. Differential privacy for directional data. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 1205–1222, 2021.

[Woo94] Andrew TA Wood. Simulation of the von mises fisher distribution. *Communications in statistics-simulation and computation*, 23(1):157–164, 1994.