PRACTICAL TOOLS

Methods in Ecology and Evolution

Bayesian inference using the BI Python package

Sebastian Sosa[1] | **Cody T. Ross**[2]

[1]Department of Human Behavior, Ecology and Culture. Max Planck Institute for Evolutionary Anthropology. Leipzig, Germany

Correspondence
Sebastian Sosa
Email: sebastian_sosa@eva.mpg.de

Funding information
Max Planck Institute for Evolutionary Anthropology. Leipzig, Germany

This is a generic template designed for use by multiple journals, which includes several options for customization. Please consult the author guidelines for the journal to which you are submitting in order to confirm that your manuscript will comply with the journal's requirements. Please replace this text with your abstract. Please consult the author guidelines for the journal to which you are submitting in order to confirm that your manuscript will comply with the journal's requirements. Please replace this text with your abstract.

KEYWORDS
Bayesian methods, scalability, software, Python, GPU parallelization, social networks

1 | INTRODUCTION

Bayesian modeling has emerged as an essential part of the toolbox of modern statistics and machine learning, providing a framework for robust inference under uncertainty. Bayesian methods are valuable across the academic disciplines, but are especially useful in evolution, ecology, and animal behavior, where missing data, measurement bias, and non-standard (i.e., scientifically motivated) data-generating models are ubiquitous. The potential of Bayesian methods to address applied challenges in the field is large, but the current ecosystem of Bayesian software is difficult for many end-users (i.e., academic researchers) to navigate, due in large part to limited formal training in Bayesian methods in most field-oriented academic programs, the specialized or bespoke nature of many Bayesian models, the large diversity in coding languages (e.g., `Stan`, `TensorFlow`, `NumPyro`, `JAX`) and software environments (e.g., `R`, `Python`, `C++`), and issues with the scalability of Bayesian models which generally require computationally expensive Markov Chain Monte Carlo (MCMC) sampling. As the software environment evolves to fix one issue (e.g., model scalability is improve by GPU computation), it often creates another (more languages for end-users to learn).

One of the key obstacles in Bayesian modeling is the limited formal training many researchers receive in Bayesian methods and generative model development. Although Bayesian inference has gained traction for its rigor, the com-

plexity of bespoke model formulation, coupled with the need for a deeper understanding of probabilistic programming languages, can be overwhelming for practicing scientists looking to get started. The gap between *the model demanded by theory* and *the easiest generalized linear model to code* in standard software environments (e.g., using `brm` in `R`) often results in users defaulting to generative models that may not fully leverage the utility of Bayesian analysis. Accessible packages, like Paul Bürkner's `brms`, facilitate the coding of multilevel, multivariate Bayesian models in R, to the great benefit of the scientific community. However, many generative scientific models are not easily defined in `brms`, requiring users to either write their own bespoke models in Stan code or resort to using generalized linear models. In some cases, entire software packages have been custom built to deal with niche, but still routinely encountered, data analysis problems. For instance, specialized tools like `STRAND` and `bisonR` have been designed to let users code complex social network analysis models in `Stan` using simple base-`R` syntax, much like `brms`, but with more limited scope.

Another significant barrier to the adoption of Bayesian tools is the diversity of coding languages and software in current use. The `Stan` language remains a gold standard in Bayesian inference due to its highly optimized sampling algorithms and broad support for arbitrarily complex, bespoke models. However, it requires users to learn and write models in its own specialized syntax, which more closely resembles compiled programming languages like `C++` than simpler, interpreted scripting languages like `R` or `Python`. For researchers accustomed to working with basic `R` or `Python` packages, the need to switch to an entirely different programming language can be a considerable hurdle. More generally, whether one uses `Stan` itself, or a user-interface wrapper like `brms`, `STRAND`, or `bisonR`, which all use `Stan` as a back-end—there are challenges related to *scalability* and *parallelization*, which can limit efficiency when handling large datasets or highly parameterized models (e.g., social network models in `STRAND` have a parameter complexity that scales with the square of the number of nodes, and so MCMC run-times can become unacceptable with as few as a few hundred individuals in the sample). In recent years, new software solutions based on `JAX` through `TensorFlow` or `NumPyro` deal with scalability, parallelization, and computational efficiency through GPU integration, but at the cost introducing new model syntax that is even more opaque to typical end-users than `Stan` code.

We contend that the added cognitive load of learning Bayesian methods while simultaneously struggling with various programming language back-ends may be discouraging end-users from fully exploring the capabilities of Bayesian methods, simply due to the friction introduced by language barriers. We therefore see a clear need for a more user-friendly, scalable, and versatile Bayesian modeling environment that can address the gaps in the current software landscape. Our Python package, `BI`, aims to meet this need by providing a platform that serves as a middle-ground between user-friendly, highly constrained wrappers like `brms` or `STRAND`, and unconstrained, but high technical languages like `Stan` and `JAX`. The `BI` package combines the power of unconstrained Bayesian inference languages with the greater accessibility of custom-use wrappers, making it attractive both novice and experienced users. Our package leverages the computational efficiency of `JAX`, through the use of either `TensorFlow` or `NumPyro` as a back-end, while simplifying the process of model specification, fulfilling a role similar to `brms` in `R`, but while retaining most of the flexibility and power of `JAX` (including parallelization of mathematical operations and GPU computation).

2 | SOFTWARE PRESENTATION

Our Bayesian Inference (BI) software comes in two versions: one relying on `NumPyro` and the other on `TensorFlow`. We have simplified the process of model specification by providing a more user-friendly interface for defining Bayesian models, making it accessible to both novice users and experienced practitioners. Since both `NumPyro` and `TensorFlow` use `JAX` under-the-hood, we inherit the advantages of `JAX`, including speed, parallelization, scalability, and GPU sup-

port. User-code in `BI` remains consistent for running parallelized mathematical operations with either back-end, and using either GPU or CPU computation. From start to finish, users can employ a single class object to: 1) organize data, 2) define models, 3) specify estimation algorithms, 4) compute posterior distributions, 5) compute model predictions, and 6) plot results. Additionally, as one of the major issue in the field is the lack of training in Bayesian modeling, we present 20 different tutorials (see Table 1) to cover a wide range of models and research protocols. All presented models are accompanied by detailed explanations, making it easier for users to understand the underlying assumptions of each and then apply similar models to their own specific research questions.

2.1 | User-Friendly Interface for Defining Bayesian Models

A key feature of the Bayesian Inference (`BI`) package is its user-friendly interface designed for defining Bayesian models. We understand that Bayesian modeling can be daunting, especially for those who may lack formal training in programming. To mitigate this, our interface offers a simplified version of `NumPyro` and `TensorFlow` functions. Typically, to define a parameter in a model, both libraries require three different functions. We have merged these functions into a single functional call, with a less opaque syntax. The three lines of code below illustrates the differences between `NumPyro`, `TensorFlow`, and `BI` for declaring a parameter of length 10 that follows a unit-normal distribution:

```
# NumPyro version of a random normal parameter vector---------------------------------
numpyro.sample("mu", dist.Normal(0, 1)).expand([10])

# TensorFlow version of a random normal parameter vector------------------------------
yield Root(tfd.Sample(tfd.Normal(loc=0, scale=1), sample_shape=10))

# BI version of a random normal parameter vector--------------------------------------
bi.dist.normal("mu", 0, 1, shape = (10,))
```

Additionally, while `NumPyro` and `TensorFlow` require entirely different functions for sampling a parameter versus declaring a parameter, we have combined these into a single function, which lets users switch between data simulation and parameter inference just by changing a single function argument: `sample = True`. The code below demonstrates the differences between NumPyro, TFP, and BI for sampling a parameter of length 10 that follows a unit-normal distribution:

```
# NumPyro version of a random normal sampling statement-------------------------------
numpyro.sample("samples", dist.Normal(0, 1).expand([10]), rng_key=jax.random.PRNGKey(0))

# TensorFlow version of a random normal sampling statement----------------------------
tfp.distributions.Normal(loc=0, scale=1).sample(10)

# BI version of a random normal sampling statement------------------------------------
bi.dist.normal("mu", 0, 1, shape = (10,), sample = True)
```

By combining the functions used to declare parameters within a model, and those used to sample parameters in a model, we allow users to easily test and debug their models by first building them as data-simulators in a scripting-style-syntax; then, the exact same generative model can be used as an analytic model. This approach enables users to quickly see the shapes of the underlying objects (e.g., data and parameter arrays) and better manage and combine such objects.

Finally, we include several custom functions tailored for advanced statistical and network modeling. These functions support specialized tasks such as centering random effects, stochastic block modeling, and the computation

of various network measures from social network analysis models. The code below demonstrates the differences between `NumPyro`, `TensorFlow`, and `BI` for declaring a centered random effect in a model:

```
# NumPyro version of centered random 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]
```

```
# TensorFlow version of centered random 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 centered random effect-----------------------------------------------------
    Sigma_individual = bi.dist.exponential("Sigma_individual", [ni], 1 )
    L_individual = bi.dist.lkjcholesky("L_individual", [], ni, 1)
    z_individual = bi.dist.normal("z_individual", [ni,K], 0, 1)
    alpha = random_centered2(Sigma_individual, L_individual, z_individual)
```

By generating these custom functions, we eliminate some data manipulation steps for users, allowing them to more easily adhere to the model's mathematical formulæ. All custom functions are discussed in their respective model descriptions within the `BI` documentation.

Finally, the architecture of our software is built around a versatile class object that encapsulates the entire modeling process. The code below demonstrates how user can: 1) organize data, 2) define models, 3) specify estimation algorithms, 4) compute posterior distributions, 5) compute model predictions, and 6) plot results:

```
# Setup device and call BI object-------------------------------------------------
m = bi(platform="cpu")

# Import data ---------------------------------------------------
m.data("../data/Howell1.csv", sep=";")
m.df = m.df[m.df.age > 18]
m.scale(["weight"]) # Scale continuous data
m.data_to_model(["weight", "height"]) # Convert data to JAX arrays

# Define model ---------------------------------------------------
def model(height, weight):
```

```
    alpha = dist.normal(178, 20, name = "alpha")
    beta = dist.normal(0, 1, name = "beta")
    sigma = dist.uniform(0, 50, name = 'sigma')
    lk("Y", Normal(alpha + beta * weight , sigma), obs = height)

# Run mcmc ------------------------------------------------
m.run(model)

# Summary -------------------------------------------------
m.sampler.print_summary(0.89)

# Plot posterior distributions ----------------------------------------------

#TODO
```

This cohesive structure allows users to efficiently manage different stages of their analysis, from data organization to model evaluation without the need to switch from one library to another. Internally, `BI` manages data handling through the `pandas` library and posterior evaluation through the `Arviz` library.

2.1.1 | Comprehensive Documentation of Models

To support users in their Bayesian modeling journey, we provide comprehensive documentation for 20 different model-types commonly used in evolution, ecology, and animal behavior (Table 1). These tutorials are specifically designed to address a wide array of research questions that typical end-users might use as starting places for their own models. This extensive set of tutorials not only provides users with essential code snippets for tackling diverse research questions, but also serves as an educational resource for training or classes. By presenting various modeling options alongside detailed guidance, we empower users to make informed modeling decisions and effectively apply Bayesian methods in their research. The documentation for each model: 1) discusses general principles, 2) outlines underlying assumptions, 3) provides code snippets, and 4) provides the mathematical formalism, enabling users to gain a deeper understanding of the modeling process and its nuances. Whether users are interested in hierarchical models, time-series analysis, or cutting-edge network modeling approaches, our library provides implementations.

For example, the state-of-the-art network models implemented in `BI` include methods for censoring and exposure control, stochastic block modeling, multiplex and multilayer networks, network-based diffusion analysis, and computation of network metrics. Each approach is discussed in detail in the documentation, as are any custom functions used in those models. This comprehensive documentation ensures that users can easily navigate the software and identify the model-types that best suit their research needs. Furthermore, all of the different functions can be combined into a single model—e.g., users can create a multiplex network model with controls for censoring and exposure biases, and then compute various social network measures on the latent network (see code snippets below). This versatility showcases the flexibility and power of the software, allowing users to tailor their analyses to meet specific research needs. This functionality exceeds what is possible with other specialized Bayesian network analysis tools like `STRAND` and `bisonR`, which rely on pre-built models.

2.2 | Conclusion

The `BI` package 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 extendable, al-

lowing users to easily design or incorporate their own custom models and data types into the framework. We have also developed a set of tutorials and examples to demonstrate the capabilities of `BI` and show that it yields equivalent estimates to models written in Stan when applied to the same data. These tutorial will also help users get started with the framework.

Acknowledgements

Acknowledgements should include contributions from anyone who does not meet the criteria for authorship (for example, to recognize contributions from people who provided technical help, collation of data, writing assistance, acquisition of funding, or a department chairperson who provided general support), as well as any funding or other support information.

Conflict of interest

You may be asked to provide a conflict of interest statement during the submission process. Please check the journal's author guidelines for details on what to include in this section. Please ensure you liaise with all co-authors to confirm agreement with the final statement.

Supporting Information

Supporting information is information that is not essential to the article, but provides greater depth and background. It is hosted online and appears without editing or typesetting. It may include tables, figures, videos, datasets, etc. More information can be found in the journal's author guidelines or at `http://www.wileyauthors.com/suppinfoFAQs`. Note: if data, scripts, or other artefacts used to generate the analyses presented in the paper are available via a publicly available data repository, authors should include a reference to the location of the material within their paper.

TABLE 1 This is a table. Tables should be self-contained and complement, but not duplicate, information contained in the text. They should be not be provided as images. Legends should be concise but comprehensive – the table, legend and footnotes must be understandable without reference to the text. All abbreviations must be defined in footnotes.

Documented Models	Documentation link	Source	Citation
Linear Regression	www.sosa.com/chapter_1.html	Statistical Rethinking, chapter X	[?]
Multiple Regression	www.sosa.com/chapter_1.html	Statistical Rethinking, chapter X	[?]
Interactions	www.sosa.com/chapter_1.html	Statistical Rethinking, chapter X	[?]
Categorical models	www.sosa.com/chapter_1.html	Statistical Rethinking, chapter X	[?]
Binomial models	www.sosa.com/chapter_1.html	Statistical Rethinking, chapter X	[?]
Beta-binomial models	www.sosa.com/chapter_1.html	Statistical Rethinking, chapter X	[?]
Poisson models	www.sosa.com/chapter_1.html	Statistical Rethinking, chapter X	[?]
Gamma-Poisson models	www.sosa.com/chapter_1.html	Statistical Rethinking, chapter X	[?]
Dirichlet models	www.sosa.com/chapter_1.html	Statistical Rethinking, chapter X	[?]
Multinomial models	www.sosa.com/chapter_1.html	Statistical Rethinking, chapter X	[?]
Dirichlet models	www.sosa.com/chapter_1.html	Statistical Rethinking, chapter X	[?]
Zero-inflated models	www.sosa.com/chapter_1.html	Statistical Rethinking, chapter X	[?]
Varying-intercept models	www.sosa.com/chapter_1.html	Statistical Rethinking, chapter X	[?]
Varying-slopes models	www.sosa.com/chapter_1.html	Statistical Rethinking, chapter X	[?]
Gaussian process models	www.sosa.com/chapter_1.html	Statistical Rethinking, chapter X	[?]
Measurement-error models	www.sosa.com/chapter_1.html	Statistical Rethinking, chapter X	[?]
Missing-data models	www.sosa.com/chapter_1.html	Statistical Rethinking, chapter X	[?]
Basic social network models	www.sosa.com/chapter_1.html	STRAND	[?]
Sender-receiver network models	www.sosa.com/chapter_1.html	STRAND	[?]
Stochastic blockmodels	www.sosa.com/chapter_1.html	STRAND	[?]
Network feature computations	www.sosa.com/chapter_1.html	ANTs	[?]
Network-based diffusion analysis	www.sosa.com/chapter_1.html	ANTs	[?]
Multiplex network models	www.sosa.com/chapter_1.html	STRAND	[?]
Longitudinal network models	www.sosa.com/chapter_1.html	STRAND	[?]
Multilayer network models	www.sosa.com/chapter_1.html	ANTs	[?]