Reproducible sampling from intractable distributions with Pigeons.jl

Miguel Biron-Lattes [1] **Nikola Surjanovic** [2] **Paul Tiede** [3] **Saifuddin Syed** [4] **Trevor Campbell** [2]
Alexandre Bouchard-Côté [2]

Abstract

Sampling from complicated probability distributions—such as multimodal posteriors arising in Bayesian inference and high-dimensional distributions in statistical mechanics—is a challenging problem. *Pigeons.jl* is a Julia package that implements *Parallel Tempering*, a technique that leverages parallelism to improve the effectiveness of *Markov chain Monte Carlo* algorithms. Pigeons.jl provides a simple user interface to perform such computations single-threaded, multi-threaded, and/or distributed over thousands of MPI-communicating machines. In this paper, we discuss a feature of Pigeons.jl that we call *strong parallelism invariance* (SPI): a guarantee that the output for a given seed is identical irrespective of the number of threads and processes. This feature is crucial for scientific reproducibility and software validation. We describe the key features of Pigeons.jl that enable a distributed and randomized algorithm satisfying SPI. Finally, we briefly discuss some recent developments in Pigeons.jl.

1. Introduction

In many scientific application domains, the ability to obtain samples from a probability distribution is of central interest. For instance, sampling methods have been used to discover magnetic polarization in the black hole of galaxy M87 (Akiyama et al., 2021) and to image the Sagittarius A* black hole (Akiyama et al., 2022). They have also been used to model the evolution of single-cell cancer genomes (Salehi et al., 2021), infer plasma dynamics inside nuclear fusion reactors (Gota et al., 2021), and to identify gerrymandering in Georgia's 2021 congressional districting plan (Zhao et al., 2022). Similarly, evaluating high-dimensional integrals or sums over complicated combinatorial spaces are related tasks that can also be solved with sampling via Markov chain Monte Carlo (MCMC) methods. However, such calculations can often be bottlenecks in the scientific process, with simulations that can last days or even weeks to finish.

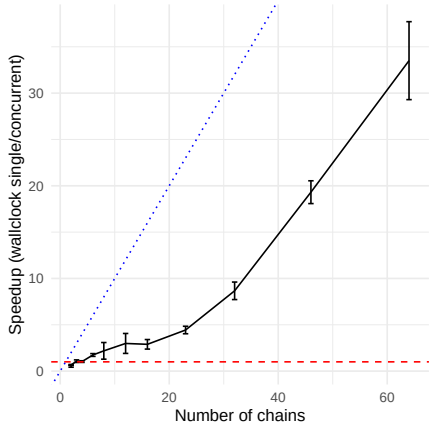

Figure 1: Speedup relative to single-thread execution for Pigeons.jl in the distributed (MPI) setting. Error bars denote approximate 95% confidence intervals. The blue dotted line represents a hypothetical speedup equal to the number of processes. The red dashed line indicates a speedup of one for reference.

Pigeons.jl[1] enables users to sample efficiently from challenging probability distributions. To achieve this, Pigeons.jl provides a multi-threaded and distributed implementation of non-reversible *parallel tempering* (PT; Syed et al., 2021a;b; Surjanovic et al., 2022; 2024), a state-of-the-art MCMC algorithm. Crucially, Pigeons.jl follows the tuning guidelines described in Syed et al. (2021a) to provide a hyperparameter-free implementation. Pigeons.jl's simple user interface facilitates running PT single-threaded, multi-threaded, and/or distributed over

This is a summarized and revised version of Surjanovic et al. (2025). [1]Department of Statistics and Actuarial Sciences, Simon Fraser University, Canada [2]Department of Statistics, University of British Columbia, Canada [3]Center for Astrophysics and Black Hole Initiative, Harvard University, United States [4]Department of Statistics, University of Oxford, United Kingdom. Correspondence to: Miguel Biron-Lattes <mbironla@sfu.ca>.

Proceedings of the ICML 2025 Workshop on Championing Open-source Development in Machine Learning (CODEML '25).

[1]Documentation available at `https://pigeons.run/dev/`. Public repository: `https://github.com/Julia-Tempering/Pigeons.jl`.

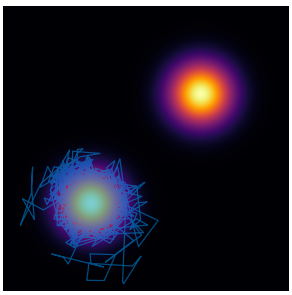

Figure 2: A bimodal distribution. Blue lines display output from 1,000 iterations of a random walk Metropolis–Hastings sampler.

MPI-communicating machines. We have stress-tested Pigeons successfully with up to 1,000 processes running concurrently on a compute cluster (Sockeye) at the University of British Columbia. Importantly, Pigeons comes with guarantees on *strong parallelism invariance* (SPI), wherein the output for a given seed is *identical* irrespective of the number of threads or processes. Such a level of reproducibility is rare in distributed software but of great use for the purposes of debugging in the context of sampling algorithms, which produce stochastic output. Specifically, Pigeons.jl is designed to be suitable and yield reproducible output for:

1. one machine running on one thread;
2. one machine running on several threads;
3. several machines running, each using one thread, and
4. several machines running, each using several threads.

2. Parallel tempering in a nutshell

Consider the problem of sampling from a probability density (or mass) function $\pi(x)$, henceforth called the *target*. Solving this task is often challenging, as the distribution π can exhibit features which traditional MCMC methods—from random walk Metropolis–Hastings to Hamiltonian Monte Carlo—struggle to describe accurately. For example, in bimodal distributions such as the one illustrated in Fig. 2, traditional methods might remain stuck in one of the two modes for an extremely long period of time.

To resolve this issue, PT constructs a sequence of N distributions, $\pi_1, \pi_2, \ldots, \pi_N$, where π_N is usually equal to the target π. The distributions are chosen so that it is easy to obtain samples from π_1—called the *reference* distribution—with the sampling difficulty increasing as one approaches π_N. An example of such a sequence of distributions, referred to as an *annealing path*, is shown in Fig. 3.

PT operates by first obtaining samples from each distribution on the path in parallel; this is called an *exploration* phase. Then, in the *communication* phase, PT proposes swapping the samples between adjacent distributions. This step is crucial: successive swaps along the path amount to transporting samples from the reference to the target. This allows for the discovery of new regions of the space of the target distribution such as the top-right mode in Fig. 2.

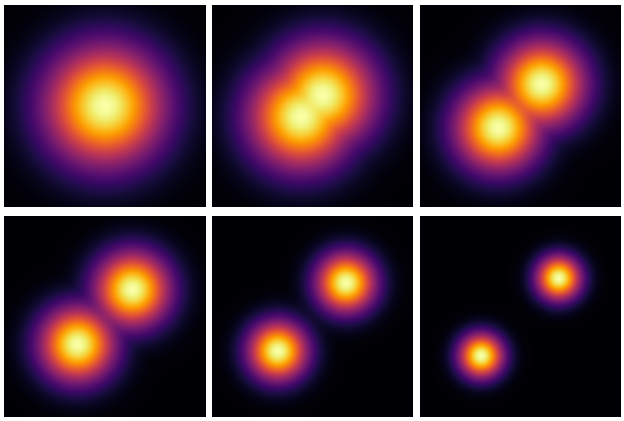

Figure 3: Heatmaps of six distributions lying on an annealing path, starting from a unimodal reference (top-left), and ending at the target distribution from Fig. 2.

3. Overview of Pigeons.jl

Pigeons.jl exploits the fact that the exploration phase in PT is embarrassingly parallel to provide a multithreaded and distributed Julia implementation of the algorithm.

In many problems, e.g. in Bayesian statistics, the target π is typically known only up to a normalizing constant,

$$\pi(x) = \frac{\gamma(x)}{Z}, \qquad Z = \int \gamma(x)\,\mathrm{d}x, \tag{1}$$

where γ can be evaluated pointwise but Z is typically unknown. Pigeons.jl takes as input the function γ.

The output of Pigeons.jl can be used for two main tasks:

1. Approximating integrals of the form $\int f(x)\pi(x)\,\mathrm{d}x$. For example, the choice $f(x) = x$ computes the mean and $f(x) = I[x \in A]$ computes the probability of A under π, where $I[\cdot]$ denotes the indicator function.
2. Approximating the value of the normalization constant Z. For example, in Bayesian statistics, this corresponds to the marginal likelihood, which can be used for model selection.

The Pigeons package particularly shines compared to traditional sampling approaches in the following scenarios:

- When the target density π is challenging due to a complex structure (e.g., high-dimensional, multi-modal, etc.).
- When the user needs not only $\int f(x)\pi(x)\,\mathrm{d}x$ but also the normalizing constant Z. Many existing tools focus on the former and struggle or fail to do the latter.

- When the target distribution π is defined over a non-standard space, e.g. a combinatorial object such as a phylogenetic tree.

There are two fundamental concepts in the implementation that we will use in the following sections. For a distribution π_i in the path, we call its index the i-th *chain*. During execution, we assign *replicas* to work on the distributions associated to the chains. The map between chains and replicas is one-to-one, and thus can be seen as a permutation of the set $\{1, \ldots, N\}$. Replicas are the fundamental unit of parallelization in PT: they may be distributed across threads, processes, and machines.

In Appendix A we provide examples of how to use Pigeons.jl. For more details on the implementation, see Surjanovic et al. (2025).

4. Strong parallelism invariance

In this section we describe potential violations of *strong parallelism invariance* (SPI)—which we define as a guarantee that the output for a given seed is identical irrespective of the number of threads and processes—that can occur in a distributed setting. We also explain how we avoid these issues by using special distributed reduction schemes and splittable random number generators. Insights provided in this section can be applied to general distributed software beyond Julia.

We have identified two factors that can cause violations of our previously-defined SPI that standard Julia libraries do not automatically take care of:

1. **Non-associativity of floating point operations:** When several workers perform distributed reduction of floating point values, the output of this reduction will be slightly different depending on the order taken during reduction. When these reductions are then fed into further random operations, this implies that two randomized algorithms with the same seed but using a different number of workers will eventually arbitrarily diverge.
2. **Global, thread-local, and task-local random number generators:** These are the dominant approaches to parallel random number generators in current languages, and an appropriate understanding of these RNGs is necessary. In particular, in Julia it is important to understand the behaviour of the `@threads` macro.

Our focus in the remainder of this section is to describe how our implementation solves the two above issues.

4.1. Distributed reduction and floating point non-associativity

The execution of PT in Pigeons.jl is divided into *rounds*, which are sequences of *scans*—an exploration phase followed by a communication phase—of exponentially increasing length. At the end of each round, replicas need to exchange information. This can be related to the output of the program—e.g., the value of the normalization constant Z—as well as statistics needed to adapt the sampler so that the next round runs more efficiently. Almost all of these cases involve summing floating-point numbers that are located on different threads, processes, or compute nodes (in the following we use the generic term *machine*).

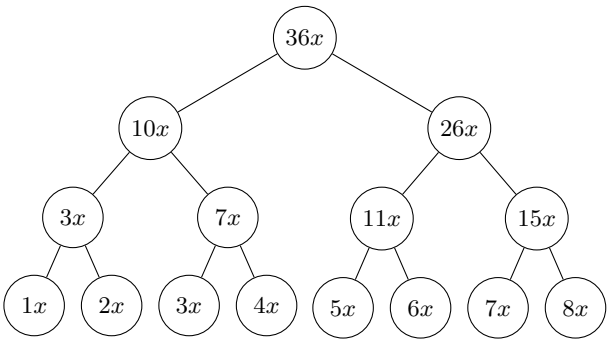

Figure 4: Adding eight floating point numbers $\{1x, 2x, \ldots, 8x\}$ across eight machines. Additions in each row of the tree can be performed in parallel. The final result is stored in the root node of the tree and can be represented by the expression $((1x + 2x) + (3x + 4x)) + ((5x + 6x) + (7x + 8x))$, indicating the order of operations.

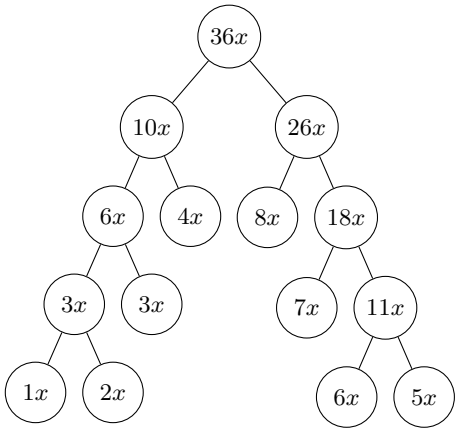

Figure 5: One possible way of adding eight floating point numbers $\{1x, 2x, \ldots, 8x\}$ across two machines. The final result is stored in the root node of the tree and can be represented by the expression $(((1x + 2x) + 3x) + 4x) + (((5x + 6x) + 7x) + 8x)$. Note that the order of operations in this expression is different from the one presented in Fig. 4.

To illustrate why distributed reduction with floating point values can violate strong parallelism invariance if not properly implemented, we consider the following toy example. Suppose we have 8 machines storing the floating point numbers $\{1x, 2x, \ldots, 8x\}$, as illustrated in Fig. 4, where we use $x = 10e^1 \approx 27.1828$ in the following examples. In this case, if our reduction procedure is to sum the float-

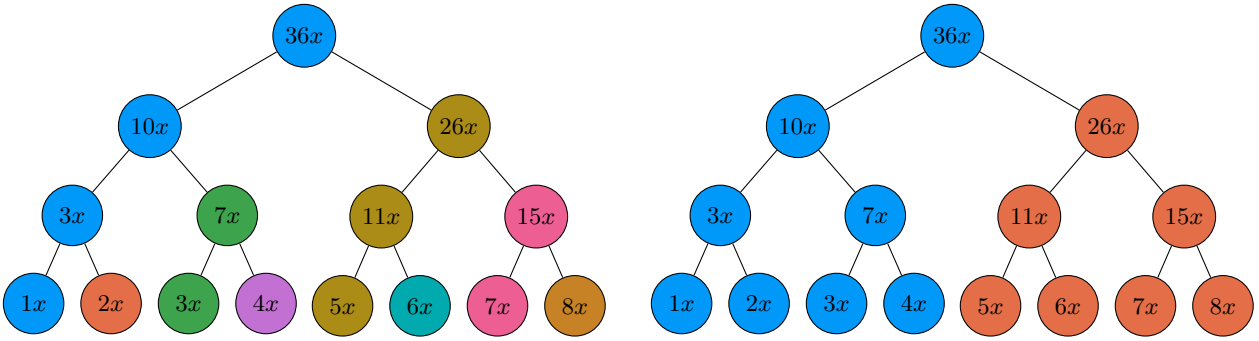

Figure 6: Addition of eight floating point numbers across eight (**left**) and two (**right**) machines with a guarantee on SPI. Each machine is represented by a different colour. In both cases, the final result can be represented by the same expression $((1x + 2x) + (3x + 4x)) + ((5x + 6x) + (7x + 8x))$.

ing point numbers, we know that our final answer should be approximately $36x$. However, depending on the exact order in which floating point addition is carried out, the answers might not all be the same and exactly equal to $36x$. For instance, in Fig. 4 we see that the order of operations for eight machines would be given by

$$((1x + 2x) + (3x + 4x)) + ((5x + 6x) + (7x + 8x)) \approx 978.5814582452562.$$

In contrast, with two machines, one possible order of operations might be

$$(((1x + 2x) + 3x) + 4x) + (((5x + 6x) + 7x) + 8x) \approx 978.5814582452563.$$

Fig. 5 shows a possible reduction tree for two machines.

To avoid the issue of non-associativity of floating point arithmetic, we ensure that the order in which operations are performed is exactly the same, irrespective of the number of processes/machines and threads. This is achieved by making sure that every value to be added—if addition is our reduction operation—is a leaf node in a reduction tree that is invariant to the number of machines available to perform the computation. For instance, if N values are to be reduced, then the reduction tree would have N leaf nodes. If M machines are available, these machines are then assigned in such a way that the order of operations is as if there were N machines available. To do so, it may be necessary for a machine to "communicate with itself", imitating the behaviour that would be present if there were N machines available. Fig. 6 illustrates the reduction procedure for eight and two machines.

4.2. Splittable random streams

Another building block towards achieving SPI is a *splittable random stream* (L'Ecuyer, 1988; Burton & Page, 1992). Julia uses *task-local* random number generators, a notion that is related to (but does not necessarily imply) strong parallelism invariance. A *task* is a unit of work on a machine. A task-local RNG would then mean that a separate RNG is used for each unit of work, hopefully implying strong parallelism invariance if the number of tasks is assumed to be constant. Unfortunately, this is not the case when a separate task is created for each thread of execution in Julia. We note that the `@threads` macro in Julia creates `nthreads()` tasks and thus `nthreads()` pseudo-random number generators. This can break strong parallelism invariance as the output may depend on the number of threads.

To motivate splittable random streams, consider the following toy example that violates our notion of SPI:

```
using Random
import Base.Threads.@threads

println("Num. of threads: $(Threads.nthreads())")

const n_iters = 10000
result = zeros(n_iters)
Random.seed!(1)
@threads for i in 1:n_iters
    result[i] = rand()
end
println("Result: $(last(result))")
```

With eight threads, this outputs:

```
Num. of threads: 8
Result: 0.25679999169092793
```

Julia guarantees that if we rerun this code, as long as we are using eight threads, we will always get the same result, irrespective of the multi-threading scheduling decisions implied by the `@threads`-loop. Internally, Julia works with task-local RNGs and the `@threads` macro spawns `nthreads()` number of task-local RNGs. For this reason, with a different number of threads, the result is different:

```
Num. of threads: 1
Result: 0.8785201210435906
```

In this simple example above, the difference in output is perhaps not too concerning, but for our parallel tempering use case, the distributed version of the algorithm is significantly more complex and difficult to debug compared to the single-threaded one. We therefore take task-local random number generation one step further and achieve SPI, which guarantees that the output is not only reproducible with respect to repetitions for a fixed number of threads, but also for different numbers of threads or processes.

A first step to achieve this is to associate one random number generator to each replica. To do so, we use our SplittableRandoms.jl package, which is a Julia implementation of Java SplittableRandoms. Our package offers an implementation of SplitMix64 (Steele et al., 2014), which allows us to turn one seed into an arbitrary collection of pseudo-independent RNGs. A quick example of how to use the SplittableRandoms.jl library is given below. By splitting a master RNG using the `split()` function, we can achieve SPI even with the use of the `@threads` macro.

```
using Random
using SplittableRandoms: SplittableRandom, split
import Base.Threads.@threads

println("Num. of threads: $(Threads.nthreads())")

const n_iters = 10000
const master_rng = SplittableRandom(1)
result = zeros(n_iters)
rngs = [split(master_rng) for _ in 1:n_iters]
Random.seed!(1)
@threads for i in 1:n_iters
    result[i] = rand(rngs[i])
end
println("Result: $(last(result))")
```

With one and eight threads, the code above outputs

```
Num. of threads: 1
Result: 0.4394633333251359

Num. of threads: 8
Result: 0.4394633333251359
```

5. Recent developments

We discuss briefly some important additions to Pigeons.jl since its first release (v0.1.0).

autoMALA Since version v0.2.0, Pigeons.jl's default explorer for most models in continuous space has been autoMALA (Biron-Lattes et al., 2024), a novel MCMC sampler based on the Metropolis-adjusted Langevin algorithm (MALA). autoMALA continually adapts its step size at each iteration based on the local geometry of the target distribution. Such automatic adjustments are highly desirable in the exploration phase, since otherwise we would need to find optimal step sizes for each distribution in the annealing path—a prohibitively costly process.

Support for BUGS models The latest version of Pigeons.jl at the time of writing (v0.4.9) added support for sampling from programs written in BUGS (Lunn et al., 2000; 2009; 2013), by leveraging the Julia implementation of the language in JuliaBUGS.jl[2].

Tempered Particle Gibbs for Turing-complete PPLs Particle Gibbs (PG, Andrieu et al., 2010) is an MCMC algorithm that has been shown (Wood et al., 2014) to be a natural sampler for arbitrary programs written in Turing-complete probabilistic programming languages (PPLs). For example, the Julia package Turing.jl (Ge et al., 2018) provides a PG sampler that can handle any program written in DynamicPPL (Tarek et al., 2020), its underlying PPL. However, this generality comes at the expense of poor performance in moderately complex models. The reason is that PG is, essentially, a smart way for selecting samples from the prior that are close to the posterior. When these distributions are very different, PG can be inefficient.

With Pigeons.jl we can partially alleviate this issue by using PG as an explorer in PT. Indeed, PG should be efficient for exploring distributions closer to the prior. Such samples can then be transported towards the posterior chain via the PT process. In order to construct the annealing sequence of distributions for an arbitrary PPL program, it suffices to inject an inverse temperature parameter $\beta \in [0, 1]$ in the call to compute the log density of every `observe` statement (see Gordon et al., 2014, for an explanation of this syntax) in the program. Leveraging multiple dispatch in Julia, we have successfully applied this technique to DynamicPPL programs in our package TPGExplorers.jl[3].

6. Conclusion

Pigeons.jl is a Julia package that enables users with no experience in distributed computing to efficiently approximate posterior distributions and solve challenging Lebesgue integration problems over a distributed computing environment. The core algorithm behind Pigeons.jl is distributed, non-reversible parallel tempering (Syed et al., 2021a; Surjanovic et al., 2022). Pigeons.jl can be used in a multi-threaded context, as well as distributed over up to thousands of MPI-communicating machines. Further, Pigeons.jl is designed so that for a given seed, the output is *identical* regardless of the number of threads or processes used.

References

Akiyama, K., Algaba, J. C., Alberdi, A., Alef, W., Anantua, R., Asada, K., Azulay, R., Baczko, A.-K., Ball, D., Baloković, M., et al. First M87 Event Horizon

[2] https://turinglang.org/JuliaBUGS.jl/stable/

[3] https://github.com/Julia-Tempering/TPGExplorers.jl

Telescope results. VII. Polarization of the ring. *The Astrophysical Journal Letters*, 910(1):L12, 2021. doi: 10.3847/2041-8213/abe71d.

Akiyama, K., Alberdi, A., Alef, W., Algaba, J. C., Anantua, R., Asada, K., Azulay, R., Bach, U., Baczko, A.-K., Ball, D., et al. First Sagittarius A* Event Horizon Telescope results. IV. Variability, morphology, and black hole mass. *The Astrophysical Journal Letters*, 930(2): L15, 2022. doi: 10.3847/2041-8213/ac6736.

Andrieu, C., Doucet, A., and Holenstein, R. Particle Markov chain Monte Carlo methods. *Journal of the Royal Statistical Society Series B: Statistical Methodology*, 72(3):269–342, 05 2010. ISSN 1369-7412. doi: 10.1111/j.1467-9868.2009.00736.x. URL https://doi.org/10.1111/j.1467-9868.2009.00736.x.

Biron-Lattes, M., Surjanovic, N., Syed, S., Campbell, T., and Bouchard-Côté, A. autoMALA: Locally adaptive Metropolis-adjusted Langevin algorithm. In *International Conference on Artificial Intelligence and Statistics*, pp. 4600–4608. PMLR, 2024. doi: 10.48550/arXiv.2310.16782.

Bouchard-Côté, A., Chern, K., Cubranic, D., Hosseini, S., Hume, J., Lepur, M., Ouyang, Z., and Sgarbi, G. Blang: Bayesian declarative modeling of general data structures and inference via algorithms based on distribution continua. *Journal of Statistical Software*, 103:1–98, 2022. doi: 10.18637/jss.v103.i11.

Burton, F. W. and Page, R. L. Distributed random number generation. *Journal of Functional Programming*, 2(2): 203–212, 1992. doi: 10.1017/S0956796800000320.

Carpenter, B., Gelman, A., Hoffman, M. D., Lee, D., Goodrich, B., Betancourt, M., Brubaker, M., Guo, J., Li, P., and Riddell, A. Stan: a probabilistic programming language. *Journal of Statistical Software*, 76(1), 2017. doi: 10.18637/jss.v076.i01.

Day, J. and Zhou, H. OnlineStats.jl: A Julia package for statistics on data streams. *Journal of Open Source Software*, 5(46), 2020. doi: 10.21105/joss.01816.

Ge, H., Xu, K., and Ghahramani, Z. Turing: A language for flexible probabilistic inference. In *International Conference on Artificial Intelligence and Statistics*, pp. 1682–1690. PMLR, 2018. doi: 10.17863/CAM.42246.

Gordon, A. D., Henzinger, T. A., Nori, A. V., and Rajamani, S. K. Probabilistic programming. In *Future of Software Engineering Proceedings*, FOSE 2014, pp. 167–181, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450328654. doi: 10.1145/2593882.2593900. URL https://doi.org/10.1145/2593882.2593900.

Gota, H., Binderbauer, M., Tajima, T., Smirnov, A., Putvinski, S., Tuszewski, M., Dettrick, S., Gupta, D., Korepanov, S., Magee, R., et al. Overview of C-2W: high temperature, steady-state beam-driven field-reversed configuration plasmas. *Nuclear Fusion*, 61(10): 106039, 2021. doi: 10.1088/1741-4326/ac2521.

L'Ecuyer, P. Efficient and portable combined random number generators. *Communications of the ACM*, 31(6): 742–751, June 1988. ISSN 0001-0782. doi: 10.1145/62959.62969. URL https://doi.org/10.1145/62959.62969.

Lunn, D., Spiegelhalter, D., Thomas, A., and Best, N. The BUGS project: evolution, critique and future directions. *Statistics in Medicine*, 28(25):3049–3067, 2009. doi: 10.1002/sim.3680.

Lunn, D., Jackson, C., Best, N., Thomas, A., and Spiegelhalter, D. *The BUGS Book: A Practical Introduction to Bayesian Analysis*. CRC Press, 2013. doi: 10.1201/b13613.

Lunn, D. J., Thomas, A., Best, N., and Spiegelhalter, D. WinBUGS — a Bayesian modelling framework: concepts, structure, and extensibility. *Statistics and Computing*, 10:325–337, 2000. doi: 10.1023/A:1008929526011.

Salehi, S., Kabeer, F., Ceglia, N., Andronescu, M., Williams, M. J., Campbell, K. R., Masud, T., Wang, B., Biele, J., Brimhall, J., et al. Clonal fitness inferred from time-series modelling of single-cell cancer genomes. *Nature*, 595(7868):585–590, 2021. doi: 10.1038/s41586-021-03648-3.

Steele, G. L., Lea, D., and Flood, C. H. Fast splittable pseudorandom number generators. *ACM SIGPLAN Notices*, 49(10):453–472, 2014. doi: 10.1145/2660193.2660195.

Surjanovic, N., Syed, S., Bouchard-Côté, A., and Campbell, T. Parallel tempering with a variational reference. In *Advances in Neural Information Processing Systems*, 2022.

Surjanovic, N., Syed, S., Bouchard-Côté, A., and Campbell, T. Uniform ergodicity of parallel tempering with efficient local exploration. *arXiv:2405.11384*, 2024. doi: 10.48550/arXiv.2405.11384.

Surjanovic, N., Biron-Lattes, M., Tiede, P., Syed, S., Campbell, T., and Bouchard-Côté, A. Pigeons.jl: Distributed sampling from intractable distributions. *Proceedings of the JuliaCon Conferences*, 7(69):139, 2025. doi: 10.21105/jcon.00139. URL https://doi.org/10.21105/jcon.00139.

Syed, S., Bouchard-Côté, A., Deligiannidis, G., and Doucet, A. Non-reversible parallel tempering: a scalable highly parallel MCMC scheme. *Journal of Royal Statistical Society, Series B*, 84:321–350, 2021a. doi: 10.1111/rssb.12464.

Syed, S., Romaniello, V., Campbell, T., and Bouchard-Côté, A. Parallel tempering on optimized paths. In *International Conference on Machine Learning*, pp. 10033–10042. PMLR, 2021b. doi: 10.48550/arXiv.2102.07720.

Tarek, M., Xu, K., Trapp, M., Ge, H., and Ghahramani, Z. Dynamicppl: Stan-like speed for dynamic probabilistic models. *arXiv e-prints*, art. arXiv:2002.02702, February 2020. doi: 10.48550/arXiv.2002.02702.

Wood, F., Meent, J. W., and Mansinghka, V. A New Approach to Probabilistic Programming Inference. In *Proceedings of the Seventeenth International Conference on Artificial Intelligence and Statistics*, volume 33 of *Proceedings of Machine Learning Research*, pp. 1024–1032, 22–25 Apr 2014. URL `https://proceedings.mlr.press/v33/wood14.html`.

Xie, W., Lewis, P. O., Fan, Y., Kuo, L., and Chen, M.-H. Improving marginal likelihood estimation for Bayesian phylogenetic model selection. *Systematic Biology*, 60 (2):150–160, 2011. doi: 10.1093/sysbio/syq085.

Zhao, Z., Hettle, C., Gupta, S., Mattingly, J. C., Randall, D., and Herschlag, G. J. Mathematically quantifying non-responsiveness of the 2021 Georgia congressional districting plan. In *EAAMO '22: Equity and Access in Algorithms, Mechanisms, and Optimization*, pp. 1–11, 2022. doi: 10.1145/3551624.3555300.

A. Examples

In this section we present a set of minimal examples that demonstrate how to use Pigeons.jl for sampling. We also direct readers to our growing list of examples at InferHub (`https://julia-tempering.github.io/InferHub/`), which hosts a collection of posterior distributions with an emphasis on difficult (non-log-concave) problems.

We begin by installing the latest official release of Pigeons.jl:

```
using Pkg; Pkg.add("Pigeons")
```

A.1. Targets

To use Pigeons.jl, we must specify a target distribution, given by γ in Eq. (1). Numerous possible types of target distributions are supported, including custom probability densities (specified up to a normalizing constant) written in Julia. We also allow to interface with models written in common probabilistic programming languages, including:

- Turing.jl (Ge et al., 2018) models (`TuringLogPotential`)
- Stan (Carpenter et al., 2017) models (`StanLogPotential`)
- Comrade.jl[4] models for black hole imaging (`ComradeLogPotential`)
- Non-Julian models with foreign-language Markov chain Monte Carlo (MCMC) code (e.g. Blang (Bouchard-Côté et al., 2022) code for phylogenetic inference over combinatorial spaces)

Additional targets are currently being accommodated and will be introduced to Pigeons.jl in the near future.

In what follows, we demonstrate how to use Pigeons with a Julia Turing model applied to a non-identifiable "coinflip" data set. The Bayesian model can be formulated as

$$\begin{aligned} p_1, p_2 &\overset{\text{i.i.d.}}{\sim} U(0,1), \\ y \mid p_1, p_2 &\sim \text{Binomial}(n, p_1 p_2). \end{aligned} \tag{2}$$

The random variable y is the number of heads observed on n coin flips where the probability of heads is $p_1 p_2$. This model is non-identifiable, meaning that it is not possible to distinguish the effects of the two different parameters p_1 and p_2. As a consequence, the target distribution exhibits a complicated structure, as displayed in Fig. 7. The density of interest corresponding to this model is

$$\pi(p_1, p_2) = \gamma(p_1, p_2)/Z,$$

where

$$\gamma(p_1, p_2) = \binom{n}{y} (p_1 p_2)^y (1 - p_1 p_2)^{n-y} I[p_1, p_2 \in [0,1]] \tag{3}$$

$$Z = \int_0^1 \int_0^1 \gamma(p_1, p_2)\, \mathrm{d}p_1\, \mathrm{d}p_2. \tag{4}$$

The distribution π is also known as the *posterior distribution* in Bayesian statistics.

Suppose that we perform $n = 100,000$ coin tosses and observe $y = 50,000$ heads. We would like to obtain samples from our posterior, π, having collected this data. We begin by installing Turing

```
Pkg.add("Turing")
```

and then defining our Turing model and storing it in the variable `model`:

```
using Turing
@model function coinflip(n, y)
    p1 ~ Uniform(0.0, 1.0)
```

[4] `https://github.com/ptiede/Comrade.jl`

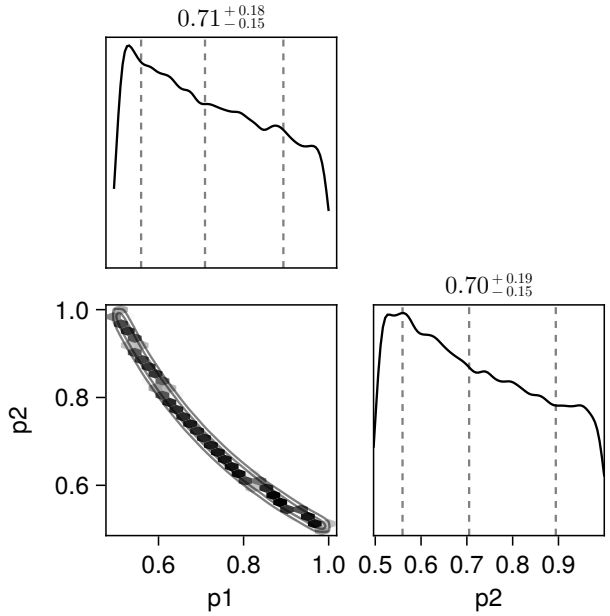

Figure 7: Posterior distribution for the model given by Eq. (2) with $n = 100,000$ coin flips and $y = 50,000$ observed heads, estimated using 2^{17} samples from Pigeons.jl. We present the pairwise plot for p_1 and p_2, as well as the estimated densities of the marginal of the posterior for each of the two parameters. Note that because the model is non-identifiable, as we collect more data the posterior distribution concentrates around the curve $p_1 p_2 = 0.5$, instead of a single point, assuming that the true probability of observing heads is 0.5.

```
    p2 ~ Uniform(0.0, 1.0)
    y ~ Binomial(n, p1 * p2)
    return y
end
model = coinflip(100000, 50000)
```

From here, it is straightforward to sample from the density given by Eq. (3) up to a normalizing constant. We use non-reversible parallel tempering (Syed et al., 2021a;b; Surjanovic et al., 2022; 2024) (PT), Pigeons.jl's state-of-the-art sampling algorithm, to sample from the target distribution. PT comes with several tuning parameters and Syed et al. (2021a) describe how to select these parameters effectively, which Pigeons.jl implements under the hood. We also specify that we would like to store the obtained samples in memory to be able to produce trace-plots, as well as some basic online summary statistics of the target distribution and useful diagnostic output by specifying `record = [traces, online, round_trip, Pigeons.timing_extrema, Pigeons.allocation_extrema]`. It is also possible to leave the `record` argument empty and reasonable defaults will be selected for the user. The code below runs Pigeons.jl on one machine with one thread. We use the default values for most settings, however we explain later how one can obtain improved performance by setting arguments more carefully (see Appendix A.4).

```
using Pigeons
pt = pigeons(
    target = TuringLogPotential(model),
    record = [
        traces, online, round_trip,
        Pigeons.timing_extrema,
        Pigeons.allocation_extrema])
```

Note that to convert the Turing model into an appropriate Pigeons.jl target for sampling, we pass the model as an argument to `TuringLogPotential()`. Once we have stored the PT output in the variable `pt` we can access the results, as described in the following section. The standard output after running the above code chunk is displayed in Fig. 8 and explained in the next section. For purposes of comparison, we also run a traditional (single-chain Markov chain Monte Carlo) method.

#scans	restarts	Λ	time(s)	allc(B)	$\log(Z_1/Z_0)$	min(α)	mean(α)
2	0	1.04	0.383	3.48e+07	-4.24e+03	0	0.885
4	0	4.06	0.00287	1.79e+06	-16.3	4.63e-06	0.549
8	0	3.49	0.00622	3.55e+06	-12.1	0.215	0.612
16	0	2.68	0.0161	7.46e+06	-10.2	0.518	0.703
32	0	4.29	0.0353	1.37e+07	-11.8	0.222	0.524
64	3	3.17	0.0699	2.86e+07	-11.5	0.529	0.648
128	8	3.56	0.139	5.53e+07	-11.5	0.523	0.605
256	12	3.38	0.241	1.1e+08	-11.6	0.526	0.625
512	37	3.48	0.473	2.22e+08	-12	0.527	0.614
1.02e+03	77	3.55	0.895	4.46e+08	-11.8	0.571	0.605

Figure 8: Standard output provided by Pigeons.jl. Rows indicate tuning rounds of the PT algorithm with an exponentially increasing number of PT iterations (`#scans`). Columns indicate various useful diagnostics, such as the amount of memory allocations per round (in bytes), time (in seconds), and estimates of the log of the normalization constants. The output is described in greater detail in Appendix A.2.6 and has been modified to exclude columns that are not described in the paper.

A.1.1. Other targets

As mentioned previously, it is also possible to specify targets with custom probability densities, as well as Stan and Turing models. Additionally, suppose we have some code implementing vanilla MCMC, written in an arbitrary "foreign" language such as C++, Python, R, Java, etc. Surprisingly, it is very simple to bridge such code with Pigeons.jl. See `https://pigeons.run/stable/input-overview/`.

A.2. Outputs

Pigeons.jl provides many useful types of output, such as: plots of samples from the distribution, estimates of normalization constants, summary statistics of the target distribution, and various other diagnostics. We describe several examples of possible output below.

A.2.1. Standard output

An example of the standard output provided by Pigeons.jl is displayed in Fig. 8. Each row of the table in the output indicates a new tuning round in parallel tempering, with the `#scans` column indicating the number of scans/samples in that tuning round. During these tuning rounds, Pigeons.jl searches for optimal values of certain PT tuning parameters. Other outputs include:

- `restarts`: a higher number is better. Informally, PT propagates samples from an easy-to-sample distribution (the reference) to the more difficult target distribution. A tempered restart happens when a sample from the reference successfully percolates to the target. (See the subsequent sections for a more detailed description of parallel tempering.) When the reference supports i.i.d. sampling, tempered restarts can enable large jumps in the state space.
- Λ: the global communication barrier, as described in Syed et al. (2021a), which measures the inherent difficulty of the sampling problem. A rule of thumb to configure the number of PT chains is also given by Syed et al. (2021a), where they suggest that stable performance should be achieved when the number of chains is set to roughly 2Λ. See Appendix A.2.6 for more information.
- `time` and `allc`: the time (in seconds) and number of allocations (in bytes) used in each round.
- $\log(Z_1/Z_0)$: an estimate of the logarithm of the ratio of normalization constants between the target and the reference. In many cases, $Z_0 = 1$.
- `min`(α) and `mean`(α): minimum and average swap acceptance rates during the communication phase across the PT chains.

A.2.2. Plots

It is straightforward to obtain plots of samples from the target distribution, such as trace-plots, pairwise plots, and density plots of the marginals.

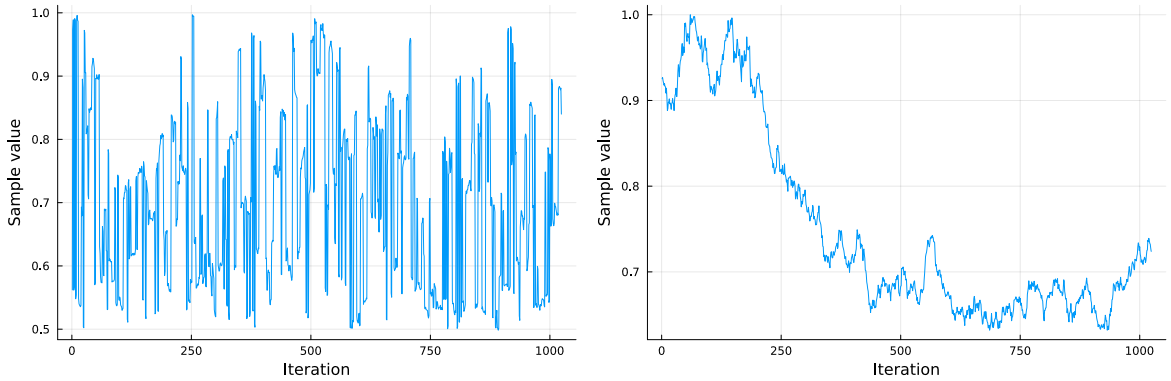

Figure 9: Trace-plots for the first parameter, p_1, in the non-identifiable coinflip Turing model. **Left:** Samples from Pigeons.jl using PT with 10 chains. Note that the trace-plot indicates fast mixing/exploration across the state space. **Right:** Single-chain Markov chain Monte Carlo. Note that the trace-plot explores the state space much more slowly when we do not use PT.

To obtain posterior densities and trace-plots, we first make sure that we have the third-party MCMCChains.jl[5], StatsPlots.jl[6], and PlotlyJS.jl[7] packages installed via

```
Pkg.add("MCMCChains", "StatsPlots", "PlotlyJS")
```

With the `pt` output object from before for our non-identifiable coinflip model, we can run the following:

```
using MCMCChains, StatsPlots, PlotlyJS
plotlyjs()
samples = Chains(
    sample_array(pt), variable_names(pt))
my_plot = StatsPlots.plot(samples)
display(my_plot)
```

The output of the above code chunk is an interactive plot that can be zoomed in or out and exported as an HTML webpage. A modified static version of the output for the first parameter, p_1, is displayed in the top panel of Fig. 9, along with a comparison to the output from a single-chain algorithm in the bottom panel of the same figure.

To obtain pair plots, we add the PairPlots.jl[8] and CairoMakie.jl[9] packages:

```
Pkg.add("PairPlots", "CairoMakie")
```

and then run

```
using PairPlots, CairoMakie
my_plot = PairPlots.pairplot(samples)
display(my_plot)
```

The output of the above code chunk with an increased number of samples is displayed in Fig. 7. [10]

[5] https://github.com/TuringLang/MCMCChains.jl
[6] https://github.com/JuliaPlots/StatsPlots.jl
[7] https://github.com/JuliaPlots/PlotlyJS.jl
[8] https://github.com/sefffal/PairPlots.jl
[9] https://github.com/JuliaPlots/CairoMakie.jl
[10] The code chunk above requires Julia 1.9.

A.2.3. ESTIMATE OF NORMALIZATION CONSTANT

The (typically unknown) constant Z in Eq. (1) is referred to as the *normalization constant*. In many applications, it is useful to approximate this constant. For example, in Bayesian statistics, this corresponds to the marginal likelihood and can be used for model selection.

As a side-product of PT, we automatically obtain an approximation to the natural logarithm of the normalization constant. This is done automatically using the stepping stone estimator (Xie et al., 2011). The estimate can be accessed using

```
stepping_stone(pt)
```

In the case of the normalization constant given by Eq. (4) for $n = 100,000$ and $y = 50,000$, we can exactly obtain its value as $\log(Z) \approx -11.8794$. Note that this is very close to the output provided in Fig. 8.

A.2.4. ONLINE STATISTICS

Pigeons has facilities to support cases requiring large memory. For instance, we allow for the computation of online statistics, as well as off-memory sample storage. Having specified the use of the `online` recorder in our call to `pigeons()`, we can output some basic summary statistics of the marginals of our target distribution. For instance, it is straightforward to estimate the mean and variance of each of the marginals of the target with

```
using Statistics
mean(pt); var(pt)
```

Other constant-memory statistic accumulators are made available in the OnlineStats.jl (Day & Zhou, 2020) package. To add additional constant-memory statistic accumulators, we can register them via `Pigeons.register_online_type()`, as described in our online documentation[11]. For instance, we can also compute constant-memory estimates of extrema of our distribution.

A.2.5. OFF-MEMORY PROCESSING

When either the dimensionality of the model or the number of samples is large, the obtained samples may not fit in memory. In some cases it may be necessary to store samples to disk if our statistics of interest cannot be calculated online and with constant-memory (see Appendix A.2.4). We show here how to save samples to disk when Pigeons.jl is run on a single machine. A similar interface can be used over MPI.

First, we make sure that we set `checkpoint = true`, which saves a snapshot at the end of each round in the directory `results/all/<unique directory>` and is symlinked to `results/latest`. Second, we make sure that we use the `disk` recorder by setting `record = [disk]`, along with possibly any other desired recorders. Accessing the samples from disk can then be achieved in a simple way using the Pigeons.jl function `process_sample()`.

A.2.6. PT DIAGNOSTICS

We describe how to produce some key parallel tempering diagnostics from Syed et al. (2021a).

The global communication barrier, denoted Λ in Pigeons.jl output, can be used to approximately inform the appropriate number of chains. Based on Syed et al. (2021a), stable PT performance should be achieved when the number of chains is set to roughly 2Λ. This can be achieved by modifying the `n_chains` argument in the call to `pigeons()`. The global communication barrier is shown at each round and can also be accessed with

```
Pigeons.global_barrier(pt)
```

The number of restarts per round can be accessed with

```
n_tempered_restarts(pt)
```

[11] https://pigeons.run/dev/

These quantities are also displayed in Fig. 8. Many other useful PT diagnostic statistics and plots can be obtained, as described in our full documentation.

A.3. Parallel and distributed PT

One of the main benefits of Pigeons.jl is that it allows users to easily parallelize and/or distribute their PT sampling efforts. We explain how to run MPI locally on one machine and also how to use MPI when a cluster is available.

A.3.1. Running MPI locally

To run MPI locally on one machine using four MPI processes and one thread per process, use

```
pigeons(
    target = TuringLogPotential(model),
    on = ChildProcess(
            n_local_mpi_processes = 4,
            n_threads = 1))
```

A.3.2. Running MPI on a cluster

Often, MPI is available via a cluster scheduling system. To run MPI over several machines:

1. In the cluster login node, follow the Pigeons.jl installation instructions in our online documentation.
2. Start Julia in the login node, and perform a one-time setup by calling `Pigeons.setup_mpi()`.
3. In the Julia REPL running in the login node, run:[12]

```
pigeons(
    target = TuringLogPotential(model),
    n_chains = 1_000,
    on = MPI(n_mpi_processes = 1_000,
             n_threads = 1))
```

The code above will start a distributed PT algorithm with 1,000 chains on 1,000 MPI processes each using one thread. Note that for the above code chunks involving `ChildProcess()` and `MPI()` to work, it may be necessary to specify dependencies in their function calls. See `https://pigeons.run/stable/mpi/` for details.

A.4. Additional options

In the preceding example we only specified the target distribution and let Pigeons.jl decide on default values for most other settings of the inference engine. There are various settings we can change, including: the random seed (`seed`), the number of PT chains (`n_chains`), the number of PT tuning rounds/samples (`n_rounds`), and a variational reference distribution family (`variational`), among other settings. For instance, we can run

```
pigeons(
    target = TuringLogPotential(model),
    n_rounds = 10,
    n_chains = 10,
    variational = GaussianReference(),
    seed = 2
)
```

which runs PT with the same Turing model target as before and explicitly states that we should use 10 PT tuning rounds with 10 chains (described below). In the above code chunk we also specify that we would like to use a Gaussian variational reference distribution. That is, the reference distribution is chosen from a multivariate Gaussian family that lies as close as possible to the target distribution in order to improve the efficiency of PT. We refer readers to Surjanovic et al. (2022) for more details. When only continuous parameters are of interest, we encourage users to consider using `variational = GaussianReference()` and setting `n_chains_variational = 10`, for example, as the number of restarts may substantially increase with these settings.

[12] In version 0.4.0 of Pigeons, the function MPI has been renamed MPIProcesses to avoid a clash with the library MPI.jl.