ParaToric 1.0-beta: Continuous-time quantum Monte Carlo for the toric code in a parallel field

Simon M. Linsel[⋆] **and Lode Pollet**[†]

Department of Physics and Arnold Sommerfeld Center for Theoretical Physics (ASC), Ludwig-Maximilians-Universität München, Theresienstr. 37, München D-80333, Germany
Munich Center for Quantum Science and Technology (MCQST), Schellingstr. 4, D-80799 München, Germany

⋆ simon.linsel@lmu.de , † lode.pollet@lmu.de

Abstract

We introduce ParaToric, a C++ package for simulating the toric code in a parallel field (i.e., X- and Z-fields) at finite temperature. We implement and extend the continuous-time quantum Monte Carlo algorithm of Wu, Deng, and Prokof'ev on the square, triangular, honeycomb, and cubic lattices with open and periodic boundaries, respectively. The package is expandable to arbitrary lattice geometries and custom observables diagonal in either the X- or Z-basis. ParaToric also supports snapshot extraction in both bases, making it ideal for generating training/benchmarking data for other methods, such as lattice gauge theories, cold atom or other quantum simulators, quantum spin liquids, artificial intelligence, and quantum error correction. The software provides bindings to C/C++ and Python, and is thus almost universally integrable into other software projects.

Received Date
Accepted Date
Published Date

Contents

1 Introduction

The toric code is one of the most fundamental and most-studied models in modern condensed matter physics. It was first written down by Kitaev [1] and is the simplest example of a model hosting a topological phase (a gapped $\mathbb{Z}_2$ quantum spin liquid) and anyonic excitations. The toric code is also the foundational model for error-correcting codes [2, 3] and has deep connections to the Ising gauge theory [4].

The toric code can be extended with fields which, when strong enough, destroy the topological order. This model is sign-problem-free, thus making quantum Monte Carlo the method of choice. Wu, Deng, and Prokof'ev developed a continuous-time quantum Monte Carlo algorithm [5]. ParaToric implements and extends this algorithm with new updates which enable ergodicity at large temperatures and at zero off-diagonal field, thus significantly improving the applicability of the algorithm.

ParaToric implements a wide range of lattices, boundary conditions, and observables. It is also possible to extend ParaToric with new interactions, observables, and lattices. We provide documented interfaces in C, C++, and Python as well as command-line interfaces, making the integration of ParaToric into other projects and programming languages straightforward. ParaToric will save simulation results to HDF5 files and snapshots to GraphML files (XML-based), with a focus on interoperability with other packages. ParaToric comes with an MIT license.

2 The toric code in a parallel field

2.1 Hamiltonian

ParaToric implements and extends the continuous-time quantum Monte Carlo (QMC) algorithm by Wu, Deng, and Prokof'ev [5] to simulate the toric code in a parallel field (also called perturbed toric code or extended toric code)

$$\hat{\mathcal{H}} = -\mu \sum_v \hat{A}_v \; - J \sum_p \hat{B}_p \; - h \sum_l \hat{\sigma}_l^x \; - \lambda \sum_l \hat{\sigma}_l^z, \tag{1}$$

where $J, \lambda > 0$ in the $\hat{\sigma^x}$-basis and $\mu, h > 0$ in the $\hat{\sigma^z}$-basis (otherwise the model has a sign-problem). $\hat{\sigma}_l^x$ and $\hat{\sigma}_l^z$ are Pauli matrices defined on the links of the underlying lattice. The star term $\hat{A}_v$ contains all links adjacent to lattice site v, the plaquette term $\hat{B}_p$ contains all links that belong to the same elementary plaquette p of the underlying lattice. The temperature $T = 1/\beta$ is finite. For readers interested in extending the code, we note that it is relatively straightforward to add interactions that are diagonal in the chosen basis, such as (long-range) Ising interactions. Off-diagonal interactions require a more careful review and extension of the Monte Carlo updates to ensure ergodicity. However, diagonal interactions can also lead to sampling problems, especially when they introduce frustration.

2.2 Lattice geometries

We implement the square, honeycomb, triangular, and cubic lattices, see Fig. 1. On the cubic lattice, the plaquettes contain the four links of cube faces, *not* the twelve links of the cube

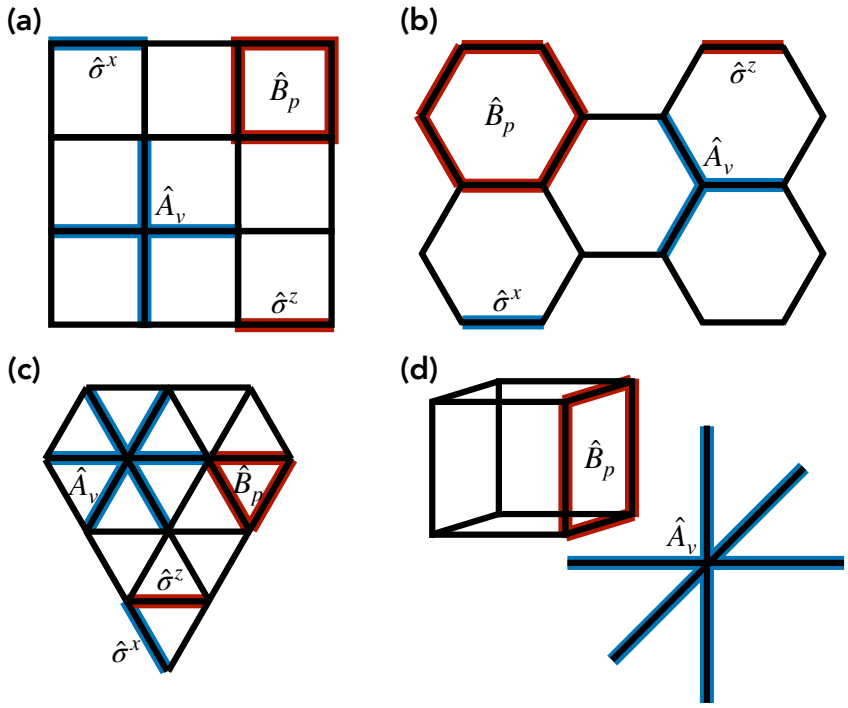

Figure 1: **Implemented lattices**. We implement the extended toric code (1) on the square (a), honeycomb (b), triangular (c), and cubic lattices (d). For each lattice, we show the star ($\hat{A}_v$) and plaquette ($\hat{B}_p$) terms. The cubic lattice importantly features star interactions of six links and plaquette interactions of four links on the faces of cubes.

(that model has a different m-anyon structure). We implement open and periodic boundaries, respectively. New lattices can be added in `src/lattice/lattice.cpp`.

2.3 Observables

Here we list all observables that ParaToric implements for the extended toric code. Custom observables can be added in `src/mcmc/extended_toric_code_qmc.hpp`. For each observable $\hat{O}$, we calculate the expectation value $\langle\hat{O}\rangle$ and the Binder ratio $U_O = \frac{\langle\hat{O}^4\rangle}{\langle\hat{O}^2\rangle^2}$ with error bars obtained from bootstrapping (see below), respectively.

`anyon_count` The number of e-anyons ($\hat{\sigma^x}$-basis) or m-anyons ($\hat{\sigma^z}$-basis) in the system.

`anyon_density` $\hat{\sigma^x}$-basis: The number of e-anyons divided by the number of lattice sites. $\hat{\sigma^z}$-basis: The number of m-anyons divided by the number of plaquettes.

`delta` The difference of the star and plaquette expectation values: $\Delta = \langle\hat{A}_v\rangle - \langle\hat{B}_p\rangle$.

`energy` The total energy $E = \langle\hat{\mathcal{H}}\rangle$.

`energy_h` The electric field term $E_h = \langle -h\sum_l \hat{\sigma}^x_l\rangle$.

`energy_lmbda` The gauge field term $E_\lambda = \langle -\lambda\sum_l \hat{\sigma}^z_l\rangle$. We write λ as `lmbda` because some programming languages feature `lambda` as a keyword.

`energy_J` The plaquette term $E_J = \langle -J \sum_p \hat{B}_p \rangle$.

`energy_mu` The star term $E_\mu = \langle -\mu \sum_v \hat{A}_v \rangle$.

`fredenhagen_marcu` The equal-time Fredenhagen-Marcu loop operator [6–8]:

$$O_{\mathrm{FM}}^{x/z} = \lim_{L\to\infty} \frac{\langle \prod_{l\in\mathcal{C}_{1/2}^{x/z}} \hat{\sigma}_l^{x/z} \rangle}{\sqrt{|\langle \prod_{l\in\mathcal{C}^{x/z}} \hat{\sigma}_l^{x/z} \rangle|}}, \tag{2}$$

$\mathcal{C}_{1/2}^{x/z}$ is half, $\mathcal{C}^{x/z}$ is a full Wilson loop in the $\hat{\sigma}^x$-basis ('t Hooft loop in the $\hat{\sigma}^z$-basis). The loop is automatically constructed for all supported lattices, and the perimeter scales with $\mathcal{O}(L)$, where L is the linear system size. When probing perimeter/area laws, the user should change L. We currently do not support off-diagonal loop operators, e.g., measuring products of $\hat{\sigma}^x$-operators in the $\hat{\sigma}^z$-basis.

`largest_cluster` The largest connected cluster of neighboring bonds with $\hat{\sigma}^x = -1$ ($\hat{\sigma}^z = -1$) in the $\hat{\sigma}^x$-basis ($\hat{\sigma}^z$-basis). This observable is used to calculate the percolation strength, see [9].

`percolation_probability` Measures the bond percolation probability, i.e. if we can wind around the system while only traversing bonds with $\hat{\sigma}^x = -1$ ($\hat{\sigma}^z = -1$) in the $\hat{\sigma}^x$-basis ($\hat{\sigma}^z$-basis). Formally, it is the expectation value $\langle \hat{\Pi}^{x/z} \rangle$ of the projector

$$\hat{\Pi}^{x/z} = \sum_{W(j)\neq 0} |\{\hat{\sigma}^{x/z}\}_j\rangle\langle\{\hat{\sigma}^{x/z}\}_j|, \tag{3}$$

over all possible configurations $\{\hat{\sigma}^{x/z}\}_j$ with non-zero winding number $W(j)$ of connected link clusters of neighboring $\hat{\sigma}^{x/z} = -1$. These clusters are called percolating clusters. For details, see [9–11].

`percolation_strength` If a snapshot does not have a percolating cluster, the percolation strength is 0. If a snapshot has a percolating cluster, the percolation strength is defined as the result of `largest_cluster` divided by the total number of links in the system. For details, see [9, 11].

`plaquette_percolation_probability` Similar to the percolation probability of bonds. Two plaquettes are in the same cluster if they share a link l with $\hat{\tau}_l^x = -1$. For details, see [11].

`plaquette_z` The plaquette expectation value $\langle \hat{B}_p \rangle$.

`sigma_x` The electric field expectation value $\langle \hat{\sigma}^x \rangle$.

`sigma_x_susceptibility` The static susceptibility

$$\chi^x = \frac{1}{N} \int_0^\beta \langle \hat{\sigma}^x(0)\, \hat{\sigma}^x(\tau) \rangle_{\mathrm{c}}\, \mathrm{d}\tau, \tag{4}$$

where N is the total number of links (qubits) and the integral is over the imaginary time τ[1]. Importantly, χ^x can be calculated both in the $\hat{\sigma}^x$- and the $\hat{\sigma}^z$-basis.

[1] The dynamical (fidelity) susceptibility is a trivial extension and contains an extra τ dependency in the integral.

`sigma_z` The gauge field expectation value $\langle\hat{\sigma}^z\rangle$.

`sigma_z_susceptibility` The static susceptibility

$$\chi^z = \frac{1}{N}\int_0^\beta \langle\hat{\sigma}^z(0)\,\hat{\sigma}^z(\tau)\rangle_c \,\mathrm{d}\tau, \tag{5}$$

where N is the total number of links (qubits) and the integral is over the imaginary time τ. Importantly, χ^z can be calculated both in the $\hat{\sigma}^x$- and the $\hat{\sigma}^z$-basis.

`staggered_imaginary_times` Order parameter from [5]. It is defined as

$$\begin{aligned} O_{\mathrm{SIT}}^{x/z} = &\frac{1}{\beta}\big[(\tau_1^k - 0) - (\tau_2^k - \tau_1^k) + \ldots + (-1)^{N(k)-1} \\ &(\tau_{N(k)}^k - \tau_{N(k)-1}^k) + (-1)^{N(k)}(\beta - \tau_{N(k)}^k)\big], \end{aligned} \tag{6}$$

where τ_n^k is the imaginary time of the n-th tuple spin flip of type k. k is a plaquette p (star s) of links in the $\hat{\sigma}^x$-basis ($\hat{\sigma}^z$-basis). This order parameter can neither be evaluated from snapshots, nor from any other method that does not have access to imaginary time.

`star_x` The star expectation value $\langle\hat{A}_v\rangle$.

`string_number` The total number of links with $\hat{\sigma}^x = -1$ in the $\hat{\sigma}^x$-basis ($\hat{\sigma}^z = -1$ in the $\hat{\sigma}^z$-basis).

3 Installation & interfaces

There are five ways to use paratoric, directly from within code (C, C++, Python) or via the command-line (C++, Python). All interfaces require compiling C++ code. We tested the compilation with GCC 15 and Clang 20.

All interfaces implement three functionalities. *Thermalization* simulations are used to benchmark the thermalization process of a Markov chain and are primarily a diagnostic tool. Regular *sampling* routines are used for generating snapshots and measuring observables, e.g., in the context of continuous phase transitions. *Hysteresis* routines are a variant of the regular sampling routines where not one but an array of Hamiltonian parameters is provided and only one Markov chain is used for all parameters. The order of the Hamiltonian parameters in the input array matters: The last state of the previous parameter is used as an initial state for the thermalization phase of the next parameters. This simulation type should primarily be used when mapping out hysteresis curves in the vicinity of first-order phase transitions, hence the name. Since the hysteresis simulation returns the values of not one but many parameter sets, the output types are generally different from the regular sampling. It is also much slower than regular sampling, because the simulation for different parameters can in general not be parallelized.

3.1 C++ interface

The C++ interface enables users to use a ParaToric public header from within another C++ project.

3.1.1 Build & Installation

The core requires C++23, CMake ≥3.23, and Boost ≥1.87 (older Boost versions may work, but were not tested). To compile it, run:

```
cmake -S . -B build -DCMAKE_BUILD_TYPE=Release \
-DPARATORIC_ENABLE_NATIVE_OPT=ON -DPARATORIC_LINK_MPI=OFF \
-DPARATORIC_BUILD_TESTS=ON
cmake --build build -jN
ctest --test-dir build -jN --output-on-failure
cmake --install build
```

Replace `N` with the number of cores to use, e.g. `-j4` for 4 cores.

- `-DCMAKE_BUILD_TYPE=Release`. Only set to `Debug` if you're a developer.
- `-DCMAKE_INSTALL_PREFIX`. By default, executables install to `${CMAKE_SOURCE_DIR}/${CMAKE_INSTALL_BINDIR}/`, headers to `${CMAKE_INSTALL_INCLUDEDIR}/paratoric`, and static libraries to `${CMAKE_SOURCE_DIR}/${CMAKE_INSTALL_LIBDIR}/`. The Python scripts expect `${CMAKE_SOURCE_DIR}/bin/`; this directory always contains the `paratoric` executable. To install into a custom directory, pass it via `-DCMAKE_INSTALL_PREFIX`, e.g. `-DCMAKE_INSTALL_PREFIX=/your/custom/directory/`.
- `-DPARATORIC_EXPORT_COMPILE_COMMANDS=ON`. Export `compile_commands.json` for tooling.
- `-DPARATORIC_LINK_MPI=OFF`. Link the core to MPI, required on some clusters. The core itself does not need MPI.
- `-DPARATORIC_ENABLE_NATIVE_OPT=OFF`. Turn on `-march=native` on GCC and Clang.
- `-DPARATORIC_ENABLE_AVX2=OFF`. Enable AVX2 (Haswell New Instructions). Requires a CPU which supports AVX2.
- `-DPARATORIC_BUILD_TESTS=PROJECT_IS_TOP_LEVEL`. Compile the tests (recommended).

CMake usage (installed package)

```
cmake_minimum_required(VERSION 3.23)
project(my_qmc_app CXX)

find_package(paratoric CONFIG REQUIRED)   # provides paratoric::core

add_executable(myapp main.cpp)
target_link_libraries(myapp PRIVATE paratoric::core)
```

CMake usage (as subdirectory)

If the core lives in `deps/paratoric`, add it and link to the same target:

```
add_subdirectory(deps/paratoric)
add_executable(myapp main.cpp)
target_link_libraries(myapp PRIVATE paratoric::core)
```

3.1.2 Public class `ExtendedToricCode`

The interface class `ExtendedToricCode` lives in the public header `#include<paratoric/mcmc/extended_toric_code.hpp>`. All symbols are in the `paratoric` namespace. All methods are static, take a single `Config` object and return a `Result` object. The required fields in config are documented for each method within the docstrings.

`Result ExtendedToricCode::get_thermalization(Config config)` Run thermalization only. Required fields: `lat_spec.{basis,lattice_type,system_size,beta,boundaries,default_spin}`, `param_spec.{mu,h,J,lmbda,h_therm,lmbda_therm}`, `sim_spec.{N_thermalization,N_resamples,custom_therm,observables,seed}`, `out_spec.{path_out,save_snapshots}`.

`Result ExtendedToricCode::get_sample(Config config)` Run a production measurement pass. Returns the observables selected in `config`. Required fields: `lat_spec.{basis,lattice_type,system_size,beta,boundaries,default_spin}`, `param_spec.{mu,h,J,lmbda}`, `sim_spec.{N_samples,N_thermalization,N_between_samples,N_resamples,observables,seed}`, `out_spec.{path_out,save_snapshots}`.

`Result ExtendedToricCode::get_hysteresis(Config config)` Perform a hysteresis sweep, where the last state of the previous parameter is used as the initial state of the following parameter in `h_hys` & `lmbda_hys`. Required fields: `lat_spec.{basis,lattice_type,system_size,beta,boundaries,default_spin}`, `param_spec.{mu,J,h_hys,lmbda_hys}`, `sim_spec.{N_samples,N_thermalization,N_between_samples,N_resamples,observables,seed}`, `out_spec.{paths_out,save_snapshots}`.

3.1.3 Configuration type

The struct `Config` (declared in `<paratoric/types/types.hpp>`) contains multiple nested specifications.

Top-level configuration: `Config`

Field	Type	Purpose
`sim_spec`	`SimSpec`	Simulation / MC controls (backend-consumed).
`param_spec`	`ParamSpec`	Model couplings / parameters (backend-consumed).
`lat_spec`	`LatSpec`	Lattice geometry and basis.
`out_spec`	`OutSpec`	Output folders and snapshot toggles.

Simulation specification (`config.sim_spec`)

Field	Type	Meaning / Defaults
`N_samples`	`int`	Number of recorded snapshots. Default 1000.
`N_thermalization`	`int`	Number of warmup steps before sampling. Typically $O(L^d)$, where L is the system size and d is the dimensionality. Default 10000.
`N_between_samples`	`int`	Steps between consecutive snapshots. Higher value decreases autocorrelation and improves error bars. Typically $O(L^d)$, where L is the system size and d is the dimensionality. Default 1000.
`N_resamples`	`int`	Bootstrap resamples for errors. Default 1000.
`custom_therm`	`bool`	Use custom thermalization schedule. Default `false`.
`seed`	`int`	PRNG seed. 0 means "random seed." Default 0.
`observables`	`vector<string>`	Names of observables to record each snapshot. For options, see Sec. 2.3.

Parameter specification (`config.param_spec`)

Field	Type	Meaning / Defaults
`mu`	`double`	Star term coefficient. Default `1.0`.
`h`	`double`	Electric field term. Default `0.0`.
`J`	`double`	Plaquette term. Default `1.0`.
`lmbda`	`double`	Gauge-field term. Default `0.0`.
`h_therm`	`double`	Thermalization value for `h` when using custom schedules. Default `NaN` (unused).
`lmbda_therm`	`double`	Thermalization value for `lmbda` when using custom schedules. Default `NaN` (unused).
`h_hys`	`vector<double>`	Sweep values of `h` for hysteresis runs. Default empty. Length must match `lmbda_hys`.
`lmbda_hys`	`vector<double>`	Sweep values of `lmbda` for hysteresis runs. Default empty. Length must match `h_hys`.

Lattice specification (`config.lat_spec`)

Field	Type	Meaning / Valid values
`basis`	`char`	Spin eigenbasis for the simulation. Must be `'x'` or `'z'`.
`lattice_type`	`string`	The lattice (`"square"`, `"triangular"`, `"honeycomb"` or `"cubic"`).
`system_size`	`int`	Linear system size (per dimension).
`beta`	`double`	Inverse temperature $\beta > 0$.
`boundaries`	`string`	Boundary condition: `"periodic"` or `"open"`.
`default_spin`	`int`	Initial link spin, must be $+1$ or -1.

Output specification (`config.out_spec`)

Field	Type	Meaning
`path_out`	`string`	Primary output folder name.
`paths_out`	`vector<string>`	Hysteresis subfolder names. Length must match `h_hys`.
`save_snapshots`	`bool`	Save snapshots toggle. Default `false`.

253 ### 3.1.4 Return type

Field	C++ Type	Meaning
series	vector<vector<variant<complex<double>,double>>>	Time series of all requested observables, thermalization is excluded (except for thermalization simulation). Outer index = observable, inner index = time point.
acc_ratio	vector<double>	Time series of Monte Carlo acceptance ratios.
mean	vector<double>	Bootstrap observable means.
mean_std	vector<double>	Bootstrap standard errors of the mean.
binder	vector<double>	Bootstrap binder ratios.
binder_std	vector<double>	Bootstrap standard errors of the binder ratios.
tau_int	vector<double>	Estimated integrated autocorrelation times.
series_hys	vector<vector<vector<variant<complex<double>,double>>>>	Hysteresis time series of all requested observables, thermalization is excluded (except for thermalization simulation).
		254 Outer vector = hysteresis parameters (order as in h_hys,lmbda_hys), middle vector = observables (order as in observables), inner vector = time series.
mean_hys	vector<vector<double>>	Hysteresis bootstrap observable means. Outer vector = hysteresis parameters (order as in h_hys,lmbda_hys), inner vector = observables (order as in observables).
mean_std_hys	vector<vector<double>>	Hysteresis bootstrap standard errors of the mean. Indices as above.
binder_hys	vector<vector<double>>	Hysteresis bootstrap binder ratios. Indices as above.
binder_std_hys	vector<vector<double>>	Hysteresis bootstrap standard errors of the binder ratios. Indices as above.
tau_int_hys	vector<vector<double>>	Hysteresis estimated integrated autocorrelation times. Indices as above.

255 ### 3.1.5 C++ usage examples

Listing 1: C++ API - Minimal call

```cpp
// C++23
#include <iostream>
#include <print>
#include <vector>
#include <string>
#include <paratoric/mcmc/extended_toric_code.hpp>
#include <paratoric/types/types.hpp>

int main() {
  using namespace paratoric;

  Config cfg{};

  // ---- lattice sub-config (required) ----
  cfg.lat_spec.basis = 'z'; // or 'x'
  cfg.lat_spec.lattice_type = "square"; // or "cubic", "honeycomb", ...
  cfg.lat_spec.system_size = 16;
  cfg.lat_spec.beta = 8.0;
  cfg.lat_spec.boundaries = "periodic"; // or "open"
  cfg.lat_spec.default_spin = 1;

  // ---- Hamiltonian parameters ----
  cfg.param_spec.mu = 1.0; // star term
  cfg.param_spec.J = 1.0; // plaquette term
  cfg.param_spec.h = 0.20; // electric field term
  cfg.param_spec.lmbda = 0.00; // gauge-field term

  // Optional thermalization schedule values (used if custom_therm = true)
  cfg.param_spec.h_therm = std::numeric_limits<double>::quiet_NaN();
  cfg.param_spec.lmbda_therm = std::numeric_limits<double>::quiet_NaN();

  // (Optional) Hysteresis sweep grids - only read by get_hysteresis(...)
  cfg.param_spec.h_hys = {}; // e.g. {0.0, 0.1, 0.2, 0.3, 0.2, 0.1, 0.0}
  cfg.param_spec.lmbda_hys = {}; // e.g. {0.0, 0.1, 0.2, 0.3, 0.2, 0.1, 0.0}

  // ---- Simulation (MC) controls ----
  cfg.sim_spec.N_samples = 0; // 0 => thermalization-only
  cfg.sim_spec.N_thermalization = 5000; // warmup steps
  cfg.sim_spec.N_between_samples = 10; // thinning between snapshots
  cfg.sim_spec.N_resamples = 1000; // bootstrap
  cfg.sim_spec.custom_therm = false; // set true to use *_therm values
  cfg.sim_spec.seed = 12345; // 0 => random seed

  // Observables to record each snapshot (backend-recognized names)
  cfg.sim_spec.observables = {
    "energy", // total energy
    "plaquette_z", // plaquette energy
    "anyon_count", // number of anyons (x-basis: e-anyons, z-basis: m-anyons
        )
    "fredenhagen_marcu" // example: Wilson/'t Hooft loop proxy
  };

  // ---- Output / I/O policy ----
  cfg.out_spec.path_out = "runs/sample"; // single-run output dir
  cfg.out_spec.paths_out = {}; // filled only for hysteresis
  cfg.out_spec.save_snapshots = false; // set true to dump every snapshot
```

```
  cfg.out_spec.full_time_series = true; // save full time series (FCS)

  // 1) Check thermalization
  Result warmup = ExtendedToricCode::get_thermalization(cfg);
  std::print("Thermalization series: {}", warmup.series);

  // 2) Production sample (set N_samples > 0 and call get_sample)
  cfg.sim_spec.N_samples = 2000;
  Result out = ExtendedToricCode::get_sample(cfg);
  std::print("Production autocorrelations: {}", out.tau_int);

  return 0;
}
```

3.2 C++ command-line interface

ParaToric ships a C++ command-line interface `${CMAKE_INSTALL_PREFIX}/${CMAKE_INSTALL_BINDIR}/paratoric` that orchestrates C++ backends, runs sweeps, and writes HDF5 (observables) and XML (snapshots) outputs.

3.2.1 Build & Installation

The command-line interface requires HDF5 ≥1.14.3 (older HDF5 versions may work, but were not tested). The core requires C++23, CMake ≥3.23, and Boost ≥1.87 (older Boost versions may work, but were not tested). To compile it, run:

```
cmake -S . -B build -DCMAKE_BUILD_TYPE=Release \
-DPARATORIC_ENABLE_NATIVE_OPT=ON -DPARATORIC_LINK_MPI=OFF \
-DPARATORIC_BUILD_TESTS=ON -DPARATORIC_BUILD_CLI=ON
cmake --build build -jN
ctest --test-dir build -jN --output-on-failure
cmake --install build
```

Replace `N` with the number of cores to use, e.g. `-j4` for 4 cores.

- `-DCMAKE_BUILD_TYPE=Release`. Only set to `Debug` if you're a developer.
- `-DCMAKE_INSTALL_PREFIX`. By default, executables install to `${CMAKE_SOURCE_DIR}/${CMAKE_INSTALL_BINDIR}/`, headers to `${CMAKE_INSTALL_INCLUDEDIR}/paratoric`, and static libraries to `${CMAKE_SOURCE_DIR}/${CMAKE_INSTALL_LIBDIR}/`. The Python scripts expect `${CMAKE_SOURCE_DIR}/bin/`; this directory always contains the `paratoric` executable. To install into a custom directory, pass it via `-DCMAKE_INSTALL_PREFIX`, e.g. `-DCMAKE_INSTALL_PREFIX=/your/custom/directory/`.
- `-DPARATORIC_EXPORT_COMPILE_COMMANDS=ON`. Export `compile_commands.json` for tooling.
- `-DPARATORIC_LINK_MPI=OFF`. Link the core to MPI, required on some clusters. The core itself does not need MPI.
- `-DPARATORIC_ENABLE_NATIVE_OPT=OFF`. Turn on `-march=native` on GCC and Clang.

- `-DPARATORIC_ENABLE_AVX2=OFF`. Enable AVX2 (Haswell New Instructions). Requires a CPU which supports AVX2.
- `-DPARATORIC_BUILD_TESTS=PROJECT_IS_TOP_LEVEL`. Compile the tests (recommended).
- `-DPARATORIC_BUILD_CLI=PROJECT_IS_TOP_LEVEL` Required for both the C++ and Python command line interface.

362 **Global options**

Long flag	Short	Type	Description
`--simulation`	`-sim`	string	Simulation mode: `etc_sample`, `etc_hysteresis`, `etc_thermalization`.
`--N_samples`	`-Ns`	int	Number of recorded samples.
`--N_thermalization`	`-Nth`	int	Thermalization (warmup) steps.
`--N_between_samples`	`-Nbs`	int	Steps between samples (thinning).
`--beta`	`-bet`	double	Inverse temperature $\beta = 1/T$.
`--mu_constant`	`-muc`	double	Star-term coupling μ.
`--J_constant`	`-Jc`	double	Plaquette coupling J.
`--h_constant`	`-hc`	double	Field h.
`--lmbda_constant`	`-lmbdac`	double	Field λ.
`--h_constant_therm`	`-hct`	double	Thermalization value for h (used if custom therm).
`--lmbda_constant_therm`	`-lmbdact`	double	Thermalization value for λ.
`--h_hysteresis`	`-hhys`	list<double>	Hysteresis schedule for h (space-separated). Length must match `lmbdahys`.
`--lmbda_hysteresis`	`-lmbdahys`	list<double>	Hysteresis schedule for λ. Length must match `hhys`.
`--N_resamples`	`-Nr`	int	Bootstrap resamples (error bars).
`--custom_therm`	`-cth`	bool	Use thermalization values (`0`/`1`).
`--observables`	`-obs`	list<string>	Measured observables (space-separated).
`--seed`	`-s`	int	PRNG seed; `0` means random seed.
`--basis`	`-bas`	char	Spin basis: `'x'` or `'z'`.
`--lattice_type`	`-lat`	string	Lattice type (e.g. `square`, `cubic`, ...).
`--system_size`	`-L`	int	Linear lattice size (per dimension).
`--boundaries`	`-bound`	string	`periodic` or `open`.
`--default_spin`	`-dsp`	int	Initial link spin (`+1` or `-1`).
`--output_directory`	`-outdir`	path	Output directory path.
`--folder_name`	`-fn`	string	Subfolder (of output directory) name for single run.
`--folder_names`	`-fns`	list<string>	Subfolders (of output directory) for hysteresis steps. Length must match `lmbdahys`.
`--snapshots`	`-snap`	bool	Save snapshots into specified subfolders of output directory.
`--full_time_series`	`-fts`	bool	Save full time series toggle.
`--process_index`	`-procid`	int	Process identifier (logging/debug).

364 ### 3.2.2 `etc_sample`

365 Runs a production measurement pass with the supplied configuration.

Listing 2: Example usage

366

```
./paratoric -sim etc_sample -Ns 2000 -Nth 5000 -Nbs 10 -Nr 1000 -bet 16.0 -
    muc 1 -Jc 1 -hc 0.2 -lmbdac 0.0 -obs energy plaquette_z anyon_count -bas
     z -lat square -L 16 -bound periodic -dsp 1 -outdir ./runs/sample -snap
    =0 -fcs=1
```

3.2.3 etc_hysteresis

Runs a parameter sweep where the last state of step i initializes step i+1. Provide `--h_hysteresis` and `--lmbda_hysteresis` as space-separated lists, and `--folder_names` for per-step outputs.

Listing 3: Example usage

```
./paratoric -sim etc_hysteresis -Ns 1000 -Nth 2000 -Nbs 50 -Nr 500 -bet
    12.0 -muc 1 -Jc 1 -lmbdahys 0.2 0.2 0.2 0.2 0.2 0.2 0.2 -hhys 0.0 0.1
    0.2 0.3 0.2 0.1 0.0 -obs energy fredenhagen_marcu -bas x -lat square -L
    12 -bound periodic -dsp 1 -outdir ./runs/hys -fns step0 step1 step2
    step3 step4 step5 step6
```

3.2.4 etc_thermalization

Performs thermalization only (no production sampling).

Listing 4: Example usage

```
./paratoric -sim etc_thermalization -Ns 0 -Nth 5000 -Nbs 10 -Nr 500 -bet
    10.0 -muc 1 -Jc 1 -hc 0.3 -lmbdac 0.1 -hct 0.4 -lmbdact 0.2 -cth 1 -obs
    energy anyon_density -bas z -lat square -L 10 -bound open -dsp 1 -outdir
     ./runs/therm -snap=1
```

3.2.5 HDF5 structure

The output HDF5 has the structure `simulation/results/acc_ratio` for an array of the acceptance weights (only for thermalization), `simulation/results/observable_name/series` for the time series (if it was enabled), and `simulation/results/observable_name/{mean,mean_error,binder,binder_error,autocorrelation_time}`. For the regular sampling, `mean`, `mean_error`, `binder`, `binder_error` and `autocorrelation_time` contain doubles. For the hysteresis, they contain an array of values for the hysteresis parameters (in the order of `h_hysteresis` and `lmbda_hysteresis`).

3.3 C interface

The C interface enables users to use a ParaToric public header from within another C project or another programming language that supports a C-style interface. The C interface exposes a stable ABI. It mirrors the C++ interface. Include the public header `#include<paratoric/mcmc/extended_toric_code_c.h>`. All functions return `ptc_status_t`.

3.3.1 Build & Installation

The code can be compiled in exactly the same fashion as for the C++ interface.

CMake usage (installed package).

```
cmake_minimum_required(VERSION 3.23)
project(my_qmc_c C)
find_package(paratoric CONFIG REQUIRED)   # provides paratoric::core
add_executable(cdemo main.c)
target_link_libraries(cdemo PRIVATE paratoric::core)
```

CMake usage (as subdirectory).

```
add_subdirectory(deps/paratoric)
add_executable(cdemo main.c)
target_link_libraries(cdemo PRIVATE paratoric::core)
```

3.3.2 Status & error handling

Name	C type/values	Meaning
`ptc_status_t`	`{PTC_STATUS_OK=0, PTC_STATUS_INVALID_ARGUMENT=1, PTC_STATUS_RUNTIME_ERROR=2, PTC_STATUS_NO_MEMORY=3, PTC_STATUS_INTERNAL_ERROR=4}`	Return code of every API call.
`ptc_last_error()`	`const char*`	Thread–local error string. Valid until next call.

3.3.3 Opaque handle

Create and destroy the interface instance. Use `ptc_create(ptc_handle_t **out)` and `ptc_destroy(ptc_handle_t *h)`.

3.3.4 Configuration type

Top-level `ptc_config_t` aggregates four nested specs. Field names mirror the C++ `Config`.

Top-level configuration: `ptc_config_t`

Field	Type	Purpose
`sim`	`ptc_sim_spec_t`	Monte Carlo parameters.
`params`	`ptc_param_spec_t`	Hamiltonian parameters.
`lat`	`ptc_lat_spec_t`	Lattice parameters.
`out`	`ptc_out_spec_t`	Output paths and snapshot toggle.

Simulation specification (`config.sim`)

Field	Type	Meaning
`N_samples`	`int`	Number of snapshots.
`N_thermalization`	`int`	Thermalization steps.
`N_between_samples`	`int`	Thinning between snapshots.
`N_resamples`	`int`	Bootstrap resamples.
`custom_therm`	`bool`	Custom thermalization schedule.
`seed`	`int`	PRNG seed (0 = random).
`observables`	`const char* const*`	Array of observable names (nullable).
`N_observables`	`size_t`	Length of `observables`.

Parameter specification (`config.params`)

Field	Type	Meaning
`mu,h,J,lmbda`	`double`	Couplings (star, electric, plaquette, gauge).
`h_therm`	`double`	Thermalization value for `h` if `custom_therm=true`.
`lmbda_therm`	`double`	Thermalization value for `lmbda` if `custom_therm=true`.
`h_hys`	`const double*`	Hysteresis schedule for `h` (nullable).
`h_hys_len`	`size_t`	Length of `h_hys`. Must match `lmbda_hys_len`.
`lmbda_hys`	`const double*`	Hysteresis schedule for `lmbda` (nullable).
`lmbda_hys_len`	`size_t`	Length of `lmbda_hys`. Must match `h_hys_len`.

Lattice specification (`config.lat`)

Field	Type	Meaning / Valid values
`basis`	`char`	Spin basis: `'x'` or `'z'`.
`lattice_type`	`const char*`	E.g. `"triangular"`, `"square"`, …
`system_size`	`int`	Linear system size per dimension.
`beta`	`double`	Inverse temperature.
`boundaries`	`const char*`	`"periodic"` or `"open"`.
`default_spin`	`int`	Initial link spin: +1 or −1.

Output specification (`config.out`)

Field	Type	Meaning
`path_out`	`const char*`	Single output directory (nullable).
`paths_out`	`const char* const*`	Output directories for hysteresis steps (nullable).
`N_paths_out`	`size_t`	Length of `paths_out`. Must match `h_hys_len` and `lmbda_hys_len`.
`save_snapshots`	`bool`	Toggle snapshot dumping.

3.3.5 Return type

All outputs are owned by the caller. Call `ptc_result_destroy(&r)` to free and zero.

Field	C type	Meaning
series	ptc_series_t	Time series (real/complex) of all requested observables, thermalization is excluded (except for thermalization simulation). Outer index = observable, inner index = time point.
acc_ratio	ptc_dvec_t	MC acceptance ratios for each update (thermalization).
mean, mean_std	ptc_dvec_t	Bootstrap mean and standard error (order as in observables).
binder, binder_std	ptc_dvec_t	Binder ratios and standard error (order as in observables).
tau_int	ptc_dvec_t	Integrated autocorrelation time (order as in observables).
series_hys	ptc_series_blocks_t	Hysteresis time series. Outer index = hysteresis parameters (order as in h_hys,lmbda_hys), middle index = observables (order as in observables), inner vector = time series.
mean_hys, mean_std_hys, binder_hys, binder_std_hys, tau_int_hys	ptc_dmat_t	Outer index = hysteresis parameters (order as in h_hys,lmbda_hys), inner index = observables (order as in observables).

3.3.6 Procedures (mirror the C++ API)

All fill a ptc_result_t *out on success. Return PTC_STATUS_OK on success.

ptc_get_thermalization(ptc_handle_t *h, const ptc_config_t *cfg, ptc_result_t *out) Run thermalization only. **Required fields:** cfg->lat.{basis,lattice_type,system_size,beta,boundaries,default_spin}, cfg->params.{mu,h,J,lmbda}, cfg->sim.{N_thermalization,N_resamples,observables,N_observables,seed}, cfg->out.{path_out,save_snapshots}.

ptc_get_sample(ptc_handle_t *h, const ptc_config_t *cfg, ptc_result_t *out) Run a production measurement pass. **Required fields:** cfg->lat.{basis,lattice_type,system_size,beta,boundaries,default_spin}, cfg->params.{mu,h,J,lmbda,h_therm,lmbda_therm}, cfg->sim.{N_samples,N_thermalization,N_between_samples,N_resamples,custom_therm,observables,N_observables,seed}, cfg->out.{path_out,save_snapshots}.

ptc_get_hysteresis(ptc_handle_t *h, const ptc_config_t *cfg, ptc_result_t *out) Run a hysteresis sweep over h_hys and/or lmbda_hys. The last state of step i initializes step i+1. **Required fields:** cfg->lat.{basis,lattice_type,system_size,beta,boundaries,default_spin}, cfg->params.{mu,h_hys,h_hys_len,J,lmbda_hys,lmbda_hys_len}, cfg->sim.{N_samples,N_thermalization,N_between_samples,N_resamples,observables,N_observables,seed}, cfg->out.{paths_out,N_paths_out,save_snapshots}.

3.3.7 C usage example

Listing 5: C API - Minimal call

```c
#include <stdio.h>
#include <math.h>
#include <paratoric/mcmc/extended_toric_code_c.h>

int main(void) {
  ptc_handle_t* h = NULL;
  if (ptc_create(&h) != PTC_STATUS_OK) { puts("create failed"); return 1; }

  ptc_lat_spec_t lat = {
    .basis = 'z',
    .lattice_type = "square",
    .system_size = 16,
    .beta = 8.0,
    .boundaries = "periodic",
    .default_spin = 1
  };

  ptc_param_spec_t ps = {
    .mu = 1.0, .h = 0.2, .J = 1.0, .lmbda = 0.0,
    .h_therm = NAN, .lmbda_therm = NAN,
    .h_hys = NULL, .h_hys_len = 0,
    .lmbda_hys = NULL, .lmbda_hys_len = 0
  };

  const char* obs[] = {"energy","plaquette_z","anyon_count"};
  ptc_sim_spec_t sim = {
    .N_samples = 0, /* thermalization-only initially */
    .N_thermalization = 5000,
    .N_between_samples = 10,
    .N_resamples = 1000,
    .custom_therm = false,
    .seed = 12345,
    .observables = obs,
    .N_observables = sizeof(obs)/sizeof(obs[0])
  };

  ptc_out_spec_t outspec = {
    .path_out = "runs/sample",
    .paths_out = NULL, .N_paths_out = 0,
    .save_snapshots = false
  };

  ptc_config_t cfg = { .sim = sim, .params = ps, .lat = lat, .out = outspec
      };

  ptc_result_t warm = {0};
  ptc_status_t st = ptc_get_thermalization(h, &cfg, &warm);
  if (st != PTC_STATUS_OK) { puts(ptc_last_error()); ptc_destroy(h); return
      2; }
  ptc_result_destroy(&warm);

  cfg.sim.N_samples = 2000;
  ptc_result_t res = {0};
  st = ptc_get_sample(h, &cfg, &res);
  if (st != PTC_STATUS_OK) { puts(ptc_last_error()); ptc_destroy(h); return
      3; }
```

```
  // use res.mean, res.tau_int, ...
  ptc_result_destroy(&res);
  ptc_destroy(h);
  return 0;
}
```

Memory rules. You own all buffers in `ptc_result_t`. Call `ptc_result_destroy` once per successful call.

3.4 Python bindings

ParaToric exposes a compiled Python extension module `_paratoric` with a submodule `extended_toric_code`. The bindings convert C++ vectors into NumPy arrays and release the global interpreter lock (GIL) while running the C++ kernels.

3.4.1 Build & Installation

The core requires C++23, CMake ≥3.23, and Boost ≥1.87 (older Boost versions may work, but were not tested). The Python bindings require a Python installation with Numpy and PyBind11 (tested with version 3.0.1). Pybind11 is included as a git submodule (you need to pull it!). To compile the Python bindings, run:

```
cmake -S . -B build -DCMAKE_BUILD_TYPE=Release \
-DPARATORIC_ENABLE_NATIVE_OPT=ON -DPARATORIC_LINK_MPI=OFF \
-DPARATORIC_BUILD_TESTS=ON -DPARATORIC_BUILD_PYBIND=ON \
-DPython3_EXECUTABLE="$(which python)" -DPYBIND11_FINDPYTHON=ON \
-DPARATORIC_INSTALL_TO_SITE=ON -DPARATORIC_PIP_EDITABLE_INSTALL=ON
cmake --build build -jN
ctest --test-dir build -jN --output-on-failure
cmake --install build
```

Replace `N` with the number of cores to use, e.g. `-j4` for 4 cores.

- `-DCMAKE_BUILD_TYPE=Release`. Only set to `Debug` if you're a developer.
- `-DCMAKE_INSTALL_PREFIX`. By default, executables install to `${CMAKE_SOURCE_DIR}/${CMAKE_INSTALL_BINDIR}/`, headers to `${CMAKE_INSTALL_INCLUDEDIR}/paratoric`, and static libraries to `${CMAKE_SOURCE_DIR}/${CMAKE_INSTALL_LIBDIR}/`. The Python scripts expect `${CMAKE_SOURCE_DIR}/bin/`; this directory always contains the `paratoric` executable. To install into a custom directory, pass it via `-DCMAKE_INSTALL_PREFIX`, e.g. `-DCMAKE_INSTALL_PREFIX=/your/custom/directory/`.
- `-DPARATORIC_EXPORT_COMPILE_COMMANDS=ON`. Export `compile_commands.json` for tooling.
- `-DPARATORIC_LINK_MPI=OFF`. Link the core to MPI, required on some clusters. The core itself does not need MPI.
- `-DPARATORIC_ENABLE_NATIVE_OPT=OFF`. Turn on `-march=native` on GCC and Clang.

- `-DPARATORIC_ENABLE_AVX2=OFF`. Enable AVX2 (Haswell New Instructions). Requires a CPU which supports AVX2.
- `-DPARATORIC_BUILD_TESTS=PROJECT_IS_TOP_LEVEL`. Compile the tests (recommended).
- `-DPARATORIC_BUILD_PYBIND=OFF`. Compile Python bindings.
- `-DDPARATORIC_INSTALL_TO_SITE=OFF`. Install ParaToric Python module to site packages.
- `-DPARATORIC_PIP_EDITABLE_INSTALL=OFF`. Install ParaToric Python module via pip as an editable module.
- `-DPARATORIC_PIP_OFFLINE_INSTALL=OFF`. Turn on when installing to pip without internet access. Requires `NumPy` and `setuptools`.

3.4.2 Module layout

- `paratoric._paratoric`: compiled extension (PyBind11). Submodule: `extended_toric_code`.
- `paratoric.extended_toric_code`: convenient alias
- Running `python -m paratoric` enters the package entry point (`__main__.py`).

NumPy return formats

All time series with potentially complex values are returned as `complex128`. Real observables appear with zero imaginary part. Shapes are documented in the function references below.

API reference (`paratoric.extended_toric_code`)

`get_thermalization(...)` Run only the warmup and return per-snapshot observables and MC acceptance ratios. Internally converts `std::variant<complex<double>,double>` to `complex128` and `std::vector<double>` to `float64` arrays. The GIL is released while the C++ routine executes.

Parameter	Type / default	Meaning
`N_thermalization`	`int`	Warmup steps.
`N_resamples`	`int=1000`	Bootstrap resamples.
`observables`	`list[str]`	Names per snapshot.
`seed`	`int=0`	PRNG seed (0 ⇒ random).
`mu,h,J,lmbda`	`float`	Hamiltonian parameters.
`basis`	`{'x','z'}='x'`	Spin eigenbasis.
`lattice_type`	`str`	E.g. `"triangular"`, `"square"`, …
`system_size`	`int`	Linear size per dimension.
`beta`	`float`	Inverse temperature.
`boundaries`	`str="periodic"`	Boundary condition.
`default_spin`	`int=1`	Initial link spin (+1/-1).
`save_snapshots`	`bool=false`	Enable snapshot files.
`path_out`	`path|None=None`	Output directory (if saving).

Returns: `(series, acc_ratio)` `series`: `ndarray(complex128)` of shape `(n_obs, N_thermalization)`; `acc_ratio`: `ndarray(float64)` of shape `(N_thermalization,)`.

`get_sample(...)` Run thermalization and production sampling; return series and bootstrap statistics. Converts nested C++ containers to NumPy arrays and releases the GIL during computation.

Parameter	Type / default	Meaning
`N_samples`	`int`	Stored samples per observable.
`N_thermalization`	`int`	Warmup steps before sampling.
`N_between_samples`	`int`	Thinning between samples.
`N_resamples`	`int=1000`	Bootstrap resamples.
`custom_therm`	`bool=false`	Use `h_therm`, `lmbda_therm` during warmup.
`observables`	`list[str]`	Names per snapshot.
`seed`	`int=0`	PRNG seed (0 ⇒ random).
`mu,h,J,lmbda`	`float`	Hamiltonian parameters.
`h_therm,lmbda_therm`	`float=0`	Warmup parameters if custom therm.
`basis`	`{'x','z'}='x'`	Spin eigenbasis.
`lattice_type,system_size,beta`	`str,int,float`	Lattice and temperature.
`boundaries,default_spin`	`str,int=("periodic",1)`	BC and initial spin.
`save_snapshots,path_out`	`bool=False,path|None=None`	Optional I/O.

Returns: tuple of six arrays `series (complex128): (n_obs, N_samples)`; `mean`, `mean_std`, `binder`, `binder_std`, `tau_int (float64)`: each `(n_obs,)`.

`get_hysteresis(...)` Run a sweep where each step uses the previous state as its initial condition. Returns stacked arrays across steps; path handling validates per-step output directories when saving snapshots.

Parameter	Type / default	Meaning
`N_samples,N_thermalization,N_between_samples`	`int,int,int`	Cadence per step.
`N_resamples`	`int=1000`	Bootstrap resamples.
`observables`	`list[str]`	Names per snapshot.
`seed`	`int=0`	PRNG seed.
`mu,J`	`float`	Star and plaquette couplings.
`h_hys,lmbda_hys`	`list[float]`	Hysteresis values, length must match.
`basis`	`{'x','z'}='x'`	Spin asis.
`lattice_type,system_size,beta`	`str,int,float`	Lattice and temperature.
`boundaries,default_spin`	`str,int=("periodic",1)`	BC and initial spin.
`save_snapshots`	`bool=false`	Enable stepwise I/O.
`paths_out`	`list[path]|None=None`	Output path per step (size must match `h_hys` if saving).

Returns: tuple of six arrays `series3d (complex128): (n_steps, n_obs, N_samples)`; `mean2d`, `std2d`, `binder2d`, `binder_std2d`, `tau2d (float64)`: each `(n_steps, n_obs)`. The number of steps equals `len(h_hys)` (and `len(lmbda_hys)`).

Array dtypes and shapes (summary)

Function	Name / dtype	Shape
`get_thermalization`	`series` (`complex128`)	`(n_obs, N_thermalization)`
	`acc_ratio` (`float64`)	`(N_thermalization,)`
`get_sample`	`series` (`complex128`)	`(n_obs, N_samples)`
	`mean`, `mean_std`, `binder`, `binder_std`, `tau_int` (`float64`)	each `(n_obs,)`
`get_hysteresis`	`series3d` (`complex128`)	`(n_steps, n_obs, N_samples)`
	`mean2d`, `std2d`, `binder2d`, `binder_std2d`, `tau2d` (`float64`)	each `(n_steps, n_obs)`

Notes on performance

The bindings release the global interpreter lock (GIL) during heavy compute (`py::gil_scoped_release`), enabling multi-threaded C++ execution if the backend uses threads or when calling from multiprocessing workers. Conversions handle 1D/2D/3D containers and enforce consistent inner lengths before copying to NumPy.

3.4.3 Usage example

Listing 6: Importing and calling from Python

```
>>> import numpy as np
>>> from paratoric import extended_toric_code as etc
>>> series, acc_ratio = etc.get_thermalization(
... N_thermalization=2000, N_resamples=500,
... observables=["energy","plaquette_z","anyon_count"],
... seed=0, mu=1.0, h=0.2, J=1.0, lmbda=0.0,
... basis='z', lattice_type="square", system_size=16, beta=8.0,
... boundaries="periodic", default_spin=1,
... save_snapshots=False, path_out=None)
>>> series.shape, series.dtype
((3, 2000), dtype('complex128'))
>>> out = etc.get_sample(
... N_samples=1000, N_thermalization=5000, N_between_samples=10,
... N_resamples=1000, custom_therm=False,
... observables=["energy","plaquette_z"],
... seed=0, mu=1.0, h=0.2, h_therm=0.0,
... J=1.0, lmbda=0.0, lmbda_therm=0.0,
... basis='z', lattice_type="square", system_size=16, beta=8.0,
... boundaries="periodic", default_spin=1,
... save_snapshots=False, path_out=None)
>>> (series_s, mean, mean_std, binder, binder_std, tau_int) = out
```

3.5 Python command-line interface

ParaToric ships a Python command-line interface `/python/cli/paratoric.py` that orchestrates C++ backends, runs sweeps, and writes HDF5/XML outputs. It requires NumPy, matplotlib, and h5py.

3.5.1 Build & Installation

The command-line interface requires HDF5 ≥1.14.3 (older HDF5 versions may work, but were not tested). The core requires C++23, CMake ≥3.23, and Boost ≥1.87 (older Boost versions may work, but were not tested). To compile it, run:

```
cmake -S . -B build -DCMAKE_BUILD_TYPE=Release \
-DPARATORIC_ENABLE_NATIVE_OPT=ON -DPARATORIC_LINK_MPI=OFF \
-DPARATORIC_BUILD_TESTS=ON -DPARATORIC_BUILD_CLI=ON
cmake --build build -jN
ctest --test-dir build -jN --output-on-failure
cmake --install build
```

Replace `N` with the number of cores to use, e.g. `-j4` for 4 cores.

- `-DCMAKE_BUILD_TYPE=Release`. Only set to `Debug` if you're a developer.
- `-DCMAKE_INSTALL_PREFIX`. By default, executables install to `${CMAKE_SOURCE_DIR}/${CMAKE_INSTALL_BINDIR}/`, headers to `${CMAKE_INSTALL_INCLUDEDIR}/paratoric`, and static libraries to `${CMAKE_SOURCE_DIR}/${CMAKE_INSTALL_LIBDIR}/`. The Python scripts expect `${CMAKE_SOURCE_DIR}/bin/`; this directory always contains the `paratoric` executable. To install into a custom directory, pass it via `-DCMAKE_INSTALL_PREFIX`, e.g. `-DCMAKE_INSTALL_PREFIX=/your/custom/directory/`.
- `-DPARATORIC_EXPORT_COMPILE_COMMANDS=ON`. Export `compile_commands.json` for tooling.
- `-DPARATORIC_LINK_MPI=OFF`. Link the core to MPI, required on some clusters. The core itself does not need MPI.
- `-DPARATORIC_ENABLE_NATIVE_OPT=OFF`. Turn on `-march=native` on GCC and Clang.
- `-DPARATORIC_ENABLE_AVX2=OFF`. Enable AVX2 (Haswell New Instructions). Requires a CPU which supports AVX2.
- `-DPARATORIC_BUILD_TESTS=PROJECT_IS_TOP_LEVEL`. Compile the tests (recommended).
- `-DPARATORIC_BUILD_CLI=PROJECT_IS_TOP_LEVEL` Required for both the C++ and Python command line interface.

662 **General options**

Long flag	Short	Description
`--help`	`-h`	Show help and exit.
`--simulation`	`-sim`	Simulation type selector.
`--N_thermalization`	`-Nth`	Thermalization steps (proposed updates).
`--N_samples`	`-Ns`	Number of samples/snapshots.
`--N_between_steps`	`-Nbs`	Steps between successive samples (thinning).
`--N_resamples`	`-Nr`	Bootstrap resamples.
`--custom_therm`	`-cth`	Use thermalization values for h, λ (0 or 1).
`--observables`	`-obs`	Space-separated list, e.g. `fredenhagen_marcu percolation_probability energy`.
`--seed`	`-seed`	PRNG seed; 0 means random seed.
`--mu_constant`	`-muc`	Value of μ.
`--J_constant`	`-Jc`	Value of J.
`--h_constant`	`-hc`	Value of h.
`--h_constant_therm`	`-hct`	Thermalization value of h.
`--lmbda_constant`	`-lmbdac`	Value of λ.
`--lmbda_constant_therm`	`-lmbdact`	Thermalization value of λ.
`--output_directory`	`-outdir`	Output directory.
`--snapshots`	`-snap`	Save snapshots toggle (0/1).
`--full_time_series`	`-fts`	Save full time series toggle (0/1).
`--processes`	`-proc`	Logical CPU count for Python multiprocessing. 0 means all available cores. Negative numbers $-x$ mean use all cores minus x. Default is -4.

664 **Lattice-specific options**

Long flag	Short	Description
`--help`	`-h`	Show help and exit.
`--basis`	`-bas`	Spin basis: `x` or `z`.
`--lattice_type`	`-lat`	`square`, `cubic`, `triangular`, `honeycomb`, ...
`--system_size`	`-L`	Linear size; in 2D, 30 yields a 30×30 lattice (unit cells).
`--temperature`	`-T`	Temperature $T = 1/\beta > 0$.
`--boundaries`	`-bound`	`periodic` or `open`.
`--default_spin`	`-dsp`	Initial edge spin: `1` or `-1`.

666 The command line interface offers several sweep modes. All are embarrassingly parallel;
667 set `--processes` close to the number of steps when possible.

668 ### 3.5.2 T-sweep

669 Runs `T_steps` independent Markov chains for evenly spaced temperatures in [T_lower, T_upper]
670 and plots all requested observables.

Listing 7: Example usage

671

```
python3 ./python/cli/paratoric.py -sim etc_T_sweep -Ns 1000 -muc 1 -Nth
    2000 -Nbs 100 -Tl 0.5 -Tu 5 -Ts 30 -hc 0.1 -Jc 1 -lmbdac 0.1 -Nr 1000 -
    obs percolation_strength percolation_probability
    plaquette_percolation_probability largest_cluster string_number energy
    energy_h energy_mu energy_J energy_lmbda sigma_x sigma_z star_x
    plaquette_z staggered_imaginary_times delta anyon_count anyon_density
    fredenhagen_marcu sigma_x_susceptibility sigma_z_susceptibility -s 0 -
    bas x -lat square -L 4 -bound periodic -dsp 1 -outdir /path/to/out
```

Sweep-specific flags.

Long flag	Short	Description
`--simulation`	`-sim`	Use `etc_T_sweep`.
`--T_lower`	`-Tl`	Lower bound of T.
`--T_upper`	`-Tu`	Upper bound of T.
`--T_steps`	`-Ts`	Number of temperatures between bounds.

3.5.3 h-sweep

Runs `h_steps` independent chains in parallel for evenly spaced h in [h_lower, h_upper].

Listing 8: Example usage

```
python3 ./python/cli/paratoric.py -sim etc_h_sweep -Ns 1000 -muc 1 -Nth
    2000 -Nbs 100 -hl 0.1 -hu 0.5 -hs 8 -T 0.03 -Jc 1 -lmbdac 0.2 -Nr 1000 -
    obs percolation_strength percolation_probability
    plaquette_percolation_probability largest_cluster string_number energy
    energy_h energy_mu energy_J energy_lmbda sigma_x sigma_z star_x
    plaquette_z staggered_imaginary_times delta anyon_count anyon_density
    fredenhagen_marcu sigma_x_susceptibility sigma_z_susceptibility -s 0 -
    bas x -lat square -L 6 -bound periodic -dsp 1 -outdir /path/to/out
```

Sweep-specific flags.

Long flag	Short	Description
`--simulation`	`-sim`	Use `etc_h_sweep`.
`--h_lower`	`-hl`	Lower bound of h.
`--h_upper`	`-hu`	Upper bound of h.
`--h_steps`	`-hs`	Number of field steps between bounds.

3.5.4 λ-sweep

Runs `lmbda_steps` independent chains in parallel for evenly spaced λ in [λ_lower, λ_upper].

Listing 9: Example usage

```
python3 ./python/cli/paratoric.py -sim etc_lmbda_sweep -Ns 1000 -muc 1 -Nth
     2000 -Nbs 100 -lmbdal 0.01 -lmbdau 1.0 -lmbdas 15 -T 0.1 -hc 0.3 -Jc 1
    -Nr 1000 -obs percolation_strength percolation_probability
    plaquette_percolation_probability largest_cluster string_number energy
    energy_h energy_mu energy_J energy_lmbda sigma_x sigma_z star_x
    plaquette_z staggered_imaginary_times delta anyon_count anyon_density
    fredenhagen_marcu sigma_x_susceptibility sigma_z_susceptibility -s 0 -
    bas x -lat square -L 4 -bound periodic -dsp 1 -outdir /path/to/out
```

Sweep-specific flags.

Long flag	Short	Description
`--simulation`	`-sim`	Use `etc_lmbda_sweep`.
`--lmbda_lower`	`-lmbdal`	Lower bound of λ.
`--lmbda_upper`	`-lmbdau`	Upper bound of λ.
`--lmbda_steps`	`-lmbdas`	Number of field steps between bounds.

3.5.5 ∘-sweep

Runs `Theta_steps` independent chains in parallel along a circle in (λ, h) centered at `(lmbda_constant,h_constant)` with radius `radius`, for angles $\Theta \in$ [Θ_lower, Θ_upper] (angles measured anti-clockwise from the λ-axis).

Listing 10: Example usage

```
python3 ./python/cli/paratoric.py -sim etc_circle_sweep -Ns 1000 -muc 1 -
    Nth 2000 -Nbs 100 -lmbdac 0.4 -rad 0.3 -Thl 0 -Thu 3.141 -Ths 15 -T 0.1
    -hc 0.4 -Jc 1 -Nr 1000 -obs percolation_strength percolation_probability
     plaquette_percolation_probability largest_cluster string_number energy
    energy_h energy_mu energy_J energy_lmbda sigma_x sigma_z star_x
    plaquette_z staggered_imaginary_times delta anyon_count anyon_density
    fredenhagen_marcu sigma_x_susceptibility sigma_z_susceptibility -s 0 -
    bas x -lat square -L 4 -bound periodic -dsp 1 -outdir /path/to/out
```

Sweep-specific flags.

Long flag	Short	Description
`--simulation`	`-sim`	Use `etc_circle_sweep`.
`--lmbda_constant`	`-lmbdac`	Circle center in λ.
`--h_constant`	`-hc`	Circle center in h.
`--radius`	`-rad`	Circle radius.
`--Theta_lower`	`-Thl`	Lower bound of Θ.
`--Theta_upper`	`-Thu`	Upper bound of Θ.
`--Theta_steps`	`-Ths`	Number of angles between bounds.

3.5.6 Hysteresis-sweep

Use the hysteresis schedule specified in `hhys` and `lmbdahys`. This mode will run two Markov chains, one in the original parameter order specified in `hhys` and `lmbdahys`, and one with a reversed parameter order, i.e., it calculates both branches of the hysteresis loop.

Listing 11: Example usage

```
python3 ./python/cli/paratoric.py -sim etc_hysteresis -Nbs 5000 -Ns 10000 -
    muc 1 -Nth 20000 -hhys 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 -T
    0.1 -Jc -1 -lmbdahys 0.5 0.525 0.55 0.575 0.6 0.625 0.65 0.675 0.7 0.725
     0.75 -Nr 1000 -obs plaquette_percolation_probability
    percolation_strength percolation_probability largest_cluster
    string_number energy energy_h energy_mu energy_J energy_lmbda sigma_x
    sigma_z star_x plaquette_z staggered_imaginary_times delta anyon_count
    anyon_density fredenhagen_marcu sigma_x_susceptibility
    sigma_z_susceptibility -s 0 -bas z -lat square -L 4 -bound periodic -dsp
     1 -outdir /path/to/out
```

Sweep-specific flags.

Long flag	Short	Description
`--simulation`	`-sim`	Use `etc_hysteresis`.
`--lmbda_hysteresis`	`-lmbdahys`	Hysteresis schedule for λ. Length must match `hhys`.
`--h_hysteresis`	`-hhys`	Hysteresis schedule for h. Length must match `lmbdahys`.

3.5.7 Thermalization

Runs `repetitions` independent chains in parallel and reports observables and MC acceptance ratios every step, averaged over chains.

Listing 12: Example usage

```
python3 ./python/cli/paratoric.py -sim etc_thermalization -muc 1 -Nth 2000
    -reps 10 -lmbdac 2 -T 0.1 -hc 0.3 -Jc 1 -Nr 1000 -obs
    percolation_strength percolation_probability
    plaquette_percolation_probability largest_cluster string_number energy
    energy_h energy_mu energy_J energy_lmbda sigma_x sigma_z star_x
    plaquette_z staggered_imaginary_times delta anyon_count anyon_density
    fredenhagen_marcu sigma_x_susceptibility sigma_z_susceptibility -s 0 -
    bas x -lat square -L 4 -bound periodic -dsp 1 -outdir /path/to/out
```

Sweep-specific flags.

Long flag	Short	Description
`--simulation`	`-sim`	Use `etc_thermalization`.
`--repetitions`	`-reps`	Number of Markov chains to average.

4 Using ParaToric

4.1 Monte Carlo Updates

There is no need for the user to explicitly call specific updates or interact with internal C++ classes when using the documented interfaces. Internally, we use all five updates described in the original algorithm by Wu, Deng, and Prokof'ev [5]. These must furthermore be supplemented with the following two updates: Because for high temperatures and for zero off-diagonal fields the spin at imaginary time $0 = \beta$ cannot be flipped, we allow for flipping the spin on the entire imaginary axis on one bond or on a plaquette (star) in the $\hat{\sigma}^x$-basis ($\hat{\sigma}^z$-basis).

These updates only change the energy terms diagonal in the given basis and is trivial when caching the total integrated diagonal energy (the update locally flips the sign of the total integrated potential energy). Another advantage is that integrated autocorrelation times for observables diagonal in the given basis improve even in regimes that were previously accessible. All seven updates are equally likely to be proposed, and we use a 64-bit Mersenne-Twister for pseudorandom numbers [12] with the ability to externally set the seed. Some updates have early exits for input parameters for which they will always be rejected.

4.2 Monte Carlo Diagnostics

There are two compilation modes, `Release` and `Debug`. In production runs, one should always use the `Release` mode; however, it still gives the user enough information to diagnose sampling problems without severe performance impacts.

4.2.1 Thermalization mode

We provide thermalization routines which should be used before production runs to ensure proper thermalization (also known as burn-in). Thermalization times can vary drastically between different observables and initial conditions. We provide an example of sufficient and insufficient thermalization in Fig. 2. We recommend using the provided Python command-line interface, which will also plot the thermalization of all measured observables for the user.

In thermalization runs, we also return the Monte Carlo acceptance ratio of every update. This can also be used to diagnose freezing (in the measurement phase, use the integrated autocorrelation time instead), e.g., when the acceptance ratio is always identical and/or very low.

In case one suspects experiencing a serious sampling problem, we recommend recompiling the project in the `Debug` mode, which provides a wide array of runtime debug information about the proposed steps, acceptance ratios, and intermediate results. However, do not use the `Debug` mode in production runs, as it negatively impacts performance.

4.2.2 Integrated autocorrelation time

When measuring observables, we first thermalize the system with `N_thermalization` steps, then measure `N_samples` times with `N_between_samples` steps between measurements. The normalized autocorrelation function $\rho_O(k)$ of an observable O_k (observable O measured at time k) applied to a discrete time series of length N is given by:

$$\rho_O(k) = \frac{C(k)}{C(0)}, \qquad C(k) = \frac{1}{N-k}\sum_{i=0}^{N-k-1}(O_i - \bar{O})(O_{i+k} - \bar{O}), \qquad \bar{O} = \frac{1}{N}\sum_{i=0}^{N-1} O_i. \tag{7}$$

It is a statistical measure of the correlations between measurements of observable O at times i and $i+k$.[2] We define the *integrated autocorrelation time*

$$\tau_{\text{int}}^{O} = \frac{1}{2} + \sum_{k\geq 1}\rho_O(k). \tag{8}$$

Large τ_{int} are generally undesirable since they increase error bars and can lead to bias. In case of perfect sampling, we would have $\rho_O(0) = 1$ and $\rho_O(k) = 0\ \forall k \geq 1$, i.e., each measurement is only correlated with itself but not with other measurements and $\tau_{\text{int}} = 1/2$. In practice, this is usually not feasible, and we have to work with a finite autocorrelation time $\tau_{\text{int}} > 1/2$.

When using ParaToric, we strongly recommend monitoring τ_{int} for all simulations and all observables. It is automatically calculated for every observable based on the full time series. As a rule of thumb, the autocorrelation is fine as long as $\tau_{\text{int}} \ll N_{\text{samples}}$, otherwise it leads to bias and seriously underestimated error bars. In the vicinity of phase transitions, τ_{int} dramatically increases ("critical slowing down") [13]. Importantly, τ_{int} can differ vastly between different observables! If the autocorrelation is too high, increase the number of steps between samples. In more complicated cases, one may need to adapt the update proposal distributions and/or the updates themselves as a last resort.

[2] In ParaToric, the autocorrelation function is calculated efficiently using fast Fourier transforms.

It is also important to mention that ParaToric only computes a statistical *estimate* of τ_{int}. Many factors determine how accurate this estimate is, and crucially, the system needs to be properly *thermalized*. In principle, one can use τ_{int} in the way it is computed above directly for calculating error bars of correlated time series; however, ParaToric uses a more robust bootstrapping approach.

4.2.3 Error bars

ParaToric applies the stationary bootstrap [14–16] for all error bars, thus capturing autocorrelation effects. Large τ_{int} will lead to worse error bars. The only parameter that the user can change is the number of bootstrap resamples `N_between_steps`. The default is 1000, which is enough in most cases. Note that a too low value of `N_between_steps` increases the relative computational cost of performing the measurements, which may negatively affect the code efficiency at no statistical gain. If the error bars are too large, either the number of samples is too low (in which case one should increase `N_samples`) or the autocorrelation is too large (in which case one could additionally increase `N_between_samples`).

4.3 Tips & tricks

4.3.1 Probing ground state physics

The algorithm implemented by ParaToric fundamentally requires a finite temperature $T > 0$. However, in QMC simulations, there is always a finite-size energy gap (the difference between the energy of the ground state and the first excited state). Additionally, some phases like the topological ground state of the toric code have a physical bulk gap (even at $L \to \infty$). As long as the temperature is well below the total gap, we are exponentially close to the ground state. Usually, a temperature $T \sim 1/L$ suffices for the toric code although other situations may arise.

4.3.2 Probing first-order transitions

ParaToric provides functionalities to probe weak and strong first-order phase transitions. The hysteresis mode can be used to probe hysteresis loops in the vicinity of strong first-order phase transitions, by repeating the simulation two times and mirroring the order of the parameters in `h_hysteresis` and `lmbda_hysteresis`. Weak first-order transitions can be detected by plotting a time series histogram of an observable (it exhibits a double-peak structure). Both approaches have been used in the context of the toric code [11].

4.3.3 Choosing the basis

Sometimes, one can work in both the $\hat{\sigma}^x$ and the $\hat{\sigma}^z$-basis. The performance can vary drastically! Generally, the $\hat{\sigma}^x$-basis is more efficient for $h/J > \lambda/\mu$ and vice versa.

4.3.4 Choosing `N_thermalization`

Based on our experience, we can say that $N_{\text{thermalization}} = 500L^d/T$ is a sensible choice for small fields, where d is the dimensionality of the system. Nevertheless, one should make use of the provided tools to benchmark thermalization, see Sec. 4.2, and rather err on the side of safety.

4.3.5 Choosing `N_samples`

Neglecting autocorrelation effects, the error of an observable ΔO scales as $\Delta O \sim 1/\sqrt{N_{\text{samples}}}$. More samples are, in principle, always better and lead to lower error

bars. Smoothness of a curve of statistical results also requires that error bars be small in relation to the parameter grid size. If one increases the parameter resolution (e.g., in the field h), then one typically also increases `N_samples`.

4.3.6 Choosing `N_between_samples`

The optimal choice for `N_samples` is the integrated autocorrelation time. A good guess of the autocorrelation time based on previous simulations for smaller system sizes or nearby parameter points can result in substantial computational saving costs in production runs for large system sizes. Near continuous phase transitions, the integrated autocorrelation time has an additional dependence $\tau_{\text{int}} \sim L^z$, where z is the dynamical exponent of the universality class of the transition. For a 2D system in the vicinity of a continuous phase transition, a sensible scaling for `N_between_samples` could be $\mathcal{O}(L^2 \times \beta \times L^z)$ ($\mathcal{O}(L^2)$ links, each has off-diagonal spin flips $\mathcal{O}(\beta)$).

4.3.7 Choosing `N_resamples`

As with `N_samples`, more is better (but also more costly). Usually `N_resamples` ≈ 1000 is a sensible choice.

4.3.8 Extracting snapshots

When the option `save_snapshots` is enabled, ParaToric will write the snapshots into the directory specified in `path_out` (or in the paths `paths_out` for hysteresis sweeps). The snapshots are saved in the GraphML format (XML-based), which is supported by many major graph libraries. One snapshot will be saved for every measurement of observables, i.e., `N_samples` snapshots in total. All snapshots are written into a single file to save disk space and simultaneously offer a structured, self-documenting format. Every edge stores a list of spins. The first spin belongs to the first snapshot, the second one to the second snapshot, and so on. There are no special requirements for disks or memory bandwidth; the snapshots are kept in RAM and are only written to disk after the simulation has finished.

4.3.9 Adding new observables/lattices/updates

After adding features to the code, *always* benchmark them using analytical results, other numerical methods (exact diagonalization, tensor networks, ...), and unit tests. We advise using a fixed seed during development, e.g., when checking whether two methods produce the exact same result. The code has some built-in features to check self-consistency, e.g., at the end of each simulation, the code checks whether the cached total energy is numerically close to the total energy calculated from scratch. Do not turn off these features, as they will point you toward bugs!

4.4 Benchmarks

4.4.1 Thermalization

In Fig. 2 (which we already discussed before and repeat here for completeness) we plot the gauge field energy $\propto \lambda$ for two systems: one is sufficiently thermalized, the other one is not. The plots are a direct output of the Python command-line interface. Always make sure that the system is well thermalized.

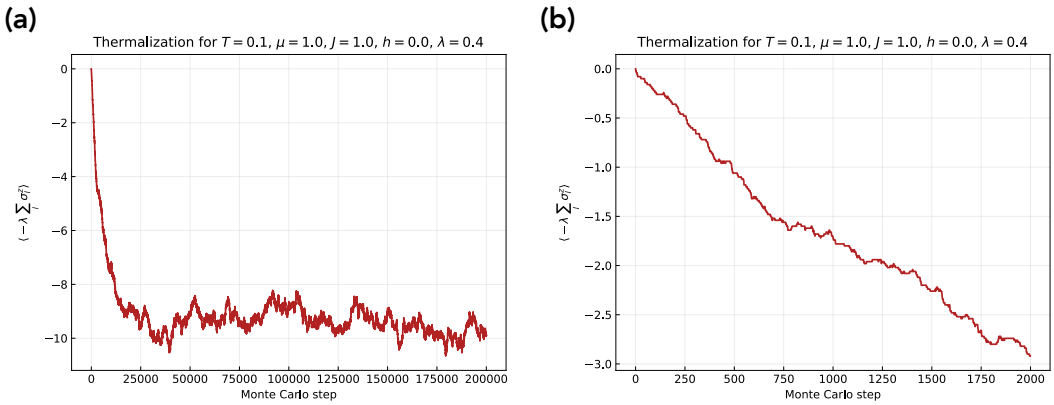

Figure 2: **Good and bad thermalization** plots produced by the Python command-line interface. We show the gauge field energy $\propto \lambda$. (a) The system is well thermalized; after its initial drop, the energy fluctuates around the expectation value. (b) The system is not yet thermalized; the floating average is still decreasing.

4.4.2 Integrated autocorrelation time

Here, we demonstrate how the integrated autocorrelation time τ_{int} grows with decreasing `N_between_samples`. We use the following setup:

Listing 13: `N_between_samples` benchmarking setup

```
>>> import numpy as np
>>> from paratoric import extended_toric_code as etc
>>> series, mean, std, binder, binder_std, tau_int = extended_toric_code.
    get_sample(N_samples=100000, N_thermalization=10000, N_between_samples
    =1, N_resamples=1000, custom_therm=False, observables=["energy"], seed
    =0, mu=1.0, h=0.0, h_therm=0.0, J=1.0, lmbda=0, lmbda_therm=0.0, basis='
    x', lattice_type="square", system_size=4, beta=10, boundaries="periodic"
    , default_spin=1, save_snapshots=False)
```

We only run the simulation once per `N_between_samples`. The results are:

`N_between_samples`	**1**	**10**	**100**	**500**	**1000**
τ_{int} (energy)	1895	141.4	19.7	3.24	1.64

For very small `N_between_samples`, τ_{int} is very high: in cases where the update is rejected, the configuration is identical to the one measured before! A choice of `N_between_samples` between 500 and 1000 would be a good tradeoff between τ_{int} and runtime for this example. Increasing `N_between_samples` to well over 1000 would be a waste of CPU time.

4.4.3 Run-time

We benchmark the run-time for two realistic parameter sets on the square lattice and varying system size. The first setup simulates the toric code without fields:[3]

Listing 14: L benchmarking setup 1

```
>>> import numpy as np
>>> from paratoric import extended_toric_code as etc
```

[3] All tests were run on a laptop; some conditions, like the CPU temperature, were not identical for all simulations. The benchmarks are therefore only an approximation.

```
>>> L=20
>>> series, mean, std, binder, binder_std, tau_int = extended_toric_code.
    get_sample(N_samples=10000, N_thermalization=500*L*L*L,
    N_between_samples=8*L*L*L, N_resamples=1000, custom_therm=False,
    observables=["energy", "sigma_x", "sigma_z"], seed=0, mu=1.0, h=0.0,
    h_therm=0.0, J=1.0, lmbda=0, lmbda_therm=0.0, basis='x', lattice_type="
    square", system_size=L, beta=L, boundaries="periodic", default_spin=1,
    save_snapshots=False)
```

We only run one test per system size. The results are:

L	**4**	**8**	**12**	**16**	**20**
Runtime (s)	3.1	21.3	75	197	379

From our experience, for large L the update complexity is approximately $\mathcal{O}(L^3 \log \beta)$, owing to the chosen cubic dependency of `N_thermalization` and `N_between_samples` and a $\mathcal{O}(\log \beta)$ dependence of operations on the imaginary time axis, see β benchmark below. The system size itself does not impact the performance, as the interactions are local. On computing clusters, we have realized system sizes of up to $L = 80$ for the square lattice; this number will only increase in the future as CPUs get faster.

The second setup simulates the toric code with fields in both $\hat{\sigma}^x$ and $\hat{\sigma}^z$-direction:

Listing 15: L benchmarking setup 2

```
>>> import numpy as np
>>> from paratoric import extended_toric_code as etc
>>> L=20
>>> series, mean, std, binder, binder_std, tau_int = extended_toric_code.
    get_sample(N_samples=10000, N_thermalization=500*L*L*L,
    N_between_samples=8*L*L*L, N_resamples=1000, custom_therm=False,
    observables=["energy", "sigma_x", "sigma_z"], seed=0, mu=1.0, h=0.2, J
    =1.0, lmbda=0.2, basis='x', lattice_type="square", system_size=L, beta=L
    , boundaries="periodic", default_spin=1, save_snapshots=False)
```

We only run one test per system size. The results are:

L	**4**	**8**	**12**	**16**	**20**
Runtime (s)	3.9	34.1	133	323	689

We also test the run-time dependence of the inverse temperature β, with the following setup:

Listing 16: β benchmarking setup

```
>>> import numpy as np
>>> from paratoric import extended_toric_code as etc
>>> series, mean, std, binder, binder_std, tau_int = extended_toric_code.
    get_sample(N_samples=10000, N_thermalization=20000, N_between_samples
    =2000, N_resamples=1000, custom_therm=False, observables=["energy", "
    sigma_x", "sigma_z"], seed=0, mu=1.0, h=0.2, J=1.0, lmbda=0.2, basis='x'
    , lattice_type="square", system_size=10, beta=20, boundaries="periodic",
     default_spin=1, save_snapshots=False)
```

We only run one test per β. The results are:

β	**4**	**8**	**12**	**16**	**20**
Runtime (s)	14.9	17.2	19.1	20.0	22.1

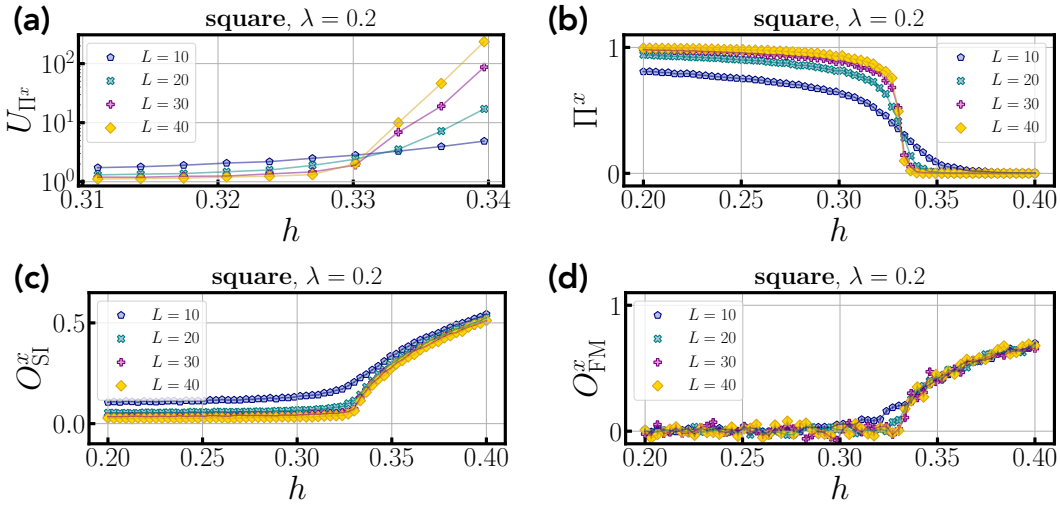

Figure 3: **Topological phase transition** of the extended toric code (1) on the square lattice. The critical field is known and located at $h_c(\lambda = 0.2) \approx 0.33$ [5, 9]. Our results agree with the value published in the literature, within error bars. (a) The percolation probability Binder ratio U_{Π^x} [9] features a crossing point around $h = 0.33$. (b) The percolation probability Π^x is non-zero in the topological phase and zero in the trivial phase. The transition gets sharper with increasing system size. (c) The staggered imaginary time order parameter O^x_{SI} is zero in the topological phase and non-zero in the trivial phase. (d) The Fredenhagen-Marcu order parameter O^x_{FM} is zero in the topological phase and non-zero in the trivial phase. The loop length grows with $\mathcal{O}(L)$. Compared to the other order parameters, it is very noisy because it is a multi-body correlator and on top of that a division of two exponentially small numbers.

This benchmark illustrates an appealing feature of our implementation: there is almost no slowing down when increasing β implying that very low temperatures are within reach with ParaToric. This paradoxical result (because the number n of off-diagonal star/plaquette and magnetic field operators must physically scale linearly in β) is explained by the fact that most searches within the imaginary time axis scale as $\mathcal{O}(\log n)$ by making use of binary searches.

4.4.4 Topological phase transition

We probe the well-known topological phase transition in the ground state of the extended toric code (1) on the square lattice, where we have a gapped $\mathbb{Z}_2$ quantum spin liquid for small fields h, λ and a topologically trivial phase for high fields. We set $J = \mu = 1$, $\lambda = 0.2$ and sweep h over the known critical value $h_c(\lambda = 0.2) \approx 0.33$ [5, 9] for $L \in \{10, 20, 30, 40\}$ in the $\hat{\sigma}^x$-basis. The temperature is set to $T = 1/L$ to capture ground state physics. We take 30000 snapshots, with $8L^3$ steps in between snapshots and $500L^3$ thermalization steps.[4]. We confirm that the system is well thermalized and that all integrated autocorrelation times are below 10, i.e., the produced snapshots can safely be considered identically and independently distributed. We show the percolation probability, the Fredenhagen-Marcu string order parameter, and the staggered imaginary time order parameter in Fig. 3. All of them reproduce the known phase boundary.

[4] If we were interested in quantities like critical exponents and we need to go very close to the critical field, we should take into account the dynamical exponent z ($\tau_{\mathrm{int}} \sim L^z$) in the number of steps between snapshots to account for critical slowing down. Now the error bars are just larger near the critical field.

5 Conclusion & Outlook

We have presented ParaToric, a continuous-time quantum Monte Carlo solver for the toric code in a parallel field. ParaToric builds on the existing work by Wu, Deng, and Prokof'ev [5] and is also applicable to high temperature and low off-diagonal couplings.

ParaToric can store snapshots, which makes it ideally suited to generate training/benchmarking data for applications in other fields, such as lattice gauge theories, cold atom or other quantum simulators, quantum spin liquids, artificial intelligence, and quantum error correction. We believe it also serves a pedagogical purpose. Another strength of ParaToric is its interoperability with other programming languages. The C interface is compatible with virtually all programming languages, thus ParaToric can be seamlessly integrated into other projects.

ParaToric comes with an MIT license. For future release of ParaToric we plan extensions along the following lines:

- Additional lattices such as the kagome and the ruby lattice. Given the underlying graph structure used in ParaToric, such extensions are straightforward.

- Additional observables: we think here of, for instance, the finite temperature extension of the fidelity susceptibility to diagnose the phase transitions in the absence of a local order parameter. It would be worthwhile to have additional off-diagonal observables such as the off-diagonal Fredenhagen-Marcu string operators, or correlation functions between off-diagonal operators in space and or time. Measurements of the Renyi entropy are also high on the to-do list. The latter two classes require however major changes to the code, and testing.

- Additional interaction types. There are many classes of models in which topological order may be emergent instead of explicit as in the toric code. Such models typically have additional interactions than the ones covered in ParaToric, such as longer-range Ising interactions, and miss some others (typically the plaquette type interactions, and sometimes even the star terms). It is in general an open problem how to efficiently simulate such models at the lowest temperatures (even for sign-free models). Extending ParaToric to dealing with other types of interactions can thus serve as an additional tool for benchmarking purposes and algorithmic exploration.

Acknowledgements

The authors acknowledge fruitful discussions with A. Bohrdt, G. De Paciani, G. Dünnweber, F. Grusdt, L. Homeier, and N. V. Prokof'ev.

Author contributions SML did the main coding and planning work with input from LP. All authors contributed to the writing of the manuscript.

Funding information This research was funded by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program – ERC Starting Grant SimUcQuam (Grant Agreement No. 948141), and by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – EXC-2111 – project number 390814868.

References

[1] A. Kitaev, *Fault-tolerant quantum computation by anyons*, Annals of Physics **303**(1), 2 (2003), doi:https://doi.org/10.1016/S0003-4916(02)00018-0.

[2] E. Dennis, A. Kitaev, A. Landahl and J. Preskill, *Topological quantum memory*, Journal of Mathematical Physics **43**(9), 4452 (2002), doi:10.1063/1.1499754, https://pubs.aip.org/aip/jmp/article-pdf/43/9/4452/19183135/4452_1_online.pdf.

[3] A. G. Fowler, M. Mariantoni, J. M. Martinis and A. N. Cleland, *Surface codes: Towards practical large-scale quantum computation*, Phys. Rev. A **86**, 032324 (2012), doi:10.1103/PhysRevA.86.032324.

[4] J. B. Kogut, *An introduction to lattice gauge theory and spin systems*, Rev. Mod. Phys. **51**, 659 (1979), doi:10.1103/RevModPhys.51.659.

[5] F. Wu, Y. Deng and N. Prokof'ev, *Phase diagram of the toric code model in a parallel magnetic field*, Phys. Rev. B **85**, 195104 (2012), doi:10.1103/PhysRevB.85.195104.

[6] K. Fredenhagen and M. Marcu, *Charged states in z2 gauge theories*, Communications in Mathematical Physics **92**(1), 81 (1983), doi:10.1007/BF01206315.

[7] K. Fredenhagen and M. Marcu, *Confinement criterion for qcd with dynamical quarks*, Phys. Rev. Lett. **56**, 223 (1986), doi:10.1103/PhysRevLett.56.223.

[8] K. Fredenhagen and M. Marcu, *Dual interpretation of order parameters for lattice gauge theories with matter fields*, Nuclear Physics B - Proceedings Supplements **4**, 352 (1988), doi:https://doi.org/10.1016/0920-5632(88)90124-7.

[9] S. M. Linsel, A. Bohrdt, L. Homeier, L. Pollet and F. Grusdt, *Percolation as a confinement order parameter in $z2$ lattice gauge theories*, Phys. Rev. B **110**, L241101 (2024), doi:10.1103/PhysRevB.110.L241101.

[10] G. Dünnweber, S. M. Linsel, A. Bohrdt and F. Grusdt, *Percolation renormalization group analysis of confinement in $z2$ lattice gauge theories*, Phys. Rev. B **111**, 024314 (2025), doi:10.1103/PhysRevB.111.024314.

[11] S. M. Linsel, L. Pollet and F. Grusdt, *Independent e- and m-anyon confinement in the parallel field toric code on non-square lattices*, doi:10.48550/arXiv.2504.03512 (2025).

[12] M. Matsumoto and T. Nishimura, *Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator*, ACM Trans. Model. Comput. Simul. **8**, 3 (1998).

[13] U. Wolff, *Critical slowing down*, Nuclear Physics B - Proceedings Supplements **17**, 93 (1990), doi:https://doi.org/10.1016/0920-5632(90)90224-I.

[14] D. N. Politis and J. P. Romano, *The stationary bootstrap*, Journal of the American Statistical Association **89**(428), 1303 (1994), doi:10.1080/01621459.1994.10476870.

[15] D. N. Politis and H. White, *Automatic block-length selection for the dependent bootstrap*, Econometric Reviews **23**(1), 53 (2004), doi:10.1081/ETC-120028836.

[16] A. Patton, D. N. Politis and H. White, *Correction to "automatic block-length selection for the dependent bootstrap" by d. politis and h. white*, Econometric Reviews **28**(4), 372 (2009), doi:10.1080/07474930802459016.