

PyGFMC: GPU-accelerated Green's Function Monte Carlo

Robin Schäfer

Department of Physics, Harvard University, Cambridge, MA, USA

robin_schaefer@fas.harvard.edu

Abstract

PyGFMC is a JAX-based implementation of Green's function Monte Carlo for ground states of sign-problem-free quantum lattice models. On GPUs it reaches nearly three orders of magnitude higher throughput than on CPUs, and is scalable across multiple devices. This enables ensembles of up to 10^7 walkers and expressive guiding wavefunctions, including high-dimensional neural quantum states. As a result, PyGFMC can access ground states of systems in which other quantum Monte Carlo methods struggle to equilibrate. We demonstrate its capabilities on the notoriously difficult problem of quantum spin ice. The package provides several built-in Hamiltonians and guiding wavefunctions that can be trained using NetKet. It computes both diagonal and off-diagonal observables. Documentation and code are available at <https://github.com/robin-schaefer/PyGFMC>.

Contents

1	Introduction	2
2	Green's Function Monte Carlo	3
2.1	Walker propagation	3
2.2	Branching	4
2.3	Propagator	5
2.4	Guiding wavefunction	6
2.5	Observables	7
2.6	Convergence and error control	8
2.6.1	Sources of error	8
2.6.2	Control parameters	10
3	Examples	12
3.1	Transverse-field Ising model	12
3.2	Nearest-neighbor XXZ on the pyrochlore lattice	14
3.3	Ring-exchange on the pyrochlore lattice	16
4	Package structure and API	16
4.1	Hamiltonian	17
4.2	Walker ensemble	20
4.3	Guiding wavefunction (optional)	20
4.4	Observables (optional)	22
4.5	Simulation driver	23
5	Discussion	26

References	27
A Arguments for the production run	31
B Validation	33
B.1 Core tests	33
B.2 Benchmark against other methods	34
C Convergence in the pyrochlore XXZ model	35
D Implemented operator classes	35
E User-defined operator classes	37
E.1 An n -body Hamiltonian	38
E.2 $S^z S^z$ structure factor	38
F Implemented guiding wavefunctions	39
G User-defined guiding wavefunctions	41
H NetKet integration	41

1 Introduction

Quantum Monte Carlo methods [1, 2] hold the promise of efficiently solving sign-problem-free quantum many-body problems. Where they apply, their statistical approach makes them the method of choice. They are neither limited to small system sizes, as exact diagonalization [3–7] is, nor to low dimensionality and area-law entanglement, as tensor network methods [8–10] are. They also avoid the challenging variational optimization on which neural quantum states [11–13] rely. However, while efficient in principle, much of the exotic physics encountered in sign-problem-free models, such as quantum spin liquids [14–21] or spin glasses [22–27], can pose severe challenges in practice. In particular, when nearly degenerate low-energy states are present, reaching low temperatures or the ground state requires long equilibration times that grow rapidly with system size. Consequently, obtaining clean statistical data may require substantial computational resources.

We present PyGFMC, a Python library for quantum Monte Carlo, aimed at sign-problem-free models that are hard to equilibrate. A Hamiltonian H is sign-problem-free if

$$H_{x,x'} = \langle x | H | x' \rangle \leq 0 \quad (1)$$

for all basis states $|x\rangle \neq |x'\rangle$. PyGFMC implements the Green’s function Monte Carlo (GFMC) algorithm [28–30], a stochastic projection onto the ground state (see also Refs. [31, 32]). By performing a walk with population control, GFMC samples the ground state and provides access to diagonal and off-diagonal observables. Its advantage over other quantum Monte Carlo methods is the ability to include arbitrarily complex guiding wavefunctions, which are either chosen by hand or optimized numerically beforehand [33–35]. These approximate the ground state and thereby improve the performance substantially.

PyGFMC is written in JAX [36, 37], allowing parallelized execution on CPUs and GPUs. GPU acceleration achieves a speedup of roughly *three orders of magnitude* over the CPU baseline

and is scalable across multiple GPUs, making walker populations of up to 10^7 possible. This hardware-based improvement makes more expressive guiding wavefunctions possible. The GPU compatibility naturally suits the NetKet [38, 39] infrastructure, which provides access to generic variational states and training environments for guiding wavefunctions. These can be included in PyGFMC through its modular design, which likewise allows users to add their own Hamiltonian classes. The following Hamiltonian classes are pre-implemented:

- generic spin- S Hamiltonian (including an optimized class for $S = 1/2$)
- Bose–Hubbard model
- Kitaev’s toric code model [40]
- Rokhsar–Kivelson model [41]

The source code, installation instructions, example notebooks, and the documentation are available at <https://github.com/robin-schaefer/PyGFMC>.

The paper is organized as follows: **Section 2** provides a brief overview of the GFMC algorithm, including the role of the guiding wavefunction, and the estimation of diagonal and off-diagonal observables. It also discusses different sources of errors and how to mitigate them by adjusting the control parameters of the algorithm. **Section 3** applies PyGFMC to concrete models. First, as a proof of concept, we solve the one-dimensional transverse-field Ising model and discuss the convergence. Second, we tackle the notoriously difficult, three-dimensional, and frustrated problem of quantum spin ice [42–45]. For small, ferromagnetic transverse couplings, it hosts a $U(1)$ quantum spin liquid. We study the phase transition from the liquid into the ferromagnet, which is challenging to equilibrate at low and zero temperature for quantum Monte Carlo due to the small transverse exchange parameters [19, 20]. Furthermore, we study excitations in the effective model describing the spin liquid. **Section 4** documents the package layout and walks through the API. Lastly, we conclude in **Section 5**.

2 Green’s Function Monte Carlo

The GFMC algorithm is a stochastic projection onto the ground state of sign-problem-free Hamiltonians, using a fixed number of walkers [1, 28, 29, 46]. Unlike most other quantum Monte Carlo methods, such as the stochastic series expansion [2, 47, 48], GFMC is a ground-state method and does not give access to finite-temperature properties. However, excited states can be computed in different symmetry sectors of the Hamiltonian, which are, for example, defined by the particle number or a topological quantity (see **Section 3.3**).

We give a brief overview of the algorithm, important details within our implementation, sources of errors, and the control knobs needed for good convergence. For further reading we recommend Refs. [1, 29].

2.1 Walker propagation

A fixed number of walkers sample the ground-state amplitudes by exploring different configurations in the Hilbert space. The algorithm can be understood by means of the power method. We define the nonnegative matrix

$$T = \lambda \mathbb{1} - H. \quad (2)$$

Let $H|\Psi_0\rangle = E_0|\Psi_0\rangle$ be the ground state of H . For sufficiently large λ , $|\Psi_0\rangle$ is the eigenstate of T with the largest eigenvalue in magnitude, and can therefore be obtained by applying T a

total of p times to a random initial state $|\Psi\rangle$ with $|\langle\Psi|\Psi_0\rangle| > 0$:

$$T^p |\Psi\rangle \xrightarrow{p \rightarrow \infty} |\Psi_0\rangle \quad (3)$$

which converges to the ground state for sufficiently large p .

The projection is carried out by a population of n_{walker} *walkers* represented in a basis of the Hamiltonian, e.g., the computational S^z basis. Each application of T acts as a stochastic process whose matrix entries determine the transition probabilities of every walker. Given a current configuration $|x\rangle$, the probabilities of transitioning to a new configuration $|x'\rangle$ or remaining in the same configuration are

$$p_{x',x} = -\frac{h_{x',x}}{b_x} \geq 0 \quad \text{and} \quad p_{x,x} = \frac{\lambda - h_{x,x}}{b_x} \geq 0 \quad \text{with} \quad b_x = \lambda - \sum_{x'} h_{x',x}, \quad (4)$$

where $h_{x',x} = \langle x' | H | x \rangle$ are the matrix elements of the Hamiltonian connecting $|x\rangle$ to $|x'\rangle$. The sum runs over all x' (including $x' = x$). We refer to each application of T as a *transition*, whether or not the configuration changes.

The probabilistic interpretation requires the positivity of all entries of T , which is guaranteed by the sign-problem-free property of the Hamiltonian: $-h_{x',x} \geq 0$ for $|x\rangle \neq |x'\rangle$. λ must be chosen large enough that $\lambda - h_{x,x} \geq 0$ for every configuration visited, allowing mixed signs in the diagonal part of the Hamiltonian.

Since the transition matrix is not stochastic, correct sampling requires a weight attached to each walker given by the column sum b_x . For each transition out of $|x\rangle$ (whether or not it enters a new configuration), the walker acquires a weight b_x that must be tracked. Note that if T were a truly stochastic matrix (nonnegative entries with columns summing to one), every walker would acquire the same weight and the weights would not need to be considered.

For large enough p , the walker ensemble, weighted by these accumulated factors, samples the amplitudes of the ground-state wavefunction $|\Psi_0\rangle$. Note that the sign-problem-free structure of H guarantees that the ground-state wavefunction has no relative signs or phases.

2.2 Branching

Different walkers accumulate different weights over the course of their evolution. Since each weight is a product over many transitions, the spread of the weights grows exponentially, and a few walkers dominate the ensemble while most others contribute only negligibly. This leads to a collapse of the effective sample size. To circumvent this, the population is systematically controlled and walkers that have accumulated large weights are replicated, while those with small weights are removed [49, 50]. The ensemble size is fixed.

The branching step is performed every n_{step} transitions, by which point the i -th walker has accumulated the weight

$$w_i = \prod_{k=1}^{n_{\text{step}}} b_{x_{i,k}}, \quad (5)$$

where $b_{x_{i,k}}$ is the column sum encountered at the k -th transition among the walker's last n_{step} transitions. We use a *correlated* branching scheme where a single random number $\zeta \sim U(0, 1)$ generates the n_{walker} quantiles $z_\alpha = (\alpha + \zeta)/n_{\text{walker}}$ for $\alpha = 0, \dots, n_{\text{walker}} - 1$, and the new ensemble is drawn from the cumulative distribution induced by the weights $\{w_i\}$ at these $\{z_\alpha\}$. This systematic scheme reduces the variance without introducing bias [1, 29]. Notably, when all weights are equal every walker survives the branching step unchanged.

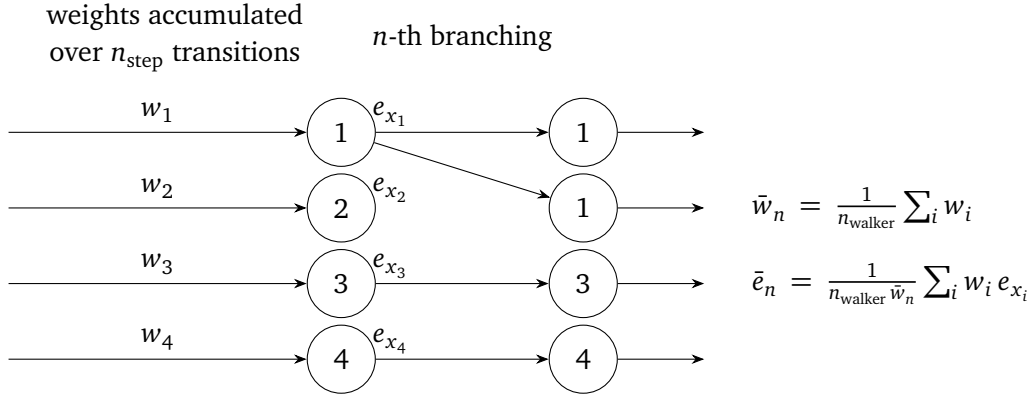


Figure 1: Illustration of a branching step on a four-walker ensemble. The weights w_i above the incoming arrows are accumulated over the preceding n_{step} transitions. Each walker also carries a local energy $e_{x_i} = \lambda - b_{x_i}$. From these the average weight $\bar{w}_n$ and weighted local energy $\bar{e}_n$ of Equation 6 are computed and recorded for the energy estimator Equation 7. The correlated reconfiguration then redraws n_{walker} walkers from the distribution of weights: walker 1 has a large weight and is replicated (two outgoing arrows); walker 2 has a small weight and is dropped (no outgoing arrow).

Before the n -th branching step we compute the average weight $\bar{w}_n$ across the ensemble and the weighted local-energy estimator $\bar{e}_n$:

$$\bar{w}_n = \frac{1}{n_{\text{walker}}} \sum_{i=1}^{n_{\text{walker}}} w_i, \quad \bar{e}_n = \frac{1}{n_{\text{walker}} \bar{w}_n} \sum_{i=1}^{n_{\text{walker}}} w_i e_{x_i} \quad \text{with} \quad e_{x_i} = \lambda - b_{x_i}, \quad (6)$$

where b_{x_i} is evaluated on the current configuration of the i -th walker. The weights are set to unity after the branching. Figure 1 illustrates the branching step on a four-walker example.

The final ground-state energy estimate as a function of p branching steps is obtained from a single Markov chain by evolving the walker ensemble and recording $(\bar{w}_n, \bar{e}_n)$ at every branching step:

$$E_0[p] = \frac{\sum_n G_n \bar{e}_n}{\sum_n G_n} \quad \text{with} \quad G_n = \prod_{j=0}^{p-1} \bar{w}_{n-j}. \quad (7)$$

Note that from here on p counts *branching steps* rather than individual applications of T .

2.3 Propagator

PYGFMC offers two schemes for accelerating walker propagation. One is discrete and based on the single-step walker update described above, whereas the other is formally obtained in the limit $\lambda \rightarrow \infty$, leading to a continuous imaginary-time variable.

Discrete formulation. Using the diagonal-stay probability $p_d = p_{x,x}$ from Equation 4, we sample the number k of consecutive diagonal moves before the walker finally transitions to an off-diagonal configuration. k is geometrically distributed and can be sampled via

$$k = \left\lfloor \frac{\log(1-\psi)}{\log p_d} \right\rfloor, \quad (8)$$

where $\psi \sim U(0, 1)$ and $\lfloor \dots \rfloor$ denotes the floor function. The walker accumulates the weight b_x^{k+1} from k diagonal moves and the subsequent off-diagonal transition, after which it is propagated to a new configuration $|x'\rangle$ drawn from the off-diagonal part of Equation 4. This procedure repeats until n_{step} substeps are reached.

Continuous formulation. In the limit of $\lambda \rightarrow \infty$ with fixed imaginary-time length $\beta = p/\lambda$ the discrete power method turns into the exact imaginary-time propagator $e^{-\beta H}$,

$$(\lambda \mathbb{1} - H)^p = \lambda^p \left(\mathbb{1} - \frac{H}{\lambda} \right)^{\lambda \beta} \xrightarrow{\lambda \rightarrow \infty} \lambda^p e^{-\beta H}, \quad (9)$$

so that p steps realize an imaginary-time evolution $e^{-\beta H}$ up to a configuration-independent prefactor λ^p that cancels in the end. In this limit, the discrete hold count of Equation 8 becomes a continuous variable,

$$d\beta = \frac{\log(1 - \psi)}{\sum_{x' \neq x} h_{x',x}}, \quad (10)$$

where $\sum_{x' \neq x} h_{x',x} < 0$ is the sum over all off-diagonal elements from $|x\rangle$. Since $\log(1 - \psi) < 0$, the hold time $d\beta$ is positive. Over a duration of $d\beta$ the walker accumulates the weight $e^{-e_x d\beta}$, after which it transitions to a new configuration with the same off-diagonal probabilities as in the discrete formulation. The procedure repeats until the imaginary-time step β is reached. Note that λ no longer enters the formalism.

PyGFMC implements three different propagators, which can be chosen using the `"method"` keyword:

- `"single_step"`: the elementary single step.
- `"multi_step"`: multiple discrete steps at once (discrete formulation).
- `"cont_time"`: continuous imaginary-time variable (continuous formulation).

Neither accelerating scheme (`"multi_step"` or `"cont_time"`) is GPU-optimal, since both rely on a `while`-loop. However, both still deliver substantial speedups whenever the diagonal-stay probability p_d of Equation 4 is non-negligible.

2.4 Guiding wavefunction

The performance of the algorithm can be significantly improved by providing a guiding wavefunction ψ_T that approximates the ground-state amplitudes, $\psi_T(x) \approx \langle x | \Psi_0 \rangle$. In some rare cases the ansatz can also eliminate a sign problem present in the original basis, for instance via the Marshall sign rule [51].

The guiding wavefunction alters the transition probabilities of Equation 4 to

$$p_{x',x} = -\frac{h_{x',x}}{b_x} \longrightarrow -\frac{\psi_T(x')}{\psi_T(x)} \frac{h_{x',x}}{\tilde{b}_x}, \quad \tilde{b}_x = \lambda - \sum_{x'} \frac{\psi_T(x')}{\psi_T(x)} h_{x',x}. \quad (11)$$

Similarly, the energy estimator and observable expectation values (see below) pick up the same factor $\psi_T(x')/\psi_T(x)$. Crucially, the algorithm requires only the *ratio* of guide amplitudes between two configurations. Within PyGFMC, the guiding-wavefunction class returns the log-difference $\log[\psi_T(x')/\psi_T(x)] = \log \psi_T(x') - \log \psi_T(x)$ for every possible transition of the Hamiltonian.

When ψ_T is close to the true ground state, statistical fluctuations are suppressed. In the limit $\psi_T(x) = |\langle x | \Psi_0 \rangle|$, the fluctuations in the local energy and branching weight vanish entirely. In this ideal case every walker carries the same weight and the same local-energy estimator, equal to the ground-state energy itself.

2.5 Observables

The walker ensemble lets us evaluate diagonal and off-diagonal observables in the ground state. We begin by discussing the forward-walker technique required to compute off-diagonal expectation values. Diagonal observables can be evaluated within the main Markov chain, reducing computational cost significantly.

Off-diagonal observables. To measure an off-diagonal operator O , a fresh short Markov chain of length $p/2$ must be launched. Let $|x\rangle$ be the current configuration of a walker. The operator O is applied stochastically, and a target configuration $|x'\rangle$ is drawn with probability

$$p_{x',x} = \frac{\psi_T(x')}{\psi_T(x)} \frac{o_{x',x}}{o_x}, \quad o_x = \sum_{x'} \frac{\psi_T(x')}{\psi_T(x)} o_{x',x}, \quad (12)$$

where $o_{x',x} = \langle x' | O | x \rangle$ are the matrix elements of O . The configuration of the walker is updated accordingly and the estimator o_x is assigned to it. The updated ensemble is then evolved under the Hamiltonian for $p/2$ branching steps. At each branching event a replicated walker duplicates its estimator o_x , while a discarded walker's estimator is lost. This procedure is sketched in [Figure 2](#), where four walkers are evolved over two branching events, and the estimator values attached to each walker propagate according to the reconfigurations of the auxiliary chain. After $p/2$ steps we record the mean of the inherited o_x values

$$\bar{o}_{n-p/2} = \frac{1}{n_{\text{walker}}} \sum_{i=1}^{n_{\text{walker}}} o_{x_i(n)}, \quad (13)$$

together with the branching-weight product of the auxiliary chain

$$G_n^{\text{aux}} = \prod_{j=0}^{p/2-1} \bar{w}_{n-j}^{\text{aux}} = \bar{w}_{n-p/2+1}^{\text{aux}} \cdots \bar{w}_n^{\text{aux}}. \quad (14)$$

The final estimator then resembles [Equation 7](#), but combines $p/2$ weights of the main chain with $p/2$ weights of the auxiliary chain:

$$O[p] = \frac{\sum_n G_n^{\text{half}} G_n^{\text{aux}} \bar{o}_{n-p/2}}{\sum_n G_n} \quad \text{with} \quad G_n^{\text{half}} = \prod_{j=p/2}^{p-1} \bar{w}_{n-j}. \quad (15)$$

G_n^{half} denotes the half of G_n covering the steps $n-p+1, \dots, n-p/2$. The other half is obtained from the auxiliary-chain weights G_n^{aux} . The main-chain G_n in the denominator is the same as in [Equation 7](#).

The branching in [Equation 12](#) requires nonnegative matrix elements of O . If O has negative matrix elements, it can be decomposed into positive and negative parts, $O = O_{\text{pos}} + O_{\text{neg}}$, and each contribution is computed separately:

$$\langle O \rangle = \langle O_{\text{pos}} \rangle - \langle -O_{\text{neg}} \rangle. \quad (16)$$

The same trick handles complex operators by splitting them into real and imaginary parts.

Note that launching an auxiliary chain to measure the off-diagonal operator increases the runtime by a factor of $p/2$ per observable.

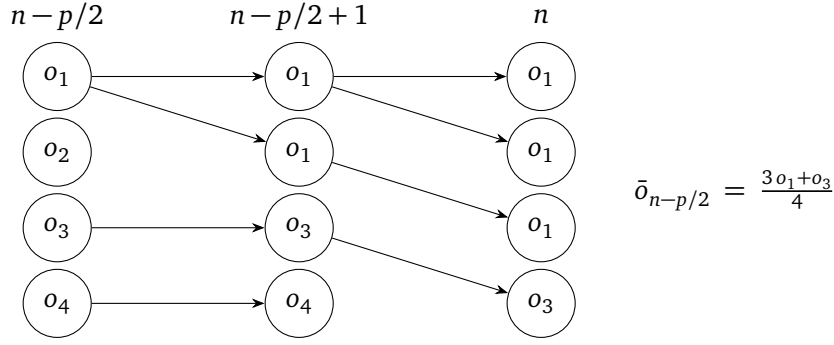


Figure 2: Forward-walking estimator illustrated for four walkers over $p/2 = 2$ branching steps of the auxiliary chain. The operator is applied at step $n - p/2$, attaching the estimator o_x to each walker. The estimators are updated following the reconfiguration at every step, where replicated walkers inherit their parent's o_x . The walker mean of the surviving estimators at step n defines $\bar{o}_{n-p/2}$ in Equation 13.

Diagonal observables. No auxiliary Markov chain is needed to compute the expectation value of a diagonal operator O since it does not trigger a transition. We can simply obtain the estimator from the main Markov chain. At branching step $n - p/2$ we attach to every walker i its current diagonal element $o_{x_i} \equiv \langle x_i | O | x_i \rangle$ and follow these estimators through the next $p/2$ branching events of the main chain. The walker mean at step n defines $\bar{o}_{n-p/2}$ as in Equation 13, and the diagonal estimator $O[p]$ has the same form as Equation 15 except that the auxiliary-chain product G_n^{aux} is replaced by the corresponding main-chain product $\prod_{j=0}^{p/2-1} \bar{w}_{n-j}$ since both halves of the projection are along the main Markov chain:

$$O[p] = \frac{\sum_n G_n \bar{o}_{n-p/2}}{\sum_n G_n}. \quad (17)$$

No probabilistic branching of O is involved here, so the diagonal pipeline imposes no positivity restriction on O .

2.6 Convergence and error control

Obtaining well-converged results is difficult. It requires understanding why a simulation has failed (or is converging poorly) and how to improve it. Therefore, we first discuss the two main sources of error and how to identify them, and then explain how to mitigate them.

2.6.1 Sources of error

The two sources of error are (i) a systematic bias from an insufficient projection depth and (ii) statistical uncertainty.

(i) Insufficient projection. The energy and every observable must be examined as a function of projection steps (branching events). The estimator initially drifts until it settles to a plateau. The data have converged when the plateau is stable over several consecutive values of p . However, when the number of projection steps p is pushed too large, the plateau deteriorates, either through a slow drift or rapid changes. This signals a failure of convergence, and the data cannot be trusted (see below). Importantly, the deterioration can happen before the plateau is reached, meaning that the simulation is not capable of reliably resolving the ground state with the current settings.

(ii) **Statistical errors.** The deterioration comes from a population collapse in which too many walker lineages have died out, ruining the statistics at deep projection depths. In fact, after sufficiently many branching events, the entire population descends from a single initial walker. PyGFMC provides diagnostics for estimating the largest reliably accessible projection depth. These can be obtained using the `statistics` function that analyzes the evolution of a given walker population. It returns a dictionary `stats` (and an evolved walker ensemble). The two most important entries are:

- `stats['unique_fraction']`: the fraction of unique configurations present after each branching step. For the correlated branching scheme employed here, we recommend that this fraction remain above roughly 0.9. When the step size between two branching events is too large, a few walkers dominate the evolution and many others die out. Therefore, the number of distinct configurations decays. Conversely, when the step size is too small, the walkers have too little time to accumulate different weights and essentially none die out. The fraction stays close to 1.
- `stats['effective_sample_size']`: the effective sample size counts how many of the initial walkers effectively contribute descendants after a given number of branching steps. We define c_i as the number of present walkers descending from the i -th walker. The effective sample size is

$$n_{\text{eff}}(p) = \frac{(\sum_i c_i)^2}{\sum_i c_i^2} = \frac{n_{\text{walker}}^2}{\sum_i c_i^2} \in [1, n_{\text{walker}}]. \quad (18)$$

If the entire population has collapsed after p steps (with $c_i = n_{\text{walker}}$ for one i and zero otherwise), $n_{\text{eff}} = 1$, indicating that p has been pushed too large. If, on the other hand, no walker has died out after each step ($c_i = 1$ for all i), $n_{\text{eff}} = n_{\text{walker}}$ and the statistics are optimal. In the single-step example of Figure 1, we have $c_1 = 2$, $c_2 = 0$, and $c_3 = c_4 = 1$, leading to $n_{\text{eff}}(1) = 16/6$. We recommend that $n_{\text{eff}} \gtrsim 50$ in order to obtain sufficient statistics. However, this may vary between different observables and models. Note that an observable is measured at the midpoint of its projection window, so the relevant depth is $m = p/2$ rather than the full p .

The other source of statistical noise is an insufficient number of measurements. The algorithm is run for n_{steps} branching steps, which must be chosen large enough to resolve the numerator and denominator of the estimator to the required precision. For a well-adjusted simulation, the error bar decreases as $1/\sqrt{n_{\text{steps}}}$, where n_{steps} is the number of measurements (branching steps) performed. However, since consecutive measurements are correlated, only a reduced, effective number (defined through the autocorrelation time [52]) of measurements is statistically independent.

The autocorrelation time is returned by the production run (see Section 4.5) as part of the `stats` dictionary. It is computed for the logarithmic weights, the energy estimator, and every measured observable. Given a measurement series O_n of length n_{steps} , we compute the normalized autocorrelation function

$$\rho(t) = \frac{C(t)}{C(0)}, \quad C(t) = \frac{1}{n_{\text{steps}} - t} \sum_n (O_n - \bar{O})(O_{n+t} - \bar{O}), \quad (19)$$

and sum it to a cutoff M ,

$$A(M) = \frac{1}{2} + \sum_{t=1}^M \rho(t). \quad (20)$$

We choose M self-consistently as the smallest value M satisfying $M \geq 5A(M)$. The effective number of measurements is reduced to $n_{\text{steps}}/(2A)$. The values are stored in `stats` under:

```
stats['autocorrelation/energy_estimator']      # scalar
stats['autocorrelation/log_weights']           # scalar
stats['autocorrelation/diag/<name>']           # one entry per p
stats['autocorrelation/off_diag/<name>']       # one entry per p
```

The observables are listed per number of branching steps p , which corresponds to the averaged estimator after $m = p/2$ branching steps. Note that this is just the observable estimator and does not include the weight obtained over these m steps.

PYGPMC estimates the statistical error from a single Markov chain using a blocked jackknife [53, 54] by calling `evaluate_observables` (see Section 4.5). We divide the entire Markov chain into n_{blocks} contiguous blocks. For $b = 1, \dots, n_{\text{blocks}}$, we compute the estimator by deleting the b -th block from the Markov chain. We define this reduced estimator as Θ_b . The jackknife error is obtained from the variance of these estimates,

$$\sigma^2 = \frac{n_{\text{blocks}} - 1}{n_{\text{blocks}}} \sum_{b=1}^{n_{\text{blocks}}} (\Theta_b - \bar{\Theta})^2, \quad \bar{\Theta} = \frac{1}{n_{\text{blocks}}} \sum_b \Theta_b. \quad (21)$$

However, we recommend performing several independent simulations and taking the standard error of their mean to obtain error bars on an observable or on the ground-state energy.

2.6.2 Control parameters

The performance of a GPMC calculation can be greatly improved by carefully adjusting the simulation parameters. We discuss these control parameters and how to tune them, but first emphasize an important point: The quality of a simulation, especially for systems that are difficult to equilibrate, depends crucially on the guiding wavefunction. A good variational ansatz is the key to tackling challenging problems, and effort should be invested in optimizing it before undertaking large-scale simulations.

Projection depth. The projection depth τ is determined by the product of the branching interval (n_{step} in the discrete method or β in the continuous method) and the number of branching steps p divided by 2:

$$\tau = \frac{p}{2} n_{\text{step}} \quad (\text{discrete}), \quad \tau = \frac{p}{2} \beta \quad (\text{continuous}). \quad (22)$$

Note that τ in the discrete and continuous methods should not be compared directly.

The goal is to reach the largest possible τ while maintaining good statistics. The statistics at a given τ depend on the step size between branching steps, which must balance two competing demands: First, if it is too small, walkers do not have sufficient time to accumulate weight, and branching cannot efficiently remove low-weight walkers. Second, if it is too large, weight accumulates excessively and only a few walkers survive each reconfiguration.

To identify a suitable step size in advance, we recommend scanning a range of step sizes and maximizing the effective sample size $n_{\text{eff}}(\tau)$ from Equation 18 as a function of τ . We perform this optimization in Figure 4(a), which shows n_{eff} as a two-dimensional heat map on a log scale as a function of step size β (continuous propagation) and projection depth τ . It includes contour lines to guide the eye. We find that, for a given τ , there is an optimum around $\beta \approx 0.15$, and n_{eff} decreases as β increases from approximately 0.25 to 1. Note that the computational cost of propagation is usually lower for larger values of β , and the aim is typically to pick the largest value of β with sufficiently large n_{eff} .

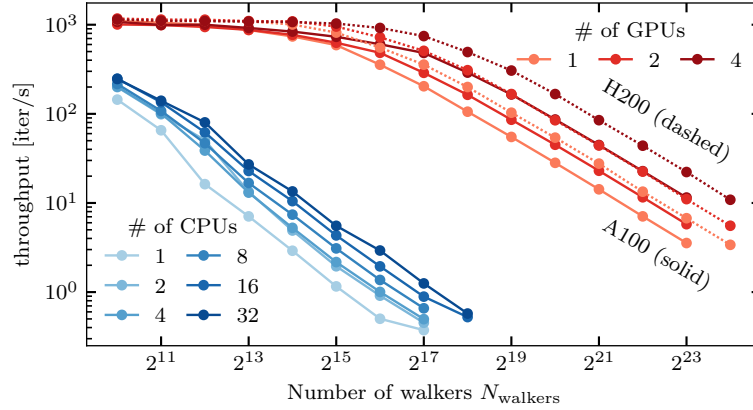


Figure 3: Throughput in iterations per second as a function of the walker population n_{walker} for the one-dimensional Heisenberg chain at $L = 100$, $n_{\text{step}} = 6$, and $\lambda = 25$. Blue points: CPU runs on an Intel Xeon Platinum 8480 using 1 to 32 cores. Red points: GPU runs on 1, 2, and 4 NVIDIA A100 (solid) and H200 (dashed) devices. Each point is averaged over 10 independent runs.

Number of walkers n_{walker} . The number of walkers is the simplest control parameter: a larger population generally yields better statistics and allows larger projection depths. On modern GPUs, PYGFMC can reach up to $n_{\text{walker}} \sim 10^7$, which is demonstrated in Figure 3. However, blindly scaling up the population is not the best strategy, as it is often more effective to invest in a better guiding wavefunction. Also note that beyond 10^7 , the limited resolution of `float32` starts to affect the cumulative weight distribution used in the branching step.

Diagonal shift λ . The diagonal shift λ is only present in the discrete methods, and it must be at least as large as the largest diagonal matrix element encountered to ensure positivity. A larger λ increases the diagonal-stay probability p_d , so that walkers require many transitions before changing their configuration.

The shift can be set to a value smaller than the maximal diagonal element of the Hamiltonian as long as the configurations with a larger value are practically never visited. Take a ferromagnet as an example, where the antiferromagnetic Néel configuration is highly unlikely and has an exponentially suppressed amplitude. Hence, it is practically never visited within a finite simulation, and the λ value can be safely reduced.

Guiding wavefunction. The key ingredient for tackling problems that are hard to equilibrate is a good ansatz wavefunction [33–35, 55]. The following criteria should be considered:

- The ansatz should be close to the true ground state (the overlap should be maximized). A simple benchmark for this is a minimal variational energy.
- It should not strongly suppress configurations that are required to access different regions of Hilbert space.
- The ratio between amplitudes that are connected by the Hamiltonian should not be too large, as this leads to large fluctuations in the simulation.
- Symmetries of the Hamiltonian should be respected.
- The amplitudes of the wavefunction are evaluated at every possible transition, so they must be cheap to compute. Since the algorithm only requires the ratio between the

amplitudes of two states connected by a few-body Hamiltonian, we recommend Jastrow-type wavefunctions [56–58] of the form:

$$\log \psi_T(x) = \sum_{c \in \mathcal{C}} \Phi_c(c[x]), \quad (23)$$

where the sum runs over a collection $\mathcal{C}$ of clusters c . $c[x]$ refers to the configuration of $|x\rangle$ on this cluster, and Φ_c maps a cluster configuration to a real number.

Each Hamiltonian term typically acts on only a small number of sites, so the logarithmic difference between two configurations $|x\rangle$ and $|x'\rangle$ involves only the subset $\mathcal{C}' \subset \mathcal{C}$ of clusters that contain the updated sites:

$$\log \psi_T(x) - \log \psi_T(x') = \sum_{c \in \mathcal{C}'} [\Phi_c(c[x]) - \Phi_c(c[x'])]. \quad (24)$$

The bulk of the clusters in $\mathcal{C}$ remains unchanged, which drastically reduces the computational cost.

The Jastrow-type ansatz further allows for an easy implementation of symmetries, such as translations, by simply setting $\Phi_c = \Phi$ for all c .

3 Examples

We demonstrate the capabilities of PyGFMC with three examples. First, we study the one-dimensional transverse-field Ising model to benchmark the convergence of observables (Section 3.1). Second, we resolve the transition between a $U(1)$ spin liquid and a ferromagnet on the three-dimensional pyrochlore lattice. This problem is notoriously difficult to equilibrate due to its small off-diagonal matrix elements (Section 3.2). Third, we compute string excitations in the $U(1)$ quantum spin liquid phase on the pyrochlore lattice (Section 3.3).

3.1 Transverse-field Ising model

We study the one-dimensional transverse-field Ising model on a periodic chain with $N = 100$ sites:

$$H_{\text{TFIM}} = -J_z \sum_i \sigma_i^z \sigma_{i+1}^z - h_x \sum_i \sigma_i^x, \quad (25)$$

with $J_z = 1$. The model has a zero-temperature phase transition at the critical point $h_x = J_z$, separating a ferromagnetic ($h_x < J_z$) from a paramagnetic ($h_x > J_z$) phase.

We use a single-parameter, nearest-neighbor Jastrow ansatz as a guiding wavefunction:

$$\log \psi_T(x) = -\beta_{\text{Jastrow}}(h_x) \sum_i \sigma_i^z \sigma_{i+1}^z. \quad (26)$$

We cap its magnitude at $|\beta_{\text{Jastrow}}| \leq 0.3$ to prevent the walker population from freezing. This affects the values in the ferromagnetic phase for $h_x \leq 0.9$. In the paramagnetic phase, $\beta_{\text{Jastrow}} = 0$ (or equivalently, no guiding wavefunction) is a good ansatz, as it becomes exact for $h_x/J_z \rightarrow \infty$. The model is exactly solvable and serves as our benchmark [59].

Figure 4 presents results at the critical point and examines the optimal choice of step size β for continuous-time propagation. Panel (a) shows the effective sample size n_{eff} from Equation 18 (median over 16 simulations) as a function of the step size β (x-axis) and projection depth $\tau = m\beta$ (y-axis) on a logarithmic scale. We find that survival is optimal (n_{eff} is maximal)

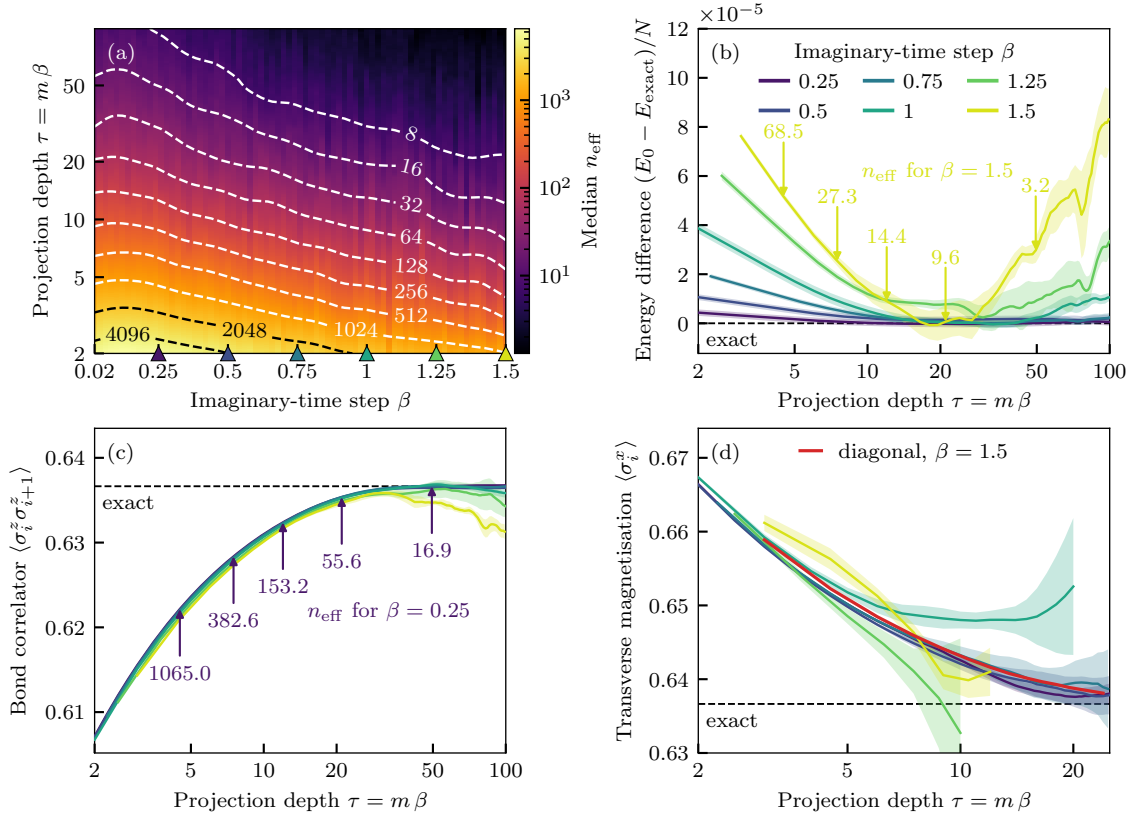


Figure 4: Data for the transverse-field Ising model in Equation 25 at the critical point with $N = 100$ on a periodic chain. Panel (a) shows the effective sample size from Equation 18 (median over 16 independent runs) on a logarithmic scale as a function of step size β (continuous propagation) and the total projection depth $\tau = m\beta$. Panel (b) shows the difference from the exact ground-state energy for six different step sizes as a function of total projection depth. It further indicates n_{eff} at different values of τ for $\beta = 1.5$. Panel (c) shows the averaged nearest-neighbor $(1/N) \sum_i \sigma_i^z \sigma_{i+1}^z$ correlator and marks n_{eff} for $\beta = 0.25$. The data in panels (b) and (c) were obtained using $n_{\text{walker}} = 2^{17}$ and $n_{\text{steps}} = 2 \cdot 10^5$. Panel (d) shows the transverse magnetization $(1/N) \sum_i \sigma_i^x$ per site, which has been computed using the off-diagonal forward-walking technique. These data were obtained with only $n_{\text{walker}} = 2^{15}$ and $n_{\text{steps}} = 64 \cdot 10^4$, and an off-diagonal stride of 64 (10^4 measurements in total). The red line shows the magnetization computed from the energy and the correlator using the data from panels (b) and (c) for $\beta = 1.5$. Error bars show the standard error of the mean over five independent runs.

near $\beta \approx 0.15$ for the tested values of τ . The effective sample size decreases as β increases from approximately 0.25 to 1.

Panel (b) shows the convergence of the energy as a function of projection depth for the step sizes marked on the axis of panel (a). Remarkably, the energy is resolved to a precision of 10^{-5} after only a few τ . The two largest step sizes ($\beta = 1.5, 1.25$) fail to resolve the plateau to this precision, since the walker distribution collapses beyond $\tau \approx 10$. For $\beta = 1.5$, we mark the n_{eff} values at selected projection depths τ . The simulations with $\beta = 1$ and $\beta = 0.75$ lose convergence beyond $\tau \approx 30$ and $\tau \approx 80$, respectively. The smallest values of β ($\beta = 0.25, 0.5$) resolve the plateau up to the largest projection depth considered, $\tau = 100$.

Panel (c) shows the averaged nearest-neighbor correlator $\sigma_i^z \sigma_{i+1}^z$. The correlator is less precise than the energy, and its initial deviation from the exact value at small τ is much larger. All step sizes follow the same curve until the larger step sizes lose convergence. We mark the effective sample size for $\beta = 0.25$, allowing comparison with the $\beta = 1.5$ results in panel (b).

Panel (d) shows the off-diagonal transverse magnetization per site. These data are obtained for fewer walkers and shorter projection depths because the off-diagonal measurement is substantially more expensive, scaling as $\max(\tau)$. Consequently, convergence is lost earlier than in the diagonal calculations in panels (b) and (c). However, the transverse magnetization can also be computed from the energy, panel (b), and the $\sigma^z \sigma^z$ correlator, panel (c), and is shown in red for the largest step size $\beta = 1.5$. This shows that a larger walker ensemble is capable of resolving the transverse magnetization more reliably.

Figure 5 shows the averaged nearest-neighbor correlator and the total transverse magnetization per site at increasing projection depths τ for different h_x values. Away from the critical point ($h_x < 0.9$ or $h_x > 1.1$), both observables converge rapidly for $\tau \leq 10$. At and near the critical point, comparable convergence requires larger projection depths, $\tau \sim 50$. Our simulations resolve the diagonal correlator reliably across the entire parameter space. The off-diagonal forward walking was carried out only up to $\tau = 10$ and therefore does not converge near the critical point. However, the transverse magnetization obtained from the diagonal run at $\tau = 120$ via $\sum_i \langle \sigma_i^x \rangle = -(E_0 + J_z \sum_i \langle \sigma_i^z \sigma_{i+1}^z \rangle) / h_x$ (red squares in **Figure 5**) converges across the entire parameter range considered.

3.2 Nearest-neighbor XXZ on the pyrochlore lattice

The pyrochlore lattice is the archetypal three-dimensional frustrated lattice. It hosts a $U(1)$ quantum spin liquid phase known as quantum spin ice [42–45]. The lattice consists of corner-sharing tetrahedra whose centers form a diamond lattice. We use spin-1/2 degrees of freedom coupled by an XXZ Hamiltonian:

$$H = \sum_{\langle i,j \rangle} [J_{xy} (S_i^x S_j^x + S_i^y S_j^y) + J_z S_i^z S_j^z]. \quad (27)$$

It is sign-problem-free for $J_{xy} < 0$.

In the Ising limit $J_{xy} = 0$, the model has an extensive classical ground-state degeneracy equivalent to water ice [60–62]. Weak transverse coupling $|J_{xy}| \ll J_z$ lifts this degeneracy through quantum fluctuations described by a ring-exchange Hamiltonian, which arises at third order in perturbation theory [42]:

$$H_{\text{eff}} = -g \sum_{(i,j,k,l,m,n) \in \square} S_i^+ S_j^- S_k^+ S_l^- S_m^+ S_n^- + \text{h.c.} \quad (28)$$

The sum runs over the hexagonal plaquettes of the pyrochlore lattice and $g \sim |J_{xy}|^3 / J_z^2$. The ring model gives rise to an emergent quantum electrodynamics with photons, electric charges (spinons), and magnetic monopoles.

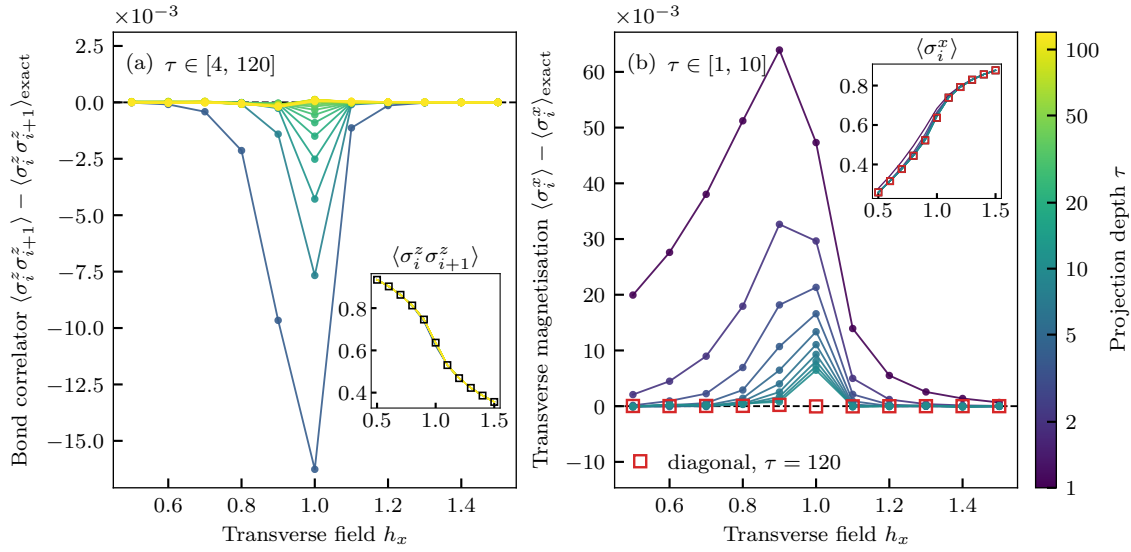


Figure 5: Deviation of the averaged nearest-neighbor correlator $\sum_i \langle \sigma_i^z \sigma_{i+1}^z \rangle / N$ (left) and of the per-site transverse magnetization $\sum_i \langle \sigma_i^x \rangle / N$ (right) from the exact values versus h_x in the transverse-field Ising model from Equation 25 with $N = 100$ on a periodic chain. Different colors encode the projection depth τ . The maximal values for the diagonal and off-diagonal observables are $\tau = 120$ and $\tau = 10$, respectively. The red squares show the transverse magnetization obtained from the diagonal correlator and energy at $\tau = 120$. The same GFMC settings as in Figure 4 were used.

The system undergoes a phase transition into a ferromagnetic ground state as $|J_{xy}|/J_z$ increases, which is understood as the condensation of spinons. The transition occurs at a small value, $|J_{xy}|/J_z \approx 0.1036$ [16], which makes equilibration difficult and Monte Carlo simulations challenging [19, 20]. To overcome this difficulty, we employ a Jastrow-type guiding wavefunction that provides a good approximation of the ground state. We parameterize the guide as $\log \psi_T(x) = \sum_c \Phi(c[x])$, where c runs over all 18-site clusters formed by the closed loop of six tetrahedra surrounding a hexagonal plaquette. We refer to these clusters as *crowns*. Here, $c[x]$ denotes the configuration of the 18 spins on cluster c . A pyrochlore lattice with N sites hosts N crowns, and each Φ_c maps the 2^{18} configurations to real numbers. We set the same $\Phi = \Phi_c$ for each cluster and employ various spatial symmetries of the crown cluster that, together with time-reversal symmetry, drastically reduce the number of independent parameters. Instead of 2^{18} coefficients, we end up with only 2110 independent parameters that must be optimized. Optimization was performed using NetKet [38, 39].

To resolve the transition into the ferromagnet, we measure the transverse structure factor at $\mathbf{q} = 0$:

$$S^\pm(\mathbf{q} = 0) = \frac{1}{N} \sum_{i < j} \langle S_i^+ S_j^- + S_i^- S_j^+ \rangle. \quad (29)$$

$S^\pm(\mathbf{q} = 0)$ is extensive in the ferromagnetic phase and intensive in the spin liquid, such that the order parameter $S^\pm(\mathbf{q} = 0)/N$ remains finite in the ferromagnet and vanishes in the spin liquid as $N \rightarrow \infty$. Figure 6 shows the structure factor as a function of J_{xy} for different system sizes. The inset zooms into the transition region $J_{xy}/J_z \in [-0.11, -0.10]$ and highlights the transition at

$$\frac{J_{xy}}{J_z} = -\frac{31}{300} \pm \frac{1}{1200}, \quad (30)$$

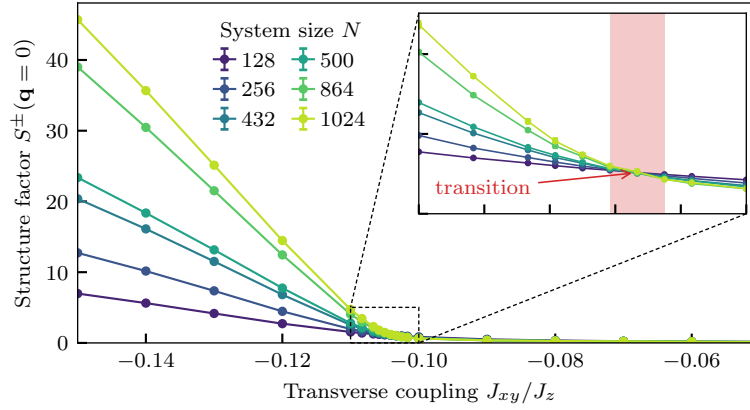


Figure 6: Transverse structure factor at the ordering wavevector, $S^\pm(\mathbf{q} = 0)$ from Equation 29, of the nearest-neighbor pyrochlore XXZ model in Equation 27 as a function of J_{xy}/J_z , for different system sizes. In the ferromagnetic regime $S^\pm(\mathbf{q} = 0)$ grows extensively with N , whereas it remains intensive in the spin-liquid phase. The inset zooms into the transition region $J_{xy}/J_z \in [-0.11, -0.10]$. The shaded band marks the transition at $J_{xy}/J_z = -\frac{31}{300} \pm \frac{1}{1200}$, where the curves for different N cross. GFMC parameters: 2^{14} walkers (2^{15} for $N = 1024$ near the transition), 2000 measurements with an off-diagonal stride of $\tau \approx 3$ to $\tau \approx 20$, and 2 independent samples per point. We used projection depths from $\tau = 14$ to $\tau = 60$ depending on N and J_{xy} . See Appendix C for further information.

which is in agreement with Ref. [16]. Appendix C discusses the convergence with the projection depth τ and compares the results to the order parameter evaluated for the guiding wavefunction alone (without GFMC). While the guide captures the overall trend, it fails to resolve the transition reliably. This illustrates how the GFMC projection and a good variational state complement each other.

3.3 Ring-exchange on the pyrochlore lattice

We demonstrate how PyGFMC can access non-trivial excitations by computing the flux cost of the effective ring model of Equation 28. The ring Hamiltonian conserves the total electric flux threading each of the periodic boundaries of the cluster. Preparing the initial walker population in a chosen flux sector therefore lets us probe the energy cost of flux excitations. The energy cost of a flux configuration $\phi = (\phi_x, \phi_y, \phi_z)$ in a cluster of linear size L scales as $|\phi|^2/L$. This is the characteristic signature of quantum spin ice [44, 63, 64]. Figure 7 shows the quadratic cost as a function of ϕ_x and the reciprocal system-size dependence. We have set the flux through the other surfaces to zero, $\phi_y = \phi_z = 0$.

4 Package structure and API

The package is organized around five main components, which we describe in turn:

- Section 4.1: Hamiltonian;
- Section 4.2: Initial walker ensemble;
- Section 4.3: Guiding wavefunction (optional);

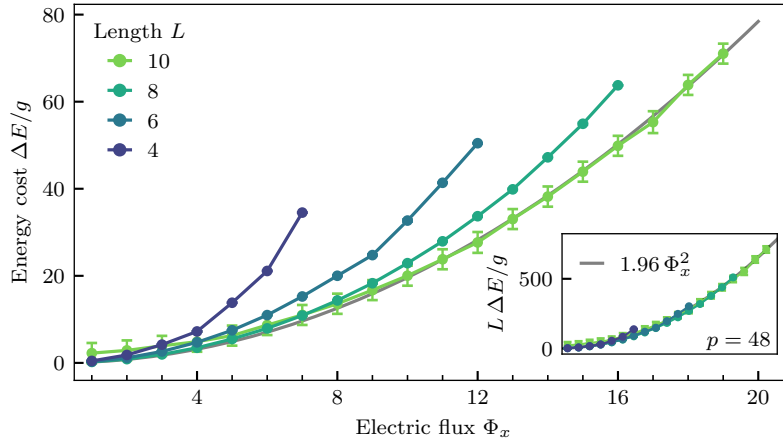


Figure 7: Energy cost of a flux sector $(\phi_x, 0, 0)$ in the pyrochlore ring-exchange model for several linear system sizes L with $N = 4L^3$ spins in total. The inset shows the $1/L$ scaling collapse. GPMC parameters: step size $n_{\text{step}} = 15$, $\lambda = 0$, 2^{12} walkers and 10^5 steps per flux sector, number of branching steps $p = 48$; no guiding wavefunction has been used.

- **Section 4.4:** Observables (optional);
- **Section 4.5:** Simulation driver.

PyGFMC can be installed with `pip install pygfmc` (`pip install "pygfmc[netket]"` if you want to use NetKet [38, 39]), or from the source code at <https://github.com/robin-schaefer/PyGFMC>.

4.1 Hamiltonian

The Hamiltonian determines the walker propagation. We represent any operator as a sum

$$H = \sum_{i=0}^{M_{\text{total}}-1} h_i, \quad (31)$$

where each term h_i produces *at most one* new state when acting on a basis state $|x\rangle$. Operators like S^x therefore must be split into separate operators, such as S^+ and S^- for spin $S > 1/2$.

As a minimal example, consider a nearest-neighbor Hamiltonian for spin-1/2 on a one-dimensional chain with periodic boundary conditions, non-uniform coupling amplitudes $c_j^{z/\pm}$, and local z -fields m_j :

$$H = \underbrace{\sum_{j=0}^{L-1} m_j S_j^z + \sum_{j=0}^{L-1} c_j^z S_j^z S_{(j+1) \bmod L}^z}_{\equiv D} - \sum_{j=0}^{L-1} c_j^{\pm} (S_j^+ S_{(j+1) \bmod L}^- + \text{h.c.}). \quad (32)$$

We absorb *all* diagonal operators into one single operator D .

A Hamiltonian class must include the following attributes and methods:

- `M_total`: total number of terms in the sum of Equation 31. For the example in Equation 32, `M_total` = $L + 1$ (one diagonal + L hopping bonds).

- `get_rates(config, return_configs=False)`: given a configuration `config`, returns the pair `(rates, new_configs)`. `rates` is a one-dimensional array of length `M_total` containing the weight produced by each h_i . When `return_configs=True`, `new_configs` contains the configurations in an array of shape `(M_total, N)` obtained by applying each h_i to `config`. It is `None` otherwise.

Taking the example Hamiltonian in Equation 32 for $L = 4$ and `config` = `|1100>`, the weight array is

$$\text{rates} = (d \quad 0 \quad -c_1^\pm \quad 0 \quad -c_3^\pm), \quad (33)$$

with $d = \frac{1}{2}(m_0 + m_1 - m_2 - m_3) + \frac{1}{4}(c_0^z - c_1^z + c_2^z - c_3^z)$ as the diagonal element.

The first entry is *always* reserved for the diagonal operator. It is zero if no diagonal terms are present. The two interior zeros come from hopping operators acting on aligned neighbors, as those bonds cannot flip and, therefore, do not generate a transition. The corresponding new configurations form a two-dimensional array, where row i corresponds to applying h_i to `config`,

$$\text{new_configs} = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix}. \quad (34)$$

Rows for diagonal or non-flippable operators are not considered. They simply repeat the input `config` in this case.

- `generate_state(config, idx)`: returns the configuration corresponding to row `idx` of Equation 34. The propagator calls it whenever `get_rates` was not asked to materialize the successor configurations. We recommend *not* computing all `M_total` possible configurations, since only the one stochastically chosen is required. This saves memory and runtime.
- `diagonal_weight(config)` (optional): returns the diagonal contribution d , which is used to compute the diagonal expectation values. It can also return multiple values via an array, which is beneficial if a set of diagonal operators shall be evaluated at once (see Section 4.4).

Implemented Hamiltonian classes. The following classes are implemented:

- `Spin.create(edges, S, J_z, J_pm, J_pmpm, h_z, h_pm, N)`: spin- S Hamiltonian with one- and two-body operators:

$$H = \sum_{\langle i,j \rangle} [J_z S_i^z S_j^z + J_\pm (S_i^+ S_j^- + \text{h.c.}) + J_{\pm\pm} (S_i^+ S_j^+ + \text{h.c.})] - \sum_i [h_z S_i^z + h_\pm (S_i^+ + S_i^-)]. \quad (35)$$

- `SpinHalf.create(edges, J_x, J_y, J_z, h_z, h_x, N)`: optimized XYZ Hamiltonian for $S = 1/2$ (used in Section 3.1 and Section 3.2):

$$H = \sum_{\langle i,j \rangle} [J_x S_i^x S_j^x + J_y S_i^y S_j^y + J_z S_i^z S_j^z] - \sum_i [h_z S_i^z + h_x S_i^x]. \quad (36)$$

- `ToricCode.create(Lx, Ly, A_s, B_p, h_x, h_z)`: toric code Hamiltonian with local x and z fields:

$$H = -A_s \sum_v \prod_{i \in \star(v)} \sigma_i^x - B_p \sum_p \prod_{i \in \square(p)} \sigma_i^z - h_z \sum_i \sigma_i^z - h_x \sum_i \sigma_i^x. \quad (37)$$

- `BoseHubbard.create(edges, t, eta, U, mu, h_pm, N, n_max)`: generic Bose–Hubbard Hamiltonian:

$$H = -t \sum_{\langle i,j \rangle} (b_i^\dagger b_j + \text{h.c.}) - \eta \sum_{\langle i,j \rangle} (b_i^\dagger b_j^\dagger + \text{h.c.}) + \sum_i \left[\frac{U}{2} n_i (n_i - 1) - \mu n_i - h_\pm (b_i^\dagger + b_i) \right]. \quad (38)$$

- `PlaquetteModel.create(plaquettes, g, mu)`: implements ring-exchange flips of amplitude $-g$ on a collection of even-length plaquettes, with a Rokhsar–Kivelson potential [41] (used in Section 3.3). The `plaquettes` are simply given by a two-dimensional array containing the sites of each plaquette. We define $|\uparrow_p\rangle$ and $|\downarrow_p\rangle$ for the two *flippable* configurations on a plaquette p where spins alternate $+, -, +, -, \dots$ clockwise or counterclockwise:

$$H = -g \sum_p (|\uparrow_p\rangle\langle\downarrow_p| + |\downarrow_p\rangle\langle\uparrow_p|) + \mu \sum_p (|\uparrow_p\rangle\langle\uparrow_p| + |\downarrow_p\rangle\langle\downarrow_p|). \quad (39)$$

The off-diagonal term flips a plaquette between its two alternating configurations and the potential counts the number of flippable plaquettes.

A simple code example to construct the Hamiltonian from Equation 32 is:

```
import numpy as np
from pygfm.hamiltonian import SpinHalf

L      = 8
edges = [(i, (i + 1) % L) for i in range(L)]          # 1D PBC chain

J_xy   = -np.random.uniform(0.8, 1.2, size=len(edges)) # random per-bond
m       = np.linspace(-0.3, 0.3, L)                   # local z-fields m_j

H = SpinHalf.create(edges,
                    J_x=J_xy, J_y=J_xy,                # random per-edge XY
                    J_z=1.0,                           # uniform Ising coupling
                    h_z=-m)                             # SpinHalf has -h_z S^z, so h_z = -m
```

Exact diagonalization. PYGFMC provides an option to compute the ground state (and excited states) using exact diagonalization:

```
energies, diag_obs, off_diag_obs = exact_diagonalization(
    hamiltonian,          # any class following the rate protocol
    local_hilbert_space=None, # per-site values, defaults to
    hamiltonian.local_hilbert_space
    N=None,               # number of sites, defaults to hamiltonian.N
    diag_operators=None,  # measured as sum_x |psi_x|^2 O(x)
    off_diag_operators=None, # measured as psi^T O psi
    number_of_states=1,    # how many lowest states to solve for
)
```

It relies on the `get_rates` function of the Hamiltonian. The local Hilbert space for N sites is given by `local_hilbert_space`, which must match the convention of the Hamiltonian, e.g., `(-1,1)` for spin-1/2. Besides the ground-state energy, it returns observables for diagonal and off-diagonal operators in a dictionary (see [Section 4.4](#)).

More details, including a custom Hamiltonian class, can be found in [Appendix D](#) and [Appendix E](#).

4.2 Walker ensemble

The walker ensemble is stored as a two-dimensional array of shape $n_{\text{walker}} \times N$, where N is the system size. Entry $[i, j]$ holds the basis-state value, such as a σ^z eigenvalue or an occupation number, of the i -th walker at site j . Any encoding can be used, as long as it is consistent with the Hamiltonian and the (optional) guiding wavefunction.

Note that the initial walker ensemble must be chosen such that every walker lies within the symmetry sector of interest, for example, a sector defined by fixed total particle number or magnetization. We use this in [Section 3.3](#) to target topological electric-flux sectors. If the initial walker ensemble is spread across different sectors, it samples the different ground states within these symmetry sectors, and the simulation is corrupted.

PYGPMC provides several routines for initializing the walker ensemble:

- `prepare_walkers_random(N_walkers, N_sites, rng_key, states=(-1,1))`: each site is drawn independently and uniformly from `states`, which encodes, for example, spin-1/2 degrees of freedom by setting it to `(-1,1)` or `(-0.5,0.5)`, higher-spin values to `(-1,0,1)`, or bosonic occupation numbers to `(0,...,n_max)`.
- `prepare_walkers_sz(N_walkers, N_sites, rng_key, sz=0, states=(-1,1))`: random configurations at a fixed total charge. `states` defines the local Hilbert space and `sz` simply refers to the sum over all sites $\sum_i \sigma_i^z$.
- `prepare_walkers_from_configs(configs)`: the walkers are initialized with the given configurations `configs` of shape $(N_{\text{walkers}}, N_{\text{sites}})$.

The configuration's data type is determined by `states` or `configs`.

We note that randomly generated initial states are usually poor representations of the ground-state ensemble, so a warm-up should be executed before the main pipeline.

Continuing the example code for the XXZ chain of [Equation 32](#), which conserves the total magnetization, we initialize the walkers in the $S_{\text{tot}}^z = 0$ sector.

```
import time, jax
from pygfm.walkers import prepare_walkers_sz

rng      = jax.random.PRNGKey(int(time.time())) # time-seeded
rng, key = jax.random.split(rng)
walkers  = prepare_walkers_sz(N_walkers=512, N_sites=H.N, rng_key=key, sz=0)
```

4.3 Guiding wavefunction (optional)

The algorithm requires only the ratio $\psi_T(x')/\psi_T(x)$, so the guide class returns the log-difference $\log \psi_T(x') - \log \psi_T(x)$ for all `M_total` successor states of the current configuration. At least one of the following three methods must be implemented:

- `log_diff_from_operator(config, hamiltonian, cache=None)`: evaluates the log-differences directly from `config` and the `hamiltonian`. This function does not require the materialized `M_total` configurations and computes the logarithmic difference on the fly. This saves memory and runtime.
- `log_diff_from_configs(config, new_configs, cache=None)`: takes the pre-materialized `new_configs` array from `get_rates(config, return_configs=True)` (see Equation 34). This function is used as a fallback when the previous method is not available.
- `log_psi(config, cache=None)`: evaluates the log-amplitude $\log \psi_T$ directly for a given `config`. It is used as a last fallback when neither of the other two methods is available.

The guide may optionally provide a function `build_cache(hamiltonian)`, which returns a `cache` object that is passed to the methods above. For example, it can store system-specific information, which often helps to speed up the calculation.

PyGFMC has different pre-implemented guides:

- `JastrowSpinHalf(N, pairs, beta, h=None)` for `SpinHalf`: the pairwise ansatz used in Equation 26

$$\psi_T(x) = \exp\left(-\sum_{\langle i,j \rangle} \beta_{ij} \sigma_i^z \sigma_j^z + \sum_i h_i \sigma_i^z\right), \quad (40)$$

where β_{ij} (h_i) can vary across spin pairs (sites) or be a single global parameter.

- `JastrowSpin(N, pairs, beta, h=None, d=None)` for `Spin`: similar to the ansatz in Equation 40:

$$\psi_T(x) = \exp\left(-\sum_{\langle i,j \rangle} \beta_{ij} x_i x_j + \sum_i h_i x_i - \sum_i d_i x_i^2\right), \quad (41)$$

where $x_i \in \{-S, \dots, S\}$ is the eigenvalue of S_i^z and d_i is a single-ion term.

- `JastrowBoseHubbard(N, pairs, beta, h=None, d=None)` for `BoseHubbard`: similar to the ansatz in Equation 41:

$$\psi_T(x) = \exp\left(-\sum_{\langle i,j \rangle} \beta_{ij} x_i x_j + \sum_i h_i x_i - \sum_i d_i x_i^2\right), \quad (42)$$

where x_i is the occupation of site i and d_i is the Gutzwiller factor.

- `ClusterCPS(groups, amplitudes=None, init='zero', seed=None)` for `SpinHalf`: a *correlator product state* [57, 58], which assigns one variational parameter to every configuration of a local cluster, $\Phi_c(x_c) \mapsto \mathbb{R}$, using a lookup table. Here x_c is the local configuration on a cluster c . Symmetries can be embedded, making local clusters of up to 18 sites possible. This ansatz has been used in Section 3.2.
- `ClusterMLP(groups, params=None, seed=None)` for `SpinHalf`: implements the cluster functions Φ_c [57, 58] as multilayer perceptrons (MLPs) with a configurable number of hidden layers.

PyGFMC further provides a generic function to optimize a guide with NetKet [38, 39]:

```

guide = train_with_netket(
    hamiltonian,          # only get_rates is required
    guide,                # only log_psi is required
    local_hilbert_space=None, # defaults to hamiltonian.local_hilbert_space
    N=None,               # defaults to hamiltonian.N
    ...,                  # sampler and further VMC settings
)

```

A new guide of the same class is returned and the original is left untouched.

Continuing the running example, a one-parameter Jastrow ansatz $\beta_{ij} = \beta$ defined on the Hamiltonian edges can be created via:

```

from pygfmc.wavefunction import JastrowSpinHalf

guide = JastrowSpinHalf(H.N, H.edges, beta=0.1)

```

Further implementation details are given in the appendix: implemented guides ([Appendix F](#)), user-defined guides ([Appendix G](#)), and the NetKet [\[38, 39\]](#) integration ([Appendix H](#)).

4.4 Observables (optional)

An observable can be created with the same interface as the Hamiltonian. While an off-diagonal observable needs `M_total` and `get_rates(config)`, as a Hamiltonian does, a diagonal observable needs only `diagonal_weight(config)`, which is cheaper to evaluate. As discussed in [Section 2.5](#), the matrix elements of the off-diagonal operator must be nonnegative. Operators with mixed signs must be split into positive and negative parts and measured separately (see [Equation 16](#)). The observables are passed to the main pipeline as dictionaries keyed by name, via the `diag_operators` and `off_diag_operators` arguments (see [Section 4.5](#)), and the returned estimates are mirrored back in the same key structure.

Launching an auxiliary Markov chain to measure an off-diagonal observable is expensive and scales with $m_{\max} = \max(p)/2$. It is therefore worth determining the autocorrelation time first and launching the chain only once the walker ensemble has decorrelated. Within the production run, this is controlled by the `off_diag_stride` keyword, which measures the off-diagonal observables only every `off_diag_stride`-th branching step. The energy and the diagonal observables are unaffected and remain resolved at every step.

PYGPMC allows `diagonal_weight(config)` to return multiple values, which can be beneficial in some cases. This is the case for the diagonal $S^z S^z$ structure factor that can be computed for several momenta simultaneously via `SzSzStructureFactor(positions, k_vectors)` for the `SpinHalf` class. The function takes the positions of the sites, given as an array of shape (N, dim) , and a selection of k vectors of the same dimension:

$$S^{zz}(\mathbf{k}) = \sum_{i < j} 2 \cos(\mathbf{k} \cdot (\mathbf{r}_i - \mathbf{r}_j)) S_i^z S_j^z. \quad (43)$$

The $\mathbf{k}$ vectors must be compatible with the periodic boundary conditions, $\mathbf{k} \cdot \mathbf{a}_i \in 2\pi\mathbb{Z}$ for the lattice vectors $\mathbf{a}_i$ spanning the system.

`StructureFactor(positions, k, channel='SpSm')` returns the corresponding off-diagonal

operator for the transverse structure factor of the `SpinHalf` class,

$$O = \sum_{i < j} [1 + \cos(\mathbf{k} \cdot (\mathbf{r}_i - \mathbf{r}_j))] (A_i B_j + \text{h.c.}) = AB(0) + AB(\mathbf{k}), \quad (44)$$

$$\text{with } AB(\mathbf{k}) = \sum_{i < j} \cos(\mathbf{k} \cdot (\mathbf{r}_i - \mathbf{r}_j)) (A_i B_j + \text{h.c.}). \quad (45)$$

There are different options for the channel: `'SpSm'`: $A = S^+$ and $B = S^-$; `'SpSp'`: $A = S^+$ and $B = S^+$; `'SpSm+SpSp'`: the sum of the two, $S_i^+ S_j^- + S_i^+ S_j^+ + \text{h.c.} = 4S_i^x S_j^x$. The channel `'SzSz'` ($A = B = S^z$) yields the diagonal operator $S^{\text{zz}}(0) + S^{\text{zz}}(\mathbf{k})$ of Equation 43. Notably, we compute the structure factor at a given k vector plus the zero vector. This avoids the problem that the cosine becomes negative.

We demonstrate how to compute the total and on-site magnetization operators $S_{\text{tot}}^z = \sum_i S_i^z$ and S_0^z , their transverse counterparts S_{tot}^x and S_0^x , and the diagonal and transverse structure factors. We set `sign_check=False`, which disables the sign-problem-free check that is enforced for Hamiltonians but not needed for observables.

```
import jax.numpy as jnp
from pygfmc.hamiltonian import SpinHalf, SzSzStructureFactor, StructureFactor

h_all = jnp.ones(L)
h_0 = jnp.zeros(L).at[0].set(1.0)

Sztot = SpinHalf.create(edges=[], N=L, h_z=-h_all)
Sz0 = SpinHalf.create(edges=[], N=L, h_z=-h_0)
Sxtot = SpinHalf.create(edges=[], N=L, h_x=-h_all, sign_check=False)
Sx0 = SpinHalf.create(edges=[], N=L, h_x=-h_0, sign_check=False)

# structure factors: all allowed k of the L-site ring at once
positions = jnp.arange(L)[: , None] # (N, dim)
k_vectors = 2 * jnp.pi * jnp.arange(L)[: , None] / L # (n_k, dim)

Szz = SzSzStructureFactor(positions, k_vectors)
Spm = StructureFactor(positions, k_vectors[L // 2], channel='SpSm')

diag_operators = {'Sztot': Sztot, 'Sz0': Sz0, 'Szz': Szz}
off_diag_operators = {'Sxtot': Sxtot, 'Sx0': Sx0, 'Spm': Spm}
```

Appendix E demonstrates how to implement custom operators that can be fed into the PYGPMC pipeline, such as multi-body terms or a simplified version of the structure factor.

4.5 Simulation driver

The four components above (Hamiltonian, walkers, guide, and observables) are tied together using the `run` function, which is the core of PYGPMC. It executes an optional warm-up, the production scan of n_{steps} branching steps, and the post-run statistics. When several GPUs are visible, `run` automatically distributes the walker population across all of them. Measuring diagonal observables is restricted to a single device.

Key input keywords. A complete list of the input arguments of `run` is provided in Appendix A. The most important keywords are:

- `walkers`: the initial walker ensemble (Section 4.2);
- `n_steps`: number of production branching steps;

- `hamiltonian`: the Hamiltonian;
- `step_size`: an integer number of steps for the discrete methods ("`single_step`", "`multi_step`"), or a positive real imaginary-time length β for "`cont_time`" (see [Section 2.3](#));
- `guide`: guiding wavefunction (use `None` if no guide should be used);
- `p_array`: list containing the number of branching steps p defining the projection depth (see [Section 2.6.2](#));
- `method`: propagation scheme (see [Section 2.3](#)):
 - "`single_step`" for step-by-step propagation,
 - "`multi_step`" for batched discrete substeps, or
 - "`cont_time`" for the continuous-time scheme.
 (Default: "`cont_time`");
- `rng_key`: JAX PRNG key (Default: `jax.random.PRNGKey(secrets.randbits(32))`);
- `lam`: diagonal shift λ (only required for discrete propagation methods);
- `diag_operators`: diagonal observables as a keyed dictionary as described in [Section 4.4](#) (Default: `None`);
- `off_diag_operators`: off-diagonal observables as a keyed dictionary as described in [Section 4.4](#) (Default: `None`).

Return values. The function returns the tuple `(walkers, energy, obs, stats, raw)` (`raw` is optional), where

- `walkers`: the updated walker batch;
- `energy`: a `dict` of length `len(p_array)`;
- `obs`: a nested `dict` accessed as `obs[name][p]`, where the outer key is the name of the operator and the inner key is the number of projection steps p ;
- `stats`: a dictionary with the attributes of the `statistics` function described in [Section 2.6](#), together with '`avg_log_weight`', '`it_per_sec`', '`runtime`' and '`total_wall_time`', the autocorrelation times under the '`autocorrelation/`' prefix, and the run's own inputs under '`parameters/`';
- `raw`: a dictionary with the raw, step-resolved GFMC values: `log_weights` and `energy` are 1D arrays of length n_{steps} . Observables are accessed via nested keys with `['diag']['<name>']` and `['off_diag']['<name>']` that contain arrays of shape $n_{\text{steps}} \times n_p$, where n_p is the length of `p_array`. Diagonal operators may hold an array of shape $n_{\text{steps}} \times n_k \times n_p$, where n_k is the number of return values of `diagonal_weight`. n_{steps} is reduced if `off_diag_stride` is greater than 1 for the off-diagonal observable. They contain the raw estimators of the observables. The dictionary also includes `off_diag_norm`, which has the same dimensions and contains the logarithmic weight obtained by the auxiliary chain evolved over $m = p/2$ steps.

Storing and resuming a simulation. Objects returned from the production run can simply be stored using the function:

```
write_to_HDF5(
    filename,          # created if absent, appended to otherwise
    stats,             # statistics and input parameters, required
    walkers=None,      # final walker ensemble
    energy=None,        # energy per p
    obs=None,          # observables per p
    raw=None,          # per-step energy, log weights and observables
    walker_filename=None, # separate file for the walker ensemble
)
```

It is designed both to create a file initially and to append data from consecutive simulation runs. Data are categorized into four groups:

- (i) The initial creation stores all fixed input parameters, including data types and hashes for the Hamiltonian, guide, and observables. These are stored in the group `'/parameters'`. When a simulation is resumed (see below), for example, when more steps are required for better statistics, all relevant parameters are checked to ensure consistency between simulations.
- (ii) Each simulation also stores data in a designated group, `'/run_<n>'` for the n -th simulation. This group contains the following (when provided): the energy per p (`energy`), the observables per p (`obs`), and statistics (`stats`). The statistics include the branching diagnostics of [Section 2.6](#), the autocorrelation times, and other information such as the iterations per second or the runtime. We further store a hash of the final walker ensemble and the seed to ensure a continuous, reproducible simulation.
- (iii) The final walker ensemble (needed to relaunch the simulation) is stored in `'/current_walkers'` if provided. Storing the walker ensemble might require a large amount of disk space. Hence, we provide the option to store it instead in `walker_filename` when it is provided. It is overwritten with each new call.
- (iv) The raw data (per-step energy estimator, logarithmic weights, or observables) are appended in the group `'/raw'`. These arrays continuously grow in size whenever another production run is launched.

The simulation can be relaunched from the HDF5 file using

```
walkers, energy, obs, stats, raw = run_from_HDF5(
    filename,          # file written by write_to_HDF5
    hamiltonian,       # checked against the stored hash
    guide=None,        # checked against the stored hash
    diag_operators=None, # checked against the stored hashes
    off_diag_operators=None, # checked against the stored hashes
    n_steps=None,      # None reuses the last number of steps
    walker_filename=None, # read the ensemble from a separate file
)
```

This is essentially a wrapper for the `run` function and returns the same objects. Before it launches the calculation, it verifies that the input classes (Hamiltonian, observables, and guide) have matching hashes. Since the simulation is relaunched from an equilibrated ensemble, the warm-up is skipped. When `n_steps` is not given, it reuses the number of steps used in the previous calculation.

The file grows continuously and holds the entire main chain. Continuing with the stored RNG key reproduces the trajectory of one contiguous run. Therefore, it can be used to resume

a long simulation after a timeout or crash when intermediate steps are stored.

PyGFMC can compute observables and energy from the `raw` dictionary, which is either returned from the `run` function or loaded from disk via `raw = load_raw_from_HDF5(filename)`:

```
energy, obs = evaluate_observables(
    raw,                # raw data
    p_array=None,       # None reuses the stored branching steps
    n_blocks=None,      # number of jackknife blocks; None returns no errors
)
```

The energy can be assessed for arbitrary branching steps in post-processing by passing an array of p values to the function. This is not the case for observables, as they have already been computed for a given number of steps p .

The function can also return estimates of the error using a blocked jackknife as discussed in [Section 2.6.1](#). If `n_blocks` is given, the function returns `(energy, obs, energy_err, obs_err)` instead. However, we recommend running independent simulations and comparing their means rather than relying solely on the jackknife analysis. Note that both approaches estimate only the statistical error and do not capture the systematic bias from a finite projection depth.

We complete our example with:

```
from pygfmc import run, write_to_HDF5, run_from_HDF5

walkers, energy, obs, stats, raw = run(
    walkers, n_steps=5000, hamiltonian=H, step_size=0.5,
    guide=guide, p_array=[4, 8, 16, 32], warmup_steps=500,
    method='cont_time', rng_key=rng,
    diag_operators=diag_operators,
    off_diag_operators=off_diag_operators,
)
write_to_HDF5('run.h5', stats, walkers, energy, obs, raw)

# continue the same chain for another 5000 steps
walkers, energy, obs, stats, raw = run_from_HDF5(
    'run.h5', H, guide, diag_operators, off_diag_operators,
)
write_to_HDF5('run.h5', stats, walkers, energy, obs, raw)
```

Parts of the code snippets above are collected in `notebooks/xxz_chain.ipynb`.

5 Discussion

PyGFMC provides a GPU-accelerated implementation of Green's function Monte Carlo for sign-problem-free quantum lattice models. By exploiting the substantial computational capabilities of modern GPUs, it can study problems that were previously hardly accessible. The resulting speedup of nearly three orders of magnitude enables more expressive guiding wavefunctions and walker populations of up to 10^7 (see [Figure 3](#)). We have demonstrated that PyGFMC reliably resolves the ground state of frustrated magnets for which other quantum Monte Carlo methods, such as the stochastic series expansion, struggle to reach low temperatures [[19,20](#)]. This makes it a competitive alternative for ground-state properties.

The library provides built-in Hamiltonians and guiding wavefunctions, diagonal and off-diagonal estimators, HDF5-based storage and continuation, and support for custom classes without modifying the core engine. Furthermore, it integrates with NetKet [[38,39](#)], providing access to a broad range of variational wavefunctions and robust training procedures. The

repository includes various notebooks with examples, and detailed documentation is available at <https://github.com/robin-schaefer/PyGFMC>.

The code was developed from our earlier C++ implementation with the assistance of coding tools based on large language models and has been validated through extensive test suites (see [Appendix B](#)). The manuscript, documentation, and code comments are designed to make the library accessible to both human users and coding agents. We encourage agent-assisted development on top of PyGFMC.

We see two directions for future work. First, while GFMC with its fixed walker ensemble is especially well suited to GPUs, other variants of quantum Monte Carlo (see Refs. [[27](#), [65](#)]) would likely benefit from the same computational power. Thus, it is worth exploring GPU implementations of other quantum Monte Carlo methods. Second, we find that a carefully chosen ansatz wavefunction is the key to a well-converged simulation. Implementing and benchmarking further variational wavefunctions, such as physically motivated neural quantum states [[66](#)], therefore offers a promising route to increasingly challenging problems such as spin glasses.

Acknowledgments

Many thanks to David Luitz, Nils Niggemann, and Johannes Reuther for helpful discussions and feedback.

Funding information This work was funded by the DFG under Project No. 575641691.

References

- [1] F. Becca and S. Sorella, *Quantum Monte Carlo Approaches for Correlated Systems*, Cambridge University Press, Cambridge, doi:[10.1017/9781316417041](https://doi.org/10.1017/9781316417041) (2017).
- [2] A. W. Sandvik and J. Kurkijärvi, *Quantum Monte Carlo simulation method for spin systems*, Phys. Rev. B **43**, 5950 (1991), doi:[10.1103/PhysRevB.43.5950](https://doi.org/10.1103/PhysRevB.43.5950).
- [3] P. Weinberg and M. Bukov, *QuSpin: a Python package for dynamics and exact diagonalisation of quantum many body systems part I: spin chains*, SciPost Phys. **2**, 003 (2017), doi:[10.21468/SciPostPhys.2.1.003](https://doi.org/10.21468/SciPostPhys.2.1.003).
- [4] P. Weinberg and M. Bukov, *QuSpin: a Python package for dynamics and exact diagonalisation of quantum many body systems. Part II: bosons, fermions and higher spins*, SciPost Phys. **7**, 020 (2019), doi:[10.21468/SciPostPhys.7.2.020](https://doi.org/10.21468/SciPostPhys.7.2.020).
- [5] R. Schäfer and D. J. Luitz, *DanceQ: high-performance library for number conserving bases*, SciPost Phys. Codebases p. 48 (2025), doi:[10.21468/SciPostPhysCodeb.48](https://doi.org/10.21468/SciPostPhysCodeb.48).
- [6] R. Schäfer and D. J. Luitz, *Codebase release 1.0 for DanceQ*, SciPost Phys. Codebases p. 48-r1.0 (2025), doi:[10.21468/SciPostPhysCodeb.48-r1.0](https://doi.org/10.21468/SciPostPhysCodeb.48-r1.0).
- [7] A. Wietek, L. Staszewski, M. Ulaga, P. L. Ebert, H. Karlsson, S. Sarkar, L. Shackleton, A. Sinha and R. D. Soares, *XDiag: exact diagonalization for quantum many-body systems*, SciPost Phys. Codebases p. 70 (2026), doi:[10.21468/SciPostPhysCodeb.70](https://doi.org/10.21468/SciPostPhysCodeb.70).
- [8] S. R. White, *Density matrix formulation for quantum renormalization groups*, Phys. Rev. Lett. **69**, 2863 (1992), doi:[10.1103/PhysRevLett.69.2863](https://doi.org/10.1103/PhysRevLett.69.2863).

- [9] U. Schollwöck, *The density-matrix renormalization group in the age of matrix product states*, Ann. Phys. **326**, 96 (2011), doi:[10.1016/j.aop.2010.09.012](https://doi.org/10.1016/j.aop.2010.09.012).
- [10] J. Hauschild, J. Unfried, S. Anand, B. Andrews, M. Bintz, U. Borla, S. Divic, M. Drescher, J. Geiger, M. Hefel, K. Hémerly, W. Kadow *et al.*, *Tensor network Python (TeNPy) version 1*, SciPost Phys. Codebases p. 41 (2024), doi:[10.21468/SciPostPhysCodeb.41](https://doi.org/10.21468/SciPostPhysCodeb.41).
- [11] G. Torlai and R. G. Melko, *Learning thermodynamics with Boltzmann machines*, Phys. Rev. B **94**, 165134 (2016), doi:[10.1103/PhysRevB.94.165134](https://doi.org/10.1103/PhysRevB.94.165134).
- [12] G. Carleo and M. Troyer, *Solving the quantum many-body problem with artificial neural networks*, Science **355**, 602 (2017), doi:[10.1126/science.aag2302](https://doi.org/10.1126/science.aag2302).
- [13] Y. Nomura, A. S. Darmawan, Y. Yamaji and M. Imada, *Restricted Boltzmann machine learning for solving strongly correlated quantum systems*, Phys. Rev. B **96**, 205152 (2017), doi:[10.1103/PhysRevB.96.205152](https://doi.org/10.1103/PhysRevB.96.205152).
- [14] S. V. Isakov, Y. B. Kim and A. Paramekanti, *Spin-liquid phase in a spin-1/2 quantum magnet on the kagome lattice*, Phys. Rev. Lett. **97**, 207204 (2006), doi:[10.1103/PhysRevLett.97.207204](https://doi.org/10.1103/PhysRevLett.97.207204).
- [15] S. V. Isakov, S. Wessel, R. G. Melko, K. Sengupta and Y. B. Kim, *Hard-core bosons on the kagome lattice: valence-bond solids and their quantum melting*, Phys. Rev. Lett. **97**, 147202 (2006), doi:[10.1103/PhysRevLett.97.147202](https://doi.org/10.1103/PhysRevLett.97.147202).
- [16] A. Banerjee, S. V. Isakov, K. Damle and Y. B. Kim, *Unusual liquid state of hard-core bosons on the pyrochlore lattice*, Phys. Rev. Lett. **100**, 047208 (2008), doi:[10.1103/PhysRevLett.100.047208](https://doi.org/10.1103/PhysRevLett.100.047208).
- [17] S. V. Isakov, M. B. Hastings and R. G. Melko, *Topological entanglement entropy of a Bose–Hubbard spin liquid*, Nature Phys. **7**, 772 (2011), doi:[10.1038/nphys2036](https://doi.org/10.1038/nphys2036).
- [18] S. V. Isakov, R. G. Melko and M. B. Hastings, *Universal signatures of fractionalized quantum critical points*, Science **335**, 193 (2012), doi:[10.1126/science.1212207](https://doi.org/10.1126/science.1212207).
- [19] Y. Kato and S. Onoda, *Numerical evidence of quantum melting of spin ice: quantum-to-classical crossover*, Phys. Rev. Lett. **115**, 077202 (2015), doi:[10.1103/PhysRevLett.115.077202](https://doi.org/10.1103/PhysRevLett.115.077202).
- [20] C.-J. Huang, C. Liu, Z. Meng, Y. Yu, Y. Deng and G. Chen, *Extended Coulomb liquid of paired hardcore boson model on a pyrochlore lattice*, Phys. Rev. Res. **2**, 042022 (2020), doi:[10.1103/PhysRevResearch.2.042022](https://doi.org/10.1103/PhysRevResearch.2.042022).
- [21] N. Niggemann, M. Adhikary, Y. Schaden-Thillmann and J. Reuther, *Gapless fracton quantum spin liquid and emergent photons in a 2D spin-1 model*, Nature Commun. **17**, 6131 (2026), doi:[10.1038/s41467-026-74797-0](https://doi.org/10.1038/s41467-026-74797-0).
- [22] H. Rieger and A. P. Young, *Zero-temperature quantum phase transition of a two-dimensional Ising spin glass*, Phys. Rev. Lett. **72**, 4141 (1994), doi:[10.1103/PhysRevLett.72.4141](https://doi.org/10.1103/PhysRevLett.72.4141).
- [23] M. Guo, R. N. Bhatt and D. A. Huse, *Quantum critical behavior of a three-dimensional Ising spin glass in a transverse magnetic field*, Phys. Rev. Lett. **72**, 4137 (1994), doi:[10.1103/PhysRevLett.72.4137](https://doi.org/10.1103/PhysRevLett.72.4137).

- [24] H. Rieger and A. P. Young, *Griffiths singularities in the disordered phase of a quantum Ising spin glass*, Phys. Rev. B **54**, 3328 (1996), doi:[10.1103/PhysRevB.54.3328](https://doi.org/10.1103/PhysRevB.54.3328).
- [25] M. Guo, R. N. Bhatt and D. A. Huse, *Quantum Griffiths singularities in the transverse-field Ising spin glass*, Phys. Rev. B **54**, 3336 (1996), doi:[10.1103/PhysRevB.54.3336](https://doi.org/10.1103/PhysRevB.54.3336).
- [26] A. D. King, J. Raymond, T. Lanting, R. Harris, A. Zucca, F. Altomare, A. J. Berkley, K. Boothby, S. Ejtemaee, C. Enderud, E. Hoskinson, S. Huang *et al.*, *Quantum critical dynamics in a 5,000-qubit programmable spin glass*, Nature **617**, 61 (2023), doi:[10.1038/s41586-023-05867-2](https://doi.org/10.1038/s41586-023-05867-2).
- [27] M. Bernaschi, I. González-Adalid Pemartín, V. Martín-Mayor and G. Parisi, *The quantum transition of the two-dimensional Ising spin glass*, Nature **631**, 749 (2024), doi:[10.1038/s41586-024-07647-y](https://doi.org/10.1038/s41586-024-07647-y).
- [28] N. Trivedi and D. M. Ceperley, *Ground-state correlations of quantum antiferromagnets: a Green-function Monte Carlo study*, Phys. Rev. B **41**, 4552 (1990), doi:[10.1103/PhysRevB.41.4552](https://doi.org/10.1103/PhysRevB.41.4552).
- [29] M. Calandra Buonaura and S. Sorella, *Numerical study of the two-dimensional Heisenberg model using a Green function Monte Carlo technique with a fixed number of walkers*, Phys. Rev. B **57**, 11446 (1998), doi:[10.1103/PhysRevB.57.11446](https://doi.org/10.1103/PhysRevB.57.11446).
- [30] W. M. C. Foulkes, L. Mitas, R. J. Needs and G. Rajagopal, *Quantum Monte Carlo simulations of solids*, Rev. Mod. Phys. **73**, 33 (2001), doi:[10.1103/RevModPhys.73.33](https://doi.org/10.1103/RevModPhys.73.33).
- [31] G. H. Booth, A. J. W. Thom and A. Alavi, *Fermion Monte Carlo without fixed nodes: A game of life, death, and annihilation in Slater determinant space*, J. Chem. Phys. **131**, 054106 (2009), doi:[10.1063/1.3193710](https://doi.org/10.1063/1.3193710).
- [32] M. Ćufar, C. J. Bradley, R. Yang, E. Pahl and J. Brand, *Rimu.jl: Random integrators for many-body quantum systems*, arXiv:2601.19505 (2026), doi:[10.48550/arXiv.2601.19505](https://doi.org/10.48550/arXiv.2601.19505).
- [33] M. Qin, *Combination of tensor network states and Green's function Monte Carlo*, Phys. Rev. B **102**, 125143 (2020), doi:[10.1103/PhysRevB.102.125143](https://doi.org/10.1103/PhysRevB.102.125143).
- [34] H.-Y. Lin, R.-Q. He and Z.-Y. Lu, *Green's function Monte Carlo method combined with restricted Boltzmann machine approach to the frustrated J_1 - J_2 Heisenberg model*, Chin. Phys. B **31**, 080203 (2022), doi:[10.1088/1674-1056/ac615f](https://doi.org/10.1088/1674-1056/ac615f).
- [35] H.-Y. Lin, Y. Guo, R.-Q. He, Z. Y. Xie and Z.-Y. Lu, *Green's function Monte Carlo combined with projected entangled pair state approach to the frustrated J_1 - J_2 Heisenberg model*, Phys. Rev. B **109**, 235133 (2024), doi:[10.1103/PhysRevB.109.235133](https://doi.org/10.1103/PhysRevB.109.235133).
- [36] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne and Q. Zhang, *JAX: composable transformations of Python+NumPy programs*, <https://github.com/google/jax>, Version 0.9.2 (2018).
- [37] J. Heek, A. Levskaya, A. Oliver, M. Ritter, B. Rondepierre, A. Steiner and M. van Zee, *Flax: a neural network library and ecosystem for JAX*, <https://github.com/google/flax>, Version 0.12.5 (2024).

- [38] F. Vicentini, D. Hofmann, A. Szabó, D. Wu, C. Roth, C. Giuliani, G. Pescia, J. Nys, V. Vargas-Calderón, N. Astrakhantsev and G. Carleo, *NetKet 3: machine learning toolbox for many-body quantum systems*, SciPost Phys. Codebases p. 7 (2022), doi:[10.21468/SciPostPhysCodeb.7](https://doi.org/10.21468/SciPostPhysCodeb.7).
- [39] F. Vicentini, D. Hofmann, A. Szabó, D. Wu, C. Roth, C. Giuliani, G. Pescia, J. Nys, V. Vargas-Calderón, N. Astrakhantsev and G. Carleo, *Codebase release 3.4 for NetKet*, SciPost Phys. Codebases p. 7-r3.4 (2022), doi:[10.21468/SciPostPhysCodeb.7-r3.4](https://doi.org/10.21468/SciPostPhysCodeb.7-r3.4).
- [40] A. Y. Kitaev, *Fault-tolerant quantum computation by anyons*, Ann. Phys. **303**, 2 (2003), doi:[10.1016/S0003-4916\(02\)00018-0](https://doi.org/10.1016/S0003-4916(02)00018-0).
- [41] D. S. Rokhsar and S. A. Kivelson, *Superconductivity and the quantum hard-core dimer gas*, Phys. Rev. Lett. **61**, 2376 (1988), doi:[10.1103/PhysRevLett.61.2376](https://doi.org/10.1103/PhysRevLett.61.2376).
- [42] M. Hermele, M. P. A. Fisher and L. Balents, *Pyrochlore photons: the $U(1)$ spin liquid in a $S = 1/2$ three-dimensional frustrated magnet*, Phys. Rev. B **69**, 064404 (2004), doi:[10.1103/PhysRevB.69.064404](https://doi.org/10.1103/PhysRevB.69.064404).
- [43] O. Benton, O. Sikora and N. Shannon, *Seeing the light: experimental signatures of emergent electromagnetism in a quantum spin ice*, Phys. Rev. B **86**, 075154 (2012), doi:[10.1103/PhysRevB.86.075154](https://doi.org/10.1103/PhysRevB.86.075154).
- [44] N. Shannon, O. Sikora, F. Pollmann, K. Penc and P. Fulde, *Quantum ice: a quantum Monte Carlo study*, Phys. Rev. Lett. **108**, 067204 (2012), doi:[10.1103/PhysRevLett.108.067204](https://doi.org/10.1103/PhysRevLett.108.067204).
- [45] N. Shannon, *Quantum Monte Carlo simulations of quantum spin ice*, In M. Udagawa and L. Jaubert, eds., *Spin Ice*, pp. 273–301. Springer International Publishing, Cham, ISBN 978-3-030-70860-3, doi:[10.1007/978-3-030-70860-3_10](https://doi.org/10.1007/978-3-030-70860-3_10) (2021).
- [46] S. Sorella, *Green function Monte Carlo with stochastic reconfiguration*, Phys. Rev. Lett. **80**, 4558 (1998), doi:[10.1103/PhysRevLett.80.4558](https://doi.org/10.1103/PhysRevLett.80.4558).
- [47] A. W. Sandvik, *Stochastic series expansion method with operator-loop update*, Phys. Rev. B **59**, R14157 (1999), doi:[10.1103/PhysRevB.59.R14157](https://doi.org/10.1103/PhysRevB.59.R14157).
- [48] O. F. Syljuåsen and A. W. Sandvik, *Quantum Monte Carlo with directed loops*, Phys. Rev. E **66**, 046701 (2002), doi:[10.1103/PhysRevE.66.046701](https://doi.org/10.1103/PhysRevE.66.046701).
- [49] K. Ghanem, N. Liebermann and A. Alavi, *Population control bias and importance sampling in full configuration interaction quantum Monte Carlo*, Phys. Rev. B **103**, 155135 (2021), doi:[10.1103/PhysRevB.103.155135](https://doi.org/10.1103/PhysRevB.103.155135).
- [50] S. Chen, Y. Yang, M. Morales and S. Zhang, *Algorithm for branching and population control in correlated sampling*, Phys. Rev. Res. **5**, 043169 (2023), doi:[10.1103/PhysRevResearch.5.043169](https://doi.org/10.1103/PhysRevResearch.5.043169).
- [51] W. Marshall, *Antiferromagnetism*, Proc. R. Soc. Lond. A **232**, 48 (1955), doi:[10.1098/rspa.1955.0200](https://doi.org/10.1098/rspa.1955.0200).
- [52] N. Madras and A. D. Sokal, *The pivot algorithm: A highly efficient Monte Carlo method for the self-avoiding walk*, J. Stat. Phys. **50**, 109 (1988), doi:[10.1007/BF01022990](https://doi.org/10.1007/BF01022990).
- [53] H. Flyvbjerg and H. G. Petersen, *Error estimates on averages of correlated data*, J. Chem. Phys. **91**, 461 (1989), doi:[10.1063/1.457480](https://doi.org/10.1063/1.457480).

- [54] B. Efron, *The Jackknife, the Bootstrap and Other Resampling Plans*, Society for Industrial and Applied Mathematics, Philadelphia, doi:[10.1137/1.9781611970319](https://doi.org/10.1137/1.9781611970319) (1982).
- [55] S. Pilati, E. M. Inack and P. Pieri, *Self-learning projective quantum Monte Carlo simulations guided by restricted Boltzmann machines*, Phys. Rev. E **100**, 043301 (2019), doi:[10.1103/PhysRevE.100.043301](https://doi.org/10.1103/PhysRevE.100.043301).
- [56] R. Jastrow, *Many-body problem with strong forces*, Phys. Rev. **98**, 1479 (1955), doi:[10.1103/PhysRev.98.1479](https://doi.org/10.1103/PhysRev.98.1479).
- [57] H. J. Changlani, J. M. Kinder, C. J. Umrigar and G. K.-L. Chan, *Approximating strongly correlated wave functions with correlator product states*, Phys. Rev. B **80**, 245116 (2009), doi:[10.1103/PhysRevB.80.245116](https://doi.org/10.1103/PhysRevB.80.245116).
- [58] F. Mezzacapo, N. Schuch, M. Boninsegni and J. I. Cirac, *Ground-state properties of quantum many-body systems: entangled-plaquette states and variational Monte Carlo*, New J. Phys. **11**, 083026 (2009), doi:[10.1088/1367-2630/11/8/083026](https://doi.org/10.1088/1367-2630/11/8/083026).
- [59] P. Pfeuty, *The one-dimensional Ising model with a transverse field*, Ann. Phys. **57**, 79 (1970), doi:[10.1016/0003-4916\(70\)90270-8](https://doi.org/10.1016/0003-4916(70)90270-8).
- [60] L. Pauling, *The structure and entropy of ice and of other crystals with some randomness of atomic arrangement*, J. Am. Chem. Soc. **57**, 2680 (1935), doi:[10.1021/ja01315a102](https://doi.org/10.1021/ja01315a102).
- [61] W. F. Giauque and J. W. Stout, *The entropy of water and the third law of thermodynamics. The heat capacity of ice from 15 to 273°K*, J. Am. Chem. Soc. **58**, 1144 (1936), doi:[10.1021/ja01298a023](https://doi.org/10.1021/ja01298a023).
- [62] P. W. Anderson, *Ordering and antiferromagnetism in ferrites*, Phys. Rev. **102**, 1008 (1956), doi:[10.1103/PhysRev.102.1008](https://doi.org/10.1103/PhysRev.102.1008).
- [63] O. Sikora, F. Pollmann, N. Shannon, K. Penc and P. Fulde, *Quantum liquid with deconfined fractional excitations in three dimensions*, Phys. Rev. Lett. **103**, 247001 (2009), doi:[10.1103/PhysRevLett.103.247001](https://doi.org/10.1103/PhysRevLett.103.247001).
- [64] O. Sikora, N. Shannon, F. Pollmann, K. Penc and P. Fulde, *Extended quantum U(1)-liquid phase in a three-dimensional quantum dimer model*, Phys. Rev. B **84**, 115129 (2011), doi:[10.1103/PhysRevB.84.115129](https://doi.org/10.1103/PhysRevB.84.115129).
- [65] J. Kim, A. D. Baczewski, T. D. Beaudet, A. Benali, M. C. Bennett, M. A. Berrill, N. S. Blunt, E. J. L. Borda, M. Casula, D. M. Ceperley, S. Chiesa, B. K. Clark *et al.*, *QMCPACK: an open source ab initio quantum Monte Carlo package for the electronic structure of atoms, molecules and solids*, J. Phys.: Condens. Matter **30**, 195901 (2018), doi:[10.1088/1361-648X/aab9c3](https://doi.org/10.1088/1361-648X/aab9c3).
- [66] D. S. Kufel, J. Kemp, D. Vu, S. M. Linsel, C. R. Laumann and N. Y. Yao, *Approximately symmetric neural networks for quantum spin liquids*, Phys. Rev. Lett. **135**, 056702 (2025), doi:[10.1103/pgnx-11ph](https://doi.org/10.1103/pgnx-11ph).

A Arguments for the production run

The production `run` takes the following arguments:

- `walkers`: the initial walker ensemble ([Section 4.2](#));

- `n_steps`: number of production branching steps;
- `hamiltonian`: the Hamiltonian;
- `step_size`: an integer number of steps for the discrete methods ("`single_step`", "`multi_step`"), or a positive real imaginary-time length β for "`cont_time`" (see [Section 2.3](#));
- `guide`: guiding wavefunction (use `None` if no guide should be used);
- `p_array`: list containing the number of branching steps p defining the projection depth (see [Section 2.6.2](#));
- `method`: propagation scheme (see [Section 2.3](#)):
 - "`single_step`" for step-by-step propagation,
 - "`multi_step`" for batched discrete substeps, or
 - "`cont_time`" for the continuous-time scheme.
 (Default: "`cont_time`");
- `rng_key`: JAX PRNG key (Default: `jax.random.PRNGKey(secrets.randbits(32))`);
- `lam`: diagonal shift λ (only required for discrete propagation methods);
- `n_devices`: number of devices used by `run` (Default: `None`, meaning `jax.device_count()`). Set 1 to force single-device execution. Runs containing diagonal observables automatically fall back to a single device;
- `diag_operators`: diagonal observables as a keyed dictionary as described in [Section 4.4](#) (Default: `None`);
- `off_diag_operators`: off-diagonal observables as a keyed dictionary as described in [Section 4.4](#) (Default: `None`);
- `off_diag_stride`: measure the off-diagonal operators only every `off_diag_stride`-th production step (see [Section 4.4](#)) instead of at every step (Default: 1);
- `off_diag_batch_size`: number of walkers to which an off-diagonal operator is applied at once. This reduces the memory peak when an off-diagonal operator is applied to an ensemble by a factor of $n_{\text{walker}}/\text{off_diag_batch_size}$ (Default: `None`);
- `warmup_steps`: total number of warm-up steps, split across four stages. If `avg_log_weight` (see below) is not given, the warm-up determines `avg_log_weight` (Default: `None`);
- `avg_log_weight`: normalization of the weights at each branching step (see [Equation 6](#)). When not given and `warmup_steps` is not set, 1000 warm-up steps are executed to determine it (Default: `None`);
- `verbosity`: integer 0–10 controlling the amount of printed output (Default: 7);
- `number_of_outputs`: number of progress lines printed during production (Default: 100);
- `return_raw`: return `(walkers, energy, obs, stats, raw)` when `True` (Default), or `(walkers, energy, obs, stats)` when `False`.

The first six arguments are positional; all others are keyword arguments.

B Validation

We have implemented several tests to verify the code within two test suites. The first suite, `core`, checks individual parts of the code. The second suite, `benchmark`, performs end-to-end benchmarks of various models against QuSpin [3, 4], DanceQ [5, 6], and TeNPy [10]. The suites take about 50 and 140 minutes to run on a single CPU, respectively.

B.1 Core tests

The core suite contains over 500 tests. We summarize the most relevant ones below.

Diagonal and off-diagonal estimators.

- We test the auxiliary chain used to measure the off-diagonal operator by measuring the identity operator. When the auxiliary chain is propagated with the same seed as the main Markov chain, the weights of the auxiliary chain equal those of the main Markov chain.
- Diagonal observables are measured both directly and with the (off-diagonal) forward-walking technique. The two estimates must agree to floating-point precision.
- We compare runs with different `off_diag_stride` values against a run without stride.

Guiding wavefunctions.

- A guide may implement any of the three functions: `log_psi`, `log_diff_from_configs`, and `log_diff_from_operator`. We check that all implemented methods are mutually consistent.
- We test consistent writing to and loading from disk using HDF5.
- We have implemented the option to use the exact wavefunction as a guide. We check that the local energy equals the ground-state energy on every configuration. This tests the correct implementation of the transition rates. We further compute off-diagonal estimators with the exact guide.
- We test the NetKet training of each guide.

Other tests.

- For every Hamiltonian class, we probe the functions `get_rates` and `generate_state` against an explicit matrix construction for every configuration of a small system.
- We further check that the `diagonal_weight` function matches the first entry of `get_rates`.
- We ensure that single-device and multi-device runs reproduce the same data within statistical noise.
- We probe writing and loading the walker population to and from disk in HDF5.
- We verify the generation of different walker ensembles.
- We check bitwise identity by splitting a simulation into different segments using the `write_to_HDF5` and `run_from_HDF5` functions.
- A golden-output regression test pins short runs over every combination of propagator, Hamiltonian class, and compatible guide. We measure the observables, configurations, weights, local energies, and statistics.

B.2 Benchmark against other methods

We benchmark PYGFMC against QuSpin [3,4], DanceQ [5,6], and TeNPy [10]. Table 1 lists all tested combinations of Hamiltonian class and couplings. For each combination, we compare the ground-state energy and available diagonal reference observables. We further compare off-diagonal observables for the combinations marked in the last column of Table 1.

The geometries we used are labeled by type and number of sites N . They are partially shown in Figure 8. R4, R9, and R13 are randomly generated graphs with 4, 9, and 13 sites, shown in Figure 8(a–c). T8, T12, and T16 are toric-code lattices [40] on periodic 2×2 , 2×3 , and 2×4 square lattices, where the spins sit on the $N = 2L_x L_y$ edges. To benchmark larger systems with DMRG, we use a two-leg ladder with $L = 50$ rungs and open boundary conditions (L100), see Figure 8(d). The ring-flip model is benchmarked on a pyrochlore lattice containing $N = 128$ sites constructed from $2 \times 2 \times 2$ cubic unit cells and is labeled P128. We compute the ground state in the zero-flux ice sector, whose Hilbert-space dimension is 8.1×10^{10} , using constrained exact diagonalization. The data are obtained by a massively parallelized Lanczos calculation based on DanceQ [5,6]. All models are run with the "multi_step" method. The SpinHalf sets are additionally run with "single_step" and "cont_time" for every geometry.

Hamiltonian	Parameters	Geometry	Off-diag.
Spin	$S = 1, J_z = 1, J_{\pm} = -1$	R4, R9, L100	✓
Spin	$S = 1, J_z = 1, J_{\pm} = -1, J_{\pm\pm} = -0.5, h_z = 4, h_{\pm} = 0.5$	R4, R9, L100	—
Spin	$S = 1, J_z = 1, J_{\pm} = -1, h_{\pm} = 2$	R4, R9, L100	—
SpinHalf	$J_x = -1, J_y = -1, J_z = 1$	R4, R9, R13, L100	—
SpinHalf	$J_x = 0, J_y = 0, J_z = -4, h_x = 4$	R4, R9, R13, L100	✓
SpinHalf	$J_x = 0, J_y = 0, J_z = -4, h_x = 8$	R4, R9, R13, L100	—
BoseHubbard	$t = 0.5, \eta = 0.5, U = 4, \mu = 2, n_{\max} = 2$	R4, R9, L100	—
BoseHubbard	$t = 1, \eta = 0, U = 2, \mu = 1, n_{\max} = 2$	R4, R9, L100	✓
ToricCode	$A_s = 1, B_p = 1, h_z = 0, h_x = 0$	T8, T12, T16	✓
ToricCode	$A_s = 1, B_p = 1, h_z = 0.1, h_x = 0.1$	T8, T12, T16	—
ToricCode	$A_s = 1, B_p = 1, h_z = 0.5, h_x = 0.5$	T8, T12, T16	—
ToricCode	$A_s = 1, B_p = 1, h_z = 2, h_x = 2$	T8, T12, T16	—
PlaquetteModel	$g = 1, \mu = 0$	P128	—
PlaquetteModel	$g = 1, \mu = 1$	P128	✓

Table 1: Models benchmarked against QuSpin [3,4], DanceQ [5,6], and TeNPy [10]. The first column lists the Hamiltonian class, the second lists its parameters, the third lists the underlying lattice, and the fourth marks whether off-diagonal observables were measured. The lattices are labeled by type and number of sites N : RN denotes randomly generated graphs (R4, R9, and R13; Figure 8(a–c)); LN denotes the two-leg ladder with open legs (L100; Figure 8(d)); TN denotes the toric code model on a periodic $L_x \times L_y$ torus with $N = 2L_x L_y$; and P128 denotes the pyrochlore lattice with 128 sites. The R and T geometries are checked against QuSpin (exact diagonalization), and L100 is checked against TeNPy (DMRG). P128 is checked against a Lanczos calculation performed inside the ice manifold ($\mu = 0$) and against exact analytical results at the Rokhsar–Kivelson point ($\mu = 1$). Each row is compared with its reference for the ground-state energy and a set of diagonal observables.

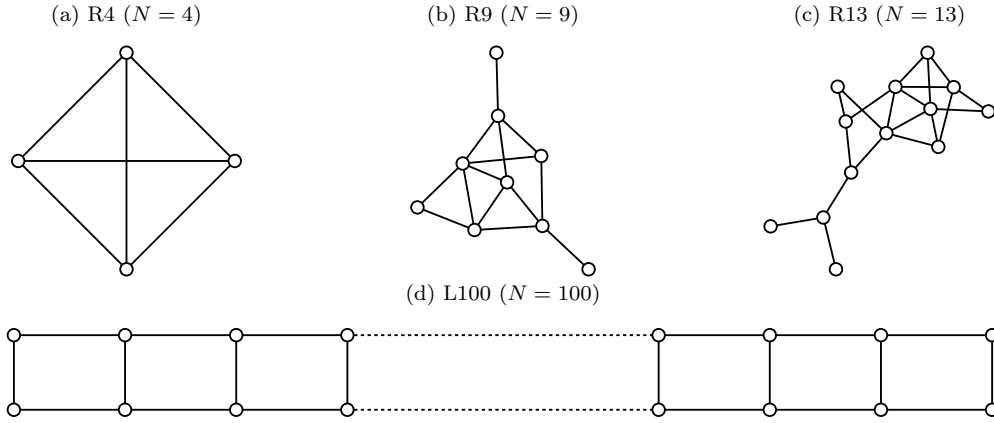


Figure 8: Geometries used in Table 1. (a)–(c): The randomly generated graphs R4, R9, and R13 with $N = 4, 9$, and 13 sites. (d): The two-leg ladder L100 with open boundary conditions.

C Convergence in the pyrochlore XXZ model

We provide additional data on the convergence of the structure factor $S^\pm(\mathbf{q} = 0)$ from Equation 29. Figure 9 shows how $S^\pm(\mathbf{q} = 0)$ converges with the projection depth τ for $N = 1024$. We find that convergence is achieved reliably for $\tau \approx 10$ “deep” within the ferromagnet ($J_{xy}/J_z \leq -0.12$) and “deep” within the quantum spin liquid ($J_{xy}/J_z \geq -0.09$). Hence, simulations were conducted for $\tau \approx 14$ in the ferromagnet and $\tau \approx 20$ in the quantum spin ice phase. In contrast, the transition region $J_{xy}/J_z \in [-0.11, -0.10]$ requires a larger projection depth, as shown in the inset of Figure 9. The order parameter $S^\pm(\mathbf{q} = 0)$ decreases by approximately 2.5% to 8% between $\tau = 20$ and $\tau = 30$, and by 0.2% to 2.7% between $\tau = 30$ and $\tau = 40$. To estimate the remaining error from the finite projection depth, we fit $S^\pm(\mathbf{q} = 0)$ for $\tau \geq 10$ to $f(\tau) = c e^{-a\tau} + b$. The extrapolated value b , which corresponds to the $\tau \rightarrow \infty$ limit, is shown using red markers in Figure 9. It is visually indistinguishable from the data at the largest τ .

We further show the order parameter computed with the guiding wavefunction alone (without GFMC) using black diamonds in Figure 9. While the guide captures the overall trend of a small order parameter in the disordered phase and a strong signal in the ordered phase, it clearly struggles to obtain reliable data around the transition region, making a systematic finite-size scaling impossible. In particular, the guide estimates vary irregularly between neighboring values of J_{xy}/J_z . The variation suggests that optimization ended up in different local minima. However, the GFMC algorithm reliably projects onto the true ground state, resulting in a smooth curve.

D Implemented operator classes

All pre-implemented Hamiltonian classes provide the necessary functions and attributes introduced in Section 4.1: `M_total`, `get_rates`, `generate_state`, and `diagonal_weight`. The classes allow the couplings to be given as a uniform scalar or as per-edge/per-site/per-plaquette arrays. Each constructor verifies that the resulting model is sign-problem-free. The check can be bypassed with `sign_check=False`, which is necessary when a Hamiltonian class is used as an off-diagonal operator. All constructors accept a `dtype` keyword (default `numpy.float32`). We

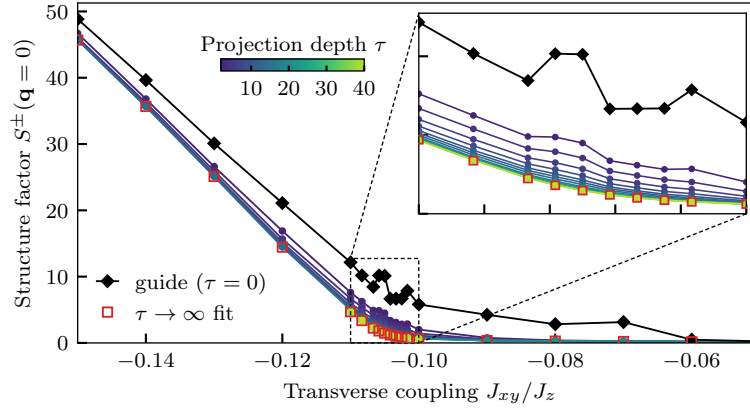


Figure 9: Convergence of $S^\pm(\mathbf{q} = 0)$ from Equation 29 with the projection depth τ for the pyrochlore XXZ model from Section 3.2 at $N = 1024$. The color bar refers to different τ values. Red squares show the extrapolation $\tau \rightarrow \infty$, obtained by fitting $c e^{-a\tau} + b$ to the data for $\tau \geq 10$. Black diamonds show the expectation value from the guiding wavefunction without GFMC. The inset zooms into the transition region $J_{xy}/J_z \in [-0.11, -0.10]$.

recommend using `jnp.float32`, as GPUs are substantially slower when handling `jnp.float64`.

Spin. Spin- S Hamiltonian with single- and two-body terms and a local Hilbert space of dimension $2S + 1$ spanned by the values $-S, \dots, S$:

$$H = \sum_{\langle ij \rangle} [J_{z,ij} S_i^z S_j^z + J_{\pm,ij} (S_i^+ S_j^- + S_i^- S_j^+) + J_{\pm\pm,ij} (S_i^+ S_j^+ + S_i^- S_j^-)] - \sum_i [h_{z,i} S_i^z + h_{\pm,i} (S_i^+ + S_i^-)], \quad (46)$$

where $\langle ij \rangle$ runs over a given edge list. The model is sign-problem-free for $J_\pm, J_{\pm\pm} \leq 0$ and $h_\pm \geq 0$.

```
Spin.create(edges, S=0.5, J_z=0.0, J_pm=0.0, J_pmpm=0.0,
            h_z=None, h_pm=None, N=None, sign_check=True)
```

SpinHalf. This class is optimized for spin-1/2 Hamiltonians:

$$H = \sum_{\langle ij \rangle} (J_{x,ij} S_i^x S_j^x + J_{y,ij} S_i^y S_j^y + J_{z,ij} S_i^z S_j^z) - \sum_i (h_{z,i} S_i^z + h_{x,i} S_i^x). \quad (47)$$

Internally, the transverse couplings are stored as $J_\pm = (J_x + J_y)/4$ (spin exchange) and $J_{\pm\pm} = (J_x - J_y)/4$ (double spin flip). The model is sign-problem-free for $J_\pm, J_{\pm\pm} \leq 0$ and $h_x \geq 0$. Notice that, when $J_{\pm\pm} > 0$ and $h_x = 0$, it is possible to swap J_x and J_y .

```
SpinHalf.create(edges, J_x=0, J_y=0, J_z=0, h_z=None, h_x=None,
                N=None, sign_check=True)
```

BoseHubbard. Bose–Hubbard model with hopping t on each edge, a pairing amplitude η on each edge, on-site interaction U , chemical potential μ , an optional amplitude h_{pm} multiplying both $b_i^\dagger$ and b_i , and a hard occupation cutoff n_{max} :

$$H = - \sum_{\langle ij \rangle} \left[t_{ij} (b_i^\dagger b_j + \text{h.c.}) + \eta_{ij} (b_i^\dagger b_j^\dagger + \text{h.c.}) \right] + \sum_i \left[\frac{U_i}{2} n_i (n_i - 1) - \mu_i n_i - h_{\pm, i} (b_i^\dagger + b_i) \right]. \quad (48)$$

Configurations are occupation numbers $0, \dots, n_{\text{max}}$. The model is sign-problem-free for $t, \eta, h_{\pm} \geq 0$.

```
BoseHubbard.create(edges, t=0.0, eta=0.0, U=0.0, mu=0.0,
                  h_pm=None, N=None, n_max=4, sign_check=True)
```

ToricCode. Toric code on an $L_x \times L_y$ torus with periodic boundaries and stabilizer couplings A_s , B_p and local fields h_x , h_z [40].

$$H = - \sum_v A_s^{(v)} \prod_{i \in \star(v)} \sigma_i^x - \sum_p B_p^{(p)} \prod_{i \in \square(p)} \sigma_i^z - \sum_i h_{z,i} \sigma_i^z - \sum_i h_{x,i} \sigma_i^x. \quad (49)$$

Note that this class uses Pauli operators rather than spin-1/2 operators. The spins live on the $2L_x L_y$ edges of the square lattice. The model is sign-problem-free for $A_s^{(v)}, h_{x,i} \geq 0$ everywhere.

```
ToricCode.create(Lx, Ly, A_s=1.0, B_p=1.0, h_x=0.0, h_z=0.0, sign_check=True)
```

PlaquetteModel. Ring-exchange terms of amplitude g on an arbitrary collection of plaquettes of equal length ℓ , given as an $(N_{\text{plaquette}}, \ell)$ array, together with a Rokhsar–Kivelson potential μ [41]:

$$H = - \sum_{p=(i,j,\dots,k) \in \mathcal{P}} g_p (S_i^+ S_j^- \cdots S_k^- + \text{h.c.}) + \sum_{p \in \mathcal{P}} \mu_p n_p^{\text{flip}}, \quad (50)$$

where the sites in $p = (i, j, \dots, k)$ are ordered around the loop, and n_p^{flip} indicates whether p is flippable. Note that ℓ must be even. Writing the two flippable configurations as $|\odot_p\rangle$ and $|\circlearrowleft_p\rangle$, the same Hamiltonian reads:

$$H = - \sum_p g_p (|\odot_p\rangle \langle \odot_p| + |\circlearrowleft_p\rangle \langle \circlearrowleft_p|) + \sum_p \mu_p (|\odot_p\rangle \langle \odot_p| + |\circlearrowleft_p\rangle \langle \circlearrowleft_p|). \quad (51)$$

The model is sign-problem-free for $g \geq 0$ with μ of either sign.

```
PlaquetteModel.create(plaquettes, g=None, mu=None, sign_check=True)
```

E User-defined operator classes

We demonstrate how the user can construct their own operator classes using spin-1/2 degrees of freedom, which are encoded as $(-1, 1)$. [Section E.1](#) defines an n -body Hamiltonian that acts through a product of σ^z and σ^x on multiple sites. [Section E.2](#) defines the diagonal $S^z S^z$ structure factor.

E.1 An n -body Hamiltonian

The Hamiltonian consists of an on-site field and products of operators acting on given clusters of n sites:

$$H = -J_z \sum_{c \in \mathcal{C}} \prod_{i \in c} \sigma_i^z - J_x \sum_{c \in \mathcal{C}} \prod_{i \in c} \sigma_i^x - h \sum_i \sigma_i^z, \quad (52)$$

where $\mathcal{C}$ is a collection of n -site clusters. Note that we use Pauli operators instead of spin operators. The σ^z part is diagonal and the σ^x part flips every site of a cluster at once. Therefore, each cluster contributes exactly one off-diagonal transition and $M_{\text{total}} = |\mathcal{C}| + 1$. The model is sign-problem-free for $J_x \geq 0$.

```
import jax
import jax.numpy as jnp

class NBody:
    def __init__(self, clusters, J_z, J_x, h=0.0):
        self.clusters = jnp.asarray(clusters)           # (n_terms, n)
        self.J_z, self.J_x, self.h = J_z, J_x, h
        self.M_total = 1 + self.clusters.shape[0]

    def get_rates(self, config, return_configs=False):
        diag = self.diagonal_weight(config)
        off = jnp.full(self.clusters.shape[0], -self.J_x)
        rates = jnp.concatenate([jnp.array([diag]), off])
        if not return_configs:
            return rates, None
        flips = jax.vmap(lambda c: config.at[c].multiply(-1))(self.clusters)
        return rates, jnp.concatenate([config[None, :], flips])

    def generate_state(self, config, idx):
        flipped = config.at[self.clusters[idx - 1]].multiply(-1)
        return jnp.where(idx == 0, config, flipped)

    def diagonal_weight(self, config):
        return (-self.J_z * jnp.sum(jnp.prod(config[self.clusters], axis=1))
                - self.h * jnp.sum(config))

jax.tree_util.register_pytree_node(
    NBody,
    lambda h: ((h.clusters,), (h.J_z, h.J_x, h.h)),
    lambda aux, children: NBody(children[0], *aux))
```

The class is registered as a JAX pytree.

E.2 $S^z S^z$ structure factor

The $S^z S^z$ structure factor at wavevector $\mathbf{k}$,

$$S^{zz}(\mathbf{k}) = \sum_{i < j} (e^{i\mathbf{k} \cdot \mathbf{dr}_{ij}} + e^{-i\mathbf{k} \cdot \mathbf{dr}_{ij}}) S_i^z S_j^z = \sum_{i < j} 2 \cos(\mathbf{k} \cdot \mathbf{dr}_{ij}) S_i^z S_j^z, \quad (53)$$

is diagonal in the computational basis. Hence, it only needs the `diagonal_weight(config)` function. $\mathbf{dr}_{ij}$ is the displacement between sites i and j and is passed to the constructor as one vector per pair so that the weights $2 \cos(\mathbf{k} \cdot \mathbf{dr}_{ij})$ can be precomputed.

```
class PairSzSz:
```

```

def __init__(self, pairs, dr, k):
    self.pairs = jnp.asarray(pairs)                # (n_pairs, 2)
    self.w = 2.0 * jnp.cos(jnp.asarray(dr) @ jnp.asarray(k))

def diagonal_weight(self, config):
    s = config[self.pairs]
    return 0.25 * jnp.sum(self.w * s[:, 0] * s[:, 1])

```

F Implemented guiding wavefunctions

We provide different Jastrow-type guides that can be directly used within different classes. These wavefunctions have the form

$$\log \psi_T(x) = \sum_{c \in \mathcal{C}} \Phi_c(c[x]), \quad (54)$$

where the log-amplitude of each configuration is simply obtained by adding up the contributions of the finite-size clusters $c \in \mathcal{C}$. This ansatz has two advantages. First, it can easily be constructed to respect the symmetries of the system. Second, it can be evaluated quickly, as a transition between configurations typically affects only a subset of the clusters. The guides can be stored in and read from HDF5 via `guide.save(path)` and `Class.load(path)`, so that a previously trained guide can later be used for the Monte Carlo simulation. All constructors accept a `dtype` keyword (default `jnp.float32`).

JastrowSpinHalf. The clusters are the spin pairs $c = \{i, j\}$ listed in `pairs` together with the single sites,

$$\Phi_{\{i,j\}}(\sigma_i^z \sigma_j^z) = -\beta_{ij} \sigma_i^z \sigma_j^z, \quad \Phi_{\{i\}}(\sigma_i^z) = h_i \sigma_i^z. \quad (55)$$

`beta` is either a scalar, resulting in a uniform coupling, or an array whose length matches the number of pairs. Similarly, `h` can be a scalar or an array of size N .

```
JastrowSpinHalf(N, pairs, beta, h=None)
```

JastrowSpin. Similar to `JastrowSpinHalf()` but implemented for the `Spin` class, with $x_i \in \{-S, \dots, S\}$:

$$\Phi_{\{i,j\}}(x_i, x_j) = -\beta_{ij} x_i x_j, \quad \Phi_{\{i\}}(x_i) = h_i x_i - d_i x_i^2. \quad (56)$$

```
JastrowSpin(N, pairs, beta, h=None, d=None)
```

JastrowBoseHubbard. Similar to `JastrowSpinHalf()` but implemented for the `BoseHubbard` class, with x_i the occupation of site i :

$$\Phi_{\{i,j\}}(x_i, x_j) = -\beta_{ij} x_i x_j, \quad \Phi_{\{i\}}(x_i) = h_i x_i - d_i x_i^2. \quad (57)$$

```
JastrowBoseHubbard(N, pairs, beta, h=None, d=None)
```

ClusterCPS. The correlator product state [57, 58] allows for different groups of clusters, where N_G is the number of groups and the log-amplitude is obtained by:

$$\log \psi_T(x) = \sum_{g=1}^{N_G} \sum_{c \in C_g} \Phi_g(c[x]). \quad (58)$$

C_g is a collection of clusters of equal size L_g and Φ_g is a lookup table that maps a configuration to a real number. The lookup table has 2^{L_g} entries since it is defined for the `SpinHalf` class.

Each group g is passed as `[clusters, symmetries, parity]`, where `clusters` is a two-dimensional array of shape $(|C_g|, L_g)$ defining the clusters of the group. `symmetries` contains all automorphisms to be applied to the lookup table, meaning that configurations that are connected by a given automorphism will have the same amplitude. It is a two-dimensional array of size (N_A, L_g) , where N_A is the number of automorphisms provided, or `None` if no automorphism is imposed. `parity` is a Boolean (`True` or `False`) referring to the global spin flip $\sigma^z \rightarrow -\sigma^z$ symmetry. The symmetry holds for $h_z = 0$. The spatial symmetries and parity reduce the table to its inequivalent orbits. For example, the number of free parameters in the lookup table was reduced from $2^{18} = 262144$ to 2110 in the pyrochlore example of Section 3.2.

```
ClusterCPS(groups, amplitudes=None, init='zero', seed=None)
```

`groups` is the list of group specifications introduced above, with one entry per group. An already trained set of lookup tables is passed as `amplitudes` with one array per group. If `amplitudes=None`, the tables are initialized either to zeros (`'zero'`) or to small Gaussian noise (`'gaussian'`). A seed can be set by `seed`.

ClusterMLP It has a similar structure to `ClusterCPS`, but encodes each cluster group by a multilayer perceptron Φ_g instead of a lookup table. See NetKet [38, 39] for a detailed description of such architectures. All clusters of a group share one network, which may have several layers, so that the number of parameters is set by the width and depth of the network rather than by the cluster size L_g . The arguments of the constructor are the direct analog of those of `ClusterCPS`. Each group g is passed as `[clusters, hidden]`. `clusters` is again a two-dimensional array of shape $(|C_g|, L_g)$, and `hidden` lists the widths of the hidden layers of that group's network. For example, if `hidden=[8, 4]` and the clusters of the group contain $L_g = 6$ sites, the network will have the following dimensions: (6, 8, 4, 1). The input layer is fixed by the cluster size and the output layer is the final scalar.

```
ClusterMLP(groups, params=None, seed=None)
```

The weights and biases ($W^{(l)}, b^{(l)}$) of all groups can be passed as `params`, one pair per layer and group, and must be consistent with the layer dimensions implied by `clusters` and `hidden`. If `params=None`, the biases are set to zero and the weights are drawn from a Gaussian of width 0.01.

In our benchmarks, we find that `ClusterMLP` achieves lower energies than `ClusterCPS`. It is possible to train `ClusterMLP` and load the amplitudes per state in `ClusterCPS`. However, `ClusterMLP` breaks symmetries and potentially requires deeper projections within the GFMC algorithm.

ExactGroundState. We further provide the option to use the exact ground state as a guide, $\psi_T(x) = |\langle x | \Psi_0 \rangle|$, obtained by exact diagonalization of the given Hamiltonian. This is mainly

meant for validation.

```
ExactGroundState(
    hamiltonian,
    log_amp=None,          # precomputed log|psi_0|; None diagonalizes
    hamiltonian
    local_hilbert_space=None, # None uses hamiltonian.local_hilbert_space
    N=None,                # number of sites, defaults to hamiltonian.N
)
```

G User-defined guiding wavefunctions

We show how to implement a simple custom guide whose logarithmic amplitude for a configuration is given by the total magnetization times a control parameter β :

$$\log \psi_T(x) = \beta \sum_i \sigma_i^z. \quad (59)$$

While the functions `log_psi` and `log_diff_from_configs` are independent of the underlying Hamiltonian, we also implement `log_diff_from_operator`, which is tailored to the n -body Hamiltonian in [Section E.1](#). The local Hilbert space is encoded as $(-1, 1)$, representing spin-1/2 degrees of freedom.

```
class MagnetizationGuide:
    def __init__(self, beta):
        self.beta = jnp.asarray(beta)

    def log_psi(self, config, cache=None):
        # config is (N,) or (n, N)
        return self.beta * jnp.sum(config, axis=-1)

    def log_diff_from_configs(self, config, new_configs, cache=None):
        return self.log_psi(new_configs) - self.log_psi(config)

    def log_diff_from_operator(self, config, op, cache=None):
        # flipping the sites of cluster k changes the magnetization
        # by -2 sum_{i in cluster k} sigma_i; entry 0 is the diagonal
        dm = -2.0 * jnp.sum(config[op.clusters], axis=1)
        return self.beta * jnp.concatenate([jnp.zeros(1, dtype=dm.dtype), dm])

jax.tree_util.register_pytree_node(
    MagnetizationGuide,
    lambda g: ((g.beta,), None),
    lambda aux, children: MagnetizationGuide(*children),
)
```

H NetKet integration

PyGFMC offers a bridge for training PyGFMC guiding wavefunctions directly with NetKet [38, 39]. By passing the PyGFMC Hamiltonian and the guide, together with the settings for the training procedure (see the NetKet documentation for more details), the variational optimization is carried out by NetKet. The guide only requires `log_psi` and a registration as a pytree whose floating-point children are the variational parameters (see the implementa-

tion of the guides in [Appendix F](#) and [Appendix G](#) for details). The Hamiltonian only requires `get_rates`. A new guide of the same class is returned with optimized parameters.

It is often desirable to constrain the variational states to respect the underlying symmetries of the Hamiltonian, such as particle number conservation. For this purpose, one can pass `initial_states` and a `sampler` that restrict the states to a given symmetry sector. The default option is `MetropolisExchange`, which conserves the particle number.

```
from pygfmc.wavefunction import train_with_netket

guide = train_with_netket(
    hamiltonian,          # only get_rates is required
    guide,                # only log_psi is required
    local_hilbert_space=None, # defaults to hamiltonian.local_hilbert_space
    N=None,               # defaults to hamiltonian.N
    sampler=None,         # NetKet sampler; carries its own Hilbert space
    initial_states=None,  # (n, N) states seeding the Metropolis chains,
                        # resampled to n_chains, so n is unrestricted
    n_iter=100,           # VMC iterations
    learning_rate=0.02,
    n_samples=1024,       # samples per iteration
    n_chains=16,          # parallel Metropolis chains
    n_discard_per_chain=8, # warm-up samples dropped per chain
    sr=False,             # stochastic reconfiguration
    diag_shift=1e-3,      # SR regularization
    qgt=None,             # quantum geometric tensor implementation
    seed=0,
    show_progress=False,
    return_history=False, # also return the per-iteration energies
)
```