qdjj_solver: User's Guide and Reference Manual

Version 0.3.0

The qdjj_solver developers

Development snapshot d0fa801df2b8

CONTENTS

1 Installation and a first checked calculation 2

2 Physical models and conventions 5

3 Using the solver and saving results 13

4 Static states, observables and transitions 21

5 Choosing and refining superconducting reservoirs 25

6 QP calculations and observable-specific convergence 28

7 DMRG states and bond-dimension refinement 32

8 Parameter scans, saved results and reproducible analysis 36

9 General impurities and custom operators 39

10 Floquet calculations 41

11 Probe matrix elements and spectra 48

12 Troubleshooting 50

A Option reference 53

B Model and drive construction 61

C Result reference 74

D Observables and probes 80

E Command-line reference 82

F Conventions 85

G From the equations to the code 87

H Calculation input format 91

I Saved records and artifacts 96

J API reference 104

K Citation and method references 136

Index 139

Software version 0.3.0. Development snapshot d0fa801df2b8.

This handbook explains how to set up calculations, solve models, assess numerical accuracy and save results with `qdjj_solver`. It assumes basic Python and condensed-matter physics. The companion physics manual develops the Hamiltonians, subgap-state physics and driven systems.

Read the user guide in sequence for a first calculation, then use the reference chapters to look up parameters, return values and input formats. To obtain continuum results at the desired accuracy, check convergence of the eigensolver, the many-body representation and the bath discretization separately.

The HTML edition and this PDF use the same canonical chapters. Source-file links identify the corresponding software revision; the matching source distribution contains the examples and public inputs needed to run them offline.

CHAPTER

ONE

INSTALLATION AND A FIRST CHECKED CALCULATION

This chapter explains how to install the package from source and calculate a quantum-impurity state. It assumes basic Python, including imports, function calls and dictionaries. The numerical methods are introduced as they are used. The physics manual explains the Hamiltonian and subgap states; this handbook explains how to set up and reproduce the corresponding numerical calculation.

1.1 Install the package

Use Python 3.11 or later. Building from source needs a C++17 compiler and the development headers for that Python interpreter. On Debian/Ubuntu these normally come from `build-essential` and the matching `python3-dev`; macOS uses the Xcode Command Line Tools; Windows uses Visual Studio C++ Build Tools and the Windows SDK. Install the compiler and headers first; pip supplies CMake and pybind11.

```
git clone https://github.com/rokzitko/qdjj_solver.git
cd qdjj_solver
python -m venv .venv
```

Activate the environment with `source .venv/bin/activate` on macOS/Linux or `.venv\Scripts\Activate.ps1` in Windows PowerShell. Choose one installation:

```
python -m pip install .
# Or include the optional DMRG backend and plotting:
python -m pip install '.[dmrg,plots]'
```

The base installation supports the model-building API and QP solver. The `dmrg` extra adds TeNPy and support for saving MPS states in HDF5 files; `plots` adds Matplotlib. Development and documentation dependencies are only needed for their corresponding tasks. Run the following command in that same environment to check which package is used:

```
python -c "import qdjj_solver; print(qdjj_solver.__version__,␣
↪qdjj_solver.__file__)"
```

Commands below that begin with `examples/` are run from the source root. The example scripts resolve their supplied inputs relative to their own locations, so an absolute script path also works from a separate working directory.

1.2 Calculate one state

The following code is generated from the complete, independently runnable `examples/quickstart.py`. Copy it into a Python session, or run `python -B examples/quickstart.py`.

```
from qdjj_solver import Sector, cosh_grid, reference_model, solve


bath = cosh_grid(pairs=1, delta=1.0, bandwidth=5.0)
h = reference_model(bath, u=1.6, gamma=0.3, phi=0.8, compress=False)
sector = Sector(parity=1, twice_sz=1)
qp = solve(h, cutoff=2, sector=sector, backend="qp")
print(qp.energies, qp.qp_weights)

assert abs(qp.energies[0] - (-0.2950595967187083)) < 1e-10
assert max(qp.residuals) < 1e-9
assert abs(qp.qp_weights[0].sum() - 1) < 1e-12
```

The output contains one energy followed by the probabilities of zero, one and two bath quasiparticles. They are approximately 89.27%, 10.10% and 0.63% here. The script checks the energy against its finite-model reference, the residual of the eigenvalue equation, and normalization of those probabilities.

An **object** stores related information: `bath` holds a discretized reservoir, `h` holds the finite Hamiltonian, `sector` identifies allowed quantum numbers, and `qp` holds the numerical result. A **backend** is the numerical method used to solve that same Hamiltonian. The top-level `solve` selects one with `backend`.

Input	Meaning in this calculation
`pairs=1`	Two signed normal levels per lead, each with two spin modes
`delta=1`	Energy unit: the superconducting gap
`bandwidth=5`	Normal-state half-bandwidth
`gamma=0.3`	Total hybridization, shared equally by the two leads
`compress=False`	Retain the full declared finite bath, including dark modes
`Sector(1, 1)`	Odd total parity and total spin projection +1/2
`cutoff=2`	At most two bath QPs; all impurity configurations are retained

The reported energy is the many-body energy of a state in the selected sector. Excitation energies are differences between state energies, and identifying the global ground state requires comparison with the other relevant sectors. The projected residual measures eigensolver accuracy within the chosen space; cutoff and bath refinement assess the accuracy of the approximations defining that space.

1.3 Run the same case through an input file

Tutorial 00 supplies complete inputs for QP, unrestricted same-bath ED and DMRG:

```
qdjj-solver solve examples/00_first_calculation/input/qp.json --output␣
↪results/first.json --save-states
```

This creates a JSON record and a companion NPZ file containing the wavefunctions. Inspect the JSON filename printed by the command and the saved diagnostics. With DMRG, the convergence flags identify whether the returned states satisfy the finite-problem checks; its default also permits diagnostic output from unconverged runs.

Python and CLI defaults differ. In particular, Python `solve` defaults to `cutoff=None`, while the QP CLI defaults to two. Always set consequential numerical and representation choices explicitly in a scientific calculation. The full *defaults table* (page 14) lists these differences.

1.4 Compare methods and keep the result

With `.[dmrg]` installed, run the full first-calculation tutorial:

```
python -B examples/00_first_calculation/run.py --profile full --output␣
↪runs/first-comparison
```

This compares the same finite Hamiltonian using QP truncation, DMRG and full finite-space ED. Agreement with ED validates the solution of that finite problem. Assess its accuracy as a continuum approximation through separate bath refinement.

Use the *saved-result workflow* (page 36) to distinguish scalar-only analysis from reloading wavefunctions. The latter requires keeping the JSON and its named companion file together. A **checkpoint** here is a completed saved MPS state used as the initial wavefunction for a new optimization.

1.5 Choose your next route

- To calculate subgap states and current, read *model construction* (page 5), *running the solver* (page 13), then *static calculations* (page 21).
- To establish accuracy, read *reservoir refinement* (page 25), *QP convergence* (page 28) or *DMRG refinement* (page 32).
- To calculate driven spectra, first obtain and interpret the static states, then follow *Floquet calculations* (page 41) and *spectroscopy* (page 48).
- To look up a parameter, use the *constructor reference* (page 61), *solver options* (page 53) and *input-format reference* (page 91).

The small installation test above and the reference junction used in the next workflow are deliberately different cases. Compare expected energies using matched Hamiltonians and numerical settings.

CHAPTER

TWO

PHYSICAL MODELS AND CONVENTIONS

2.1 From objects to a calculation

The usual sequence is `DiscreteBath` → `Hamiltonian` → `solve` → `Eigenstates`. `reference_model` constructs a symmetric dot junction; `make_model` combines an explicit `Impurity`, reservoir list and tunnelling matrices. `Sector` selects conserved total quantum numbers. Solver options control the numerical method applied to that fixed Hamiltonian.

The *static workflow* (page 21) maps a complete physical example onto these objects. The *constructor reference* (page 61) lists parameter names and defaults extracted from the implementation; this chapter explains their physical conventions. The *custom-model chapter* (page 39) develops the general construction with matched Python and JSON examples.

2.2 Units and energy reference

Energies, gaps, hybridizations, and energy/residual tolerances use the units of the supplied Hamiltonian. Both numerical methods use these units directly. `Impurity.anderson(u, detuning, field)` constructs

$$H_d = \frac{U}{2}(n_d - 1)^2 + \epsilon(n_d - 1) + BS_d^z, \qquad S_d^z = (n_\uparrow - n_\downarrow)/2.$$

Here $n_d = n_\uparrow + n_\downarrow$. The empty, spin-up, spin-down, and doubly occupied states have energies $U/2 - \epsilon$, $B/2$, $-B/2$, and $U/2 + \epsilon$, respectively, with `detuning` $= \epsilon$ and `field` $= B$. B is a Zeeman **energy**, supplied as `field`. Convert a magnetic field in tesla to this energy using the relevant g-factor. Positive `field` raises the spin-up level. For the conventional Anderson impurity $H_A = \epsilon_d n + U n_\uparrow n_\downarrow + BS_z$, the conversion is

$$\epsilon = \epsilon_d + \frac{U}{2}, \qquad H_d = H_A - \epsilon_d.$$

Thus set `detuning=epsilon_d+u/2`. Add ϵ_d to the reported impurity energy to recover the conventional additive reference; excitation differences are unaffected. For a complete reservoir model, also account for the bath reference below. Spin observables use units of $\hbar$, and `field` is the energy multiplying that dimensionless spin.

The functions `make_model` and `reference_model` subtract the disconnected BCS vacuum energy

$$E_{\mathrm{BCS}}^{(0)} = \sum_l \sum_i \left(\xi_{li} - \sqrt{\xi_{li}^2 + \Delta_l^2} \right).$$

For one Anderson impurity, the full finite-system energy in the conventional electron Hamiltonian is therefore

$$E_{\mathrm{conventional}} = E_{\mathrm{reported}} + \epsilon_d + E_{\mathrm{BCS}}^{(0)}.$$

A coupled-reservoir QP vacuum retains its exact energy difference from that same reference. Additive constants are retained in the operator expression and saved Hamiltonian. A user-built `Hamiltonian` uses exactly its specified additive constant.

2.3 Reservoir measure

The bath representation guide gives complete parameter tables, switching examples, convergence properties, and measured accuracy/cost comparisons for the quadratures and surrogate fits summarized here.

`DiscreteBath(xi, weights, delta, bandwidth)` describes spin-degenerate signed normal levels. For a flat band of half-width D, the normal-state density of states **per spin** is $\rho = 1/(2D)$, and

$$C_{l\sigma} = \sum_i \sqrt{w_{li}} c_{li\sigma}, \qquad w_{li} \simeq \rho_l \, d\xi.$$

Weights represent the normal-state measure. The contact operator has normalization $\{C_{l\sigma}, C_{l\sigma}^\dagger\} = \sum_i w_{li}$, which may differ from one. The constructor preserves the supplied weights, since rescaling them to unit sum would change the physical tunnelling amplitudes. Each signed level contributes two positive-energy Bogoliubov modes. Weights must be positive. The bath keeps a fixed copy of the supplied energies and weights; exact symmetry reductions require exactly mirrored grids.

`cosh_grid(pairs, delta, bandwidth)` uses Gauss–Legendre quadrature in

$$\xi = \pm\Delta \sinh t, \quad E = \Delta \cosh t, \quad 0 \le t \le \operatorname{asinh}(D/\Delta).$$

It yields `2*pairs` signed levels **per reservoir** and includes the $\Delta \cosh t$ Jacobian. The weights retain their quadrature values; the zeroth-moment error is `bath.metadata['measure_error']`.

`fit_surrogate(levels, ...)` fits positive residues to the finite-band function

$$g(i\omega) = \frac{2}{\pi} \frac{\arctan[D/\sqrt{\omega^2 + \Delta^2}]}{\sqrt{\omega^2 + \Delta^2}}.$$

Its default logarithmic mesh and absolute least-squares criterion follow the published construction in *references* (page 136). Settings and fit errors are recorded. Check excitation energies themselves under bath refinement to assess their accuracy.

`chain_expansion(levels, ...)` constructs the Padé ChE bath of Bobok et al. (2025), with a finite-band target by default and analytic wide-band coefficients selected by `wide_band=True`. The returned bath has exact mirrored star coordinates and unnormalized positive contact weights. In wide-band mode, `bandwidth` only fixes the density normalization and cancels from physical hybridizations. See chain-expansion reservoirs for the construction, parameters and independent validation.

2.4 Contacts, phases, and spin mixing

The phase convention places the superconducting phases in the hopping terms. In this gauge, each reservoir has the real-gap Hamiltonian

$$H_l = \sum_{i,\sigma} \xi_{li} c_{li\sigma}^\dagger c_{li\sigma} - \Delta_l \sum_i \left(c_{li\uparrow}^\dagger c_{li\downarrow}^\dagger + c_{li\downarrow} c_{li\uparrow} \right), \qquad \Delta_l > 0.$$

This fixes the pairing sign and operator order used for anomalous expectation values. Its vacuum contribution is included in the subtraction described above. In the physical gauge, with real tunnelling amplitudes, the pairing term of lead l is instead $-\Delta_l \sum_i (e^{-i\phi_l} c_{li\uparrow}^\dagger c_{li\downarrow}^\dagger + \text{h.c.})$, so the pair amplitude $\langle c_{li\uparrow}^\dagger c_{li\downarrow}^\dagger \rangle$

is proportional to $e^{i\phi_l}$. The substitution $c_{li\sigma} \to e^{-i\phi_l/2}c_{li\sigma}$ makes the pair potential real and moves the phase into the tunnelling terms.

In `make_model`, `tunneling[l]` is a matrix with two reservoir-spin rows and `2*impurity.orbitals` impurity-spin-orbital columns. It enters as $e^{i\phi_l/2}C_l^\dagger T_l d + \text{h.c.}$ Here C_l and d are columns of reservoir-contact and impurity annihilation operators. `direct[(l,m)]` specifies W_{lm}, for $l < m$, and contributes $e^{i(\phi_l-\phi_m)/2}C_l^\dagger W_{lm} C_m + \text{h.c.}$ Both matrices may be complex and may mix spin projections. Use `twice_sz=None` if spin projection is not conserved.

`bath_reference='coupled'` first diagonalizes the quadratic reservoir Hamiltonian; that transformation currently requires spin-conserving direct hopping and an unpolarized quasiparticle vacuum with a nonzero excitation gap. The isolated-vacuum construction supports general spin-mixing contacts.

`reference_model` constructs a single dot with symmetric lead couplings. `gamma` is **total** hybridization: $\Gamma_l = \pi\rho\, t_l^2$ for the tunnelling amplitude t_l of lead l, $\Gamma = \sum_l \Gamma_l$. A symmetric junction uses $\Gamma/2$ per lead and phases $[-\phi/2, +\phi/2]$. Phases are in radians, with $\phi = \phi_R - \phi_L$. In the `(up, down)` spin order, the reference model's direct contact matrix is

$$W_{LR} = W_N I + iW_S\sigma_z = \begin{pmatrix} W_N + iW_S & 0 \\ 0 & W_N - iW_S \end{pmatrix}, \qquad W_N = \frac{\texttt{rho_wn}}{\rho}, \quad W_S = \frac{\texttt{rho_ws}}{\rho}.$$

Both `rho_ws` and `rho_wn` are dimensionless ρW inputs. The Hamiltonian term is $e^{-i\phi/2}C_L^\dagger W_{LR} C_R + \text{h.c.}$ This spin-diagonal W_S term conserves S_z; more general spin-mixing contacts can be supplied to `make_model`.

2.5 Model constructors

Import `reference_model` and `make_model` from `qdjj_solver`. Both return a `Hamiltonian`; calculate its eigenstates by choosing the backend and QP cutoff in `solve`. The name `reference_model` denotes the convenience constructor used for the reference junction in the authors' *ASQ variational manuscript* (page 137). The earlier single-reservoir and QD-junction variational papers are listed there as well.

2.5.1 `reference_model`

Parameter	Default	Meaning and constraints
`bath`	Required	One `DiscreteBath`, reused for both leads in a symmetric junction
`u`	`2.0`	Impurity interaction U, in energy units
`gamma`	`0.4`	Nonnegative **total** hybridization, in energy units; split equally between junction leads
`phi`	`3*pi/5`	Junction phase difference in radians; ignored for a single reservoir
`rho_ws`	`0.0`	Dimensionless spin-dependent direct contact ρW_S defined above
`rho_wn`	`0.0`	Dimensionless spin-independent direct contact ρW_N defined above

continues on next page

Table 1 – continued from previous page

Parameter	Default	Meaning and constraints
`geometry`	`"junction"`	`"junction"` for two leads or `"single"` for one; the latter requires both direct contacts to be zero
`detuning`	`0.0`	$\epsilon_d + U/2$, in energy units; zero is half filling
`field`	`0.0`	Local Zeeman energy; positive values raise spin-up
`symmetry`	`True`	Use eta coordinates when eligible; otherwise keep physical-impurity coordinates
`compress`	`True`	Freeze dark modes in their vacuum when eligible; use `False` for the full finite-bath spectrum
`bath_reference`	`"isolated"`	`"isolated"` BCS reservoirs or `"coupled"` quadratic environment
`coefficient_tolerance`	`1e-14`	Nonnegative Hamiltonian-coefficient cleanup tolerance, in energy units

All scalar physical parameters and the cleanup tolerance must be finite. Eta coordinates require a junction with an exactly mirrored bath, `detuning=0`, `rho_wn=0`, and `bath_reference="isolated"`. Compression additionally requires `rho_ws=0`. Inspect `h.metadata["paired_mode_compression"]` for the actual choice. These are Python defaults; the unified CLI instead defaults to `compress=false`.

2.5.2 `make_model`

This constructor supports multiple impurity orbitals and distinct reservoirs. Let `L` be the number of reservoirs and `M=impurity.orbitals`.

Parameter	Default	Meaning and constraints
`impurity`	Required	An `Impurity` with `M` positive-integer orbitals and a Hermitian fermion operator
`baths`	Required	Nonempty sequence of `L` `DiscreteBath` objects
`tunneling`	Required	`L` finite complex matrices of shape `(2, 2*M)`, in energy units; reservoir-spin rows, impurity-spin-orbital columns
`phases`	`None`	`L` finite phases in radians; `None` means all zero

continues on next page

Table 2 – continued from previous page

Parameter	Default	Meaning and constraints
`direct`	`None`	Mapping `(l, m)` to finite complex `(2, 2)` contact matrices in energy units, with `0 <= l < m < L`; `None` means no direct contacts
`phase_velocities`	`None`	`L` finite phase derivatives with respect to the chosen parameter; `None` means zero, so no `phase_derivative` observable is added
`eta_basis`	`False`	Request verified symmetric single-dot eta coordinates; incompatible models raise an error
`compress_pairs`	`False`	Freeze dark modes; requires eligible eta coordinates and zero direct hopping
`bath_reference`	`"isolated"`	`"isolated"` or `"coupled"`; coupled coordinates require spin-conserving direct hopping and a gapped, unpolarized reservoir vacuum
`coefficient_tolerance`	`1e-14`	Finite nonnegative Hamiltonian-coefficient cleanup tolerance, in energy units
`direct_derivatives`	`None`	Mapping of observable names to contact mappings in the same form as `direct`; choose names distinct from impurity observables and `phase_derivative`

Eta coordinates require two identical mirrored baths, a half-filled Anderson impurity, equal real spin-independent tunnel contacts, and only the pure W_S direct contact defined above, together with `bath_reference="isolated"`. See coordinates and compression for representation choices and *input files* (page 17) for the JSON equivalents.

2.6 General impurities and finite geometries

Impurity modes are `(orbital 0 up, down, orbital 1 up, down, ...)`; their Hilbert space is untruncated. `Impurity(orbitals, operator)` accepts arbitrary fermionic interactions, exchange, pair hopping, and local pairing. `Impurity.from_integrals(one_body, interaction=None, constant=0.0)` uses

$$H = C + \sum_{ab} h_{ab} d_a^\dagger d_b + \frac{1}{4} \sum_{abcd} U_{abcd} d_a^\dagger d_b^\dagger d_d d_c,$$

with $h_{ab} =$ `one_body`$[a, b]$, antisymmetrized two-electron integrals, and $C =$ `constant`. All indices label spin-orbitals. The factor $1/4$ accounts for antisymmetry in both index pairs.

For arbitrary chains, stars, or imported discretizations, use `Hamiltonian(operator, nimp, spins, eta_labels=None, observables={}, metadata={})`. `spins[i]` is a twice-spin mode label `+1` or `-1`; the first `nimp` modes are exempt from the QP cutoff. The operator expression specifies all hoppings and interactions and hence the physical geometry.

2.7 Fermionic algebra

Monomials use signed, one-based indices: `+(i+1)` creates, `-(i+1)` annihilates. Products have mathematical order, with the rightmost factor acting first. The stored normal-ordered form lists creation operators in ascending mode order and annihilation operators in descending order. Saved terms contain this operator sequence and the real and imaginary parts of its coefficient.

The Python functions `create(i)`, `annihilate(i)`, and `number(i)` instead take zero-based mode indices. For example, `number(0).to_records()` returns `[{"operators": [1, -1], "real": 1.0, "imag": 0.0}]`.

Operator multiplication and changes of basis apply anticommutation relations to the full operator **before** projection. For a bath-QP annihilation operator c and a projector P onto its vacuum, $Pcc^\dagger P = P$, whereas $(PcP)(Pc^\dagger P) = 0$. This matters for composite observables at cutoff boundaries.

`Hamiltonian` checks mode ranges, parity, Hermiticity, and requested symmetries. These symmetry checks also apply to small coefficients. If small coefficients are discarded when constructing a model, their summed magnitudes are recorded in `metadata["coefficient_cleanup_bound"]` as an upper bound on the operator-norm change. Conversion to a matrix product operator (MPO) preserves all remaining coefficients.

2.8 Sectors and canonical coordinates

Both solvers use `Sector(parity=1, twice_sz=1, eta=None, particle_number=None)`. Parity is zero for even, one for odd. `twice_sz=None` retains all spin projections. `eta` is the product of occupied modes' assigned eta labels and is available only in compatible representations. `particle_number` fixes total occupation of the **chosen canonical modes**, when conserved. In a Bogoliubov basis this is the occupation of the transformed modes; physical electron number is a distinct operator.

Model constructors save the electron-to-canonical transformation in `metadata['coordinates']`. `h.physical_operator(op)` transforms an expression in electron operators using that fixed map. Electron ordering is impurity modes, followed by reservoir, level, and spin indices. User-built models may specify the transformation explicitly. If none is specified, compatibility checks require an identical Hamiltonian before reusing states in another calculation.

Eta/paired-mode reduction is restricted to verified symmetric single-dot models. **`compress=True` fixes discarded dark modes in their vacuum.** The resulting spectrum is restricted to that subspace. Both QP and DMRG act on the modes retained by the constructor. For a full finite-bath spectrum, retain all modes with `compress=False`; the `qdjj-solver` terminal command uses this setting for both methods.

Transforming an arbitrary electron operator with `h.physical_operator` requires an uncompressed model: multiplying separately projected electron operators would miss vacuum contractions of the discarded modes. The supplied observables include these contractions. Saved model information identifies the restriction and the energies of discarded modes.

2.9 Built-in observables

`h.observables` maps names to operators in the model's canonical coordinates. Solvers report their expectation values as `result.observables[name][i]` for eigenstate `i`. `Impurity` supplies the default impurity observables; specify its `observables` dictionary to add operators or replace these definitions.

The names, definitions, availability and units are listed in the *observable reference* (page 80).

The `geometry="single"` reference model has the three impurity observables and no inter-lead/phase derivative. General models can supply additional impurity observables and named `direct_derivatives`. Their units follow the supplied operator coefficients. Built-in operators already include the transformations and contractions associated with the constructor's representation.

2.10 Local orbital density matrix

For one impurity orbital, use the ordered basis $|0\rangle, |\uparrow\rangle, |\downarrow\rangle, |2\rangle$, with $|2\rangle = d_\uparrow^\dagger d_\downarrow^\dagger |0\rangle$. In a state of fixed total fermion parity its local RDM is

$$\rho = \begin{pmatrix} P_0 & 0 & 0 & F^* \\ 0 & P_\uparrow & C & 0 \\ 0 & C^* & P_\downarrow & 0 \\ F & 0 & 0 & P_2 \end{pmatrix}, \qquad F = \langle d_\downarrow d_\uparrow \rangle, \qquad C = \langle d_\downarrow^\dagger d_\uparrow \rangle.$$

Here $P_1 = P_\uparrow + P_\downarrow$ and $\langle N \rangle = P_1 + 2P_2$. In particular, $\rho_{ab} = \langle |b\rangle\langle a| \rangle$: the matrix element $\rho_{0,2} = F^*$ is conjugate to the expectation of $|0\rangle\langle 2| = d_\downarrow d_\uparrow$.

For normalized local paired states,

$$|\pm\rangle = \frac{|0\rangle \pm |2\rangle}{\sqrt{2}}, \qquad P_\pm = \frac{P_0 + P_2}{2} \pm \mathrm{Re}\, F.$$

These are unconditional probabilities and sum to the local even weight $P_0 + P_2$, not necessarily one. Divide by that weight for probabilities conditioned on even local occupancy. For a variable relative phase,

$$P_+(\theta) = \frac{P_0 + P_2}{2} + \mathrm{Re}(e^{-i\theta} F), \qquad \max_\theta P_+(\theta) = \frac{P_0 + P_2}{2} + |F|.$$

The fixed $P_\pm$ refer to the declared electron gauge. The optimized probability is different if F is complex. Unit occupation alone does not distinguish a local spin from a coherent empty/double state or an incoherent mixture. `orbital_density_properties` reports all of these diagnostics and the purity, validating Hermiticity, trace, and positivity without renormalizing the matrix. *Usage examples* (page 15) cover eigenstates and degenerate mixtures.

2.11 Physical derivatives and transitions

To vary the lead phases along a parameter x, supply $\mathrm{d}\phi_l/\mathrm{d}x$ as `phase_velocities`. With all other physical parameters fixed, the resulting `phase_derivative` is

$$O_x = \sum_l \frac{\mathrm{d}\phi_l}{\mathrm{d}x} \frac{\partial H}{\partial \phi_l}.$$

This is the derivative along the specified phase variation, at fixed tunnelling amplitudes, gaps and other model parameters. `direct_derivatives[name][(l,m)]` specifies a physical contact observable even

when the corresponding Hamiltonian coupling is zero. Operators are constructed in the physical gauge and then transformed. Differentiating a parameter-dependent basis without connection terms would give a different operator. For phase bias, multiply $\partial H/\partial\phi$ by $2e/\hbar$ to obtain current.

For an exact eigenstate on a differentiable energy branch, the Hellmann–Feynman relation gives

$$I_i = \frac{2e}{\hbar}\left\langle \psi_i \left| \frac{\partial H}{\partial \phi} \right| \psi_i \right\rangle = \frac{2e}{\hbar}\frac{\partial E_i}{\partial \phi}.$$

At zero temperature, use the lowest branch for the equilibrium current. At a level crossing, take the appropriate one-sided branch derivatives. For approximate states, check agreement between the current expectation and the energy derivative as part of convergence. A parameter-dependent truncated variational space can contribute additional terms to the derivative of its approximate energy.

If the numerical Hamiltonian is expressed in an energy unit E_0 (`E0`), the current is `I = (2*e*E0/hbar) * result.observables["phase_derivative"][i]`, with $e > 0$. Convert `E0` to joules to obtain amperes. With gap `delta=1`, the numerical expectation is a current in units of $2e\Delta/\hbar$. With $\phi = \phi_R - \phi_L$, a positive I is an electric current from the left to the right lead; the *conventions* (page 85) also define the sign of the bias voltage.

Transition matrix elements require states and operators in the same canonical basis. DMRG also checks the MPS mode ordering, site grouping, and conserved quantum numbers, and includes the required Jordan–Wigner strings for operators of odd fermion parity. Independently calculated eigenvectors have arbitrary phases; compare squared matrix-element magnitudes (transition strengths), or complete degenerate subspaces, when comparing calculations.

CHAPTER

THREE

USING THE SOLVER AND SAVING RESULTS

This chapter uses the small installation-test Hamiltonian. Its Python examples form one session. Begin with the complete setup below; the larger reference junction is developed in *static calculations* (page 21).

```
from qdjj_solver import Sector, cosh_grid, reference_model, solve


bath = cosh_grid(pairs=1, delta=1.0, bandwidth=5.0)
h = reference_model(bath, u=1.6, gamma=0.3, phi=0.8, compress=False)
sector = Sector(parity=1, twice_sz=1)
qp = solve(h, cutoff=2, sector=sector, backend="qp")
print(qp.energies, qp.qp_weights)

assert abs(qp.energies[0] - (-0.2950595967187083)) < 1e-10
assert max(qp.residuals) < 1e-9
assert abs(qp.qp_weights[0].sum() - 1) < 1e-12
```

For the DMRG examples, install the `dmrg` extra and add:

```
mps = solve(h, sector=sector, backend="dmrg",
            options={"eigenpairs": 2, "chi_max": 32, "require_convergence":␣
↪True})
```

3.1 Calculations in Python

Import model constructors and the functions `solve`, `save_result`, and `load_result` from `qdjj_solver`. Their arguments are:

```
solve(hamiltonian, cutoff=None, sector=Sector(), options=None, initial=None, *,␣
↪backend="qp")
save_result(result, path, save_states=False)
load_result(path)
```

- `solve` returns the calculated `Eigenstates`. Supply the physical `Hamiltonian` as `hamiltonian` and the conserved quantum numbers as `sector`. Choose the method with `backend="qp"` or `"dmrg"`. `options=None` uses default numerical settings; a dictionary changes individual settings. A `SolverOptions` object must come from the chosen solver's module.
- `save_result` saves energies, observables, calculation settings, and diagnostics to a filename ending in `.json`. Set `save_states=True` to save the wavefunctions in a companion file. The function returns `None`.

- `load_result` reconstructs QP or DMRG `Eigenstates` from the JSON and its companion wavefunction file. It also reads the earlier QP result format.

The modules `qdjj_solver.qp_solver` and `qdjj_solver.dmrg_solver` also provide their own `solve` and `SolverOptions`. When calling either module's `solve` directly, use its `SolverOptions`; the top-level `qdjj_solver.solve` additionally accepts a dictionary. Starting states are described *below* (page 16).

3.1.1 Defaults

Setting	Python	`qdjj-solver` command
Numerical method	QP	QP
QP cutoff	`None`: all bath configurations allowed by the sector	`2`
Reference-model compression	`reference_model(..., compress=True)`	`false`
Sector	Odd parity, `twice_sz=1`, unrestricted eta/number	Same
Requested eigenstates	`1`	`1`

Compression applies only to eligible symmetric reference models. Use `compress=False` explicitly in Python when the full finite-bath spectrum is intended. DMRG always requires `cutoff=None`, and its default `require_convergence=False` returns diagnostics even for coarse, unconverged calculations. Set `require_convergence=True` for calculations that must meet all convergence checks for the finite Hamiltonian. QP results must always pass the projected-residual and orthonormality checks.

3.1.2 QP solver options

Use `qdjj_solver.qp_solver.SolverOptions` or a dictionary of its fields. All fields, defaults and units are listed in the *option reference* (page 53).

Requesting nearly the whole spectrum can force dense diagonalization even with an iterative method.

3.1.3 DMRG solver options

Use `qdjj_solver.dmrg_solver.SolverOptions`; this requires installation with `[dmrg]`. All fields, defaults and units are listed in the *option reference* (page 54).

Require `min_sweeps <= max_sweeps` and, with the mixer enabled, `mixer_sweeps < min_sweeps`. `require_convergence=True` requires residual evaluation. Disabling residuals leaves `finite_problem_converged=False`. Residual evaluation has its own temporary memory cost, described in numerics.

The local Lanczos threshold `lanczos_probability_tolerance` (TeNPy's `P_tol`, default `1e-14`) is independent of `energy_tolerance` and of the residual tolerance `residual_tolerance`. Check the full-state residual explicitly to assess the combined accuracy. The solver passes the local threshold to TeNPy for each calculation, preserving TeNPy's defaults. Saved results record the value used in `metadata["options"]`.

3.1.4 Results and transitions

The fields of both result types are listed in the *result reference* (page 74): `energies`, `residuals`, `observables`, `hamiltonian`, `metadata` and `timings` for both methods; `vectors`, `basis` and `qp_weights` for QP; `states` and `prepared` for DMRG. State indices are zero-based.

Named observable definitions and units are in the *observable reference* (page 80). For QP, all named observables are measured by default; the backend's `solve` and `QPWorkspace.solve` accept `observables` to select a subset. DMRG's `observables` option can also select a subset. Residuals measure how well the state satisfies the eigenvalue equation for the finite Hamiltonian; QP residuals refer to the chosen projected space.

The snippets below continue this chapter's `h`, `sector`, `qp`, and (with the optional DMRG installation) `mps` calculation. **Functions on DMRG results** return overlaps or transition matrices with bra eigenstates as rows and ket eigenstates as columns:

```
gram = mps.overlaps()
matrix = mps.matrix_elements(h.observables["impurity_spin_z"])
```

`bra_result.matrix_elements(operator, ket_result)[i, j]` evaluates $\langle \text{bra}_i | \text{operator} | \text{ket}_j \rangle$. DMRG transition results require identical canonical coordinates, mode ordering, grouping, and types of conserved quantum numbers. Their values may differ. For example, both results must use parity-only tensors or both must use parity-plus-spin tensors.

QP results offer the same functions. The operator may be a `FermionOperator` in canonical coordinates or a probe name (`h.probe(name)`; see the *probe list* (page 80)); any other name selects a model observable. The ket result may belong to another sector or QP cutoff.

```
odd = solve(h, cutoff=2, sector=Sector(1, 1))
even = solve(h, cutoff=2, sector=Sector(0, 0), options={"eigenpairs": 2})
addition = odd.matrix_elements("d_up_dagger", even)   # <D|d_up^dagger|S>,
↪<D|d_up^dagger|S*>
pair = even.matrix_elements("current")                 # includes <S|dH/dphi|S*>
```

Requesting matrix elements between sectors that no term of the operator connects raises an error. For single vectors, `ProjectedOperator` gives direct access:

```
from qdjj_solver.qp_solver import ProjectedOperator

spin = ProjectedOperator(h.observables["impurity_spin_z"], qp.basis)
mean_spin = spin.expectation(qp.vectors[:, 0])
```

`ProjectedOperator(operator, ket_result.basis).transition(bra, bra_basis, ket)` acts between compatible Fock bases, allowing different sectors and cutoffs. The caller is responsible for supplying operators and vectors in the same canonical coordinates, in addition to satisfying the dimension and spin-label checks. Use `h.physical_operator(electron_operator)` to transform physical electron operators through an uncompressed model's electron-to-canonical transformation.

3.1.5 Local orbital density matrices

`orbital_density_matrix` works with both QP and DMRG eigenstates. Orbital and state indices are zero-based. Its basis is **empty, up, down, up-down**, with the doubly occupied state defined by creating up before down.

```
from qdjj_solver import orbital_density_matrix, orbital_density_properties

rho = orbital_density_matrix(qp, orbital=0, state=0)  # shape (4, 4)
properties = orbital_density_properties(rho)
print(properties["empty"], properties["single"], properties["double"])
print(properties["pair_amplitude"], properties["plus"], properties["minus"])
all_rho = orbital_density_matrix(qp, orbital=0, state=None)  # shape (k, 4, 4)
# Equal mixture of a complete two-state multiplet, for a result with k=2:
# mixed = orbital_density_matrix(qp, 0, state=None, weights=[1, 1])
```

`weights` must be real, finite, nonnegative, one per retained state, and have a positive sum. They are normalized to sum one and describe an **incoherent** mixture. A complete, equally weighted degenerate eigenspace has a basis-independent RDM; an individual degenerate eigenvector generally does not.

The default `physical=True` transforms physical impurity-electron operators through the saved coordinate map. The impurity-only eta map also supports verified paired-mode compression. For a manually constructed canonical-mode Hamiltonian without an electron map, pass `physical=False` explicitly. This API concerns impurity orbitals, not arbitrary reservoir subsystems.

`orbital_density_operators(orbital=0, hamiltonian=h)` supplies the canonical operators for independent measurement or matrix caching. It returns a mapping `(row,column) -> operator`, where the expectation equals `rho[row,column]`. Without `hamiltonian`, it returns physical-electron expressions. Composite projectors are built before QP truncation, preserving vacuum contractions. See *density-matrix conventions* (page 11) for the signs, coherences, and paired-state probabilities.

3.1.6 Floquet calculations

Periodic drives of model parameters are described by `PeriodicHamiltonian`, which samples a model builder over one period:

```
from qdjj_solver import ParameterDrive, PeriodicHamiltonian, reference_model
from qdjj_solver.qp_solver import floquet_propagator, rwa_model, solve_floquet

periodic = PeriodicHamiltonian.from_parameters(
    reference_model, dict(bath=bath, phi=.8, symmetry=False, compress=False),
    ParameterDrive.cosine("phi", .05), omega=.57)
floquet = solve_floquet(periodic, cutoff=2, sector=sector,␣
↪options=dict(floquet_cutoff=4, states=2))
```

`solve_floquet`, `floquet_propagator` and `rwa_model` return `FloquetStates` with quasienergies, harmonic weights, static-state weights and Floquet modes. `periodic.van_vleck(order)` returns a static high-frequency `Hamiltonian` for either backend. Options are collected in `FloquetOptions` (*option reference* (page 57)). Persistence through `save_result` currently supports static QP and DMRG eigenstates. Conventions, methods, result fields and convergence checks are described in *Floquet calculations* (page 41).

3.1.7 Starting states and continuation

QP iterative solvers accept a finite, nonzero vector of shape `(D,)` in the target basis ordering. For example, refine the same sector and cutoff with:

```
continued_qp = solve(h, cutoff=2, sector=sector,
                     options={"method": "sparse", "tolerance": 1e-12},
                     initial=qp.vectors[:, 0])
```

The vector is used by sparse/matrix-free iterations. The block eigensolver also accepts a starting array of shape `(D, n)` and supplements it with random columns up to the requested number of states. Dense and Schur methods compute their states directly. QP continuation across cutoffs requires an explicit embedding into the new Fock basis. A *QP workspace* (page 29) reuses compatible preparation and starting states across repeated calculations.

DMRG takes a prior DMRG `Eigenstates` object:

```
continued_mps = solve(h, sector=sector, backend="dmrg", initial=mps,
                      options={"eigenpairs": 2, "chi_max": 64,
                               "require_convergence": True})
```

The canonical basis, MPS mode ordering and grouping, types of conserved quantum numbers, and sector must match. Parameter changes are allowed when the electron-to-canonical transformation remains identical; changing bath nodes generally changes that transformation. The first trial for each previously calculated eigenstate uses its previous MPS; other trials remain independent.

3.2 Input files and terminal commands

The `qdjj-solver` command reads a JSON input file, with settings enclosed in `{...}`. Version 1 accepts the following entries:

Field	JSON type	When omitted
`format_version`	Integer `1`	`1`
`backend`	String `"qp"` or `"dmrg"`	`"qp"`
`cutoff`	Integer from `0` through `h.bath_modes`, or `null` for full space	`2` for QP; `null` for DMRG
`model`	Object	*Reference-model defaults* (page 7), with `compress=false` in `qdjj-solver`
`bath`	Object	`{"pairs": 8}` for a reference model; other model kinds specify their own modes/reservoirs
`sector`	Object	`{"parity": 1, "twice_sz": 1}`; unrestricted eta/number
`solver`	Object	Chosen method's `SolverOptions` defaults

Unknown setting names are reported with their location, such as `model.impurity.terms[0]`, to help identify errors in the input. Use JSON `true`/`false` for yes/no settings and integers for the cutoff and sector labels. Where a bath or model specification accepts `metadata`, that dictionary can contain your own notes. Place such notes inside the supported bath or model entry. See the examples. The command-line option `--backend` overrides the `backend` of the input file. DMRG requires an omitted or null `cutoff`. Cutoff and solver options must be valid for the selected method, also when `--backend` overrides it. The

`qdjj-solver` command defaults to QP with cutoff two; Python `solve` defaults instead to full finite space (`cutoff=None`). DMRG `--resume` names a result JSON with a companion saved MPS file.

`--resume` initializes a new optimization from the previously saved states; settings such as `chi_max` and tolerances come from the new configuration. The saved file supplies completed MPS states as the initial wavefunctions for this new optimization. The canonical basis, MPS mode ordering and grouping, sector, and types of conserved quantum numbers must be compatible. See `examples/dmrg_refined.json` for a refinement of `examples/dmrg.json`.

The command uses an uncompressed reference-model bath by default for both methods. Switching only the method therefore preserves the finite Hamiltonian; any dark-mode restriction must be requested explicitly with `compress=true`.

Model kinds:

- `reference`: parameters for *`reference_model`* (page 7), plus a top-level bath.
- `multi-orbital`: `impurity` with `orbitals` and polynomial `terms`, `reservoirs`, `tunneling`, and optional `direct` entries with `leads` and `matrix`. An optional `direct_derivatives` entry associates observable names with lists in the same contact format. Duplicate direct contacts are rejected. Other *`make_model`* (page 8) options, including `phases`, `phase_velocities`, `bath_reference`, `eta_basis`, `compress_pairs`, and `coefficient_tolerance`, also belong inside `model`.
- `hamiltonian`: the fields of `Hamiltonian.record()`, plus `kind`.

Bath kinds are `cosh-grid` (the default; the arguments of `cosh_grid`), `surrogate` (the arguments of `fit_surrogate`), `chain-expansion` (`levels`, `delta`, `bandwidth` and the optional boolean `wide_band`), and `discrete` (`xi`, `weights`, `delta`, `bandwidth` and optional `metadata`). Complex arrays use `{"real": [...], "imag": [...]}`. Impurity observables may also be given as lists of fermion-operator terms in a multiorbital input file.

For example, this complete QP input describes two interacting orbitals and two reservoirs. Its literal impurity terms specify the energy zero; include the constant explicitly to reproduce the convention of `Impurity.anderson`. Operator indices in `terms` are signed and one-based, while reservoir indices in `leads` are zero-based:

```
{
  "format_version": 1,
  "backend": "qp",
  "model": {
    "kind": "multi-orbital",
    "impurity": {
      "orbitals": 2,
      "terms": [
        {"operators": [1, -1], "real": -0.4, "imag": 0.0},
        {"operators": [2, -2], "real": -0.4, "imag": 0.0},
        {"operators": [3, -3], "real": -0.3, "imag": 0.0},
        {"operators": [4, -4], "real": -0.3, "imag": 0.0},
        {"operators": [1, 2, -2, -1], "real": 0.8, "imag": 0.0},
        {"operators": [3, 4, -4, -3], "real": 0.6, "imag": 0.0}
      ]
    },
    "reservoirs": [
      {"kind": "cosh-grid", "pairs": 1, "delta": 1.0, "bandwidth": 5.0},
      {"kind": "cosh-grid", "pairs": 1, "delta": 1.0, "bandwidth": 5.0}
    ],
    "tunneling": [
      [[0.2, 0.0, 0.0, 0.0], [0.0, 0.2, 0.0, 0.0]],
      [[0.0, 0.0, 0.15, 0.0], [0.0, 0.0, 0.0, 0.15]]
    ],
    "phases": [-0.4, 0.4],
    "phase_velocities": [-0.5, 0.5],
    "direct": [{"leads": [0, 1], "matrix": [[0.02, 0.0], [0.0, 0.02]]}],
    "bath_reference": "isolated",
    "eta_basis": false,
    "compress_pairs": false,
    "coefficient_tolerance": 1e-14
  },
  "cutoff": 2,
  "sector": {"parity": 0, "twice_sz": 0},
  "solver": {"eigenpairs": 1}
}
```

The bath representation guide provides examples of changing bath representations and reusing saved fits, all generator parameters, and accuracy/time benchmarks. The ChE guide explains its low-frequency Padé construction and the density-normalization meaning of `bandwidth` when `wide_band=true`.

3.3 Saved results and reproducibility

For a complete scan, scalar-only analysis and saved-wavefunction workflow, see *Parameter scans, saved results and reproducible analysis* (page 36). The *record reference* (page 96) specifies nested fields and companion files.

The JSON format `qdjj-eigenstates`, version 1, contains the numerical method, energies, residuals, observables, elapsed times, solver settings, symmetry sectors, Hilbert-space restrictions, and convergence diagnostics. It also saves the complete Hamiltonian, software versions, and SHA-256 checksums identifying the Hamiltonian and the solver files used. Wavefunctions are optional and saved in a separate file. Non-finite JSON numbers are rejected; undefined initial sweep differences are stored

as `null`. The Python result's `metadata` is stored under `calculation` in JSON, and `timings` under `timings_seconds`.

`qdjj_solver.common.io.source_provenance()` calculates checksums of the Python modules and C++ library actually used by the calculation. It stores their package-relative names, so the result can be compared between computers with different installation directories.

```
from qdjj_solver import save_result, load_result
save_result(qp, "results/state.json", save_states=True)
restored = load_result("results/state.json")
```

`qdjj_solver.save_result` uses this format for both methods. `load_result` identifies the method from the saved file and requires the companion wavefunctions. Results saved without wavefunctions can still be read with Python's standard `json` module. `restored.backend` is `"qp"` or `"dmrg"`. The base installation suffices for loading QP states. To move a calculation, copy its JSON and the companion file named in `record["state_artifact"]["file"]` together into the destination directory.

The DMRG wavefunction file `state.<unique-id>.mps.h5` is identified by `kind="tenpy-mps-hdf5"`. Its checksum and recorded basis identify the matching JSON result and prevent combining files from different saves. Reading the MPS depends on TeNPy's support for its saved-state format; the JSON remains readable independently. Use TeNPy MPS files from trusted calculations.

QP wavefunction files have `kind="fock-amplitudes-npz"` and contain energies, eigenvector columns, Fock occupations, and QP counts. They are read with NumPy's `allow_pickle=False`. New NPZ files identify the matching JSON by its `result_sha256` checksum. Loading verifies both file checksums, the energies, basis ordering, dimensions, and orthonormality. Earlier four-array NPZ files remain readable, with checksum, energy, and basis checks.

Always obtain the companion filename from the JSON record. Each save uses a unique wavefunction filename and replaces the JSON only after that file has been written successfully. A failed save therefore preserves the previous result. Older wavefunction files remain available for copies of the earlier JSON. They can be removed once they are no longer needed by a retained result or a running analysis. An interrupted save may also leave an unused wavefunction file.

Version 1 allows additional information to be saved. Loading rejects unsupported JSON or wavefunction-format versions. Reading MPS files also depends on the installed TeNPy version.

3.4 Troubleshooting and exit status

Exit codes, common error messages and their remedies are collected in *troubleshooting* (page 50).

CHAPTER

FOUR

STATIC STATES, OBSERVABLES AND TRANSITIONS

The aim is to find the ground-state parity, the lowest parity-changing and pair excitations, and the current of a superconducting dot junction. This chapter uses the reference junction of the physics manual and Tutorial 01. Its finite basis is larger than that of the installation example, but the calculation remains inexpensive at QP cutoff two.

4.1 Declare the physical and numerical problem

The shared input is `examples/01_subgap_states/input/reference.json`. It is also a complete unified-CLI input for the even sector. Its physical model and bath parameters are also used by the tutorial and physics-manual figure scripts.

```
{
  "format_version": 1,
  "backend": "qp",
  "bath": {
    "kind": "cosh-grid",
    "pairs": 4,
    "delta": 1.0,
    "bandwidth": 10.0
  },
  "model": {
    "kind": "reference",
    "u": 1.0,
    "gamma": 0.5,
    "phi": 0.8,
    "detuning": 0.2,
    "field": 0.0,
    "symmetry": false,
    "compress": false
  },
  "sector": {
    "parity": 0,
    "twice_sz": 0
  },
  "cutoff": 2,
  "solver": {
    "eigenpairs": 2
  }
}
```

`model` and `bath` define a finite physical Hamiltonian. `sector`, `cutoff` and `solver` define the calculation performed on it. In particular, changing the cutoff changes the retained many-body space at fixed bath; bath refinement changes the finite Hamiltonian approximating the continuum. The labels and units are explained in *physical models and conventions* (page 5).

4.2 Run the complete example

Install the base package, then run from the source root:

```
python -B examples/handbook/static.py --output runs/handbook/static
qdjj-solver solve examples/01_subgap_states/input/reference.json --output␣
↪runs/handbook/cli-even.json --save-states
```

The complete Python script reads that JSON, constructs the model, solves even and both odd spin sectors, evaluates matrix elements, and verifies that saving and reloading the states preserves the results. The CLI command computes the matching even states; use the Python script for the full cross-sector analysis.

The relevant part of the tested script is shown below. `config` is the parsed input above and `output` is the directory selected by `--output`; imports and argument handling are included in the downloadable script.

```
h = model_from_config(config, default_compress=False)
even = solve(h, cutoff=config["cutoff"], sector=Sector(0, 0), options={
↪"eigenpairs": 2})
odd = solve(h, cutoff=config["cutoff"], sector=Sector(1, 1))
partner = solve(h, cutoff=config["cutoff"], sector=Sector(1, -1))
signed_gap = float(odd.energies[0] - even.energies[0])
pair_gap = float(even.energies[1] - even.energies[0])
addition = odd.matrix_elements("d_up_dagger", even)
pair = even.matrix_elements("current")
current = float(np.real(even.observables["phase_derivative"][0]))
save_result(even, output/"even.json", save_states=True)
restored = load_result(output/"even.json")
np.testing.assert_allclose(restored.energies, even.energies, rtol=0, atol=1e-13)
```

4.3 Read the energies before plotting

Quantity	Expected value	Meaning
`signed_gap`	0.09578810633	Signed doublet-minus-singlet energy, in gap units
`pair_gap`	0.7159930829	Even pair excitation, in gap units
`current`	0.05034773273	Singlet current in units of 2e Delta/hbar
`ground_parity`	0	Global ground-state parity (zero is even)
`addition_weight`	0.4821031006	Squared dot spin-up electron-addition matrix element

continues on next page

Table 1 – continued from previous page

Quantity	Expected value	Meaning
`pair_weight`	0.0007372261697	Squared phase-derivative pair matrix element, in squared gap units

At this point the lowest even state is the global ground state. The positive quantity `signed_gap = E_D - E_S` gives the lowest electron-addition/removal excitation. When a sweep makes it negative, the doublet becomes the ground state; the physical excitation energy is then its absolute value. The even-sector pair excitation is `E_Sstar - E_S` and is evaluated independently of the parity-changing gap.

At zero field the two doublet spin projections are degenerate; the example checks this. In a field, compute both projections before identifying the global minimum. For spin-mixing Hamiltonians choose `twice_sz=None` and classify states using the appropriate observables in the resulting space of all spin projections.

4.4 Interpret observables and array axes

`result.energies[i]`, `result.residuals[i]` and `result.observables[name][i]` refer to the same zero-based state index. QP vectors are columns of `result.vectors`, with rows in `result.basis` order. `qp_weights[i, n]` gives the probability of finding `n` bath quasiparticles in state `i`. Physical dot charge is a local observable, while total parity labels the full dot-plus-reservoir state and is specified by the chosen sector.

`bra_result.matrix_elements(operator, ket_result)[i, j]` has bra states as rows and ket states as columns. Thus the shape of `addition` in the example is one odd final state by two even initial states. For the pair matrix, the transition from the lower to the upper even state is `[1, 0]`. Its squared magnitude is a probe strength. The golden-rule transition rate also depends on the probe's amplitude and frequency; initial-state populations determine the contributions to the measured response.

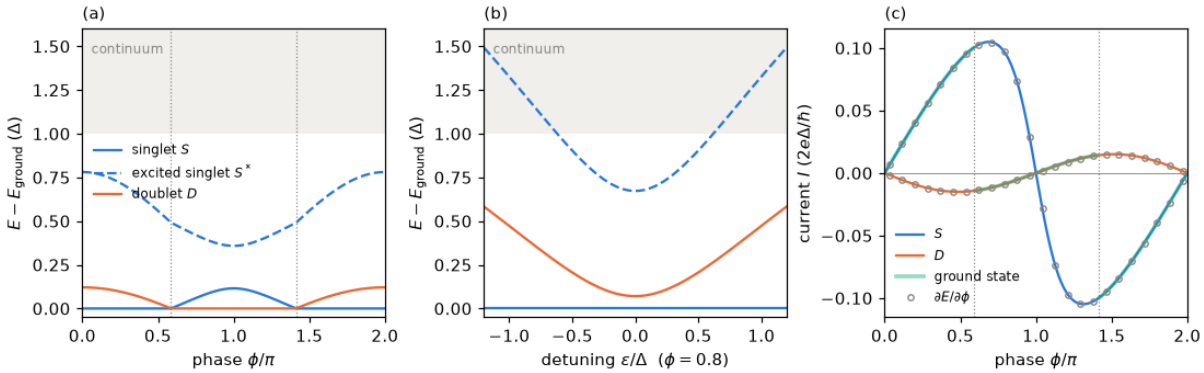
Use named probes such as `d_up_dagger` to obtain physical dot-electron operators in the Hamiltonian's canonical coordinates. For an arbitrary electron operator, use `h.physical_operator(...)` with an uncompressed declared electron map. Multiplying individually projected operators can omit virtual intermediate states: form the full fermionic product before projecting it.

4.5 Convert the current to physical units

The observable `phase_derivative` and the named probe `current` represent the derivative of the Hamiltonian along its declared phase coordinate. Multiply its expectation value by $2e/\hbar$ to obtain the physical current. With all energies normalized to the gap, the example's number is $I/(2e\Delta/\hbar)$.

Check a branch current against a finite difference of the **same sector's** energy. Across a ground-state parity crossing, first follow each smooth sector branch, then take the lower energy. Differentiating an excitation energy gives a difference of state currents. At the crossing, evaluate the appropriate one-sided ground-state derivatives, which can differ.

4.6 Sweep and assess the result



This Tutorial 01 figure uses the same reference parameters and QP cutoff. Its shading marks the lowest continuum threshold across sectors; determine the threshold for each probe from its allowed final sector. The full tutorial additionally checks current derivatives, spin degeneracy, particle–hole symmetry and the decoupled-dot limit. These are independent physical checks of the selected finite calculation. Establish suitable cutoffs and bath sizes by refinement in each parameter regime.

Next use the *scan-and-save workflow* (page 36), then vary the *QP cutoff* (page 28) and *bath resolution* (page 25) separately. The *result reference* (page 74) describes all fields; the *record reference* (page 96) describes their on-disk representation.

CHAPTER

FIVE

CHOOSING AND REFINING SUPERCONDUCTING RESERVOIRS

The reservoir approximation determines which continuum problem a finite calculation represents. Choose the physical gap and half-bandwidth first, then choose a representation and demonstrate convergence for the observable you intend to report. The number of leads, signed normal levels, spin modes and retained quasiparticles must be distinguished.

5.1 Choose a representation

Representation	Typical use	What to inspect
`cosh_grid`	A transparent quadrature for a flat finite band	Level-pair refinement and the normal-measure diagnostic
`fit_surrogate`	Compact fits to the imaginary-frequency hybridization	Fitting interval, objective, initial guesses, seed and independent validation error
`chain_expansion`	Deterministic Padé-derived reservoirs, returned in star coordinates	Signed level count, odd/even sequences and finite/wide-band target
`DiscreteBath`	Explicit or independently designed quadrature	Nodes, positive weights, density normalization and provenance

The *constructor reference* (page 61) lists parameters and defaults. The detailed bath representation guide explains the quadratures and complete benchmarks; chain expansion gives the Padé construction. This chapter explains how to choose and assess a reservoir in practice.

5.2 Construct and retain explicit coefficients

This self-contained example uses the same physical energy scales for all three methods:

```
import json
from pathlib import Path
from qdjj_solver import DiscreteBath, chain_expansion, cosh_grid, fit_surrogate

grid = cosh_grid(pairs=2, delta=1., bandwidth=10.)
surrogate = fit_surrogate(levels=4, delta=1., bandwidth=10., seed=1729)
chain = chain_expansion(levels=4, delta=1., bandwidth=10.)
Path("bath.json").write_text(json.dumps(grid.record(), indent=2))
restored = DiscreteBath.from_record(json.loads(Path("bath.json").read_text()))
assert restored.levels == grid.levels
```

Four signed normal levels mean eight Bogoliubov spin modes **per lead**. For `cosh_grid`, `pairs=2` produces those four levels; for the other constructors, `levels=4` specifies them directly. Equal level counts provide an equal-size comparison; compare physical observables to assess the accuracy of each bath.

The weights represent the normal-state measure. Preserve their quadrature values to retain the contact's physical hybridization, including for a coarse rule. Saving the explicit coefficients separates solver comparisons from fitting randomness or numerical-library differences in bath construction.

5.3 Inspect the hybridization before solving

```
import numpy as np
from qdjj_solver.common.baths import discrete_g, hybridization_g

frequencies = np.geomspace(1e-3, 100., 101)
target = hybridization_g(frequencies, delta=1., bandwidth=10.)
for name, bath in [("cosh", grid), ("surrogate", surrogate), ("chain", chain)]:
    error = np.max(abs(discrete_g(bath, frequencies) - target))
    print(name, bath.levels, error)
```

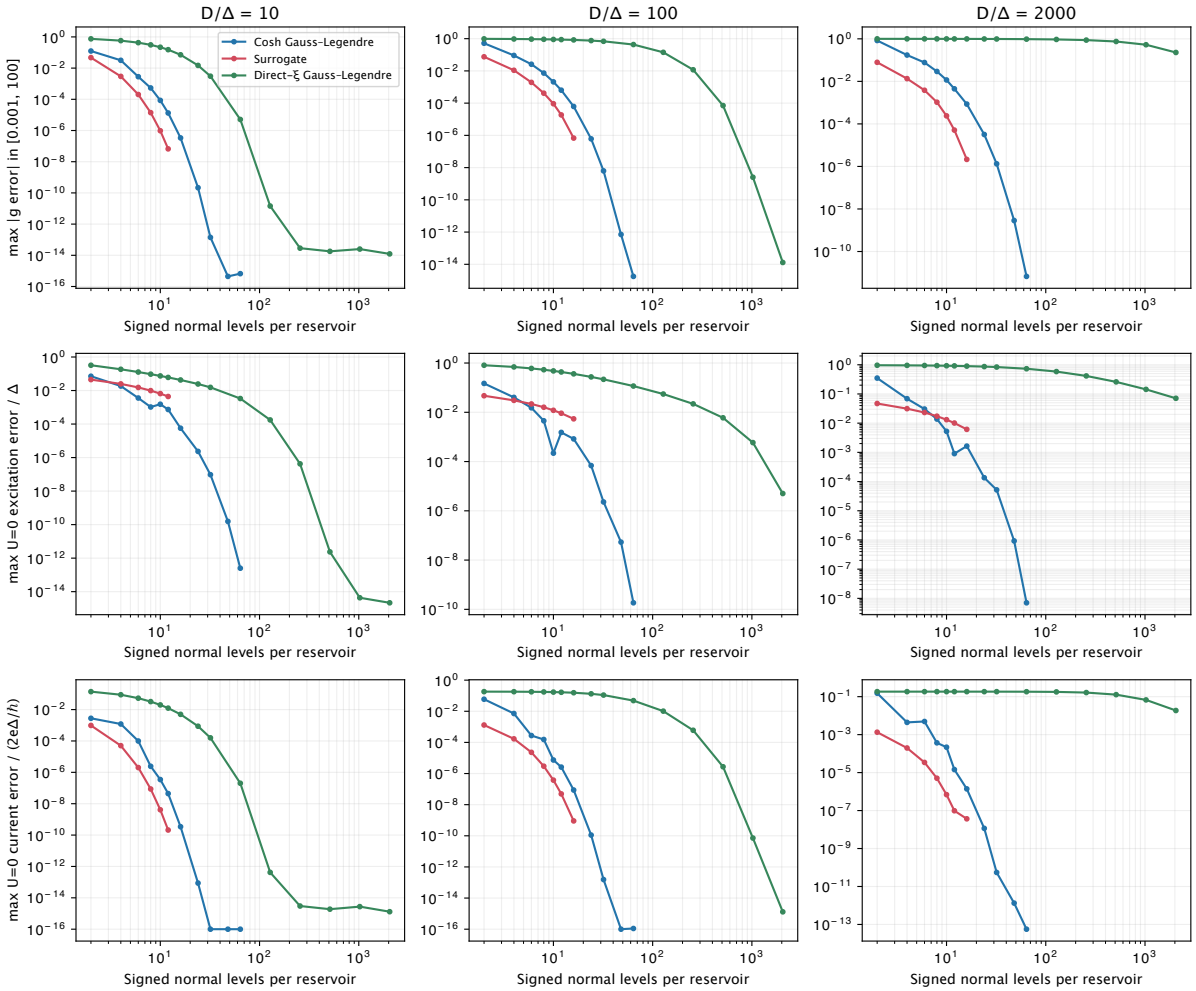
This diagnostic measures the error at the sampled imaginary frequencies. Also check zero frequency, the fitting interval and relevant frequencies outside it. Assess the accuracy of a gap-edge excitation or current by refining the bath and comparing that observable. `max_nfev` sets the computational budget for each fit start; the resulting accuracy is determined by these separate checks.

5.4 Refine the physical observable

1. Fix the physical Hamiltonian and the target continuum: gap, bandwidth, couplings, phase, field, detuning and energy reference.
2. Choose a bath sequence. For surrogate and chain families inspect odd and even sizes separately where their structure differs.
3. At each size, converge the finite problem using the QP cutoff or MPS bond limit. Record the controls needed to reach convergence at that size.
4. Compare the actual excitation, current or correlation being reported.
5. Repeat with an independently motivated representation when near the gap edge or where a small difference decides a parity boundary.
6. Report unresolved changes and the scope of the reference calculation.

Keep the bath fixed when comparing QP and DMRG, and keep the physical bandwidth fixed during a finite-band refinement. `wide_band=True` in `chain_expansion` selects an infinite-band target; its bandwidth parameter then specifies only the density normalization.

5.5 Interpret comparative evidence



This figure presents selected results from the complete benchmark report, which specifies the models, quadratures, independent references and hardware. Kernel, excitation and current errors need not rank the methods identically. Timings in that study characterize the recorded one-thread Apple M1 Max environment. Measure runtimes in the intended environment when planning a calculation. The data distinguish equal-size comparisons from cost at an achieved observable accuracy.

A finite bath replaces continua by discrete states. Its lowest excitation can lie above the physical gap edge. Use the appropriate continuum threshold when assessing bound-state binding. A change of coordinates from a star to a chain preserves a specified finite Hamiltonian, while removing modes or changing coefficients changes the physical approximation.

5.6 Connect to input files

The top-level `bath` object describes a reference model's reservoir. A general multiorbital model has its reservoirs under `model.reservoirs`. The *configuration reference* (page 91) specifies all four kinds and the defaults used when fields are omitted. Changing a bath usually changes the electron-to-canonical transformation, so check compatibility before reusing saved state vectors or MPS after such a change.

CHAPTER

SIX

QP CALCULATIONS AND OBSERVABLE-SPECIFIC CONVERGENCE

The QP expansion retains every impurity configuration but restricts the total number of bath quasiparticles. Increasing the cutoff enlarges a variational space for the **same finite Hamiltonian**, keeping the bath fixed. Diagonalization optimizes all amplitudes within the retained space, including repeated tunnelling processes.

6.1 Estimate the basis dimension before allocating memory

```
from qdjj_solver import Sector, cosh_grid, reference_model, solve
from qdjj_solver.qp_solver import dimension

h = reference_model(cosh_grid(pairs=1, bandwidth=5.), u=1.6,
                    gamma=.3, phi=.8, symmetry=False, compress=False)
sector = Sector(1, 1)
for q in [1, 2, 3, 4, h.bath_modes]:
    print(q, dimension(h.nimp, h.spins, q, sector, h.eta_labels))
```

`dimension` counts basis states without explicitly constructing them. It takes an integer cutoff; use `h.bath_modes` for the full finite-space count. A solver call instead accepts `cutoff=None` as the full-space shorthand. Counts grow combinatorially with bath size and cutoff, so start with a deliberately small case.

The automatic eigensolver chooses dense, sparse or matrix-free operation based on dimension. Matrix-free operation reduces matrix storage but still needs a basis, iterative vectors and workspace for applying operators. `max_memory_gib` limits estimated basis and matrix storage; allow additional memory for the running process. The *option reference* (page 53) lists thresholds and controls.

6.2 Refine the cutoff at fixed bath

```
import numpy as np

exact = solve(h, cutoff=None, sector=sector)
for q in [1, 2, 3, 4]:
    state = solve(h, cutoff=q, sector=sector)
    print(q, state.energies[0], state.residuals[0],
          state.observables["impurity_spin_z"][0], state.qp_weights[0, -1])
    assert state.energies[0] >= exact.energies[0] - 1e-10
```

For nested spaces and an accurately solved ground state of one sector, the energy approaches the finite-space value from above. Differences between two sector energies can approach their converged values from either side. A parity-transition location can therefore move even when both absolute energies look stable.

The projected residual measures eigensolver accuracy for $P_q H P_q$. Comparing successive cutoffs measures the effect of including additional QP configurations in the variational space.

6.3 Reuse preparation and check spectrum coverage

`QPWorkspace` reuses compatible bases, projected operators and matrices within a memory budget. Dense calculations can reuse the complete eigensystem when the requested number of states or measured observables changes:

```
from qdjj_solver.qp_solver import ExecutionContext, QPWorkspace, SolverOptions

workspace = QPWorkspace(max_memory_gib=0.1)
options = SolverOptions(method="dense", eigenpairs=2)
events = []
prepared = workspace.solve(h, cutoff=2, sector=sector, options=options,
                           context=ExecutionContext(progress=events.append))
reused = workspace.solve(h, cutoff=2, sector=sector, options=options,
                         observables=["impurity_spin_z"])
np.testing.assert_allclose(reused.energies, prepared.energies, rtol=0,␣
↪atol=1e-12)
assert reused.metadata["reuse"]["eigensystem"]
assert not prepared.vectors.flags.writeable

edge = float((prepared.energies[0] + prepared.energies[1])/2)
count = workspace.count_below(h, 2, sector, edge, margin=1e-8)
print(count)  # below/through count states at the two boundaries edge +/- margin
assert count["status"] == "checked" and count["below"] == count["through"] == 1
workspace.clear()
```

Returned numerical arrays are read-only snapshots. Clearing the workspace releases its caches; retained results remain valid and count towards its live memory. A coefficient change can reuse the enumeration, while automatic starting vectors require matching physical coordinates, cutoff and sector. The `sparse`, `matrix-free` and `block` methods use these starting states when explicitly selected. They retain the same residual checks as fresh solves.

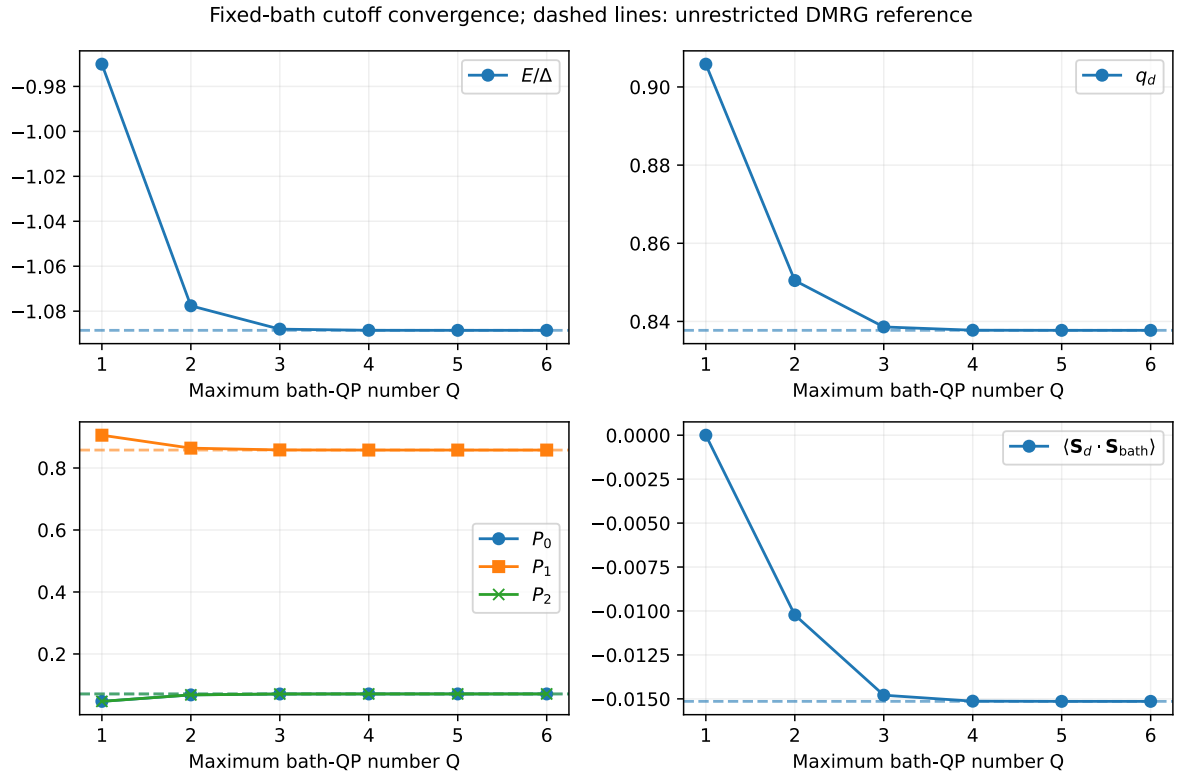
`ExecutionContext` reports stages and elapsed seconds. A `cancelled` callback can request `CalculationCancelled` at a checkpoint; native QP loops also check Python signals. Completed results remain usable after cancellation.

State counts use the complete cached dense spectrum when available. Otherwise, for dimensions within `count_limit` and the memory budget, an independent Hermitian LDL factorization checks the two energy boundaries. Inspect `status`: `"unavailable"` means a resource or preparation limit, and `"unresolved"` means the boundary count could not be validated. A checked count concerns this finite bath and QP cutoff; it does not determine continuum accuracy.

6.4 Decide which accuracy matters

Choose a tolerance for the reported quantity before claiming convergence. For an excitation, compare excitation differences; for a current, compare the phase derivative; for a correlation, compare the correlation itself. Assess wavefunction and observable accuracy through their own convergence checks.

The observable-convergence study provides a useful example:



That study uses a different junction from the installation example: interaction two, total hybridization 0.4, half-bandwidth 100 and phase $3\pi/5$, in a selected odd eta sector with dark modes frozen. Its full report specifies finite-bath and MPS checks. At two QPs the local-spin discrepancy is about 1.52%, while the dot–bath spin-correlation magnitude differs by about 32.46%. Those are properties of that recorded case; determine the cutoff error for each new parameter regime.

The probability at the QP cutoff is a useful diagnostic of the state composition. Observable-error estimates require explicit refinement: enlarging the space changes amplitudes already retained, and observables can connect QP sectors through coherent matrix elements. Store the distribution alongside the refinement results.

6.5 Recognize numerical rather than truncation failures

- Increase `ncv` or `maxiter` to converge the iterative solver before assessing the QP-cutoff error.
- At exact degeneracy, a single-vector iteration can miss repeated eigenvalues. Resolve symmetries or compare the complete subspace with a small dense reference calculation.
- For an empty or incompatible sector, check the model's symmetries and choose compatible quantum numbers.
- Requesting most eigenpairs can force dense diagonalization. Estimate storage for the actual requested problem, including Floquet harmonic blocks if used.

- Compare states in compatible coordinates and basis orderings. Reusing a previous vector after a cutoff change requires an explicit embedding into the new basis.

The numerical guide gives eigensolver and residual details. Then refine the *bath* (page 25) independently, at fixed physical parameters and adequate many-body convergence.

CHAPTER

SEVEN

DMRG STATES AND BOND-DIMENSION REFINEMENT

DMRG represents a correlated state as a matrix-product state (MPS). The bond dimension limits the entanglement represented by the MPS while retaining all bath-QP numbers. The same finite `Hamiltonian` can be passed to QP and DMRG. Matching its bath, coordinates, compression and sector is essential for a meaningful comparison.

7.1 Start with a small reference problem

Install `.[dmrg,plots]` from the source root and run:

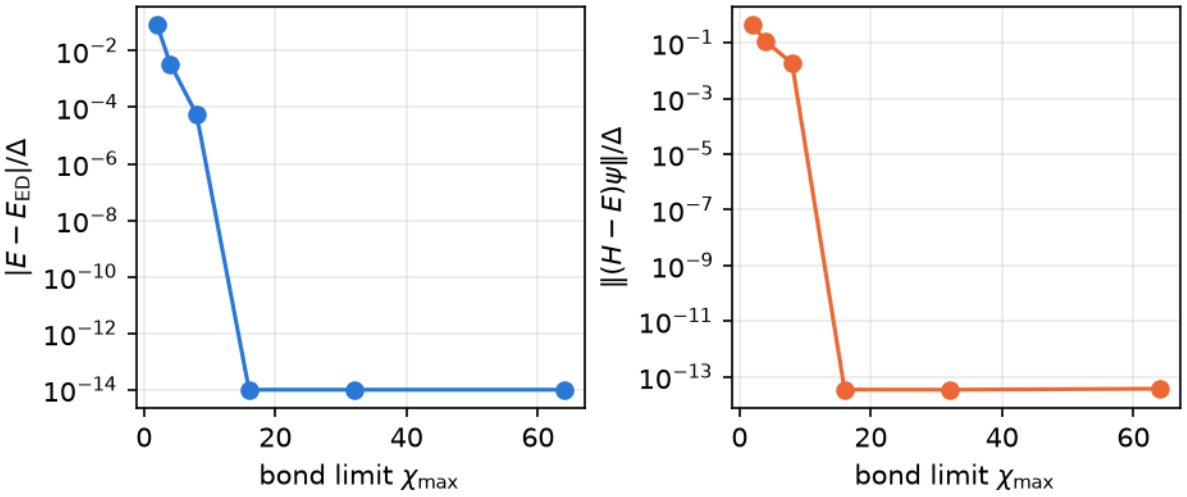
```
python -B examples/11_dmrg_convergence/run.py --output runs/dmrg-ladder
```

This complete tutorial uses the small installation-test Hamiltonian, retains all bath modes, and compares with full ED of the same finite Hamiltonian. It uses a sequence of increasing bond limits, saves each completed step, and finally saves, reloads and continues the MPS. The full profile adds intermediate and final steps:

```
python -B examples/11_dmrg_convergence/run.py --profile full --output␣
↪runs/dmrg-ladder-full
```

Both commands use a fixed seed and one computational thread. Small bond limits are intentionally inadequate: TeNPy can print truncation/canonical-form notices at those steps. The saved diagnostics show how the approximation improves. The final result must pass the residual and empirical-stability checks at the stated tolerances. All diagnostic messages remain visible throughout the refinement.

7.2 Interpret the refinement sequence



The energy discrepancy is relative to full ED on the same bath. The residual is the norm of the full finite-Hamiltonian eigenvalue equation. Values below 1e-14 are placed at that plotting floor. The two curves quantify different aspects of finite-Hamiltonian accuracy; bath refinement assesses continuum accuracy.

The first coarse state is inaccurate and marked unconverged; the refined energy agrees with ED within the checked 1e-9 tolerance. `history.json` records the requested and actual bond dimensions, energies, charge, residuals and overlap errors for each completed step. The final assessment is also saved in `results.json`.

7.3 Which controls should be changed?

Question	Controls and evidence
Is the state representation large enough?	Increase `chi_max`; compare energy and the observable being reported
Has each optimization converged?	Inspect sweep histories; vary the number of sweeps and local-solver precision
Is an excited state reliable?	Use independent trials; inspect energy ordering and the full Gram matrix
Is the finite eigenvalue equation satisfied?	Evaluate `residuals`; inspect `finite_problem_converged`
Does the bath represent the target continuum?	Refine the bath separately, with fixed physical parameters

`require_convergence=False`, the default, returns coarse states with their diagnostics. Use `True` for a calculation that must satisfy finite-problem criteria. The residual calculation has temporary memory costs and can be more demanding than an expectation value. Turning it off leaves convergence unverified.

The local Lanczos probability tolerance and the full-state residual tolerance are distinct. Convergence requires both adequate bond dimension and sufficiently accurate local solves. Increasing the bond dimension can reveal an optimization precision limit that was previously hidden by truncation.

7.4 Use the existing refinement helper

The tested tutorial uses the following code. Here `h` and `sector` specify the finite problem, and the function `archive`, defined in the complete script, saves each completed step:

```
options = SolverOptions(eigenpairs=1, seed_trials=1, seed=1729,
                        max_sweeps=40, min_sweeps=8, mixer_sweeps=4,
                        lanczos_probability_tolerance=1e-22)
chis = [2, 4, 16, 32] if profile == "quick" else [2, 4, 8, 16, 32, 64]
result, assessment = converge_bond_dimension(
    h, chis, sector=sector, options=options, stable_steps=1 if profile == "quick
↪" else 2,
    energy_tolerance=1e-9, observable_tolerance=1e-8, callback=archive)
if not assessment["empirical_convergence"]:
    raise RuntimeError("bond ladder did not converge; inspect history.json")
save_result(result, output/"state.json", save_states=True)
restored = load_result(output/"state.json")
continued = solve(h, sector=sector, options=SolverOptions(
    eigenpairs=1, chi_max=chis[-1], seed_trials=1, require_convergence=True,
    lanczos_probability_tolerance=1e-22), initial=restored)
```

`converge_bond_dimension` runs the supplied sequence of increasing bond limits and returns its **last result**, together with an assessment. It deliberately permits coarse unconverged steps, then checks finite-problem convergence and successive changes in energies and measured observables. The `empirical_convergence` flag reports whether the sequence met these criteria. Two stable steps are normally required; the quick teaching profile requests one, while the full profile requests two.

7.5 Request excited states deliberately

Set `eigenpairs` to the number of low states needed in the selected sector. They are optimized sequentially with penalties against previously found states. Different initial trials can converge to different candidate eigenstates. Inspect the diagnostics for each state and the overlap matrix, especially near degeneracies. Comparing retained subspaces is more meaningful than comparing individual eigenvectors inside an exactly degenerate multiplet.

`compare_states(previous, current)` reports a maximum-overlap assignment, energy/observable changes and singular values of the overlap matrix. Those singular values are invariant under rotations within the compared retained subspaces. Establishing the completeness of the low-energy spectrum requires additional state searches or comparison with an independent reference.

7.6 Save and continue

`save_result(..., save_states=True)` creates JSON plus a named HDF5 companion. `load_result` checks the record and restores the states; pass that result as `initial` to a new DMRG solve. The new call supplies its own tolerances and bond limit. Compatible continuation requires matching canonical coordinates, mode ordering, grouping, sector and types of conserved quantum numbers.

A checkpoint supplies completed states as initial wavefunctions for a new optimization. Bath changes usually change the canonical coordinates, so perform a new optimization when refining the bath. A parameter change is allowed when those coordinates and the MPS layout remain identical.

The numerical guide explains the diagnostic definitions, the *option reference* (page 54) lists every control, and the *record reference* (page 96) specifies saved fields and companion artifacts.

CHAPTER

EIGHT

PARAMETER SCANS, SAVED RESULTS AND REPRODUCIBLE ANALYSIS

A reproducible parameter scan retains the results at each point alongside the final curve. Record the physical model, approximation, sector, diagnostics and failed points so that a change of ground-state parity can be distinguished from a numerical failure. This chapter follows the reference junction of *static calculations* (page 21).

8.1 Run a phase scan

With `.[plots]` installed, run from the source root:

```
python -B examples/handbook/scan.py --output runs/handbook/scan
python -B examples/handbook/scan.py --output runs/handbook/scan --plot-only
```

The first command evaluates nine phases between zero and pi, saving both sector records and a scan table. The second redraws the figure directly from the saved records. The complete script accepts `--input` for another reference-junction JSON and `--points` for a denser grid.

The loop below is extracted from that script. `config`, `output` and `points` are its explicit arguments, and `records` starts as an empty list.

```
for index, phi in enumerate(np.linspace(0., np.pi, points)):
    point = {**config, "model": {**config["model"], "phi": float(phi)}}
    h = model_from_config(point, default_compress=False)
    row = dict(index=index, phi=float(phi), status="ok")
    try:
        even = solve(h, cutoff=point["cutoff"], sector=Sector(0, 0))
        odd = solve(h, cutoff=point["cutoff"], sector=Sector(1, 1))
        save_result(even, output/f"{index:03d}-even.json")
        save_result(odd, output/f"{index:03d}-odd.json")
        row.update(singlet=float(even.energies[0]), doublet=float(odd.
↪energies[0]),
                   current_singlet=float(np.real(even.observables[
↪"phase_derivative"][0])),
                   current_doublet=float(np.real(odd.observables[
↪"phase_derivative"][0])),
                   residual=max(float(even.residuals[0]), float(odd.
↪residuals[0])))
    except (ValueError, RuntimeError) as error:
        row.update(status="failed", error=str(error))
```

(continues on next page)

(continued from previous page)

```
    records.append(row)
    write_json(output/"scan.json", dict(input=config, points=records))
```

Results are saved immediately after each completed point. A rejected or failed calculation remains in `scan.json` together with its error message. Plotting a complete scan requires successful results at every point. For DMRG, also store and inspect the convergence flags to identify which states satisfy the finite-problem checks.

8.2 Identify branches before taking a minimum

Follow the lowest even energy and lowest odd energy separately. Their difference can cross zero; the nonnegative excitation is its absolute value. Construct the global ground-state current from the lower branch only after retaining both state-resolved currents. Refine scan density to locate crossings, and refine the bath and many-body representation to converge the underlying energy branches.

The energy ordering of excited states can change within a sector. To follow a particular state, compare symmetry labels, observables and overlaps in compatible coordinates. At exact degeneracy, compare the whole relevant subspace. Use these physical labels and overlaps to track states as their energy ordering changes.

8.3 Analyse scalar records without a solver

Scalar JSON records suffice for energies, previously calculated observables, diagnostics and plotting. From the output directory, ordinary Python can read them:

```
import json
from pathlib import Path

record = json.loads(Path("000-even.json").read_text())
print(record["energies"])
print(record["observables"]["phase_derivative"])
print(record["calculation"]["qp_cutoff"])
```

This output-dependent excerpt is tested against a freshly generated scan by the handbook checks. Resolve paths relative to the chosen output directory. Complex values use real/imaginary objects; `qdjj_solver.common.io.complex_array` decodes that representation when needed.

8.4 Reload states when new matrix elements are needed

Wavefunction amplitudes are stored in the companion state file. Reuse the static example's `even.json`, saved with `save_states=True`, to calculate a new matrix element:

```
from qdjj_solver import load_result

states = load_result("even.json")
charge = states.matrix_elements("charge")
print(abs(charge[1, 0])**2)
```

The companion filename is stored in `state_artifact.file`. Keep it alongside the JSON when moving the result. QP states use NPZ; DMRG uses TeNPy HDF5. Checksums associate the companion with

the scalar record. Preserve that pair and save derived results separately to retain a valid checkpoint. `save_result` currently supports static QP and DMRG eigenstates.

8.5 What belongs in the calculation record?

- The resolved physical model, bath coefficients and energy reference.
- Sector, QP cutoff or MPS settings, eigensolver tolerances and seeds.
- State-resolved energies and observables, with named units in derived tables.
- Residuals and other convergence diagnostics.
- Software/numerical-library information and implementation identity.
- The scan coordinate and a status for every attempted point.

The standard result format already records much of this information. The scan table adds the relationship between points and preserves failures. A repository commit identifies the source checkout; the stored implementation digest identifies the installed solver files actually used. They answer different provenance questions.

8.6 Continue safely

QP `initial` is a nonzero vector in the target basis ordering. Reusing a vector after changing cutoff needs an explicit embedding. DMRG `initial` is a prior result with compatible coordinates and layout. The detailed constraints are in *starting states and continuation* (page 16).

A reproducible report also states its limitations: describe the achieved convergence of the reported quantity and retain unresolved differences. Support continuum-accuracy estimates with explicit many-body and bath refinements. See the *configuration* (page 91) and *record* (page 96) appendices for complete field definitions.

CHAPTER

NINE

GENERAL IMPURITIES AND CUSTOM OPERATORS

`reference_model` is convenient for a symmetric one-orbital junction. `make_model` accepts an explicit impurity, reservoirs and contact matrices, and supports unequal gaps or couplings, multiple orbitals and spin mixing. Both return a `Hamiltonian`; choosing a solver remains a separate step.

9.1 Match a Python model to its JSON input

The complete files `examples/handbook/custom.py` and `custom.json` describe the same two-orbital impurity coupled to one superconducting reservoir. Run:

```
python -B examples/handbook/custom.py --output runs/custom-python.json
qdjj-solver solve examples/handbook/custom.json --output runs/custom-cli.json --
↪save-states
```

The Python script constructs the operator explicitly, compares it with the JSON construction, and compares the two sets of calculated energies. The relevant excerpt is:

```
n = [number(i) for i in range(4)]
impurity = Impurity(2, -.3*(n[0]+n[1]) + .2*(n[2]+n[3]) + n[0]*n[1])
bath = cosh_grid(pairs=1, delta=1., bandwidth=5.)
tunneling = np.array([[.3, 0., .1, 0.], [0., .3, 0., .1]])
h = make_model(impurity, [bath], [tunneling], phases=[0.])
result = solve(h, cutoff=2, sector=Sector(0, 0), options={"eigenpairs": 2})
save_result(result, output, save_states=True)
```

The first orbital has one-electron energy -0.3 and interaction one; the second has one-electron energy 0.2 and no local interaction. This explicitly written Hamiltonian retains the specified additive constant. To use the shifted Anderson energy convention, include its constant explicitly. Supply tunnelling amplitudes in energy units; convert hybridization widths using each reservoir's normal-state density of states.

9.2 Understand shapes and ordering

For `M` impurity orbitals and `L` reservoirs:

- Impurity modes are `(0 up, 0 down, 1 up, 1 down, ...)`.
- `tunneling[l]` has two reservoir-spin rows and `2*M` impurity-spin columns.
- `phases` has one value in radians per reservoir.
- A direct contact matrix for `(l, m)` has shape `(2, 2)` with `l < m`.

- Reservoir physical electron modes follow the impurity modes, ordered by reservoir, signed level and spin.

Unequal lead couplings are specified through the corresponding tunnelling matrices, using each bath's normal density convention. Off-diagonal spin entries can break spin-projection conservation, requiring `Sector(parity, None)`. Request eta or compression reductions only when their stated symmetry conditions are satisfied. Check those conditions explicitly for a multiorbital device.

9.3 Build operators before projecting them

`create(i)`, `annihilate(i)` and `number(i)` take zero-based mode indices. The serialized monomials instead use `+(i+1)` for creation and `-(i+1)` for annihilation. Their products are written in mathematical order, with the rightmost operator acting first. The algebra normal-orders complete products.

For example, form `number(0)*number(1)` as a fermionic polynomial, then evaluate its projection. This retains the contributions of virtual intermediate configurations at the cutoff boundary.

An impurity operator supplied to `Impurity` is in physical impurity coordinates. `make_model` transforms it along with the Hamiltonian. An arbitrary full-system electron operator supplied afterwards must be transformed with `h.physical_operator`. This requires an uncompressed declared electron map. Built-in observables include the contractions required by any supported dark-mode restriction.

9.4 Supply general one- and two-body integrals

`Impurity.from_integrals(one_body, interaction, constant)` uses spin-orbital indices and the convention

$$H_d = \sum_{ab} h_{ab} d_a^\dagger d_b + \frac{1}{4} \sum_{abcd} U_{abcd} d_a^\dagger d_b^\dagger d_d d_c + C.$$

The one-body matrix is square with even dimension; the interaction tensor has that dimension on all four axes. The factor one quarter assumes antisymmetrized two-electron integrals. Match operator ordering and normalization before importing integrals from another program. Small direct-electron-space checks are useful for validating a new model.

9.5 Direct finite Hamiltonians

`Hamiltonian(operator, nimp, spins, ...)` supports a completely explicit finite fermionic problem. Declare all modes, identify the first `nimp` as impurity modes, and retain the desired additive constant. The constructor checks Hermiticity and parity conservation; the solver checks compatibility with the requested sector.

For later coordinate-sensitive operations, supply consistent coordinate metadata or retain the model constructors' declarations. Verify that the bases describe the same physical modes as well as having compatible dimensions.

The *constructor reference* (page 61) gives every keyword, and the *configuration reference* (page 91) specifies complex arrays, operator-term records and contact lists. The API reference documents the methods for conversion, fingerprints and probe construction.

CHAPTER

TEN

FLOQUET CALCULATIONS

Periodic modulation of model parameters, such as the impurity level, the superconducting phase, the hybridization or the interaction, dresses the subgap states and splits them at resonances. This page explains how to specify a drive, which method to use, how to read the results, how to compute probe matrix elements and spectra, and how to check convergence. The calculation yields quasienergies, harmonic weights, Floquet modes and transition matrix elements. Specify Floquet-state populations explicitly; predicting steady-state populations requires an additional model of dissipation.

The theory is developed in the physics manual: chapters "Floquet theory for periodically driven systems" (`ch:floquet`), "Driving the junction" (`ch:driving`), "Numerical methods for driven problems" (`ch:drivenmethods`) and "Probing the driven junction" (`ch:probes`). The tutorials work through examples from an isolated dot to spectroscopy of the driven junction.

Sambe diagonalization, the one-period propagator and the rotating-wave model use the QP backend. The high-frequency effective Hamiltonian is an ordinary static `Hamiltonian`, so it can also be solved with DMRG.

10.1 Conventions

For a first driven calculation, read *Specifying a drive* (page 42) and *Methods* (page 44), then use the conventions below to interpret the output. Start from a checked *static calculation* (page 21), retain fixed coordinates, and identify the relevant undriven transition before choosing a frequency. *Probe matrix elements and spectra* (page 48) supplies a small, self-contained spectroscopy example.

Energies, the drive frequency Ω and tolerances are in the units of the Hamiltonian (typically $\Delta = 1$), with $\hbar = 1$. The period is $T = 2\pi/\Omega$ and the time origin is $t = 0$. A periodic Hamiltonian is stored as harmonics

$$H(t) = \sum_k H_k\, e^{ik\Omega t}, \qquad H_{-k} = H_k^\dagger, \qquad H_k = \frac{1}{T}\int_0^T H(t)\, e^{-ik\Omega t}\, dt.$$

Only $k \geq 0$ is stored. A Floquet state and its periodic modes u_m are

$$\psi(t) = e^{-i\varepsilon t} \sum_m e^{im\Omega t}\, u_m,$$

and the truncated Sambe (extended-space) eigenvalue problem is

$$\sum_{m'} \left(H_{m-m'} + m\Omega\, \delta_{mm'}\right) u_{m'} = \varepsilon\, u_m, \qquad -M \leq m, m' \leq M.$$

The Floquet cutoff M is `FloquetOptions.floquet_cutoff`. The pair $(\varepsilon + \ell\Omega,\ u_{m-\ell})$ describes the same physical state for every integer ℓ, and moves the harmonic centroid $\langle m\rangle = \sum_m m\|u_m\|^2$ to $\langle m\rangle + \ell$.

The mean energy

$$\bar{E} = \varepsilon - \Omega\langle m\rangle = \frac{1}{T}\int_0^T \langle\psi(t)|H(t)|\psi(t)\rangle\, dt$$

is invariant under this choice and is always reported as `mean_energies`.

Quasienergies are reported on one branch:

- `branch="label"`, the default for tracked states: let a be the static state assigned to a Floquet state. The branch is chosen so that the centroid of the projection onto a, $c_a = \sum_m m|\langle a|u_m\rangle|^2 / \sum_m |\langle a|u_m\rangle|^2$, lies in the interval from $-1/2$ to $1/2$. The quasienergy then tends to the static energy as the drive vanishes. It also varies continuously through a one-photon resonance and under strong diagonal driving.
- `branch="centroid"`, the default for `selection="all"`: the total centroid $\langle m\rangle$ lies in the interval from $-1/2$ to $1/2$.
- `folded` maps every quasienergy into the zone from $c - \Omega/2$ to $c + \Omega/2$, where c is `zone_center`.

The branch shift can move mode components beyond $|m| = M$. The stored harmonic window then grows accordingly, and the `harmonic_indices` field lists the retained m.

The **energy zero** is that of the model (*models* (page 5)). H_0 contains all c-number constants, so quasienergies are directly comparable with static energies. Driving ϵ or U also drives the constants of $H_d = (U/2)(n_d - 1)^2 + \epsilon(n_d - 1)$. A constant term in H_k with $k \neq 0$ multiplies $\psi(t)$ by a periodic global phase. Such terms are kept by default, so the Floquet problem uses $H(t)$ exactly as built; their values are recorded in `periodic.metadata["floquet"]["scalar_harmonics"]`. With `drop_scalar_harmonics=True` they are removed. This leaves the quasienergies, mean energies and static weights unchanged; only the harmonic weights of u_m change. For a decoupled dot driven by $\epsilon(t) = \epsilon_0 + A\cos\Omega t$, a state of charge n_d has harmonic weights $J_m((n_d - 1)A/\Omega)^2$ with the constants kept and $J_m(n_d A/\Omega)^2$ with them dropped. Keeping them measures the charge from half filling, which centres the sidebands and needs a smaller Floquet cutoff M.

10.2 Specifying a drive

`PeriodicHamiltonian.from_parameters` samples a model builder over one period:

```
import numpy as np
from qdjj_solver import ParameterDrive, PeriodicHamiltonian, Sector, cosh_grid,␣
↪reference_model
from qdjj_solver.qp_solver import solve_floquet

base = dict(bath=cosh_grid(pairs=2, bandwidth=5.), u=1.6, gamma=.3, phi=.8,␣
↪detuning=.2,
            symmetry=False, compress=False)
drives = [ParameterDrive.cosine("phi", .05),                        # phi0 +␣
↪0.05 cos(omega t)
          ParameterDrive.cosine("detuning", .02, phase=np.pi/2)]    # epsilon0␣
↪+ 0.02 cos(omega t + pi/2)
periodic = PeriodicHamiltonian.from_parameters(reference_model, base, drives,␣
↪omega=.57)
states = solve_floquet(periodic, cutoff=2, sector=Sector(0, 0),␣
↪options=dict(floquet_cutoff=4, states=2))
print(states.quasienergies, states.static_weights[:, :2])
```

Each `ParameterDrive` displaces one builder keyword, $p(t) = p_0 + \sum_n A_n \cos(n\Omega t + \theta_n)$, with one `Harmonic(order, amplitude, phase)` per term. Amplitudes carry the parameter's units, so phase amplitudes are in radians. `index=` drives one element of an array keyword, such as the `phases` of `make_model`. Several parameters may be driven together, with relative phases θ_n. `PeriodicHamiltonian.from_callable(function, omega)` samples an arbitrary `function(t)` that returns a `Hamiltonian`. The static states that label the dressed states are eigenstates of a reference Hamiltonian: the undriven model `builder(**base)` for `from_parameters`, and for `from_callable` the `reference=` argument or, without it, the time average H_0. A builder failure, such as $\Gamma(t) < 0$, is reported with the time at which it occurred.

Sampling. The model is built at `samples` equally spaced times (default 64). Each operator coefficient is Fourier transformed, which retains harmonics up to $k = \lceil N/2 \rceil - 1$ for N samples. With `verify=True` the builder is also evaluated at the midpoints between samples. There the ℓ_1 norm of the coefficient differences must not exceed `sampling_tolerance` (default 10^{-9}). This norm bounds the operator-norm error caused by aliasing, harmonic truncation and the removal of small coefficients. `periodic.metadata["floquet"]` records the norms `harmonic_l1`, `interpolation_error`, `hermiticity_error` and `cleanup_bound`. Model observables, for example `phase_derivative`, are sampled as well and stored as `periodic.observables[name][k]` for later Floquet matrix elements.

Fixed canonical coordinates. The QP basis is defined in the canonical (Bogoliubov) coordinates of the model. Every sample must have the same canonical coordinates as the reference, compared through `coordinate_fingerprint()`, a checksum of the declared electron-to-canonical map; otherwise construction raises `ValueError`. The implemented methods assume fixed coordinates; a moving basis would additionally require the connection term $-iU^\dagger \partial_t U$. The following drives are therefore rejected:

- the gap Δ or the bath grid, which define the Bogoliubov transformation;
- a phase drive with paired-mode compression, whose retained modes depend on ϕ;
- a detuning or ρW_N drive of `reference_model(symmetry=True)`, which switches the eta representation on and off;
- driven phases or direct hopping with `bath_reference="coupled"`.

Build Floquet models with `symmetry=False, compress=False`. The check compares coordinates rather than options, so a drive that leaves them fixed, such as U in eta coordinates, is accepted. A Hamiltonian without `metadata["coordinates"]` must be accepted explicitly with `assume_fixed_coordinates=True`; only its mode labels are then compared.

Each harmonic must also conserve the requested sector, because a projection silently drops terms that leave it. Drives of ϵ, ϕ, Γ, U and a longitudinal field conserve parity and S_z, while a transverse field requires `twice_sz=None`. A detuning drive breaks eta parity.

10.2.1 Phase drive and voltage

Phases enter the hoppings as $e^{i\phi_l/2} C_l^\dagger T_l d$, so a time-dependent phase describes a voltage drive. With $\phi = \phi_R - \phi_L$ and the bias voltage $V = V_L - V_R$, the electric potential of the left lead relative to the right one, the Josephson relation is $\dot{\phi} = 2eV/\hbar$ (*conventions* (page 85)). For a weak drive $\phi(t) = \phi_0 + A\cos(\Omega t + \theta)$, $H_{\pm 1} \approx (A/2) e^{\pm i\theta}\, \partial H/\partial\phi$: the drive couples through the `phase_derivative` operator; the current operator is $2e/\hbar$ times it. The exact harmonics (a Jacobi–Anger series) and the gauge equivalence with a level drive are derived in the physics manual, section "Phase drive, voltage and the Josephson relation" (`sec:phase-voltage`).

10.3 Methods

10.3.1 Sambe diagonalization

`solve_floquet(periodic, cutoff=None, sector=..., options=None, static=None)` diagonalizes the truncated Sambe matrix.

- `basis="fock"` uses the full QP Fock basis of the sector at the given cutoff, so the Sambe dimension is $(2M+1)d$, with d the dimension of the sector. Problems up to `dense_dimension` (default 2000) are diagonalized densely; larger ones use shift-invert.
  - Shift-invert factorizes $Q - \sigma$ by sparse LU at each target σ and runs ARPACK. The default targets are the static energies of the tracked states.
  - The vectors from all targets are merged by pivoted QR and a Rayleigh–Ritz step, and duplicate replicas are removed.
  - Each tracked state must be assigned a distinct Floquet state with at least `tracking_threshold` weight on it. Near a resonance, the eigenpairs closest to one target can be replicas of the state assigned at another target; the number of eigenpairs of a target is then doubled, up to `max_eigenpairs_per_target`.
  - LU fill-in grows quickly with D and M. When the estimated memory exceeds `max_memory_gib`, the solver raises `MemoryError` and suggests the static basis.
- `basis="eigen"` truncates to the `basis_states` lowest static eigenstates of the reference Hamiltonian, uses the matrix elements $\langle a|H_k|b\rangle$ between them and is diagonalized densely.
  - Choose the truncation to retain complete degenerate static multiplets; splitting a multiplet raises `ValueError`.
  - `fock_residuals` reports each state's residual in the full cutoff Fock space, providing a diagnostic of convergence in the number K of retained static states.
- `selection="tracked"` (default) assigns the lowest `states` static states to Floquet states by maximizing their static weights. A weight below `tracking_threshold` raises `RuntimeError`.
- `selection="all"` returns one replica of every Floquet state, ordered by quasienergy.

`residuals` combine two parts:

- the window residual $\|(Q - \varepsilon)u\|$, which measures eigensolver accuracy and is enforced against `residual_tolerance`;
- the leakage of H into the omitted harmonics $|m| > M$.

Both are also stored separately, as `metadata["window_residuals"]` and `metadata["leakage"]`. Because the untruncated Sambe operator is self-adjoint, an exact quasienergy in the chosen many-body space lies within the combined residual. The bound is first order; for an isolated quasienergy the actual error is usually much smaller.

A precomputed QP `Eigenstates` result can be passed as `static=` to reuse the static states in a scan. It is checked for coordinates, Fock basis and count.

10.3.2 One-period propagator

`floquet_propagator` has the same signature. Like QuTiP's `FloquetBasis`, it avoids the extended (Sambe) space.

- **Integration.** The Schrödinger equation with $H(t) = \sum_k H_k e^{ik\Omega t}$ is integrated over one period, with all stored harmonics and no Floquet cutoff. The scheme is the fourth-order commutator-free Magnus integrator, with `steps` steps per period. Exponentials come from eigendecompositions, so every step is exactly unitary.
- **Quasienergies.** Folded quasienergies are $-\arg\lambda/T$ for the eigenvalues λ of $U(T)$, obtained by complex Schur decomposition.
- **Modes.** Mean energies from time averages select the central replica. The modes are then Fourier transformed and truncated to $|m| \le M$; `boundary_weights` reports the weight outside this window.
- **Step check.** With `step_check=True` (default) the propagation is done with `steps` and with `2*steps` steps, and the finer result is reported. `metadata["step_errors"]` holds the Richardson estimate $|\Delta\varepsilon|/15$ of its error, which `step_tolerance` can enforce.

The representation is dense and limited to `dense_dimension`. A propagator in the Fock basis is an independent check of the Floquet cutoff M for small D. In the static basis it is practical for production calculations.

10.3.3 High-frequency expansion

`periodic.van_vleck(order)` returns the van Vleck effective Hamiltonian through order Ω^{-n}, where n is `order` (0, 1 or 2, default 2); the expansion is given in the physics manual, section "The high-frequency expansion" (`sec:van-vleck`).

The commutators are formed in the full fermionic algebra before any QP truncation. The result is a static `Hamiltonian` in the same coordinates, which `solve` can treat with either backend. Its eigenvalues approximate the quasienergies with error $O(\Omega^{-(n+1)})$ at order n. The first-order term vanishes when $H_{-k} = H_k$, as for a single-parameter cosine drive.

The expansion requires Ω to exceed the many-body spectral width, including the bath bandwidth. It provides a check for small models in that regime. For subgap resonances with Ω of the order of Δ, use Sambe diagonalization or the propagator. The number of operator terms grows with products of the harmonics' term counts. The copied observables are time averages without the $O(1/\Omega)$ kick-operator correction.

10.3.4 Rotating-wave model

`rwa_model(periodic, states, photons, ...)` restricts the Sambe matrix to the pairs $|a_i, n_i\rangle$ of static states and photon numbers:

$$Q^{\mathrm{RWA}}_{ij} = \left(H_{n_i - n_j}\right)_{a_i a_j} + n_i \Omega\, \delta_{ij}.$$

Choose the photon numbers so that the energies $E_{a_i} + n_i\Omega$ are nearly degenerate. For absorption from a to b with $E_b - E_a \approx \Omega$, use `states=[a, b]` and `photons=[0, -1]`. At resonance the splitting is $2|(H_{n_a - n_b})_{ab}|$, which for a weak phase drive is $A|\langle a|\partial_\phi H|b\rangle|$. The model neglects counter-rotating terms (Bloch–Siegert shift) and the AC Stark shifts from all other states, both of second order in the drive amplitude. `effective_matrix` holds Q^{RWA}.

At the particle–hole-symmetric point ($\epsilon = 0$, $\rho W_N = 0$), a phase drive conserves eta parity, restricting allowed couplings to states within each eta sector. For the parameters of `examples/floquet.py` the lowest two even states have opposite eta parity at $\epsilon = 0$, so the example uses $\epsilon = 0.2$.

10.4 Transitions and spectra

A weak probe $X(t) = \sum_k X_k\, e^{ik\Omega t}$ induces transitions between Floquet states at the frequencies $\pm(\varepsilon_\alpha - \varepsilon_\beta + q\Omega)$. By the golden rule, the line strengths are $|X^{(q)}_{\alpha\beta}|^2$, where $X^{(q)}_{\alpha\beta}$ are the harmonics of the matrix element between the two Floquet states. Positive line frequencies absorb a probe quantum; negative ones describe stimulated emission. For a complete set of final states the strengths obey a sum rule. The physics manual derives these results and discusses the Mollow and Autler–Townes patterns (`ch:probes`).

```
states = solve_floquet(periodic, cutoff=2, sector=Sector(0, 0),
                       options=dict(floquet_cutoff=6, selection="all", method=
↪"dense"))
table = states.matrix_elements("charge")                    # FloquetTransitions
lines = table.lines(initial=states.dressed_state(0))        # out of the␣
↪dressed ground state
spectrum = table.spectrum(omega_grid, width=0.002, kind="rate")
odd = solve_floquet(periodic, cutoff=2, sector=Sector(1, 1),␣
↪options=dict(floquet_cutoff=6, states=2))
tunnelling = odd.matrix_elements("d_up_dagger", states)     # across parity␣
↪sectors
```

- **Probes.** Probes are given by name (`periodic.probe(name)`), as a `FermionOperator` (treated as time independent), or as a dictionary of harmonics. Named observables use their sampled harmonics: the current under a phase drive depends on time, while charge, spin and electron operators remain time independent in fixed coordinates.
- **Compatibility.** The two results must belong to the same drive (equal checksums `PeriodicHamiltonian.fingerprint()`) and the same scalar gauge (`drop_scalar_harmonics`). Sectors, QP cutoffs, representations and methods may differ.
- **`FloquetTransitions`.** It holds `photons` (q), `elements` $X^{(q)}_{\alpha\beta}$, `frequencies`, `strengths`, `total` (the sum-rule value) and `completeness` (the fraction of `total` captured by the computed final states); `complete` marks a complete set of final states.
- **`lines(initial, weights)`.** It returns `SpectralLines` out of the chosen initial states, by default the dressed ground state. Degenerate multiplets are merged, which makes the strengths independent of the basis inside a multiplet. Expectation values (same state, $q = 0$) are dropped; elastic lines at $q\Omega$ are kept.
- **`spectrum(..., kind)`.** It broadens the lines with Lorentzian or Gaussian shapes (`width` is the half width at half maximum). The default `kind="signed"` gives $S(\omega) = \sum_l s_l\, L(\omega - \omega_l)$, with absorption at $\omega > 0$ and stimulated emission at $\omega < 0$; `kind="rate"` gives $S(\omega) + S(-\omega)$, the golden-rule rate at probe frequency $\omega > 0$; `kind="net"` gives absorption minus stimulated emission.

Steady-state populations of Floquet states, which determine the initial-state weights, require a separate model of the dissipative environment. Here the initial states and their weights are chosen explicitly. A probe of the pumped transition itself, started from one dressed state, shows the Mollow pattern of absorption, stimulated emission and elastic lines. The Autler–Townes doublet appears for an initial state outside the pumped pair.

10.5 Results

`FloquetStates` uses one layout for all methods; its fields are listed in the *result reference* (page 75).

`fock_modes()` expresses static-basis modes in the Fock basis, and `mode_at(t)` and `state_at(t)` return $u(t)$ and $\psi(t)$. Persistence through `save_result` currently supports static eigenstates. `PeriodicHamiltonian.record()` and `from_record()` give a JSON-compatible description of the drive.

10.6 Convergence and pitfalls

- **Separate approximations.** Check convergence in the Floquet cutoff M (leakage, boundary weights, propagator comparison), the static basis K (`fock_residuals`), the QP cutoff, the bath discretization, the number of samples (`interpolation_error`) and the propagator steps (`step_errors`). Assess continuum accuracy through bath refinement with the other approximations converged at each size.
- **Discretized continuum.** A finite bath replaces the continuum by discrete levels. Replicas of these levels cross the subgap quasienergies at multiphoton resonances and produce many narrow avoided crossings. Their positions and gaps depend on the discretization. In the continuum limit they represent the decay of the dressed subgap state; the closed finite-bath calculation represents their effect as a redistribution of spectral weight among discrete states. Follow dressed states through their `static_weights` and identify physical features by their stability under bath refinement.
- **Degeneracies.** The modes and labels of states inside a degenerate static multiplet depend on the basis. Compare whole multiplets. At a resonance both dressed states carry weights close to one half on the resonant static states.
- **Resonant chains.** At a resonance the replicas $|a, m\rangle$ and $|b, m-1\rangle$ form a chain through the whole Sambe space. A small Floquet cutoff cuts it and creates edge states that perturb the interior replicas at the order of the Rabi coupling. Use several harmonics even for weak drives ($M \gtrsim 5$). The solver raises an error when two tracked states are replicas of one Floquet state.
- **Branches.** Quasienergies that differ by multiples of Ω describe the same state. Branch choices near zone edges and resonances are conventions; `folded` and `mean_energies` are the robust quantities. The choice of a lowest quasienergy depends on the selected zone; use the label of the dressed static ground state to identify that reference state.

10.7 Validation

The tests of the Floquet methods, of probe matrix elements and of spectra, and the independent electron-space propagation, are listed in validation.

10.8 References

The Sambe (extended-space) formulation follows Shirley and Sambe; the high-frequency expansion follows Eckardt and Anisimovas, and Bukov, D'Alessio and Polkovnikov; the propagator uses the commutator-free Magnus integrators of Blanes and Moan, and of Alvermann and Fehske. Full citations are listed under *Floquet methods* (page 137).

CHAPTER

ELEVEN

PROBE MATRIX ELEMENTS AND SPECTRA

Calculating a spectrum requires the probe operator and initial-state populations, as well as the energies. Specify which operator acts, which sector contains the final states and which initial states are occupied. The physics manual develops the selection rules in "Subgap spectroscopy and Andreev qubits" (`ch:experiments`) and "Probing the driven junction" (`ch:probes`).

11.1 Start with a static matrix element

The *static workflow* (page 21) computes both the pair matrix element and electron-addition strength. Rows are final bra states and columns are initial ket states. A charge/phase probe preserves parity; an electron operator changes the probed junction's parity. The complete system, including the electron source, still conserves parity.

`matrix_elements` returns amplitudes. Combine coherent terms with their complex phases intact, then square the resulting magnitudes to obtain strengths. Energy differences determine possible line positions, while matrix elements and populations determine visibility.

11.2 A small independent Floquet spectrum

The following complete example drives a proximitized dot. It is deliberately small so that all final states in its even sector can be retained.

```
import numpy as np
from qdjj_solver import Hamiltonian, PeriodicHamiltonian, Sector
from qdjj_solver import create, hermitian_pair, number
from qdjj_solver.qp_solver import solve_floquet

def paired_dot(detuning=.2):
    n = number(0) + number(1)
    op = detuning*(n-1) - .3*hermitian_pair(create(0)*create(1))
    return Hamiltonian(op, 2, (1, -1), observables={"impurity_charge": n})

periodic = PeriodicHamiltonian.from_callable(
    lambda t: paired_dot(.2 + .02*np.cos(.72*t)), omega=.72,
    reference=paired_dot(), assume_fixed_coordinates=True)
states = solve_floquet(periodic, cutoff=0, sector=Sector(0, 0),
                       options=dict(floquet_cutoff=4, selection="all", method=
↪"dense"))
table = states.matrix_elements("charge")
```

(continues on next page)

(continued from previous page)

```
initial = states.dressed_state(0)
lines = table.lines(initial=initial)
grid = np.linspace(-1.5, 1.5, 1001)
spectrum = table.spectrum(grid, initial=initial, width=.01, kind="signed")
print(lines.frequencies, lines.strengths)
assert np.all(np.isfinite(spectrum))
```

The explicit assumption of fixed coordinates is appropriate here because the specified modes are fixed physical dot modes at every detuning. Use this option after verifying that the canonical coordinates remain fixed throughout the drive.

11.3 Inspect the line list before broadening

Floquet transition frequencies include integer multiples of the drive frequency. `FloquetTransitions.photons` holds those harmonic indices; `elements`, `frequencies` and `strengths` share the documented bra/ket axes. `lines` merges degenerate multiplets, making the combined strength invariant under changes of basis within a retained degenerate subspace. It omits the same-state, zero-photon expectation contribution while retaining elastic harmonics.

The named probes and their physical meanings are listed in the *observable/probe reference* (page 80). Under a parameter drive, a named observable uses its sampled time-dependent harmonics. Replacing a driven current operator by its static value changes the probe problem.

11.4 Choose populations and line-shape conventions

- `kind="signed"` displays absorption at positive frequency and stimulated emission at negative frequency.
- `kind="rate"` combines the positive- and negative-frequency contributions at a positive probe frequency.
- `kind="net"` subtracts stimulated emission from absorption.
- `width` is the half width at half maximum of the selected line shape.

Supply the initial-state populations explicitly through their weights. Predicting steady-state populations requires an additional model of relaxation. Broadening specifies the plotted line shape; physical line widths require a dynamical model of the relevant decay and dephasing processes. A visible line requires both a nonzero matrix element and population in its initial state.

11.5 Check captured strength and convergence

Inspect `completeness`, the fraction of the sum-rule strength included in the calculation. Increase the number of final states and the harmonic window, and verify that the spectral lines are stable. A complete finite basis still needs QP and bath convergence. For transitions between different sectors, use results from the same periodic Hamiltonian and scalar gauge.

For a realistic junction, first check the static spectrum and weak-drive limit using a small retained state space. Then use Tutorial 08, photon-assisted tunnelling and spin resonance for worked extensions. The *Floquet guide* (page 46) describes compatibility rules and the *result reference* (page 74) gives complete array layouts.

CHAPTER

TWELVE

TROUBLESHOOTING

The `qdjj-solver solve` command returns exit status **0** after saving a result and **2** for recognized errors in input, required libraries, numerical calculations or file access; invalid command arguments also return 2. With DMRG's default `require_convergence=False`, exit 0 can accompany `Finite-problem checks passed: False`; set `require_convergence=true` in JSON when a calculation must meet those checks. For continuum calculations, also assess bath convergence through refinement (numerics).

To report a problem, provide a small input file that reproduces it, the error message, the software version and any version information saved with the result on the issue page.

Messages are abridged; the source is `docs/troubleshooting.toml`, and every quoted message is checked against the error messages in the code.

12.1 Input files, models and static calculations

DMRG/TeNPy or HDF5 dependency missing
: **What to check:** From the source directory, run `python -m pip install '.[dmrg]'` with the interpreter used for the calculation.

The configuration or one of its sections (model, bath, sector, solver) must be an object
: **What to check:** Use `{...}` for these objects, JSON booleans for yes/no settings and integers for sector values. Check the reported field path for typos.

Hamiltonian does not conserve S_z
: **What to check:** For spin-mixing terms or spin-flip drives choose `Sector(parity, twice_sz=None)`, or JSON `"twice_sz": null`.

Fermion parity and twice_sz are incompatible
: **What to check:** Even parity requires an even integer `twice_sz`; odd parity an odd one.

Empty symmetry sector, or more eigenpairs requested than states
: **What to check:** Check the quantum numbers, the QP cutoff, the included modes and the number of requested eigenstates.

Invalid QP cutoff
: **What to check:** Use `0 <= cutoff <= h.bath_modes` after model compression, or `None` (JSON `null`) for the full space. Set `cutoff` explicitly to 0 or `null` for impurity-only QP command-line inputs.

Sector dimension exceeds max_dimension
: **What to check:** Estimate the basis size (`qpsolver dimension`) before raising `max_dimension`; consider a converged QP cutoff or DMRG.

An array needs more memory than max_memory_gib

What to check: The message names the array and a method or option that avoids it. Raise `max_memory_gib` only if the machine has that much free memory.

Sparse matrix exceeds max_nnz

What to check: Dense, sparse and Schur methods build this matrix. Check both `max_nnz` and the memory-derived cap from `max_memory_gib`, or use `method="matrix-free"`, whose basis and Krylov-space limits still apply.

The iterative eigensolver missed eigenvalues below the diagonal bound

What to check: ARPACK missed some of the lowest eigenvalues, even after a second spectral shift. Use `method="dense"` if the dimension allows it, or change `seed` or `ncv`, and report the case.

DMRG requires cutoff=null

What to check: Omit the JSON cutoff or set it to `null`; the MPS bond dimension `chi_max` controls the approximation.

Finite-problem convergence failed

What to check: Inspect residuals, orthonormality, energy ordering and sweep histories; increase the bond dimension, the number of sweeps or the number of initial-state trials as appropriate.

Initial or compared states have incompatible coordinates or layout

What to check: Reusing MPS states requires the same canonical basis, mode ordering and site grouping; start a new calculation if these change.

No supported state artifact, or checkpoint missing

What to check: Save with `save_states=True` or `--save-states`, and keep both the JSON and its named wavefunction file.

Checksum or scalar-record mismatch

What to check: Recover the matching JSON and wavefunction files from the same save. Editing the JSON invalidates the checksum that links it to the wavefunction file.

12.2 Floquet calculations, matrix elements and spectra

The canonical coordinates change during the drive period

What to check: Keep the gap and bath grid fixed. Build the model with `symmetry=False, compress=False`, and with `bath_reference="isolated"` when phases or direct hopping are driven.

Fourier interpolation error … increase samples

What to check: Increase `samples` to resolve the relevant drive harmonics.

Too few Floquet candidates for the tracked states

What to check: The Floquet cutoff is too small for the spread of the sidebands; increase `floquet_cutoff` (or `eigenpairs_per_target` for shift-invert).

Two tracked states are replicas of one Floquet state

What to check: A small Floquet cutoff cuts a resonant chain of replicas; increase `floquet_cutoff` (use $M \gtrsim 5$ even for weak drives).

Floquet weight below tracking_threshold

What to check: A tracked state is strongly hybridized; increase `floquet_cutoff`, `eigenpairs_per_target` or `basis_states`, lower `tracking_threshold`, or use `selection="all"`.

basis_states splits a degenerate static multiplet
What to check: Choose the static basis size K at a gap of the static spectrum.

Sambe dimension exceeds max_dimension
What to check: Use `basis="eigen"`, a smaller static basis or fewer harmonics.

selection='all' with shift-invert
What to check: Use `method="dense"` (with `basis="eigen"` for large models) or give explicit `targets`.

Floquet states of different drives or scalar gauges
What to check: Build both results from one `PeriodicHamiltonian` with the same `drop_scalar_harmonics`.

No harmonic of the probe connects the sectors
What to check: Choose a destination sector compatible with the change in parity and spin induced by the probe.

Probe observable was not sampled, or the model lacks it
What to check: Build the `PeriodicHamiltonian` with `observables=True`, from a model that declares the observable.

The computed states split a degenerate static multiplet
What to check: Increase `states` or `basis_states`, or pass `merge_degenerate=False` to `lines`.

Physical operators require an uncompressed electron map
What to check: Electron probes (`d_up`, `spin_x`, …) and `physical_operator` need a model built with `compress=False`. A model assembled directly as a `Hamiltonian` declares no electron map: pass an operator built with `create` and `annihilate` instead of a probe name.

The probe images need more memory than max_memory_gib
What to check: Restrict `ket_states` or raise `max_memory_gib`.

APPENDIX

A

OPTION REFERENCE

Option objects are immutable dataclasses with fields fixed at construction. Unified `qdjj_solver.solve` and the Floquet entry points also accept dictionaries; direct static backend solvers require their own option objects. Energies and tolerances are in the units of the Hamiltonian. The descriptions are the field docstrings in the source; *Using the solver and saving results* (page 13) explains how to choose the settings.

A.1 QP solver

`qdjj_solver.qp_solver.SolverOptions` (source: `src/qdjj_solver/qp_solver/solver.py`)

method

Type: `str`

Default: `"auto"`

Meaning: Eigensolver: `"auto"`, `"dense"`, `"sparse"`, `"matrix-free"`, `"schur"` or `"block"`. `"auto"` diagonalizes densely up to dimension 128, uses sparse ARPACK up to 400,000 and matrix-free ARPACK above. `"block"` uses LOBPCG for several eigenstates, including degenerate multiplets.

eigenpairs

Type: `int`

Default: `1`

Meaning: Number of lowest eigenstates computed in the selected sector.

tolerance

Type: `float`

Default: `1e-11`

Meaning: Relative ARPACK eigenvalue tolerance, for the internally shifted operator H - sigma with sigma below the smallest diagonal element by an offset of order one; for `"schur"` the root-search tolerance in energy units. For `"block"`, the residual tolerance in energy units, capped at `residual_tolerance/10`. Unused by dense diagonalization.

residual_tolerance

Type: `float`

Default: `2e-9`

Meaning: Largest accepted projected residual norm, in energy units.

maxiter
Type: `int`

Default: `10000`

Meaning: Maximum number of ARPACK or block-LOBPCG iterations.

ncv
Type: `int`

Default: `40`

Meaning: Requested Krylov-space dimension; increased as needed for the number of eigenstates and limited by the sector dimension.

max_dimension
Type: `int`

Default: `5_000_000`

Meaning: Largest allowed number of Fock basis states, checked before the basis is enumerated.

max_nnz
Type: `int`

Default: `100_000_000`

Meaning: Largest number of stored sparse-matrix entries (dense, sparse and Schur methods); the remaining `max_memory_gib` budget can lower it.

max_memory_gib
Type: `float`

Default: `24.`

Meaning: Limit on the estimated basis and matrix storage, in GiB; the total memory use can be larger.

threads
Type: `int`

Default: `1`

Meaning: Maximum number of threads used by the numerical libraries.

seed
Type: `int`

Default: `1729`

Meaning: Seed of the random starting vector of the iterative eigensolvers.

A.2 DMRG solver

`qdjj_solver.dmrg_solver.SolverOptions` (source: `src/qdjj_solver/dmrg_solver/solver.py`)

eigenpairs
Type: `int`

Default: `1`

Meaning: Number of eigenstates to calculate, optimized sequentially.

chi_max

Type: `int`

Default: `128`

Meaning: Maximum MPS bond dimension.

max_sweeps

Type: `int`

Default: `60`

Meaning: Maximum number of sweeps per initial-state trial and eigenstate.

min_sweeps

Type: `int`

Default: `10`

Meaning: Minimum number of sweeps per initial-state trial and eigenstate.

energy_tolerance

Type: `float`

Default: `1e-11`

Meaning: Absolute energy tolerance, in energy units: TeNPy's sweep-to-sweep energy criterion and the stability test of the last sweep energies; the local Lanczos tolerance `E_tol` is derived from it.

entropy_tolerance

Type: `float`

Default: `1e-8`

Meaning: Dimensionless entropy-change threshold of TeNPy's stopping test.

residual_tolerance

Type: `float`

Default: `1e-7`

Meaning: Maximum accepted physical residual norm for convergence of the finite problem, in energy units.

orthogonality_tolerance

Type: `float`

Default: `1e-8`

Meaning: Tolerance on deviations of the overlap matrix from the identity and on overlaps with earlier eigenstates.

svd_min

Type: `float`

Default: `1e-14`

Meaning: Smallest retained singular value during optimization; may be zero.

lanczos_maxiter
Type: `int`

Default: `100`

Meaning: Maximum number of local Lanczos iterations.

seed_trials
Type: `int`

Default: `4`

Meaning: Number of initial-state trials per eigenstate, including randomized states.

seed
Type: `int`

Default: `1729`

Meaning: Seed of the local random transformations of initial states.

seed_randomization_steps
Type: `int`

Default: `4`

Meaning: Layers of local random unitary transformations for additional initial states; zero disables randomization.

excitation_operator
Type: `str | None`

Default: `None`

Meaning: Name of a model observable that conserves the sector's quantum numbers; its fluctuation applied to the previous eigenstate supplies the first trial state for the next eigenstate.

mixer
Type: `bool`

Default: `True`

Meaning: Enable TeNPy's optimization mixer.

mixer_amplitude
Type: `float`

Default: `1e-5`

Meaning: Initial mixer strength.

mixer_sweeps
Type: `int`

Default: `6`

Meaning: Number of sweeps after which the mixer is switched off.

group_size
Type: `int`

Default: `2`

Meaning: One or two canonical fermionic modes per MPS site.

mode_order
Type: `tuple[int, ...] | None`

Default: `None`

Meaning: Permutation of all zero-based canonical mode indices; `None` keeps the model order.

observables
Type: `tuple[str, ...] | None`

Default: `None`

Meaning: Observables to evaluate: `None` selects all named observables, a tuple selects specified names, and `()` selects none.

calculate_residuals
Type: `bool`

Default: `True`

Meaning: Evaluate the residual norm of (H - E) psi for each state after optimization, without MPS compression.

require_convergence
Type: `bool`

Default: `False`

Meaning: Raise an error when the finite-problem convergence checks fail.

threads
Type: `int`

Default: `1`

Meaning: Maximum number of threads used by the numerical libraries.

lanczos_probability_tolerance
Type: `float`

Default: `1e-14`

Meaning: Local Lanczos probability-error threshold, passed to TeNPy as `P_tol`.

A.3 Floquet calculations

`qdjj_solver.qp_solver.FloquetOptions` (source: `src/qdjj_solver/qp_solver/floquet.py`)

floquet_cutoff
Type: `int`

Default: `4`

Meaning: Floquet cutoff M: the Sambe matrix retains the harmonic blocks m = -M, …, M.

basis
Type: `str`

Default: `"fock"`

Meaning: `"fock"` uses the full QP Fock space of the sector, `"eigen"` the `basis_states` lowest static eigenstates.

states
Type: `int`

Default: `4`

Meaning: Number of lowest static states tracked through the drive; they label the dressed states.

basis_states
Type: `int`

Default: `32`

Meaning: Size K of the static eigenbasis for `basis="eigen"`; choose K at a gap of the static spectrum, retaining complete degenerate multiplets.

selection
Type: `str`

Default: `"tracked"`

Meaning: `"tracked"` returns the continuations of the tracked static states, `"all"` every distinct Floquet state of the truncated Sambe matrix (dense solver or explicit `targets`).

branch
Type: `str`

Default: `"auto"`

Meaning: Replica that defines the reported quasienergy: `"label"` continues the assigned static state, `"centroid"` is the replica whose harmonic centroid lies in [-1/2, 1/2); `"auto"` uses `"label"` for `selection="tracked"` and `"centroid"` for `selection="all"`.

method
Type: `str`

Default: `"auto"`

Meaning: Sambe eigensolver: `"dense"`, `"shift-invert"` or `"auto"` (dense up to `dense_dimension`).

dense_dimension
Type: `int`

Default: `2000`

Meaning: Largest dimension diagonalized densely: the Sambe matrix for `method="auto"`, the static solution and the propagator.

targets
Type: `tuple[float, ...] | None`

Default: `None`

Meaning: Explicit shift-invert target quasienergies; by default the static energies of the tracked states.

eigenpairs_per_target
Type: `int`

Default: `8`

Meaning: Initial number of shift-invert eigenpairs per target; doubled for a target until its tracked state is assigned a distinct Floquet state with at least `tracking_threshold` weight.

max_eigenpairs_per_target
Type: `int`

Default: `64`

Meaning: Upper limit on `eigenpairs_per_target` during successive doubling.

target_shift
Type: `float`

Default: `1e-7`

Meaning: Offset added to the default shift-invert targets to separate them from the reference eigenvalues.

tracking_threshold
Type: `float`

Default: `.25`

Meaning: Minimum time-averaged weight of a tracked static state in its assigned Floquet state.

steps
Type: `int`

Default: `256`

Meaning: Number of time steps per period of the one-period propagator; doubled for the reported result when `step_check=True`.

step_check
Type: `bool`

Default: `True`

Meaning: Repeat the propagation with twice as many steps and estimate the time-step error from the change of the quasienergies; the finer result is reported.

step_tolerance
Type: `float | None`

Default: `None`

Meaning: If set, raise an error when the estimated propagator step error exceeds it, in energy units; requires `step_check=True`.

zone_center
Type: `float`

Default: `0.`

Meaning: Centre c of the zone [c - omega/2, c + omega/2) of the `folded` quasienergies.

drop_scalar_harmonics
Type: `bool`

Default: `False`

Meaning: Remove the c-number parts of the harmonics H_k with k != 0; the quasienergies are unchanged, but the harmonic weights of the modes are redistributed.

tolerance
Type: `float`

Default: `1e-11`

Meaning: Eigensolver tolerance of the static solution and of the shift-invert iterations.

residual_tolerance
Type: `float`

Default: `2e-9`

Meaning: Largest accepted window residual of the Sambe and static solutions, in energy units.

maxiter
Type: `int`

Default: `10000`

Meaning: Maximum number of iterations of the static and shift-invert eigensolvers.

max_dimension
Type: `int`

Default: `5_000_000`

Meaning: Largest allowed Fock-basis and Sambe dimension.

max_nnz
Type: `int`

Default: `100_000_000`

Meaning: Largest number of stored sparse-matrix entries.

max_memory_gib
Type: `float`

Default: `24.`

Meaning: Limit on the estimated storage of dense Sambe matrices, sparse LU factors, propagators and probe images, in GiB.

threads
Type: `int`

Default: `1`

Meaning: Maximum number of threads used by the numerical libraries.

seed
Type: `int`

Default: `1729`

Meaning: Seed of the random starting vectors of the iterative eigensolvers.

APPENDIX

B

MODEL AND DRIVE CONSTRUCTION

Parameter names and defaults are taken from implementation signatures; explanations are in `docs/parameters.toml`. Dataclass fields use their source attribute docstrings. Required arguments are marked explicitly. The *model guide* (page 5) explains physical conventions.

B.1 Reference junction

`qdjj_solver.reference_model`

B.1.1 `bath`

Default: `required`. Required DiscreteBath, reused for matched leads. Its weights represent the normal-state measure.

B.1.2 `u`

Default: `2.0`. Finite charging energy U of the single orbital.

B.1.3 `gamma`

Default: `0.4`. Nonnegative finite total hybridization, summed over the leads; energy units.

B.1.4 `phi`

Default: `1.8849555921538759`. Finite phase difference phi_R-phi_L in radians; junction phases are [-phi/2, +phi/2].

B.1.5 `rho_ws`

Default: `0.0`. Finite dimensionless rho W_S, the spin-dependent direct contact; zero for a single lead.

B.1.6 `rho_wn`

Default: `0.0`. Finite dimensionless rho W_N, the normal direct contact; zero for a single lead.

B.1.7 `geometry`

Default: `'junction'`. ‘junction’ for two leads or ‘single’ for one.

B.1.8 `detuning`

Default: `0.0`. Finite epsilon_d+U/2 in energy units; zero is particle-hole symmetry.

B.1.9 `field`

Default: `0.0`. Finite Zeeman energy acting only on the dot and multiplying S_z; positive values raise the spin-up level.

B.1.10 `symmetry`

Default: `True`. Use eta coordinates when the gate, contacts and bath satisfy the required symmetries.

B.1.11 `compress`

Default: `True`. Freeze eligible dark modes in their vacuum. Use False for a full finite-bath spectrum and arbitrary electron operators. The unified CLI defaults this to false.

B.1.12 `bath_reference`

Default: `'isolated'`. 'isolated' BCS reservoirs or a 'coupled' quadratic environment; the latter requires a gapped, unpolarized, spin-conserving reservoir vacuum.

B.1.13 `coefficient_tolerance`

Default: `1e-14`. Nonnegative finite absolute threshold for numerical coefficient truncation, in energy units.

B.2 General impurity model

`qdjj_solver.make_model`

B.2.1 `impurity`

Default: `required`. Impurity defining physical electron operators on M spinful orbitals.

B.2.2 `baths`

Default: `required`. Nonempty sequence of L DiscreteBath reservoirs.

B.2.3 `tunneling`

Default: `required`. L finite complex matrices of shape (2, 2*M), in energy units; reservoir-spin rows and impurity-spin columns.

B.2.4 `phases`

Default: `None`. L finite phases in radians; None means zero phases.

B.2.5 `direct`

Default: `None`. Mapping (l,m) to a finite complex 2-by-2 contact matrix, l<m; None means no direct contacts.

B.2.6 `phase_velocities`

Default: `None`. L finite derivatives of lead phase with respect to the chosen phase coordinate. None means zero; nonzero values generate phase_derivative.

B.2.7 `eta_basis`

Default: `False`. Request verified eta coordinates; restricted to a matched symmetric single-dot junction.

B.2.8 `compress_pairs`

Default: `False`. Freeze discarded dark modes; requires eligible eta coordinates and no finite direct W_S coupling.

B.2.9 `bath_reference`

Default: `'isolated'`. 'isolated' or 'coupled'. Coupled coordinates require spin-conserving direct hopping and a nonzero reservoir excitation gap.

B.2.10 `coefficient_tolerance`

Default: `1e-14`. Nonnegative finite threshold for numerical coefficient truncation, in energy units.

B.2.11 `direct_derivatives`

Default: `None`. Named contact-operator mappings in the same form as direct; each name must be distinct from impurity observables and phase_derivative.

B.3 Cosh quadrature

`qdjj_solver.cosh_grid`

B.3.1 `pairs`

Default: `required`. Positive integer number of mirrored pairs of signed normal levels per reservoir; each level has two spin modes.

B.3.2 `delta`

Default: `1.0`. Positive finite superconducting gap in energy units.

B.3.3 `bandwidth`

Default: `100.0`. Positive finite normal-state half-bandwidth in the same energy units. Coarse weights retain their quadrature values.

B.4 Fitted surrogate reservoir

`qdjj_solver.fit_surrogate`

B.4.1 `levels`

Default: `required`. Positive integer number of signed spinful normal levels per reservoir. An odd number of levels includes a normal-state node at zero energy.

B.4.2 `delta`

Default: `1.0`. Positive finite superconducting gap in energy units.

B.4.3 `bandwidth`

Default: `100.0`. Positive finite normal-state half-bandwidth in energy units.

B.4.4 `frequency_cutoff`

Default: `100.0`. Upper positive imaginary-frequency fit boundary; same energy units as the gap.

B.4.5 `frequency_min`

Default: `0.001`. Positive lower fit boundary, smaller than frequency_cutoff.

B.4.6 `frequency_points`

Default: `1000`. Integer number of logarithmically spaced fit points, at least ten.

B.4.7 `starts`

Default: `4`. Positive number of optimization attempts with different initial guesses; the successful fit with the smallest objective value is retained.

B.4.8 `seed`

Default: `1729`. Random-generator seed for reproducible initial guesses.

B.4.9 `relative_weight`

Default: `0.0`. Finite real exponent controlling frequency weighting of the residual; zero uses the absolute objective.

B.4.10 `max_nfev`

Default: `5000`. Positive maximum number of function evaluations per optimization attempt. Assess the resulting bath accuracy through fit diagnostics and observable convergence.

B.5 Padé chain-expansion reservoir

`qdjj_solver.chain_expansion`

B.5.1 `levels`

Default: `required`. Positive number of spinful sites in one reservoir, returned in mirrored star coordinates.

B.5.2 `delta`

Default: `1.0`. Positive finite superconducting gap in energy units.

B.5.3 `bandwidth`

Default: `100.0`. Positive finite half-bandwidth; in wide-band mode it only sets density normalization.

B.5.4 `wide_band`

Default: `False`. Boolean selecting an infinite-band target; False retains the specified finite flat band.

B.6 Anderson impurity

`qdjj_solver.Impurity.anderson`

B.6.1 `u`

Default: `2.0`. Finite coefficient U in U(n-1)^2/2, in energy units.

B.6.2 `detuning`

Default: `0.0`. Finite epsilon_d+U/2, multiplying n-1.

B.6.3 `field`

Default: `0.0`. Finite Zeeman energy multiplying S_z, acting on the impurity only.

B.7 Impurity from integrals

`qdjj_solver.Impurity.from_integrals`

B.7.1 `one_body`

Default: `required`. Finite even-dimensional square spin-orbital matrix h_ab in sum h_ab d_a† d_b.

B.7.2 `interaction`

Default: `None`. Optional finite tensor U_abcd with all four dimensions equal to one_body's dimension; enters as one quarter times U_abcd d_a† d_b† d_d d_c.

B.7.3 `constant`

Default: `0.0`. Finite additive scalar energy retained by the Hamiltonian.

B.8 Single cosine drive

`qdjj_solver.ParameterDrive.cosine`

B.8.1 `parameter`

Default: `required`. Builder keyword name; its undriven value must be present in base.

B.8.2 `amplitude`

Default: `required`. Finite real amplitude in that parameter's units (radians for phases).

B.8.3 `phase`

Default: `0.0`. Finite real cosine phase offset, in radians.

B.8.4 `order`

Default: `1`. Positive integer harmonic order relative to omega.

B.8.5 `index`

Default: `None`. Optional nonnegative array-element index; None selects a scalar parameter.

B.9 Periodic model from driven parameters

`qdjj_solver.PeriodicHamiltonian.from_parameters`

B.9.1 `builder`

Default: `required`. Callable returning a Hamiltonian from the keyword arguments in base.

B.9.2 `base`

Default: `required`. Undriven keyword mapping; it must explicitly include every driven parameter.

B.9.3 `drives`

Default: `required`. One ParameterDrive or a sequence, with each parameter/index pair driven only once.

B.9.4 `omega`

Default: `required`. Positive finite drive angular frequency, in energy units with hbar=1.

B.9.5 `samples`

Default: `64`. Integer number of equally spaced samples per period, at least three; refine to check Fourier resolution.

B.9.6 `coefficient_tolerance`

Default: `1e-14`. Nonnegative coefficient cleanup scale applied to harmonics.

B.9.7 `sampling_tolerance`

Default: `1e-09`. Acceptance scale for the midpoint interpolation-error bound, in energy units.

B.9.8 `verify`

Default: `True`. Verify the Fourier reconstruction at sample midpoints; failures raise an error.

B.9.9 `observables`

Default: `True`. Also sample model observables and store their time-dependent harmonics.

B.10 Periodic model from a time-dependent function

`qdjj_solver.PeriodicHamiltonian.from_callable`

B.10.1 `function`

Default: `required`. Callable function(t) returning a Hamiltonian at time t.

B.10.2 `omega`

Default: `required`. Positive finite angular frequency, defining the period 2*pi/omega.

B.10.3 `samples`

Default: `64`. Integer number of samples over the period, at least three.

B.10.4 `reference`

Default: `None`. Static Hamiltonian whose eigenstates define the tracked-state labels; None uses the time average.

B.10.5 `assume_fixed_coordinates`

Default: `False`. Explicitly assume fixed coordinates for a model without coordinate metadata. Valid use requires a time-independent canonical basis.

B.10.6 `coefficient_tolerance`

Default: `1e-14`. Nonnegative coefficient cleanup scale applied to harmonics.

B.10.7 `sampling_tolerance`

Default: `1e-09`. Midpoint interpolation-error acceptance scale, in energy units.

B.10.8 `verify`

Default: `True`. Check the reconstructed Fourier series at sample midpoints.

B.10.9 `observables`

Default: `True`. Sample the function's named observables as well as the Hamiltonian.

B.11 High-frequency effective Hamiltonian

`qdjj_solver.PeriodicHamiltonian.van_vleck`

B.11.1 `order`

Default: `2`. Perturbative order 0, 1 or 2 in inverse omega; requires high frequency relative to the many-body width.

B.11.2 `tolerance`

Default: `1e-14`. Absolute coefficient cleanup scale after full-operator commutators.

B.11.3 `max_harmonic`

Default: `None`. Largest nonnegative Hamiltonian harmonic used; None uses every stored harmonic.

B.12 Discrete reservoir fields

`qdjj_solver.DiscreteBath`; source: `src/qdjj_solver/common/baths.py`.

B.12.1 `xi`

Type: `np.ndarray`. Default: `required`.

Nonempty one-dimensional finite signed normal energies, in Hamiltonian energy units.

B.12.2 `weights`

Type: `np.ndarray`. Default: `required`.

Strictly positive normal-state measure weights, same shape as `xi`; their supplied values are preserved.

B.12.3 `delta`

Type: `float`. Default: `1.`.

Positive finite superconducting gap in energy units.

B.12.4 `bandwidth`

Type: `float`. Default: `100.`.

Positive finite normal-state half-bandwidth; sets the density per spin to 1/(2D).

B.12.5 `metadata`

Type: `dict`. Default: `dict()`.

Construction parameters, fit diagnostics and optional user provenance.

B.13 Impurity fields

`qdjj_solver.Impurity`; source: `src/qdjj_solver/common/models.py`.

B.13.1 `orbitals`

Type: `int`. Default: `required`.

Positive number of physical impurity orbitals; each contributes up/down spin modes.

B.13.2 `operator`

Type: `FermionOperator`. Default: `required`.

Hermitian, parity-conserving impurity operator in physical electron coordinates.

B.13.3 `observables`

Type: `dict[str, FermionOperator]`. Default: `dict()`.

Additional impurity operators; charge, spin-z and orbital double occupancies are added automatically.

B.13.4 `metadata`

Type: `dict`. Default: `dict()`.

Physical construction parameters and optional user information.

B.14 Hamiltonian fields

`qdjj_solver.Hamiltonian`; source: `src/qdjj_solver/common/problem.py`.

B.14.1 `operator`

Type: `FermionOperator`. Default: `required`.

Finite Hermitian, parity-conserving polynomial in the declared canonical modes.

B.14.2 `nimp`

Type: `int`. Default: `required`.

Number of impurity spin modes, occupying the first mode indices; zero is allowed.

B.14.3 `spins`

Type: `tuple[int, ...]`. Default: `required`.

Twice-spin labels +1 or -1 for every canonical mode, in basis order.

B.14.4 `eta_labels`

Type: `tuple[int, ...] | None`. Default: `None`.

Optional +1/-1 eta label per mode; required when imposing eta parity.

B.14.5 `observables`

Type: `dict[str, FermionOperator]`. Default: `dict()`.

Named operators in the same canonical coordinates as the Hamiltonian.

B.14.6 `metadata`

Type: `dict`. Default: `dict()`.

Construction, energy-reference and coordinate information; may include user entries.

B.15 Symmetry sector fields

`qdjj_solver.Sector`; source: `src/qdjj_solver/common/problem.py`.

B.15.1 `parity`

Type: `int`. Default: `1`.

Total fermion parity: zero is even and one is odd; it includes impurity and bath.

B.15.2 `twice_sz`

Type: `int | None`. Default: `1`.

Twice total spin projection, an integer compatible with parity; `None` leaves it unrestricted.

B.15.3 `eta`

Type: `int | None`. Default: `None`.

Optional eta parity, +1 or -1; requires declared eta labels and a conserving Hamiltonian.

B.15.4 `particle_number`

Type: `int | None`. Default: `None`.

Optional nonnegative total occupation of the chosen canonical modes, compatible with parity and spin.

B.16 Cosine component fields

`qdjj_solver.Harmonic`; source: `src/qdjj_solver/common/floquet.py`.

B.16.1 `order`

Type: `int`. Default: `required`.

Positive integer multiple of the fundamental drive frequency.

B.16.2 `amplitude`

Type: `float`. Default: `required`.

Finite real cosine amplitude in the driven parameter's units.

B.16.3 `phase`

Type: `float`. Default: `0.`.

Finite cosine phase offset in radians.

B.17 Parameter modulation fields

`qdjj_solver.ParameterDrive`; source: `src/qdjj_solver/common/floquet.py`.

B.17.1 `parameter`

Type: `str`. Default: `required`.

Nonempty keyword name in the model builder's undriven parameter mapping.

B.17.2 `harmonics`

Type: `tuple[Harmonic, ...]`. Default: `required`.

Nonempty sequence of cosine components with distinct positive orders.

B.17.3 `index`

Type: `int | None`. Default: `None`.

Optional nonnegative index of an array parameter; `None` selects a scalar parameter.

B.18 Periodic Hamiltonian fields

`qdjj_solver.PeriodicHamiltonian`; source: `src/qdjj_solver/common/floquet.py`.

B.18.1 `omega`

Type: `float`. Default: `required`.

Positive finite fundamental angular frequency in energy units with hbar=1.

B.18.2 `harmonics`

Type: `dict[int, FermionOperator]`. Default: `required`.

Nonnegative Fourier indices mapped to operators; negative harmonics are Hermitian conjugates.

B.18.3 `nimp`

Type: `int`. Default: `required`.

Number of impurity spin modes, identical for every harmonic and the reference.

B.18.4 `spins`

Type: `tuple[int, ...]`. Default: `required`.

Twice-spin label +1 or -1 per fixed canonical mode.

B.18.5 `eta_labels`

Type: `tuple[int, ...] | None`. Default: `None`.

Optional eta label per mode; fixed across all harmonics.

B.18.6 `reference`

Type: `Hamiltonian | None`. Default: `None`.

Static Hamiltonian defining tracked state labels; `None` uses the time average.

B.18.7 `observables`

Type: `dict[str, dict[int, FermionOperator]]`. Default: `dict()`.

Named operator harmonics, allowing both positive and negative Fourier indices.

B.18.8 `metadata`

Type: `dict`. Default: `dict()`.

Coordinate declarations, drive settings, sampling diagnostics and optional user entries.

B.19 Prepared MPS model fields

`qdjj_solver.dmrg_solver.PreparedModel`; source: `src/qdjj_solver/dmrg_solver/mpo.py`.

B.19.1 `hamiltonian`

Type: `Hamiltonian`. Default: `required`.

Original physical finite Hamiltonian, before MPS layout selection.

B.19.2 `sector`

Type: `Sector`. Default: `required`.

Selected total symmetry quantum numbers.

B.19.3 `order`

Type: `tuple[int, ...]`. Default: `required`.

Permutation of zero-based canonical mode indices along the MPS.

B.19.4 `groups`

Type: `tuple[tuple[int, ...], ...]`. Default: `required`.

Canonical modes grouped into each MPS site, one or two per site.

B.19.5 `sites`

Type: `list`. Default: `required`.

TeNPy local fermionic sites with the selected conserved charges.

B.19.6 `lattice`

Type: `TrivialLattice`. Default: `required`.

Finite open TeNPy lattice of the grouped sites; a computational ordering.

B.19.7 `mode_operators`

Type: `dict`. Default: `required`.

Canonical mode mapped to its site index and local annihilation/creation names.

B.19.8 `target_charge`

Type: `tuple[int, ...]`. Default: `required`.

Total TeNPy charge labels encoding the selected sector.

B.19.9 `coordinate_id`

Type: `str`. Default: `required`.

Fingerprint of physical coordinates, ordering, grouping and conserved-charge types.

APPENDIX

C

RESULT REFERENCE

Fields of the result objects. Array shapes are expressed in terms of k eigenstates, n Floquet states, D Fock states, K static basis states and 2M'+1 stored harmonics. The descriptions are the field docstrings in the source.

C.1 QP eigenstates

`qdjj_solver.qp_solver.Eigenstates` (source: `src/qdjj_solver/qp_solver/solver.py`)

energies
: **Type:** `np.ndarray`

  Content: Eigenenergies, shape (k,), in the units of the Hamiltonian.

vectors
: **Type:** `np.ndarray`

  Content: Eigenvectors in the Fock basis, shape (D, k); state i is the column `vectors[:, i]`.

residuals
: **Type:** `np.ndarray`

  Content: Projected residual norms, shape (k,), in energy units.

basis
: **Type:** `FockBasis`

  Content: The `FockBasis` that defines the row order of `vectors` and the canonical occupations.

qp_weights
: **Type:** `np.ndarray`

  Content: Shape (k, q+1); `qp_weights[i, n]` is the probability of n bath quasiparticles in state i.

observables
: **Type:** `dict[str, np.ndarray]`

  Content: Expectation values of the requested named model observables, each of shape (k,); all are measured by default.

timings
: **Type:** `dict[str, float]`

  Content: Elapsed times of the calculation stages, in seconds.

metadata
: **Type:** `dict`

Content: Numerical settings, sector, Hilbert-space restrictions and convergence diagnostics.

hamiltonian
Type: `Hamiltonian | None`

Content: The `Hamiltonian` that was solved.

C.2 DMRG eigenstates

`qdjj_solver.dmrg_solver.Eigenstates` (source: `src/qdjj_solver/dmrg_solver/solver.py`)

energies
Type: `np.ndarray`

Content: Eigenenergies, shape (k,), in the units of the Hamiltonian.

states
Type: `list`

Content: The k eigenstates as TeNPy MPS.

residuals
Type: `np.ndarray | None`

Content: Physical residual norms, shape (k,), in energy units; `None` if residuals were disabled.

observables
Type: `dict[str, np.ndarray]`

Content: Expectation values of the measured observables, each of shape (k,).

timings
Type: `dict[str, float]`

Content: Elapsed times of the calculation stages, in seconds.

metadata
Type: `dict`

Content: Numerical settings, sector, convergence diagnostics and sweep statistics.

hamiltonian
Type: `Hamiltonian`

Content: The `Hamiltonian` that was solved.

prepared
Type: `object`

Content: Canonical basis, conserved quantum numbers, and MPS mode ordering and grouping.

C.3 Floquet states

`qdjj_solver.qp_solver.FloquetStates` (source: `src/qdjj_solver/qp_solver/floquet.py`)

method
Type: `str`

Content: Solver that produced the states: `"sambe"`, `"propagator"` or `"rwa"`.

representation
Type: `str`

Content: `"fock"` or `"static"`: the basis of `modes`.

omega
Type: `float`

Content: Drive frequency.

quasienergies
Type: `np.ndarray`

Content: Quasienergies, shape (n,), on the chosen branch.

folded
Type: `np.ndarray`

Content: Quasienergies folded into the zone around `zone_center`, shape (n,).

mean_energies
Type: `np.ndarray`

Content: Replica-independent time-averaged energies eps - omega , shape (n,).

harmonic_indices
Type: `np.ndarray`

Content: Harmonic indices m of the stored modes, shape (2M'+1,).

modes
Type: `np.ndarray`

Content: Mode harmonics, shape (2M'+1, d, n): u_m of state j is `modes[i, :, j]` with `harmonic_indices[i] == m`, in the basis given by `representation`.

harmonic_weights
Type: `np.ndarray`

Content: Harmonic weights, the squared norms of u_m, shape (n, 2M'+1).

centroids
Type: `np.ndarray`

Content: Harmonic centroids , shape (n,).

boundary_weights
Type: `np.ndarray`

Content: Weight in the outermost harmonic blocks m = -M and M (Sambe) or outside the window -M <= m <= M (propagator); zero for the rotating-wave model; shape (n,).

static_weights
Type: `np.ndarray`

Content: Time-averaged weights of the static eigenstates a, the sum over m of the squared overlaps of a with u_m; shape (n, s), with s = `states` for the Fock basis and s = K for the static basis.

labels
Type: `np.ndarray`

Content: Index of the static state assigned to each Floquet state, shape (n,).

residuals
Type: `np.ndarray`

Content: Residual bound, shape (n,): the Sambe window residual combined with the leakage into the omitted harmonics, or the eigendecomposition residual of U(T) or of the rotating-wave matrix.

basis
Type: `FockBasis`

Content: The QP `FockBasis` of the sector.

static
Type: `Eigenstates`

Content: Static `Eigenstates` of the reference Hamiltonian `periodic.reference`.

periodic
Type: `PeriodicHamiltonian`

Content: The `PeriodicHamiltonian` that was solved.

fock_residuals
Type: `np.ndarray | None`

Content: Fock-space residuals of static-basis results, shape (n,); `None` otherwise.

effective_matrix
Type: `np.ndarray | None`

Content: Rotating-wave matrix of `rwa_model` results; `None` otherwise.

timings
Type: `dict`

Content: Elapsed times of the calculation stages, in seconds.

metadata
Type: `dict`

Content: Settings and diagnostics, for example window residuals, leakage and step errors.

C.4 Probe matrix elements

`qdjj_solver.FloquetTransitions` (source: `src/qdjj_solver/common/spectra.py`)

omega
Type: `float`

Content: Drive frequency.

photons
Type: `np.ndarray`

Content: Photon numbers q of the stored harmonics.

elements
Type: `np.ndarray`

Content: `elements[i, a, b]` is X^(q) for `photons[i]`, final state a and initial state b.

bra_quasienergies
Type: `np.ndarray`

Content: Quasienergies of the final (bra) states.

ket_quasienergies
Type: `np.ndarray`

Content: Quasienergies of the initial (ket) states.

bra_labels
Type: `np.ndarray`

Content: Static labels of the final states.

ket_labels
Type: `np.ndarray`

Content: Static labels of the initial states.

bra_indices
Type: `np.ndarray`

Content: Indices of the final states in their `FloquetStates`.

ket_indices
Type: `np.ndarray`

Content: Indices of the initial states in their `FloquetStates`.

total
Type: `np.ndarray`

Content: Time-averaged squared norm of X(t) applied to each initial state: the right-hand side of the sum rule.

same_states
Type: `bool`

Content: True when bra and ket come from the same `FloquetStates` object.

complete
Type: `bool`

Content: True when the bra states form the complete set of their representation.

default_initial
Type: `int`

Content: Initial state used by `lines()` without arguments: the dressed ground state of the ket sector, `dressed_state(0)`, or the first ket state if that state was not selected.

probe
Type: `str`

Content: Name of the probe.

metadata
Type: `dict`

Content: Probe harmonics, sectors, methods and the matrix-element convention.

C.5 Spectral lines

`qdjj_solver.SpectralLines` (source: `src/qdjj_solver/common/spectra.py`)

frequencies
: **Type:** `np.ndarray`

  Content: Sorted line frequencies; positive values correspond to absorption of a probe quantum.

strengths
: **Type:** `np.ndarray`

  Content: Line strengths, weighted by the initial populations.

final
: **Type:** `list`

  Content: Final-state indices merged into each line (a degenerate group).

initial
: **Type:** `list`

  Content: Initial-state indices that contribute to each line.

photons
: **Type:** `np.ndarray`

  Content: Photon number q of each line.

elastic
: **Type:** `np.ndarray`

  Content: True for elastic lines, between states of one degenerate group at q != 0.

metadata
: **Type:** `dict`

  Content: Probe, initial states, weights and merging options.

APPENDIX

D

OBSERVABLES AND PROBES

Built-in observables are the operators in `h.observables`; solvers report their expectation values as `result.observables[name]`. The definitions are in `OBSERVABLE_DOCS` (`src/qdjj_solver/common/models.py`); conventions and transformations are explained in *models* (page 11).

impurity_charge
: **Definition:** Total physical impurity electron occupation

  Availability: Every `Impurity`

  Units: Electron count; multiply by $-e$ for the electric charge

impurity_spin_z
: **Definition:** Sum of $(n_\uparrow - n_\downarrow)/2$ over the impurity orbitals

  Availability: Every `Impurity`

  Units: Units of $\hbar$

double_occupancy_0
: **Definition:** $n_{0\uparrow} n_{0\downarrow}$; further orbitals use `_1`, `_2`, …

  Availability: Every `Impurity`, one per orbital

  Units: Dimensionless probability

phase_derivative
: **Definition:** $\partial H/\partial\phi$ of a reference junction; in general the derivative along the supplied `phase_velocities` of `make_model`

  Availability: Reference junctions, or nonzero `phase_velocities`

  Units: Energy per radian; the current is $2e/\hbar$ times its expectation value

rho_ws_derivative
: **Definition:** $\partial H/\partial(\rho W_S)$

  Availability: Reference junctions, including `rho_ws=0`

  Units: Energy

D.1 Probes

Probe names accepted by `matrix_elements` and by `Hamiltonian.probe` and `PeriodicHamiltonian.probe` (`PROBE_DOCS` in `src/qdjj_solver/common/probes.py`). The spin-flip and single-electron probes, and `charge` and `spin_z` when `orbital` is given, act on one impurity orbital (`orbital`, default 0) and need an uncompressed model; any other name selects a model observable.

current
Operator: $\partial H/\partial\phi$ (observable `phase_derivative`), with its drive harmonics

Typical use: Flux or phase probe; supercurrent

charge
Operator: n_d (observable `impurity_charge`)

Typical use: Gate probe

spin_z
Operator: S_d^z (observable `impurity_spin_z`)

Typical use: Longitudinal spin probe

spin_x
Operator: $S_d^x = (S_d^+ + S_d^-)/2$

Typical use: Transverse spin resonance

spin_y
Operator: $S_d^y = (S_d^+ - S_d^-)/(2\mathrm{i})$

Typical use: Transverse spin resonance

spin_plus
Operator: $S_d^+ = d_\uparrow^\dagger d_\downarrow$

Typical use: Spin raising

spin_minus
Operator: $S_d^- = d_\downarrow^\dagger d_\uparrow$

Typical use: Spin lowering

d_up
Operator: $d_\uparrow$

Typical use: Electron removal (tunnelling spectroscopy)

d_dn
Operator: $d_\downarrow$

Typical use: Electron removal (tunnelling spectroscopy)

d_up_dagger
Operator: $d_\uparrow^\dagger$

Typical use: Electron addition (tunnelling spectroscopy)

d_dn_dagger
Operator: $d_\downarrow^\dagger$

Typical use: Electron addition (tunnelling spectroscopy)

APPENDIX

E

COMMAND-LINE REFERENCE

Generated from the argument parsers of both programs. Calculation files are described in *Input files and terminal commands* (page 17). Both programs return exit status 0 on success and 2 when a calculation fails.

E.1 `qdjj-solver`

QP and DMRG solvers for superconducting quantum impurities. Module: `qdjj_solver.cli`.

--version
: **Values:** no value

Default:

Meaning: Print the version and exit

E.1.1 `qdjj-solver solve`

Solve a finite Hamiltonian.

configuration
: **Values:** path

Default: required

Meaning: JSON calculation file: model, bath, sector, cutoff, backend and solver options

--backend
: **Values:** `qp`, `dmrg`

Default:

Meaning: Solver backend; overrides the configuration's backend, which defaults to qp

--output
: **Values:** path

Default: required

Meaning: Path of the output JSON result file

--save-states
: **Values:** no value

Default:

Meaning: Also save the states (QP eigenvectors or DMRG matrix-product states) in a companion file next to the result

--resume
Values: path

Default:

Meaning: DMRG result JSON whose saved MPS checkpoint provides the initial states

E.2 `qpsolver`

Variational QP expansions for superconducting impurities. Module: `qdjj_solver.qp_solver.cli`.

--version
Values: no value

Default:

Meaning: Print the version and exit

E.2.1 `qpsolver solve`

Solve a Hamiltonian.

configuration
Values: path

Default: required

Meaning: JSON calculation file in the legacy QP format

--output
Values: path

Default: required

Meaning: Path of the output JSON result file

--save-vectors
Values: no value

Default:

Meaning: Also save the eigenvectors in a companion file next to the result

E.2.2 `qpsolver dimension`

Report the Hilbert-space dimension of the requested sector.

configuration
Values: path

Default: required

Meaning: JSON calculation file in the legacy QP format

E.2.3 `qpsolver fit`

Fit a particle-hole-symmetric BCS surrogate.

--levels
Values: int

Default: required

Meaning: Number of signed spinful levels in the surrogate reservoir

--delta

Values: float

Default: `1.0`

Meaning: Superconducting gap

--bandwidth

Values: float

Default: `100.0`

Meaning: Normal-state half-bandwidth

--frequency-cutoff

Values: float

Default: `100.0`

Meaning: Largest imaginary frequency of the logarithmic fitting mesh

--starts

Values: int

Default: `4`

Meaning: Number of optimization attempts with different initial guesses

--output

Values: path

Default: required

Meaning: Path of the output JSON bath file

`qpsolver` is the legacy QP-only program; `qdjj-solver` is the unified interface.

APPENDIX

F

CONVENTIONS

Physical and numerical conventions shared by the code, this manual, the physics manual (table `tab:conventions`) and the tutorials. Source: `docs/conventions.toml`; *models* (page 5) explains them in detail.

Units

Convention: Energies use the units of the Hamiltonian parameters, usually with $\Delta = 1; \hbar = 1$, so Ω and $1/t$ have energy units

Impurity

Convention: $H_d = \frac{U}{2}(n_d - 1)^2 + \epsilon(n_d - 1) + B\, S_d^z$; the detuning $\epsilon = \epsilon_d + U/2$ vanishes at half filling; $B > 0$ raises the spin-up level

Reservoirs

Convention: BCS leads with the real pairing term $-\Delta_l \sum_i (c_{li\uparrow}^\dagger c_{li\downarrow}^\dagger + \text{h.c.})$, $\Delta_l > 0$, and normal-state half-bandwidth D; normal-state density of states per spin $\rho = 1/(2D)$

Hybridization

Convention: $\Gamma = \sum_l \Gamma_l$ is the total hybridization, $\Gamma_l = \pi\rho|t_l|^2$ with t_l the hopping between the dot and the contact orbital of lead l

Phases

Convention: Factors $\mathrm{e}^{\mathrm{i}\phi_l/2}$ on the tunnelling terms $C_{l\sigma}^\dagger d_\sigma$, i.e. the physical pair potential $\Delta_l \mathrm{e}^{-\mathrm{i}\phi_l}$ and the pair amplitude $\langle c_{li\uparrow}^\dagger c_{li\downarrow}^\dagger \rangle \propto \mathrm{e}^{\mathrm{i}\phi_l}$; the reference junction uses $\phi_{L,R} = \mp\phi/2$, $\phi = \phi_R - \phi_L$

Energy zero

Convention: The isolated BCS vacuum energies are subtracted: at $\Gamma = 0$ the energies are the atomic energies of H_d, with the reservoirs in their quasiparticle vacuum

Current

Convention: $I = (2e/\hbar)\, \partial E/\partial\phi = (2e/\hbar)\langle \partial H/\partial\phi \rangle$, the electric current from the left to the right lead (electron charge $-e$, $e > 0$); reported as $\langle \partial H/\partial\phi \rangle$ in units of Δ

Voltage

Convention: $V = V_L - V_R$, the electric potential of the left lead relative to the right one, so $eV = \mu_R - \mu_L$; Josephson relation $\dot{\phi} = 2eV/\hbar$; IV is the power supplied by the source

Sectors

Convention: Fermion parity (0 even, 1 odd), $2S_z$, optional eta parity and particle number

Drives

Convention: $p(t) = p_0 + \sum_n A_n \cos(n\Omega t + \theta_n)$; A for a monochromatic drive, A_ϕ (radians) for the phase and A_ϵ (energy) for the detuning

Harmonics

Convention: $H(t) = \sum_k H_k \mathrm{e}^{\mathrm{i}k\Omega t}$, $H_{-k} = H_k^\dagger$, time origin $t = 0$

Floquet states

Convention: $\psi(t) = \mathrm{e}^{-\mathrm{i}\varepsilon t} \sum_m \mathrm{e}^{\mathrm{i}m\Omega t} u_m$ with quasienergy ε; Sambe matrix $H_{m-m'} + m\Omega\delta_{mm'}$, truncated at the Floquet cutoff $|m| \leq M$

APPENDIX

G

FROM THE EQUATIONS TO THE CODE

Where each concept of the physics manual lives in the code, the user manual and the tutorials. Source: `docs/concordance.toml`; the physics manual prints the same table (chapter "From the equations to the code", `ch:equations-to-code`).

Superconducting Anderson impurity and reference junction

Physics manual: `eq:reference`

Code: `reference_model`, `Impurity.anderson`, `make_model`

User manual: *Physical models and conventions* (page 5)

Tutorial: `01_subgap_states`

Discretized superconducting reservoirs

Physics manual: `sec:bath`

Code: `cosh_grid`, `fit_surrogate`, `chain_expansion`, `DiscreteBath`

User manual: *Choosing and refining superconducting reservoirs* (page 25)

Tutorial: `05_convergence`

Symmetry sectors

Physics manual: `sec:symmetries`

Code: `Sector`

User manual: *Physical models and conventions* (page 5)

Tutorial: `01_subgap_states`

Canonical coordinates

Physics manual: `sec:coordinates`

Code: `Hamiltonian.physical_operator`, `Hamiltonian.coordinate_fingerprint`

User manual: *Physical models and conventions* (page 5)

Tutorial:

Quasiparticle expansion

Physics manual: `eq:qp-projection`

Code: `qp_solver.solve`, `qp_solver.SolverOptions`

User manual: *QP calculations and observable-specific convergence* (page 28)

Tutorial: `00_first_calculation`

Density-matrix renormalization group
Physics manual: `sec:dmrg`

Code: `dmrg_solver.solve`, `dmrg_solver.SolverOptions`

User manual: *DMRG states and bond-dimension refinement* (page 32)

Tutorial: `11_dmrg_convergence`

Checked calculations and saved results
Physics manual: `ch:practical`

Code: `solve`, `save_result`, `load_result`

User manual: *Parameter scans, saved results and reproducible analysis* (page 36)

Tutorial: `01_subgap_states`

Supercurrent
Physics manual: `eq:current`

Code: `phase_derivative`

User manual: *Physical models and conventions* (page 5)

Tutorial: `01_subgap_states`

Transition frequencies and static matrix elements
Physics manual: `eq:transitions`

Code: `Eigenstates.matrix_elements`, `Hamiltonian.probe`

User manual: *Using the solver and saving results* (page 13)

Tutorial: `06_phase_drive`

Parameter drives and harmonics
Physics manual: `eq:drive`

Code: `ParameterDrive`, `Harmonic`, `PeriodicHamiltonian.from_parameters`

User manual: *Floquet calculations* (page 41)

Tutorial: `06_phase_drive`

Floquet states and quasienergies
Physics manual: `eq:floquet-state`

Code: `FloquetStates.quasienergies`, `FloquetStates.modes`

User manual: *Floquet calculations* (page 41)

Tutorial: `02_driven_dot`

Mean energy
Physics manual: `eq:mean-energy`

Code: `FloquetStates.mean_energies`

User manual: *Floquet calculations* (page 41)

Tutorial: `02_driven_dot`

Sambe matrix and Floquet cutoff
Physics manual: `eq:sambe-matrix`

Code: `solve_floquet`, `FloquetOptions.floquet_cutoff`

User manual: *Floquet calculations* (page 41)

Tutorial: `05_convergence`

Static basis of size K

Physics manual: `sec:sambe-method`

Code: `FloquetOptions.basis`, `FloquetOptions.basis_states`

User manual: *Floquet calculations* (page 41)

Tutorial: `05_convergence`

Replicas, branches and folded quasienergies

Physics manual: `sec:sambe-method`

Code: `FloquetOptions.selection`, `FloquetOptions.branch`, `FloquetStates.folded`

User manual: *Floquet calculations* (page 41)

Tutorial:

Residual bound

Physics manual: `eq:residual-bound`

Code: `FloquetStates.residuals`, `FloquetStates.fock_residuals`

User manual: *Floquet calculations* (page 41)

Tutorial: `05_convergence`

Gauge freedom and c-number terms

Physics manual: `sec:gauge`

Code: `FloquetOptions.drop_scalar_harmonics`

User manual: *Floquet calculations* (page 41)

Tutorial: `02_driven_dot`

Fixed canonical coordinates

Physics manual: `sec:fixed-coordinates`

Code: `PeriodicHamiltonian.coordinate_fingerprint`

User manual: *Floquet calculations* (page 41)

Tutorial:

One-period propagator

Physics manual: `eq:cfm4`

Code: `floquet_propagator`, `FloquetOptions.steps`

User manual: *Floquet calculations* (page 41)

Tutorial: `05_convergence`

High-frequency expansion

Physics manual: `eq:van-vleck`

Code: `PeriodicHamiltonian.van_vleck`

User manual: *Floquet calculations* (page 41)

Tutorial: `04_high_frequency`

Rotating-wave model

Physics manual: `eq:rwa`

Code: `rwa_model`

User manual: *Floquet calculations* (page 41)

Tutorial: `03_two_level_system`

Floquet matrix elements

Physics manual: `eq:floquet-element`

Code: `FloquetStates.matrix_elements`, `FloquetTransitions`

User manual: *Floquet calculations* (page 41)

Tutorial: `08_two_tone`

Floquet golden rule and lines

Physics manual: `eq:floquet-golden`

Code: `FloquetTransitions.lines`, `SpectralLines.spectrum`, `line_shape`

User manual: *Floquet calculations* (page 41)

Tutorial: `08_two_tone`

Sum rule

Physics manual: `eq:sum-rule`

Code: `FloquetTransitions.completeness`

User manual: *Floquet calculations* (page 41)

Tutorial: `02_driven_dot`

APPENDIX

H

CALCULATION INPUT FORMAT

The descriptions are maintained in `docs/formats.toml`; implementation locations are listed below. See the *constructor reference* (page 61) for parameter names and defaults, *options* (page 53) for solver settings, and *results* (page 74) for the structure of result objects in memory.

H.1 Calculation envelope

Implementation: `src/qdjj_solver/common/io.py: validate_config; src/qdjj_solver/cli.py`.

The unified CLI reads one JSON object. Unknown keys are rejected, with their location included in the error message. Version 1 describes static calculations; use Python to construct periodic Hamiltonians and calculate spectra.

H.1.1 `format_version`

Integer 1; omitted means 1. Booleans and other versions are rejected.

H.1.2 `backend`

'qp' or 'dmrg'; omitted means QP in the unified CLI. –backend overrides the file. The legacy qpsolver input rejects this key.

H.1.3 `model`

Model object; omitted selects reference-model defaults. The kind defaults to 'reference'. See the constructor reference for every parameter.

H.1.4 `bath`

Reference-model reservoir object. Omitted means a cosh grid with pairs=8; an explicitly empty object lacks the required pairs argument. Other model kinds ignore this top-level field; general models specify model.reservoirs instead.

H.1.5 `sector`

Sector constructor keys parity, twice_sz, eta, particle_number. Omitted uses Sector(); non-null labels must be integers.

H.1.6 `cutoff`

Nonnegative integer or null. Omitted means 2 for QP CLI, null for DMRG. DMRG requires null, including after a backend override. Python solve instead defaults to None.

H.1.7 `solver`

Object containing exactly the selected backend's SolverOptions keys. Omitted fields use that backend's defaults. Unknown or backend-incompatible settings fail.

H.2 Reference model

Implementation: `src/qdjj_solver/common/io.py: model_from_config`.

model.kind='reference'. Remaining model keys are reference_model keywords except bath, which is supplied by the top-level bath object. The unified CLI sets compress=false when omitted; the legacy QP interface and Python constructor default to true.

H.2.1 `kind`

Literal 'reference', or omitted.

H.2.2 `parameters`

u, gamma, phi, rho_ws, rho_wn, geometry, detuning, field, symmetry, compress, bath_reference, coefficient_tolerance; generated names/defaults and constraints are in the constructor reference.

H.3 Reservoir objects

Implementation: `src/qdjj_solver/common/io.py: bath_from_config`.

Used as bath for reference models and as each model.reservoirs element for general models. All gaps and half-bandwidths are positive finite energies.

H.3.1 `kind`

'cosh-grid' (default), 'surrogate', 'chain-expansion' or 'discrete'.

H.3.2 `cosh-grid`

Arguments of cosh_grid: pairs is required; delta and bandwidth have constructor defaults.

H.3.3 `surrogate`

Arguments of fit_surrogate: levels is required; all mesh, objective, start and budget controls follow the constructor reference.

H.3.4 `chain-expansion`

Arguments of chain_expansion: levels is required; wide_band is boolean.

H.3.5 discrete

Arguments of DiscreteBath: xi and weights are required equal-length nonempty arrays; weights are positive. Optional delta, bandwidth and metadata follow the class reference.

H.3.6 metadata

Optional string-keyed object for constructors that accept it, notably an explicit discrete bath. Check the selected constructor's supported keywords.

H.4 Multi-orbital models

Implementation: `src/qdjj_solver/common/io.py: model_from_config`.

model.kind='multi-orbital'. Required fields are impurity, reservoirs and tunneling. Other fields follow make_model, with reservoirs replacing its baths argument.

H.4.1 kind

Literal 'multi-orbital'.

H.4.2 impurity

Object with required orbitals and terms; optional observables and metadata. orbitals is positive; terms are a list of operator records.

H.4.3 reservoirs

Nonempty array of reservoir objects, one per lead.

H.4.4 tunneling

Array of L matrices, each shape (2, 2*orbitals), real or explicit complex-array objects; amplitudes in energy units.

H.4.5 phases

Optional list of L finite phases in radians; omitted means zero.

H.4.6 direct

Optional list of contact records, each with leads and matrix; omitted means none.

H.4.7 phase_velocities

Optional L phase derivatives for the phase_derivative observable.

H.4.8 eta_basis

Optional boolean; only a verified matched single-dot symmetry permits true.

H.4.9 `compress_pairs`

Optional boolean dark-mode restriction; requires eligible eta coordinates.

H.4.10 `bath_reference`

'isolated' or 'coupled'; restrictions are those of make_model.

H.4.11 `coefficient_tolerance`

Nonnegative finite coefficient cleanup scale.

H.4.12 `direct_derivatives`

Object mapping observable names to lists of contact records; choose names distinct from impurity observables and phase_derivative.

H.5 Direct Hamiltonians

Implementation: `src/qdjj_solver/common/io.py: model_from_config; common/problem.py: Hamiltonian`.

model.kind='hamiltonian'. Other fields are the Hamiltonian record: operator, nimp and spins are required. This uses the specified additive constant and canonical coordinates.

H.5.1 `operator`

List of operator-term records. Must define a finite Hermitian parity-conserving operator on declared modes.

H.5.2 `nimp`

Number of initial canonical spin modes treated as impurity modes; between zero and the total mode count.

H.5.3 `spins`

Array of +1/-1 twice-spin labels, one per mode.

H.5.4 `eta_labels`

Optional +1/-1 labels with the same length as spins, or null.

H.5.5 `observables`

Optional string-keyed mapping from names to operator-term lists.

H.5.6 `metadata`

Optional object, including coordinate and energy-reference declarations where applicable.

H.6 Nested numerical encodings

Implementation: `src/qdjj_solver/common/io.py: complex_array, _operator_from_config, _contacts_from_config`.

Specify input indices and array dimensions from the physical mode ordering and contact definitions.

H.6.1 `complex-array`

Either an ordinary real array or an object with exactly real and imag arrays of identical shape. Complex scalars use the analogous real/imag encoding in records.

H.6.2 `operator-term`

Exactly three fields: operators (an array of signed, nonzero, one-based mode indices), real and imag (finite coefficient components). Positive indices denote creation, negative indices annihilation; the rightmost operator acts first. An empty operators array represents a scalar.

H.6.3 `contact`

Exactly leads and matrix. leads contains two integer indices 0<=l<m; duplicates fail. matrix is a finite complex 2-by-2 spin matrix.

H.6.4 `impurity-observables`

Object mapping names to lists of impurity operator terms; uses physical impurity modes.

H.6.5 `user-metadata`

Store user notes in the metadata dictionaries supported by the model and bath constructors.

APPENDIX

I

SAVED RECORDS AND ARTIFACTS

The descriptions are maintained in `docs/formats.toml`; implementation locations are listed below. See the *constructor reference* (page 61) for parameter names and defaults, *options* (page 53) for solver settings, and *results* (page 74) for the structure of result objects in memory.

I.1 Common scalar result envelope

Implementation: `src/qdjj_solver/common/io.py: result_record; qp_solver/io.py; dmrg_solver/io.py`.

save_result supports static QP and DMRG eigenstates. It writes JSON with format='qdjj-eigenstates', format_version=1 and backend='qp' or 'dmrg', and returns None. Scalar JSON can be analysed with ordinary JSON tools; load_result additionally requires a state artifact.

I.1.1 `format`

Literal 'qdjj-eigenstates'.

I.1.2 `format_version`

Integer 1.

I.1.3 `backend`

'qp' or 'dmrg'; selects the loader and backend-specific metadata.

I.1.4 `energies`

One real many-body energy per state in the declared Hamiltonian reference. Obtain excitation gaps by taking energy differences.

I.1.5 `residuals`

One residual norm per state; DMRG can use null when residual evaluation was disabled.

I.1.6 `observables`

Mapping from observable name to one value per state, using complex encoding where necessary.

I.1.7 `timings_seconds`

Named elapsed-time measurements, with method-dependent keys; use them to assess computational cost in the recorded environment.

I.1.8 `calculation`

The result object's metadata dictionary, stored under the key calculation: settings, sector and diagnostics.

I.1.9 `environment`

Package/library versions, Python, platform and machine information.

I.1.10 `implementation`

Imported implementation version, portable relative file hashes and their aggregate SHA256.

I.1.11 `hamiltonian_sha256`

Fingerprint of the complete Hamiltonian record.

I.1.12 `hamiltonian`

Full Hamiltonian record in standard save_result output. The lower-level record helper can omit it; retain it for state restoration.

I.1.13 `qp_weights`

QP only: state-by-bath-QP-number probabilities, with shape (states, cutoff+1).

I.1.14 `state_artifact`

Present only when states are saved; kind, format_version, file and sha256 identify the companion.

I.1.15 `state_amplitudes`

QP saves also retain the legacy file/sha256 companion descriptor.

I.2 QP calculation metadata

Implementation: `src/qdjj_solver/qp_solver/solver.py`.

The QP calculation copies the model metadata and adds the following fields. Additional method-specific diagnostics may also be present; this dictionary is extensible.

I.2.1 `sector`

All four Sector fields.

I.2.2 `qp_cutoff`

Resolved integer bath-QP cutoff.

I.2.3 `dimension`

Number of retained Fock basis states.

I.2.4 `bath_modes`

Number of declared non-impurity modes.

I.2.5 `impurity_modes`

Number of impurity spin modes.

I.2.6 `method`

Selected eigensolver method after automatic dispatch.

I.2.7 `options`

Resolved SolverOptions dictionary.

I.2.8 `basis_bytes`

Native basis storage estimate in bytes.

I.2.9 `estimated_solver_bytes`

Estimated solver storage in bytes; actual process memory also includes allocations outside this estimate.

I.2.10 `real_arithmetic`

Whether the projected calculation used real arithmetic.

I.2.11 `nnz`

Stored sparse entry count if available, otherwise null.

I.3 DMRG calculation metadata

Implementation: `src/qdjj_solver/dmrg_solver/solver.py`.

The finite-problem convergence flag requires satisfactory residuals, stable sweep energies, energy ordering and orthogonality. Assess bath convergence by refinement, and check the completeness of the low-energy spectrum through additional state searches or an independent reference.

I.3.1 `backend`

Literal ‘dmrg’.

I.3.2 `tenpy`

TeNPy version used.

I.3.3 `options`

Resolved DMRG SolverOptions.

I.3.4 `sector`

Selected Sector fields.

I.3.5 `qp_cutoff`

Null: DMRG retains all QP numbers allowed by the declared model and sector.

I.3.6 `bath_modes`

Declared non-impurity mode count.

I.3.7 `impurity_modes`

Impurity spin-mode count.

I.3.8 `mode_order`

Canonical mode permutation along the MPS.

I.3.9 `groups`

Modes assigned to each MPS site.

I.3.10 `coordinate_id`

Physical-coordinate and layout fingerprint.

I.3.11 `energy_reference`

Description inherited from the Hamiltonian, or 'as specified in Hamiltonian'.

I.3.12 `restriction`

Any declared dark-mode restriction inherited from model construction; DMRG acts on the retained modes.

I.3.13 `search_energy_shift`

Scalar search shift used during excited-state optimization.

I.3.14 `operator_norm_bound`

Operator-coefficient norm bound used in the search.

I.3.15 `mpo_dimensions`

Physical MPO bond dimensions, or null for a zero operator.

I.3.16 `mpo_original_dimensions`

Dimensions before exact parallel-channel merging, or null.

I.3.17 `search_shift_bound`

Method used to choose the search bound, or null.

I.3.18 `roots`

Optimization diagnostics for each state, including independent trials, sweep histories, actual bond dimensions and stopping information.

I.3.19 `gram`

Full state overlap matrix, bra rows and ket columns.

I.3.20 `gram_error`

Norm of the deviation of gram from the identity.

I.3.21 `roots_energy_ordered`

Whether the retained eigenstates are ordered by energy.

I.3.22 `residual_method`

'untruncated-(H-E)-MPS-QR' when residuals are evaluated, otherwise null.

I.3.23 `finite_problem_converged`

Combined finite-Hamiltonian convergence assessment.

I.3.24 `bath_convergence_checked`

False: bath convergence requires a separate study.

I.4 DMRG root and trial diagnostics

Implementation: `src/qdjj_solver/dmrg_solver/solver.py`.

Each calculation.roots entry describes the selected candidate for one eigenstate. Iteratively obtained states include sweep and trial data; exact single-site and zero-operator solutions have shorter records.

I.4.1 `sweep_converged`

TeNPy's combined energy/entropy stopping flag, separate from the full finite-problem assessment.

I.4.2 `sweep_energy_converged`

Stability of the last three sweep energies within four times energy_tolerance.

I.4.3 `sweeps`

Number of optimization sweeps; zero for exact special cases.

I.4.4 `chi_actual`

Largest actual MPS bond dimension; one for exact single-site/zero-operator cases.

I.4.5 `seed`

Product-state indices, 'continuation', an excitation-operator/source-root object, or randomized-seed details. Absent for exact special cases.

I.4.6 `lower_root_overlap`

Largest magnitude of the overlap amplitude with a previously calculated eigenstate; absent for exact special cases.

I.4.7 `norm_error`

Norm of TeNPy's canonical-form diagnostic; absent for exact special cases.

I.4.8 `sweep_statistics`

TeNPy per-sweep arrays; non-finite diagnostics are encoded as null. Names depend on the installed TeNPy version.

I.4.9 `seed_trials`

List of attempted candidates, each with energy, sweep_converged, lower_root_overlap and seed; empty for exact special cases.

I.4.10 `method`

Present for exact special cases: 'single-site-exact' or 'zero-operator'.

I.5 Companion wavefunctions

Implementation: `src/qdjj_solver/qp_solver/io.py; src/qdjj_solver/dmrg_solver/io.py`.

Keep the JSON record and its named companion file in the same directory. Saving uses unique companion filenames and replaces the JSON record atomically. Obtain the companion filename from the JSON record's state_artifact.file field.

I.5.1 `kind`

'fock-amplitudes-npz' or 'tenpy-mps-hdf5'.

I.5.2 `format_version`

Companion descriptor version 1.

I.5.3 `file`

Basename of the companion file in the result directory.

I.5.4 `sha256`

Checksum of the companion bytes, checked before loading.

I.5.5 `NPZ`

energies, vectors (basis rows, eigenstate columns), occupations (uint64 word rows), qp_counts, result_sha256. The last binds the scalar record; older four-array files remain readable.

I.5.6 `HDF5`

TeNPy data containing format='qdjj-mps', format_version=1, coordinate_id, result_sha256 and states. Load trusted MPS artifacts with compatible TeNPy/HDF5 support.

I.5.7 `validation`

Loaders check format, checksum, Hamiltonian and coordinate/basis identities, shapes and state consistency. Editing scalar records can invalidate their binding to wavefunctions.

I.5.8 `legacy-QP`

format='bcs-qp-eigenstates', version 1, is also readable. It uses state_amplitudes and lacks the common backend/provenance envelope. New unified saves use the common format.

I.6 Model and spectral records

Implementation: `common/baths.py`; `common/problem.py`; `common/floquet.py`; `common/spectra.py`.

Each object's record() method provides the fields listed below for explicit JSON encoding of arrays and complex values. These records describe models, drives and spectral lines; save_result and load_result handle static eigenstates and their companion wavefunctions.

I.6.1 `DiscreteBath`

xi, weights, delta, bandwidth, metadata. from_record reconstructs and validates coefficients.

I.6.2 `Hamiltonian`

operator, nimp, spins, eta_labels, observables, metadata. Operators are serialized term lists; the additive constant is retained.

I.6.3 `Harmonic`

order, amplitude, phase.

I.6.4 `ParameterDrive`

parameter, index, harmonics (Harmonic records).

I.6.5 `PeriodicHamiltonian`

omega, nimp, spins, eta_labels, harmonics (string nonnegative indices to operator lists), reference (Hamiltonian record), observables (signed harmonic keys), metadata.

I.6.6 `SpectralLines`

frequencies, strengths, final, initial, photons, elastic, metadata. Merged initial/final index lists and population-weighted strengths retain their interpretation.

APPENDIX

J

API REFERENCE

Each public object is documented once, under the first module below that exports it; the lists are the modules' `__all__`. The legacy `qpsolver` package re-exports `qdjj_solver.qp_solver`, so those entries also serve as its reference. Option and result fields are also tabulated in the *option* (page 53) and *result* (page 74) references.

J.1 Package `qdjj_solver`

Models, reservoirs, the operator algebra, periodic drives, spectra and the backend dispatch.

`qdjj_solver.`**`solve`**`(`
hamiltonian`,`
cutoff=None`,`
sector=DEFAULT_SECTOR`,`
options=None`,`
initial=None`,`
** (Keyword-only parameters separator (PEP 3102))*`,`
backend='qp'`,`
`)`

Lowest eigenstates of a finite Hamiltonian in one symmetry sector with the chosen backend.

`backend` is `"qp"` (`cutoff` limits the number of bath quasiparticles; `None` keeps the full finite bath) or `"dmrg"` (`cutoff` must be `None`). `options` is the backend's `SolverOptions` or a dictionary of its fields. `initial` is a QP starting vector in the target Fock basis, or a compatible prior DMRG result. Returns the backend's `Eigenstates`.

`qdjj_solver.`**`save_result`**`(`*result*, *path*, *save_states=False*`)`

Write the common version-1 result record and, with `save_states=True`, a companion file with the states (QP eigenvectors or DMRG matrix-product states).

`qdjj_solver.`**`load_result`**`(`*path*`)`

Restore saved states, selecting the backend from the versioned record.

The companion state file is required. Legacy QP records are also readable. QP loading uses the base installation; DMRG loading additionally requires TeNPy.

`class` `qdjj_solver.`**`DiscreteBath`**

Spin-degenerate normal levels and contact weights for one BCS reservoir.

`weights` represents the normal-state measure rho d xi. Each normal level is a spinful orbital. All QP excitation energies are positive. At finite size the sum of surrogate contact weights may differ from one.

xi: ndarray

Nonempty one-dimensional finite signed normal energies, in Hamiltonian energy units.

weights: ndarray

Strictly positive normal-state measure weights, same shape as `xi`; their supplied values are preserved.

delta: float = 1.0

Positive finite superconducting gap in energy units.

bandwidth: float = 100.0

Positive finite normal-state half-bandwidth; sets the density per spin to 1/(2D).

metadata: dict

Construction parameters, fit diagnostics and optional user provenance.

property levels

Number of signed spinful normal levels in this reservoir; each level has two QP spin modes.

property rho

Normal-state density per spin, 1/(2 bandwidth), in inverse energy units.

property energies

Positive Bogoliubov energies sqrt(xi**2 + delta**2), one per spinful level.

property paired

Exact mirrored coordinates and weights, as required for exact mode reduction.

record()

JSON-ready coefficients and metadata; retains the actual unnormalized weights.

classmethod from_record(*record*)

Reconstruct and validate the coefficient record returned by `record`.

__init__(
xi: ndarray,
weights: ndarray,
delta: float = 1.0,
bandwidth: float = 100.0,
metadata: dict = <factory>,
) → None

class qdjj_solver.**FermionOperator**

Finite sum of fermionic monomials; scalar multiplication uses *.

Multiplication normal-orders the *full* operator, not matrices of singly projected creation operators. The distinction matters at the QP boundary.

__init__(*terms=None*)

classmethod scalar(*value*)

A finite real or complex multiple of the identity (the empty monomial).

dagger()

Hermitian conjugate, reversing operator order and conjugating coefficients.

substitute(*annihilators*)

Substitute canonical annihilation operators (e.g. a Bogoliubov map).

cleaned(*tolerance=1e-14*)

Remove only floating-point cancellation at the specified absolute scale.

hermiticity_error()

Largest coefficient magnitude in O - O.dagger(); zero for a Hermitian polynomial.

to_records()

Sorted monomials with signed one-based indices and real/imaginary coefficients.

classmethod from_records(*records*)

Sum and normal-order the serialized monomials; repeated terms are accumulated.

class qdjj_solver.**Hamiltonian**

A finite, Hermitian, parity-conserving Hamiltonian in canonical fermionic coordinates.

The first `nimp` modes are impurity spin orbitals; `spins` holds +1 or -1 for every mode and `eta_labels` optional eta-parity labels. `observables` maps names to operators in the same coordinates; `metadata` records the construction, including the electron-to-canonical map that `physical_operator` uses.

operator: *FermionOperator* (page 105)

Finite Hermitian, parity-conserving polynomial in the declared canonical modes.

nimp: int

Number of impurity spin modes, occupying the first mode indices; zero is allowed.

spins: tuple[int, ...]

Twice-spin labels +1 or -1 for every canonical mode, in basis order.

eta_labels: tuple[int, ...] | None = None

Optional +1/-1 eta label per mode; required when imposing eta parity.

observables: dict[str, *FermionOperator* (page 105)]

Named operators in the same canonical coordinates as the Hamiltonian.

__init__(*operator: ~qdjj_solver.common.algebra.FermionOperator, nimp: int, spins: tuple[int, ...], eta_labels: tuple[int, ...] | None = None, observables: dict[str, ~qdjj_solver.common.algebra.FermionOperator] = <factory>, metadata: dict = <factory>*) → None

metadata: dict

Construction, energy-reference and coordinate information; may include user entries.

property bath_modes

Number of declared non-impurity spin modes, including retained dark modes.

record()

Full operator, mode labels, observables and metadata as a JSON-ready record.

classmethod from_record(*record*)

Reconstruct and validate a Hamiltonian record, preserving its additive constant.

check_sector(*sector*)

Require every term of the Hamiltonian to conserve the sector's quantum numbers.

fingerprint()

SHA256 of the full canonical record, including observables and metadata.

coordinate_fingerprint()

A hash that identifies the canonical coordinates, independent of the numerical backend.

User-built Hamiltonians can declare their basis in `metadata['coordinates']`. Without this declaration the hash covers the entire Hamiltonian, so that states computed in different bases are never combined by accident.

probe(*name*, *orbital=None*)

A named probe operator in canonical coordinates (see `common.probes`).

physical_operator(*operator*)

Transform a physical electron operator into the model's canonical coordinates.

The electron modes are ordered as the impurity modes, then the reservoir modes by reservoir, level and spin. Arbitrary products require the complete bath map, so this method rejects models with paired-mode compression, whose discarded dark modes are frozen in their vacuum; the model's built-in observables already include the contractions required by that restriction.

class qdjj_solver.**Impurity**

Untruncated spinful impurity; modes are (orbital 0 up, down, 1 up, …).

orbitals: int

Positive number of physical impurity orbitals; each contributes up/down spin modes.

operator: *FermionOperator* (page 105)

Hermitian, parity-conserving impurity operator in physical electron coordinates.

observables: dict[str, *FermionOperator* (page 105)]

Additional impurity operators; charge, spin-z and orbital double occupancies are added automatically.

metadata: dict

Physical construction parameters and optional user information.

classmethod anderson(*u=2.*, *detuning=0.*, *field=0.*)

H_imp=U(n-1)^2/2+detuning(n-1)+field S_z (single orbital).

classmethod from_integrals(*one_body*, *interaction=None*, *constant=0.*)

H=sum h_ab d†_a d_b + (1/4)sum U_abcd d†_a d†_b d_d d_c.

All indices are spin-orbital indices. The factor 1/4 is appropriate to antisymmetrized two-electron integrals. A general FermionOperator may instead be supplied through the main constructor.

__init__(
orbitals: int,
operator: FermionOperator *(page 105)*,
observables: dict[str,
~qdjj_solver.common.algebra.FermionOperator]=<factory>,
metadata: dict = <factory>,
) → None

class qdjj_solver.**Sector**

Symmetry sector: fermion parity (0 even, 1 odd), twice S_z, and optionally the eta parity and the particle number of the canonical modes; `None` leaves a quantum number unrestricted.

parity: int = 1

Total fermion parity: zero is even and one is odd; it includes impurity and bath.

twice_sz: int | None = 1

Twice total spin projection, an integer compatible with parity; `None` leaves it unrestricted.

eta: int | None = None

Optional eta parity, +1 or -1; requires declared eta labels and a conserving Hamiltonian.

particle_number: int | None = None

Optional nonnegative total occupation of the chosen canonical modes, compatible with parity and spin.

__init__(
parity: int = 1,
twice_sz: int | None = 1,
eta: int | None = None,
particle_number: int | None = None,
) → None

qdjj_solver.**annihilate**(*mode: int*) → *FermionOperator* (page 105)

Annihilation operator of a canonical fermionic mode (zero-based index).

qdjj_solver.**chain_expansion**(
levels: int,
delta=1.,
bandwidth=100.,
***,
wide_band=False,
) → *DiscreteBath* (page 104)

Bobok–Frk–Pokorny–Zonda ChE(L), returned in exact mirrored star coordinates.

`levels` sets the number of spinful sites in one reservoir; choose the QP cutoff separately. The default targets a finite flat band. `wide_band=True` uses Eq. (17)'s analytic infinite-band coefficients; then `bandwidth` only specifies the density convention rho=1/(2D), and cancels from physical hybridizations.

Finite-band Padé coefficients are obtained from equivalent positive Gauss (even L) or endpoint-Radau (odd L) quadrature, avoiding a Hankel moment solve. Coarse contact weights retain the measure obtained from that construction.

qdjj_solver.**cosh_grid**(
pairs: int,
delta=1.,
bandwidth=100.,
) → *DiscreteBath* (page 104)

Gauss–Legendre quadrature in xi=±Delta sinh(t), E=Delta cosh(t).

There are `2*pairs` signed normal levels *per reservoir*. Coarse rules retain their actual quadrature weights to preserve the contact measure. The zeroth-moment error is recorded.

qdjj_solver.**create**(*mode: int*) → *FermionOperator* (page 105)

Creation operator of a canonical fermionic mode (zero-based index).

qdjj_solver.**fit_surrogate**(
levels: int,
delta=1.,
bandwidth=100.,
frequency_cutoff=100.,
frequency_min=1e-3,
frequency_points=1000,
starts=4,
seed=1729,
relative_weight=0.,
***,
max_nfev=5000,
) → *DiscreteBath* (page 104)

Positive, particle-hole symmetric least-squares fit of the BCS g function.

The default `relative_weight=0` is the absolute least-squares criterion on a logarithmic frequency mesh used by Baran, Frost and Paaske (2023). A nonzero exponent changes the criterion and is therefore recorded. `frequency_min` and `frequency_cutoff` are in the Hamiltonian's units. `max_nfev` bounds function evaluations per start; assess accuracy using the resulting fit diagnostics.

qdjj_solver.**hermitian_pair**(
operator: FermionOperator *(page 105)*,
) → *FermionOperator* (page 105)

The operator plus its Hermitian conjugate.

qdjj_solver.**make_model**(
impurity: Impurity *(page 107)*,
baths,
tunneling,
phases=None,
direct=None,
phase_velocities=None,
eta_basis=False,
compress_pairs=False,
bath_reference='isolated',
coefficient_tolerance=1e-14,
direct_derivatives=None,
) → *Hamiltonian* (page 106)

Construct a multi-orbital impurity coupled to one or more BCS reservoirs.

`tunneling[l]` is a (2, 2*n_orbitals) matrix T_l in C_l† T_l d + h.c., with C_l=sum_i sqrt(w_li)c_li. `direct[(l,m)]` multiplies C_l† W_lm C_m for l<m. Pair potentials are real, -Delta_l (c†_up c†_dn + h.c.); the phases enter as exp(i phi_l/2) T_l and exp(i(phi_l-phi_m)/2) W_lm. This is the gauge c_l -> exp(-i phi_l/2) c_l of leads with the physical pair potential Delta_l exp(-i phi_l) and real tunnelling amplitudes.

The optional eta/paired-mode reduction is restricted to the symmetric, particle-hole-symmetric single-dot junction. It preserves bath QP number. Decoupled modes are kept in their vacuum when compress_pairs=True; this option targets bound impurity states, not arbitrary continuum excitations.

`direct_derivatives[name][(l,m)]` specifies a physical contact-hopping observable with the same convention as `direct`. It is transformed even when the corresponding Hamiltonian parameter is zero. Discarded-mode vacuum contractions are retained, including in a compressed bath.

`qdjj_solver.`**`number`**(*mode: int*) → *FermionOperator* (page 105)

Occupation-number operator of a canonical fermionic mode.

`qdjj_solver.`**`reference_model`**(
bath,
u=2.,
gamma=.4,
*phi=3 * np.pi / 5*,
rho_ws=0.,
rho_wn=0.,
geometry='junction',
detuning=0.,
field=0.,
symmetry=True,
compress=True,
bath_reference='isolated',
coefficient_tolerance=1e-14,
)

Reference single-dot model; Gamma is the *total* hybridization.

The dimensional interlead amplitude is W_S=(rho W_S)/rho. It acquires sqrt(w_i w_j), including fitted surrogate residues, upon discretization.

class `qdjj_solver.`**`Harmonic`**

One cosine component A cos(n omega t + theta) of a parameter modulation.

`order: int`

Positive integer multiple of the fundamental drive frequency.

`amplitude: float`

Finite real cosine amplitude in the driven parameter's units.

`phase: float = 0.0`

Finite cosine phase offset in radians.

`record`()

JSON-ready order, amplitude and phase for this cosine component.

`__init__`(*order: int*, *amplitude: float*, *phase: float = 0.0*) → None

class `qdjj_solver.`**`ParameterDrive`**

p(t)=p0+sum_n A_n cos(n omega t+theta_n) for one builder keyword.

`index` selects one element of an array keyword, such as `phases` of *`make_model()`* (page 109). Amplitudes carry the parameter's units (radians for phases).

`parameter: str`

Nonempty keyword name in the model builder's undriven parameter mapping.

`harmonics: tuple[`*`Harmonic`* **`(page 110), ...]`

Nonempty sequence of cosine components with distinct positive orders.

index: int | None = None

Optional nonnegative index of an array parameter; None selects a scalar parameter.

classmethod cosine(*parameter*, *amplitude*, *phase=0.*, *order=1*, *index=None*)

A single harmonic: p(t)=p0+amplitude*cos(order*omega*t+phase).

property max_order

Highest cosine order in this parameter modulation.

value(*t*, *omega*)

Displacement p(t)-p0.

record()

JSON-ready parameter name, optional array index and harmonic records.

__init__(
parameter: str,
*harmonics: tuple[*Harmonic *(page 110), ...]*,
index: int | None = None,
) → None

class qdjj_solver.**PeriodicHamiltonian**

Hermitian H(t+T)=H(t), stored as harmonics H_k (k>=0) in fixed canonical coordinates.

reference is the static Hamiltonian whose eigenstates label the dressed states; it defaults to the time average H_0. observables[name][k] holds the harmonics (both signs of k) of the model observables, used for probe matrix elements between Floquet states.

omega: float

Positive finite fundamental angular frequency in energy units with hbar=1.

harmonics: dict[int, *FermionOperator* (page 105)]

Nonnegative Fourier indices mapped to operators; negative harmonics are Hermitian conjugates.

nimp: int

Number of impurity spin modes, identical for every harmonic and the reference.

spins: tuple[int, ...]

Twice-spin label +1 or -1 per fixed canonical mode.

eta_labels: tuple[int, ...] | None = None

Optional eta label per mode; fixed across all harmonics.

reference: *Hamiltonian* (page 106) | None = None

Static Hamiltonian defining tracked state labels; None uses the time average.

observables: dict[str, dict[int, *FermionOperator* (page 105)]]

Named operator harmonics, allowing both positive and negative Fourier indices.

metadata: dict

Coordinate declarations, drive settings, sampling diagnostics and optional user entries.

property period

Drive period 2*pi/omega in inverse Hamiltonian-energy units.

property max_harmonic

Largest stored nonnegative Hamiltonian harmonic index.

harmonic(*k*)

H_k for any integer k, with H_{-k}=H_k^dagger.

operator_at(*t*)

H(t) as a FermionOperator.

__init__(*omega: float, harmonics: dict[int, ~qdjj_solver.common.algebra.FermionOperator], nimp: int, spins: tuple[int, ...], eta_labels: tuple[int, ...] | None = None, reference: ~qdjj_solver.common.problem.Hamiltonian | None = None, observables: dict[str, dict[int, ~qdjj_solver.common.algebra.FermionOperator]] = <factory>, metadata: dict = <factory>*) → None

time_average() → *Hamiltonian* (page 106)

The time average H_0 as a Hamiltonian, with the time-averaged observables.

check_sector(*sector*)

Require every harmonic to conserve the sector; a projection would silently drop terms.

coordinate_fingerprint()

Coordinate identity of the static reference; independent of Fourier truncation.

probe(*name*, *orbital=None*)

Harmonics {k: operator} of a named probe under this drive (see `common.probes`).

record()

JSON-ready periodic Hamiltonian; harmonic keys are strings and operators are term lists.

classmethod from_record(*record*)

Reconstruct a periodic-Hamiltonian record, including reference and observable harmonics.

fingerprint()

SHA256 of the full periodic record, used to check transition compatibility.

van_vleck(
order=2,
***,
tolerance=1e-14,
max_harmonic=None,
) → *Hamiltonian* (page 106)

High-frequency (van Vleck) effective Hamiltonian through O(omega^-order).

Commutators are formed in the full Fock space, before any QP truncation. The expansion requires omega to exceed the many-body spectral width; its cost grows with products of term counts. Observables are copied from the time average without the O(1/omega) kick-operator correction.

classmethod from_parameters(
builder,
base: Mapping,
drives,
omega,
***,

samples=64,
coefficient_tolerance=1e-14,
sampling_tolerance=1e-9,
verify=True,
observables=True,
) → *PeriodicHamiltonian* (page 111)

Harmonics of a model constructor whose keyword arguments are driven periodically.

`base` holds the undriven keywords; each *`ParameterDrive`* (page 110) displaces one of them. `builder` is evaluated at `samples` equally spaced times of one period, and the operator coefficients are Fourier transformed. With `verify=True` the Fourier series is compared with the model at the midpoints between samples and must agree within `sampling_tolerance`. The undriven model `builder(**base)` labels the dressed states. For the reference junction use, e.g., `builder=reference_model, base=dict(bath=bath, phi=.8, symmetry=False, compress=False)`.

classmethod from_callable(
function,
omega,
*,
samples=64,
reference=None,
assume_fixed_coordinates=False,
coefficient_tolerance=1e-14,
sampling_tolerance=1e-9,
verify=True,
observables=True,
) → *PeriodicHamiltonian* (page 111)

Harmonics of an arbitrary periodic `function(t) -> Hamiltonian`.

Without `reference` the time average labels the dressed states. The sampled Hamiltonians must declare their canonical coordinates in `metadata['coordinates']`, and these must not change during the period. `assume_fixed_coordinates=True` accepts Hamiltonians without this declaration; only their mode labels are then compared.

qdjj_solver.**PROBES**

Built-in immutable sequence.

If no argument is given, the constructor returns an empty tuple. If iterable is specified the tuple is initialized from iterable's items.

If the argument is a tuple, the return value is the same object.

class qdjj_solver.**FloquetTransitions**

Probe matrix elements X^(q) between two sets of Floquet states of one drive.

`elements[i, a, b]` is X^(q) for `photons[i]`, bra state `bra_indices[a]` (final) and ket state `ket_indices[b]` (initial) of the two FloquetStates. `total[b]` is `sum_m ||(X u_b)_m||^2`, the right-hand side of the sum rule `sum_a sum_q |X^(q)_ab|^2 = total[b]`, which holds for a complete set of bra states.

omega: float

Drive frequency.

photons: ndarray

Photon numbers q of the stored harmonics.

elements: ndarray

`elements[i, a, b]` is X^(q) for `photons[i]`, final state a and initial state b.

bra_quasienergies: ndarray

Quasienergies of the final (bra) states.

ket_quasienergies: ndarray

Quasienergies of the initial (ket) states.

bra_labels: ndarray

Static labels of the final states.

ket_labels: ndarray

Static labels of the initial states.

bra_indices: ndarray

Indices of the final states in their `FloquetStates`.

__init__(
omega: float,
photons: ndarray,
elements: ndarray,
bra_quasienergies: ndarray,
ket_quasienergies: ndarray,
bra_labels: ndarray,
ket_labels: ndarray,
bra_indices: ndarray,
ket_indices: ndarray,
total: ndarray,
same_states: bool,
complete: bool,
default_initial: int,
probe: str,
metadata: dict = <*factory*>,
) → None

ket_indices: ndarray

Indices of the initial states in their `FloquetStates`.

total: ndarray

Time-averaged squared norm of X(t) applied to each initial state: the right-hand side of the sum rule.

same_states: bool

True when bra and ket come from the same `FloquetStates` object.

complete: bool

True when the bra states form the complete set of their representation.

default_initial: int

Initial state used by `lines()` without arguments: the dressed ground state of the ket sector, `dressed_state(0)`, or the first ket state if that state was not selected.

probe: str

Name of the probe.

metadata: dict

Probe harmonics, sectors, methods and the matrix-element convention.

property frequencies

Line frequencies eps_a - eps_b + q Omega, with the shape of `elements`.

property strengths

Line strengths `|X^(q)_ab|^2`, with the shape of `elements`.

property captured

Total transition strength from each initial state to the computed final (bra) states.

property completeness

`captured/total`: the fraction of the sum rule exhausted by the computed bra states.

element(*q*)

The matrix X^(q), shape (bra states, ket states); zero if q is not stored.

lines(
initial=None,
weights=None,
***,
merge_degenerate=True,
elastic=True,
threshold=None,
tolerance=1e-8,
)

Spectral lines originating from the specified initial states (indices of the ket result).

Default: `default_initial` with unit weight. Merging distributes each initial weight uniformly over its degenerate group and sums the strengths of degenerate final states, which makes the lines independent of the basis chosen within degenerate multiplets. Terms between states of one group at q=0 are expectation values, not transitions, and are dropped. Lines weaker than `threshold` (default: 1e-12 of the strongest line, i.e. numerical zeros) are omitted.

spectrum(
frequencies,
width,
***,
shape='lorentzian',
kind='signed',
***line_options*,
)

Broadened spectrum of `lines(**line_options)`; see `SpectralLines.spectrum`.

class qdjj_solver.**SpectralLines**

Discrete lines omega_l with strengths s_l, weighted by the initial populations.

`final` and `initial` list the state indices merged into each line (degenerate groups), `photons` the photon number q, and `elastic` is true for lines between states of one group at q != 0 (the central line of a Mollow triplet).

frequencies: ndarray

Sorted line frequencies; positive values correspond to absorption of a probe quantum.

strengths: ndarray

Line strengths, weighted by the initial populations.

final: list

Final-state indices merged into each line (a degenerate group).

initial: list

Initial-state indices that contribute to each line.

photons: ndarray

Photon number q of each line.

elastic: ndarray

True for elastic lines, between states of one degenerate group at q != 0.

metadata: dict

Probe, initial states, weights and merging options.

spectrum(*frequencies*, *width*, *, *shape='lorentzian'*, *kind='signed'*)

Broadened spectrum S(omega) = sum_l s_l L(omega - omega_l) with a unit-area line shape L.

`kind="signed"` returns S(omega); `kind="rate"` returns S(omega)+S(-omega), proportional to the golden-rule transition rate out of the initial states at probe frequency omega>0; `kind="net"` returns S(omega)-S(-omega), absorption minus stimulated emission. `width` is the half width at half maximum.

record()

JSON-ready line positions, strengths, contributing state indices and metadata.

__init__(
frequencies: ndarray,
strengths: ndarray,
final: list,
initial: list,
photons: ndarray,
elastic: ndarray,
metadata: dict = <*factory*>,
) → None

qdjj_solver.**line_shape**(*x*, *width*, *shape='lorentzian'*)

Unit-area line shape; `width` is the half width at half maximum.

qdjj_solver.**orbital_density_matrix**(
result,
orbital=0,
*,
state=0,
weights=None,
physical=True,
)

Physical impurity-orbital RDM for QP or DMRG states.

The basis is [empty, up, down, up-down]. state is a zero-based eigenstate index; state=None returns all RDMs with shape (k,4,4). With state=None and nonnegative weights of length k, return their normalized incoherent mixture. Average complete degenerate multiplets for a basis-independent result.

physical=True uses the declared electron-to-canonical map, including verified impurity-only eta compression. For a manually constructed canonical-mode Hamiltonian without an electron map use physical=False explicitly. Composite projectors are formed before QP projection. Local odd/even coherences vanish in the solvers' fixed total parity sectors.

`qdjj_solver.`**`orbital_density_operators`**(*orbital=0*, *, *hamiltonian=None*, *physical=True*)

Operators O[a,b] whose expectations equal rho[a,b] in a parity sector.

Return a dictionary keyed by (row, column). The local basis is `[|0>, |up>, |down>, |up down>]`, with `|up down>=d_up^dagger d_down^dagger|0>`. Consequently `O[a,b]=|b><a|`, not `|a><b|`. Entries connecting different local parities vanish in fixed-total-parity states and are omitted.

Without a Hamiltonian the operators use physical electron mode indices (2*orbital, 2*orbital+1). With one, transform to its canonical coordinates. physical=False explicitly interprets those indices as canonical modes. Verified eta compression is safe for impurity-only operators: its impurity map has no discarded bath factors. Arbitrary compressed-bath RDMs are not covered by this function.

`qdjj_solver.`**`orbital_density_properties`**(*density*)

Occupations and coherent pair probabilities for one normalized 4x4 RDM.

`pair_amplitude=<d_down d_up>=rho[3,0]`. plus/minus refer to the fixed-gauge normalized states `(|0> +/- |up down>)/sqrt(2)`. phase_optimized_plus maximizes over the relative pair phase; it is different when pair_amplitude is complex. The input is validated, not renormalized or projected onto positive matrices.

J.2 QP backend `qdjj_solver.qp_solver`

Quasiparticle-expansion solver, projected operators and Floquet methods.

`class` `qdjj_solver.qp_solver.`**`FockBasis`**

Fock basis of one symmetry sector with at most `cutoff` bath quasiparticles.

The impurity modes are unrestricted; `dimension` is the number of basis states, checked against `max_dimension` before the basis is enumerated.

`__init__`(
nimp,
spins,
cutoff,
sector=DEFAULT_SECTOR,
eta_labels=None,
max_dimension=5_000_000,
)

`property bytes`

Native basis storage in bytes; excludes Hamiltonian matrices and solver workspaces.

`occupations`()

Copy occupations as rows of 64-bit words (least-significant word first).

qp_counts()

Bath-QP count of each basis ket, in exactly the same row order as `occupations`.

qdjj_solver.qp_solver.**dimension**(*nimp*, *spins*, *cutoff*, *sector*, *eta_labels=None*)

Exact integer dimension by counting occupations, without forming kets.

class qdjj_solver.qp_solver.**Eigenstates**

Lowest eigenstates of a Hamiltonian in one sector, from the QP solver.

backend: ClassVar[str] = 'qp'

energies: ndarray

Eigenenergies, shape (k,), in the units of the Hamiltonian.

vectors: ndarray

Eigenvectors in the Fock basis, shape (D, k); state i is the column `vectors[:, i]`.

residuals: ndarray

Projected residual norms, shape (k,), in energy units.

basis: *FockBasis* (page 117)

The `FockBasis` that defines the row order of `vectors` and the canonical occupations.

qp_weights: ndarray

Shape (k, q+1); `qp_weights[i, n]` is the probability of n bath quasiparticles in state i.

observables: dict[str, ndarray]

Expectation values of the requested named model observables, each of shape (k,); all are measured by default.

__init__(
energies: ndarray,
vectors: ndarray,
residuals: ndarray,
basis: FockBasis *(page 117)*,
qp_weights: ndarray,
observables: dict[str, ndarray],
timings: dict[str, float],
metadata: dict,
hamiltonian: Hamiltonian *(page 106)* | *None = None*,
) → None

timings: dict[str, float]

Elapsed times of the calculation stages, in seconds.

metadata: dict

Numerical settings, sector, Hilbert-space restrictions and convergence diagnostics.

hamiltonian: *Hamiltonian* (page 106) | None = None

The `Hamiltonian` that was solved.

matrix_elements(*operator*, *ket=None*, *, *orbital=None*)

All `<self_i|operator|ket_j>` in common canonical coordinates.

`operator` is a FermionOperator in canonical coordinates or a probe name (`hamiltonian.probe`). The ket may belong to another sector or QP cutoff. Eigenvector phases are arbitrary: compare `|elements|^2` or complete multiplets.

overlaps(*other=None*)

All `<self_i|other_j>`; the states may use different QP cutoffs of one sector.

class qdjj_solver.qp_solver.**ProjectedOperator**

A `FermionOperator` acting within a QP Fock basis, or from it into another compatible basis.

Components outside the retained basis are removed by the QP projection. `expectation` and `transition` evaluate matrix elements; `action_between` and `sparse_matrix` provide the operator action and matrix representation.

__init__(*operator*, *basis*)

action(*vector*)

The projected operator applied to a vector of the basis.

property bytes

Storage owned by the native operator, in bytes, excluding its shared basis.

linear_operator(*dtype=None*)

The action as a SciPy `LinearOperator`, without storing a matrix.

sparse_matrix(*max_nnz=100_000_000*)

The operator as a sparse CSR matrix with at most `max_nnz` stored entries.

expectation(*bra*, *ket=None*)

`<bra|O|ket>` within the basis; `ket` defaults to `bra`.

diagonal()

Return diagonal matrix elements without constructing the sparse matrix or diagonalizing it.

action_between(*ket*, *destination_basis*)

Apply the operator to `ket` and express the result in another basis (sector or QP cutoff) with the same one-particle coordinates.

transition(*bra*, *bra_basis*, *ket*)

`<bra|O|ket>` between sectors in the *same* one-particle coordinates.

The bra and ket may have different parity, S_z, eta and QP cutoff. The supplied operator must already be expressed in these coordinates.

class qdjj_solver.qp_solver.**SolverOptions**

Numerical settings of the QP solver; energies and tolerances share the Hamiltonian's units.

method: str = 'auto'

Eigensolver: `"auto"`, `"dense"`, `"sparse"`, `"matrix-free"`, `"schur"` or `"block"`. `"auto"` diagonalizes densely up to dimension 128, uses sparse ARPACK up to 400,000 and matrix-free ARPACK above. `"block"` uses LOBPCG for several eigenstates, including degenerate multiplets.

eigenpairs: int = 1

Number of lowest eigenstates computed in the selected sector.

tolerance: float = 1e-11

Relative ARPACK eigenvalue tolerance, for the internally shifted operator H - sigma with sigma below the smallest diagonal element by an offset of order one; for `"schur"` the root-search tolerance in energy units. For `"block"`, the residual tolerance in energy units, capped at `residual_tolerance/10`. Unused by dense diagonalization.

residual_tolerance: float = 2e-09

Largest accepted projected residual norm, in energy units.

maxiter: int = 10000

Maximum number of ARPACK or block-LOBPCG iterations.

ncv: int = 40

Requested Krylov-space dimension; increased as needed for the number of eigenstates and limited by the sector dimension.

max_dimension: int = 5000000

Largest allowed number of Fock basis states, checked before the basis is enumerated.

max_nnz: int = 100000000

Largest number of stored sparse-matrix entries (dense, sparse and Schur methods); the remaining `max_memory_gib` budget can lower it.

max_memory_gib: float = 24.0

Limit on the estimated basis and matrix storage, in GiB; the total memory use can be larger.

threads: int = 1

Maximum number of threads used by the numerical libraries.

seed: int = 1729

Seed of the random starting vector of the iterative eigensolvers.

__init__(
method: str = 'auto',
eigenpairs: int = 1,
tolerance: float = 1e-11,
residual_tolerance: float = 2e-09,
maxiter: int = 10000,
ncv: int = 40,
max_dimension: int = 5000000,
max_nnz: int = 100000000,
max_memory_gib: float = 24.0,
threads: int = 1,
seed: int = 1729,
) → None

qdjj_solver.qp_solver.**solve**(
hamiltonian: Hamiltonian *(page 106)*,
cutoff: int | None = None,
sector=DEFAULT_SECTOR,
options=DEFAULT_OPTIONS,
initial=None,
*,
context=None,

observables=None,
_prepared=None,
) → *Eigenstates* (page 118)

Lowest Ritz states in one sector. `cutoff=None` retains the complete finite bath.

Tolerances and residuals use the same energy units as the Hamiltonian, and the returned energies and residuals refer to H as given. The ARPACK eigensolvers internally work with H - sigma (sigma is recorded as `metadata["spectral_shift"]`) and check their results against the diagonal (Ky Fan) bound.

Use `QPWorkspace.solve` for prepared execution and `context` for cooperative progress and cancellation. `observables=None` measures all named operators; an iterable of names selects a subset, and an empty iterable skips measurements.

class qdjj_solver.qp_solver.**FloquetOptions**

Floquet settings; energies and tolerances share the Hamiltonian's units.

`floquet_cutoff` is M (blocks m=-M..M). `basis='eigen'` truncates the many-body space to the `basis_states` lowest static eigenstates of the reference Hamiltonian. The lowest `states` static states are tracked through the drive and label the dressed states.

floquet_cutoff: int = 4

Floquet cutoff M: the Sambe matrix retains the harmonic blocks m = -M, …, M.

basis: str = 'fock'

`"fock"` uses the full QP Fock space of the sector, `"eigen"` the `basis_states` lowest static eigenstates.

states: int = 4

Number of lowest static states tracked through the drive; they label the dressed states.

basis_states: int = 32

Size K of the static eigenbasis for `basis="eigen"`; choose K at a gap of the static spectrum, retaining complete degenerate multiplets.

selection: str = 'tracked'

`"tracked"` returns the continuations of the tracked static states, `"all"` every distinct Floquet state of the truncated Sambe matrix (dense solver or explicit `targets`).

branch: str = 'auto'

Replica that defines the reported quasienergy: `"label"` continues the assigned static state, `"centroid"` is the replica whose harmonic centroid lies in [-1/2, 1/2); `"auto"` uses `"label"` for `selection="tracked"` and `"centroid"` for `selection="all"`.

method: str = 'auto'

Sambe eigensolver: `"dense"`, `"shift-invert"` or `"auto"` (dense up to `dense_dimension`).

dense_dimension: int = 2000

Largest dimension diagonalized densely: the Sambe matrix for `method="auto"`, the static solution and the propagator.

targets: tuple[float, ...] | None = None

Explicit shift-invert target quasienergies; by default the static energies of the tracked states.

eigenpairs_per_target: int = 8

Initial number of shift-invert eigenpairs per target; doubled for a target until its tracked state is assigned a distinct Floquet state with at least `tracking_threshold` weight.

max_eigenpairs_per_target: int = 64

Upper limit on `eigenpairs_per_target` during successive doubling.

target_shift: float = 1e-07

Offset added to the default shift-invert targets to separate them from the reference eigenvalues.

tracking_threshold: float = 0.25

Minimum time-averaged weight of a tracked static state in its assigned Floquet state.

steps: int = 256

Number of time steps per period of the one-period propagator; doubled for the reported result when `step_check=True`.

step_check: bool = True

Repeat the propagation with twice as many steps and estimate the time-step error from the change of the quasienergies; the finer result is reported.

step_tolerance: float | None = None

If set, raise an error when the estimated propagator step error exceeds it, in energy units; requires `step_check=True`.

zone_center: float = 0.0

Centre c of the zone [c - omega/2, c + omega/2) of the `folded` quasienergies.

drop_scalar_harmonics: bool = False

Remove the c-number parts of the harmonics H_k with k != 0; the quasienergies are unchanged, but the harmonic weights of the modes are redistributed.

tolerance: float = 1e-11

Eigensolver tolerance of the static solution and of the shift-invert iterations.

residual_tolerance: float = 2e-09

Largest accepted window residual of the Sambe and static solutions, in energy units.

maxiter: int = 10000

Maximum number of iterations of the static and shift-invert eigensolvers.

max_dimension: int = 5000000

Largest allowed Fock-basis and Sambe dimension.

max_nnz: int = 100000000

Largest number of stored sparse-matrix entries.

max_memory_gib: float = 24.0

Limit on the estimated storage of dense Sambe matrices, sparse LU factors, propagators and probe images, in GiB.

threads: int = 1

Maximum number of threads used by the numerical libraries.

seed: int = 1729

Seed of the random starting vectors of the iterative eigensolvers.

__init__(
floquet_cutoff: int = 4,
basis: str = 'fock',
states: int = 4,
basis_states: int = 32,
selection: str = 'tracked',
branch: str = 'auto',
method: str = 'auto',
dense_dimension: int = 2000,
targets: tuple[float, ...] | None = None,
eigenpairs_per_target: int = 8,
max_eigenpairs_per_target: int = 64,
target_shift: float = 1e-07,
tracking_threshold: float = 0.25,
steps: int = 256,
step_check: bool = True,
step_tolerance: float | None = None,
zone_center: float = 0.0,
drop_scalar_harmonics: bool = False,
tolerance: float = 1e-11,
residual_tolerance: float = 2e-09,
maxiter: int = 10000,
max_dimension: int = 5000000,
max_nnz: int = 100000000,
max_memory_gib: float = 24.0,
threads: int = 1,
seed: int = 1729,
) → None

class qdjj_solver.qp_solver.**FloquetStates**

Floquet states psi(t)=exp(-i eps t) sum_m exp(i m omega t) u_m.

`modes[i, :, n]` holds u_m of state n for m = `harmonic_indices[i]`, in the Fock basis or in the static eigenbasis (`representation='static'`). The stored window `|m| <= M'` can exceed the Floquet cutoff M, because the branch choice shifts the harmonic index. Quasienergies belong to the chosen branch; `mean_energies` (eps-omega<m>) are branch invariant. `static_weights[n, a]` is the time-averaged weight of static state a.

kind: ClassVar[str] = 'floquet'

method: str

Solver that produced the states: `"sambe"`, `"propagator"` or `"rwa"`.

representation: str

`"fock"` or `"static"`: the basis of `modes`.

omega: float

Drive frequency.

quasienergies: ndarray

Quasienergies, shape (n,), on the chosen branch.

folded: ndarray

Quasienergies folded into the zone around `zone_center`, shape (n,).

mean_energies: ndarray

Replica-independent time-averaged energies eps - omega <m>, shape (n,).

harmonic_indices: ndarray

Harmonic indices m of the stored modes, shape (2M'+1,).

modes: ndarray

Mode harmonics, shape (2M'+1, d, n): u_m of state `j` is `modes[i, :, j]` with `harmonic_indices[i] == m`, in the basis given by `representation`.

harmonic_weights: ndarray

Harmonic weights, the squared norms of u_m, shape (n, 2M'+1).

centroids: ndarray

Harmonic centroids <m>, shape (n,).

boundary_weights: ndarray

Weight in the outermost harmonic blocks m = -M and M (Sambe) or outside the window -M <= m <= M (propagator); zero for the rotating-wave model; shape (n,).

static_weights: ndarray

Time-averaged weights of the static eigenstates a, the sum over m of the squared overlaps of a with u_m; shape (n, s), with s = `states` for the Fock basis and s = K for the static basis.

labels: ndarray

Index of the static state assigned to each Floquet state, shape (n,).

residuals: ndarray

Residual bound, shape (n,): the Sambe window residual combined with the leakage into the omitted harmonics, or the eigendecomposition residual of U(T) or of the rotating-wave matrix.

basis: *FockBasis* (page 117)

The QP `FockBasis` of the sector.

static: *Eigenstates* (page 118)

Static `Eigenstates` of the reference Hamiltonian `periodic.reference`.

periodic: *PeriodicHamiltonian* (page 111)

The `PeriodicHamiltonian` that was solved.

fock_residuals: ndarray | None = None

Fock-space residuals of static-basis results, shape (n,); `None` otherwise.

effective_matrix: ndarray | None = None

Rotating-wave matrix of `rwa_model` results; `None` otherwise.

timings: dict

Elapsed times of the calculation stages, in seconds.

__init__(
method: str,
representation: str,

omega: float,
quasienergies: ndarray,
folded: ndarray,
mean_energies: ndarray,
harmonic_indices: ndarray,
modes: ndarray,
harmonic_weights: ndarray,
centroids: ndarray,
boundary_weights: ndarray,
static_weights: ndarray,
labels: ndarray,
residuals: ndarray,
basis: FockBasis *(page 117)*,
static: Eigenstates *(page 118)*,
periodic: PeriodicHamiltonian *(page 111)*,
fock_residuals: ndarray | None = None,
effective_matrix: ndarray | None = None,
timings: dict = <factory>,
metadata: dict = <factory>,
) → None

metadata: dict

Settings and diagnostics, for example window residuals, leakage and step errors.

fock_modes()

Mode harmonics u_m in the QP Fock basis, shape (2M'+1, Fock dimension, n).

mode_at(*t*)

Periodic part u(t)=sum_m exp(i m omega t) u_m in the stored representation.

state_at(*t*)

Floquet state psi(t)=exp(-i eps t) u(t) in the stored representation.

dressed_state(*label=0*)

Index of the Floquet state assigned to the static state `label`.

If no state, or more than one, carries this label, the state with the largest weight on the static state `label` is returned.

matrix_elements(
probe,
ket=None,
*,
orbital=None,
ket_states=None,
bra_states=None,
max_memory_gib=None,
)

Probe matrix elements X^(q) from the ket states (initial) to these states (final).

`probe` is a probe name (`PeriodicHamiltonian.probe`), a FermionOperator (treated as time independent) or a dict {k: FermionOperator} of harmonics; `orbital` selects the impurity orbital of a named probe. `ket` defaults to these states and may belong to another sector of the

same drive. `ket_states` and `bra_states` restrict the state indices, and `max_memory_gib` limits the storage of the probe images X(t)|u(t)>.

qdjj_solver.qp_solver.**floquet_propagator**(
periodic,
cutoff=None,
sector=DEFAULT_SECTOR,
options=None,
static=None,
) → *FloquetStates* (page 123)

Floquet states from the eigenphases of the one-period propagator U(T).

The trigonometric polynomial H(t) is integrated with the fourth-order commutator-free Magnus scheme, without a Floquet cutoff. The harmonics of the modes are then truncated to `|m| <= M`, as in the Sambe results, and `boundary_weights` report the weight outside that window.

qdjj_solver.qp_solver.**rwa_model**(
periodic,
states,
photons,
cutoff=None,
sector=DEFAULT_SECTOR,
options=None,
static=None,
) → *FloquetStates* (page 123)

Few-level rotating-wave model: the Sambe matrix restricted to `|a_i, n_i>`.

Q_ij=(H_{n_i-n_j})_{a_i a_j}+n_i omega delta_ij, in static eigenstates of the reference Hamiltonian. Choose photon numbers so that E_a+n_a omega are nearly degenerate; at resonance the splitting is `2|(H_{n_a-n_b})_{ab}|`.

qdjj_solver.qp_solver.**solve_floquet**(
periodic,
cutoff=None,
sector=DEFAULT_SECTOR,
options=None,
static=None,
) → *FloquetStates* (page 123)

Floquet states from the Sambe matrix truncated at the Floquet cutoff, `|m| <= M`.

`residuals` combine the window residual with the leakage of H into the omitted harmonics; since the infinite Sambe operator is self-adjoint, an exact quasienergy lies within that distance. `basis='eigen'` also reports `fock_residuals`, the same bound in the full Fock space of the QP cutoff.

qdjj_solver.qp_solver.**floquet_matrix_elements**(
bra,
probe,
ket=None,
*,
orbital=None,
ket_states=None,
bra_states=None,
max_memory_gib=None,
) → *FloquetTransitions* (page 113)

Probe matrix elements X^(q) from the ket states (initial) to the bra states (final) of one drive.

See `FloquetStates.matrix_elements` for the arguments; `max_memory_gib` defaults to the setting of the bra calculation.

class qdjj_solver.qp_solver.**QPWorkspace**

Reuse QP preparation with least-recently-used eviction and bounded storage.

Keys distinguish enumeration structure from physical canonical coordinates. Changing coefficients can reuse a basis, but only identical coordinates authorize a warm start. Returned arrays are read-only snapshots.

`max_memory_gib` bounds accounted live data plus estimated working storage; `max_entries` limits each cache. `count_limit` is the largest dimension for an independent LDL state count. `retained`, when supplied, is a callable returning additional numerical data to include in the memory accounting.

__init__(
max_memory_gib=0.5,
max_entries=16,
count_limit=512,
retained=None,
)

property live_bytes

Accounted bytes of caches and live results, including caller-retained arrays.

reserve(*estimated_bytes*, *context=None*)

Admit temporary model/operator construction alongside retained data.

prepare(*h*, *cutoff*, *sector*, *options=DEFAULT_OPTIONS*, *context=None*)

Check resource limits and return the cache key and prepared basis entry for a problem.

solve(
h,
cutoff=None,
sector=DEFAULT_SECTOR,
options=DEFAULT_OPTIONS,
initial=None,
*,
context=None,
observables=None,
)

Return validated QP eigenstates, reusing compatible completed preparation.

Arguments follow `qp_solver.solve`. A compatible prior result supplies the starting vector or block when `initial` is omitted for an explicitly iterative method. `observables=None` measures all named model operators. Returned numerical arrays are read-only; reuse diagnostics are in `metadata`.

count_below(*h*, *cutoff*, *sector*, *edge*, *margin=1e-8*, *, *context=None*)

Count states below edge +/- margin; refuse oversized/uncertain certificates.

project_operator(*h*, *cutoff*, *sector*, *operator*, *, *context=None*)

Cache a canonical-coordinate transition operator in an already prepared basis.

clear()

Release cached preparation and warm starts; caller-owned results remain valid.

class qdjj_solver.qp_solver.**ExecutionContext**

Cooperative QP progress and cancellation callbacks.

`progress(event)` receives a dictionary with the stage, elapsed seconds and stage-specific details. Repeated events in one stage are separated by at least `interval` seconds. `cancelled()` is checked at every checkpoint and a true result raises `CalculationCancelled`. Callback exceptions propagate to the caller.

__init__(*progress=None*, *cancelled=None*, *interval=0.1*)

checkpoint(*stage*, ***details*)

Check cancellation and emit a progress event when the stage or interval changes.

class qdjj_solver.qp_solver.**CalculationCancelled**

A requested cancellation, distinct from a numerical convergence failure.

J.3 DMRG backend `qdjj_solver.dmrg_solver`

Matrix-product-state solver (requires the `dmrg` extra).

class qdjj_solver.dmrg_solver.**PreparedModel**

A Hamiltonian mapped onto MPS sites: mode order, grouping, conserved charges and operators.

hamiltonian: *Hamiltonian* (page 106)

Original physical finite Hamiltonian, before MPS layout selection.

sector: *Sector* (page 108)

Selected total symmetry quantum numbers.

order: tuple[int, ...]

Permutation of zero-based canonical mode indices along the MPS.

groups: tuple[tuple[int, ...], ...]

Canonical modes grouped into each MPS site, one or two per site.

sites: list

TeNPy local fermionic sites with the selected conserved charges.

lattice: TrivialLattice

Finite open TeNPy lattice of the grouped sites; a computational ordering.

mode_operators: dict

Canonical mode mapped to its site index and local annihilation/creation names.

target_charge: tuple[int, ...]

Total TeNPy charge labels encoding the selected sector.

coordinate_id: str

Fingerprint of physical coordinates, ordering, grouping and conserved-charge types.

term(*key*, *, *lattice=False*)

Map a signed one-based monomial to TeNPy operator/site pairs, preserving product order.

__init__(
hamiltonian: Hamiltonian *(page 106)*,
sector: Sector *(page 108)*,
order: tuple[int, ...],
groups: tuple[tuple[int, ...], ...],
sites: list,
lattice: TrivialLattice,
mode_operators: dict,
target_charge: tuple[int, ...],
coordinate_id: str,
) → None

explicit_jw_term(*key*)

Tensor-product operators, including the left boundary string for odd terms.

compile(*operator*, *, *merge_parallel=True*)

Compile an even, charge-preserving operator without coefficient truncation.

model(*operator*)

TeNPy MPOModel for an even, charge-preserving operator in this layout.

matrix_element(*operator*, *bra*, *ket*)

Exact contraction of a general polynomial, including parity-changing terms.

conserves_charges(*operator*)

Whether every term preserves the selected charges; validates declared mode indices.

matrix_elements(*operator*, *bras*, *kets*)

Complex transition array with bras as rows and kets as columns, including odd operators.

onsite_costs()

Diagonal local energies used only to select distinct product-state seeds.

qdjj_solver.dmrg_solver.**prepare**(*hamiltonian*, *sector=DEFAULT_SECTOR*, *, *group_size=2*, *mode_order=None*)

Choose an MPS layout without modifying physical coordinates or removing modes.

class qdjj_solver.dmrg_solver.**Eigenstates**

Lowest eigenstates of a Hamiltonian in one sector, as matrix-product states.

__init__(
energies: ndarray,
states: list,
residuals: ndarray | None,
observables: dict[str, ndarray],
timings: dict[str, float],
metadata: dict,
hamiltonian: Hamiltonian *(page 106)*,
prepared: object,
) → None

backend: ClassVar[str] = 'dmrg'

energies: ndarray

Eigenenergies, shape (k,), in the units of the Hamiltonian.

states: list

The k eigenstates as TeNPy MPS.

residuals: ndarray | None

Physical residual norms, shape (k,), in energy units; `None` if residuals were disabled.

observables: dict[str, ndarray]

Expectation values of the measured observables, each of shape (k,).

timings: dict[str, float]

Elapsed times of the calculation stages, in seconds.

metadata: dict

Numerical settings, sector, convergence diagnostics and sweep statistics.

hamiltonian: *Hamiltonian* (page 106)

The `Hamiltonian` that was solved.

prepared: object

Canonical basis, conserved quantum numbers, and MPS mode ordering and grouping.

overlaps(*other=None*)

Overlap matrix with this result's states as bras and `other` as kets; by default, return the Gram matrix of this result.

matrix_elements(*operator*, *ket=None*)

All `<self_i|operator|ket_j>` in common prepared coordinates.

class qdjj_solver.dmrg_solver.**SolverOptions**

Numerical settings of the DMRG solver; energies and tolerances share the Hamiltonian's units.

eigenpairs: int = 1

Number of eigenstates to calculate, optimized sequentially.

chi_max: int = 128

Maximum MPS bond dimension.

max_sweeps: int = 60

Maximum number of sweeps per initial-state trial and eigenstate.

min_sweeps: int = 10

Minimum number of sweeps per initial-state trial and eigenstate.

energy_tolerance: float = 1e-11

Absolute energy tolerance, in energy units: TeNPy's sweep-to-sweep energy criterion and the stability test of the last sweep energies; the local Lanczos tolerance `E_tol` is derived from it.

entropy_tolerance: float = 1e-08

Dimensionless entropy-change threshold of TeNPy's stopping test.

residual_tolerance: float = 1e-07

Maximum accepted physical residual norm for convergence of the finite problem, in energy units.

orthogonality_tolerance: float = 1e-08

Tolerance on deviations of the overlap matrix from the identity and on overlaps with earlier eigenstates.

svd_min: float = 1e-14

Smallest retained singular value during optimization; may be zero.

lanczos_maxiter: int = 100

Maximum number of local Lanczos iterations.

seed_trials: int = 4

Number of initial-state trials per eigenstate, including randomized states.

seed: int = 1729

Seed of the local random transformations of initial states.

seed_randomization_steps: int = 4

Layers of local random unitary transformations for additional initial states; zero disables randomization.

excitation_operator: str | None = None

Name of a model observable that conserves the sector's quantum numbers; its fluctuation applied to the previous eigenstate supplies the first trial state for the next eigenstate.

mixer: bool = True

Enable TeNPy's optimization mixer.

mixer_amplitude: float = 1e-05

Initial mixer strength.

mixer_sweeps: int = 6

Number of sweeps after which the mixer is switched off.

group_size: int = 2

One or two canonical fermionic modes per MPS site.

mode_order: tuple[int, ...] | None = None

Permutation of all zero-based canonical mode indices; `None` keeps the model order.

observables: tuple[str, ...] | None = None

Observables to evaluate: `None` selects all named observables, a tuple selects specified names, and `()` selects none.

calculate_residuals: bool = True

Evaluate the residual norm of (H - E) psi for each state after optimization, without MPS compression.

require_convergence: bool = False

Raise an error when the finite-problem convergence checks fail.

__init__(
eigenpairs: int = 1,
chi_max: int = 128,
max_sweeps: int = 60,
min_sweeps: int = 10,
energy_tolerance: float = 1e-11,
entropy_tolerance: float = 1e-08,
residual_tolerance: float = 1e-07,
orthogonality_tolerance: float = 1e-08,
svd_min: float = 1e-14,
lanczos_maxiter: int = 100,
seed_trials: int = 4,
seed: int = 1729,
seed_randomization_steps: int = 4,
excitation_operator: str | None = None,
mixer: bool = True,
mixer_amplitude: float = 1e-05,
mixer_sweeps: int = 6,
group_size: int = 2,
mode_order: tuple[int, ...] | None = None,
observables: tuple[str, ...] | None = None,
calculate_residuals: bool = True,
require_convergence: bool = False,
threads: int = 1,
lanczos_probability_tolerance: float = 1e-14,
) → None

threads: int = 1

Maximum number of threads used by the numerical libraries.

lanczos_probability_tolerance: float = 1e-14

Local Lanczos probability-error threshold, passed to TeNPy as `P_tol`.

qdjj_solver.dmrg_solver.**solve**(
hamiltonian,
cutoff=None,
sector=DEFAULT_SECTOR,
options=DEFAULT_OPTIONS,
initial=None,
)

Lowest states in the complete finite-mode space defined by `hamiltonian`.

`cutoff` must be None: a QP cutoff and a Schmidt-rank limit are different approximations. `initial` may be an earlier Eigenstates in identical coordinates/layout for bond-dimension or parameter continuation.

Convergence here concerns the finite Hamiltonian, not bath discretization. A small residual identifies an eigenstate; independent seeds and root/bath ladders are still needed to establish that the desired roots were found.

J.4 Supported supplementary helpers

These qualified submodule functions support the workflows in this handbook. Legacy DMRG helpers use their own reference interfaces, with the signatures documented below.

`qdjj_solver.common.baths.`**`hybridization_g`**(*frequency*, *delta=1.*, *bandwidth=100.*)

Flat finite-band imaginary-frequency kernel, with inverse-energy units.

`frequency` may be a scalar or array of real Matsubara-frequency arguments; the result has the same shape. The gap and half-bandwidth use the same units.

`qdjj_solver.common.baths.`**`discrete_g`**(*bath*, *frequency*)

Imaginary-frequency kernel of a DiscreteBath, with the shape of `frequency`.

Uses its actual contact weights and density convention; compare with `hybridization_g` at the same physical gap and half-bandwidth.

`qdjj_solver.dmrg_solver.convergence.`**`converge_bond_dimension`**(
hamiltonian,
chi_values,
***,
sector=DEFAULT_SECTOR,
options=DEFAULT_OPTIONS,
energy_tolerance=1e-8,
observable_tolerance=1e-7,
stable_steps=2,
callback=None,
)

Run the specified sequence of increasing bond dimensions and retain the convergence diagnostics.

Requires convergence of the finite-problem residuals and sweep energies, together with stable energies and observables between successive bond dimensions. The assessment gives empirical convergence evidence for the retained states at fixed bath. Assess bath convergence by refinement and check low-energy completeness through additional state searches or an independent reference. `callback(result, entry)` can archive each completed step immediately.

`qdjj_solver.dmrg_solver.convergence.`**`compare_states`**(*previous*, *current*)

Compare eigenstates in identical coordinates using individual-state and subspace overlaps.

The assignment maximizes individual overlaps; singular values are invariant under rotations within a degenerate retained subspace. Individual-state comparisons use phase-invariant squared overlap magnitudes.

`qdjj_solver.common.io.`**`model_from_config`**(*config*, ***, *default_compress=True*)

Construct one shared physical model; no numerical backend is imported.

`qdjj_solver.common.io.`**`bath_from_config`**(*config*, ***, *path='bath'*)

Construct a reservoir from a version-1 bath object; `path` prefixes validation errors.

Supported kinds are cosh-grid, surrogate, chain-expansion and discrete. Omitted kind selects cosh-grid; required constructor arguments remain required.

`qdjj_solver.common.io.`**`validate_config`**(*config*, ***, *allow_backend=True*)

Validate the versioned envelope without importing either numerical backend.

qdjj_solver.common.io.**source_provenance**()

Hash the implementation actually imported, including an installed native core.

Relative module paths are portable between editable and wheel installations; no checkout location, user home directory, or Git working tree is recorded.

qdjj_solver.common.io.**json_value**(*value*)

Convert NumPy values and complex scalars/arrays to portable JSON values.

qdjj_solver.common.io.**complex_array**(*value*)

Read a real array or the explicit `{real: ..., imag: ...}` encoding.

qdjj_solver.common.io.**write_json**(*path*, *data*)

Write atomically, rejecting non-finite values before replacing a record.

qdjj_solver.common.floquet.**fold_quasienergies**(*energies*, *omega*, *center=0.*)

Map quasienergies into the zone [center-omega/2, center+omega/2).

qdjj_solver.common.floquet.**quasienergy_distance**(*a*, *b*, *omega*)

Distance between quasienergies modulo omega.

qdjj_solver.common.spectra.**degenerate_groups**(*values*, *omega=None*, *tolerance=1e-8*)

Assign common labels to (quasi)energies that agree within `tolerance` (modulo `omega`).

```
qdjj_solver.dmrg_solver.legacy.dmrg_reference(
    bath,
    u=2.,
    gamma=.4,
    phi=3 * np.pi / 5,
    rho_ws=0.,
    geometry='junction',
    twice_sz=1,
    chi=64,
    max_sweeps=60,
    energy_tolerance=1e-11,
    threads=1,
    calculate_variance=True,
    calculate_current=True,
)
```

Unrestricted DMRG in QPs of the exactly diagonalized quadratic bath.

No QP-number cutoff is made. The bath vacuum is a product in these coordinates, which substantially reduces the required Schmidt rank for weakly hybridized superconducting impurities. The isolated BCS energy reference is retained, including the exact background-junction shift.

```
qdjj_solver.dmrg_solver.legacy.dmrg_chain_reference(
    bath,
    u=2.,
    gamma=.4,
    phi=3 * np.pi / 5,
    rho_ws=0.,
    geometry='junction',
    twice_sz=1,
    chi=128,
    max_sweeps=40,
```

energy_tolerance=1e-11,
threads=1,
calculate_variance=True,
)

Unrestricted finite-bath result, using optional TeNPy ([reference]).

This reference routine treats the symmetric single-dot model at half filling. The general multi-orbital QP solver does not depend on TeNPy.

qdjj_solver.dmrg_solver.legacy.**chain_coefficients**(*bath*)

Lanczos tridiagonalization starting at the normalized contact orbital.

Full reorthogonalization is used. The returned contact norm must multiply dot–lead hopping (and both contact norms multiply interlead hopping).

APPENDIX

K

CITATION AND METHOD REFERENCES

K.1 Software citation

Please identify the software version used. A ready-to-copy citation for the current version is:

> Teodor Iličin and Rok Žitko (2026). *qdjj_solver: QP and DMRG solvers for superconducting quantum impurities*, version 0.3.0. Zenodo. https://doi.org/10.5281/zenodo.23104978.

For BibLaTeX:

```
@software{qdjj_solver_030,
  author    = {Iličin, Teodor and Žitko, Rok},
  title     = {{qdjj\_solver}: {QP} and {DMRG} solvers for superconducting␣
↪quantum impurities},
  version   = {0.3.0},
  date      = {2026-10-26},
  publisher = {Zenodo},
  doi       = {10.5281/zenodo.23104978},
  url       = {https://doi.org/10.5281/zenodo.23104978}
}
```

A BibTeX citation is also available for import into a paper's bibliography. Its citation key is `qdjj_solver_030`.

These citations currently use the all-versions DOI. Replace it with the version-specific DOI once the 0.3.0 Zenodo archive is available. The earlier 0.2.0 archive retains its own DOI.

CITATION.cff contains the citation information in a format readable by reference-management tools. For modified or unreleased source code, also record its Git revision identifier (`git rev-parse HEAD`). Saved results include file checksums identifying the solver code actually used.

For reproducibility, cite the DOI of the version used in your calculation. The all-versions DOI identifies the software project across releases.

K.2 Which methods to cite

- **QP expansion:** cite the *QP variational method* (page 137) as well as the software version. State the QP cutoff, symmetry sector, finite bath, and canonical bath reference; numerics defines the projection and the required convergence checks.
- **Fitted surrogate reservoirs:** cite Baran, Frost, and Paaske when using `fit_surrogate`. Report the fitting mesh and criterion alongside the bath size.

- **Padé chain-expansion reservoirs:** cite Bobok, Frk, Pokorný and Žonda when using `chain_expansion`. Report the chain length and finite-/wide-band target.
- **DMRG:** cite the DMRG method references and TeNPy below when using this solver. Record the bond dimension, eigenstate-search settings, and convergence evidence.
- **Floquet calculations:** cite the *Floquet methods* (page 137) used. Report the drive, frequency, Floquet cutoff, many-body basis and the convergence evidence listed in *Floquet calculations* (page 47).

The references below describe those numerical components; the software version specifies the implementation used in a calculation.

K.3 QP variational method

- Teodor Iličin and Rok Žitko, *Quasiparticle-resolved variational theory of Andreev spin qubits*, manuscript. This is the ASQ method manuscript associated with the archived reference-junction calculations. No public DOI or arXiv identifier is recorded here.
- Teodor Iličin and Rok Žitko, *Effects of electron-electron interaction and spin-orbit coupling on Andreev pair qubits in quantum dot Josephson junctions*, SciPost Physics **20**, 147 (2026). DOI: 10.21468/SciPostPhys.20.5.147; arXiv:2512.23015. Section 3.3 and Appendix B extend the two-QP variational approach to a junction.
- Teodor Iličin and Rok Žitko, *Variational solution of the superconducting Anderson impurity model and the band-edge singularity phenomena*, SciPost Physics **19**, 006 (2025). DOI: 10.21468/SciPostPhys.19.1.006; arXiv:2503.18902. This is the earlier single-reservoir two-QP construction.

K.4 Method references

1. V. V. Baran, E. J. P. Frost, and J. Paaske, *Surrogate model solver for impurity-induced superconducting subgap states*, Physical Review B **108**, L220506 (2023). DOI: 10.1103/PhysRevB.108.L220506.
2. S. R. White, *Density matrix formulation for quantum renormalization groups*, Physical Review Letters **69**, 2863 (1992). DOI: 10.1103/PhysRevLett.69.2863.
3. U. Schollwöck, *The density-matrix renormalization group in the age of matrix product states*, Annals of Physics **326**, 96–192 (2011). DOI: 10.1016/j.aop.2010.09.012.
4. J. Hauschild and F. Pollmann, *Efficient numerical simulations with Tensor Networks: Tensor Network Python (TeNPy)*, SciPost Physics Lecture Notes **5** (2018). DOI: 10.21468/SciPostPhysLectNotes.5.
5. D. Bobok, L. Frk, V. Pokorný and M. Žonda, *Scalable effective models for superconducting nanostructures: Applications to double, triple, and quadruple quantum dots*, Physical Review B **112**, 205418 (2025). DOI: 10.1103/mxsl-fc96.

K.5 Floquet methods

1. J. H. Shirley, *Solution of the Schrödinger equation with a Hamiltonian periodic in time*, Physical Review **138**, B979 (1965). DOI: 10.1103/PhysRev.138.B979.
2. H. Sambe, *Steady states and quasienergies of a quantum-mechanical system in an oscillating field*, Physical Review A **7**, 2203 (1973). DOI: 10.1103/PhysRevA.7.2203.

3. A. Eckardt and E. Anisimovas, *High-frequency approximation for periodically driven quantum systems from a Floquet-space perspective*, New Journal of Physics **17**, 093039 (2015). DOI: 10.1088/1367-2630/17/9/093039.

4. M. Bukov, L. D'Alessio, and A. Polkovnikov, *Universal high-frequency behavior of periodically driven systems: from dynamical stabilization to Floquet engineering*, Advances in Physics **64**, 139 (2015). DOI: 10.1080/00018732.2015.1055918.

5. S. Blanes and P. C. Moan, *Fourth- and sixth-order commutator-free Magnus integrators for linear and non-linear dynamical systems*, Applied Numerical Mathematics **56**, 1519 (2006). DOI: 10.1016/j.apnum.2005.11.004.

6. A. Alvermann and H. Fehske, *High-order commutator-free exponential time-propagation of driven quantum systems*, Journal of Computational Physics **230**, 5930 (2011). DOI: 10.1016/j.jcp.2011.04.006.

TeNPy supplies MPS contractions and the optional DMRG engine. Consult its documentation for current citation guidance. NumPy, SciPy, pybind11, scikit-build-core, threadpoolctl, and h5py are separate dependencies with their own licenses. This repository's BSD license applies to its own implementation; dependencies are not relicensed or vendored.

INDEX

Symbols

__init__() (*qdjj_solver.DiscreteBath method*), 105
__init__() (*qdjj_solver.FermionOperator method*), 105
__init__() (*qdjj_solver.FloquetTransitions method*), 114
__init__() (*qdjj_solver.Hamiltonian method*), 106
__init__() (*qdjj_solver.Harmonic method*), 110
__init__() (*qdjj_solver.Impurity method*), 107
__init__() (*qdjj_solver.ParameterDrive method*), 111
__init__() (*qdjj_solver.PeriodicHamiltonian method*), 112
__init__() (*qdjj_solver.Sector method*), 108
__init__() (*qdjj_solver.SpectralLines method*), 116
__init__() (*qdjj_solver.dmrg_solver.Eigenstates method*), 129
__init__() (*qdjj_solver.dmrg_solver.PreparedModel method*), 129
__init__() (*qdjj_solver.dmrg_solver.SolverOptions method*), 131
__init__() (*qdjj_solver.qp_solver.Eigenstates method*), 118
__init__() (*qdjj_solver.qp_solver.ExecutionContext method*), 128
__init__() (*qdjj_solver.qp_solver.FloquetOptions method*), 123
__init__() (*qdjj_solver.qp_solver.FloquetStates method*), 124
__init__() (*qdjj_solver.qp_solver.FockBasis method*), 117
__init__() (*qdjj_solver.qp_solver.ProjectedOperator method*), 119
__init__() (*qdjj_solver.qp_solver.QPWorkspace method*), 127
__init__() (*qdjj_solver.qp_solver.SolverOptions method*), 120

A

action() (*qdjj_solver.qp_solver.ProjectedOperator method*), 119
action_between() (*qdjj_solver.qp_solver.ProjectedOperator method*), 119
amplitude, 66, 71
amplitude (*qdjj_solver.Harmonic attribute*), 110
anderson() (*qdjj_solver.Impurity class method*), 107
annihilate() (*in module qdjj_solver*), 108
assume_fixed_coordinates, 67

B

backend, 91, 96, 98
backend (*qdjj_solver.dmrg_solver.Eigenstates attribute*), 129
backend (*qdjj_solver.qp_solver.Eigenstates attribute*), 118
bandwidth, 63--65, 68
bandwidth (*qdjj_solver.DiscreteBath attribute*), 105
base, 66
basis (*qdjj_solver.qp_solver.Eigenstates attribute*), 118
basis (*qdjj_solver.qp_solver.FloquetOptions attribute*), 121
basis (*qdjj_solver.qp_solver.FloquetStates attribute*), 124
basis_bytes, 98
basis_states (*qdjj_solver.qp_solver.FloquetOptions attribute*), 121
bath, 61, 91
bath_convergence_checked, 100
bath_from_config() (*in module qdjj_solver.common.io*), 133
bath_modes, 98, 99
bath_modes (*qdjj_solver.Hamiltonian property*), 106
bath_reference, 62, 63, 94
baths, 62

boundary_weights (*qdjj_solver.qp_solver.FloquetStates attribute*), 124
bra_indices (*qdjj_solver.FloquetTransitions attribute*), 114
bra_labels (*qdjj_solver.FloquetTransitions attribute*), 114
bra_quasienergies (*qdjj_solver.FloquetTransitions attribute*), 114
branch (*qdjj_solver.qp_solver.FloquetOptions attribute*), 121
builder, 66
bytes (*qdjj_solver.qp_solver.FockBasis property*), 117
bytes (*qdjj_solver.qp_solver.ProjectedOperator property*), 119

C

calculate_residuals (*qdjj_solver.dmrg_solver.SolverOptions attribute*), 131
calculation, 97
CalculationCancelled (*class in qdjj_solver.qp_solver*), 128
captured (*qdjj_solver.FloquetTransitions property*), 115
centroids (*qdjj_solver.qp_solver.FloquetStates attribute*), 124
chain_coefficients() (*in module qdjj_solver.dmrg_solver.legacy*), 135
chain_expansion() (*in module qdjj_solver*), 108
check_sector() (*qdjj_solver.Hamiltonian method*), 106
check_sector() (*qdjj_solver.PeriodicHamiltonian method*), 112
checkpoint() (*qdjj_solver.qp_solver.ExecutionContext method*), 128
chi_actual, 101
chi_max (*qdjj_solver.dmrg_solver.SolverOptions attribute*), 130
cleaned() (*qdjj_solver.FermionOperator method*), 106
clear() (*qdjj_solver.qp_solver.QPWorkspace method*), 128
coefficient_tolerance, 62, 63, 67, 94
compare_states() (*in module qdjj_solver.dmrg_solver.convergence*), 133
compile() (*qdjj_solver.dmrg_solver.PreparedModel method*), 129
complete (*qdjj_solver.FloquetTransitions attribute*), 114
completeness (*qdjj_solver.FloquetTransitions property*), 115
complex_array() (*in module qdjj_solver.common.io*), 134
compress, 62
compress_pairs, 63, 94
conserves_charges() (*qdjj_solver.dmrg_solver.PreparedModel method*), 129
constant, 65
contact, 95
converge_bond_dimension() (*in module qdjj_solver.dmrg_solver.convergence*), 133
coordinate_fingerprint() (*qdjj_solver.Hamiltonian method*), 107
coordinate_fingerprint() (*qdjj_solver.PeriodicHamiltonian method*), 112
coordinate_id, 73, 99
coordinate_id (*qdjj_solver.dmrg_solver.PreparedModel attribute*), 128
cosh_grid() (*in module qdjj_solver*), 108
cosine() (*qdjj_solver.ParameterDrive class method*), 111
count_below() (*qdjj_solver.qp_solver.QPWorkspace method*), 127
create() (*in module qdjj_solver*), 109
cutoff, 92

D

dagger() (*qdjj_solver.FermionOperator method*), 105
default_initial (*qdjj_solver.FloquetTransitions attribute*), 114
degenerate_groups() (*in module qdjj_solver.common.spectra*), 134
delta, 63--65, 68
delta (*qdjj_solver.DiscreteBath attribute*), 105
dense_dimension (*qdjj_solver.qp_solver.FloquetOptions attribute*), 121
detuning, 62, 65
diagonal() (*qdjj_solver.qp_solver.ProjectedOperator method*), 119
dimension, 98
dimension() (*in module qdjj_solver.qp_solver*), 118
direct, 63, 93
direct_derivatives, 63, 94
discrete, 93
discrete_g() (*in module qdjj_solver.common.baths*), 133
DiscreteBath (*class in qdjj_solver*), 104
dmrg_chain_reference() (*in module qdjj_solver.dmrg_solver.legacy*), 134

dmrg_reference() (*in module qdjj_solver.dmrg_solver.legacy*), 134
dressed_state() (*qdjj_solver.qp_solver.FloquetStates method*), 125
drives, 66
drop_scalar_harmonics (*qdjj_solver.qp_solver.FloquetOptions attribute*), 122

E

effective_matrix (*qdjj_solver.qp_solver.FloquetStates attribute*), 124
eigenpairs (*qdjj_solver.dmrg_solver.SolverOptions attribute*), 130
eigenpairs (*qdjj_solver.qp_solver.SolverOptions attribute*), 119
eigenpairs_per_target (*qdjj_solver.qp_solver.FloquetOptions attribute*), 121
Eigenstates (*class in qdjj_solver.dmrg_solver*), 129
Eigenstates (*class in qdjj_solver.qp_solver*), 118
elastic (*qdjj_solver.SpectralLines attribute*), 116
element() (*qdjj_solver.FloquetTransitions method*), 115
elements (*qdjj_solver.FloquetTransitions attribute*), 114
energies, 96
energies (*qdjj_solver.DiscreteBath property*), 105
energies (*qdjj_solver.dmrg_solver.Eigenstates attribute*), 130
energies (*qdjj_solver.qp_solver.Eigenstates attribute*), 118
energy_reference, 99
energy_tolerance (*qdjj_solver.dmrg_solver.SolverOptions attribute*), 130
entropy_tolerance (*qdjj_solver.dmrg_solver.SolverOptions attribute*), 130
environment, 97
estimated_solver_bytes, 98
eta, 70
eta (*qdjj_solver.Sector attribute*), 108
eta_basis, 63, 93
eta_labels, 70, 72, 94
eta_labels (*qdjj_solver.Hamiltonian attribute*), 106
eta_labels (*qdjj_solver.PeriodicHamiltonian attribute*), 111
excitation_operator (*qdjj_solver.dmrg_solver.SolverOptions attribute*), 131
ExecutionContext (*class in qdjj_solver.qp_solver*), 128
expectation() (*qdjj_solver.qp_solver.ProjectedOperator method*), 119
explicit_jw_term() (*qdjj_solver.dmrg_solver.PreparedModel method*), 129

F

FermionOperator (*class in qdjj_solver*), 105
field, 62, 65
file, 102
final (*qdjj_solver.SpectralLines attribute*), 116
fingerprint() (*qdjj_solver.Hamiltonian method*), 107
fingerprint() (*qdjj_solver.PeriodicHamiltonian method*), 112
finite_problem_converged, 100
fit_surrogate() (*in module qdjj_solver*), 109
floquet_cutoff (*qdjj_solver.qp_solver.FloquetOptions attribute*), 121
floquet_matrix_elements() (*in module qdjj_solver.qp_solver*), 126
floquet_propagator() (*in module qdjj_solver.qp_solver*), 126
FloquetOptions (*class in qdjj_solver.qp_solver*), 121
FloquetStates (*class in qdjj_solver.qp_solver*), 123
FloquetTransitions (*class in qdjj_solver*), 113
fock_modes() (*qdjj_solver.qp_solver.FloquetStates method*), 125
fock_residuals (*qdjj_solver.qp_solver.FloquetStates attribute*), 124
FockBasis (*class in qdjj_solver.qp_solver*), 117
fold_quasienergies() (*in module qdjj_solver.common.floquet*), 134
folded (*qdjj_solver.qp_solver.FloquetStates attribute*), 124
format, 96
format_version, 91, 96, 101
frequencies (*qdjj_solver.FloquetTransitions property*), 115
frequencies (*qdjj_solver.SpectralLines attribute*), 115
frequency_cutoff, 64
frequency_min, 64
frequency_points, 64
from_callable() (*qdjj_solver.PeriodicHamiltonian class method*), 113
from_integrals() (*qdjj_solver.Impurity class method*), 107
from_parameters() (*qdjj_solver.*

PeriodicHamiltonian class method), 112
from_record() (*qdjj_solver.DiscreteBath class method*), 105
from_record() (*qdjj_solver.Hamiltonian class method*), 106
from_record() (*qdjj_solver.PeriodicHamiltonian class method*), 112
from_records() (*qdjj_solver.FermionOperator class method*), 106
function, 67

G

gamma, 61
geometry, 61
gram, 100
gram_error, 100
group_size (*qdjj_solver.dmrg_solver.SolverOptions attribute*), 131
groups, 72, 99
groups (*qdjj_solver.dmrg_solver.PreparedModel attribute*), 128

H

hamiltonian, 72, 97
Hamiltonian (*class in qdjj_solver*), 106
hamiltonian (*qdjj_solver.dmrg_solver.Eigenstates attribute*), 130
hamiltonian (*qdjj_solver.dmrg_solver.PreparedModel attribute*), 128
hamiltonian (*qdjj_solver.qp_solver.Eigenstates attribute*), 118
hamiltonian_sha256, 97
Harmonic (*class in qdjj_solver*), 110
harmonic() (*qdjj_solver.PeriodicHamiltonian method*), 112
harmonic_indices (*qdjj_solver.qp_solver.FloquetStates attribute*), 124
harmonic_weights (*qdjj_solver.qp_solver.FloquetStates attribute*), 124
harmonics, 71
harmonics (*qdjj_solver.ParameterDrive attribute*), 110
harmonics (*qdjj_solver.PeriodicHamiltonian attribute*), 111
hermitian_pair() (*in module qdjj_solver*), 109
hermiticity_error() (*qdjj_solver.FermionOperator method*), 106
hybridization_g() (*in module qdjj_solver.common.baths*), 133

I

implementation, 97
impurity, 62, 93
Impurity (*class in qdjj_solver*), 107
impurity_modes, 98, 99
index, 66, 71
index (*qdjj_solver.ParameterDrive attribute*), 110
initial (*qdjj_solver.SpectralLines attribute*), 116
interaction, 65

J

json_value() (*in module qdjj_solver.common.io*), 134

K

ket_indices (*qdjj_solver.FloquetTransitions attribute*), 114
ket_labels (*qdjj_solver.FloquetTransitions attribute*), 114
ket_quasienergies (*qdjj_solver.FloquetTransitions attribute*), 114
kind, 92, 93, 101
kind (*qdjj_solver.qp_solver.FloquetStates attribute*), 123

L

labels (*qdjj_solver.qp_solver.FloquetStates attribute*), 124
lanczos_maxiter (*qdjj_solver.dmrg_solver.SolverOptions attribute*), 131
lanczos_probability_tolerance (*qdjj_solver.dmrg_solver.SolverOptions attribute*), 132
lattice, 73
lattice (*qdjj_solver.dmrg_solver.PreparedModel attribute*), 128
levels, 64, 65
levels (*qdjj_solver.DiscreteBath property*), 105
line_shape() (*in module qdjj_solver*), 116
linear_operator() (*qdjj_solver.qp_solver.ProjectedOperator method*), 119
lines() (*qdjj_solver.FloquetTransitions method*), 115
live_bytes (*qdjj_solver.qp_solver.QPWorkspace property*), 127
load_result() (*in module qdjj_solver*), 104
lower_root_overlap, 101

M

make_model() (*in module qdjj_solver*), 109
matrix_element() (*qdjj_solver.dmrg_solver.PreparedModel method*), 129

matrix_elements() (*qdjj_solver.dmrg_solver.Eigenstates method*), 130
matrix_elements() (*qdjj_solver.dmrg_solver.PreparedModel method*), 129
matrix_elements() (*qdjj_solver.qp_solver.Eigenstates method*), 118
matrix_elements() (*qdjj_solver.qp_solver.FloquetStates method*), 125
max_dimension (*qdjj_solver.qp_solver.FloquetOptions attribute*), 122
max_dimension (*qdjj_solver.qp_solver.SolverOptions attribute*), 120
max_eigenpairs_per_target (*qdjj_solver.qp_solver.FloquetOptions attribute*), 122
max_harmonic, 68
max_harmonic (*qdjj_solver.PeriodicHamiltonian property*), 111
max_memory_gib (*qdjj_solver.qp_solver.FloquetOptions attribute*), 122
max_memory_gib (*qdjj_solver.qp_solver.SolverOptions attribute*), 120
max_nfev, 64
max_nnz (*qdjj_solver.qp_solver.FloquetOptions attribute*), 122
max_nnz (*qdjj_solver.qp_solver.SolverOptions attribute*), 120
max_order (*qdjj_solver.ParameterDrive property*), 111
max_sweeps (*qdjj_solver.dmrg_solver.SolverOptions attribute*), 130
maxiter (*qdjj_solver.qp_solver.FloquetOptions attribute*), 122
maxiter (*qdjj_solver.qp_solver.SolverOptions attribute*), 120
mean_energies (*qdjj_solver.qp_solver.FloquetStates attribute*), 124
metadata, 69, 70, 72, 93, 94
metadata (*qdjj_solver.DiscreteBath attribute*), 105
metadata (*qdjj_solver.dmrg_solver.Eigenstates attribute*), 130
metadata (*qdjj_solver.FloquetTransitions attribute*), 115
metadata (*qdjj_solver.Hamiltonian attribute*), 106
metadata (*qdjj_solver.Impurity attribute*), 107
metadata (*qdjj_solver.PeriodicHamiltonian attribute*), 111
metadata (*qdjj_solver.qp_solver.Eigenstates attribute*), 118
metadata (*qdjj_solver.qp_solver.FloquetStates attribute*), 125
metadata (*qdjj_solver.SpectralLines attribute*), 116
method, 98, 101
method (*qdjj_solver.qp_solver.FloquetOptions attribute*), 121
method (*qdjj_solver.qp_solver.FloquetStates attribute*), 123
method (*qdjj_solver.qp_solver.SolverOptions attribute*), 119
min_sweeps (*qdjj_solver.dmrg_solver.SolverOptions attribute*), 130
mixer (*qdjj_solver.dmrg_solver.SolverOptions attribute*), 131
mixer_amplitude (*qdjj_solver.dmrg_solver.SolverOptions attribute*), 131
mixer_sweeps (*qdjj_solver.dmrg_solver.SolverOptions attribute*), 131
mode_at() (*qdjj_solver.qp_solver.FloquetStates method*), 125
mode_operators, 73
mode_operators (*qdjj_solver.dmrg_solver.PreparedModel attribute*), 128
mode_order, 99
mode_order (*qdjj_solver.dmrg_solver.SolverOptions attribute*), 131
model, 91
model() (*qdjj_solver.dmrg_solver.PreparedModel method*), 129
model_from_config() (*in module qdjj_solver.common.io*), 133
modes (*qdjj_solver.qp_solver.FloquetStates attribute*), 124
mpo_dimensions, 99
mpo_original_dimensions, 100

N

ncv (*qdjj_solver.qp_solver.SolverOptions attribute*), 120
nimp, 69, 71, 94
nimp (*qdjj_solver.Hamiltonian attribute*), 106
nimp (*qdjj_solver.PeriodicHamiltonian attribute*), 111
nnz, 98
norm_error, 101
number() (*in module qdjj_solver*), 110

O

observables, 67--70, 72, 94, 96
observables (*qdjj_solver.dmrg_solver.Eigenstates attribute*), 130
observables (*qdjj_solver.dmrg_solver.SolverOptions attribute*), 131

observables (*qdjj_solver.Hamiltonian attribute*), 106
observables (*qdjj_solver.Impurity attribute*), 107
observables (*qdjj_solver.PeriodicHamiltonian attribute*), 111
observables (*qdjj_solver.qp_solver.Eigenstates attribute*), 118
occupations() (*qdjj_solver.qp_solver.FockBasis method*), 117
omega, 66, 67, 71
omega (*qdjj_solver.FloquetTransitions attribute*), 113
omega (*qdjj_solver.PeriodicHamiltonian attribute*), 111
omega (*qdjj_solver.qp_solver.FloquetStates attribute*), 123
one_body, 65
onsite_costs() (*qdjj_solver.dmrg_solver.PreparedModel method*), 129
operator, 69, 94
operator (*qdjj_solver.Hamiltonian attribute*), 106
operator (*qdjj_solver.Impurity attribute*), 107
operator_at() (*qdjj_solver.PeriodicHamiltonian method*), 112
operator_norm_bound, 99
options, 98, 99
orbital_density_matrix() (*in module qdjj_solver*), 116
orbital_density_operators() (*in module qdjj_solver*), 117
orbital_density_properties() (*in module qdjj_solver*), 117
orbitals, 69
orbitals (*qdjj_solver.Impurity attribute*), 107
order, 66, 68, 70, 72
order (*qdjj_solver.dmrg_solver.PreparedModel attribute*), 128
order (*qdjj_solver.Harmonic attribute*), 110
orthogonality_tolerance (*qdjj_solver.dmrg_solver.SolverOptions attribute*), 131
overlaps() (*qdjj_solver.dmrg_solver.Eigenstates method*), 130
overlaps() (*qdjj_solver.qp_solver.Eigenstates method*), 119

P

paired (*qdjj_solver.DiscreteBath property*), 105
pairs, 63
parameter, 66, 71
parameter (*qdjj_solver.ParameterDrive attribute*), 110
ParameterDrive (*class in qdjj_solver*), 110
parameters, 92
parity, 70
parity (*qdjj_solver.Sector attribute*), 108
particle_number, 70
particle_number (*qdjj_solver.Sector attribute*), 108
period (*qdjj_solver.PeriodicHamiltonian property*), 111
periodic (*qdjj_solver.qp_solver.FloquetStates attribute*), 124
PeriodicHamiltonian (*class in qdjj_solver*), 111
phase, 66, 71
phase (*qdjj_solver.Harmonic attribute*), 110
phase_velocities, 63, 93
phases, 62, 93
phi, 61
photons (*qdjj_solver.FloquetTransitions attribute*), 113
photons (*qdjj_solver.SpectralLines attribute*), 116
physical_operator() (*qdjj_solver.Hamiltonian method*), 107
prepare() (*in module qdjj_solver.dmrg_solver*), 129
prepare() (*qdjj_solver.qp_solver.QPWorkspace method*), 127
prepared (*qdjj_solver.dmrg_solver.Eigenstates attribute*), 130
PreparedModel (*class in qdjj_solver.dmrg_solver*), 128
probe (*qdjj_solver.FloquetTransitions attribute*), 115
probe() (*qdjj_solver.Hamiltonian method*), 107
probe() (*qdjj_solver.PeriodicHamiltonian method*), 112
PROBES (*in module qdjj_solver*), 113
project_operator() (*qdjj_solver.qp_solver.QPWorkspace method*), 127
ProjectedOperator (*class in qdjj_solver.qp_solver*), 119

Q

qp_counts() (*qdjj_solver.qp_solver.FockBasis method*), 117
qp_cutoff, 97, 99
qp_weights, 97
qp_weights (*qdjj_solver.qp_solver.Eigenstates attribute*), 118
qpsolver, 83
QPWorkspace (*class in qdjj_solver.qp_solver*), 127

quasienergies (*qdjj_solver.qp_solver.FloquetStates attribute*), 123
quasienergy_distance() (*in module qdjj_solver.common.floquet*), 134

R

real_arithmetic, 98
record() (*qdjj_solver.DiscreteBath method*), 105
record() (*qdjj_solver.Hamiltonian method*), 106
record() (*qdjj_solver.Harmonic method*), 110
record() (*qdjj_solver.ParameterDrive method*), 111
record() (*qdjj_solver.PeriodicHamiltonian method*), 112
record() (*qdjj_solver.SpectralLines method*), 116
reference, 67, 72
reference (*qdjj_solver.PeriodicHamiltonian attribute*), 111
reference_model() (*in module qdjj_solver*), 110
relative_weight, 64
representation (*qdjj_solver.qp_solver.FloquetStates attribute*), 123
require_convergence (*qdjj_solver.dmrg_solver.SolverOptions attribute*), 131
reserve() (*qdjj_solver.qp_solver.QPWorkspace method*), 127
reservoirs, 93
residual_method, 100
residual_tolerance (*qdjj_solver.dmrg_solver.SolverOptions attribute*), 130
residual_tolerance (*qdjj_solver.qp_solver.FloquetOptions attribute*), 122
residual_tolerance (*qdjj_solver.qp_solver.SolverOptions attribute*), 120
residuals, 96
residuals (*qdjj_solver.dmrg_solver.Eigenstates attribute*), 130
residuals (*qdjj_solver.qp_solver.Eigenstates attribute*), 118
residuals (*qdjj_solver.qp_solver.FloquetStates attribute*), 124
restriction, 99
rho (*qdjj_solver.DiscreteBath property*), 105
rho_wn, 61
rho_ws, 61
roots, 100
roots_energy_ordered, 100
rwa_model() (*in module qdjj_solver.qp_solver*), 126

S

same_states (*qdjj_solver.FloquetTransitions attribute*), 114
samples, 66, 67
sampling_tolerance, 67
save_result() (*in module qdjj_solver*), 104
scalar() (*qdjj_solver.FermionOperator class method*), 105
search_energy_shift, 99
search_shift_bound, 100
sector, 72, 91, 97, 99
Sector (*class in qdjj_solver*), 108
sector (*qdjj_solver.dmrg_solver.PreparedModel attribute*), 128
seed, 64, 101
seed (*qdjj_solver.dmrg_solver.SolverOptions attribute*), 131
seed (*qdjj_solver.qp_solver.FloquetOptions attribute*), 122
seed (*qdjj_solver.qp_solver.SolverOptions attribute*), 120
seed_randomization_steps (*qdjj_solver.dmrg_solver.SolverOptions attribute*), 131
seed_trials, 101
seed_trials (*qdjj_solver.dmrg_solver.SolverOptions attribute*), 131
selection (*qdjj_solver.qp_solver.FloquetOptions attribute*), 121
sha256, 102
sites, 73
sites (*qdjj_solver.dmrg_solver.PreparedModel attribute*), 128
solve() (*in module qdjj_solver*), 104
solve() (*in module qdjj_solver.dmrg_solver*), 132
solve() (*in module qdjj_solver.qp_solver*), 120
solve() (*qdjj_solver.qp_solver.QPWorkspace method*), 127
solve_floquet() (*in module qdjj_solver.qp_solver*), 126
solver, 92
SolverOptions (*class in qdjj_solver.dmrg_solver*), 130
SolverOptions (*class in qdjj_solver.qp_solver*), 119
source_provenance() (*in module qdjj_solver.common.io*), 133
sparse_matrix() (*qdjj_solver.qp_solver.ProjectedOperator method*), 119
SpectralLines (*class in qdjj_solver*), 115
spectrum() (*qdjj_solver.FloquetTransitions method*), 115
spectrum() (*qdjj_solver.SpectralLines method*),

116
spins, 69, 72, 94
spins (*qdjj_solver.Hamiltonian attribute*), 106
spins (*qdjj_solver.PeriodicHamiltonian attribute*), 111
starts, 64
state_amplitudes, 97
state_artifact, 97
state_at() (*qdjj_solver.qp_solver.FloquetStates method*), 125
states (*qdjj_solver.dmrg_solver.Eigenstates attribute*), 130
states (*qdjj_solver.qp_solver.FloquetOptions attribute*), 121
static (*qdjj_solver.qp_solver.FloquetStates attribute*), 124
static_weights (*qdjj_solver.qp_solver.FloquetStates attribute*), 124
step_check (*qdjj_solver.qp_solver.FloquetOptions attribute*), 122
step_tolerance (*qdjj_solver.qp_solver.FloquetOptions attribute*), 122
steps (*qdjj_solver.qp_solver.FloquetOptions attribute*), 122
strengths (*qdjj_solver.FloquetTransitions property*), 115
strengths (*qdjj_solver.SpectralLines attribute*), 116
substitute() (*qdjj_solver.FermionOperator method*), 105
surrogate, 92
svd_min (*qdjj_solver.dmrg_solver.SolverOptions attribute*), 131
sweep_converged, 100
sweep_energy_converged, 100
sweep_statistics, 101
sweeps, 101
symmetry, 62

T

target_charge, 73
target_charge (*qdjj_solver.dmrg_solver.PreparedModel attribute*), 128
target_shift (*qdjj_solver.qp_solver.FloquetOptions attribute*), 122
targets (*qdjj_solver.qp_solver.FloquetOptions attribute*), 121
tenpy, 98
term() (*qdjj_solver.dmrg_solver.PreparedModel method*), 128
threads (*qdjj_solver.dmrg_solver.SolverOptions attribute*), 132
threads (*qdjj_solver.qp_solver.FloquetOptions attribute*), 122
threads (*qdjj_solver.qp_solver.SolverOptions attribute*), 120
time_average() (*qdjj_solver.PeriodicHamiltonian method*), 112
timings (*qdjj_solver.dmrg_solver.Eigenstates attribute*), 130
timings (*qdjj_solver.qp_solver.Eigenstates attribute*), 118
timings (*qdjj_solver.qp_solver.FloquetStates attribute*), 124
timings_seconds, 97
to_records() (*qdjj_solver.FermionOperator method*), 106
tolerance, 68
tolerance (*qdjj_solver.qp_solver.FloquetOptions attribute*), 122
tolerance (*qdjj_solver.qp_solver.SolverOptions attribute*), 119
total (*qdjj_solver.FloquetTransitions attribute*), 114
tracking_threshold (*qdjj_solver.qp_solver.FloquetOptions attribute*), 122
transition() (*qdjj_solver.qp_solver.ProjectedOperator method*), 119
tunneling, 62, 93
twice_sz, 70
twice_sz (*qdjj_solver.Sector attribute*), 108

U

u, 61, 65

V

validate_config() (*in module qdjj_solver.common.io*), 133
validation, 102
value() (*qdjj_solver.ParameterDrive method*), 111
van_vleck() (*qdjj_solver.PeriodicHamiltonian method*), 112
vectors (*qdjj_solver.qp_solver.Eigenstates attribute*), 118
verify, 67, 68

W

weights, 68
weights (*qdjj_solver.DiscreteBath attribute*), 105
wide_band, 65
write_json() (*in module qdjj_solver.common.io*), 134

X

xi, 68
xi (*qdjj_solver.DiscreteBath attribute*), 104

Z

zone_center (*qdjj_solver.qp_solver.FloquetOptions attribute*), 122