Numpyro Implementations of Univariate & Bivariate von Mises Distributions

Bioinformatics Project – Block 4

Teemu Rönkkö (pqh443)

June 15, 2020

Contents

1 Abstract

The goal of the current project was to implement the univariate and bivariate von Mises distributions in the probabilistic programming language Numpyro [2, 16] in Python 3 [21]. The alpha release of Numpyro uses JAX for automatic differentiation and JIT compilation to TPU, GPU & CPU. [2, 16] Due to its recency, many important statistical distributions and other important features are, however, still lacking. As both univariate and bivariate von Mises distributions are widely used in modelling the distribution of torsional angles ϕ and ψ of amino acid backbone chains, these implementations will have direct uses especially in protein structure prediction. Although the current report focuses on the use of circular statistics in bioinformatics, it should be noted that these distributions are also commonly used in e.g. physics & astronomy [12].

2 Introduction

The so-called *Holy Grail* problem in bioinformatics concerns predicting the 3-dimensional structure of protein solely based on its amino acid sequence [15]. Even though protein structure prediction can be approached using various methods ranging from (potential) energy-based (e.g. [11]) to knowledge-based (e.g. [22]) methods, these methods frequently share similar tools in the prediction task. The structure prediction task is often simplified by focusing only on the backbone chain of the protein of interest. All amino acids have similar basic structure: $N - C_\alpha - C - O$ atoms that form the backbone of the amino acid and a side chain R, that is different in all 20 of the genetically coded amino acids. The structure prediction is then simplified by focusing on predicting and modelling the torsional (also known as *dihedral*) angles ψ and ϕ. As depicted in figure 1, ϕ angle is the rotational angle around the bond between atoms N and C_α, whereas the ψ angle is the angle between C_α and C. [3] The figure also depicts ω angle, which resides between N and C atoms. However, this angle has been shown to have only two stereoisomers as this bond cannot freely rotate due to planarity of the peptide bond: The *cis* isomer with $\omega = 0°$ and *trans* isomer with $\omega = 180°$. [6] As the cis isomer is very rare in amino acid backbone chains, the modelling of the backbone chain is typically further simplified by focusing on ϕ and ψ angles, and assuming that $\omega = 180°$.

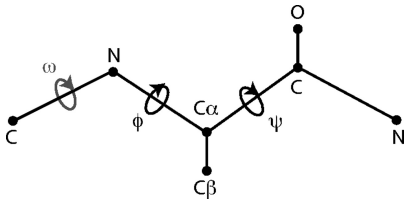

Figure 1: Illustration of a protein backbone chain. C_α signifies the chiral carbon atom, whereas C_β signifies carbon atom of a side chain [3]. However, if the amino acid is glycine, the C_α atom is not chiral and C_β is a hydrogen atom.

By considering the nature of these angles of interest, it is obvious that using conventional statistics is not the most fitting approach to this task, due to the recurring nature of angles. This means that, for example, the angles 2π rad and 4π rad are identical. Furthermore, many basic concepts used in conventional statistics, such as mean and variance, are not defined identically when dealing with circular data [12]. Thus, instead of trying to model the probability density functions of torsional angles using, for example, univariate or bivariate normal distributions that rely on conventional statistics, using circular statistics clearly fits the torsional angle prediction task better. Von Mises distribution is a so-called circular normal distribution that can be used to model the torsional angles in a more fitting way. The current project focuses on univariate and bivariate versions of

the von Mises distribution, but it should be noted that the von Mises distribution is also defined for higher dimensions. [12]

2.1 Univariate von Mises

The univariate von Mises distribution is a circular version of the normal distribution and it is one of the most central distributions in directional statistics. [1] The univariate von Mises distribution has a probability density function of the following form:

$$f(\theta) = \{2\pi I_0(\kappa)\}^{-1} e^{\kappa \cos(\theta - \mu)} \tag{1}$$

where $\kappa \geq 0$ is the concentration parameter and $-\pi < \mu \leq \pi$ is the mean direction. I_0 is the modified Bessel function of the first kind and order 0. When κ is large, the parameters μ and $\frac{1}{\kappa}$ correspond to the mean and variance in a conventional normal distribution and the distribution is concentrated at $\theta = \mu$. When $\kappa = 0$, the distribution is uniform. [13] The probability density function can be seen in figure 2.

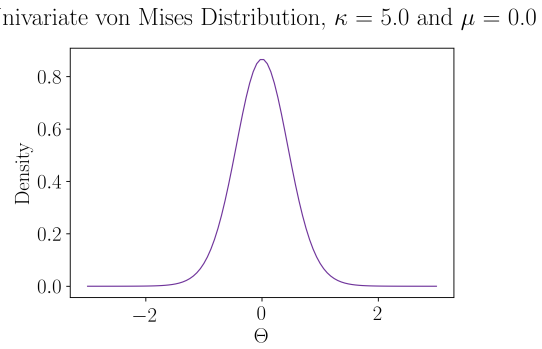

Figure 2: The probability density function of univariate von Mises distribution with $\kappa = 5.0$ and $\mu = 0.0$

2.2 Bivariate von Mises

When dealing with torsional angles, it is beneficial to model the torsional angles ϕ and ψ using a bivariate distribution that can model the two random variables simultaneously. Bivariate von Mises is thus more optimal choice for modeling torsional angles. This distribution in its original form has probability density function proportional to:

$$\begin{aligned} f(\phi, \psi) \propto \exp\{&\kappa_1 \cos(\phi - \mu) + \kappa_2 \cos(\psi - \nu) \\ &+ (\cos(\phi - \mu), \sin(\phi - \mu)) \mathbf{A} (\cos(\psi - \nu), \sin(\psi - \nu))^T\} \end{aligned} \tag{2}$$

where $\mathbf{A}$ is a two-by-two matrix which allows the angles ϕ and ψ to have dependence. [12] This original bivariate von Mises distribution is, however, overparametrised as it has 9 parameters, whereas the bivariate normal distribution has only 5 parameters. Thus, submodels have been developed based on the original, full bivariate von Mises that have only 6 parameters. [19] These submodels are derived from the full distribution by setting the off-diagonal elements in the matrix $\mathbf{A}$ to zero. Then, if the diagonal elements are denoted as $\alpha = \mathbf{A}_{1,1}$ and $\beta = \mathbf{A}_{2,2}$, different submodels are derived by changing these two parameters. In this way, we achieve the *sine* model, which is the focus of this project, by setting $\alpha = 0$ and $\beta = \lambda$. The sine model of bivariate von Mises has then a probability density function which is proportional to the following:

$$f_s(\phi, \psi) = C_s \exp\{\kappa_1 \cos(\phi - \mu) + \kappa_2 \cos(\psi - \nu) + \lambda \sin(\phi - \mu) \sin(\psi - \nu)\} \tag{3}$$

The constant C_s is a normalizing constant that ensures that the integral of the probability density function sums to one and in this way, makes the presented density function a

proper density. The inverse of the normalizing constant is defined as follows (originally derived by Rivest in 1988 [18]):

$$C_s^{-1} = 4\pi^2 \sum_{m=0}^{\infty} \binom{2m}{m} \left(\frac{\lambda^2}{4\kappa_1\kappa_2} \right)^m I_m(\kappa_1) I_m(\kappa_2) \tag{4}$$

Here, I_m denotes the modified Bessel function of the first kind and order m.

Thus, the bivariate von Mises distribution can model how the two torsional angles ϕ and ψ are distributed. Traditionally, the distribution of torsional angles is visualized as a Ramachandran plot, first introduced by Ramachandran in 1963. [17] An example of a Ramachandran plot can be seen in figure 3. If the left and right sides of this plot are brought together and the ends of this tube are then connected, the Ramachandran plot can be visualised as a torus. Now, ϕ angle represents the rotation about the axis that goes through the hole in the middle of the torus, whereas ψ represents rotation angle about the axis penetrating the torus tube. Figure 4 shows how ϕ and ψ angles from the Top 500 Database are distributed on a torus.

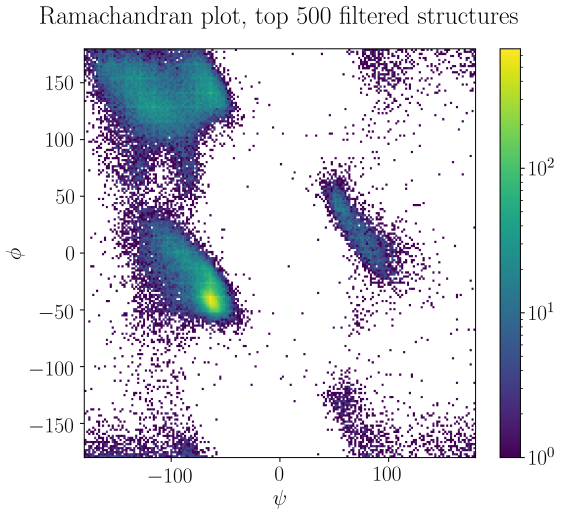

Figure 3: The figure shows the Ramachandran plot based on ϕ and ψ angles from Top 500 Database

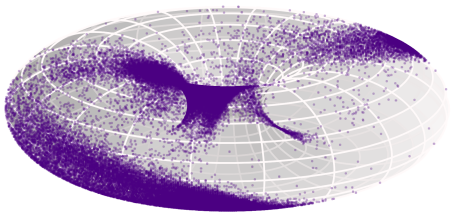

Figure 4: Torus plot based on ϕ and ψ angles from Top 500 Database

Other Submodels

As mentioned earlier, the sine model is not the only submodel derived from the original bivariate von Mises distribution. The other submodels include *cosine* models, with positive and negative interaction terms. In the models with positive interaction terms, the parameters are defined as $\alpha = \beta = -\kappa_3$, whereas in the models with negative interaction terms, the diagonal elements from $\mathbf{A}$ are set to $\alpha = \kappa_3$ and $\beta = -\kappa_3$. Additionally, the *hybrid* submodel has been derived, and it combines the sine and cosine models in order to avoid the distortions of the cosine models with positive and negative interactions that occur with certain values of κ_3. [9] Thus, the hybrid model transitions from an approximate

sine model with small correlations to cosine models when the correlation is large. The cosine and hybrid submodels are not further discussed in this report, as the sine model is the easiest to work with in practise. [12]

2.3 Numpyro

The goal of the current project was to implement both univariate and bivariate von Mises distributions in the probabilistic programming language Numpyro [2, 16]. Numpyro, as it name suggests, offers a NumPy backend for the functionalities offered in the probabilistic programming language Pyro, which relies on Pytorch backend. Even though both libraries have largely similar features and the semantics of, for example, the distribution modules are nearly identical, Numpyro has some features, such as the handling of random numbers, that are significantly different. Furthermore, Numpyro relies on Google's research project JAX for automatic differentiation and just-in-time (JIT) compilation using XLA (Accelerated Linear Algebra by TensorFlow, [10]), which is supported by all CPU, GPU, and TPU. By using these tools, Numpyro achieves significantly faster computations compared to Pyro. These architectural elements of Numpyro help speed up especially inference algorithms, namely, Hamiltonian Monte Carlo (HMC). [2, 16]

3 Methods & Results

Both the univariate and bivariate von Mises distributions were implemented in Numpyro in Python 3 [2, 16, 21]. The design of the implementations follows the structure of other distributions implemented in Numpyro. The distribution class objects in Numpyro all have two shape attributes: `batch_shape` and `event_shape`. The `batch_shape` denotes the independent distributions that are initialised, whereas the `event_shape` describes the shape of one independent draw from the distribution. For univariate von Mises, the `event_shape` was set to an empty tuple, as the samples from univariate von Mises are singular angles. Likewise, for bivariate von Mises, `event_shape` was set to `(2,)`, as the samples are always ϕ–ψ angle pairs. Both distribution classes were equipped with their own sampling methods and logarithmic probability density functions. In the following subsections, special attention is paid to the sampling methods that were implemented. Additionally, an `expand` function that can be used to initialize multiple identical, independent distributions was implemented for both univariate and bivariate cases. The full code for implementing the distribution classes and producing the plots can be found in the following GitHub repository: `https://github.com/teemuronkko/Bioinformatics_Project` and in the two Jupyter notebooks provided in the `code.zip` file.

3.1 Acceptance-Rejection Sampling

As acceptance-rejection sampling was used in simulating samples from both univariate and bivariate von Mises distributions, let's briefly consider this sampling algorithm. When one wants to sample from a distribution that does not have an implementation of simulation method of their own, acceptance-rejection algorithm can be used, with the help of an envelope distribution. [5] An envelope distribution is a probability density function (p.d.f.) that has an implemented sampling method and the p.d.f. has, preferably, very similar shape to the distribution from which the samples will be produced. However, the envelope distribution has to always satisfy the following condition:

$$f(x) \leq Mg(x) \quad \text{for all} \quad x \tag{5}$$

where $M > 1$ is a finite constant. [8] The algorithm then seeks to find samples that can be accepted as samples from the distribution of interest. These samples are generated from the envelope distribution in the following manner: [8, 5]

1. Firstly, X samples are simulated from the envelope distribution and W is simulated from an uniform distribution from range $[0, 1]$.

2. If $W < \frac{f(X)}{Mg(X)}$, the sample X is accepted, otherwise the procedure is repeated.

This procedure was used both in sampling from univariate von Mises distribution, using wrapped Cauchy distribution as the envelope distribution, and from bivariate von Mises distribution, using angular central Gaussian (ACG) distribution as the envelope distribution. [8]

3.2 Univariate von Mises

The univariate von Mises distribution was implemented by translating the Pytorch implementation of the distribution from the `torch.distributions` module to Numpyro. [14] As the Numpyro distribution class is based on the `torch.distributions` module, the translated von Mises distribution works practically identically to the Pytorch implementation. The full code for the implementation can be found in the Appendices, in the subsection Numpyro Implementation of Univariate von Mises Distribution. The major difference between the `torch` and `numpyro` implementations was that JAX JIT-compiled scripts do not support while-loops or `if ... else` statements in the same way as `@torch.jit.script` does. For this reason, the translation from Pytorch code to Numpyro often required different approaches to solve the same computations. The implementation supports sampling from multiple different independent distributions simultaneously, and the sample shape can vary from a single number (outputting a vector of angles) to a tuple containing the dimensions of the desired sample (outputting a multidimensional array, given the dimensions defined in the tuple).

Similarly to the `torch.distributions` implementation, the implemented univariate von Mises distribution used modified acceptance-rejection sampling, originally described in [1]. In this approach, the wrapped Cauchy distribution is used as the envelope distribution with such parameters that maximise the acceptance ratio. [12, 1] The efficiency of this algorithm was not tested in practise in the current report, as the original article reported high efficiency for this sampling approach. [1] The algorithm is defined as follows:

Univariate von Mises Sampling Algorithm

Set $\tau = 1 + \sqrt{1 + 4\kappa^2}$;
Set $\rho = \frac{\tau - \sqrt{2\tau}}{2\kappa}$;
Set $r = \frac{1+\rho^2}{2\rho}$;
repeat
  Sample u_1 and u_2 from uniform distribution within range $[0, 1]$;
  Set $z = \cos(u_1\pi)$;
  Set $f = \frac{1+rz}{1+z}$;
  Set $c = (r - f)k$
until $c(2 - c) - u_2 > 0$ ***or*** $\log(c/u_2) + 1 - c \geq 0$;
Sample u_3 from uniform distribution within range $[0, 1]$;
Set $\theta = \mu + \text{sign}(u_3 - 0.5) \arccos(f)$;
return θ

The figure 5 shows a million samples that were generated using this sampling algorithm from a univariate von Mises distribution with parameters $\kappa = 5.0$ and $\mu = 0.0$. Producing the samples took 0.71 seconds. As one can see on the plot, there is a strong correspondence between the actual p.d.f (plotted using `scipy.stats.vonmises`) and the sampled angles. The implementation of the logarithmic probability density was straightforward, and the

following equation was simply implemented as a function:

$$\log f(\theta) = -\log(2\pi) - \log(I_0(\kappa)) + \kappa\cos(\theta - \mu) \tag{6}$$

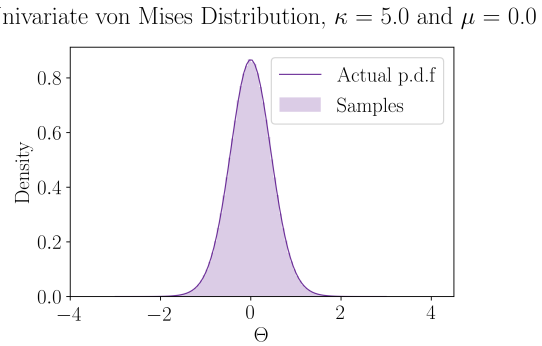

Figure 5: The figure shows the actual p.d.f of univariate von Mises against a million samples produced by the implemented sampling method. The purple line shows the actual probability density function, whereas the shaded purple area shows the histogram based on the sampled angles.

3.3 Bivariate von Mises

As the original Pytorch `torch.distributions` module does not include the implementation of bivariate von Mises distribution, the current implementation of bivariate von Mises distribution was based on the `R` package `simdd` which contained the implementation of the sampling method in `R` and Breinholt and Fischer-Rasmussen's Pytorch implementations [7, 4, 9]. As discussed earlier, the current project focuses only on the *sine* model of the bivariate von Mises. The full code for the implementation for the distribution can be found in the Appendices, in the subsection Numpyro Implementation of Bivariate von Mises Distribution. The current project focused only on unimodal bivariate von Mises distributions. The modality of the distribution can be checked as follows: If $\kappa_1\kappa_2 > \lambda^2$, the distribution is unimodal, and otherwise the distribution is bimodal. [12] The distribution class was then implemented so that this condition is checked when a distribution object is initialized, and a warning message is printed if the condition does not hold. The original implementation of bivariate von Mises in Pytorch by Fischer-Rasmussen (2019) and Breinholt (2020) also included a reparameterization parameter w that ensures that in case the distribution defined by the original parameters κ_1, κ_2 and λ does not satisfy the unimodality constraint. The motivation for this is to achieve higher efficiency for sampling by restricting the shape of the distribution to unimodal cases. [7, 4] The reparameterization parameter was also included in the current Numpyro implementation.

The sampling procedure for bivariate distributions is slightly more complex than it is for univariate distributions. When sampling from the bivariate von Mises distribution, the samples ψ' have to be first generated from the marginal distribution distribution $f(\psi)$. Then, the sampling is done from the conditional distribution $f(\phi|\psi = \psi')$. The sampling procedure for the marginal distribution was based on acceptance-rejection algorithm using the angular central Gaussian distribution (ACG) as the envelope distribution. [8] The samples from the conditional distribution were then produced by utilising the same sampling method that was used for the univariate von Mises distribution with parameters that are described later. [19] As the efficiency of the sampling method of interest has already been discussed both in the original article and in Fischer-Rasmussen (2019), this aspect of the sampling method is not investigated in the current report. [8, 7]

The marginal distribution for bivariate von Mises can be derived as the following

quantity: [12]

$$f_s(\theta) = C_s 2\pi I_0\left(\sqrt{k_2^2 + \lambda^2 \sin^2(\theta - \mu)}\right) \exp(k_1 \cos(\theta - \mu)) \tag{7}$$

This quantity is also referred to as the *Bessel* marginal density [8]. It can be shown that the Bessel marginal density can be bounded by the ACG density which is defined as follows:

$$f_{ACG}(x) = c_{ACG} f^*_{ACG}(x) \tag{8}$$

$$f^*_{ACG}(x) = (x^T \Omega x)^{-q/2} \tag{9}$$

$$c_{ACG} = |\Omega|^{1/2} \tag{10}$$

where Ω is a 2-by-2 symmetric, positive definite parameter matrix. The ACG distribution is used as the envelope distribution as the simulation method for this distribution is straightforward. [8] Firstly, let's consider a random variable y which follows a bivariate normal distribution $N(0, \Sigma)$ with positive definite parameter matrix Σ. Then, if we consider a random variable $x = \frac{y}{\|y\|}$, we know that this random variable follows ACG distribution with $\Omega = \Sigma^{-1}$. The full sampling method is described in Kent et al. (2008). [8] As mentioned earlier, the authors of the article provided the code for the simulation from the Bessel marginal density in R package `simdd`, which was used as the basis for the Numpyro implementation in the current project. The current implementation further assumes that the concentration parameters κ_1 and κ_2 are positive and that the distribution is unimodal, as when these conditions are satisfied, the distribution is similar to bivariate normal distribution with high concentration. [8]

As mentioned earlier, the sampling method for the conditional distribution used the same sampling algorithm as described in the previous section for univariate von Mises. The parameters that were used for sampling from the conditional distribution were described in Singh et al. (2002) as follows:

$$\kappa = a(\theta_1) = \sqrt{\kappa_2^2 + \lambda^2 \sin^2(\theta_1 - \mu_1)} \tag{11}$$

$$\mu = \mu_2 + \arctan((\lambda/\kappa_2)\sin(\theta_1 - \mu_1)) \tag{12}$$

where θ_1 is the angle that is sampled from the marginal distribution (in this case ψ'), and parameters κ_1, κ_2, λ, μ_1 and μ_2 are the parameters from the bivariate von Mises distribution from which the samples are produced. [19] It should be emphasised that the described sampling approach from the conditional distribution was implemented assuming that the unimodality condition $\lambda^2 < \kappa_1\kappa_2$ holds. [19]

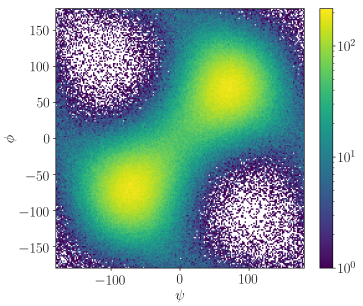

Figure 6: $\mu = \nu = 0$, $\kappa_1 = \kappa_2 = 1,\ \lambda = 3$

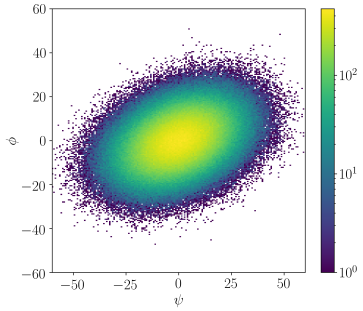

Figure 7: $\mu = \nu = 0$, $\kappa_1 = 20,\ \kappa_2 = 40,\ \lambda = 10$

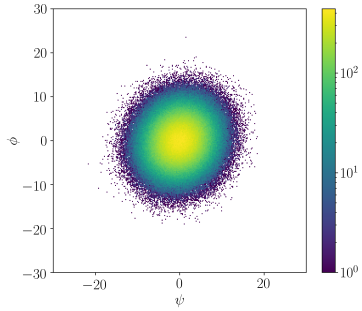

Figure 8: $\mu = \nu = 0$, $\kappa_1 = \kappa_2 = 200,\ \lambda = 20$

Figures 6, 7 and 8 show the results from sampling from three bivariate von Mises distributions with different parameters as Ramachandran plots. Note that the axes are in degrees and the axes are cropped to match the results better. Each plot shows a

million $\phi - \psi$ angle pairs. Figure 6 shows the results for a bimodal case of bivariate von Mises distribution, as $\kappa_1\kappa_2 = 1 < 3^2 = \lambda^2$. For the bimodal case, the run time of the sampling was significantly longer: Producing a million samples took 34.2 seconds, whereas for the distributions presented in figures 7 and 8, the sampling took 5.4 and 5.1 seconds respectively. This is due to the low efficiency for bimodal cases as the implemented sampling approach was originally designed for unimodal cases. [8]

Normalizing Constant

As mentioned in the Bivariate von Mises section, the probability density function of bivariate von Mises distribution includes a normalizing constant C_s. The normalizing constant includes $I_m(x)$, the modified Bessel function of the first kind and order m, which is not currently implemented in Python. However, parameter estimation for $\log I_m(x)$ has been studied and efficient methods for approximating this function have been derived. Tanabe et al. (2007) found that the logarithm of $I_m(x)$ can be approximated as follows:

$$\log I_m(x) \approx m \log(x/2) + \log(\hat{f}(x)) + \log\left(\sum_{s=0}^{N} \exp(\log f_s(x) - \log \hat{f}(x))\right) \tag{13}$$

where $\log f_s(x) - \log \hat{f}(x) > \sigma_0$ is small enough, such as $\sigma_0 = -100$ and $N = 3m$. [20] For simplicity, for all values of m, the implemented function used $N = 250$ terms in parameter estimation for all orders m of the function. [4, 7] As the normalizing constant was only directly used when calculating the logarithmic probability density for a given angle pair, the normalizing constant was computed in logarithmic space. This also helps the implementation to be more numerically stable as the normalizing constant includes large exponents that could easily lead to overflow. As the normalizing constant is a sum over infinite number of terms, in the current implementation it was chosen that only the 50 first terms were used to approximate the constant, following the design choices made by Fischer-Rasmussen (2019) and Breinholt (2020). [4, 7]

4 Discussion

As the performance of Numpyro implementations of univariate and bivariate von Mises distributions presented in this project was not yet investigated in practise, e.g. in a neural network, further studies will have to be conducted on the use of these distributions in e.g. protein structure prediction. The current implementation was merely briefly tested using a toy example neural network in order to find out whether the sampling method actually works as it is intended to. Both univariate and bivariate distributions did work when tested in a neural network, but due to lack of time, the performance of these distributions was not further investigated.

The implementation of bivariate von Mises distribution should also be extended so that it supports different sample shapes (e.g. multidimensional arrays) for sampling multiple distributions at the same time. Furthermore, the other submodels of bivariate von Mises distributions briefly mentioned in the report should be considered as well. The efficiency of the implemented distributions should also be investigated in more detail, as in this project, it was assumed that the acceptance-rejection algorithms used in the simulation of samples had high efficiency, as described in the original papers where these methods were described. [8, 1] Lastly, as the current implementation of the bivariate von Mises distribution is optimal only for the unimodal cases, the distribution class needs to be extended to include the bimodal distributions, too. In order to do this, new sampling approaches that are efficient for bimodal von Mises distributions need to be developed.

References

[1] D. Best and N. I. Fisher. Efficient simulation of the von mises distribution. *Journal of the Royal Statistical Society: Series C (Applied Statistics)*, 28(2):152–157, 1979.

[2] E. Bingham, J. P. Chen, M. Jankowiak, F. Obermeyer, N. Pradhan, T. Karaletsos, R. Singh, P. Szerlip, P. Horsfall, and N. D. Goodman. Pyro: Deep Universal Probabilistic Programming. *arXiv preprint arXiv:1810.09538*, 2018.

[3] W. Boomsma, K. V. Mardia, C. C. Taylor, J. Ferkinghoff-Borg, A. Krogh, and T. Hamelryck. A generative, probabilistic model of local protein structure. *Proceedings of the National Academy of Sciences*, 105(26):8932–8937, 2008.

[4] C. Breinholt. Bivariate von mises sine model and its sine skewed cousin part ii, 2020.

[5] G. Casella, C. P. Robert, M. T. Wells, et al. Generalized accept-reject sampling schemes. In *A Festschrift for Herman Rubin*, pages 342–347. Institute of Mathematical Statistics, 2004.

[6] P. Craveur, A. P. Joseph, P. Poulain, A. G. de Brevern, and J. Rebehmed. Cis–trans isomerization of omega dihedrals in proteins. *Amino acids*, 45(2):279–289, 2013.

[7] K. Fischer-Rasmussen. Implementation of the bivariate von mises distribution in pyro, 2019.

[8] J. T. Kent, A. M. Ganeiber, and K. V. Mardia. A new unified approach for the simulation of a wide class of directional distributions. *Journal of Computational and Graphical Statistics*, 27(2):291–301, 2018.

[9] J. T. Kent, K. V. Mardia, and C. C. Taylor. Modelling strategies for bivariate circular data. In *Proceedings of the Leeds Annual Statistical Research Conference, The Art and Science of Statistical Bioinformatics, Leeds University Press, Leeds*, pages 70–73, 2008.

[10] C. Leary and T. Wang. Xla: Tensorflow, compiled. *TensorFlow Dev Summit*, 2017.

[11] A. Liwo, J. Lee, D. R. Ripoll, J. Pillardy, and H. A. Scheraga. Protein structure prediction by global optimization of a potential energy function. *Proceedings of the National Academy of Sciences*, 96(10):5482–5485, 1999.

[12] K. V. Mardia and J. Frellsen. Statistics of bivariate von mises distributions. In *Bayesian Methods in Structural Bioinformatics*, pages 159–178. Springer, 2012.

[13] K. V. Mardia and P. E. Jupp. *Directional statistics*, volume 494. John Wiley & Sons, 2009.

[14] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

[15] G. A. Petsko. The grail problem. *Genome biology*, 1(1):comment002–1, 2000.

[16] D. Phan, N. Pradhan, and M. Jankowiak. Composable effects for flexible and accelerated probabilistic programming in numpyro. *arXiv preprint arXiv:1912.11554*, 2019.

[17] G. N. Ramachandran. Stereochemistry of polypeptide chain configurations. *J. Mol. Biol.*, 7:95–99, 1963.

[18] L.-P. Rivest. A distribution for dependent unit vectors. *Communications in Statistics-Theory and Methods*, 17(2):461–483, 1988.

[19] H. Singh, V. Hnizdo, and E. Demchuk. Probabilistic model for two dependent circular variables. *Biometrika*, 89(3):719–723, 2002.

[20] A. Tanabe, K. Fukumizu, S. Oba, T. Takenouchi, and S. Ishii. Parameter estimation for von mises–fisher distributions. *Computational Statistics*, 22(1):145–157, 2007.

[21] G. Van Rossum and F. L. Drake. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009.

[22] D. Xu and Y. Zhang. Ab initio protein structure assembly using continuous structure fragments and optimized knowledge-based force field. *Proteins: Structure, Function, and Bioinformatics*, 80(7):1715–1735, 2012.

5 Appendices

5.1 Numpyro Implementation of Univariate von Mises Distribution

```
import jax
import jax.numpy as np
import jax.random as random
from jax.scipy.special import gammaln, logsumexp
from jax import jit, lax
from numpyro.distributions import constraints
from numpyro.distributions.distribution import Distribution
from numpyro.util import copy_docs_from
from functools import partial
from numpyro.distributions.util import lazy_property, promote_shapes,
    validate_sample
import time
import warnings

"""The implementation of the univariate Von Mises distribution follows the
    implementation of the same
distribution in Pytorch torch.distributions package. The original code for
    the Pytorch implementation can be
found athttps://pytorch.org/docs/stable/_modules/torch/distributions/
    von_mises.html#VonMises"""

_I0_COEF_SMALL = np.array([1.0, 3.5156229, 3.0899424, 1.2067492, 0.2659732,
     0.360768e-1, 0.45813e-2])
_I0_COEF_LARGE = np.array([0.39894228, 0.1328592e-1, 0.225319e-2, -0.157565
    e-2, 0.916281e-2,
                  -0.2057706e-1, 0.2635537e-1, -0.1647633e-1, 0.392377e-2])
_I1_COEF_SMALL = np.array([0.5, 0.87890594, 0.51498869, 0.15084934,
    0.2658733e-1, 0.301532e-2, 0.32411e-3])
_I1_COEF_LARGE = np.array([0.39894228, -0.3988024e-1, -0.362018e-2,
    0.163801e-2, -0.1031555e-1,
                  0.2282967e-1, -0.2895312e-1, 0.1787654e-1, -0.420059e-2])

@jit
def _log_modified_bessel_fn(x, order):
    """
    Returns ``log(I_order(x))`` for ``x > 0``,
    where `order` is either 0 or 1.

    Based on https://pytorch.org/docs/stable/_modules/torch/distributions/
    von_mises.html#VonMises
    """
    # compute small solution
    y = (x / 3.75)
```

```
    y = y * y

    COEF_SMALL = np.where(np.ones((7,))*order, _I1_COEF_SMALL,
    _I0_COEF_SMALL)
    COEF_LARGE = np.where(np.ones((9,))*order, _I1_COEF_LARGE,
    _I0_COEF_LARGE)

    small = _eval_poly_small(y, COEF_SMALL)
    small = np.where(np.ones(x.shape)*order, abs(x) * small, small)
    small = np.log(small)

    # compute large solution
    y = 3.75 / x
    large = x - 0.5 * np.log(x) + np.log(_eval_poly_large(y, COEF_LARGE))

    result = np.where(x < 3.75, small, large)
    return result

@jit
def _eval_poly_small(y, coef):
    return coef[-7] + y*(coef[-6] + y*(coef[-5] + y*(coef[-4] + y*(coef[-3]
     + y*(coef[-2] + y*coef[-1])))))

@jit
def _eval_poly_large(y, coef):
    return coef[-9] + y*(coef[-8] + y*(coef[-7] + y*( coef[-6] +
                         y*(coef[-5] + y*(coef[-4] + y*(coef[-3] + y*(coef[-
    2] + y*coef[-1])))))))

def condition(args):
    return ~np.all(args[2])

def loop(args):
    x, proposal_r, done, key, concentration = args
    key, subkey = jax.random.split(key)
    u = jax.random.uniform(subkey, shape = (3,) + x.shape)
    u1, u2, u3 = u.squeeze()
    z = np.cos(np.pi * u1)
    f = (1 + proposal_r * z) / (proposal_r + z)
    c = concentration * (proposal_r - f)
    accept = ((c * (2 - c) - u2) > 0) | (np.log(c / u2) + 1 - c >= 0)
    x = np.where(accept, np.sign(u3 - 0.5) * np.arccos(f),  x)
    done = done | accept
    return x, proposal_r, done, key, concentration

@jit
def _rejection_sample(loc, concentration, proposal_r, key, x):
    """
    Acceptance-rejection sampling method     translated from the Pytorch
    univariate von Mises implementation.

    The sampling algorithm for the von Mises distribution is based on the
    following paper:
    Best, D. J., and Nicholas I. Fisher.
    "Efficient simulation of the von Mises distribution." Applied
    Statistics (1979): 152-157.
    """
    done = np.zeros(x.shape, dtype=bool)
    x = lax.while_loop(condition, loop, (x, proposal_r, done, key,
    concentration))[0]
    return (x + np.pi + loc) % (2 * np.pi) - np.pi

@copy_docs_from(Distribution)
class VonMises(Distribution):
    """
    A circular von Mises distribution.
```

```
    This implementation uses polar coordinates. The ``loc`` and ``value``
args
    can be any real number (to facilitate unconstrained optimization), but
are
    interpreted as angles modulo 2 pi.

    :param int or ndarray: an angle in radians.
    :param int ord ndarray: concentration parameter

    Based on https://pytorch.org/docs/stable/_modules/torch/distributions/
von_mises.html#VonMises
    """
    arg_constraints = {'loc': constraints.real, 'concentration':
constraints.positive}
    support = constraints.real
    has_rsample = False

    def __repr__(self):
        return str(type(self).__name__) + "(loc: " + str(self.loc) + ",
concentration: " + str(self.concentration) + ")"

    def __init__(self, loc, concentration, validate_args=None):
        self.loc, self.concentration = promote_shapes(loc, concentration)
        batch_shape = lax.broadcast_shapes(np.shape(loc), np.shape(
concentration))
        event_shape = ()
        tau = 1 + np.sqrt(1 + 4 * self.concentration ** 2)
        rho = (tau - np.sqrt(2 * tau)) / (2 * self.concentration)
        self._proposal_r = (1 + rho ** 2) / (2 * rho)
        super(VonMises, self).__init__(batch_shape, event_shape,
validate_args)

    def log_prob(self, value):
        log_prob = self.concentration * np.cos(value - self.loc)
        log_prob = log_prob - np.log(2 * np.pi) - _log_modified_bessel_fn(
self.concentration, 0)
        return log_prob.T

    def sample(self, key, sample_shape = ()):
        """
        The sampling algorithm for the von Mises distribution is based on
the following paper:
        Best, D. J., and Nicholas I. Fisher.
        "Efficient simulation of the von Mises distribution." Applied
Statistics (1979): 152-157.

        Based on https://pytorch.org/docs/stable/_modules/torch/
distributions/von_mises.html#VonMises
        """
        if isinstance(sample_shape, int):
            shape = tuple([sample_shape]) + self.batch_shape + self.
event_shape
        else:
            shape = sample_shape + self.batch_shape + self.event_shape
        x = np.empty(shape)
        return _rejection_sample(self.loc, self.concentration, self.
_proposal_r, key, x)

    def expand(self, batch_shape):
        """
        Function to initialize batch_shape number of parallel distributions
        """
        validate_args = self.__dict__.get('_validate_args')
        loc = np.ones(batch_shape)*self.loc
        concentration = np.ones(batch_shape)*self.concentration
```

```
        return VonMises(loc, concentration, validate_args=validate_args)

    @property
    def mean(self):
        """
        The provided mean is the circular one.
        """
        return self.loc

    @lazy_property
    def variance(self):
        """
        The provided variance is the circular one.
        """
        return 1 - np.exp(_log_modified_bessel_fn(self.concentration, 1) -
                          _log_modified_bessel_fn(self.concentration, 0))
```

5.2 Numpyro Implementation of Bivariate von Mises Distribution

```
@partial(jit, static_argnums = (0,1,))
def _acg_bound(sample, k1, k2, alpha, key):
    lam = np.concatenate((np.zeros([len(k1), 1]), 0.5*(k1 - alpha**2/k2).
    reshape(len(k1), 1)), axis = 1)
    lambda_min = np.min(lam, axis = 1).reshape([len(k1), 1])
    lam = lam - lambda_min
    b_values = np.concatenate((np.array(np.sqrt(lam[:,1]**2 + 1) - lam[:,1]
                                        + 1).reshape([len(k1), 1]), np.ones
    ([len(k1),1])*2), axis = 1)
    b0 = np.min(b_values, axis = 1).reshape([len(k1), 1])

    phi = 1 + 2*lam/b0
    den = _log_modified_bessel_fn(k2, 0)

    accept_shape = sample.shape[:len(sample.shape)-1] + (1,)
    accept = np.zeros(accept_shape, dtype = "bool_")
    count = 0
    args = (key, phi, k1, k2, lam, lambda_min, sample, accept, den, b0,
    alpha)
    res = lax.while_loop(loop_condition, loop_acg, args)[6]

    return np.arctan2(res[...,1], res[...,0])

def loop_acg(args):
    key, phi, k1, k2, lam, lambda_min, sample, accept, den, b0, alpha =
    args
    key, subkey = jax.random.split(key)
    x = np.where(accept, 0,  jax.random.normal(subkey, sample.shape)*np.
    sqrt(1/phi))
    r = np.sqrt(np.sum(x**2, axis = -1))
    r = np.expand_dims(r, axis=-1)
    x = x/r
    u = (x**2 * lam).sum(-1)
    v = jax.random.uniform(subkey, (sample.shape[0], k1.shape[0]))

    logf = k1*(x[...,0] - 1) + lambda_min.T + _log_modified_bessel_fn(np.
    sqrt(k2**2 + alpha**2 * x[...,1]**2), 0) - den
    loggi = 0.5 * (2 - b0.T) + np.log(1 + 2*u/b0.T) + np.log(b0.T/2)
    logfg = np.add(logf, loggi)
    logfg = logfg.reshape([sample.shape[0], k1.shape[0]])

    accept = v < np.exp(logfg)
    accept = accept[..., None]
    sample = np.where(accept, x,  sample)
```

```
    return (key, phi, k1, k2, lam, lambda_min, sample, accept, den, b0,
    alpha)

def loop_condition(args):
    return np.count_nonzero(np.isnan(args[6])) > 0

@partial(jit, static_argnums = (1,))
def log_im(order, k):
    """ x is a parameter, like k1 or k2
        Tanabe, A., Fukumizu, K., Oba, S., Takenouchi, T., & Ishii, S.
    (2007).
        Parameter estimation for von M i s e s Fisher distributions.
    Computational Statistics, 22(1), 145-157.

        The implementation is based on Christian Breinholt's Pytorch
    implementation.
    """
    k = k.reshape([len(k), 1, 1])
    s = np.arange(0 , 251).reshape(251, 1)
    fs = 2 * s * np.log(k/2) - gammaln(s + 1) - gammaln(order + s + 1)
    f_max = np.max(fs, axis = -2)
    k = k.reshape([len(k), 1])

    return (order * np.log(k/2) + f_max + logsumexp(fs - f_max[:,None], -2)
    ).squeeze()


@partial(jit, static_argnums = (0,))
def log_C(k1, k2, lam):

    """Harshinder Singh, Vladimir Hnizdo, and Eugene Demchuk
    Probabilistic model for twodependent circular variables.
    Biometrika, 89(3):719  723  , 2002.

    Closed form expression of the normalizing constant
    Vectorized and in log-space

    k1, k2 & lam are the parameters from the bivariate von Mises

    Since the closed expression is an infinite sum, 'terms' is the number
    of terms, over which the expression is summed over. Estimation by
    convergence.

    The code is translated to Numpyro from Christian Breinholt's Pytorch
    implementation."""

    terms = 51
    lam = np.abs(lam) + 1e-12
    m = np.arange(0, terms)
    log_binom = gammaln(2*m+1) - 2*gammaln(m+1)

    logC = log_binom*np.ones((len(k1), terms)) + m*np.log((lam**2)/(4*k1*k2
    ))[:,None] + log_im(m, k1) + log_im(m, k2)

    return - np.log(4*np.pi**2) - logsumexp(logC, axis = -1)

@copy_docs_from(Distribution)
class BivariateVonMises(Distribution):
    """
    Bivariate von Mises (Sine Model) distribution on the torus

    The distribution should be only used for unimodal cases. A warning
    message is printed if the distribution
    is bimodal. The distribtion is the bimodal if lam**2 > k1*k2.

    :param numpy.ndarray mu, nu: an angle in radians
```

```
    :param numpy.ndarray k1, k2 > 0: concentration parameters
    :param numpy.ndarray lam: correlation parameter
    :param numpy.ndarray w: reparameterization parameter     within range [
-1, 1]
    """
    arg_constraints = {'mu':  constraints.real,
                       'nu':  constraints.real,
                       'k1':  constraints.positive,
                       'k2':  constraints.positive,
                       'lam': constraints.real}
    support = constraints.real
    has_rsample = False

    def __repr__(self):
        param = self.lam
        printing = ", lam: "
        if self.lam is None:
            param = self.w
            printing = ", w: "
        return str(type(self).__name__) + "(mu: " + str(self.mu) + ", nu: "
     + str(self.nu) + ", k1: " + str(self.k1) + ", k2: " + str(self.k2) +
printing + str(param) + ")"

    def __init__(self, mu, nu, k1, k2, lam = None, w = None, validate_args
= None):

        if lam is None == w is None:
            raise ValueError("Either `lam` or `w` must be specified, but
not both.")
        elif w is None:
            self.mu, self.nu, self.k1, self.k2, self.lam = promote_shapes(
mu, nu, k1, k2, lam)

        elif lam is None:
            self.mu, self.nu, self.k1, self.k2, self.w = promote_shapes(mu,
 nu, k1, k2, w)
            self.lam = np.sqrt(self.k1*self.k2) * self.w

        if not np.all(self.lam**2 <= self.k1*self.k2):
             warnings.warn("The joint density is bimodal. The sampling
method is not optimal for bimodal distributions.")

        batch_shape = lax.broadcast_shapes(np.shape(self.mu),
                                           np.shape(self.nu),
                                           np.shape(self.k1),
                                           np.shape(self.k2),
                                           np.shape(self.lam))
        event_shape = (2,)

        self.logC = log_C(self.k1, self.k2, self.lam)

        super(BivariateVonMises, self).__init__(batch_shape, event_shape,
validate_args)

    def sample(self, key, sample_shape = ()):
        """ Harshinder Singh, Vladimir Hnizdo, and Eugene Demchuk
            Probabilistic model for twodependent circular variables.
            Biometrika, 89(3):719  723  , 2002.

        The sampling from marginal distribution is done using acceptance-
rejection sampling with angular central
        Gaussian (ACG) distribution as the envelope distribution. The
sampling from conditional distribution is done
        using acceptance-rejection sampling using wrapped Cauchy
distribution as the envelope distribution with
```

```
    parameters (nu + arctan((lam/k2) * sin(marg - mu))) where marg is
the sampled angle from marginal distribution.
    cond: conditional distribution using a modified univariate von
Mises (as described in Singh et al. (2002))

    The sampling method follows Christian Breinholt's Pytorch
implementation.
    """
    if sample_shape == 1: sample_shape = ()
    if isinstance(sample_shape, int):
        shape = tuple([sample_shape]) + self.batch_shape + self.
event_shape
    elif sample_shape == ():
        shape = (1,) + self.batch_shape + self.event_shape
    else:
        shape = sample_shape + self.batch_shape + self.event_shape

    x = np.empty(shape)*np.nan

    marg = _acg_bound(x, self.k1, self.k2, self.lam, key)
  #Sampling from marginal distribution
    marg = (marg + self.mu + np.pi) % (2 * np.pi) - np.pi
    marg = np.squeeze(marg)

    alpha = np.sqrt(self.k2**2 + self.lam**2 * np.sin(marg - self.mu)**
2) #Sampling from conditional distribution
    beta  = np.arctan(self.lam / self.k2 * np.sin(marg - self.mu))
    cond  = VonMises(self.nu + beta, alpha).sample(key)

    if len(self.k1) == 1 and sample_shape == ():
        marg = np.array([marg])

    return np.array([marg, cond]).T

def expand(self, batch_shape):
    validate_args = self.__dict__.get('_validate_args')
    mu = np.ones(batch_shape)*self.mu
    nu = np.ones(batch_shape)*self.nu
    k1 = np.ones(batch_shape)*self.k1
    k2 = np.ones(batch_shape)*self.k2

    if self.lam is not None:
        lam = np.ones(batch_shape)*self.lam
        w = None
    else:
        w = np.ones(batch_shape)*self.w
        lam = None

    return BivariateVonMises(mu, nu, k1, k2, lam = lam, w = w,
validate_args=validate_args)

@validate_sample
def log_prob(self, angles):
    """ Actual likelihood function, log joint distribution of phi and
psi.
    The code was translated from Christian Breinholt's Pytorch
implementation"""

    phi = angles[...,0]
    psi = angles[...,1]

    log_prob = self.k1*np.cos(phi - self.mu) + self.k2*np.cos(psi -
self.nu)
    log_prob += self.lam*np.sin(phi - self.mu)*np.sin(psi - self.nu)
    log_prob += self.logC
    return log_prob
```