

qanneal

The Complete Manual — Theory, Algorithms & API

Research-grade Ising / QUBO optimisation on classical hardware

Version 0.6.1

Simulated Annealing • Simulated Quantum Annealing
Quantum Parallel Tempering • Continuous-Time PIMC
and the Optimal Adaptive $J_{\perp}$ Schedule

This manual is self-contained: every algorithm implemented in qanneal is derived from first principles — the Metropolis rule from detailed balance, the Suzuki–Trotter path integral from the quantum partition function, the replica-exchange acceptance rule from the joint Gibbs measure, and the locally-adiabatic $J_{\perp}$ schedule from the Roland–Cerf condition — alongside the complete Python and C++ API, worked examples, tuning guidance and HPC deployment recipes. No external references are required to use the library.

July 5, 2026

Apache License 2.0

Contents

I	Foundations	1
1	Introduction	2
1.1	What is qanneal?	2
1.2	Design philosophy	2
1.3	Architecture at a glance	3
1.4	A sixty-second tour	3
1.5	How to read this manual	4
2	Mathematical Foundations: Ising and QUBO	5
2.1	The Ising model	5
2.1.1	Sign conventions	5
2.1.2	Frustration — why the problem is hard	6
2.2	The QUBO model	6
2.3	Exact equivalence of QUBO and Ising	6
2.4	Encoding classic problems	7
2.4.1	Number partitioning	7
2.4.2	MaxCut	8
2.4.3	Constrained problems via penalties: maximum-weight independent set	8
2.5	Model classes in qanneal	8
3	Statistical Mechanics Background	10
3.1	The Boltzmann distribution	10
3.2	Markov chains and stationarity	10
3.2.1	Detailed balance	11
3.3	The Metropolis rule, derived	11
3.3.1	Sweeps, ergodicity, and shuffled visit order	12
3.4	From sampling to annealing	12
II	The Annealing Engines	13
4	Simulated Annealing (sa)	14
4.1	The algorithm	14

4.2	Schedule design	14
4.2.1	Fixed linear schedule	14
4.2.2	Problem-adaptive tuned schedules (recommended)	15
4.3	Independent replicas: ReplicaAnnealer	15
4.4	Classical parallel tempering	16
4.4.1	The swap rule, derived	16
4.4.2	Usage	16
4.5	When to choose sa	17
5	Simulated Quantum Annealing (sqa)	18
5.1	Quantum annealing in one page	18
5.2	The Suzuki–Trotter mapping, in full	19
5.2.1	Step 1: split the exponential	19
5.2.2	Step 2: insert complete sets of classical states	19
5.2.3	Step 3: the single-spin transfer matrix	20
5.2.4	Step 4: the effective classical action	20
5.2.5	Trotter error and choosing M	21
5.3	Monte Carlo moves on the Trotter system	21
5.3.1	Single-spin (slice) moves	21
5.3.2	Worldline moves	22
5.3.3	Swendsen–Wang cluster moves along imaginary time	22
5.4	The annealing loop and replicas	23
5.5	Usage	23
5.5.1	One-call interface	23
5.5.2	Explicit schedules	23
5.5.3	Problem-adaptive tuned SQA schedules	24
5.6	Parameter summary	24
6	Quantum Parallel Tempering (sqapt)	25
6.1	The (β, Γ) ladder	25
6.2	The quantum swap rule, derived	25
6.3	The algorithm	26
6.4	Usage	26
6.4.1	One call	26
6.4.2	Reading the diagnostics	27
6.5	Choosing sqapt	27
7	Continuous-Time PIMC (ctpimc)	28
7.1	From $M \rightarrow \infty$ to worldlines	28

7.2	Cluster updates on worldlines	28
7.3	General mode	29
7.4	Lattice mode	30
7.5	CT-PIMC versus SQA	30
III	The Optimal Adaptive Schedule	31
8	The Optimal Adaptive $J_{\perp}$ Schedule	32
8.1	Local adiabaticity: the Roland–Cerf condition	32
8.1.1	From the gap to a measurable quantity	32
8.2	Budget calibration: fixing the scale of $\tilde{\epsilon}$	33
8.2.1	The pilot pre-pass	34
8.2.2	The inverse-CDF trajectory	34
8.2.3	Fairness of the step budget	35
8.3	Estimating χ_B without overhead	35
8.3.1	Incremental bond tracking	35
8.3.2	Pooled variance across sweeps and replicas	35
8.3.3	Single-replica guard and the χ_B floor	35
8.4	Endpoints, temperature, and the two-phase SQA protocol	36
8.4.1	Choosing J_0 and J_{end}	36
8.4.2	Two-phase protocol for SQA	37
8.4.3	SQAPT under a shared $J_{\perp}$	37
8.5	Usage	37
8.5.1	High-level	37
8.5.2	Low-level, with full diagnostics	38
8.6	Diagnosing a run	38
8.7	Attribution: schedule effect vs. cluster moves	39
IV	Using the Library	40
9	Installation and Building	41
9.1	Requirements	41
9.2	Installing the Python package	41
9.3	Building the C++ library directly	42
9.4	Platform notes	42
10	Python API Reference	43
10.1	The high-level solver: <code>solve()</code>	43

10.1.1	Accepted problem types	43
10.1.2	Common parameters (all methods)	44
10.1.3	SQA / SQAPT parameters	44
10.1.4	SQAPT-only parameters	44
10.1.5	CT-PIMC-only parameters	44
10.1.6	Optimal-schedule parameters (schedule_type="optimal")	44
10.1.7	SolveResult	45
10.2	Model classes	45
10.2.1	DenseIsing	45
10.2.2	SparseIsing and SparseEdge	46
10.2.3	QUBO	46
10.3	Schedule classes and helpers	46
10.3.1	AnnealSchedule	46
10.3.2	SQASchedule	46
10.3.3	Fixed helpers	46
10.3.4	Tuned helpers (recommended)	47
10.3.5	Optimal-schedule helpers	47
10.4	Annealer classes	47
10.4.1	Annealer	47
10.4.2	ReplicaAnnealer	48
10.4.3	ParallelTemperingAnnealer	48
10.4.4	SQAAnealer	48
10.4.5	SQAParallelTemperingAnnealer	48
10.4.6	CTPIMCAnealer	49
10.5	Result classes	49
10.5.1	Adaptive-schedule fields on SQAResult / SQAParallelTemperingResult	49
10.6	Observer classes	50
10.7	Utility functions	50
11	Worked Examples	51
11.1	A three-variable QUBO, end to end	51
11.2	Number partitioning	51
11.3	MaxCut on a random graph	52
11.4	Standard vs. optimal schedule	52
11.5	Inspecting the adaptive trajectory (low level)	53
11.6	Observers: watching an anneal converge	54
11.7	A large sparse instance	54
11.8	dimod interoperability	55

11.9 Recommended workflow for a new problem	55
12 Parallelism and HPC Deployment	56
12.1 OpenMP: inside a run	56
12.2 Process-level: the HPC launcher	56
12.3 SLURM job arrays	57
12.4 C++ MPI	57
12.5 Choosing the axis of parallelism	58
13 Tuning Guide and Troubleshooting	59
13.1 Which method, which knobs — a decision guide	59
13.2 Systematic tuning, in order	59
13.3 Symptom tables	59
13.3.1 General	60
13.3.2 SQAPT	60
13.3.3 Optimal schedule	60
13.4 Environment issues	60
13.5 Reproducibility checklist	61
14 The C++ API	62
14.1 Umbrella include and headers	62
14.2 A complete C++ example	62
14.3 Implementation notes for contributors	63
V Appendices	65
A Symbols, Conventions, and Reference Tables	66
A.1 Symbol glossary	66
A.2 Universality exponents for α	66
A.3 Key formulas on one page	67
A.4 Repository map	67
A.5 License	68

Part I

Foundations

Introduction

1.1 What is qanneal?

qanneal is a research-grade optimiser for *Ising* and *QUBO* problems — the two canonical formulations into which an enormous range of hard combinatorial optimisation problems (partitioning, scheduling, routing, satisfiability, portfolio selection, protein folding, ...) can be encoded. It provides four annealing engines, spanning the spectrum from purely classical thermal optimisation to the closest CPU-side simulation of a physical quantum annealer:

Method	Name	What it simulates	Ch.
Simulated Annealing	sa	Classical thermal fluctuations	4
Simulated Quantum Annealing	sqa	Discrete-time path integral (Trotterised QA)	5
SQA + Parallel Tempering	sqapt	QA with replica exchange, (β, Γ) ladder	6
Continuous-Time PIMC	ctpimc	Continuous-time path integral (worldlines)	7

All four engines share a single high-level Python entry point, `solve()`, and a common C++17 core exposed through `pybind11` bindings. On top of the engines, qanneal implements a theoretically grounded *optimal adaptive $J_{\perp}$ schedule* (Chapter [8](#)) derived from the local adiabaticity condition of Roland and Cerf: the annealing trajectory automatically slows down where the simulated system is close to its quantum critical point and accelerates elsewhere.

1.2 Design philosophy

One call to solve.

`solve(problem, method=..., ...)` accepts NumPy arrays, dictionaries, edge lists, `dimod` binary quadratic models and `networkx` graphs, auto-detects the input type, auto-tunes a schedule from the problem's own energy scale, runs any number of independent reads and returns the best result.

Physics first.

Every knob in the library corresponds to a physical quantity: β is an inverse temperature, Γ a transverse field, M a number of imaginary-time slices, $J_{\perp}$ a Trotter bond strength. The documentation (this manual) derives all of them so that parameter choices can be reasoned about rather than guessed.

Nothing hard-coded.

Schedule endpoints, temperature ladders and the adaptive-schedule calibration all derive

from the *problem's* coupling scale (via random-flip probing or the RMS coupling J_{rms}), so the same code works unchanged from $n = 10$ toy instances to $n > 100\,000$ sparse graphs.

Research instrumentation.

Observer classes capture per-sweep energy, magnetisation, and full spin-state traces; the adaptive schedule exposes its realised $J_{\perp}$ trajectory and bond-susceptibility trace so that every run can be audited.

HPC ready.

OpenMP parallelism within a run, a multiprocessing/MPI launcher across reads and nodes, and SLURM job-array recipes.

1.3 Architecture at a glance

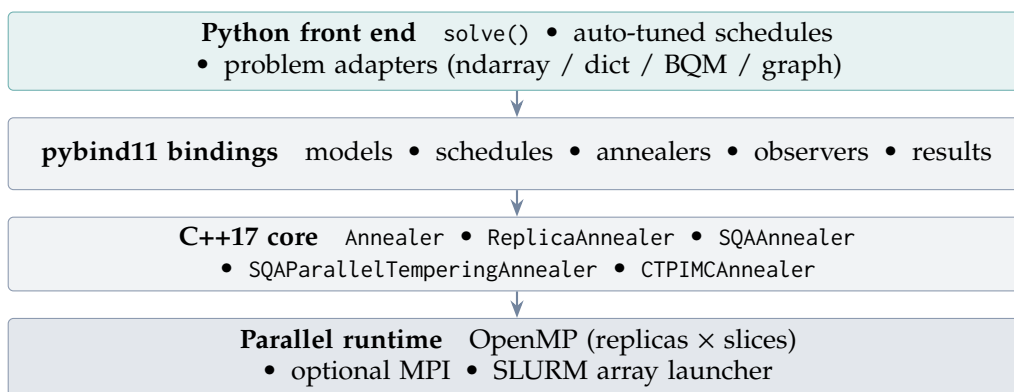


Figure 1.1: The qanneal software stack. Users normally interact only with the top layer; every lower layer remains individually scriptable.

1.4 A sixty-second tour

Install from the repository root and solve a three-variable QUBO:

```
python -m pip install . --no-build-isolation
```

```

import numpy as np
from qanneal import solve

# Minimise E(x) = x0 + x2 - 2 x0 x1 - 2 x1 x2 over bits x in {0,1}^3
Q = np.array([
    [ 1.0, -2.0,  0.0],
    [-2.0,  0.0, -2.0],
    [ 0.0, -2.0,  1.0],
], dtype=float)

result = solve(Q, method="sqapt", reads=16, seed=0, return_bits=True)
print(result.best_sample) # bit vector, e.g. [1 1 1]
  
```

```
print(result.best_energy) # minimum QUBO energy found
```

Everything that just happened — how the QUBO became an Ising Hamiltonian, how a (β, Γ) ladder was tuned to the matrix’s energy scale, what a Trotter slice is, why replica exchange helps, and how to make the run adaptive, parallel or verifiable — is explained in the rest of this manual.

1.5 How to read this manual

Part I — Foundations.

The Ising and QUBO models, their exact equivalence, encodings of classic NP-hard problems, and the statistical mechanics (Boltzmann distributions, Markov chains, detailed balance) on which every engine rests. Read this if you want the mathematics from zero.

Part II — The annealing engines.

One chapter per engine. Each derives the algorithm, states the exact acceptance rules implemented in the C++ core, and ends with parameter guidance.

Part III — The optimal adaptive schedule.

The local-adiabaticity derivation, the bond susceptibility χ_B , budget calibration, the inverse-CDF trajectory, and how to diagnose a run.

Part IV — Using the library.

Installation, the full Python API reference, worked end-to-end examples, parallel/HPC deployment, troubleshooting, and the C++ API.

Typographical conventions

Python and C++ identifiers are set in monospace. Key equations are boxed with a navy rule; physical intuition appears in plum “Physical picture” boxes; pitfalls appear in amber warning boxes. Spins are $s_i \in \{-1, +1\}$, bits are $x_i \in \{0, 1\}$, β is always an inverse temperature and M the number of Trotter slices.

Mathematical Foundations: Ising and QUBO

This chapter defines the two problem formulations qanneal accepts, proves their exact equivalence, and shows how to encode several classic NP-hard problems. Everything later in the manual reduces to minimising one of these two energy functions.

2.1 The Ising model

An *Ising problem* on n spins is defined by a vector of local fields $h \in \mathbb{R}^n$, a symmetric coupling matrix $J \in \mathbb{R}^{n \times n}$ with zero diagonal, and an optional constant offset $c \in \mathbb{R}$. Each variable is a *spin* $s_i \in \{-1, +1\}$.

$$E(s) = \sum_{i=1}^n h_i s_i + \sum_{i < j} J_{ij} s_i s_j + c, \quad s_i \in \{-1, +1\}. \quad (2.1)$$

The task is to find the *ground state* $s^* = \arg \min_{s \in \{-1, +1\}^n} E(s)$. The search space has 2^n configurations, and for general J the problem is NP-hard (it contains MaxCut as the special case $h = 0$).

2.1.1 Sign conventions

qanneal uses the convention of Eq. (2.1) with a *plus sign in front of the coupling term*. Consequences:

- $J_{ij} < 0$ is *ferromagnetic*: the term $J_{ij} s_i s_j$ is lowest when $s_i = s_j$, so the bond prefers aligned spins.
- $J_{ij} > 0$ is *antiferromagnetic*: the bond prefers $s_i = -s_j$.
- $h_i > 0$ pushes spin i toward $s_i = -1$ (the term $h_i s_i$ is then negative); $h_i < 0$ pushes toward $+1$.
- Only the upper triangle $i < j$ of J enters the energy. The Python model classes expect a *symmetric* matrix ($J_{ij} = J_{ji}$) and read the upper triangle.
- The offset c shifts all energies uniformly. It never affects *which* configuration is optimal, but it lets converted problems (e.g. QUBO→Ising) report energies in the original problem's units.

2.1.2 Frustration — why the problem is hard

If every bond could be satisfied simultaneously the ground state would be found by greedy propagation. Hardness arises from *frustration*: cycles of bonds whose preferences cannot all be met. The elementary example is a triangle of antiferromagnetic bonds $J_{12}, J_{13}, J_{23} > 0$: any spin assignment satisfies at most two of the three bonds. Frustrated systems have *rugged energy landscapes* — exponentially many local minima separated by barriers — and it is exactly this structure that annealing methods (and their quantum-inspired refinements) are designed to navigate.

A local minimum is a configuration whose energy cannot be lowered by flipping any single spin. A greedy descent gets trapped in the first such minimum it meets. Thermal annealing escapes over barriers with probability $e^{-\beta \Delta E}$ (Chapter 4); quantum annealing tunnels *through* barriers with an amplitude that depends on barrier *width* rather than height (Chapter 5). Tall, narrow barriers are precisely where the quantum route wins.

2.2 The QUBO model

A *Quadratic Unconstrained Binary Optimisation* (QUBO) problem is specified by a single matrix $Q \in \mathbb{R}^{n \times n}$ over binary variables $x_i \in \{0, 1\}$:

$$E(x) = \sum_{i=1}^n \sum_{j=1}^n Q_{ij} x_i x_j, \quad x_i \in \{0, 1\}. \quad (2.2)$$

Because $x_i^2 = x_i$ for binary variables, the diagonal entries Q_{ii} act as *linear* coefficients and the off-diagonal entries as pairwise couplings. The double sum counts both orders (i, j) and (j, i) ; if you start from an upper-triangular formulation, either mirror the off-diagonal weights or halve them, keeping the represented pair strength explicit.

2.3 Exact equivalence of QUBO and Ising

The two formulations are related by the affine change of variables

$$x_i = \frac{1 + s_i}{2} \quad \Longleftrightarrow \quad s_i = 2x_i - 1. \quad (2.3)$$

so $s_i = +1 \mapsto x_i = 1$ and $s_i = -1 \mapsto x_i = 0$. We now derive the exact Ising parameters produced by `QUBO.to_ising()`.

Split Eq. (2.2) into diagonal and off-diagonal parts and substitute Eq. (2.3). For the diagonal

part, using $x_i^2 = x_i$:

$$\sum_i Q_{ii} x_i = \sum_i Q_{ii} \frac{1+s_i}{2} = \frac{1}{2} \sum_i Q_{ii} + \frac{1}{2} \sum_i Q_{ii} s_i. \quad (2.4)$$

For the off-diagonal part, expand $x_i x_j = \frac{1}{4}(1+s_i)(1+s_j) = \frac{1}{4}(1+s_i+s_j+s_i s_j)$:

$$\begin{aligned} \sum_{i \neq j} Q_{ij} x_i x_j &= \frac{1}{4} \sum_{i \neq j} Q_{ij} + \frac{1}{4} \sum_{i \neq j} Q_{ij} (s_i + s_j) + \frac{1}{4} \sum_{i \neq j} Q_{ij} s_i s_j \\ &= \frac{1}{4} \sum_{i \neq j} Q_{ij} + \frac{1}{4} \sum_i s_i \sum_{j \neq i} (Q_{ij} + Q_{ji}) + \sum_{i < j} \frac{Q_{ij} + Q_{ji}}{4} s_i s_j. \end{aligned} \quad (2.5)$$

Collecting terms into the Ising form of Eq. (2.1) gives the exact, approximation-free conversion:

$$h_i = \frac{Q_{ii}}{2} + \frac{1}{4} \sum_{j \neq i} (Q_{ij} + Q_{ji}), \quad J_{ij} = \frac{Q_{ij} + Q_{ji}}{4} \quad (i < j), \quad c = \frac{1}{2} \sum_i Q_{ii} + \frac{1}{4} \sum_{i \neq j} Q_{ij}. \quad (2.6)$$

Every energy is preserved exactly: $E_{\text{QUBO}}(x) = E_{\text{Ising}}(s)$ whenever x and s are related by Eq. (2.3). In particular the minimisers coincide, and the offset c makes even the numerical energy values agree — this is why qanneal carries c through all its solvers.

Automatic conversion
You never need to perform this conversion by hand. `solve()` accepts a QUBO matrix (or dict, entry list, or dimod BQM) directly and converts internally; `return_bits=True` maps the resulting spins back to bits. The formulas above are given so that reported energies and parameters are fully auditable.

2.4 Encoding classic problems

The practical skill in using any Ising machine is the *encoding*: recast your objective (plus constraints, via penalty terms) as Eq. (2.1) or Eq. (2.2). Three canonical examples follow; each is used again in the worked examples of Chapter 11.

2.4.1 Number partitioning

Problem. Given positive numbers $a_1, \dots, a_n$, split them into two groups whose sums are as equal as possible.

Encoding. Let $s_i = +1$ if a_i goes to group A and $s_i = -1$ for group B. The signed discrepancy is $D(s) = \sum_i a_i s_i$ and we minimise $D(s)^2$:

$$D(s)^2 = \left(\sum_i a_i s_i \right)^2 = \sum_i a_i^2 \underbrace{s_i^2}_{=1} + 2 \sum_{i < j} a_i a_j s_i s_j = \underbrace{\sum_i a_i^2}_c + \sum_{i < j} \underbrace{2a_i a_j}_{J_{ij}} s_i s_j. \quad (2.7)$$

So the Ising instance has $h = 0$, $J_{ij} = 2a_i a_j$ and $c = \sum_i a_i^2$; a perfect partition has energy exactly 0. Note every pair is coupled (fully-connected antiferromagnet) — use `DenseIsing`.

2.4.2 MaxCut

Problem. Given a weighted graph $G = (V, E, w)$, partition V into two sets maximising the total weight of edges crossing the partition.

Encoding. With $s_i = \pm 1$ labelling the two sides, an edge (i, j) is cut iff $s_i \neq s_j$, i.e. $\frac{1-s_i s_j}{2} = 1$. Hence

$$\text{Cut}(s) = \sum_{(i,j) \in E} w_{ij} \frac{1-s_i s_j}{2} = \frac{W}{2} - \frac{1}{2} \sum_{(i,j) \in E} w_{ij} s_i s_j, \quad W = \sum_{(i,j) \in E} w_{ij}. \quad (2.8)$$

Maximising the cut is therefore *minimising* the Ising energy with $h_i = 0$, $J_{ij} = w_{ij}$ on the graph's edges and offset $c = -W/2$ if you want $E = -\text{Cut}$ numerically. Sparse graphs should use `SparseIsing`.

2.4.3 Constrained problems via penalties: maximum-weight independent set

Problem. In a graph G , choose a set of vertices, no two adjacent, maximising total vertex weight w_i .

Encoding. With bits $x_i \in \{0, 1\}$ ($x_i = 1$ means “vertex chosen”), maximise $\sum_i w_i x_i$ subject to $x_i x_j = 0$ for every edge. Constraints become penalty terms with a multiplier $\lambda > \max_i w_i$:

$$E(x) = - \sum_i w_i x_i + \lambda \sum_{(i,j) \in E} x_i x_j, \quad (2.9)$$

which is already a QUBO: $Q_{ii} = -w_i$, $Q_{ij} = \lambda/2$ for $i \neq j$ adjacent (splitting λ over both orders). The penalty is calibrated so that violating any constraint always costs more than the largest possible gain, guaranteeing that the QUBO optimum is a feasible independent set.

Penalty scale

Penalty multipliers should be as small as correctness allows. Oversized λ inflates the energy scale of the constraint terms relative to the objective, which stiffens the landscape and makes annealing (classical or quantum) mix more slowly. `qanneal`'s auto-tuned schedules measure the *combined* energy scale, which mitigates but cannot remove a badly unbalanced encoding.

2.5 Model classes in qanneal

Three model containers implement the common Hamiltonian interface and are accepted anywhere a problem is expected:

Class	Storage	Memory	Best for
DenseIsing(h, J, c)	dense $n \times n$	$O(n^2)$	$n \lesssim 3000$, fully connected
SparseIsing(h, edges, n, c)	edge list	$O(n + E)$	sparse graphs, n up to 10^5+
QUBO(Q)	dense or sparse	$O(n^2) / O(Q)$	binary problems; <code>.to_ising()</code>

Their exact constructors and methods are documented in Section 10.2. Two operations matter for performance and are worth stating now:

energy(s) evaluates Eq. (2.1) from scratch: $O(n^2)$ for DenseIsing, $O(|E|)$ for SparseIsing.

delta_energy(s, i) returns the energy change if spin i is flipped, *without* recomputing the total.

Writing the *local field* $f_i = h_i + \sum_{j \neq i} J_{ij} s_j$, flipping $s_i \rightarrow -s_i$ changes the energy by

$$\Delta E_i = -2 s_i f_i, \quad (2.10)$$

an $O(n)$ (dense) or $O(\deg i)$ (sparse) computation. The C++ annealers go one step further and *cache* all local fields, making each Metropolis proposal $O(1)$ with an $O(\deg i)$ cache update only on *accepted* flips. This caching is the single most important implementation detail behind qanneal's sweep throughput.

Statistical Mechanics Background

Every engine in qanneal is a Markov-chain Monte Carlo (MCMC) sampler of a Gibbs–Boltzmann distribution, driven slowly toward the zero-temperature (or zero-quantum-fluctuation) limit. This chapter builds that machinery from scratch: the Boltzmann distribution, why sampling it at low temperature solves optimisation problems, Markov chains, detailed balance, and the Metropolis rule. Readers fluent in MCMC can skim to Chapter 4.

3.1 The Boltzmann distribution

A system with configuration space Ω (for us, $\Omega = \{-1, +1\}^n$) and energy function $E : \Omega \rightarrow \mathbb{R}$, held in equilibrium at temperature T , occupies configuration s with probability

$$\pi_\beta(s) = \frac{e^{-\beta E(s)}}{Z(\beta)}, \quad Z(\beta) = \sum_{s \in \Omega} e^{-\beta E(s)}, \quad \beta = \frac{1}{k_B T}. \quad (3.1)$$

β is the *inverse temperature*; in qanneal it is always dimensionless (energies carry the units). $Z(\beta)$ is the *partition function*. Two limits explain the whole strategy of annealing:

- $\beta \rightarrow 0$ (infinite temperature): π_β is uniform over all 2^n configurations — pure exploration, no preference for low energy.
- $\beta \rightarrow \infty$ (zero temperature): the ratio $\pi_\beta(s)/\pi_\beta(s^*) = e^{-\beta(E(s)-E(s^*))} \rightarrow 0$ for every non-optimal s , so *all probability mass concentrates on the ground states*. Sampling at large β is optimisation.

The catch is that sampling π_β directly at large β is as hard as the optimisation problem itself: naive samplers get trapped in local minima because the distribution is a set of sharp, well-separated peaks. Annealing resolves this by *starting* at small β , where the distribution is easy to sample, and increasing β slowly enough that the sampler tracks the shifting distribution.

3.2 Markov chains and stationarity

We generate samples by a Markov chain: a random walk on Ω whose next state depends only on the current one, via a transition kernel $P(s \rightarrow s') \geq 0$ with $\sum_{s'} P(s \rightarrow s') = 1$. A distribution π is *stationary* for P if it is preserved by one step of the chain:

$$\sum_s \pi(s) P(s \rightarrow s') = \pi(s') \quad \text{for all } s'. \quad (3.2)$$

If, additionally, the chain is *irreducible* (any state reachable from any other) and *aperiodic*, the ergodic theorem guarantees that the chain's state distribution converges to π from any start, and long-run averages along the chain converge to expectations under π .

3.2.1 Detailed balance

Verifying Eq. (3.2) directly is awkward. A stronger, local condition that implies it is *detailed balance*:

$$\pi(s) P(s \rightarrow s') = \pi(s') P(s' \rightarrow s) \quad \text{for every pair } s, s'. \quad (3.3)$$

Summing both sides over s immediately recovers Eq. (3.2). Detailed balance says the equilibrium probability flux between any two states is symmetric — the chain, run at equilibrium, is statistically indistinguishable from its time reversal.

3.3 The Metropolis rule, derived

Split each transition into a *proposal* $g(s \rightarrow s')$ (in qanneal: pick a spin uniformly and flip it, so g is symmetric, $g(s \rightarrow s') = g(s' \rightarrow s)$) and an *acceptance probability* $A(s \rightarrow s')$, so $P(s \rightarrow s') = g(s \rightarrow s') A(s \rightarrow s')$ for $s' \neq s$. Substituting into Eq. (3.3) with $\pi = \pi_\beta$ from Eq. (3.1), the proposal factors cancel and the partition function cancels too:

$$\frac{A(s \rightarrow s')}{A(s' \rightarrow s)} = \frac{\pi_\beta(s')}{\pi_\beta(s)} = e^{-\beta \Delta E}, \quad \Delta E = E(s') - E(s). \quad (3.4)$$

Any A satisfying this ratio works; the choice that maximises acceptance (and hence mixing speed) is the *Metropolis–Hastings* rule:

$$A(s \rightarrow s') = \min(1, e^{-\beta \Delta E}) = \begin{cases} 1 & \Delta E \leq 0 \quad (\text{downhill: always accept}) \\ e^{-\beta \Delta E} & \Delta E > 0 \quad (\text{uphill: accept with Boltzmann weight}). \end{cases} \quad (3.5)$$

This is exactly what the C++ core executes; for a single-spin flip the ΔE comes from the cached-local-field formula Eq. (2.10), making each proposal $O(1)$.

At small β almost every move is accepted and the walker diffuses freely across the landscape. At large β uphill moves are exponentially suppressed and the walker rolls downhill into the nearest basin. Annealing is the art of spending long enough at intermediate β — where the walker can still cross barriers whose height is a few times $1/\beta$ — that it ends in the *right* basin before freezing.

3.3.1 Sweeps, ergodicity, and shuffled visit order

A *sweep* is n proposals — on average one per spin. `qanneal` visits spins in a random order re-shuffled each sweep (rather than uniformly at random with replacement), which lowers the chance of neglecting a spin for long stretches while preserving detailed balance. All engine parameters counting “sweeps” (`sweeps_per_beta`, `sweeps_per_step`, `worldline_sweeps`, `cluster_sweeps`) are in units of full sweeps.

3.4 From sampling to annealing

Simulated annealing [2] runs Metropolis dynamics while increasing β along a *schedule* $\beta_1 < \beta_2 < \dots < \beta_K$, spending a fixed number of sweeps at each value. Two classical facts frame all schedule design:

1. **(Geman–Geman)** With a logarithmically slow schedule $\beta_k \propto \log k$, SA converges to a global optimum with probability one. This is far too slow to use, but it shows the trade-off is fundamental: faster cooling risks freezing into excited states.
2. **(Acceptance targeting)** A schedule is well-matched to a problem when the *uphill acceptance rate* decays from near 1 at the start to near 0 at the end. Given a characteristic uphill move size Δ , the acceptance at inverse temperature β is $p = e^{-\beta\Delta}$, hence

$$\beta(p) = \frac{-\ln p}{\Delta}. \quad (3.6)$$

`qanneal`’s auto-tuned schedules (Section 4.2.2) measure Δ from the problem itself — the 75th percentile of $|\Delta E|$ over random single-spin flips — and place $\beta_{\text{start}}, \beta_{\text{end}}$ at target acceptances via Eq. (3.6). No manual temperature tuning is required.

Quantum analogue

Chapters 5–7 replace “thermal fluctuations controlled by $1/\beta$ ” with “quantum fluctuations controlled by a transverse field Γ ”. The mathematical machinery of this chapter survives intact: the quantum partition function is mapped (exactly in a well-defined limit) onto a *classical* Boltzmann distribution in one extra dimension, and the very same Metropolis rule Eq. (3.5) samples it.

Part II

The Annealing Engines

Simulated Annealing (sa)

The classical baseline: Metropolis dynamics under a rising- β schedule. Fast, simple, and surprisingly strong on smooth landscapes; every other engine in `qanneal` is measured against it.

4.1 The algorithm

Given a Hamiltonian $E(\cdot)$ (Chapter 2) and a schedule $\beta_1 < \dots < \beta_K$:

1. Draw a uniformly random initial state $s \in \{-1, +1\}^n$ and compute the local-field cache $f_i = h_i + \sum_{j \neq i} J_{ij} s_j$.
2. For each schedule step $k = 1, \dots, K$, repeat `sweeps_per_beta` times:
 - a. For each spin i in a freshly shuffled order: propose flipping s_i ; compute $\Delta E = -2s_i f_i$ (Eq. (2.10)); accept with probability $\min(1, e^{-\beta_k \Delta E})$ (Eq. (3.5)); on acceptance update $s_i \rightarrow -s_i$ and the affected cache entries $f_j += -2s_i J_{ij}$.
3. Track the best configuration ever visited; return it with its energy and the per-step energy trace.

The returned optimum is the *best-ever* state, not the final state — a late uphill excursion can never lose an already-found solution.

Complexity. One sweep costs $O(n)$ proposals; each proposal is $O(1)$, and each *accepted* flip costs $O(n)$ (dense) or $O(\deg i)$ (sparse) to refresh the cache. A full run is $O(K \cdot \text{sweeps_per_beta} \cdot n \cdot \bar{d})$ with $\bar{d}$ the mean accepted-flip degree — in practice linear in problem size for sparse graphs.

4.2 Schedule design

4.2.1 Fixed linear schedule

The simplest choice ramps β linearly:

```
from qanneal import AnnealSchedule
schedule = AnnealSchedule.linear(beta_start=0.1, beta_end=5.0, steps=60)
```

Guidance: β_{start} should be hot enough that nearly all moves are accepted ($\beta_{\text{start}} \Delta \lesssim 0.15$), and β_{end} cold enough that uphill moves are rare ($\beta_{\text{end}} \Delta \gtrsim 3$), where Δ is the typical $|\Delta E|$ of a

flip. Since Δ depends on your coupling scale, fixed values only suit problems of a known scale — which motivates:

4.2.2 Problem-adaptive tuned schedules (recommended)

`auto_schedule_sa_tuned(problem, mode=...)` removes the guesswork:

1. Sample probes (default 256) random single-spin flips on random states and record $|\Delta E|$ for each.
2. Set the characteristic scale Δ to the *75th percentile* of the $|\Delta E|$ sample (robust to outliers, biased toward the barrier-sized moves that matter).
3. Choose target acceptances ($p_{\text{start}}, p_{\text{end}}$) from the mode preset and invert Eq. (3.6): $\beta_{\text{start}} = -\ln p_{\text{start}}/\Delta$, $\beta_{\text{end}} = -\ln p_{\text{end}}/\Delta$.
4. Pick the step count from the preset, growing as $\sqrt{n}$.

mode	p_{start}	p_{end}	steps	use case
"fast"	0.85	0.22	$30 + 6\sqrt{n}$	prototyping, large sweeps
"balanced"	0.88	0.10	$60 + 6\sqrt{n}$	default production
"accurate"	0.90	0.04	$110 + 6\sqrt{n}$	best solution quality

```
from qanneal import auto_schedule_sa_tuned, solve

schedule = auto_schedule_sa_tuned(ising, mode="balanced")
result = solve(ising, method="sa", schedule=schedule, reads=32, seed=7)
```

The optional `beta_end_scale` multiplies β_{end} (values > 1 freeze harder); `steps` overrides the preset count.

4.3 Independent replicas: ReplicaAnnealer

Independent restarts are the cheapest variance reduction: run R chains from different random seeds and keep the best. `ReplicaAnnealer` does this inside one C++ call, parallelised over replicas with OpenMP:

```
from qanneal import ReplicaAnnealer

ann = ReplicaAnnealer(ising, schedule, replicas=16)
res = ann.run(sweeps_per_beta=50)
res.global_best_energy      # best over all replicas
res.replicas                # per-replica results
res.average_energy_trace    # mean energy across replicas per step
```

If a single run finds the optimum with probability p , then R independent runs succeed with probability $1 - (1 - p)^R$ — restarts convert a mediocre per-run success rate into a high

aggregate one, at perfectly parallel cost. The high-level `solve(..., reads=R)` applies the same principle across full solver invocations for every method.

4.4 Classical parallel tempering

Annealing commits to a one-way trajectory in β . *Parallel tempering* (replica exchange) instead runs R chains *simultaneously* at a fixed ladder $\beta_1 < \beta_2 < \dots < \beta_R$ and lets configurations migrate between temperatures.

4.4.1 The swap rule, derived

The joint system samples the product measure $\Pi(s_1, \dots, s_R) \propto \prod_r e^{-\beta_r E(s_r)}$. A *swap move* proposes exchanging the configurations at adjacent rungs $r, r+1$. Detailed balance for the joint measure requires acceptance

$$A_{\text{swap}} = \min\left(1, \frac{e^{-\beta_r E(s_{r+1})} e^{-\beta_{r+1} E(s_r)}}{e^{-\beta_r E(s_r)} e^{-\beta_{r+1} E(s_{r+1})}}\right) \quad (4.1)$$

which simplifies to the celebrated form

$$A_{\text{swap}} = \min\left(1, e^{(\beta_{r+1} - \beta_r)(E(s_{r+1}) - E(s_r))}\right). \quad (4.2)$$

A configuration that found a low energy while “hot” (small β , high mobility) is passed down the ladder toward large β , where it is refined; conversely, stuck cold configurations are recycled upward to remelt. Each chain still satisfies detailed balance at its own β_r , so all equilibrium properties are preserved.

4.4.2 Usage

```
import numpy as np
from qanneal import ParallelTemperingAnnealer

betas = np.linspace(0.1, 5.0, 8).tolist()
ann = ParallelTemperingAnnealer(ising, betas)
res = ann.run(sweeps_per_step=50, steps=100, swap_interval=1)

res.best_energy
res.swap_acceptance_trace # fraction of accepted swaps per step
```

Healthy swap acceptance

Aim for a mean swap acceptance of roughly 0.2–0.5. Near zero means the rungs are too far apart (energy distributions at adjacent β barely overlap — add rungs or shrink the range); near one means rungs are wastefully close. A geometric β ladder is the standard

starting point because the overlap between adjacent rungs is then roughly constant across the ladder.

The quantum generalisation of this machinery — replica exchange on a joint (β, Γ) ladder — is the sqapt engine of Chapter 6.

4.5 When to choose sa

- You need a fast, dependency-free baseline or a sanity check for an encoding.
- The landscape is smooth or funnel-like (many greedy paths reach the optimum).
- Very large sparse instances where the cost per sweep dominates and the quantum engines' extra dimension (M slices) is unaffordable.

For rugged, frustrated landscapes with tall narrow barriers, continue to Chapter 5.

Simulated Quantum Annealing (sqa)

This chapter contains the theoretical heart of qanneal. We introduce the transverse-field Ising model, state the adiabatic theorem that motivates quantum annealing, and then derive — in full detail — the Suzuki–Trotter path-integral mapping that turns the quantum partition function into a classical Ising model in one extra dimension. The result is the exact effective action and acceptance rules executed by the C++ `SQAAnealer`, including the Trotter coupling $J_{\perp} = \frac{1}{2} \ln \coth(\beta\Gamma/M)$ that reappears throughout the rest of the manual.

5.1 Quantum annealing in one page

Quantum annealing [4, 5] encodes the optimisation problem in the *longitudinal* part of a quantum spin Hamiltonian and adds a *transverse field* that induces quantum fluctuations:

$$\hat{H}(\Gamma) = \underbrace{\sum_i h_i \hat{\sigma}_i^z + \sum_{i<j} J_{ij} \hat{\sigma}_i^z \hat{\sigma}_j^z}_{\hat{H}_{\text{cl}} \text{ (the problem)}} - \Gamma \sum_i \hat{\sigma}_i^x, \quad (5.1)$$

where $\hat{\sigma}_i^{x,z}$ are Pauli operators on spin i . $\hat{H}_{\text{cl}}$ is diagonal in the computational (σ^z) basis and its ground state is exactly the Ising optimum we seek. The transverse term $-\Gamma \sum_i \hat{\sigma}_i^x$ flips spins ($\hat{\sigma}^x|\pm 1\rangle = |\mp 1\rangle$) and therefore lets the state *tunnel* between classical configurations.

The protocol: start at large Γ , where the ground state is the trivial uniform superposition $\bigotimes_i \frac{1}{\sqrt{2}}(|+1\rangle + |-1\rangle)$; reduce $\Gamma \rightarrow 0$ slowly. The **adiabatic theorem** guarantees that if the sweep is slow compared to the inverse square of the spectral gap $\Delta_{\text{gap}}(\Gamma)$ (the energy difference between the ground and first excited state), the system remains in its instantaneous ground state throughout and ends in the classical optimum. The total time required scales as

$$T \gtrsim \frac{\max_{\Gamma} |\langle 1 | \partial_{\Gamma} \hat{H} | 0 \rangle|}{\min_{\Gamma} \Delta_{\text{gap}}^2(\Gamma)}, \quad (5.2)$$

so the bottleneck is the *minimum gap*, typically encountered at a quantum phase transition separating the disordered (large Γ) phase from the ordered (small Γ) phase. This observation drives the optimal schedule of Chapter 8: sweep slowly *only* near the transition.

Thermal hopping surmounts a barrier of height B at rate $\sim e^{-\beta B}$, indifferent to the barrier's width. Quantum tunnelling through a barrier of width w and height B proceeds at rate $\sim e^{-c w \sqrt{B} / \Gamma}$: it pays for *width*, not height. Rugged landscapes whose minima are separated by tall-but-thin walls are therefore the natural habitat of quantum annealing — and of its classical-hardware simulation, SQA.

5.2 The Suzuki–Trotter mapping, in full

Real-time simulation of Eq. (5.1) on a CPU is exponentially expensive. *Simulated* quantum annealing instead samples the quantum system's *thermal equilibrium* at inverse temperature β — the density matrix $e^{-\beta \hat{H}} / Z$ — using a mapping to a classical model that Monte Carlo can handle. We now derive it.

5.2.1 Step 1: split the exponential

The partition function is $Z = \text{Tr } e^{-\beta \hat{H}}$ with $\hat{H} = \hat{H}_{\text{cl}} + \hat{H}_q$, $\hat{H}_q = -\Gamma \sum_i \hat{\sigma}_i^x$. Since $[\hat{H}_{\text{cl}}, \hat{H}_q] \neq 0$ we cannot factorise the exponential exactly, but we can *Trotterise*: write $e^{-\beta \hat{H}} = (e^{-\frac{\beta}{M} \hat{H}})^M$ and apply the Lie–Trotter splitting to each thin slice,

$$e^{-\frac{\beta}{M} (\hat{H}_{\text{cl}} + \hat{H}_q)} = e^{-\frac{\beta}{M} \hat{H}_{\text{cl}}} e^{-\frac{\beta}{M} \hat{H}_q} + O\left(\frac{\beta^2}{M^2} \|[\hat{H}_{\text{cl}}, \hat{H}_q]\|\right). \quad (5.3)$$

The neglected term is the *Trotter error*; it vanishes as $M \rightarrow \infty$ like $O((\beta/M)^2)$ per slice, i.e. $O(\beta^2/M)$ overall. M is the `trotter_slices` parameter.

5.2.2 Step 2: insert complete sets of classical states

Insert the identity $\mathbb{1} = \sum_s |s\rangle\langle s|$ (sum over all 2^n classical spin configurations) between every pair of factors. Writing s^t for the configuration inserted at slot $t = 1, \dots, M$ and using cyclicity of the trace ($s^{M+1} \equiv s^1$, *periodic boundary in imaginary time*):

$$Z \approx \sum_{s^1, \dots, s^M} \prod_{t=1}^M \underbrace{\langle s^t | e^{-\frac{\beta}{M} \hat{H}_{\text{cl}}} | s^t \rangle}_{e^{-\frac{\beta}{M} E_{\text{cl}}(s^t)}} \underbrace{\langle s^t | e^{\frac{\beta \Gamma}{M} \sum_i \hat{\sigma}_i^x} | s^{t+1} \rangle}_{\text{transverse matrix element}}. \quad (5.4)$$

The classical factor is diagonal, giving the Boltzmann weight of each slice at the *reduced* inverse temperature β/M . The transverse factor factorises over sites, so we only need the 2×2 single-spin matrix element.

5.2.3 Step 3: the single-spin transfer matrix

Let $a = \beta\Gamma/M$. Using $e^{a\hat{\sigma}^x} = \cosh a \mathbb{I} + \sinh a \hat{\sigma}^x$:

$$\langle s | e^{a\hat{\sigma}^x} | s' \rangle = \begin{cases} \cosh a & s = s' \\ \sinh a & s \neq s'. \end{cases} \quad (5.5)$$

We now rewrite this as a classical Ising bond between the same site in adjacent slices, i.e. find Λ and $J_\perp$ such that $\langle s | e^{a\hat{\sigma}^x} | s' \rangle = \Lambda e^{J_\perp ss'}$. Matching the two cases ($ss' = +1$ and $ss' = -1$):

$$\Lambda e^{J_\perp} = \cosh a, \quad \Lambda e^{-J_\perp} = \sinh a \implies e^{2J_\perp} = \coth a, \quad \Lambda^2 = \sinh a \cosh a = \frac{1}{2} \sinh 2a. \quad (5.6)$$

$$J_\perp(\beta, \Gamma, M) = \frac{1}{2} \ln \coth\left(\frac{\beta\Gamma}{M}\right) = \frac{1}{2} \ln \frac{1}{\tanh(\beta\Gamma/M)} > 0. \quad (5.7)$$

$J_\perp$ is a *ferromagnetic* bond along imaginary time (it rewards $s_i^t = s_i^{t+1}$; note the effective action below carries it with a minus sign). Its two limits define the whole phenomenology:

- $\Gamma \rightarrow \infty$ (strong quantum fluctuations): a large, $\coth a \rightarrow 1$, $J_\perp \rightarrow 0$ — slices decouple, worldlines flap freely.
- $\Gamma \rightarrow 0^+$ (classical limit): $a \rightarrow 0$, $J_\perp \rightarrow \infty$ — slices lock together into identical copies of one classical configuration.

The Python helper `j_perp_from_beta_gamma(beta, gamma, trotter_slices)` evaluates Eq. (5.7) directly.

Direction of the ramp

Because $J_\perp$ is monotonically *decreasing* in Γ , an annealing run that lowers Γ corresponds to *raising* $J_\perp$. All qanneal adaptive-schedule APIs are parameterised by $J_\perp$ and therefore ramp it *upward*, from `j_perp_start` (quantum end) to `j_perp_end` (classical end). Keep this inversion in mind when reading traces.

5.2.4 Step 4: the effective classical action

Substituting both factors into Eq. (5.4):

$$Z \approx C \sum_{\{s_i^t\}} e^{-S[s]}, \quad S[s] = \frac{\beta}{M} \sum_{t=1}^M E_{\text{cl}}(s^t) - J_\perp \sum_{t=1}^M \sum_{i=1}^n s_i^t s_i^{t+1}, \quad (5.8)$$

with $C = \Lambda^{nM}$ an irrelevant constant and periodic slices $s^{M+1} \equiv s^1$.

This is a purely classical Ising model on $n \times M$ spins: M copies (“Trotter slices”) of the original problem, each internally coupled by the rescaled problem couplings $\frac{\beta}{M} J_{ij}$, $\frac{\beta}{M} h_i$, and neighbouring slices tied site-by-site with strength $J_\perp$. Quantum statistics at inverse temperature

β became classical statistics of a $(d+1)$ -dimensional system at unit temperature. Metropolis sampling of e^{-S} (Chapter 3, with S in place of βE) is exact for this effective model.

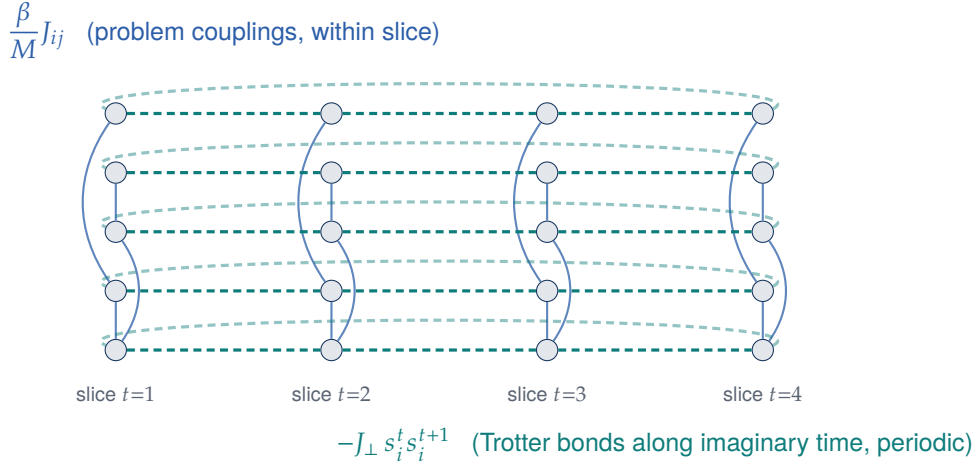


Figure 5.1: The Suzuki–Trotter mapping: M replicas of the classical problem (here $M=4$, $n=5$) coupled site-by-site along a periodic imaginary-time ring. The row of same-site spins across all slices is that site’s *worldline*.

5.2.5 Trotter error and choosing M

The systematic error of the discretisation is $O((\beta\Gamma/M)^2)$ per slice. In practice:

- $M = 16\text{--}32$ (the default is 32) suffices for optimisation use, where we care about ground-state *identification* rather than unbiased thermodynamics.
- Increase M if the per-slice energies disagree strongly with each other late in the anneal (a Trotter-error signature), or when running at large $\beta\Gamma$.
- Memory and time grow linearly in M ; the OpenMP path parallelises over replicas and slices, softening the wall-clock cost.
- `continuous_time_slices > 0` overrides M with a much larger value to approximate the continuous-time limit within the discrete engine; the genuinely continuous algorithm is Chapter 7.

5.3 Monte Carlo moves on the Trotter system

The SQAAnnealer mixes three move types, each preserving detailed balance for the action Eq. (5.8). Every acceptance test is $\min(1, e^{-\Delta S})$; what differs is the update and its ΔS .

5.3.1 Single-spin (slice) moves

Flip one spin s_i^t . The action change has a classical part within the slice and a quantum part from the two adjacent time-bonds:

$$\Delta S = \frac{\beta}{M} \Delta E_{\text{cl}} + 2J_{\perp} s_i^t (s_i^{t-1} + s_i^{t+1}), \quad \Delta E_{\text{cl}} = -2s_i^t f_i^t, \quad (5.9)$$

where f_i^t is the cached local field of slice t (Eq. (2.10)). Both pieces are $O(1)$; the implementation keeps one local-field cache per (replica, slice). `sweeps_per_beta` counts these sweeps.

5.3.2 Worldline moves

Flip site i in *every* slice simultaneously ($s_i^t \rightarrow -s_i^t \forall t$). All imaginary-time bonds $s_i^t s_i^{t+1}$ are products of two flipped spins and are therefore *invariant*: the Trotter term cancels exactly, leaving

$$\Delta S_{\text{worldline}} = \frac{\beta}{M} \sum_{t=1}^M \Delta E_{\text{cl}}^{(t)}. \quad (5.10)$$

Worldline moves are the quantum analogue of a classical single-spin flip and are essential late in the anneal: when $J_{\perp}$ is large the slices are locked and single-slice flips are strongly suppressed ($\Delta S \approx 4J_{\perp}$), while a worldline flip pays no Trotter penalty at all. `worldline_sweeps` (default 5 in `solve()`) counts these.

5.3.3 Swendsen–Wang cluster moves along imaginary time

Between the two extremes sits the cluster move: flip a *contiguous segment* of a worldline. Segments are grown stochastically with the Fortuin–Kasteleyn rule specialised to the 1-D ferromagnetic time-chain of coupling $J_{\perp}$: starting from a random seed (i, t) , extend the cluster across an aligned time-bond with the *join probability*

$$p_{\text{join}} = 1 - e^{-2J_{\perp}}. \quad (5.11)$$

Bonds inside the cluster then flip for free (the FK construction makes the grown-bond factor cancel from the acceptance ratio); the remaining action change — the classical part in every touched slice plus the two *boundary* time-bonds where the cluster ends — is Metropolis-tested:

$$\Delta S_{\text{cluster}} = \frac{\beta}{M} \sum_{t \in \text{cluster}} \Delta E_{\text{cl}}^{(t)} + 2J_{\perp} \sum_{\text{boundary bonds}} s s'. \quad (5.12)$$

Cluster moves interpolate smoothly between single-spin flips (clusters of size 1 when $J_{\perp}$ is small) and full worldline flips (system-spanning clusters when $J_{\perp}$ is large), and dramatically improve mixing in stiff, strongly-coupled regimes. `cluster_sweeps` (default 0) counts these; setting it > 0 is referred to as the +SW variant of an engine.

Which moves to enable

A robust general-purpose mix is `sweeps_per_beta=20–60`, `worldline_sweeps=3–5`, `cluster_sweeps=0`; add `cluster_sweeps=1` for strongly frustrated instances or whenever

convergence visibly stalls at late steps. All three counters multiply runtime linearly.

5.4 The annealing loop and replicas

Putting it together, `SQAAnnealer.run()` executes, for each schedule step (β_k, Γ_k) :

1. recompute $J_{\perp}(\beta_k, \Gamma_k, M)$ (Eq. (5.7)) and the slice temperature factor β_k/M ;
2. run the configured number of slice, worldline, and cluster sweeps with the acceptance rules of Section 5.3;
3. measure the energy of *every slice* of every replica; record the best classical projection seen so far.

The `replicas` parameter runs R independent copies of the whole Trotter system inside one call (OpenMP-parallel, one RNG stream per replica). Besides restart statistics, replicas are what powers the adaptive schedule's susceptibility estimator (Chapter 8), which pools samples across them.

At the end, the returned `best_state` is the best single *slice* ever observed — the “classical projection” of the quantum state — with `best_energy` its true classical energy E_{cl} , and `energy_trace` the per-step mean energy.

5.5 Usage

5.5.1 One-call interface

```
from qanneal import solve

result = solve(
    ising,
    method="sqa",
    reads=16,           # independent restarts
    trotter_slices=32,  # M
    replicas=4,         # parallel Trotter systems per read
    sweeps_per_beta=60,
    worldline_sweeps=4,
    cluster_sweeps=1,   # enable SQA+SW
    seed=42,
)
print(result.best_energy, result.best_sample)
```

5.5.2 Explicit schedules

If no schedule is given, `solve()` builds a tuned one (Section 5.5.3). To control it explicitly:

```
import numpy as np
from qanneal import SQASchedule, SQAAnnealer
```

```

schedule = SQASchedule.from_vectors(
    betas = np.linspace(0.1, 5.0, 80).tolist(),
    gammas = np.geomspace(5.0, 0.01, 80).tolist(), # geometric decay!
)
ann = SQAAnnealer(ising, schedule, trotter_slices=32, replicas=4)
ann.set_seed(7)
res = ann.run(sweeps_per_beta=60, worldline_sweeps=4, cluster_sweeps=1)
print(res.best_energy)

```

Use geometric Γ decay

$J_{\perp}$ depends on Γ logarithmically at small Γ (Eq. (5.7)), so a *linear* Γ ramp spends most of its steps in a narrow band of $J_{\perp}$ and rushes the physically important region. Geometric decay (`np.geomspace`) distributes steps roughly uniformly in $\ln \Gamma$, i.e. far more evenly in $J_{\perp}$.

5.5.3 Problem-adaptive tuned SQA schedules

`auto_schedule_sqa_tuned(problem, mode=...)` extends the SA tuner of Section 4.2.2: the same random-flip probe estimates the energy scale Δ , then

$$\Gamma_{\text{start}} = m_{\Gamma} \Delta, \quad \Gamma_{\text{end}} = \Gamma_{\text{start}} \cdot r_{\Gamma} \cdot \text{gamma_end_scale}, \quad (5.13)$$

with $m_{\Gamma} \in \{2.0, 3.0, 4.5\}$ for fast/balanced/accurate and geometric spacing in between. β ramps exactly as in the SA tuner. Setting Γ_{start} a few times the flip scale guarantees the run actually begins in the quantum-disordered phase.

```

from qanneal import auto_schedule_sqa_tuned
schedule = auto_schedule_sqa_tuned(ising, mode="balanced")

```

5.6 Parameter summary

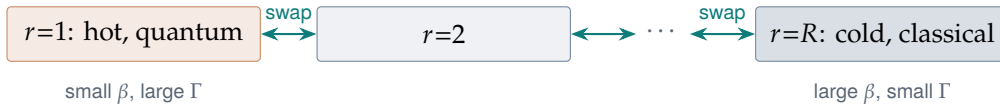
Parameter	Default	Meaning
<code>trotter_slices</code>	32	M ; more slices $\Rightarrow$ less Trotter error, linearly more memory/time.
<code>replicas</code>	1	Independent Trotter systems per run (OpenMP-parallel). Use ≥ 4 with the optimal schedule.
<code>sweeps_per_beta</code>	20	Single-spin sweeps per schedule step.
<code>worldline_sweeps</code>	5	Whole-worldline flips per step; essential at large $J_{\perp}$.
<code>cluster_sweeps</code>	0	Swendsen–Wang segment flips per step ($> 0 = \text{SQA} + \text{SW}$).
<code>continuous_time_slices</code>	0	If > 0 , overrides M to approximate continuous time.

Quantum Parallel Tempering (sqapt)

sqapt combines the two strongest ideas of the previous chapters: the Suzuki–Trotter quantum simulation of Chapter 5 and the replica-exchange machinery of Section 4.4. It is the **recommended default** for rugged, multi-modal problems: several complete SQA systems run simultaneously at different points of a (β, Γ) ladder, and adjacent systems periodically swap configurations so that solutions discovered under strong fluctuations flow toward the strongly-frozen rungs where they are polished.

6.1 The (β, Γ) ladder

Each of the R replicas $r = 1, \dots, R$ is a full Trotter system ($n \times M$ spins) equilibrating at its own pair (β_r, Γ_r) , hence its own action S_r (Eq. (5.8)) with its own $J_{\perp}^{(r)} = J_{\perp}(\beta_r, \Gamma_r, M)$. The ladder is arranged so that mobility decreases along it:



`auto_ladder_sqa_tuned(problem, replicas=R, mode=...)` builds the ladder from the problem's probed energy scale:

- β rungs are spaced *geometrically* from $\beta_{\min}$ (hot) to $\beta_{\max}$ (cold), so adjacent-rung energy distributions overlap roughly uniformly along the ladder;
- Γ rungs are spaced *inverse-geometrically* — high Γ where β is small, low Γ where β is large — so every rung sits at a physically coherent point (thermal and quantum mobility rise and fall together).

6.2 The quantum swap rule, derived

The joint stationary measure is $\Pi(\{x_r\}) \propto \prod_r e^{-S_r(x_r)}$, where x_r denotes the entire Trotter configuration of replica r and

$$S_r(x) = \frac{\beta_r}{M} \sum_t E_{\text{cl}}(s^t) - J_{\perp}^{(r)} \sum_{t,i} s_i^t s_i^{t+1} \equiv \frac{\beta_r}{M} C(x) - J_{\perp}^{(r)} B(x), \quad (6.1)$$

with $C(x)$ the summed classical energy over slices and $B(x)$ the total imaginary-time *bond sum*. (The quantity B returns as the central observable of Chapter 8.)

A swap proposes exchanging the full configurations of rungs r and $r+1$. Detailed balance for Π gives the acceptance

$$A_{\text{swap}} = \min\left(1, \exp\left[-(S_r(x_{r+1}) + S_{r+1}(x_r) - S_r(x_r) - S_{r+1}(x_{r+1}))\right]\right). \quad (6.2)$$

This is precisely what the C++ core computes: it evaluates the four cross-actions $S_r(x_r)$, $S_{r+1}(x_{r+1})$, $S_r(x_{r+1})$, $S_{r+1}(x_r)$ from Eq. (6.1) and applies Eq. (6.2). Expanding via Eq. (6.1), the exponent splits into a thermal part and a quantum part:

$$\Delta_{\text{swap}} = \frac{\beta_r - \beta_{r+1}}{M} (C(x_{r+1}) - C(x_r)) - (J_{\perp}^{(r)} - J_{\perp}^{(r+1)}) (B(x_{r+1}) - B(x_r)). \quad (6.3)$$

The first term is the classical PT criterion Eq. (4.2) applied to slice-summed energies; the second exchanges configurations according to their quantum “kinetic” content. When all replicas *share* one $J_{\perp}$ — as in the optimal-schedule mode of Section 8.4.3 — the second term vanishes identically and the swap reduces to a purely classical criterion.

6.3 The algorithm

`SQAParallelTemperingAnnealer.run()` iterates steps epochs; in each epoch every replica performs its local SQA updates (`sweeps_per_step` single-spin sweeps, `worldline_sweeps` worldline sweeps, optionally `cluster_sweeps` SW sweeps — all with the rules of Section 5.3 at that replica’s own $(\beta_r, \Gamma_r, J_{\perp}^{(r)})$), and every `swap_interval` epochs a sweep of adjacent-pair swaps is attempted with Eq. (6.2). Replicas are OpenMP-parallel; the per-epoch swap acceptance fraction is recorded in `swap_acceptance_trace`.

Unlike annealing, the ladder is *static*: no schedule ramps during the run. Optimisation progress comes from configurations drifting down the ladder. The best classical projection (best single slice, any replica, any epoch) is tracked exactly as in SQA.

6.4 Usage

6.4.1 One call

```
from qanneal import solve

result = solve(
    ising,
    method="sqapt",
    replicas=8,           # ladder rungs
    reads=8,
    trotter_slices=24,
    sweeps_per_beta=50,  # local sweeps per epoch
    worldline_sweeps=4,
    cluster_sweeps=1,    # SQAPT+SW
```



```

pt_steps=70,          # epochs
swap_interval=1,
seed=3,
)

```

If no schedule/ladder is supplied, `solve()` calls `auto_ladder_sqa_tuned` internally. Explicit ladders:

```

from qanneal import auto_ladder_sqa_tuned, SQAParallelTemperingAnnealer

ladder = auto_ladder_sqa_tuned(ising, replicas=8, mode="balanced")
ann = SQAParallelTemperingAnnealer(
    ising, ladder.betas, ladder.gammas, trotter_slices=24)
res = ann.run(sweeps_per_step=50, worldline_sweeps=4,
              steps=80, swap_interval=1, cluster_sweeps=1)
print(res.best_energy)
print(res.swap_acceptance_trace[-5:]) # aim for ~0.2-0.5

```

You may also hand explicit ladders to `solve()` via `pt_betas=[...]`, `pt_gammas=[...]` (they must pair up).

6.4.2 Reading the diagnostics

- `swap_acceptance_trace` — mean accepted-swap fraction per epoch. Healthy: 0.2–0.5. Too low $\Rightarrow$ add rungs or narrow the ladder; too high $\Rightarrow$ rungs redundant.
- `final_energies` — final projected energy of each rung; should be ordered roughly hot-to-cold. A cold rung stuck far above the best energy signals insufficient epochs or a mobility bottleneck mid-ladder.
- `average_energy_trace` — ladder-mean energy per epoch; plateaus indicate equilibration.

6.5 Choosing sqapt

- **Use it** for frustrated, glassy, multi-modal problems — anything where single-trajectory annealing shows large read-to-read variance.
- Ladder size `replicas = 8` is a solid default; scale up with problem hardness, not with n .
- `cluster_sweeps=1` (SQAPT+SW) is the strongest general-purpose configuration in the library.
- Prefer `sqa` when the problem is easy enough that restarts suffice — SQAPT's ladder overhead (R full Trotter systems) is then wasted.

The combination of `sqapt` with the adaptive $J_{\perp}$ schedule — a shared, susceptibility-paced Trotter coupling ramp across the whole ladder — is covered in Section 8.4.3.

Continuous-Time PIMC (ctpimc)

The fourth engine removes the Trotter discretisation altogether: it samples the path integral directly in *continuous* imaginary time, representing each spin as a piecewise-constant *worldline* on the circle $[0, \beta)$. There is no Trotter error, and the cluster updates it employs mix particularly well in the strongly quantum (large- Γ) regime. The implementation builds on D-Wave's localPIMC code (Apache-2.0, credited in NOTICE).

7.1 From $M \rightarrow \infty$ to worldlines

Recall the discrete action Eq. (5.8). Take $M \rightarrow \infty$ at fixed β : the slice index becomes a continuous imaginary time $\tau \in [0, \beta)$ and each site's spin becomes a function $s_i(\tau) \in \{-1, +1\}$, periodic in τ , changing value at a finite set of *kinks* (domain walls in time). In this limit:

- The classical term becomes an integral, $\frac{\beta}{M} \sum_t E_{\text{cl}}(s^t) \rightarrow \int_0^\beta E_{\text{cl}}(s(\tau)) d\tau$.
- The Trotter bonds become a *kink fugacity*: each kink on a worldline carries a weight $\propto \Gamma d\tau$, i.e. in the absence of couplings the kinks of site i form a *Poisson process* of rate Γ on the time circle.

$$Z = \sum_{\text{worldline configs}} \Gamma^K \exp \left[- \int_0^\beta E_{\text{cl}}(s(\tau)) d\tau \right], \quad (7.1)$$

where K is the total number of kinks and the “sum” is over all piecewise-constant periodic $\{s_i(\tau)\}$.

Large Γ favours many kinks (strong quantum fluctuation, rapidly flapping worldlines); $\Gamma \rightarrow 0$ suppresses kinks and worldlines freeze into constants — the classical configurations.

7.2 Cluster updates on worldlines

The engine updates worldlines with Swendsen–Wang-style cluster moves over worldline *segments* (the intervals between kinks): segments of neighbouring sites are stochastically bound together according to the couplings acting during their overlap in time, and bound clusters are flipped as a unit. Kinks are created and removed in the process, so both the spin values and the kink structure are resampled. This is the continuous-time analogue of the discrete SW move of Section 5.3.3, without any discretisation scale.

Two knobs shape the proposals:

- `qubits_per_update` — how many sites participate in one cluster proposal (1 = single-site updates);
- `qubits_per_chain` — chain length for the multi-site chain updates used by the lattice mode.

7.3 General mode

For arbitrary `DenseIsing`/`SparseIsing` problems the annealer follows a standard (β, Γ) schedule. At each schedule step the couplings are rescaled to the dimensionless combinations the sampler consumes,

$$\beta J_{ij}, \quad \beta h_i, \quad \beta \Gamma, \quad (7.2)$$

then `sweeps_per_beta` cluster sweeps are run and the classical projection of the current worldlines is recorded. Reported energies are converted back to the *original* Ising units (not scaled by β).

```
import numpy as np
from qanneal import DenseIsing, SQASchedule, CTPIMCAnealer

h = np.array([0.2, -0.3, 0.1])
J = np.array([[0.0, -1.0, 0.2],
              [-1.0, 0.0, 0.5],
              [ 0.2, 0.5, 0.0]])
ising = DenseIsing(h, J)

schedule = SQASchedule.from_vectors(
    betas = np.linspace(0.5, 4.0, 40).tolist(),
    gammas = np.linspace(2.0, 0.05, 40).tolist(),
)
annealer = CTPIMCAnealer(ising, schedule,
                          qubits_per_update=1, qubits_per_chain=1)
res = annealer.run(sweeps_per_beta=50, reads=4)
print(res.best_energy)
```

Through the high-level interface:

```
result = solve(ising, method="ctpimc", reads=8,
               ctpimc_qubits_per_update=1)
```

Γ must stay positive

The continuous-time weight is singular at $\Gamma = 0$ (zero kink rate: the sampler cannot move). End the schedule at a small positive value (e.g. 10^{-3}), never exactly zero. When building tuned schedules for CT-PIMC, pass `gamma_end_scale=0.04` to `auto_schedule_sqa_tuned` so the final Γ is near-zero for a clean classical projection, yet finite.

7.4 Lattice mode

A second constructor reproduces the cylindrical-lattice experiments the underlying sampler was designed for (triangular and square-octagonal geometries):

```
from qanneal import CTPIMCAnealer

annealer = CTPIMCAnealer(
    Lperiodic=12,          # lattice period (multiple of 6)
    inv_temp_over_J=2.0,   # beta / J
    gamma_over_J=0.6,      # Gamma / J
    initial_condition=0,    # 0 random, 1 ferro, -1 antiferro
    qubits_per_update=1,
    qubits_per_chain=1,
)
res = annealer.run(sweeps_per_beta=32768)
```

Here temperatures and fields are expressed in units of the lattice coupling J , and reported energies are likewise in units of J .

7.5 CT-PIMC versus SQA

	SQA (Chapter 5)	CT-PIMC
Imaginary time	M discrete slices	continuous
Systematic error	$O(\beta^2 \Gamma^2 / M)$ Trotter error	none
Elementary move	spin / worldline / SW segment	worldline-segment clusters
High- Γ mixing	needs large M	natural regime
Replica / PT structure	yes (sqapt)	no
Sweep-level observers	yes	no

Use CT-PIMC when you want discretisation-free sampling closest in spirit to physical-annealer statistics; use SQA/SQAPT when you need the replica machinery, adaptive $J_{\perp}$ schedules, or detailed observer traces. Neither dominates the other on all problems.

Part III

The Optimal Adaptive Schedule

The Optimal Adaptive $J_{\perp}$ Schedule

Standard SQA follows a pre-designed $\Gamma(t)$ ramp fixed before the run begins. Such a schedule cannot react to the system it is driving: it may rush through the quantum critical region (exciting the system and spoiling the solution) while wasting steps in regions where nothing interesting happens. This chapter derives qanneal's alternative: a schedule that *measures* how critical the simulated system currently is — through the bond susceptibility χ_B — and paces the Trotter coupling $J_{\perp}$ accordingly. It is available for sqa and sqapt (with or without SW cluster moves) via `schedule_type="optimal"`.

8.1 Local adiabaticity: the Roland–Cerf condition

The adiabatic theorem (Section 5.1) bounds the *total* runtime by the worst-case gap, Eq. (5.2). Roland and Cerf [8] sharpened this into a *local* condition: rather than sweeping the control parameter uniformly, demand that the *instantaneous* transition probability stay uniformly small. For a control parameter $\lambda(t)$ this reads

$$\left| \frac{d\lambda}{dt} \right| \leq \varepsilon \frac{\Delta_{\text{gap}}^2(\lambda)}{|\langle 1 | \partial_{\lambda} \hat{H} | 0 \rangle|} \sim \varepsilon \Delta_{\text{gap}}^2(\lambda), \quad (8.1)$$

with $\varepsilon \ll 1$ a fixed error budget. The schedule then *slows down exactly where the gap is small* — near the quantum critical point — and accelerates where the gap is large. For the archetypal Grover problem this converts the naive $O(1/\Delta_{\text{min}}^2)$ total time into the provably optimal $O(1/\Delta_{\text{min}})$; generically it redistributes a fixed time budget in the provably best way.

8.1.1 From the gap to a measurable quantity

Eq. (8.1) is not directly usable: the gap of an n -spin system is unknown. But near a second-order quantum phase transition, *finite-size scaling* links the gap to correlation functions that Monte Carlo can measure. With correlation length ξ , dynamical exponent z and anomalous dimension η :

$$\Delta_{\text{gap}} \sim \xi^{-z}, \quad \chi \sim \xi^{2-\eta}, \quad (8.2)$$

where χ is the susceptibility of the operator driving the transition. Eliminating ξ :

$$\Delta_{\text{gap}}^2 \sim \chi^{-2z/(2-\eta)}. \quad (8.3)$$

In the Suzuki–Trotter representation the natural control parameter is the Trotter coupling $J_{\perp}$ (Eq. (5.7)) and the natural susceptibility is that of the quantity $J_{\perp}$ multiplies in the action Eq. (6.1): the total imaginary-time **bond sum**

$$B = \sum_{t=1}^M \sum_{i=1}^n s_i^t s_i^{t+1}, \quad \chi_B = \text{Var}(B) = \langle B^2 \rangle - \langle B \rangle^2. \quad (8.4)$$

B plays the role of the conjugate “order parameter density” of the transverse-field-driven transition; χ_B is cheap to track (see Section 8.3) and peaks near the quantum transition. Substituting into Eq. (8.1) and absorbing an extra $+\frac{1}{2}$ in the exponent that accounts for the statistical width of the finite-sample variance estimator, the discrete update per adaptive step becomes

$$\Delta J_\perp = \tilde{\epsilon} \cdot \chi_B^{-\alpha}, \quad \alpha = \frac{z}{2-\eta} + \frac{1}{2}. \quad (8.5)$$

For the 1-D quantum Ising universality class ($z = 1, \eta = \frac{1}{4}$): $\alpha = \frac{1}{7/4} + \frac{1}{2} = \frac{4}{7} + \frac{1}{2} = \frac{15}{14} \approx 1.071$ — the library default (`optimal_alpha`). For the 2-D class use $\alpha = 7/4$; for mean-field (Sherrington–Kirkpatrick-like) problems $\alpha \approx 1$ is a reasonable, if tentative, choice. In practice $15/14$ is a robust default even when the true universality class is unknown, because the calibration below makes the schedule’s overall pace independent of moderate errors in α .

For this model χ_B is large near the quantum transition and collapses toward zero deep in the ordered (classical) phase — it peaks rather than diverging. Through Eq. (8.5) the schedule therefore takes small steps (χ_B large) in the critical window and large steps (χ_B small) once the system has ordered: a characteristic **fast–slow–fast** trajectory in $J_\perp$, dwelling exactly where diabatic errors would otherwise be created.

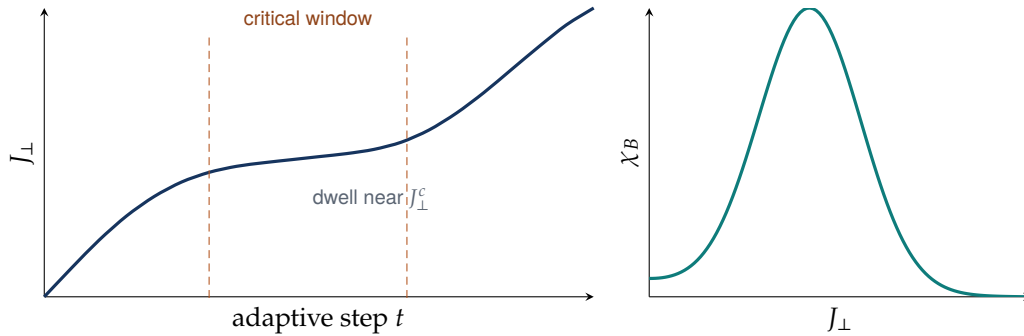


Figure 8.1: Schematic of a healthy adaptive run. Left: the realised $J_\perp(t)$ is fast–slow–fast, dwelling in the critical window. Right: the bond susceptibility $\chi_B(J_\perp)$ that drives the pacing, peaking near the quantum transition and collapsing in the ordered phase.

8.2 Budget calibration: fixing the scale of $\tilde{\epsilon}$

Eq. (8.5) separates cleanly into *shape* and *scale*: the factor $\chi_B^{-\alpha}$ dictates *where* the schedule is slow or fast, while $\tilde{\epsilon}$ only sets the overall pace. Choosing $\tilde{\epsilon}$ by hand is fragile: since χ_B ’s magnitude

grows with n , M and the coupling distribution, a value that works for one instance can make another instance's schedule either leap across its whole range in a few steps or — the more insidious failure — *freeze*, advancing $J_{\perp}$ so slowly that the run effectively anneals at constant coupling. Neither failure announces itself: the run completes and returns an energy.

qanneal therefore ties $\tilde{\epsilon}$ to the *step budget*. Treat Eq. (8.5) as an ODE in continuous step-time, $\frac{dJ}{dt} = \tilde{\epsilon} \chi_B(J)^{-\alpha}$, and separate variables: $dt = \tilde{\epsilon}^{-1} \chi_B(J)^{\alpha} dJ$. Requiring the traversal of $[J_0, J_{\text{end}}]$ to take *exactly* $T = \text{num_steps}$ steps,

$$T = \frac{1}{\tilde{\epsilon}} \int_{J_0}^{J_{\text{end}}} \chi_B(J)^{\alpha} dJ \quad (8.6)$$

$$\tilde{\epsilon} = \frac{1}{T} \int_{J_0}^{J_{\text{end}}} \chi_B(J)^{\alpha} dJ. \quad (8.7)$$

Sign of the exponent

Note the *positive* power $\chi_B^{+\alpha}$ inside the integral and the $1/T$ prefactor. The superficially plausible alternative $\tilde{\epsilon} = \Delta J_{\perp} / \sum_m \chi_B(J_m)^{-\alpha}$ (negative power, no $1/T$) is *not* budget-correct when χ_B varies along the range — it under- or over-shoots and can re-freeze the schedule. Eq. (8.7) follows uniquely from the separation of variables above.

8.2.1 The pilot pre-pass

$\chi_B(J)$ is not known in advance, so when `eps_tilde` ≤ 0 (the default `optimal_eps_tilde=0.0` requests exactly this) `run_optimal` first runs a short *pilot*: it measures χ_B at `calib_probes` (default 12) values of $J_{\perp}$ spanning $[J_0, J_{\text{end}}]$, using `calib_sweeps` (default 10) sweeps per probe with the *same* sampler the real run will use, on scratch states so the production run is unbiased. The integral in Eq. (8.7) is then evaluated by the midpoint rule over the probe grid.

8.2.2 The inverse-CDF trajectory

One more robustness step: rather than integrating Eq. (8.5) *online* with the noisy within-run χ_B (whose fluctuations can compound — a run that momentarily orders faster than the pilot predicted sees a small χ_B , takes a huge step, and skips the rest of the range), the entire trajectory is *precomputed* from the pilot profile. Define the cumulative “adiabatic cost”

$$G(J) = \int_{J_0}^J \chi_B(J')^{\alpha} dJ', \quad J^{(t)} = G^{-1}\left(\frac{t}{T-1} G(J_{\text{end}})\right), \quad t = 0, \dots, T-1. \quad (8.8)$$

Each step is placed at an equal increment of adiabatic cost. By construction — independent of the shape or scale of χ_B — the schedule (a) consumes exactly T steps, (b) reaches J_{end} , and (c) concentrates its steps where χ_B is large. The online χ_B is still measured at every step and recorded in `chi_B_trace` as a diagnostic (and, in the debug CSV, as `chi_B_online`).

A *positive* `eps_tilde` bypasses calibration entirely and drives the legacy online update $J_{\perp} += \tilde{\epsilon} \chi_B^{-\alpha}$ with that fixed scale — retained for reproducibility and for controlled experiments, not recommended for production.

8.2.3 Fairness of the step budget

The adaptive run *always* executes all `num_steps` steps; if $J_{\perp}$ saturates at J_{end} early it keeps sweeping at the endpoint rather than stopping. Total sweep effort is therefore identical to a standard-schedule run of the same length — any quality difference is attributable to the *placement* of the steps, not to extra compute.

8.3 Estimating χ_B without overhead

8.3.1 Incremental bond tracking

Recomputing B (Eq. (8.4)) from scratch costs $O(nM)$; instead, every move type updates it incrementally in $O(1)$ per flip:

Move	Bond-sum change ΔB
Metropolis flip of s_i^t	$\Delta B = -2 s_i^t (s_i^{t-1} + s_i^{t+1})$ evaluated with the <i>old</i> spin value — the two touched time-bonds reverse.
SW cluster flip	$\Delta B = -2 \sum_{\text{boundary bonds}} s s'$ — only the bonds joining the cluster to unflipped spins change; interior bonds are invariant.
Worldline flip	$\Delta B = 0$ — every bond of the worldline is a product of two flipped spins.

After each full sweep the current B is appended to the step's sample buffer, so χ_B estimation adds essentially zero cost to the run.

8.3.2 Pooled variance across sweeps and replicas

Each adaptive step yields $R \times S$ samples of B (R replicas, S sweeps within the step), pooled into one estimate $\chi_B = \widehat{\text{Var}}(B)$. The pooling is statistically well-behaved in both regimes:

- *Near criticality*, the autocorrelation time $\tau_{\text{int}} \gg S$, so each replica's S samples are nearly identical and the pooled variance collapses to the *cross-replica* variance — exactly the honest estimator for correlated chains.
- *Away from criticality*, $\tau_{\text{int}} \ll S$ and the pooled estimate enjoys $\sim RS$ effectively independent samples.

This is the reason `replicas` ≥ 4 is recommended whenever `schedule_type="optimal"` is used.

8.3.3 Single-replica guard and the χ_B floor

Two safeguards prevent pathological steps:

$R = 1$ **guard.** With a single replica near criticality all within-step samples coincide, the naive variance $\rightarrow 0$, and Eq. (8.5) would take its *maximum* step precisely where it must be smallest. The implementation therefore uses $\chi_B = \max(\widehat{\text{Var}}(B), nM[1 - (B/nM)^2])$, the second term being the single-bond variance extrapolated over the lattice — a strictly positive lower bound near criticality.

Floor. Deep in the ordered phase $\chi_B \rightarrow 0$ and $\chi_B^{-\alpha}$ blows up; a floor $\chi_B \geq 10^{-4} \max_J \chi_B^{(\text{pilot})}$ caps the step size. The debug CSV records where the floor activates (floor_hit); in a healthy run this is only the deeply ordered tail, never the critical window.

8.4 Endpoints, temperature, and the two-phase SQA protocol

8.4.1 Choosing J_0 and J_{end}

All endpoints derive from the problem's own coupling scale. Let

$$J_{\text{rms}} = \sqrt{\text{mean}(J_{ij}^2)} \quad \text{over the non-zero couplings} \quad (8.9)$$

(computable via `j_rms_from_problem(problem)`). The quantum critical coupling scales as $J_{\perp}^c \sim J_{\text{rms}}$, so:

- `j_perp_start` — from the tuned standard schedule's $(\beta, \Gamma_{\text{start}})$ through Eq. (5.7); small (quantum end).
- `j_perp_end` — default $\max(J_0, 5J_{\text{rms}})$, comfortably past the transition (classical end).
- β — for SQA, $\beta = \max(4/J_{\text{rms}}, 1)$, i.e. cold enough that $\beta J_{\text{rms}} \geq 4$: thermal fluctuations must not wash out the order the $J_{\perp}$ ramp is trying to establish.

The helper returns all three at once:

```
from qanneal import optimal_j_perp_params, j_rms_from_problem
beta, jp_start, jp_end = optimal_j_perp_params(problem, trotter_slices=32)
jr = j_rms_from_problem(problem)
```

When can the adaptive schedule help at all?

The schedule needs a quantum range to act on: it helps when $J_{\text{rms}} \gtrsim j_{\text{perp_start}}$ (in the auto-tuned setting, roughly $J_{\text{rms}} \gtrsim 3$), so that the run genuinely starts on the disordered side of the transition. If J_{rms} is far below J_0 the system is already effectively classical at step one, $\chi_B \approx 0$ everywhere, and no schedule — adaptive or not — has anything to optimise. Check a new problem class with `scripts/sanity_schedule.py`, which classifies instances as PASS or DEGENERATE before you spend compute.

Always pass an explicit `j_perp_end` in scripted runs

The C++ `run_optimal` accepts a sentinel `j_perp_end<=0` that resolves to the coldest replica's Trotter coupling. If that resolves *below* `j_perp_start` the schedule has no range; the code then installs a large fallback ceiling and prints a cerr warning. Relying on the sentinel makes runs silently sensitive to ladder details — pass an explicit `j_perp_end` (e.g. $5J_{\text{rms}}$) in

anything scripted.

8.4.2 Two-phase protocol for SQA

Plain SQA under the adaptive schedule holds a *single* fixed β — it has no thermal annealing at all, which measurably hurts it. The SQA `run_optimal` therefore runs two phases, split by `beta_ramp_fraction` (default 0.3):

1. **Thermal ramp** — for the first $\lceil 0.3 \cdot \text{num_steps} \rceil$ steps, β ramps from `beta_ramp_start` (default 0.1) up to the cold target $\beta = \max(4/J_{\text{rms}}, 1)$ at *fixed* $J_{\perp} = J_0$: a conventional thermal anneal that settles the slices into the disordered quantum state at the right temperature.
2. **Adaptive ramp** — β then stays fixed while $J_{\perp}$ follows the calibrated inverse-CDF trajectory Eq. (8.8) over the remaining steps.

SQAPT does *not* use phase 1: its (β, Γ) ladder plus replica swaps already supply thermal mobility.

8.4.3 SQAPT under a shared $J_{\perp}$

In the SQAPT variant, *all* rungs share the single evolving $J_{\perp}$; each keeps its own (β_r, Γ_r) from the ladder. Two consequences, both derived earlier:

- In the swap exponent Eq. (6.3) the quantum term carries the factor $J_{\perp}^{(r)} - J_{\perp}^{(r+1)} = 0$: **swaps reduce to the purely classical criterion** on slice-summed energies.
- χ_B is estimated by pooling B -samples across all rungs, exactly as in Section 8.3.

8.5 Usage

8.5.1 High-level

```
from qanneal import solve

result = solve(
    problem,
    method="sqa",           # or "sqapt"
    schedule_type="optimal",
    reads=15,
    replicas=4,             # >=4 for a reliable chi_B estimate
    optimal_eps_tilde=0.0,  # 0.0 = per-instance budget calibration
    optimal_num_steps=150,
    # Optional overrides (auto-computed from the problem scale if omitted):
    # optimal_j_perp_end=25.0,      # default max(j_perp_start, 5*j_rms)
    # optimal_beta_ramp_fraction=0.3, # SQA two-phase split
    # optimal_alpha=15/14,
    # optimal_debug_csv="schedule.csv" # per-step J_perp / chi_B diagnostics
)
print(result.best_energy, result.best_sample)
```

For SQAPT with SW cluster moves:

```
result = solve(
    problem,
    method="sqapt",
    schedule_type="optimal",
    replicas=8, reads=15,
    trotter_slices=32,
    worldline_sweeps=3,
    cluster_sweeps=1,          # >0 = SQAPT+SW
    swap_interval=1,
    optimal_eps_tilde=0.0,
    optimal_num_steps=150,
)
```

8.5.2 Low-level, with full diagnostics

```
from qanneal import SQAAnnealer, SQASchedule

dummy = SQASchedule.from_vectors([1.0], [1.0]) # construction only
ann = SQAAnnealer(ising, dummy, trotter_slices=32, replicas=4)

result = ann.run_optimal(
    beta=1.0,                # phase-2 (cold) inverse temperature
    j_perp_start=0.1, j_perp_end=25.0,
    eps_tilde=0.0,           # <=0 -> budget calibration
    alpha=15/14, num_steps=150, sweeps_per_step=20,
    worldline_sweeps=3, cluster_sweeps=0,
    calib_probes=12, calib_sweeps=10, # pilot grid
    beta_ramp_fraction=0.3,        # two-phase thermal ramp
)
result.j_perp_trace          # realised  $J_{\perp}(t)$ 
result.chi_B_trace          # online  $\chi_B(t)$ 
result.calibrated_eps_tilde # the eps from Eq. (calibration)
result.final_j_perp         #  $J_{\perp}$  actually reached
result.best_state, result.best_energy
```

The SQAPT twin is `SQAParallelTemperingAnnealer.run_optimal` (full signature in Chapter 10), with identical calibration semantics and no two-phase ramp.

8.6 Diagnosing a run

Plot the three traces after any adaptive run:

```
import numpy as np, matplotlib.pyplot as plt

t = np.arange(len(result.j_perp_trace))
```

```

dj = np.diff(result.j_perp_trace)

fig, axs = plt.subplots(3, 1, figsize=(8, 9), sharex=True)
axs[0].plot(t, result.j_perp_trace); axs[0].set_ylabel("J_perp")
axs[1].plot(t[:-1], dj); axs[1].set_ylabel("step size dJ_perp")
axs[2].plot(t, result.energy_trace); axs[2].set_ylabel("energy")
axs[2].set_xlabel("adaptive step")
plt.suptitle("Optimal schedule diagnostics"); plt.tight_layout(); plt.show()

```

A **healthy run** shows (Fig. 8.1): fast early movement, a pronounced dwell (small $\Delta J_{\perp}$) in the critical window near $J_{\perp} \approx J_{\text{rms}}$, fast movement again past the transition, the endpoint reached within budget, and the steepest energy descent during the dwell.

Warning signs:

- *Flat or linear $J_{\perp}(t)$ with no dwell* — the schedule is frozen or degenerate. Check that $J_{\text{end}} > J_0$ (look for the cerr fallback warning) and that the problem is in the favourable regime ($J_{\text{rms}} \gtrsim J_0$).
- *Constant $\Delta J_{\perp}$* — χ_B estimate too noisy to shape the trajectory; raise replicas and/or sweeps_per_step.
- *final_j_perp far below resolved_j_perp_end* on a non-degenerate instance — calibration failed; inspect the debug CSV (columns phase, step_index, j_perp, chi_B, delta_j, eps_tilde, alpha, floor_hit, chi_B_online; phase=calib rows are pilot probes). The helper scripts/plot_schedule_trajectory.py renders it.

Pre-flight habit

Before any long or large- n run, spend a minute on a small instance of the same problem class: run with `optimal_debug_csv` set, plot the trajectory, and confirm the fast-slow-fast shape. `python scripts/sanity_schedule.py -n 40 -steps 150 -plot` automates exactly this check and prints PASS / DEGENERATE / FAIL per instance.

8.7 Attribution: schedule effect vs. cluster moves

The adaptive *schedule* and SW *cluster sweeps* are independent mechanisms that both help on hard instances. When evaluating the schedule on your own problems, keep them separate or you will misattribute gains:

Comparison	cluster_sweeps	Isolates
sqapt-std vs sqapt-opt	0	the schedule effect alone
sqapt+SW-std vs sqapt+SW-opt	> 0	schedule $\times$ cluster interaction

Only credit “the optimal schedule” if the first comparison reproducibly favours it; if only the +SW pair improves, the gain belongs to the interaction and should be framed as such.

Part IV

Using the Library

Installation and Building

9.1 Requirements

Component	Requirement	Role
Python	≥ 3.11	front end
NumPy	any recent	array interchange
C++ compiler	C++17 (GCC ≥ 9 , Clang ≥ 10 , MSVC 2019+)	core engines
CMake	≥ 3.18	build system (via scikit-build)
OpenMP	optional	parallel replicas/slices
MPI + mpi4py	optional	multi-node runs
matplotlib, tqdm	optional	plots, progress bars
dimod, networkx	optional	BQM / graph problem inputs

9.2 Installing the Python package

From the repository root:

```
# Standard install (recommended)
python -m pip install . --no-build-isolation

# Editable install for development
python -m pip install -e . --no-build-isolation

# Convenience scripts
./setup.sh      # macOS / Linux
setup.bat      # Windows cmd
./setup.ps1    # Windows PowerShell
```

The build compiles the C++17 core and the pybind11 extension (`qanneal._qanneal`) in place. Verify:

```
python -c "import qanneal; print(qanneal.__version__)" # 0.6.1
```

9.3 Building the C++ library directly

For C++-only use, tests, or the MPI example:

```
cmake -S . -B build \
  -DQANNEAL_ENABLE_OPENMP=ON \
  -DQANNEAL_ENABLE_MPI=OFF \
  -DQANNEAL_BUILD_TESTS=ON \
  -DCMAKE_BUILD_TYPE=Release
cmake --build build -j
ctest --test-dir build
```

CMake flag	Default	Effect
QANNEAL_ENABLE_OPENMP	ON	Parallel loops in Replica/SQA/SQAPT annealers.
QANNEAL_ENABLE_MPI	OFF	Distributed run_replica_anneal() and the MPI example.
QANNEAL_BUILD_TESTS	ON	C++ unit tests (ctest).
QANNEAL_BUILD_PYTHON	OFF*	pybind11 extension (*forced ON by scikit-build during pip installs).

9.4 Platform notes

Linux Everything works out of the box with GCC; OpenMP is detected automatically.

macOS Apple's Clang ships without OpenMP. Install it and rebuild: `brew install libomp`. Without it the build succeeds but runs single-threaded.

Windows Use the provided `setup.bat/setup.ps1`; MSVC 2019 or newer.

Clusters Build in the same environment (compiler + Python) you will run in; see [Chapter 12](#) for launcher and SLURM recipes.

Stale extensions

`ModuleNotFoundError: qanneal._qanneal` almost always means a stale or mismatched build — reinstall from the repository root with the command above, and avoid mixing an installed site-packages copy with a source-tree `PYTHONPATH` overlay.

Python API Reference

Complete reference for every public class and function of qanneal 0.6.1. Anything documented here is importable directly from the top-level package: `from qanneal import ...`

10.1 The high-level solver: `solve()`

`solve(problem, method="sqa", **kwargs) → SolveResult`

One-call interface: accepts any supported problem type, auto-selects and tunes a schedule, runs reads independent runs, returns the best result.

10.1.1 Accepted problem types

The problem argument is auto-detected:

Type	Interpretation
DenseIsing, SparseIsing	used directly
QUBO	converted via <code>to_ising()</code>
<code>np.ndarray (n × n)</code>	QUBO matrix, Eq. (2.2)
<code>dict{(i,j): float}</code>	sparse QUBO dictionary
<code>list[(i, j, float)]</code>	sparse QUBO entry list
<code>dimod.BinaryQuadraticModel</code>	BINARY or SPIN vartype (requires dimod)
<code>networkx.Graph</code>	node attr bias, edge attr weight (requires networkx)

For dict/list inputs, pass `n=...` if the variable count cannot be inferred. For BQM/graph inputs, `result.var_order` maps positions back to your original variable labels.

10.1.2 Common parameters (all methods)

Parameter	Default	Meaning
method	"sqa"	"sa", "sqa", "sqapt", "ctpimc".
reads	1	Independent runs; best over all reads is returned.
sweeps_per_beta	20	Metropolis sweeps per schedule step (also serves as sweeps_per_step in optimal mode).
schedule	None	Schedule object; auto-tuned per method if None.
seed	None	RNG seed; None = random.
progress	True	tqdm progress bar.
backend	"cpu"	Compute backend.
n	None	Size hint for dict/list inputs.
return_bits	False	Return {0, 1} bits instead of {-1, +1} spins.

10.1.3 SQA / SQAPT parameters

Parameter	Default	Meaning
trotter_slices	32	Imaginary-time slices M (Section 5.2).
replicas	1	Parallel Trotter systems (sqa) or ladder rungs (sqapt).
worldline_sweeps	5	Whole-worldline flips per step (Eq. (5.10)).
cluster_sweeps	0	Swendsen–Wang segment flips per step; > 0 enables +SW.
continuous_time_slices	0	If > 0 , overrides M to approximate continuous time.

10.1.4 SQAPT-only parameters

Parameter	Default	Meaning
pt_steps	50	Local-update + swap epochs (standard schedule).
swap_interval	1	Attempt swaps every N epochs.
pt_betas	None	Explicit β ladder (overrides schedule).
pt_gammas	None	Explicit Γ ladder (must pair with pt_betas).

10.1.5 CT-PIMC-only parameters

Parameter	Default	Meaning
ctpimc_qubits_per_update	1	Spins per cluster proposal.
ctpimc_qubits_per_chain	1	Chain length for lattice-mode updates.

10.1.6 Optimal-schedule parameters (schedule_type="optimal")

Applicable to sqa and sqapt; theory in Chapter 8.

Parameter	Default	Meaning
schedule_type	"standard"	"optimal" activates the adaptive $J_{\perp}$ schedule.
optimal_eps_tilde	0.0	≤ 0 = per-instance budget calibration (recommended); > 0 = legacy fixed online scale.
optimal_alpha	15/14	Universality exponent α (Eq. (8.5)).
optimal_num_steps	100	Adaptive step budget T .
optimal_j_perp_start	auto	J_0 ; from the tuned schedule's quantum end.
optimal_j_perp_end	auto	J_{end} ; default $\max(J_0, 5J_{\text{rms}})$.
optimal_beta	auto	Fixed β ; SQA default $\max(4/J_{\text{rms}}, 1)$.
optimal_beta_ramp_fraction	0.3	SQA two-phase split (Section 8.4).
optimal_debug_csv	""	Path for the per-step diagnostic CSV.

10.1.7 SolveResult

Field	Type	Meaning
method	str	Method used.
samples	list[np.ndarray]	Best spin/bit vector from each read.
energies	list[float]	Best energy from each read.
best_sample	np.ndarray	Global best configuration.
best_energy	float	Global best energy.
trace	list[float]	Energy trace from the first read.
var_order	list	Variable ordering (BQM/graph inputs).
return_bits	bool	Whether samples are bits or spins.

Manual spin→bit conversion: `bits = ((result.best_sample + 1) // 2).astype(int)`.

10.2 Model classes

10.2.1 DenseIsing

Denselsing(h, J, c=0.0)

Fully-connected Ising model, dense $n \times n$ storage. Energy: Eq. (2.1).

Argument	Type	Meaning
h	ndarray (n,)	Local fields; $h_i > 0$ pushes spin i toward -1 .
J	ndarray (n,n)	Symmetric coupling matrix; upper triangle used; ferromagnetic < 0 , antiferromagnetic > 0 .
c	float	Constant offset (energy bookkeeping only).

Methods: `size()` → int; `energy(spins)` → float ($O(n^2)$); `delta_energy(spins, flip)` → float ($O(n)$, Eq. (2.10)); `fields()` and `couplings()` return the stored h and J .

10.2.2 SparseIsing and SparseEdge

SparseIsing(h, edges, n, c=0.0)

Edge-list Ising model; scales to $n \geq 10^5$. `energy()` is $O(|E|)$, `delta_energy()` is $O(\deg i)$.

`edges` is a `list[SparseEdge]`; `SparseEdge(i, j, value)` holds one coupling J_{ij} (one direction suffices, $i < j$; the model symmetrises).

```
from qanneal import SparseIsing, SparseEdge
import numpy as np

n = 5
edges = [SparseEdge(0, 1, 1.0), SparseEdge(1, 2, -0.5),
         SparseEdge(2, 3, 0.8), SparseEdge(3, 4, 1.2)]
ising = SparseIsing(np.zeros(n), edges, n)
```

10.2.3 QUBO

QUBO(Q) | QUBO(entries, n) | QUBO(bqm)

QUBO model, Eq. (2.2). Constructors: dense ndarray; sparse `list[(i,j,val)]` or `dict{(i,j): val}` with explicit `n`; or a dimod BQM (BINARY or SPIN, auto-converted).

Methods: `size()`; `to_ising()` $\rightarrow$ `DenseIsing`, implementing the exact conversion Eq. (2.6) (symmetrises Q , applies $x = (1 + s)/2$, carries the offset c).

10.3 Schedule classes and helpers

10.3.1 AnnealSchedule

AnnealSchedule.linear(beta_start, beta_end, steps)

Classical schedule: linear ramp of inverse temperatures. Attribute `betas`: `list[float]`.

10.3.2 SQASchedule

SQASchedule.from_vectors(betas, gammas)

Quantum schedule: paired (β, Γ) sequences (equal lengths). Attributes `betas`, `gammas`. Γ should start high and decay geometrically (see the warning in Chapter 5).

10.3.3 Fixed helpers

- `auto_schedule_sa(steps=50, beta_start=0.1, beta_end=4.0)  $\rightarrow$  AnnealSchedule` — fixed linear ramp.

- `auto_schedule_sqa(steps=50, beta_start=0.1, beta_end=4.0, gamma_start=5.0, gamma_end=0.01) → SQASchedule` — fixed ramp with geometric Γ decay.

10.3.4 Tuned helpers (recommended)

Mechanics in Sections 4.2.2 and 5.5.3.

`auto_schedule_sa_tuned(problem, mode="balanced", steps=None, probes=256, seed=1234, n=None, beta_end_sc`

Problem-adaptive SA schedule from the probed 75th-percentile $|\Delta E|$ and acceptance targets, Eq. (3.6).

`auto_schedule_sqa_tuned(..., gamma_end_scale=1.0)`

As above, additionally calibrating the Γ ramp ($\Gamma_{\text{start}} = m_{\Gamma}\Delta$, geometric decay). Set `gamma_end_scale=0.04` when the schedule feeds CT-PIMC.

`auto_ladder_sqa_tuned(problem, replicas=8, mode="balanced", probes=256, seed=1234, n=None)`

(β, Γ) ladder for SQAPT: geometric β rungs, inverse-geometric Γ rungs. Returns an `SQASchedule` whose entries are ladder points, not a time sequence.

10.3.5 Optimal-schedule helpers

`j_perp_from_beta_gamma(beta, gamma, trotter_slices=32) → float`

The Trotter coupling of Eq. (5.7): $J_{\perp} = \frac{1}{2} \ln(1/\tanh(\beta\Gamma/M))$.

`optimal_j_perp_params(problem, mode="balanced", trotter_slices=32, probes=256, seed=1234, n=None) → (beta,`

Problem-adaptive endpoints for `run_optimal`: derives $(\beta, \Gamma_{\text{start}}, \Gamma_{\text{end}})$ from the tuned-schedule probe and maps through Eq. (5.7); the returned $\beta = \max(4/J_{\text{rms}}, 1)$.

`j_rms_from_problem(problem) → float`

RMS of the non-zero couplings, $J_{\text{rms}} = \sqrt{\langle J_{ij}^2 \rangle}$; accepts `ndarray`, `DenseIsing`, `BQM` or `graph`.

10.4 Annealer classes

Low-level API; most users should call `solve()`. All annealers take an optional `backend="cpu"` argument and support `set_seed(int)`.

10.4.1 Annealer

`Annealer(hamiltonian, schedule); run(sweeps_per_beta, observer=None) → AnnealResult`

Single-chain simulated annealing (Chapter 4). Optional `MetricsObserver` / `StateTraceObserver`.

10.4.2 ReplicaAnnealer

ReplicaAnnealer(hamiltonian, schedule, replicas); run(sweeps_per_beta) → MultiAnnealResult

R independent SA chains, OpenMP-parallel (Section 4.3).

10.4.3 ParallelTemperingAnnealer

ParallelTemperingAnnealer(hamiltonian, betas); run(sweeps_per_step, steps, swap_interval=1) → ParallelTempe

Classical replica exchange with the swap rule Eq. (4.2) (Section 4.4).

10.4.4 SQAAnnealer

SQAAnnealer(hamiltonian, schedule, trotter_slices, replicas=1)

Suzuki–Trotter SQA (Chapter 5).

run(sweeps_per_beta, worldline_sweeps, cluster_sweeps=0, continuous_time_slices=0, observer=None) → SQAF

Standard (β, Γ) -schedule run with the moves of Section 5.3.

run_optimal(beta, j_perp_start, j_perp_end, eps_tilde, alpha=15/14, num_steps=100, sweeps_per_step=20, worldlin

Adaptive $J_{\perp}$ schedule (Chapter 8). $\text{eps_tilde} \leq 0$ triggers budget calibration + inverse-CDF trajectory; two-phase β ramp controlled by $\text{beta_ramp_fraction}$. Raises `ValueError` if $\text{num_steps} == 0$, if $\text{j_perp_end} \leq \text{j_perp_start}$, or if all sweep counts are zero.

10.4.5 SQAParallelTemperingAnnealer

SQAParallelTemperingAnnealer(hamiltonian, betas, gammas, trotter_slices)

SQA replica exchange on a (β, Γ) ladder (Chapter 6).

run(sweeps_per_step, worldline_sweeps, steps, swap_interval=1, cluster_sweeps=0, continuous_time_slices=0) –

Static-ladder run with the action-based swap rule Eq. (6.2).

run_optimal(num_steps, sweeps_per_step, worldline_sweeps, eps_tilde, alpha=15/14, j_perp_end=0.0, cluster_sw

Shared- $J_{\perp}$ adaptive schedule across the ladder (Section 8.4.3); same calibration semantics as the SQA version, no two-phase ramp. $\text{j_perp_end} = 0.0$ is a sentinel resolving to the coldest rung's coupling (emits a warning) — pass an explicit value in scripts. Raises `ValueError` for zero num_steps or swap_interval , or if all sweep counts are zero.

10.4.6 CTPIMCAnealer

CTPIMCAnealer(ising, schedule, qubits_per_update=1, qubits_per_chain=1) — general mode

Continuous-time PIMC for arbitrary Dense/Sparse Ising (Chapter 7).

CTPIMCAnealer(Lperiodic, inv_temp_over_J, gamma_over_J, initial_condition=0, qubits_per_update=1, qubits_per_chain=1)

Cylindrical-lattice experiments; Lperiodic must be a multiple of 6; initial_condition: 0 random, 1 ferro, -1 antiferro.

run(sweeps_per_beta, reads=1) → CTPIMCResult

Worldline cluster sweeps per schedule step; best over reads returned, trace averaged over reads. Γ must stay > 0 .

10.5 Result classes

Class	Produced by	Fields
AnnealResult	Annealer.run	best_state, best_energy, energy_trace
MultiAnnealResult	ReplicaAnnealer.run	global_best_state, global_best_energy, replicas, average_energy_trace, average_magnetization_trace
ParallelTemperingResult	ParallelTemperingAnnealer.run	best_state, best_energy, final_states, final_energies, average_energy_trace, swap_acceptance_trace
SQAResult	SQAAnnealer.run/run_optimal	best_state, best_energy, energy_trace, j_perp_trace, plus calibration fields (below)
SQAParallelTemperingResult	SQAParallelTemperingAnnealer	as ParallelTemperingResult + j_perp_trace + calibration fields
CTPIMCResult	CTPIMCAnealer.run	best_state, best_energy, energy_trace
State	(container)	spins (± 1 int8 list), size()

10.5.1 Adaptive-schedule fields on SQAResult / SQAParallelTemperingResult

Field	Meaning
j_perp_trace	Realised $J_{\perp}$ per adaptive step; empty for standard run(); monotone non-decreasing; <code>len == len(energy_trace)</code> .
chi_B_trace	Online χ_B measured at each step.
calibrated_eps_tilde	The $\tilde{\epsilon}$ from Eq. (8.7) (or the value you passed).
j_perp_start, resolved_j_perp_end	The endpoints the run actually used after sentinel/auto resolution.
final_j_perp	$J_{\perp}$ reached at the last step ($\approx$ resolved_j_perp_end when calibrated).

10.6 Observer classes

Observers hook into the annealing loop for diagnostics without touching the algorithm. Attaching a sweep-level observer to `SQAAnealer` disables its OpenMP fast path (observation is inherently sequential), so use them for analysis runs, not production sweeps.

Class	Engine	Captures
<code>MetricsObserver</code>	SA	energy_trace, magnetization_trace per step; clear() resets.
<code>StateTraceObserver(stride=1)</code>	SA	full state_trace plus energy_trace, step_trace, sweep_trace, beta_trace per recorded sweep.
<code>SQAMetricsObserver</code>	SQA	slice/replica-averaged energy_trace, magnetization_trace.
<code>SQAStateTraceObserver(stride=1)</code>	SQA	avg_energy_trace, replica_energy_trace, state_trace, beta_trace, gamma_trace, phase_trace (Slice/Worldline/Cluster), step_trace, replica_trace, sweep_trace; dimensions replicas, slices, spins.

10.7 Utility functions

```
from qanneal import magnetization, overlap

magnetization(spins)      # sum(s_i) / n
overlap(spins_a, spins_b) # sum(a_i * b_i) / n
```

Both accept list, `np.ndarray`, or a `State`. Magnetisation tracks global ordering during an anneal; overlap between two reads' solutions measures how reproducibly a problem's ground state is found (overlap ≈ 1 : same solution up to global flip).

Worked Examples

Complete, runnable programs, each exercising one layer of the library. All assume `import numpy as np` and a working install (Chapter 9).

11.1 A three-variable QUBO, end to end

Minimise $E(x) = x_0 + x_2 - 2x_0x_1 - 2x_1x_2$ over bits. Written as Eq. (2.2) with symmetric off-diagonals:

```
import numpy as np
from qanneal import solve

Q = np.array([
    [ 1.0, -1.0,  0.0],
    [-1.0,  0.0, -1.0],
    [ 0.0, -1.0,  1.0],
], dtype=float)      # off-diagonals halved: the double sum counts both orders

result = solve(Q, method="sqapt", reads=16, seed=0, return_bits=True)
print(result.best_sample, result.best_energy)  # [1 1 1], -2.0
```

With all three bits set, $E = 1 + 1 - 2 - 2 = -2$: the two rewards outweigh the two penalties. (Equivalently, keep the unhalved matrix and interpret it as an upper-triangular specification mirrored for symmetry — both conventions appear in the repository’s examples; what matters is the total pair strength.)

11.2 Number partitioning

The encoding derived in Section 2.4.1:

```
import numpy as np
from qanneal import DenseIsing, solve

nums = np.array([3, 5, 7, 11, 13], dtype=float)
n = len(nums)

J = np.zeros((n, n))
for i in range(n):
```

```

for j in range(i + 1, n):
    J[i, j] = J[j, i] = 2.0 * nums[i] * nums[j]

ising = DenseIsing(np.zeros(n), J, c=float(nums @ nums))
result = solve(ising, method="sqa", reads=20, seed=42)

spins = result.best_sample
diff = abs(float(nums @ spins))
print(f"groups: {spins}, |sum difference| = {diff}")
# 3+5+11 = 19 vs 7+13 = 20 -> diff 1.0 (best possible: total is odd)
print(f"energy = {result.best_energy}") # diff**2 exactly, thanks to c

```

11.3 MaxCut on a random graph

The encoding of Section 2.4.2, using SparseIsing and a networkx input for comparison:

```

import numpy as np, networkx as nx
from qanneal import SparseIsing, SparseEdge, solve

rng = np.random.default_rng(1)
G = nx.gnp_random_graph(60, 0.1, seed=1)
for (u, v) in G.edges:
    G.edges[u, v]["weight"] = float(rng.uniform(0.5, 2.0))

# Route 1: explicit sparse Ising ( $J_{ij} = +w_{ij}$ ; minimise  $\sum w s_i s_j$ )
edges = [SparseEdge(u, v, G.edges[u, v]["weight"]) for u, v in G.edges]
ising = SparseIsing(np.zeros(G.number_of_nodes()), edges,
                    G.number_of_nodes())
res = solve(ising, method="sqapt", replicas=8, reads=16, seed=2)

s = res.best_sample
cut = sum(G.edges[u, v]["weight"] for u, v in G.edges if s[u] != s[v])
print(f"cut weight = {cut:.3f}")

# Route 2: hand the graph straight to solve() (weights read from edges)
res2 = solve(G, method="sqapt", replicas=8, reads=16, seed=2)
print(res2.var_order[:5]) # node ordering used for res2.best_sample

```

11.4 Standard vs. optimal schedule

A dense spin glass, solved both ways at matched step budgets (cf. Chapter 8):

```

import numpy as np
from qanneal import solve, DenseIsing

rng = np.random.default_rng(0)
n = 50

```

```

J = np.triu(rng.standard_normal((n, n)), 1)
problem = DenseIsing(np.zeros(n), J + J.T)

r_std = solve(problem, method="sqapt", replicas=8, reads=10,
               progress=False, seed=5)

r_opt = solve(problem, method="sqapt", replicas=8, reads=10,
               progress=False, seed=5,
               schedule_type="optimal",
               optimal_eps_tilde=0.0,          # budget calibration
               optimal_num_steps=150,
               optimal_debug_csv="/tmp/opt_sched.csv")

print(f"standard: {r_std.best_energy:.4f}")
print(f"optimal : {r_opt.best_energy:.4f}")
# Inspect the realised trajectory:
# python scripts/plot_schedule_trajectory.py /tmp/opt_sched.csv

```

11.5 Inspecting the adaptive trajectory (low level)

```

import matplotlib.pyplot as plt
from qanneal import SQAAnnealer, SQASchedule, optimal_j_perp_params

beta, jp_start, jp_end = optimal_j_perp_params(problem, trotter_slices=32)

dummy = SQASchedule.from_vectors([1.0], [1.0])
ann = SQAAnnealer(problem, dummy, trotter_slices=32, replicas=4)
res = ann.run_optimal(beta=beta,
                     j_perp_start=jp_start, j_perp_end=jp_end,
                     eps_tilde=0.0, num_steps=200, sweeps_per_step=20,
                     worldline_sweeps=3)

fig, (ax1, ax2) = plt.subplots(2, 1, sharex=True, figsize=(7, 6))
ax1.plot(res.j_perp_trace); ax1.set_ylabel("J_perp")
ax2.plot(res.energy_trace); ax2.set_ylabel("energy")
ax2.set_xlabel("adaptive step")
ax1.set_title(f"calibrated eps_tilde = {res.calibrated_eps_tilde:.3g}")
plt.tight_layout(); plt.show()

```

Expect the fast–slow–fast shape of Fig. 8.1, with the steepest energy descent during the dwell.

11.6 Observers: watching an anneal converge

```
import matplotlib.pyplot as plt
from qanneal import Annealer, MetricsObserver, AnnealSchedule

schedule = AnnealSchedule.linear(0.1, 5.0, 60)
obs = MetricsObserver()
ann = Annealer(ising, schedule)
ann.set_seed(11)
res = ann.run(sweeps_per_beta=40, observer=obs)

fig, (ax1, ax2) = plt.subplots(2, 1, sharex=True)
ax1.plot(obs.energy_trace); ax1.set_ylabel("energy")
ax2.plot(obs.magnetization_trace); ax2.set_ylabel("magnetisation")
ax2.set_xlabel("temperature step")
plt.tight_layout(); plt.show()
```

For SQA, `SQAStateTraceObserver` additionally records per-sweep (β, Γ) , the sweep phase (slice / worldline / cluster), and full replica-slice state snapshots — everything needed to animate worldlines or to study slice decorrelation.

11.7 A large sparse instance

Ten thousand spins on a random 3-regular graph — comfortably in `SparseIsing` territory:

```
import numpy as np, networkx as nx
from qanneal import SparseIsing, SparseEdge, solve

n = 10_000
G = nx.random_regular_graph(3, n, seed=0)
rng = np.random.default_rng(0)
edges = [SparseEdge(u, v, float(rng.choice([-1.0, 1.0])))
          for u, v in G.edges]
ising = SparseIsing(np.zeros(n), edges, n)

result = solve(ising, method="sa", reads=4, seed=1) # SA first: baseline
print(result.best_energy)

result = solve(ising, method="sqa", trotter_slices=16, replicas=2,
               reads=2, seed=1) # then quantum
print(result.best_energy)
```

For instances of this size, start with `mode="fast"` tuned schedules and modest M ; scale up only what the energy traces show to be limiting.

11.8 dimod interoperability

```
import dimod
from qanneal import solve

bqm = dimod.BinaryQuadraticModel(
    {0: 1.0, 1: 3.0}, {(0, 1): -2.0}, vartype="BINARY")

result = solve(bqm, method="sqa", reads=8, return_bits=True, seed=0)
print(dict(zip(result.var_order, result.best_sample)))
```

11.9 Recommended workflow for a new problem

1. **Encode small, verify exactly.** Brute-force instances with $n \leq 20$ and check your encoding's optimum matches the known answer — encoding bugs dominate all other failure modes.
2. **Baseline with SA.** `method="sa", mode="fast"`: seconds of compute, and a reference energy.
3. **Escalate to SQA** (`mode="balanced"`) and compare; then **SQAPT** (`replicas=8` and `cluster_sweeps=1`) if reads disagree wildly — disagreement is the signature of a rugged landscape.
4. **Try the optimal schedule** when quality at fixed budget is the goal, after the pre-flight check of Section 8.6.
5. **Track variance.** Always look at `result.energies` across reads, not just the best — a method that finds the optimum in 15/16 reads beats one that finds a slightly better energy once.

Parallelism and HPC Deployment

qanneal parallelises at three nested levels: OpenMP threads inside one solver call, processes across reads on one node, and MPI/SLURM across nodes. They compose — but must be composed deliberately to avoid oversubscription.

12.1 OpenMP: inside a run

Built in by default (QANNEAL_ENABLE_OPENMP=ON). Parallel regions:

Class	Parallelised over
ReplicaAnnealer	replicas
SQAAnealer	replicas (each owning its slices), when no sweep-level observer is attached
SQAParallelTemperingAnnealer	ladder replicas

Thread count is controlled at runtime:

```
export OMP_NUM_THREADS=8
```

Each replica owns a private RNG stream, local-field cache and cluster buffers, so results are independent of the thread count; runs are reproducible for a fixed seed and fixed replica count.

Oversubscription

Threads only help while $OMP_NUM_THREADS \leq$ the number of parallel units (replicas). And when you add process-level parallelism (below), *multiply* the two: 8 worker processes $\times$ 8 OpenMP threads = 64 threads on what may be a 16-core node — a large slowdown, not a speedup. Standard practice on shared nodes: set $OMP_NUM_THREADS=1$ for many-process runs, or partition cores explicitly (e.g. 4 workers $\times$ 4 threads on 16 cores).

12.2 Process-level: the HPC launcher

examples/python/hpc_sqa_launcher.py distributes *reads* (independent restarts) over workers — the ideal parallelisation axis, since reads never communicate:

```
# Single node, multiprocessing (no extra dependencies)
python examples/python/hpc_sqa_launcher.py \
  --n 5000 --method sqapt --reads 64 --workers 8

# Multi-node MPI (requires mpi4py)
mpirun -n 32 python examples/python/hpc_sqa_launcher.py \
  --n 50000 --method sqa --reads 8 --mpi

# Built-in scaling study
python examples/python/hpc_sqa_launcher.py --scaling --method sqapt --mode fast
```

Each worker runs a full solver call with a distinct seed and writes a JSON result; the launcher merges them and reports the global best.

12.3 SLURM job arrays

The most portable route on clusters — no mpi4py, no long-running master, natural fault tolerance (a failed task costs one read batch):

```
sbatch scripts/slurm/run_sqa_array.sh
# afterwards, merge all task outputs:
python examples/python/merge_array_results.py results/run/*.json
```

Adapt the array script's template variables (problem size, method, reads per task, walltime) to your site; give each array task a distinct `--seed` derived from `$SLURM_ARRAY_TASK_ID` so reads are independent.

12.4 C++ MPI

For fully native distributed annealing:

```
cmake -S . -B build -DQANNEAL_ENABLE_MPI=ON
cmake --build build -j
mpirun -n 8 build/qanneal_mpi_example
```

The MPI layer (`MPIContext`, `run_replica_anneal()` in `include/qanneal/mpi/`) runs one replica group per rank and reduces the best result to rank 0. See `examples/mpi/sa_mpi.cpp` for the pattern.

12.5 Choosing the axis of parallelism

Level	Use when	Cost model
OpenMP (replicas)	one hard instance, want a better χ_B estimate or PT ladder	shared memory; near-linear up to replica count
Processes (reads)	success probability per read < 1 ; want $1 - (1 - p)^R$ aggregation	embarrassingly parallel; linear
Nodes (MPI/SLURM)	instance sweeps, many problems $\times$ many reads	linear; prefer job arrays for robustness

A practical default for one hard instance on one node: replicas=8 with OMP_NUM_THREADS=8, and reads spread over remaining cores via the launcher.

Tuning Guide and Troubleshooting

13.1 Which method, which knobs — a decision guide

Situation	Method	First knobs
Baseline / smooth landscape / very large sparse n	sa	tuned mode, reads
Tall narrow barriers, moderate n	sqa	trotter_slices, worldline_sweeps
Rugged, multi-modal, hard instances	sqapt	replicas, cluster_sweeps=1
Best quality at fixed budget, favourable regime ($J_{\text{rms}} \gtrsim 3$)	sqa/sqapt, schedule_type="optimal"	optimal_num_steps, replicas $\geq$ 4
Discretisation-free QA statistics	ctpimc	sweeps_per_beta, Γ_{end} small but > 0

13.2 Systematic tuning, in order

1. **Reads before sweeps.** Doubling reads is embarrassingly parallel and attacks run-to-run variance directly; double sweep counts only when single-read traces show the anneal is *unconverged* (energy still falling at the last step).
2. **Schedule mode.** fast $\rightarrow$ balanced $\rightarrow$ accurate: each step roughly doubles schedule length and tightens the final acceptance target.
3. **SQA moves.** Keep the ratio of sweeps_per_beta to worldline_sweeps around 10:1; add cluster_sweeps=1 when late-anneal traces plateau above the best known energy (a stiffness symptom, Section 5.3.3).
4. **Trotter slices.** Raise M ($32 \rightarrow 64$) only if the spread of per-slice energies remains large at the end of the run — that spread is the observable Trotter-error signature.
5. **Ladder size (sqapt).** Grow replicas until mean swap acceptance sits in 0.2–0.5.

13.3 Symptom tables

13.3.1 General

Symptom	Likely cause	Fix
Energies vary wildly across reads	rugged landscape; schedule too fast	more reads; sqapt; mode="accurate"
Energy trace flat from step one	schedule starts too cold	tuned schedules; lower beta_start
Energy still falling at last step	schedule too short	more steps / sweeps_per_beta
Solutions violate constraints	penalty λ too small	raise λ just above the max objective gain (Section 2.4.3)
Great spins, wrong reported energy	offset c omitted after manual conversion	carry c (Eq. (2.6)), or let qanneal convert

13.3.2 SQAPT

Symptom	Likely cause	Fix
Swap acceptance ≈ 0	rungs too far apart	more replicas; narrower ladder
Swap acceptance ≈ 1	rungs redundant	fewer replicas or wider ladder
Cold rung stuck above best energy	too few epochs; mid-ladder bottleneck	raise pt_steps; inspect final_energies profile

13.3.3 Optimal schedule

(Details and trace plots in Section 8.6.)

Symptom	Likely cause	Fix
$J_{\perp}(t)$ linear, no dwell	degenerate problem class ($J_{\text{rms}} \ll J_0$) or bad endpoints	run sanity_schedule.py; pass explicit j_perp_end
$\Delta J_{\perp}$ constant	noisy χ_B	replicas ≥ 4 , sweeps_per_step ≥ 10
final_j_perp $\ll$ resolved_j_perp_end	calibration failure	inspect debug CSV; more calib_probes/calib_sweeps
Endpoint reached in a handful of steps (manual $\tilde{\epsilon}$)	$\tilde{\epsilon}$ too large	use optimal_eps_tilde=0.0 (calibration)
$J_{\perp}$ barely moves (manual $\tilde{\epsilon}$)	$\tilde{\epsilon}$ too small	same fix — prefer calibration

13.4 Environment issues

ModuleNotFoundError: qanneal._qanneal Stale or mismatched build — reinstall from the repo root (python -m pip install . --no-build-isolation); don't mix an installed copy with a source-tree PYTHONPATH.

CTPIMCAnnealer is not available The installed build predates the CT-PIMC bindings; reinstall from current sources.

macOS: single-threaded despite OpenMP flag Apple Clang lacks OpenMP; brew install libomp and rebuild.

Cluster slowdowns with many workers Thread oversubscription — see the warning in Chapter 12; set OMP_NUM_THREADS=1 per worker or partition cores.

13.5 Reproducibility checklist

- Pass an explicit seed to solve() (or set_seed() on annealers); with fixed seed, replica count and thread availability, runs are bitwise reproducible.
- Record qanneal.__version__, the schedule (or tuned-mode name + probe seed), and all sweep counts alongside results.
- For adaptive runs, archive the debug CSV — it is the complete audit trail of the schedule actually executed.

The C++ API

The entire engine layer is usable directly from C++17 — the Python API is a thin binding over it, so every concept of Chapters 4–8 maps one-to-one.

14.1 Umbrella include and headers

```
#include "qanneal/core.hpp"
```

Class / function	Header
DenseIsing	dense_ising.hpp
SparseIsing, SparseEdge	sparse_ising.hpp
QUBO	qubo.hpp
AnnealSchedule	schedule.hpp
SQASchedule	sqa_schedule.hpp
Annealer	annealer.hpp
ReplicaAnnealer	replica_annealer.hpp
ParallelTemperingAnnealer	parallel_tempering.hpp
SQAAnnealer	sqa_annealer.hpp
SQAParallelTemperingAnnealer	sqa_parallel_tempering.hpp
CTPIMCAnnealer	ctpimc_annealer.hpp
State, SQASState	state.hpp, sqa_state.hpp
Observers (all)	metrics_observer.hpp
MPIContext, run_replica_anneal()	mpi/mpi_context.hpp, mpi/mpi_replica.hpp

14.2 A complete C++ example

Dense spin glass, simulated annealing:

```
#include "qanneal/core.hpp"
#include <iostream>
#include <random>
#include <vector>
```

```

int main() {
    const std::size_t n = 64;
    std::mt19937_64 rng(7);
    std::normal_distribution<double> g(0.0, 1.0);

    std::vector<double> h(n, 0.0);
    std::vector<std::vector<double>> J(n, std::vector<double>(n, 0.0));
    for (std::size_t i = 0; i < n; ++i)
        for (std::size_t j = i + 1; j < n; ++j)
            J[i][j] = J[j][i] = g(rng);

    qanneal::DenseIsing ising(h, J, /*c=*/0.0);
    auto schedule = qanneal::AnnealSchedule::linear(0.1, 5.0, 80);

    qanneal::Annealer annealer(ising, schedule);
    annealer.set_seed(42);
    auto result = annealer.run(/*sweeps_per_beta=*/60);

    std::cout << "best energy: " << result.best_energy << "\n";
    for (auto s : result.best_state.spins) std::cout << int(s) << ' ';
    std::cout << std::endl;
    return 0;
}

```

And the quantum analogue in a few lines:

```

auto sqa_sched = qanneal::SQASchedule::from_vectors(betas, gammas);
qanneal::SQAAnnealer sqa(ising, sqa_sched,
    /*trotter_slices=*/32, /*replicas=*/4);
auto res = sqa.run(/*sweeps_per_beta=*/40,
    /*worldline_sweeps=*/4,
    /*cluster_sweeps=*/1);
// adaptive schedule:
auto opt = sqa.run_optimal(/*beta=*/1.0,
    /*j_perp_start=*/0.1, /*j_perp_end=*/25.0,
    /*eps_tilde=*/0.0,    // budget calibration
    /*alpha=*/15.0/14.0,
    /*num_steps=*/150, /*sweeps_per_step=*/20,
    /*worldline_sweeps=*/3);

```

Reference build targets and CMake flags are in Chapter 9; examples/dense_ising_sa.cpp, examples/sqa_quantum_parallel_tempering.cpp and examples/mpi/sa_mpi.cpp in the repository are compiled by the default build and serve as living documentation.

14.3 Implementation notes for contributors

- **Local-field caches.** Every engine keeps $f_i = h_i + \sum_j J_{ij}s_j$ per (replica, slice); proposals are $O(1)$ (Eq. (2.10)), accepted flips update the cache along the flipped spin's neighbourhood.

Backends implement `energy()`, `delta_energy()` and `compute_local_fields()` over `raw int8_t*` slices.

- **Threading model.** OpenMP parallelises over replicas; each replica owns disjoint regions of the field cache, its own `std::mt19937_64` stream seeded from the master RNG, its own shuffled visit order and cluster buffers. No locks are needed in the hot path.
- **Bond-sum tracking.** The adaptive schedule's B (Eq. (8.4)) is updated incrementally by every move (Section 8.3) — keep this invariant when adding new move types: a move must report its exact ΔB .
- **Observers.** A sweep-level observer forces the sequential SQA path (deterministic interleaving for reproducible traces); metrics-only observers keep the parallel path.
- **Errors.** Parameter validation throws `std::invalid_argument` (surfaced as `ValueError` in Python); sentinel resolutions that alter semantics print a `cerr` warning rather than failing silently.

Part V

Appendices

Symbols, Conventions, and Reference Tables

A.1 Symbol glossary

Symbol	API name	Meaning
$s_i \in \{-1, +1\}$	spins	Ising variables; qanneal's native representation.
$x_i \in \{0, 1\}$	bits	QUBO variables; $x = (1 + s)/2$.
h_i	h	local (longitudinal) field on spin i .
J_{ij}	J	pair coupling; ferro < 0 , antiferro > 0 .
c	c	constant energy offset.
Q_{ij}	Q	QUBO matrix; diagonal = linear terms.
β	beta	inverse temperature $1/(k_B T)$, dimensionless.
Γ	gamma	transverse-field strength (quantum fluctuations).
M	trotter_slices	number of imaginary-time slices.
$J_\perp$	j_perp	Trotter coupling, $\frac{1}{2} \ln \coth(\beta \Gamma / M)$ (Eq. (5.7)).
$S[s]$	—	effective classical action (Eq. (5.8)).
B	—	imaginary-time bond sum $\sum_{t,i} s_i^t s_i^{t+1}$ (Eq. (8.4)).
χ_B	chi_B	bond susceptibility $\text{Var}(B)$.
$\tilde{\epsilon}$	eps_tilde	adiabaticity scale of the adaptive update (Eq. (8.5)).
α	alpha	universality exponent $z/(2 - \eta) + 1/2$.
J_{rms}	j_rms	RMS of non-zero couplings; sets the critical scale $J_\perp^c$.
R	replicas	Trotter systems (SQA) / ladder rungs (SQAPT).
T	num_steps	adaptive-schedule step budget.
Δ	—	probed 75th-percentile $ \Delta E $ (tuned schedules).

A.2 Universality exponents for α

Universality class	System	$\alpha = z/(2 - \eta) + 1/2$
1-D quantum Ising	transverse-field Ising chain ($z = 1, \eta = 1/4$)	$15/14 \approx 1.071$ (default)
2-D quantum Ising	2-D TFIM	$7/4 = 1.75$
Mean-field	Sherrington–Kirkpatrick-like	≈ 1.0 (tentative)

For combinatorial problems of unknown class, keep the default: budget calibration (Section 8.2) makes the schedule's overall pace insensitive to moderate errors in α .

A.3 Key formulas on one page

Ising:	$E(s) = \sum_i h_i s_i + \sum_{i < j} J_{ij} s_i s_j + c$
QUBO:	$E(x) = \sum_{ij} Q_{ij} x_i x_j, \quad x = (1 + s)/2$
Local field / flip:	$f_i = h_i + \sum_j J_{ij} s_j, \quad \Delta E_i = -2s_i f_i$
Metropolis:	$A = \min(1, e^{-\beta \Delta E})$
Trotter coupling:	$J_{\perp} = \frac{1}{2} \ln \coth(\beta \Gamma / M)$
SQA action:	$S = \frac{\beta}{M} \sum_t E_{\text{cl}}(s^t) - J_{\perp} \sum_{t,i} s_i^t s_i^{t+1}$
SQA flip:	$\Delta S = \frac{\beta}{M} \Delta E_{\text{cl}} + 2J_{\perp} s_i^t (s_i^{t-1} + s_i^{t+1})$
SW join probability:	$p_{\text{join}} = 1 - e^{-2J_{\perp}}$
PT swap (classical):	$A = \min(1, e^{(\beta_{r+1} - \beta_r)(E_{r+1} - E_r)})$
PT swap (SQA):	$A = \min(1, e^{-(S_r(x_{r+1}) + S_{r+1}(x_r) - S_r(x_r) - S_{r+1}(x_{r+1}))})$
Adaptive update:	$\Delta J_{\perp} = \tilde{\epsilon} \chi_B^{-\alpha}, \quad \alpha = \frac{z}{2-\eta} + \frac{1}{2}$
Calibration:	$\tilde{\epsilon} = \frac{1}{T} \int_{J_0}^{J_{\text{end}}} \chi_B(J)^{\alpha} dJ$
Trajectory:	$J^{(t)} = G^{-1}\left(\frac{t}{T-1} G(J_{\text{end}})\right), \quad G(J) = \int_{J_0}^J \chi_B^{\alpha} dJ'$

A.4 Repository map

Path	Contents
include/qanneal/	C++ public headers (Chapter 14).
src/	C++ engine implementations; src/third_party/ vendored localPIMC.
python/qanneal/	Python package: solver.py (dispatch, tuned schedules, helpers), bindings import.
examples/python/	Runnable demos and launchers.
examples/, examples/mpi/	C++ examples.
notebooks/	Four tutorial Jupyter notebooks (quickstart, SQA physics, SQAPT/CT-PIMC, large problems).
scripts/	sanity_schedule.py, plot_schedule_trajectory.py, SLURM templates.
docs/	Markdown docs; this manual in docs/manual/.
tests/	Python test suite; C++ tests under ctest.

A.5 License

qanneal is released under the Apache License 2.0 (see `LICENSE`). The CT-PIMC engine incorporates D-Wave's `localPIMC` (Apache-2.0), credited in `NOTICE`.

Bibliography

- [1] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, E. Teller, *Equation of State Calculations by Fast Computing Machines*, J. Chem. Phys. **21**, 1087 (1953).
- [2] S. Kirkpatrick, C. D. Gelatt, M. P. Vecchi, *Optimization by Simulated Annealing*, Science **220**, 671 (1983).
- [3] M. Suzuki, *Relationship between d -Dimensional Quantal Spin Systems and $(d+1)$ -Dimensional Ising Systems*, Prog. Theor. Phys. **56**, 1454 (1976).
- [4] T. Kadowaki, H. Nishimori, *Quantum annealing in the transverse Ising model*, Phys. Rev. E **58**, 5355 (1998).
- [5] E. Farhi, J. Goldstone, S. Gutmann, M. Sipser, *Quantum Computation by Adiabatic Evolution*, arXiv:quant-ph/0001106 (2000).
- [6] G. E. Santoro, R. Martoňák, E. Tosatti, R. Car, *Theory of Quantum Annealing of an Ising Spin Glass*, Science **295**, 2427 (2002).
- [7] R. Martoňák, G. E. Santoro, E. Tosatti, *Quantum annealing by the path-integral Monte Carlo method: The two-dimensional random Ising model*, Phys. Rev. B **66**, 094203 (2002).
- [8] J. Roland, N. J. Cerf, *Quantum search by local adiabatic evolution*, Phys. Rev. A **65**, 042308 (2002).
- [9] R. H. Swendsen, J.-S. Wang, *Nonuniversal critical dynamics in Monte Carlo simulations*, Phys. Rev. Lett. **58**, 86 (1987).