Bayesian Inference

wordcount : 2955

Abstract

1. Bayesian modeling is a powerful paradigm in modern statistics and machine learning, offering a principled framework for inference under uncertainty. However, practitioners face significant obstacles, including **interoperability** issues, a persistent **accessibility-flexibility trade-off**, the limitations of **domain-specific limitations**, and challenges in **scalability**.

2. ***Interoperability:*** The landscape of Bayesian software is fragmented across programming languages and abstraction levels. Newcomers often gravitate towards high-level interfaces (e.g., *brms*) within familiar environments due to their accessibility for standard models. However, highgh levels of abstraction frameworks can be restrictive, lacking the flexibility needed for custom or complex models as research needs evolve.

3. ***Accessibility-Flexibility Trade-off:*** To gain the necessary flexibility, researchers must often transition to lower-level probabilistic programming languages . This transition imposes a steeper learning curve and requires mastery of specific modeling languages or complex programming frameworks, hindering broader adoption and rapid iteration.

4. ***Domain-Specific Limitations:*** Similar accessibility and flexibility trade-offs exist in domain-specific Bayesian packages. While providing accessible, pre-packaged models for specific fields, customizing or extending these models often requires deep engagement with lower-level programming languages or switching tools entirely, limiting methodological innovation within those domains.

5. ***Scalability:*** Computational demands remain a significant bottleneck, limiting the application of Bayesian methods to the large datasets and complex, high-dimensional models prevalent in modern research.

6. To address these challenges, we introduce ***Bayesian Inference (BI)***, a new Bayesian modeling software available in both Python and R. It aims to unify the modeling experience by integrating an intuitive model-building syntax (enhancing **accessibility**) with the **flexibility** of low-level abstraction coding available but also pre-build function for high-level of abstraction and including hardware-accelerated computation via JAX for improved **scalability**. Its availability in both major data science languages directly tackles the **interoperability** barrier and the prebuild function for specialized model in network analysis, survial models and phylogenetic analysis allow to improved **domain-specific limitations**.

Introduction

Bayesian modeling has emerged as a vital tool in modern statistics and machine learning, providing a framework for robust inference under uncertainty and the possibility to integrate prior knowledge. Despite its potential, the practical application of Bayesian methods is often hindered by significant hurdles within the current software ecosystem, preventing researchers from fully leveraging its capabilities. Key challenges stem from the fragmented nature of software across different programming languages (**interoperability**), gaps between theoretical understanding and practical implementation (**accessibility**), complexities in model specification that force trade-offs between ease-of-use and flexibility (**accessibility-flexibility trade-off**), the constraints of overly specialized tools (**domain-specific limitations**), and persistent computational scalability limitations for complex models or large datasets (**scalability**).

The first major obstacle is the fragmented landscape of Bayesian software, scattered across different programming languages and varying levels of abstraction, posing significant **interoperability** challenges. Researchers frequently encounter a disparate collection of tools—from *Stan*'s domain-specific language (DSL) to distinct low-level of abstraction libraries (like *PyMC* (Salvatier, Wiecki, and Fonnesbeck 2016), *TensorFlow Probability (TFP)* (Abadi et al. 2015), *NumPyro* (Phan, Pradhan, and Jankowiak 2019)) and high-level of abstraction libraries (like *BRMS*). This fragmentation complicates workflows and presents a confusing landscape, especially for researchers new to Bayesian analysis. For instance, researchers new to Bayesian analysis may initially gravitate towards tools of high-level of abstraction available within their most familiar programming environment (e.g., *BRMS* (Bürkner 2017) in *R* (Wickham 2015)), potentially overlooking more suitable options elsewhere due to the steep initial learning curve or perceived incompatibility (accessibility). This linguistic and platform diversity imposes considerable cognitive overhead, potentially hindering the adoption of the most suitable tool for a given problem due to familiarity biases or the friction of switching ecosystems, ultimately impacting the effective application of Bayesian methods. This initial hurdle of navigating disparate systems naturally leads new practitioners to prioritize tools that appear easiest to learn, raising concerns about the balance between accessibility and the flexibility needed for complex research.

Compounding this fragmentation is the challenge of accessibility and the translation of theoretical knowledge into practice **accessibility-flexibility trade-off**. Indeed, while high-level interfaces like *BRMS* offer an intuitive formula-based syntax, significantly lowering the initial barrier to entry (e.g. generalized linear mixed models using *BRMS*), this accessibility often comes at the cost of flexibility. As research questions become more sophisticated, requiring custom likelihood functions (e.g., multiple likelihoods), intricate prior structures (e.g., XXX), or non-standard model components (e.g., centered-random factors), the limitations of these high-level wrappers become apparent. To gain the necessary expressive power, the researcher must typically transition to lower-level probabilistic programming languages (PPLs) such as *Stan* (Stan Development Team) (requiring mastery of its specific DSL), *PyMC*, *NumPyro*, or *TFP*. This transition imposes a much steeper learning curve, demanding a deeper understanding of probabilistic programming concepts (like computational graphs or tensor manipulation) and often more verbose code. This significant jump in complexity can deter users, divert focus from statistical modeling to software engineering challenges, and ultimately slow down the pace of research, particularly when trying to adapt models within specific scientific fields.

Similar accessibility and flexibility constraints manifest as **domain-specific limitations** within specialized Bayesian packages. Fields like phylogenetics or network analysis benefit from tools such as *BEAST* (Bouckaert 2019), *RevBayes* (Höhna et al. 2016), *STRAND* (Ross, McElreath, and Redhead 2024), or *BISON* (Hart et al. 2023), which provide accessible, pre-packaged models tailored to common domain problems. A phylogeneticist might initially find *BEAST* convenient for standard molecular clock models. However, when they wish to incorporate a novel evolutionary hypothesis requiring modification of the core model structure or integrate data types not originally envisioned by the developers, they often encounter rigid constraints. Extending these specialized tools frequently requires deep engagement with their underlying, often complex, codebase (sometimes necessitating proficiency in languages like Java or C++) or abandoning the domain-specific tool entirely in favor of a general-purpose PPL. This forces researchers to either compromise on their methodological innovation or undertake a significant software development effort, potentially switching programming ecosystems and losing the initial convenience, thereby limiting the evolution of modeling practices within specialized domains. Even when model specification is achievable, either in general or specialized tools, the computational feasibility remains a major concern.

Finally, computational **scalability** continues to be a significant bottleneck, limiting the application of Bayesian methods to the large datasets (e.g., millions of observations) and complex, high-dimensional models (e.g., thousands of parameters) prevalent in modern research across fields like genomics, neuroscience, and machine learning. While established tools like *Stan* feature highly optimized inference algorithms (particularly its NUTS sampler) and offer effective multi-core parallelization, they can still face challenges with long C++ compilation times for complex models and may require substantial code restructuring or external tooling to efficiently leverage hardware accelerators like GPUs or TPUs for certain computations. Conversely, emerging frameworks built on *JAX* (Bradbury et al. 2018) (powering *NumPyro* and parts of *TFP*) promise substantial speedups via automatic differentiation, JIT compilation, and native support for parallel hardware architectures. However, integrating these powerful backends seamlessly into user-friendly, flexible modeling front-ends that don't require deep expertise in the JAX ecosystem itself is an ongoing challenge. Domain-specific tools often inherit the scalability limitations of the frameworks they are built upon, failing to provide a universally efficient solution across different model types and data sizes.

Therefore, there is an evident and pressing demand for a Bayesian modeling framework that synergistically addresses these interconnected limitations. To address these interconnected challenges, we introduce ***Bayesian Inference (BI)***, a new Bayesian modeling software designed to unify the modeling experience across the two dominant data science languages, Python and R. *BI* tackles the **interoperability** barrier head-on by offering native interfaces in both environments. It aims to resolve the **accessibility-flexibility trade-off** by providing an intuitive model-building syntax familiar to users of statistical modeling languages, while enabling advanced customization and leveraging multiple, interchangeable inference backends for flexibility. To combat **domain-specific limitations**, *BI* includes pre-built functions and structures tailored for specialized models in areas like network analysis, survival analysis, and phylogenetic analysis, while still allowing extension and modification within its general framework. Crucially, *BI* enhances **scalability** by integrating with hardware-accelerated computation via *JAX* (using *NumPyro* or *TFP* as backends), enabling efficient execution on CPUs, GPUs, and TPUs. By providing a streamlined, efficient, and unified environment for the end-to-end Bayesian workflow—from model specification and fitting to diagnostics and prediction—***BI*** lowers the barrier to entry for sophisticated Bayesian modeling, aiming to empower a broader community of researchers across disciplines to confidently apply advanced Bayesian methods to their complex research problems.

Software Presentation

BI directly confronts the **interoperability** challenge by offering native, feature-equivalent implementations in both Python and R. While minor syntactic differences exist to adhere to the idiomatic conventions of each language, the core model specification syntax, the procedural workflow for analysis, and the underlying computational engines remain fundamentally consistent. For instance, Python utilizes dot notation for method calls on class objects (e.g., `bi.dist.normal(0,1)`), while R employs dollar sign notation for accessing elements or methods within its object system (e.g., `bi$dist$normal(0,1)`). This dual-language availability significantly lowers the adoption barrier for researchers, allowing them to work entirely within their preferred programming environment without sacrificing access to a common, powerful Bayesian modeling framework.

BI is designed to navigate the critical *accessibility-flexibility trade-off* by providing multiple layers of abstraction and utility, catering effectively to users with varying levels of Bayesian modeling expertise and diverse complexity requirements throuhg : simplified backend interaction via intuitive syntax, pre-built components for complex model features , addressing domain-specific limitations within a general framework, integrated End-to-End Workflow and extensive model library and documentation.

At its computational core, *BI* leverages the power and efficiency of established Probabilistic Programming Languages (PPLs) like *NumPyro* and *TFP*, both of which are built upon the *JAX* framework for high-performance numerical computation and automatic differentiation. However, *BI* deliberately abstracts away much of the inherent complexity of these lower-level tools (**Code block 1**). This significantly enhances **accessibility** for a broader range of users.

Code block 1: *Prior specification differences between NumPyro, TFP, and BI*

```
# NumPyro prior specification
numpyro.sample("mu", dist.Normal(0, 1)).expand([10])

# TFP prior specification (within a JointDistributionCoroutine)
yield Root(tfd.Sample(tfd.Normal(loc=1.0, scale=1.0), sample_shape=10))

# BI prior specification
bi.dist.normal(0, 1, name = "mu", shape = (10,))
```

To enhance **flexibility** without unduly sacrificing the accessibility provided by the high-level syntax, *BI* includes a library of pre-built, computationally optimized functions implemented directly in *JAX* (e.g., **Code block 2**). These components encapsulate common but potentially complex modeling structures, allowing users to incorporate them easily within the model specification. Key examples include:

1. *Centered Random Effects* and *Non-Centered Random Effects* for hierarchical (multi-level) model components (McElreath 2018). The non-centered parameterization, often crucial for efficient sampling in hierarchical models (particularly with sparse data),

is provided without requiring the user to manually implement the reparameterization logic.

2. *Kernels for Gaussian Processes* for modeling spatial, temporal, phylogenetic, or other forms of structured correlation or dependency.

3. *Block Model Effects* for implementing stochastic block models in network analysis.

4. *SRM effects* for modeling pairwise interactions in networks while accounting for sender effects, receiver effects, dyadic effects, nodal predictors, dyadic predictors, and observation biases (Sosa et al., n.d.).

5. *Network-Based Diffusion Approach (NBDA)* components for modeling the effect of network edges on the rates of transmission of phenomena (e.g., behavioral, epidemiological) while accounting for nodal or dyadic covariates.

6. *Network metrics* ranging from nodal, dyadic, and global network measures with a total of 11 that can be used to build custom models of social network analysis (Sosa, Sueur, and Puga-Gonzalez 2020).

These pre-built *JAX* functions provide tailored model components for common patterns in specific fields, while keeping them fully integrated within the general, extensible modeling framework. By providing these optimized building blocks within its general syntax, *BI* allows researchers in these fields to rapidly implement standard domain models using familiar concepts. Crucially, however, users retain the full flexibility of the *BI* framework to combine these domain-specific components with other model features (e.g., complex non-linear effects via splines, hierarchical structures across groups of networks or phylogenies) or to customize or extend them using *BI*'s underlying mechanisms if needed—a capability often missing in more narrowly focused domain-specific packages. This design aims to foster methodological innovation *within* specialized domains by lowering the barrier to implementing more complex or novel models.

Code block 2: *Example of random effect specification differences between NumPyro, TFP, and BI*

```
# Numpyro version of rendom centered effect
a = numpyro.sample("a", dist.Normal(5, 2))
    b = numpyro.sample("b", dist.Normal(-1, 0.5))
    sigma_cafe = numpyro.sample("sigma_cafe", dist.Exponential(1).expand([2]))
    sigma = numpyro.sample("sigma", dist.Exponential(1))
    Rho = numpyro.sample("Rho", dist.LKJ(2, 2))
    cov = jnp.outer(sigma_cafe, sigma_cafe) * Rho
    a_cafe_b_cafe = numpyro.sample(
        "a_cafe,b_cafe",
        dist.MultivariateNormal(jnp.stack([a, b]), cov).expand([20])
    )
a_cafe, b_cafe = a_cafe_b_cafe[:, 0], a_cafe_b_cafe[:, 1]

# TFP version of rendom centered effect
alpha = yield Root(tfd.Sample(tfd.Normal(loc=5.0, scale=2.0), sample_shape=1))
beta = yield Root(tfd.Sample(tfd.Normal(loc=-1.0, scale=0.5), sample_shape=1))
sigma = yield Root(tfd.Sample(tfd.Exponential(rate=1.0), sample_shape=1))
sigma_alpha_beta = yield Root(tfd.Sample(tfd.Exponential(rate=1.0),
sample_shape=2))
Rho = yield Root(tfd.LKJ(dimension=2, concentration=2.0))
Mu = tf.concat([alpha, beta], axis=-1)
scale = tf.linalg.LinearOperatorDiag(sigma_alpha_beta).matmul(tf.squeeze(Rho))

# BI version of rendom centered effect
Sigma = dist.exponential(1, (ni,), name = 'Sigma_individual')
L = dist.lkjcholesky(1, (ni,), name = 'L_individual', shape = (ni,))
Z = dist.normal(0, 1, name = 'z_individual', shape = (ni,K))
alpha = random_centered2(Sigma, L, Z)
```

BI is designed to encapsulate the entire Bayesian modeling workflow within a cohesive object-oriented structure, promoting a streamlined and reproducible analysis pipeline. Typically, a user interacts with a primary `BI` object, through which they can sequentially:

- **Handle Data:** Load, preprocess, and associate dataset(s) with the model object.
- **Define Model:** Specify the model structure, including the likelihood(s), priors for all parameters, and incorporate any pre-built components using an intuitive formula syntax.
- **Run Inference:** Execute the model fitting process using the No-U-Turn Sampler (NUTS), which triggers the backend PPL (e.g., *NumPyro*, *TFP*) to perform Markov Chain Monte Carlo (MCMC) sampling. Progress indicators and diagnostics are typically provided.
- **Analyze Posterior:** Access, summarize, and diagnose the posterior distributions of parameters. This includes methods for calculating posterior means, medians, credible intervals, convergence diagnostics (e.g., $\hat{R}$, Effective Sample Size - ESS), and retrieving raw posterior samples for custom analysis.

- **Visualize Results:** Generate standard diagnostic plots (e.g., trace plots, rank plots, posterior distributions) and visualizations of model parameters, effects, and predictions using integrated plotting functions that leverage the *arviz* library.

This unified structure minimizes the need for users to juggle multiple disparate software tools or manually transfer data and results between different stages of the analysis, thereby enhancing efficiency and reproducibility.

Finally, *BI* includes over 21 well-documented implementations of various standard and advanced Bayesian models. Examples include Generalized Linear Models (GLMs), Generalized Linear Mixed Models (GLMMs), survival analysis models (e.g., Cox proportional hazards), Principal Component Analysis (PCA), phylogenetic comparative methods, and various network models. Each implementation is accompanied by detailed documentation that encompasses: 1) general principles, 2) underlying assumptions, 3) code snippets in Python and R, and 4) mathematical details, enabling users to gain a deeper understanding of the modeling process and its nuances. Additionally, the framework's flexibility allows models to be combined; for example, building a zero-inflated model with varying intercepts and slopes, or constructing a joint model where principal components (derived from PCA) serve as predictors in a subsequent regression, allowing uncertainty to be propagated through all stages of the analysis.

Example : SRM model

To illustrate how these design features of *BI* coalesce to provide a streamlined, flexible, and powerful solution, effectively addressing the limitations identified in the existing Bayesian software landscape we will provide a basic example of how an SRM model is declared in BI, compare it with the equivalent model in Numpyro (Appendix 1) and STAN (Appendix 2). We will also show how this model can be build from scratch with BI (**Code block 3**) or its custom functions (**Code block 4**) to highligh the aceessibility-flexibility of our package by demonstrating how advance user can build custom model (with less code than STAN) as well as how new user can apply pre-build *BI* models. Finally we show how it is also called in R (**Code block 5**) to cross language use with *BI.* Readers interested in further details on data structure, data import, data manipulation, and model fitting for SRM models can refer directly to the *BI* documentation Modeling Network.

Code block 3: *SRM model from scratch with BI*

```
def model(N_id, idx,  result_outcomes,
          focal_individual_predictors,
          target_individual_predictors):
```

```python
# Intercept
intercept = bi.dist.normal(
    logit(0.1/jnp.sqrt(N_id)),
    2.5, shape=(1,), name = 'intercept'
)

# Sender receiver  ----------------------
N_var = focal_individual_predictors.shape[0]
N_id = focal_individual_predictors.shape[1]
focal_effects = dist.normal(0, 1,  name = 'focal_effects')
target_effects =  dist.normal( 0, 1,  name = 'target_effects')
terms = jnp.stack([
    focal_effects @ focal_individual_predictors,
    target_effects @  target_individual_predictors
    ], axis = -1)
sr_raw =  dist.normal(0, 1, shape=(2, N_id), name = 'sr_raw')
sr_sigma =  dist.exponential( 1, shape= (2,), name = 'sr_sigma')
sr_L = dist.lkjcholesky(2, 2, name = "sr_L")
rf = deterministic('sr_rf',(((sr_L @ sr_raw).T * sr_sigma)))
ids = jnp.arange(0,sr_effects.shape[0])
edgl_idx = bi.net.vec_node_to_edgle(jnp.stack([ids, ids], axis = -1))
sender = sr_effects[edgl_idx[:,0],0] + sr_effects[edgl_idx[:,1],1]
receiver = sr_effects[edgl_idx[:,1],0] + sr_effects[edgl_idx[:,0],1]
sr =  jnp.stack([sender, receiver], axis = 1)

# dyadic effects ------------------------------------------
bi.net.mat_to_edgl(dyadic_effect_mat)
dr_raw =  dist.normal(0, 1, shape=(2,N_dyads), name = 'dr_raw')
dr_sigma = dist.exponential(1,  name = 'dr_sigma' )
dr_L = dist.lkjcholesky(2, 2, name = 'dr_L')
dr_rf = deterministic('dr_rf', (
    ((dr_L @ dr_raw).T * jnp.repeat(dr_sigma, 2))
    ))

dyad_effects = dist.normal(0, 1,
    name= 'dyad_effects', shape = (dyadic_predictors.ndim - 1,
))
dr = dyad_effects * dyadic_predictors

# Likelihood
```

```
    bi.dist.poisson(jnp.exp(intercept + sr + dr), obs=result_outcomes)
```

Code block 4: *SRM model with prebuild functions*

```
def model(N_id, idx,  result_outcomes,
          focal_individual_predictors,
          target_individual_predictors):

    # Intercept
    intercept = bi.dist.normal(
        logit(0.1/jnp.sqrt(N_id)),
        2.5, shape=(1,), name = 'intercept'
    )

    # SR
    sr =  bi.net.sender_receiver(
        focal_individual_predictors,
        target_individual_predictors
    )

    # Dyadic
    dr = bi.net.dyadic_effect(shape = idx.shape[0])

    # Likelihood
    bi.dist.poisson(jnp.exp(intercept + sr + dr), obs=result_outcomes)
```

Code block 5: *SRM model with prebuild functions*

```
model <- function(N_id, idxShape, result_outcomes,
                focal_individual_predictors, target_individual_predictors){

  x=0.1/jnp$sqrt(N_id)
  tmp=jnp$log(x / (1 - x))

  # Intercept
  intercept = bi.dist.normal(tmp, 2.5, shape=c(1), name = 'block')

  # SR
  sr =  m$net$sender_receiver(
            focal_individual_predictors,
            target_individual_predictors
        )

  # Dyadic
  dr = m$net$dyadic_effect(shape = c(idxShape))

  # Likelihood
  m$poisson(jnp$exp(intercept + sr + dr), obs=result_outcomes)
}
```

Finally, regarding code performance we can time the computation time for network of size 200 in STAN and BI and observed that BI comput time is around XXX on cpu and XXX on gpu and STAN compute time around XXX.

Discussion

BI framework is built on top of the popular Python programming language, with a focus on providing a user-friendly interface for model development and interpretation. Our framework is designed to be modular and extensible, allowing users to easily incorporate their own custom models and data types into the framework. One of the key features of this software is its comprehensive library of 21 predefined Bayesian models, covering a wide range of common applications and use cases. These models are accompanied by detailed explanations, making it easier for users to understand the underlying assumptions and apply the models to their specific research questions. In addition to these built-in models, the software includes several custom functions tailored for advanced statistical and network modeling. This curated library serves not only as a collection of ready-to-use tools but also as a valuable pedagogical resource, demonstrating best practices for constructing, fitting, and interpreting models within the *BI* framework, and providing robust templates for users aiming to develop novel model variants. Whether users are interested in hierarchical models, time-series analysis, or cutting-edge network modeling approaches, our library caters to a variety of analytical needs. This accessibility fosters an environment where users can confidently explore and implement Bayesian methods, ultimately enhancing their research capabilities.

By providing a streamlined and efficient environment for the end-to-end Bayesian workflow—from model specification and fitting to diagnostics and prediction, *BI* lowers the barrier to entry for sophisticated Bayesian modeling. We aim to empower a broader community of researchers across disciplines to confidently apply advanced Bayesian methods to their complex research problems.

References

Abadi, Martín, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, et al. 2015. "TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems." https://www.tensorflow.org/.

Bouckaert, Timothy G. AND Barido-Sottani, Remco AND Vaughan. 2019. "BEAST 2.5: An Advanced Software Platform for Bayesian Evolutionary Analysis." *PLOS Computational Biology* 15 (4): 1–28. https://doi.org/10.1371/journal.pcbi.1006650.

Bradbury, James, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, et al. 2018. "JAX: Composable Transformations of Python+NumPy Programs." http://github.com/jax-ml/jax.

Bürkner, Paul-Christian. 2017. "brms: An R Package for Bayesian Multilevel Models Using Stan." *Journal of Statistical Software* 80 (1): 1–28. https://doi.org/10.18637/jss.v080.i01.

Hart, Jordan, Michael Nash Weiss, Daniel Franks, and Lauren Brent. 2023. "BISoN: A Bayesian Framework for Inference of Social Networks." *Methods in Ecology and Evolution* 14 (9): 2411–20. https://doi.org/https://doi.org/10.1111/2041-210X.14171.

Höhna, Sebastian, Michael J. Landis, Tracy A. Heath, Bastien Boussau, Nicolas Lartillot, Brian R. Moore, John P. Huelsenbeck, and Fredrik Ronquist. 2016. "RevBayes: Bayesian Phylogenetic Inference Using Graphical Models and an Interactive Model-Specification Language." *Systematic Biology* 65 (4): 726–36. https://doi.org/10.1093/sysbio/syw021.

McElreath, Richard. 2018. *Statistical Rethinking: A Bayesian Course with Examples in r and Stan*. Chapman; Hall/CRC.

Phan, Du, Neeraj Pradhan, and Martin Jankowiak. 2019. "Composable Effects for Flexible and Accelerated Probabilistic Programming in NumPyro." *arXiv Preprint arXiv:1912.11554*.

Ross, Cody T, Richard McElreath, and Daniel Redhead. 2024. "Modelling Animal Network Data in r Using STRAND." *Journal of Animal Ecology* 93 (3): 254–66.

Salvatier, John, Thomas V Wiecki, and Christopher Fonnesbeck. 2016. "Probabilistic Programming in Python Using PyMC3." *PeerJ Computer Science* 2: e55.

Sosa, Sebastian, Mary B. McElreath, Daniel Redhead, and Cody T. Ross. n.d. "Robust Bayesian Analysis of Animal Networks Subject to Biases in Sampling Intensity and Censoring." *Methods in Ecology and Evolution* n/a (n/a). https://doi.org/https://doi.org/10.1111/2041-210X.70017.

Sosa, Sebastian, Cédric Sueur, and Ivan Puga-Gonzalez. 2020. "Network Measures in Animal Social Network Analysis: Their Strengths, Limits, Interpretations and Uses."

Stan Development Team. "Stan Modeling Language Users Guide and Reference Manual, Version 2.32." https://mc-stan.org.

Wickham, Hadley. 2015. *R Packages*. 1st ed. O'Reilly Media, Inc.