Quantics Tensor Trains for Vibrational Eigenstates

Imaginary-time propagation, physical-core Fourier transforms, block Rayleigh–Ritz, and Morse-oscillator benchmarks

Victor S. Batista

Contents

1 Purpose and scope

This tutorial explains the complete numerical strategy implemented in the supplied notebook for computing multiple low-energy vibrational eigenstates without storing the full multidimensional wavefunction. The central idea is to represent functions on binary grids by *quantics tensor trains* (QTTs), apply coordinate- and momentum-space operations directly to those compressed objects, and extract a block of low-energy states by imaginary-time filtering combined with repeated orthogonalization and Rayleigh–Ritz diagonalization.

The Hamiltonian implemented in the notebook is

$$\hat{H} = \sum_{d=1}^{n} \left[\frac{\hat{p}_d^2}{2m_d} + D_{e,d} \left(1 - e^{-a_d(\hat{x}_d - x_{e,d})} \right)^2 \right] + V_{\mathrm{coup}}(x_1, \ldots, x_n). \tag{1}$$

The code supports uncoupled Morse modes, pairwise coupling in natural Morse coordinates, and an arbitrary custom many-body potential sampled by TT-cross.

Important note about the saved notebook run

The opening markdown in the notebook describes a two-mode default benchmark, but the actually executed parameter cell sets

`N_OSC = 3`, `NSTATES = 20`, `COUPLING_MODEL = "none"`.

Therefore, the numerical outputs and all saved figures in the supplied notebook correspond to **three uncoupled Morse oscillators**. The two-dimensional heat-map routine is present in the code, but its saved output is only the message that heat maps require `N_OSC == 2`; no heat-map image was generated in this run. This document includes **all eight figures that were actually produced** and all displayed numerical tables.

2 Why tensor trains are useful

2.1 The exponential storage problem

Suppose a wavefunction depends on n coordinates and coordinate d is sampled on N_d grid points. A dense representation requires

$$N_{\mathrm{dense}} = \prod_{d=1}^{n} N_d \tag{2}$$

complex amplitudes. If $N_d = N$, this grows as N^n. The three-mode demonstration is still manageable as a dense array, but the code is written to avoid dependence on dense reconstruction because the same representation is intended to extend to much larger n.

2.2 Tensor-train representation

A tensor train factorizes a tensor into a chain of low-order cores,

$$\Psi_{i_1 \cdots i_D} = \sum_{\alpha_1 \cdots \alpha_{D-1}} G^{(1)}_{1,i_1,\alpha_1} G^{(2)}_{\alpha_1,i_2,\alpha_2} \cdots G^{(D)}_{\alpha_{D-1},i_D,1}. \tag{3}$$

The intermediate dimensions r_k are the TT ranks (bond dimensions). If the physical dimension of each core is q and the ranks remain of order r, storage scales approximately as $O(Dqr^2)$ rather than exponentially in D.

2.3 Quantics tensor trains

QTT applies the TT idea to the binary digits of an ordinary grid index. For a one-dimensional grid of size $N = 2^L$,

$$i = \sum_{k=1}^{L} 2^{L-k} b_k, \qquad b_k \in \{0, 1\}. \tag{4}$$

Thus a vector f_i is reinterpreted as an L-index binary tensor $f(b_1, \ldots, b_L)$ and compressed as a TT with physical dimension 2 at every core. For multiple coordinates, the notebook orders the QTT cores as

`[bits of x1][bits of x2]...[bits of xn]`.

With seven bits per coordinate and three coordinates, the saved calculation has 21 binary cores.

2.4 Tiny TT-SVD example

The notebook begins with the vector

$$v = (1, 2, 3, 4), \tag{5}$$

reshaped into the 2×2 matrix

$$M = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}. \tag{6}$$

Its singular values are 5.4649857 and 0.36596619, so the exact bond dimension is 2. The two TT cores have shapes $(1, 2, 2)$ and $(2, 2, 1)$, and contracting them reconstructs the original vector exactly. The full routine `qtt_from_dense()` recursively repeats this SVD splitting over all binary digits.

3 Core QTT algebra used by the notebook

The notebook deliberately implements the most important TT operations explicitly on `tntorch.Tensor` cores. This makes the handling of complex tensors transparent once Fourier transforms are introduced.

3.1 Rounding and rank control

After addition or elementwise multiplication, TT ranks increase. Exact TT addition forms block-structured cores and roughly adds ranks, whereas exact Hadamard products multiply ranks. The notebook therefore applies TT rounding after these operations. The truncation rule retains the smallest number of singular values whose discarded tail satisfies the requested Frobenius-norm tolerance, subject to a hard rank cap.

Table 1: Principal compression and TT-cross parameters in the executed notebook.

Parameter	Value	Role
`QTT_EPS`	1.0×10^{-12}	Rounding after QTT algebra
`QFT_EPS`	2.0×10^{-13}	Rounding after Fourier transforms
`QTT_RMAX`	96	Hard rank ceiling
`CROSS_EPS`	1.0×10^{-11}	TT-cross target for custom/coupled nonlinear maps
`CROSS_RMAX`	96	TT-cross rank ceiling
`CROSS_MAX_ITER`	40	TT-cross iterations
`CROSS_VAL_SIZE`	4000	TT-cross validation sample size

3.2 Inner products

For QTTs a and b, the algebraic inner product is contracted core by core,

$$\langle a|b\rangle = \sum_{\boldsymbol{i}} a^*_{\boldsymbol{i}} b_{\boldsymbol{i}}. \tag{7}$$

The only intermediate object is a small environment matrix whose dimensions are TT ranks. Physical quadrature weights are inserted separately where needed. In the present uniform-grid problem,

$$\Delta V = \prod_{d=1}^{n} \Delta x_d. \tag{8}$$

4 Physical-core Fourier transform

4.1 Why a Fourier transform is needed

The potential is diagonal in coordinate space, but the kinetic energy is diagonal in momentum space. The notebook therefore needs a multidimensional Fourier transform of a QTT wavefunction.

Instead of implementing a gate-by-gate binary QFT circuit, the notebook uses a pedagogically transparent *unfold–FFT–refold* procedure for each physical coordinate:

1. Contract the L_d binary cores of coordinate d into one physical core of shape $(R_{\mathrm{L}}, N_d, R_{\mathrm{R}})$.
2. Apply an ordinary 1D FFT along the physical index N_d, independently for every pair of bond indices.
3. Split the transformed physical core back into binary cores by exact recursive SVD, concatenate all coordinate blocks, and round the complete QTT once.

The FFT uses `norm="ortho"`, making the forward and inverse transforms unitary.

The high-level implementation is:

Listing 1: High-level QTT Fourier transform from the notebook.

```
def qtt_qft(
    qtt,
    bits_per_dim,
    inverse=False,
    eps=QFT_EPS,
    rmax=QTT_RMAX,
    norm="ortho"
):
    """
    Multidimensional QFT/IQFT of a QTT wavefunction, tying the three steps
    together:

        QTT  --(unfold)-->  one physical TT core per coordinate
             --(FFT each core's columns)-->  transformed physical cores
             --(fold + round)-->  binary QTT

    Set inverse=True for the inverse transform (used to come back from
    momentum space to coordinate space after applying the kinetic
    propagator/operator).
    """
    physical_cores = qtt_unfold_to_physical_tt(
        qtt,
        bits_per_dim
```

```
    )

    transformed = fft_physical_tt_cores(
        physical_cores,
        inverse=inverse,
        norm=norm
    )

    return physical_tt_to_qtt(
        transformed,
        bits_per_dim,
        eps=eps,
        rmax=rmax
    )
```

4.2 Round-trip validation

The notebook constructs a separable test function directly as a QTT, transforms it to momentum space and back, and computes

$$\epsilon_{\mathrm{QFT}} = \frac{\|\mathrm{QFT}^{-1}\mathrm{QFT}f - f\|}{\|f\|}. \tag{9}$$

For the saved run,

$$\boxed{\epsilon_{\mathrm{QFT}} = 7.721 \times 10^{-15}}, \tag{10}$$

which is essentially machine precision for double precision. The real-space test QTT and transformed test QTT both have maximum rank 8; the coefficient counts are 630 and 864, respectively.

5 Morse Hamiltonian and numerical grid

5.1 One-dimensional Morse modes

Each coordinate uses the unshifted Morse potential

$$V_d(x) = D_{e,d}\left(1 - e^{-a_d(x-x_{e,d})}\right)^2, \tag{11}$$

with minimum zero and dissociation limit $D_{e,d}$. The natural Morse coordinate used for pair coupling is

$$s_d(x_d) = 1 - e^{-a_d(x_d-x_{e,d})}. \tag{12}$$

For pairwise coupling,

$$V_{\mathrm{coup}} = \sum_{d<e} \kappa_{de} s_d(x_d) s_e(x_e). \tag{13}$$

The saved run sets the coupling model to `none`.

Table 2: Executed three-mode Morse parameters and Fourier-grid discretization.

d	m	D_e	a	x_e	L	N	$x_{\min}$	$x_{\max}$	Δx
0	1.00	20.0	0.80	0.0	7	128	-6.0	18.0	0.1875
1	1.05	18.5	0.77	0.0	7	128	-6.0	18.0	0.1875
2	1.10	17.0	0.74	0.0	7	128	-6.0	18.0	0.1875

The run therefore uses

$$\texttt{BITS\_PER\_DIM} = [7, 7, 7], \tag{14}$$

$$L_{\text{total}} = 21, \tag{15}$$

$$N_{\text{dense}} = 128^3 = 2{,}097{,}152, \tag{16}$$

$$\Delta V = 6.591797 \times 10^{-3}. \tag{17}$$

Figure 1: All three one-dimensional Morse potentials in the executed notebook. They share $x_e = 0$ but differ in mass, dissociation energy, and range parameter. The asymptotic plateaus are $D_e = 20.0$, 18.5, and 17.0.

6 Constructing the potential and kinetic operators in QTT form

6.1 Structured potential construction

For the uncoupled and pair-coupled Morse models, the notebook does not approximate the potential by TT-cross. Instead it exploits algebraic separability. For the uncoupled case,

$$V(\boldsymbol{x}) = \sum_d V_d(x_d), \tag{18}$$

and each V_d is embedded into the full QTT by tensoring it with all-ones QTTs on the other coordinates. For pair coupling, each $s_i(x_i)s_j(x_j)$ term is likewise a tensor product of one-dimensional factors. Only the custom many-body branch uses TT-cross to learn the full potential.

For the saved run, the structured potential has maximum QTT rank 4 and 368 core coefficients. Because no TT-cross approximation is used in this branch, its approximation error is controlled only by QTT rounding.

6.2 Kinetic operator

The momentum grid is generated from `torch.fft.fftfreq`, with

$$p = 2\pi\hbar f, \tag{19}$$

and

$$T(\boldsymbol{p}) = \sum_d \frac{p_d^2}{2m_d}. \tag{20}$$

The full momentum-space kinetic diagonal is again assembled as a sum of embedded one-dimensional QTTs. In the saved run, it has maximum rank 3 and 314 coefficients.

7 Hamiltonian action without dense reconstruction

For any QTT state ψ, the notebook evaluates

$$\hat{H}\psi = V\psi + \mathcal{F}^{-1}[T(\boldsymbol{p})\,\mathcal{F}\psi]\,. \tag{21}$$

The central routine is:

Listing 2: QTT-native Hamiltonian action.

```
def apply_H_qtt(psi_qtt, eps=QTT_EPS, qft_eps=QFT_EPS):
    Vpsi = qtt_hadamard(Vtot_qtt, psi_qtt, eps=eps, rmax=QTT_RMAX)

    psi_p = qtt_qft(
        psi_qtt, BITS_PER_DIM, inverse=False,
        eps=qft_eps, rmax=QTT_RMAX
    )

    Tpsi_p = qtt_hadamard(Ttot_qtt, psi_p, eps=eps, rmax=QTT_RMAX)

    Tpsi = qtt_qft(
        Tpsi_p, BITS_PER_DIM, inverse=True,
        eps=qft_eps, rmax=QTT_RMAX
    )

    return qtt_add(Vpsi, Tpsi, eps=eps, rmax=QTT_RMAX)
```

The Rayleigh quotient is computed as

$$E[\psi] = \frac{\langle\psi|\hat{H}|\psi\rangle}{\langle\psi|\psi\rangle}, \tag{22}$$

where the common volume factor cancels. Normalization itself uses the physical quadrature norm $\Delta V\langle\psi|\psi\rangle = 1$.

8 Block orthogonalization and Rayleigh–Ritz

Independent imaginary-time propagation would drive every trial vector toward the ground state. To obtain many states simultaneously, the notebook propagates a block and repeatedly restores orthogonality.

For states $\{\psi_i\}_{i=1}^M$, the overlap matrix is

$$S_{ij} = \Delta V\langle\psi_i|\psi_j\rangle. \tag{23}$$

A symmetric Löwdin transformation constructs an orthonormal block,

$$\phi_j = \sum_i \psi_i [S^{-1/2}]_{ij}. \tag{24}$$

Only the $M \times M$ matrix S is dense; the states remain QTTs.

The projected Hamiltonian is

$$H_{ij}^{\mathrm{sub}} = \Delta V\langle\phi_i|\hat{H}|\phi_j\rangle. \tag{25}$$

Diagonalizing H^{sub} gives the Ritz energies and the rotation of the QTT block:

Listing 3: Block Rayleigh–Ritz update.

```python
def rayleigh_ritz_qtt(states):
    """QTT block orthogonalization + projected-Hamiltonian diagonalization."""
    states = orthonormalize_qtt_block(states, passes=2)
    M = len(states)

    Hstates = [apply_H_qtt(state) for state in states]
    Hsub = torch.zeros((M, M), dtype=torch.complex128, device=device)

    for i in range(M):
        for j in range(M):
            Hsub[i, j] = dvol * qtt_inner(states[i], Hstates[j])

    Hsub = 0.5 * (Hsub + Hsub.conj().T)
    evals, U = torch.linalg.eigh(Hsub)

    rotated = rotate_qtt_block(states, U)
    rotated = [qtt_normalize(state) for state in rotated]

    return rotated, evals.real
```

This is the key step that converts imaginary-time filtering into a stable multi-eigenstate method.

9 Analytical Morse reference spectrum

For one Morse oscillator, the notebook defines

$$\lambda_d = \frac{\sqrt{2m_d D_{e,d}}}{\hbar a_d}, \tag{26}$$

and the bound-state energies

$$E_{n,d} = D_{e,d} - \frac{\hbar^2 a_d^2}{2m_d}\left(\lambda_d - n - \frac{1}{2}\right)^2, \qquad \lambda_d - n - \frac{1}{2} > 0. \tag{27}$$

The three executed modes have $\lambda = (7.905694, 8.094775, 8.264262)$ and each supports eight bound levels according to this criterion.

For uncoupled modes, exact multidimensional states are products and energies add,

$$E_{n_1 n_2 n_3} = E_{n_1,1} + E_{n_2,2} + E_{n_3,3}. \tag{28}$$

The notebook uses a heap-based k-smallest-sums algorithm to generate only the requested lowest 20 product levels instead of enumerating all possible tuples.

Table 3: Lowest 20 analytical uncoupled product levels used as the reference spectrum.

Rank	Quantum numbers	E_{exact}	E_{QTT}	$\lvert\Delta E\rvert$
0	(0, 0, 0)	6.659486316765	6.659486318053	1.288e-09
1	(0, 0, 1)	10.275768058601	10.275768060342	1.741e-09
2	(0, 1, 0)	10.665669315824	10.665669317948	2.124e-09
3	(1, 0, 0)	11.079130573034	11.079130575842	2.808e-09
4	(0, 0, 2)	13.394231618620	13.394231620733	2.113e-09
5	(0, 2, 0)	14.107185648216	14.107185651099	2.883e-09
6	(0, 1, 1)	14.281951057661	14.281951060154	2.493e-09

Rank	Quantum numbers	E_{exact}	E_{QTT}	$\lvert\Delta E\rvert$
7	(1, 0, 1)	14.695412314871	14.695412318056	3.185e-09
8	(2, 0, 0)	14.858774829303	14.858774833587	4.284e-09
9	(1, 1, 0)	15.085313572093	15.085313575650	3.557e-09
10	(0, 0, 3)	16.014876996821	16.014876999249	2.428e-09
11	(0, 3, 0)	16.984035313942	16.984035317392	3.450e-09
12	(0, 1, 2)	17.400414617679	17.400414620559	2.880e-09
13	(0, 2, 1)	17.723467390053	17.723467393313	3.260e-09
14	(1, 0, 2)	17.813875874889	17.813875878483	3.594e-09
15	(3, 0, 0)	17.998419085573	17.998419090909	5.336e-09
16	(0, 0, 4)	18.137704193203	18.137704195775	2.572e-09
17	(2, 0, 1)	18.475056571140	18.475056575817	4.677e-09
18	(1, 2, 0)	18.526829904486	18.526829908837	4.351e-09
19	(1, 1, 1)	18.701595313930	18.701595317883	3.953e-09

9.1 Analytical wavefunctions

The one-dimensional reference wavefunctions are generated from

$$\psi_n(x) \propto z^s e^{-z/2} L_n^{(2s)}(z), \qquad z = 2\lambda e^{-a(x-x_e)}, \qquad s = \lambda - n - \frac{1}{2}, \tag{29}$$

then normalized on the numerical grid. Each one-dimensional factor is converted to QTT form, and the multidimensional analytical product state is formed by concatenating independent QTT core lists. Thus even the exact reference states need not be assembled as dense three-dimensional arrays.

10 Local Fourier-grid eigenstates as numerical seeds

The initial block is not built from the analytical Morse wavefunctions. Instead, the notebook solves a small one-dimensional Fourier-grid Hamiltonian for each mode,

$$H_d^{\mathrm{FGH}} = F_d^\dagger \,\mathrm{diag}[T_d(p)]\, F_d + \mathrm{diag}[V_d(x)]. \tag{30}$$

Each F_d is the unitary discrete FFT matrix. These are only 128×128 eigenproblems, so they are inexpensive even when the full product grid becomes too large. Products of these local FGH eigenvectors provide the initial QTT block.

The notebook explicitly checks the local grid accuracy before the multidimensional calculation:

Table 4: One-dimensional Fourier-grid Hamiltonian energies versus analytical Morse energies.

Mode	n	E_{exact}	$E_{\text{1D FGH}}$	Absolute error
0	0	2.449822	2.449822	7.150e-14
0	1	6.869466	6.869466	3.464e-14
0	2	10.649111	10.649111	4.459e-13
0	3	13.788755	13.788755	2.689e-12
1	0	2.214841	2.214841	2.842e-14
1	1	6.221024	6.221024	5.063e-14
1	2	9.662541	9.662541	2.203e-13
1	3	12.539390	12.539390	6.786e-13
2	0	1.994823	1.994823	1.288e-14
2	1	5.611104	5.611104	2.842e-14
2	2	8.729568	8.729568	3.553e-15
2	3	11.350213	11.350213	4.796e-14
2	4	13.473041	13.473041	1.670e-13

The largest displayed local error is only a few 10^{-12}, so the grid discretization of the low-lying levels is already much more accurate than the final QTT residual norm. This cleanly separates ordinary grid error from errors introduced by repeated QTT transformations, truncations, and propagation.

The initial product seeds have maximum rank 8. Because the executed problem is uncoupled, these FGH product states are already extremely close to eigenstates of the multidimensional discretized Hamiltonian; the initial Rayleigh–Ritz energies already match the analytical product spectrum to many digits.

11 Imaginary-time propagation

11.1 Filtering principle

Expanding an initial state in exact eigenvectors,

$$|\psi(0)\rangle = \sum_n c_n |n\rangle , \tag{31}$$

its imaginary-time evolution is

$$|\psi(\tau)\rangle = e^{-\tau \hat{H}} |\psi(0)\rangle = \sum_n c_n e^{-\tau E_n} |n\rangle . \tag{32}$$

Higher-energy components are exponentially suppressed. For a block of states, orthogonalization and Rayleigh–Ritz after each step prevent all vectors from collapsing onto the same ground state.

11.2 Second-order split operator

The notebook uses Strang splitting,

$$e^{-\Delta\tau \hat{H}} = e^{-\Delta\tau V/2} e^{-\Delta\tau T} e^{-\Delta\tau V/2} + O(\Delta\tau^3). \tag{33}$$

For the uncoupled run, $V = \sum_d V_d$ and the potential propagator factorizes exactly,

$$e^{-\Delta\tau V/2} = \bigotimes_d e^{-\Delta\tau V_d/2}. \tag{34}$$

The kinetic propagator always factorizes in momentum space,

$$e^{-\Delta\tau T} = \bigotimes_d e^{-\Delta\tau p_d^2/(2m_d)}. \tag{35}$$

For coupled potentials, the potential propagator is instead obtained by applying the scalar map $v \mapsto e^{-\Delta\tau v/2}$ to the potential QTT using TT-cross.

The one-state step is:

Listing 4: One second-order imaginary-time step in QTT form.

```
def imaginary_time_step_qtt(psi_qtt, Uv_qtt, Ut_qtt):
    # Second-order Strang splitting:
    # exp(-dt H) = exp(-dt V/2) exp(-dt T) exp(-dt V/2) + O(dt^3)
    psi = qtt_hadamard(Uv_qtt, psi_qtt, eps=QTT_EPS, rmax=QTT_RMAX)
    psi = qtt_qft(psi, BITS_PER_DIM, inverse=False, eps=QFT_EPS, rmax=QTT_RMAX)
    psi = qtt_hadamard(Ut_qtt, psi, eps=QTT_EPS, rmax=QTT_RMAX)
    psi = qtt_qft(psi, BITS_PER_DIM, inverse=True, eps=QFT_EPS, rmax=QTT_RMAX)
    psi = qtt_hadamard(Uv_qtt, psi, eps=QTT_EPS, rmax=QTT_RMAX)
    return qtt_normalize(psi)
```

11.3 Executed propagation schedule

The uncoupled schedule contains 12 block iterations:

Table 5: Imaginary-time schedule in the saved uncoupled run.

Stage	$\Delta\tau$	Number of steps
1	5.0×10^{-3}	3
2	1.0×10^{-3}	4
3	2.0×10^{-4}	5

After every step, all 20 propagated states undergo block Rayleigh–Ritz. The maximum rank is 15 at every recorded iteration.

12 Validation and state assignment

12.1 Dense validation branch

The notebook contains an optional dense multidimensional FGH validation when the total grid contains at most 200,000 points. The executed grid contains 2,097,152 points, so this branch is skipped. This is intentional: the scalable QTT calculation is unaffected by the absence of dense validation.

12.2 Exact-state matching

For the uncoupled case the notebook constructs the squared-overlap matrix

$$O_{ij} = \frac{|\Delta V \langle \Psi_i^{\text{exact}} | \Psi_j^{\text{QTT}} \rangle|^2}{(\Delta V \langle \Psi_i^{\text{exact}} | \Psi_i^{\text{exact}} \rangle)(\Delta V \langle \Psi_j^{\text{QTT}} | \Psi_j^{\text{QTT}} \rangle)}. \tag{36}$$

The Hungarian algorithm finds the globally optimal one-to-one assignment between analytical and numerical states. This is more robust than matching only by sorted energy, especially near degeneracies.

12.3 Hamiltonian residual

For each state, the notebook also evaluates

$$r_j = \frac{\|\hat{H}\psi_j - E_j\psi_j\|}{\|\psi_j\|}. \tag{37}$$

Energy error, fidelity, and residual norm probe different aspects of accuracy. In particular, a Rayleigh quotient can be extremely accurate even when a small vector residual remains; the residual is therefore an important complementary convergence diagnostic.

The summary printed by the notebook is

$$\max_j |E_j^{\mathrm{QTT}} - E_j^{\mathrm{exact}}| = 5.336 \times 10^{-9}, \tag{38}$$

$$\min_j O_{jj} = 0.999999999755, \tag{39}$$

$$\max_j r_j = 3.798 \times 10^{-4}. \tag{40}$$

Every matched state is therefore essentially identical to its analytical product reference by overlap, while the QTT coefficient counts remain between roughly one and a few thousand for a dense grid containing more than two million amplitudes.

13 QTT-native one-coordinate marginals

For dimensions higher than two, full wavefunction heat maps become both visually unhelpful and potentially expensive. The notebook instead computes

$$P_d(x_d) = \int \prod_{e \neq d} \mathrm{d}x_e \, |\Psi(x_1, \ldots, x_n)|^2. \tag{41}$$

It first forms the QTT of $|\Psi|^2$, unfolds each physical coordinate, contracts every coordinate except d, multiplies by the missing volume factor, removes tiny negative roundoff, and renormalizes the resulting one-dimensional probability distribution.

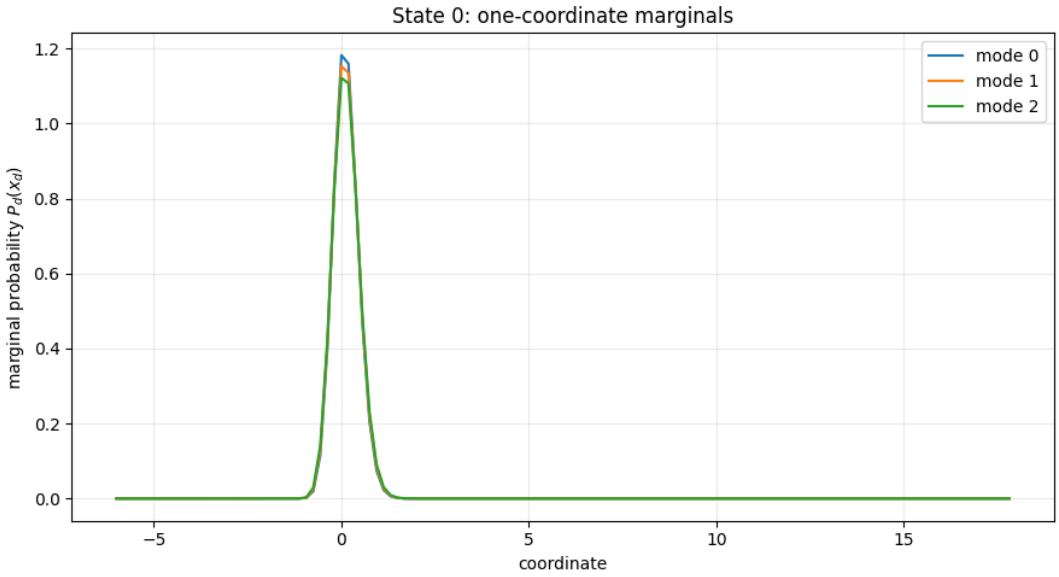

Figure 2: One-coordinate probability marginals of state 0, identified with $(n_0, n_1, n_2) = (0, 0, 0)$. All three modes show the single-peaked ground-state probability distribution near the Morse minimum.

Table 6: Complete final state-comparison table displayed by the notebook. The saved pandas display rounds the individual fidelities to 1.000000; the separately printed minimum fidelity is 0.999999999755.

Exact rank	Num. index	Quantum numbers	E_{exact}	E_{QTT}	$\lvert\Delta E\rvert$	Relative error	Fidelity	Residual	Max rank	QTT coeffs.
0	0	(0, 0, 0)	6.659486	6.659486	1.288e-09	1.935e-10	1.000000	0.000185	10	1128
1	1	(0, 0, 1)	10.275768	10.275768	1.740e-09	1.694e-10	1.000000	0.000207	10	1104
2	2	(0, 1, 0)	10.665669	10.665669	2.124e-09	1.991e-10	1.000000	0.000229	8	1024
3	3	(1, 0, 0)	11.079131	11.079131	2.808e-09	2.535e-10	1.000000	0.000270	10	1218
4	4	(0, 0, 2)	13.394232	13.394232	2.113e-09	1.578e-10	1.000000	0.000228	12	1678
5	5	(0, 2, 0)	14.107186	14.107186	2.883e-09	2.044e-10	1.000000	0.000268	10	1398
6	6	(0, 1, 1)	14.281951	14.281951	2.493e-09	1.746e-10	1.000000	0.000246	11	1538
7	7	(1, 0, 1)	14.695412	14.695412	3.185e-09	2.167e-10	1.000000	0.000284	13	1850
8	8	(2, 0, 0)	14.858775	14.858775	4.283e-09	2.883e-10	1.000000	0.000337	12	1660
9	9	(1, 1, 0)	15.085314	15.085314	3.556e-09	2.357e-10	1.000000	0.000300	11	1282
10	10	(0, 0, 3)	16.014877	16.014877	2.428e-09	1.516e-10	1.000000	0.000243	13	1948
11	11	(0, 3, 0)	16.984035	16.984035	3.450e-09	2.031e-10	1.000000	0.000295	11	1622
12	12	(0, 1, 2)	17.400415	17.400415	2.879e-09	1.655e-10	1.000000	0.000263	13	1876
13	13	(0, 2, 1)	17.723467	17.723467	3.260e-09	1.839e-10	1.000000	0.000283	12	1888
14	14	(1, 0, 2)	17.813876	17.813876	3.594e-09	2.017e-10	1.000000	0.000299	15	2272
15	15	(3, 0, 0)	17.998419	17.998419	5.336e-09	2.965e-10	1.000000	0.000380	15	2302
16	16	(0, 0, 4)	18.137704	18.137704	2.572e-09	1.418e-10	1.000000	0.000250	15	2420
17	17	(2, 0, 1)	18.475057	18.475057	4.677e-09	2.532e-10	1.000000	0.000349	14	2284
18	18	(1, 2, 0)	18.526830	18.526830	4.351e-09	2.348e-10	1.000000	0.000331	12	1748
19	19	(1, 1, 1)	18.701595	18.701595	3.953e-09	2.114e-10	1.000000	0.000313	13	2086

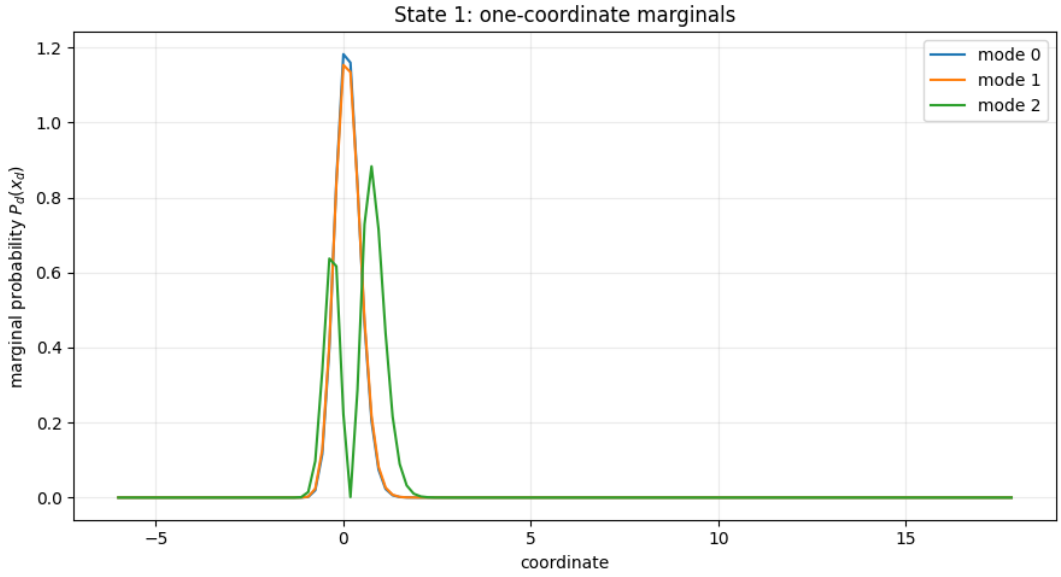

Figure 3: Marginals of state 1, identified with $(0, 0, 1)$. Modes 0 and 1 remain in their ground states, while mode 2 carries the first vibrational excitation and therefore shows the characteristic nodal separation in its probability density.

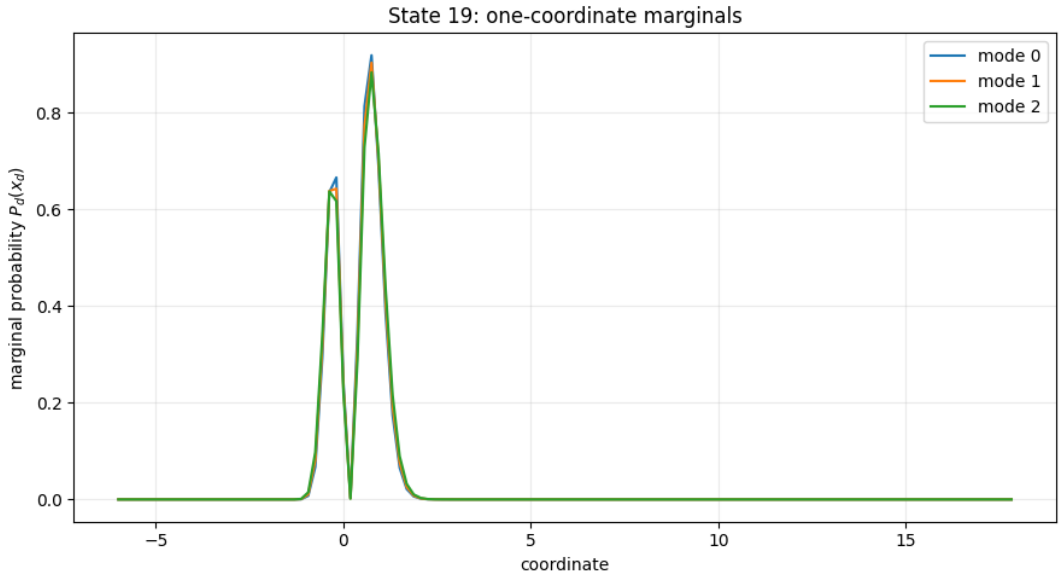

Figure 4: Marginals of state 19, identified with $(1, 1, 1)$. All three coordinates occupy first-excited Morse states, producing closely overlapping two-lobed marginal probability distributions.

14 Convergence and compression diagnostics

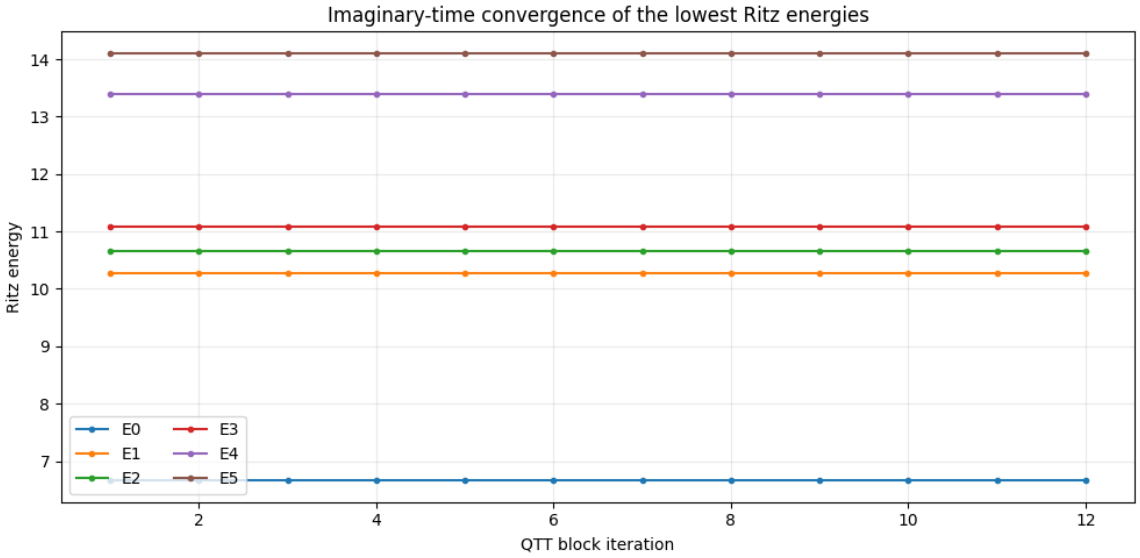

Figure 5: Imaginary-time evolution of the six lowest Ritz energies. The curves are nearly flat because the local-FGH product seeds are already extremely accurate eigenstates for the uncoupled Hamiltonian. In this run the propagation therefore serves mainly as a consistency test of the QTT split-operator and block-rotation machinery.

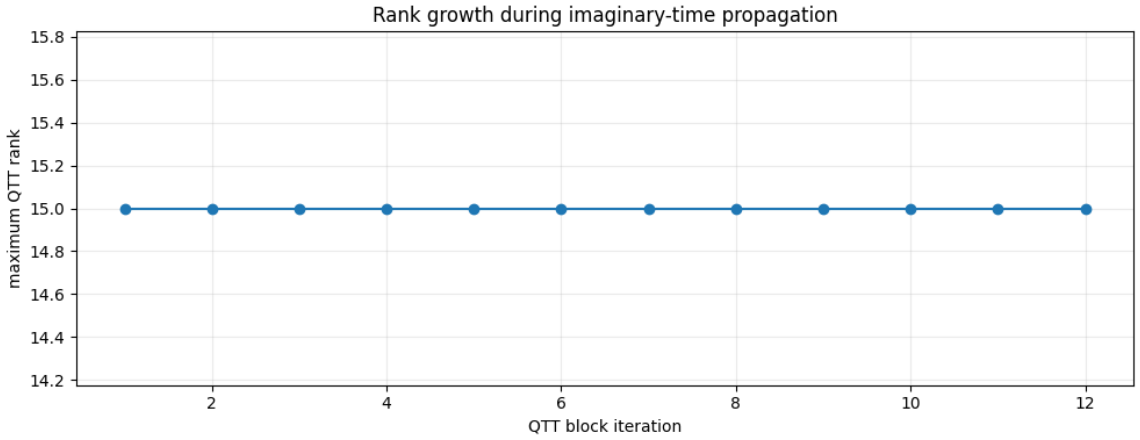

Figure 6: Maximum TT rank of the 20-state block during the 12 imaginary-time iterations. It remains fixed at 15, indicating stable compression for this uncoupled benchmark.

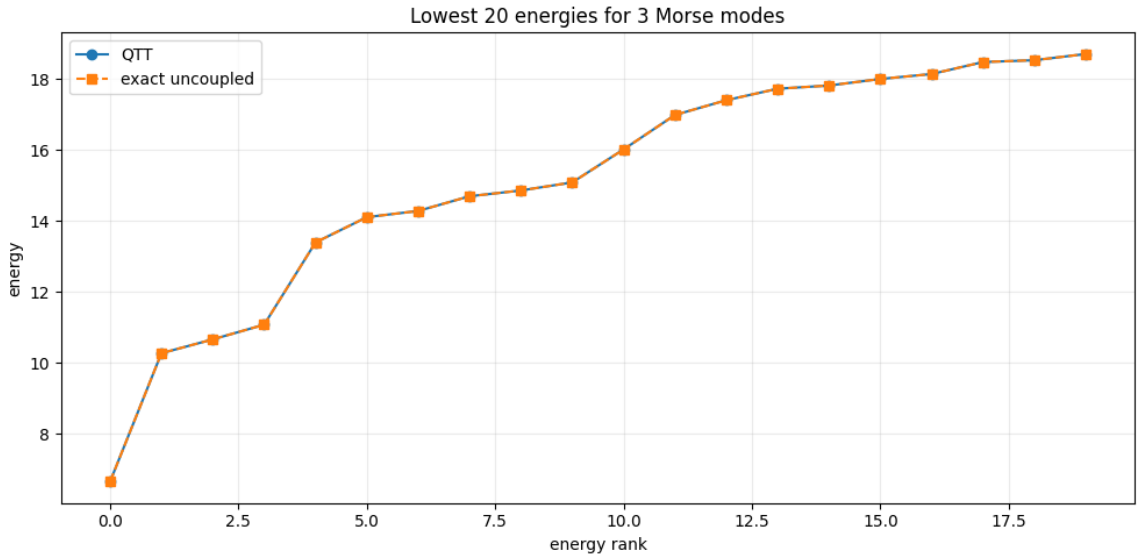

Figure 7: Lowest 20 QTT energies compared with the analytical uncoupled Morse product spectrum. The two curves overlap at plotting resolution, consistent with the maximum absolute error of 5.336×10^{-9}.

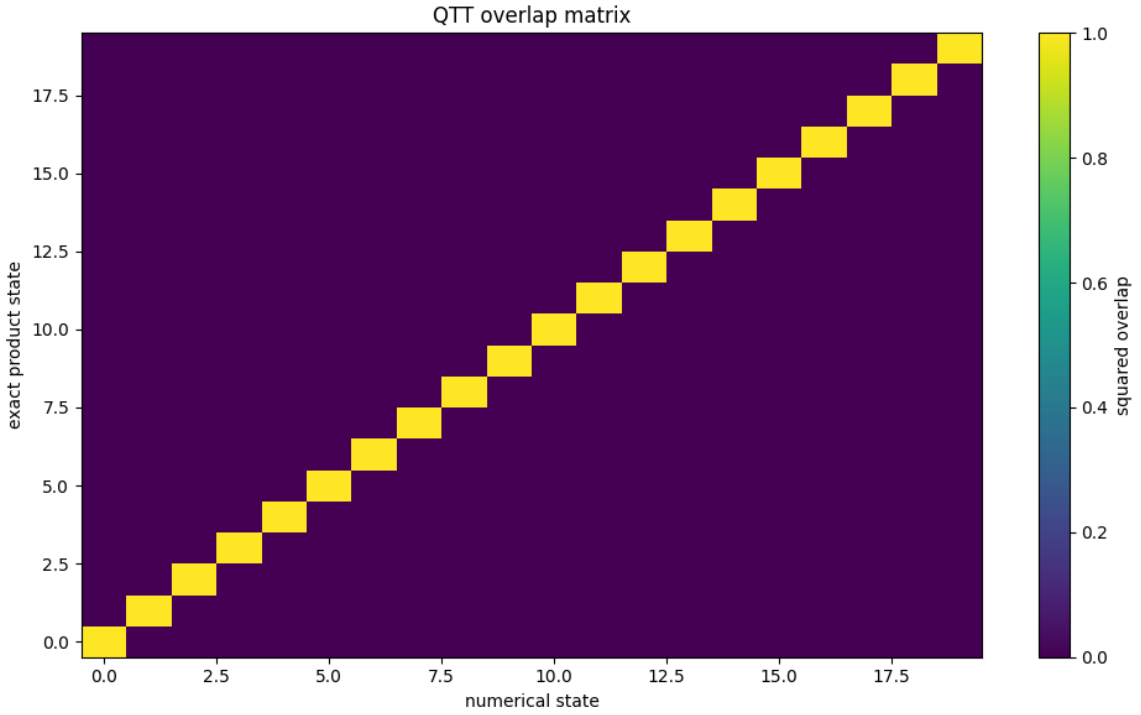

Figure 8: Squared-overlap matrix between exact product states and QTT eigenstates. The nearly perfect diagonal confirms one-to-one state recovery across the 20-state block.

15 What the results demonstrate

The benchmark isolates the QTT machinery in a setting where an independent analytical solution is available. Several conclusions follow directly from the saved outputs.

1. **The physical-core Fourier transform is numerically reversible.** A round-trip error of

7.721×10^{-15} shows that the unfold–FFT–refold construction is not introducing a measurable error at this scale beyond floating-point roundoff.

2. **The structured Hamiltonian is highly compressible.** The full three-dimensional potential and kinetic diagonals have maximum QTT ranks only 4 and 3, respectively.

3. **The one-dimensional grid is already converged for the low levels.** The FGH/analytical errors in Table 4 are 10^{-14}–10^{-12}.

4. **Block imaginary-time propagation preserves the target subspace.** Because the uncoupled FGH seeds are already excellent, the Ritz energies change only in the ninth or later decimal places while the rank remains 15.

5. **The final state identities are correct, not merely the energies.** The overlap matrix is essentially the identity and the minimum matched fidelity is 0.999999999755.

6. **Energy agreement alone is not the whole convergence story.** The largest energy error is only 5.336×10^{-9}, while the largest residual norm is 3.798×10^{-4}. This illustrates why the notebook reports both scalar energy accuracy and a vector residual.

16 How the code generalizes to coupled systems

The notebook has three potential branches.

`none` The potential and its imaginary-time propagator both factorize across coordinates. This is the clean analytical benchmark.

`morse_pair` The potential itself is still assembled exactly from one-dimensional QTT factors using $\kappa_{ij} s_i s_j$, but $e^{-\Delta\tau V/2}$ does not factorize. The scalar exponential is therefore applied to `VQTT` using TT-cross.

`custom` The many-body potential itself is constructed by TT-cross from selected binary-grid samples, and the potential propagator is likewise generated through TT-cross.

For coupled systems, analytical product states are no longer exact eigenvectors. The code consequently changes its interpretation step: it projects each numerical state onto a low-energy uncoupled product basis, reports the dominant product configuration, the captured weight, and an effective participation measure. Rank growth then becomes a physical/numerical indicator of increased intermode correlation.

17 The dormant two-dimensional heat-map branch

If the configuration is changed to

```
N_OSC = 2
COUPLING_MODEL = "none"
```

the final plotting cell reconstructs each selected two-dimensional QTT state and its analytical product reference. Because eigenvectors are defined only up to a global phase, it first computes

$$\phi = \arg\langle \Psi^{\text{exact}} | \Psi^{\text{QTT}} \rangle, \qquad \Psi^{\text{QTT}}_{\text{aligned}} = e^{-i\phi} \Psi^{\text{QTT}}, \tag{42}$$

and then plots a 2×2 panel containing the real parts and probability densities of the numerical and exact states on common color scales.

No such figures are included here because the supplied executed notebook uses three modes and therefore did not generate them. This is why the one-coordinate marginals are the appropriate state visualization for the saved run.

18 Scaling and practical limitations

With L_d binary digits per coordinate, a dense state contains

$$N_{\text{dense}} = 2^{\sum_d L_d} \tag{43}$$

amplitudes. A binary QTT with characteristic rank r stores approximately

$$O\left(2r^2 \sum_d L_d\right) \tag{44}$$

core coefficients, although actual ranks vary along the train.

The compression is useful only while the wavefunctions, operators, and Fourier-transformed states remain low rank. Strong coupling, delocalized or near-dissociation states, and complicated many-body correlations can drive the ranks upward. Accordingly, a meaningful convergence study must monitor *both* physical observables and tensor ranks.

The notebook recommends the following progression:

1. Verify the uncoupled spectrum and state fidelities against the analytical Morse solution.
2. Increase L_d until the states of interest are insensitive to the grid.
3. Tighten `QTT_EPS` and `QFT_EPS`, and increase `QTT_RMAX`, until energies, residuals, and ranks stabilize.
4. Introduce weak coupling and inspect both energy shifts and rank growth.
5. Increase the coupling gradually and track product-basis weights and one-coordinate marginals.
6. Use dense validation only for small cases; it is not part of the scalable algorithm.

19 Code architecture: how the notebook pieces fit together

Table 7: Functional map of the notebook.

Cell(s)	Main objects/functions	Purpose
2	Manual TT-SVD example	Demonstrates binary tensorization, SVD, TT cores, and exact reconstruction.
7	`qtt_from_dense`, `qtt_round`, `qtt_add`, `qtt_hadamard`, `qtt_inner`	Implements the core low-rank tensor algebra, including complex-valued support and explicit rank truncation.
9	`qtt_qft` and helper routines	Unfolds each binary coordinate block, performs column-wise FFTs, refolds by SVD, and rounds.
11, 24	Block overlap and Rayleigh–Ritz helpers	Performs QTT-native Löwdin orthogonalization and diagonalization in the small state subspace.
13	Grid and physical parameters	Defines the number of oscillators, Morse parameters, binary grids, momenta, and quadrature volume.
16	`VQTT`	Builds the potential by structured QTT algebra for built-in Morse models or TT-cross for a custom PES.

Cell(s)	Main objects/functions	Purpose
18	`Ttot_qtt`	Builds the separable momentum-space kinetic diagonal.
20	QFT validation	Tests QFT followed by inverse QFT without dense reconstruction.
22	`apply_H_qtt`, `qtt_energy`	Applies the full Hamiltonian and evaluates Rayleigh quotients.
26–30	Analytical levels, exact QTTs, local FGH seeds	Creates independent reference data and high-quality numerical initial states.
32–34	Propagators and block imaginary time	Executes Strang splitting and Rayleigh–Ritz after every propagation step.
36	Dense validation	Optional small-grid cross-check; skipped in the saved 128^3 run.
38	Final state analysis	Computes overlaps, Hungarian matching, residuals, ranks, and coefficient counts.
40	Marginals	Contracts $\|\Psi\|^2$ to obtain one-coordinate distributions without dense reconstruction.
42	Convergence diagnostics	Plots energies, maximum rank, final spectrum, and overlap matrix.
44	2D heat maps	Runs only for uncoupled two-mode calculations; skipped in the saved three-mode run.

20 Recommended modifications for further study

1. **Grid convergence:** compare $L = 5, 6, 7$ and track the maximum energy error, minimum fidelity, and residual norm.
2. **Compression convergence:** repeat with `QTT_EPS`$= 10^{-8}, 10^{-10}, 10^{-12}$ and monitor the accuracy/rank tradeoff.
3. **Rank-cap convergence:** vary `QTT_RMAX` through 32, 64, and 96.
4. **Propagation convergence:** increase the number of imaginary-time steps and reduce the smallest $\Delta\tau$ to test whether residuals decrease further.
5. **Coupling scan:** activate `morse_pair`, vary the coupling strength, and track energy shifts, product-state mixing, and rank growth.
6. **Dimensional scaling:** repeat for $n = 2, 3, 4$ at fixed L and compare nominal dense size against actual QTT coefficient counts.
7. **Custom PES:** replace `CUSTOM_COUPLING(X)` with a genuine three-mode interaction and inspect TT-cross ranks and validation errors.

21 Summary

The notebook implements a fully QTT-native eigensolver architecture in which the multidimensional wavefunctions remain tensor trains throughout the scalable path. Binary tensorization compresses each coordinate, a physical-core FFT moves efficiently between coordinate and momentum representations, structured QTT algebra builds the Morse Hamiltonian, and block imaginary-time propagation combined

with Löwdin orthogonalization and Rayleigh–Ritz extracts multiple low-energy states. The analytical Morse solution then supplies a stringent independent benchmark.

In the executed three-mode, 20-state run, the QFT is reversible to 7.721×10^{-15} relative error, the structured potential and kinetic terms remain at maximum ranks 4 and 3, propagation maintains a maximum state rank of 15, and the final 20-state spectrum agrees with the analytical product spectrum to a maximum absolute energy error of 5.336×10^{-9} with minimum fidelity 0.999999999755. The results therefore demonstrate both the numerical correctness of the implemented QTT machinery and the importance of monitoring residuals and rank growth in addition to energies.

A Complete code extracted from the notebook

The following listing is an exact concatenation of all code cells in the supplied notebook, with cell-number comments added only as separators. It is also provided as `code/notebook_code_all.py` in the accompanying bundle.

```

# ===== Notebook cell 2 =====

# A hands-on, from-scratch TT-SVD of a length-4 vector.
# This uses plain numpy so every step is visible, before we hand things
# off to tntorch's Tensor objects for the real calculation below.

import numpy as np

v = np.array([1.0, 2.0, 3.0, 4.0])   # the vector we want to compress
L = 2                                  # number of bits, N = 2**L = 4

# Reshape (1, N) -> (2, 2): first axis = bit 1 (MSB), second axis = bit 2 (LSB)
M = v.reshape(2, 2)
print("Reshaped as a matrix M[b1, b2]:\n", M)

# Standard SVD. The rank of M (number of nonzero singular values) is the
# TT bond dimension r1 needed between core 1 and core 2.
U, S, Vt = np.linalg.svd(M, full_matrices=False)
print("\nSingular values:", S)

r1 = int(np.sum(S > 1e-12))
print("Bond dimension r1 =", r1)

# Core 1: shape (r0=1, n1=2, r1) -- here U already has shape (2, r1)
core1 = U[:, :r1].reshape(1, 2, r1)

# Core 2: shape (r1, n2=2, r2=1) -- fold the singular values into core 2
core2 = (np.diag(S[:r1]) @ Vt[:r1, :]).reshape(r1, 2, 1)

print("\ncore1 shape:", core1.shape)
print("core2 shape:", core2.shape)

# Reconstruct: contract core1 with core2 over the shared bond index.
rebuilt = np.tensordot(core1, core2, axes=([2], [0]))   # -> shape (1, 2, 2, 1)
rebuilt = rebuilt.reshape(4)
print("\nReconstructed vector:", rebuilt)
print("Matches original:", np.allclose(rebuilt, v))




# ===== Notebook cell 4 =====
```

```
# Install tntorch, the tensor-train library used throughout this tutorial.
# In Google Colab this cell is normally all that is needed.
!pip -q install tntorch



# ===== Notebook cell 5 =====

# Imports

import math
import heapq
from pathlib import Path

import numpy as np
import pandas as pd
import torch
import tntorch as tn
import matplotlib.pyplot as plt

from scipy.special import eval_genlaguerre
from scipy.optimize import linear_sum_assignment
from IPython.display import display

torch.set_default_dtype(torch.float64)
device = torch.device("cpu")

OUTPUT_DIR = Path("/content") if Path("/content").exists() else Path(".")

print("PyTorch :", torch.__version__)
print("Device  :", device)
print("Output  :", OUTPUT_DIR.resolve())
print("tntorch : imported successfully")



# ===== Notebook cell 7 =====

# Cell 3 -- Generic TT/QTT routines
#
# Everything below operates on tntorch's tn.Tensor, which is just a
# Python list of cores (torch.Tensor objects), each of shape
# (r_left, 2, r_right). We reimplement TT-SVD, rounding, addition,
# Hadamard product, and inner product by hand (rather than calling
# tntorch's built-ins) so that complex dtypes -- needed once we start
# Fourier-transforming wavefunctions -- are handled correctly throughout.

QTT_EPS = 1.0e-12    # rounding tolerance for algebra results (add, hadamard, ...)
QFT_EPS = 2.0e-13    # rounding tolerance specifically after Fourier transforms
QTT_RMAX = 96         # hard ceiling on any TT bond dimension


def _rank_from_singular_values(S, delta=None, rmax=None):
    """
    Given a vector of singular values S (sorted descending, as returned by
    torch.linalg.svd), decide how many to keep.

    The rule mirrors the standard TT-SVD/TT-rounding truncation criterion:
    keep the smallest r such that the "discarded tail" satisfies
        || S[r:] ||_2 <= delta
```

```
    i.e. we keep just enough singular values that the Frobenius-norm error
    introduced by throwing away the rest is below the target tolerance.
    `rmax` additionally hard-caps r regardless of delta.
    """
    m = int(S.numel())

    if m == 0:
        return 1

    r = m

    if delta is not None:
        delta = float(delta)

        for candidate in range(1, m + 1):

            if candidate == m:
                tail = 0.0
            else:
                # Norm of the singular values we would be discarding if we
                # kept only the first `candidate` of them.
                tail = float(
                    torch.linalg.vector_norm(
                        S[candidate:]
                    ).detach().cpu()
                )

            if tail <= delta:
                r = candidate
                break

    if rmax is not None:
        r = min(
            r,
            int(rmax)
        )

    return max(1, r)


def qtt_from_dense(
    tensor,
    bits_per_dim,
    eps=QTT_EPS,
    rmax=QTT_RMAX
):
    """
    TT-SVD of a dense physical tensor into a binary QTT.

    bits_per_dim = [L1, L2, ...]
    means physical dimension d has size 2**Ld.

    This is exactly the "reshape into a matrix, SVD, keep U, fold the
    remainder into the next core, repeat" procedure you saw by hand in the
    Primer's 4-element example -- just generalized to `total_bits` bits and
    wrapped with the delta-based truncation from `_rank_from_singular_values`.
    The overall relative-error budget `eps` is spread evenly across the
    total_bits-1 SVD cuts via delta = eps * ||tensor|| / sqrt(total_bits-1),
    which is the standard TT-SVD error-balancing heuristic (Oseledets 2011):
    it guarantees the final relative Frobenius-norm error is <= eps.
    """
```

```
    bits_per_dim = list(
        map(int, bits_per_dim)
    )

    total_bits = sum(
        bits_per_dim
    )

    physical_shape = [
        2**b
        for b in bits_per_dim
    ]

    tensor = torch.as_tensor(
        tensor,
        device=device
    )

    if int(tensor.numel()) != int(
        np.prod(physical_shape)
    ):
        raise ValueError(
            "Dense tensor size does not match bits_per_dim."
        )

    # Reinterpret the dense array as a total_bits-index binary array: this
    # is the quantics reshape described in the Primer (MSB-first per
    # coordinate, with each coordinate's bits stored contiguously in order.
    work = tensor.reshape(
        [2] * total_bits
    ).contiguous()

    if total_bits == 1:
        return tn.Tensor([
            work.reshape(
                1,
                2,
                1
            ).clone()
        ])

    norm0 = float(
        torch.linalg.vector_norm(
            work.reshape(-1)
        ).detach().cpu()
    )

    delta = (
        eps * norm0
        / math.sqrt(total_bits - 1)
        if eps is not None
        else None
    )

    qcores = []
    rleft = 1

    # Sweep left to right: at each bit, fold the current bit into the
    # left "row" index, SVD, keep U as the core, and push (S @ Vh) into
    # the still-unprocessed remainder to be reshaped/truncated next.
    for k in range(
```

```
            total_bits - 1
        ):

            M = work.reshape(
                rleft * 2,
                -1
            )

            U, S, Vh = torch.linalg.svd(
                M,
                full_matrices=False
            )

            r = _rank_from_singular_values(
                S,
                delta=delta,
                rmax=rmax
            )

            U = U[:, :r]
            S = S[:r]
            Vh = Vh[:r, :]

            qcores.append(
                U.reshape(
                    rleft,
                    2,
                    r
                ).contiguous()
            )

            remaining_bits = (
                total_bits
                - k
                - 1
            )

            work = (
                S[:, None]
                * Vh
            ).reshape(
                r,
                *([2] * remaining_bits)
            ).contiguous()

            rleft = r

        # Last core absorbs whatever remains; its right bond dimension is 1
        # by construction (a TT always closes with boundary rank 1).
        qcores.append(
            work.reshape(
                rleft,
                2,
                1
            ).contiguous()
        )

        return tn.Tensor(
            qcores
        )

```

```python

def qtt_to_dense(
    qtt,
    bits_per_dim
):
    """
    The inverse of qtt_from_dense(): contract every core left-to-right
    (summing over shared bond indices) to rebuild the dense physical
    tensor. Complex-valued cores (which appear once we Fourier-transform
    a wavefunction) are preserved -- no .real is taken here.
    """
    bits_per_dim = list(
        map(int, bits_per_dim)
    )

    cores = qtt.cores

    if len(cores) != sum(
        bits_per_dim
    ):
        raise ValueError(
            "Number of QTT cores does not match bits_per_dim."
        )

    for core in cores:
        if (
            core.dim() != 3
            or core.shape[1] != 2
        ):
            raise ValueError(
                "Expected pure binary TT cores."
            )

    # Start from the (trivial) left boundary rank-1 index of the first core.
    acc = cores[0][
        0,
        :,
        :
    ]

    # Contract in the remaining cores one at a time over the shared bond.
    for core in cores[1:]:

        acc = torch.tensordot(
            acc,
            core,
            dims=(
                [-1],
                [0]
            )
        )

    if acc.shape[-1] != 1:
        raise ValueError(
            "Final TT rank is not one."
        )

    binary_tensor = (
        acc[..., 0]
    )

```

```
    physical_shape = [
        2**b
        for b in bits_per_dim
    ]

    return binary_tensor.reshape(
        physical_shape
    )


def qtt_ranks(qtt):
    """Return the full list of bond dimensions [r0=1, r1, r2, ..., rD=1]."""
    return [
        int(
            qtt.cores[0].shape[0]
        )
    ] + [
        int(core.shape[-1])
        for core in qtt.cores
    ]


def qtt_numcoef(qtt):
    """Total number of stored floating-point numbers across all cores --
    the actual memory footprint of the QTT, to compare against the full dense grid."""
    return sum(
        int(core.numel())
        for core in qtt.cores
    )


def qtt_summary(
    name,
    qtt
):
    """Pretty-print a one-line rank/size summary; used throughout the
    notebook to keep an eye on how compressed each QTT is."""
    ranks = qtt_ranks(
        qtt
    )

    print(
        f"{name:18s} "
        f"max rank={max(ranks):3d}   "
        f"coefficients={qtt_numcoef(qtt):8d}   "
        f"ranks={ranks}"
    )


def qtt_round(
    qtt,
    eps=QTT_EPS,
    rmax=QTT_RMAX
):
    """
    Complex-safe TT rounding using QR + SVD.

    Rounding re-expresses the SAME tensor at (at most) the given error
    tolerance, using the smallest possible ranks. It is a two-pass sweep:

      1. Left-orthogonalization sweep (QR, left to right): reshapes each
```

```
        core into "tall" matrices and QR-factorizes them so that every
        core except the last becomes left-orthogonal (its columns are
        orthonormal). The leftover R factor is absorbed into the next
        core. This does not change the tensor or its rank -- it just
        puts it into a canonical form that makes the next step's error
        estimate meaningful.

     2. Right-to-left SVD truncation sweep: now sweep from the right,
        SVD each core, discard the small singular values (per the same
        delta budget used in TT-SVD), and fold the U*S factor into the
        core to its left. This step is where the actual compression
        happens.

    This exact two-sweep structure is the standard TT-rounding algorithm
    (Oseledets 2011); reimplementing it here (rather than calling
    tntorch's rounding) lets us keep everything correct for complex dtype.
    """
    cores = [
        core.clone()
        for core in qtt.cores
    ]

    d = len(
        cores
    )

    if d <= 1:
        return tn.Tensor(
            cores
        )

    # --- Pass 1: left orthogonalization ------------------------------
    for k in range(
        d - 1
    ):

        r0, n, r1 = (
            cores[k].shape
        )

        M = cores[k].reshape(
            r0 * n,
            r1
        )

        Q, R = torch.linalg.qr(
            M,
            mode="reduced"
        )

        rnew = Q.shape[1]

        cores[k] = Q.reshape(
            r0,
            n,
            rnew
        ).contiguous()

        cores[k + 1] = torch.tensordot(
            R,
            cores[k + 1],
```

```
            dims=(
                [1],
                [0]
            )
        ).contiguous()

    tensor_norm = float(
        torch.linalg.vector_norm(
            cores[-1].reshape(-1)
        ).detach().cpu()
    )

    delta = (
        eps * tensor_norm
        / math.sqrt(d - 1)
        if eps is not None
        else None
    )

    # --- Pass 2: right-to-left SVD truncation ------------------------
    for k in range(
        d - 1,
        0,
        -1
    ):

        r0, n, r1 = (
            cores[k].shape
        )

        M = cores[k].reshape(
            r0,
            n * r1
        )

        U, S, Vh = torch.linalg.svd(
            M,
            full_matrices=False
        )

        r = _rank_from_singular_values(
            S,
            delta=delta,
            rmax=rmax
        )

        U = U[:, :r]
        S = S[:r]
        Vh = Vh[:r, :]

        cores[k] = Vh.reshape(
            r,
            n,
            r1
        ).contiguous()

        US = (
            U
            * S[None, :]
        )

```

```
            cores[k - 1] = torch.tensordot(
                cores[k - 1],
                US,
                dims=(
                    [2],
                    [0]
                )
            ).contiguous()

        return tn.Tensor(
            cores
        )


    def qtt_hadamard(
        a,
        b,
        eps=QTT_EPS,
        rmax=QTT_RMAX
    ):
        """
        Pointwise (Hadamard/elementwise) product of two QTTs, c(i) = a(i)*b(i).

        Core-by-core, the elementwise product of two TTs is built by taking the
        "outer product" of each pair of cores along both bond directions:
        if A has shape (rA_left, 2, rA_right) and B has shape (rB_left, 2, rB_right),
        the combined core C has shape (rA_left*rB_left, 2, rA_right*rB_right),
        with C[(a,b), i, (a',b')] = A[a,i,a'] * B[b,i,b']. Physically this means
        the Hadamard-product rank is (at most) the PRODUCT of the input ranks --
        which is why every call here is immediately followed by qtt_round() to
        bring the rank back down to what the actual accuracy tolerance needs.
        """
        if len(a.cores) != len(
            b.cores
        ):
            raise ValueError(
                "QTT dimensions do not match."
            )

        out = []

        for A0, B0 in zip(
            a.cores,
            b.cores
        ):

            if A0.shape[1] != B0.shape[1]:
                raise ValueError(
                    "Physical mode sizes do not match."
                )

            dtype = torch.promote_types(
                A0.dtype,
                B0.dtype
            )

            A = A0.to(
                dtype
            )

            B = B0.to(
```

```
            dtype
        )

        # Outer product along both bond legs, elementwise along the
        # shared physical (bit) index.
        C = (
            A[
                :,
                None,
                :,
                :,
                None
            ]
            * B[
                None,
                :,
                :,
                None,
                :
            ]
        )

        C = C.reshape(
            A.shape[0]
            * B.shape[0],
            A.shape[1],
            A.shape[2]
            * B.shape[2]
        )

        out.append(
            C.contiguous()
        )

    return qtt_round(
        tn.Tensor(out),
        eps=eps,
        rmax=rmax
    )


def qtt_add(
    a,
    b,
    eps=QTT_EPS,
    rmax=QTT_RMAX
):
    """
    Sum of two QTTs, c(i) = a(i) + b(i), by TT block concatenation.

    The standard exact-TT-addition trick: stack a's and b's cores
    block-diagonally along the bond dimensions (except at the very first
    and very last core, where you concatenate along the single free bond,
    since the boundary ranks must stay 1). Like Hadamard product, exact
    addition ADDS the ranks together (rather than multiplying), so this
    is also always followed by qtt_round().
    """
    if len(a.cores) != len(
        b.cores
    ):
        raise ValueError(
```

```
            "QTT dimensions do not match."
        )

    d = len(
        a.cores
    )

    if d == 1:

        dtype = torch.promote_types(
            a.cores[0].dtype,
            b.cores[0].dtype
        )

        return tn.Tensor([
            a.cores[0].to(dtype)
            + b.cores[0].to(dtype)
        ])

    cores = []

    for k, (A0, B0) in enumerate(
        zip(
            a.cores,
            b.cores
        )
    ):

        dtype = torch.promote_types(
            A0.dtype,
            B0.dtype
        )

        A = A0.to(
            dtype
        )

        B = B0.to(
            dtype
        )

        if k == 0:
            # First core: concatenate along the right bond only
            # (left bond must stay 1).
            C = torch.cat(
                [A, B],
                dim=2
            )

        elif k == d - 1:
            # Last core: concatenate along the left bond only
            # (right bond must stay 1).
            C = torch.cat(
                [A, B],
                dim=0
            )

        else:
            # Interior cores: block-diagonal placement.
            C = torch.zeros(
                A.shape[0]
```

```
                    + B.shape[0],
                    A.shape[1],
                    A.shape[2]
                    + B.shape[2],
                    dtype=dtype,
                    device=A.device
                )

                C[
                    :A.shape[0],
                    :,
                    :A.shape[2]
                ] = A

                C[
                    A.shape[0]:,
                    :,
                    A.shape[2]:
                ] = B

            cores.append(
                C.contiguous()
            )

        return qtt_round(
            tn.Tensor(cores),
            eps=eps,
            rmax=rmax
        )


def qtt_inner(
    a,
    b
):
    """
    Algebraic TT inner product <a|b> = sum_i conj(a_i) * b_i.

    Contract the two TTs against each other core by core, carrying a
    small (rA x rB) "environment" matrix between sites -- this is the
    standard TT/MPS inner-product contraction, with cost linear in the
    number of cores rather than exponential in the number of physical
    indices, because we never form the dense tensors.
    """
    if len(a.cores) != len(
        b.cores
    ):
        raise ValueError(
            "QTT dimensions do not match."
        )

    dtype = torch.promote_types(
        a.cores[0].dtype,
        b.cores[0].dtype
    )

    env = torch.ones(
        (1, 1),
        dtype=dtype,
        device=a.cores[0].device
    )
```

```
    for A0, B0 in zip(
        a.cores,
        b.cores
    ):

        A = A0.to(
            dtype
        )

        B = B0.to(
            dtype
        )

        # env[a,b], A*[a,i,c], B[b,i,d] -> new env[c,d]
        env = torch.einsum(
            "ab,aic,bid->cd",
            env,
            A.conj(),
            B
        )

    return env.squeeze()


def qtt_tensor_product(
    a,
    b
):
    """
    Tensor product of two QTTs describing two INDEPENDENT coordinate
    groups, e.g. combining QTTs defined on two independent coordinate groups
    into one QTT over the concatenated binary modes.

    Because a TT's right boundary rank and the next TT's left boundary
    rank are both 1, simply concatenating the two core lists produces
    exactly the product f(group1)*g(group2) in QTT form -- no rounding is needed.
    """
    if (
        a.cores[-1].shape[-1] != 1
        or b.cores[0].shape[0] != 1
    ):
        raise ValueError(
            "Expected boundary TT ranks equal to one."
        )

    return tn.Tensor(
        [
            core.clone()
            for core in (
                list(a.cores)
                + list(b.cores)
            )
        ]
    )



# ===== Notebook cell 9 =====

# Cell 4 -- Unfold -> column FFT -> fold QFT
```

```
837
838 def qtt_unfold_to_physical_tt(
839     qtt,
840     bits_per_dim
841 ):
842     """
843     Step 1 of the QFT recipe: contract each coordinate's group of binary
844     QTT cores into one "physical" TT core of shape (R_left, 2**Ld, R_right).
845
846     This is literally the same recursive tensordot-and-reshape merge you'd
847     use to undo one level of the quantics bit-splitting -- we're re-forming
848     the dense axis for coordinate d, but leaving the OTHER coordinates'
849     cores untouched, so the intermediate object is still compact whenever
850     the cross-coordinate TT rank stays small.
851     """
852     bits_per_dim = list(
853         map(int, bits_per_dim)
854     )
855
856     if sum(bits_per_dim) != len(
857         qtt.cores
858     ):
859         raise ValueError(
860             "sum(bits_per_dim) must equal the number of QTT cores."
861         )
862
863     physical_cores = []
864     offset = 0
865
866     for nbits in bits_per_dim:
867
868         group = qtt.cores[
869             offset:
870             offset + nbits
871         ]
872
873         merged = group[0].clone()
874
875         for next_core in group[1:]:
876
877             merged = torch.tensordot(
878                 merged,
879                 next_core,
880                 dims=(
881                     [2],
882                     [0]
883                 )
884             )
885
886             merged = merged.reshape(
887                 merged.shape[0],
888                 -1,
889                 merged.shape[-1]
890             ).contiguous()
891
892         if merged.shape[1] != 2**nbits:
893             raise RuntimeError(
894                 "Unexpected physical size while unfolding QTT."
895             )
896
897         physical_cores.append(
```

```
            merged
        )

        offset += nbits

    return physical_cores


def fft_physical_tt_cores(
    physical_cores,
    inverse=False,
    norm="ortho"
):
    """
    Step 2 of the QFT recipe: FFT every "column" of every physical TT core
    along its physical (length-N) mode.

    A core has shape (rleft, N, rright). We want to Fourier-transform only
    the N axis, independently for each of the rleft*rright choices of bond
    indices -- so we permute N to the front, flatten the bond indices into
    one batch axis, call torch.fft.fft along dim=0, then unflatten. Using
    norm="ortho" keeps the transform unitary, matching torch.fft.fftn's
    "ortho" convention used for validation in Cell 7 and in apply_H_dense().
    """
    transformed = []

    fft_fun = (
        torch.fft.ifft
        if inverse
        else torch.fft.fft
    )

    for core in physical_cores:

        rleft, Ndim, rright = (
            core.shape
        )

        complex_dtype = (
            torch.complex128
            if core.dtype in (
                torch.float64,
                torch.complex128
            )
            else torch.complex64
        )

        core_c = core.to(
            complex_dtype
        )

        # (rL, N, rR) -> (N, rL*rR): every column is independently
        # transformed.
        columns = (
            core_c
            .permute(
                1,
                0,
                2
            )
            .reshape(
```

```
                Ndim,
                rleft * rright
            )
        )

        columns_k = fft_fun(
            columns,
            dim=0,
            norm=norm
        )

        core_k = (
            columns_k
            .reshape(
                Ndim,
                rleft,
                rright
            )
            .permute(
                1,
                0,
                2
            )
            .contiguous()
        )

        transformed.append(
            core_k
        )

    return transformed


def _split_physical_core_exact(
    core,
    nbits
):
    """
    Step 3a: split ONE physical TT core of shape (R_left, 2**nbits, R_right)
    back into `nbits` binary cores, via the same recursive SVD used in
    qtt_from_dense(), but WITHOUT truncation here (full SVD rank is kept --
    the eventual compression happens afterward, once in
    physical_tt_to_qtt(), by a single qtt_round() call over the whole QTT).
    Keeping this step exact avoids compounding rounding error coordinate
    by coordinate.
    """
    rleft_external, Ndim, rright_external = (
        core.shape
    )

    if Ndim != 2**nbits:
        raise ValueError(
            "Physical dimension is incompatible with nbits."
        )

    if nbits == 1:
        return [
            core.reshape(
                rleft_external,
                2,
                rright_external
```

```
            ).contiguous()
        ]

    work = core.reshape(
        rleft_external,
        *([2] * nbits),
        rright_external
    ).contiguous()

    qcores = []
    rleft = (
        rleft_external
    )

    for k in range(
        nbits - 1
    ):

        M = work.reshape(
            rleft * 2,
            -1
        )

        U, S, Vh = torch.linalg.svd(
            M,
            full_matrices=False
        )

        # No truncation by delta/rmax here -- keep the full-rank SVD so
        # this split step is numerically exact.
        r = int(
            S.numel()
        )

        qcores.append(
            U.reshape(
                rleft,
                2,
                r
            ).contiguous()
        )

        remaining_bits = (
            nbits
            - k
            - 1
        )

        work = (
            S[:, None]
            * Vh
        ).reshape(
            r,
            *([2] * remaining_bits),
            rright_external
        ).contiguous()

        rleft = r

    qcores.append(
        work.reshape(
```

```
            rleft,
            2,
            rright_external
        ).contiguous()
    )

    return qcores


def physical_tt_to_qtt(
    physical_cores,
    bits_per_dim,
    eps=QFT_EPS,
    rmax=QTT_RMAX
):
    """Step 3b: fold every coordinate's physical core back to binary QTT
    cores, concatenate them all, and finally round the WHOLE QTT once
    (this is where the FFT-induced rank growth actually gets truncated)."""
    bits_per_dim = list(
        map(int, bits_per_dim)
    )

    if len(physical_cores) != len(
        bits_per_dim
    ):
        raise ValueError(
            "Need one physical TT core per physical dimension."
        )

    qcores = []

    for core, nbits in zip(
        physical_cores,
        bits_per_dim
    ):

        qcores.extend(
            _split_physical_core_exact(
                core,
                nbits
            )
        )

    return qtt_round(
        tn.Tensor(qcores),
        eps=eps,
        rmax=rmax
    )


def qtt_qft(
    qtt,
    bits_per_dim,
    inverse=False,
    eps=QFT_EPS,
    rmax=QTT_RMAX,
    norm="ortho"
):
    """
    Multidimensional QFT/IQFT of a QTT wavefunction, tying the three steps
    together:
```

```
        QTT  --(unfold)-->  one physical TT core per coordinate
             --(FFT each core's columns)-->  transformed physical cores
             --(fold + round)-->  binary QTT

    Set inverse=True for the inverse transform (used to come back from
    momentum space to coordinate space after applying the kinetic
    propagator/operator).
    """
    physical_cores = qtt_unfold_to_physical_tt(
        qtt,
        bits_per_dim
    )

    transformed = fft_physical_tt_cores(
        physical_cores,
        inverse=inverse,
        norm=norm
    )

    return physical_tt_to_qtt(
        transformed,
        bits_per_dim,
        eps=eps,
        rmax=rmax
    )




# ===== Notebook cell 11 =====

# Extra QTT helpers used by the n-dimensional block solver

def qtt_scale(qtt, scalar):
    """Multiply a QTT by a scalar without changing its TT ranks."""
    cores = [core.clone() for core in qtt.cores]
    scalar_t = torch.as_tensor(scalar, device=cores[0].device)
    dtype = torch.promote_types(cores[0].dtype, scalar_t.dtype)
    cores = [core.to(dtype) for core in cores]
    cores[0] = cores[0] * scalar_t.to(dtype)
    return tn.Tensor(cores)


def qtt_conjugate(qtt):
    """Complex conjugate of a QTT."""
    return tn.Tensor([core.conj().clone() for core in qtt.cores])


def qtt_linear_combination(states, coeffs, eps=QTT_EPS, rmax=QTT_RMAX):
    """Return sum_i coeffs[i] * states[i], rounding after additions."""
    if len(states) != len(coeffs):
        raise ValueError("states and coeffs must have the same length")

    active = []
    for i, c in enumerate(coeffs):
        c_abs = float(torch.abs(torch.as_tensor(c)).detach().cpu())
        if c_abs > 1.0e-15:
            active.append((i, c))

    if not active:
```

```
        return qtt_scale(states[0], 0.0)

    i0, c0 = active[0]
    out = qtt_scale(states[i0], c0)

    for i, c in active[1:]:
        out = qtt_add(
            out,
            qtt_scale(states[i], c),
            eps=eps,
            rmax=rmax
        )

    return qtt_round(out, eps=eps, rmax=rmax)


def tensor_product_many(qtts):
    """Tensor product of a list of independent-coordinate QTTs."""
    if len(qtts) == 0:
        raise ValueError("Need at least one QTT")
    out = qtts[0]
    for qtt in qtts[1:]:
        out = qtt_tensor_product(out, qtt)
    return out


def sum_qtts(qtts, eps=QTT_EPS, rmax=QTT_RMAX):
    """Sum a list of QTTs."""
    if len(qtts) == 0:
        raise ValueError("Need at least one QTT")
    out = qtts[0]
    for qtt in qtts[1:]:
        out = qtt_add(out, qtt, eps=eps, rmax=rmax)
    return qtt_round(out, eps=eps, rmax=rmax)


def qtt_relative_error(a, b):
    """Algebraic relative 2-norm ||a-b||/||a|| (grid weights cancel)."""
    diff = qtt_add(a, qtt_scale(b, -1.0), eps=QTT_EPS, rmax=QTT_RMAX)
    num = torch.sqrt(torch.clamp(qtt_inner(diff, diff).real, min=0.0))
    den = torch.sqrt(torch.clamp(qtt_inner(a, a).real, min=1.0e-300))
    return float((num / den).detach().cpu())



# ===== Notebook cell 13 =====

# Physical parameters and n binary Fourier grids

hbar = 1.0

# ------------------------------------------------------------
# 1. Choose the number of oscillators
# ------------------------------------------------------------
# The code remains n-dimensional, but n=2 is the default because it gives
# a stringent benchmark against analytical 2D product states and permits
# full wavefunction heat maps.
N_OSC = 3
NSTATES = 20

# Accuracy-oriented default grid:
```

```
#   L=7  ->  N=128 points per coordinate
# The wide box makes the low Morse bound states essentially zero at the
# periodic FFT boundaries, strongly reducing finite-box/periodicity error.
def make_default_oscillators(n, L=7):
    oscillators = []
    for d in range(n):
        oscillators.append({
            "mass": 1.0 + 0.05 * d,
            "De":   20.0 - 1.5 * d,
            "a":    0.80 - 0.03 * d,
            "xe":   0.0,
            "L":    L,
            "xmin": -6.0,
            "xmax": 18.0,
        })
    return oscillators

OSCILLATORS = make_default_oscillators(N_OSC, L=7)

# Example manual customization:
# OSCILLATORS[1].update({"mass": 1.2, "De": 15.0, "a": 0.65})

# ------------------------------------------------------------
# 2. Choose uncoupled or coupled dynamics
# ------------------------------------------------------------
# Keep "none" for the analytical benchmark. Switch to "morse_pair" to
# study coupled modes, or "custom" for an arbitrary many-body potential.
COUPLING_MODEL = "none"       # "none", "morse_pair", or "custom"
COUPLING_STRENGTH = 0.60       # used to build nearest-neighbor KAPPA

KAPPA = np.zeros((N_OSC, N_OSC), dtype=float)
for d in range(N_OSC - 1):
    KAPPA[d, d + 1] = COUPLING_STRENGTH
    KAPPA[d + 1, d] = COUPLING_STRENGTH

# For a fully custom coupling set COUPLING_MODEL="custom" and replace
# this function. X has shape (nsamples, N_OSC) and the return value must
# have shape (nsamples,).
def CUSTOM_COUPLING(X):
    return torch.zeros(X.shape[0], dtype=X.dtype, device=X.device)

if len(OSCILLATORS) != N_OSC:
    raise ValueError("len(OSCILLATORS) must equal N_OSC")
if KAPPA.shape != (N_OSC, N_OSC):
    raise ValueError("KAPPA must have shape (N_OSC, N_OSC)")
if not np.allclose(KAPPA, KAPPA.T):
    raise ValueError("KAPPA must be symmetric")
if not np.allclose(np.diag(KAPPA), 0.0):
    raise ValueError("The diagonal of KAPPA should be zero")

# ------------------------------------------------------------
# 3. Build coordinate and momentum grids
# ------------------------------------------------------------
BITS_PER_DIM = [int(p["L"]) for p in OSCILLATORS]
GRID_SHAPE = [2**L for L in BITS_PER_DIM]
TOTAL_BITS = sum(BITS_PER_DIM)
TOTAL_POINTS = int(np.prod(GRID_SHAPE))

XGRID, PGRID, DX, T1D, V1D, MORSE_S1D = [], [], [], [], [], []

for d, p in enumerate(OSCILLATORS):
```

```
    N = GRID_SHAPE[d]
    dx = (p["xmax"] - p["xmin"]) / N
    x = p["xmin"] + dx * torch.arange(N, dtype=torch.get_default_dtype(), device=device)
    freq = torch.fft.fftfreq(N, d=dx, device=device)
    momentum = 2.0 * np.pi * hbar * freq

    s = 1.0 - torch.exp(-p["a"] * (x - p["xe"]))
    V = p["De"] * s**2
    T = momentum**2 / (2.0 * p["mass"])

    DX.append(dx)
    XGRID.append(x)
    PGRID.append(momentum)
    MORSE_S1D.append(s)
    V1D.append(V)
    T1D.append(T)

dvol = float(np.prod(DX))

parameter_table = pd.DataFrame([
    {
        "d": d,
        "mass": p["mass"],
        "De": p["De"],
        "a": p["a"],
        "xe": p["xe"],
        "L": p["L"],
        "N": GRID_SHAPE[d],
        "xmin": p["xmin"],
        "xmax": p["xmax"],
        "dx": DX[d],
    }
    for d, p in enumerate(OSCILLATORS)
])

print(f"N_OSC        = {N_OSC}")
print(f"NSTATES      = {NSTATES}")
print(f"Coupling     = {COUPLING_MODEL}")
print(f"BITS_PER_DIM = {BITS_PER_DIM}")
print(f"TOTAL_BITS   = {TOTAL_BITS}")
print(f"Dense points = {TOTAL_POINTS:,}")
print(f"dvol         = {dvol:.6e}")
display(parameter_table)

if COUPLING_MODEL == "morse_pair":
    print("\nPair-coupling matrix KAPPA:")
    display(pd.DataFrame(KAPPA))



# ===== Notebook cell 14 =====

# Sanity plot: one-dimensional Morse wells
#
# ylim(-Dmax, Dmax) is a plotting choice only. The unshifted Morse
# potential itself still has minimum 0 and dissociation limit +D.
Dmax = max(p["De"] for p in OSCILLATORS)

plt.figure(figsize=(9, 5))
for d in range(N_OSC):
    plt.plot(
```

```
        XGRID[d].detach().cpu().numpy(),
        V1D[d].detach().cpu().numpy(),
        label=f"mode {d}"
    )
plt.ylim(0, Dmax)
plt.xlabel("coordinate")
plt.ylabel("Morse potential")
plt.title(f"One-dimensional Morse terms for {N_OSC} modes")
plt.grid(alpha=0.25)
plt.legend()
plt.tight_layout()
plt.show()



# ===== Notebook cell 16 =====

# Build V(x1,...,xn) in QTT form.
# Built-in uncoupled and pairwise Morse models use exact structured QTT
# algebra; arbitrary custom couplings use TT-cross.

CROSS_EPS = 1.0e-11
CROSS_RMAX = QTT_RMAX
CROSS_MAX_ITER = 40
CROSS_VAL_SIZE = 4000
CROSS_SEED = 1234

np.random.seed(CROSS_SEED)
torch.manual_seed(CROSS_SEED)

# Reusable one-dimensional QTT factors.
ONES_1D_QTT = [
    qtt_from_dense(
        torch.ones(GRID_SHAPE[d], dtype=torch.get_default_dtype(), device=device),
        [BITS_PER_DIM[d]], eps=1.0e-14, rmax=QTT_RMAX
    )
    for d in range(N_OSC)
]
V_1D_QTT = [
    qtt_from_dense(V1D[d], [BITS_PER_DIM[d]], eps=QTT_EPS, rmax=QTT_RMAX)
    for d in range(N_OSC)
]
S_1D_QTT = [
    qtt_from_dense(MORSE_S1D[d], [BITS_PER_DIM[d]], eps=QTT_EPS, rmax=QTT_RMAX)
    for d in range(N_OSC)
]


def embedded_product(factors):
    """Tensor product of one 1D QTT factor for every physical coordinate."""
    return tensor_product_many(factors)


def add_terms_qtt(terms):
    if not terms:
        raise ValueError("Need at least one QTT term")
    out = terms[0]
    for term in terms[1:]:
        out = qtt_add(out, term, eps=QTT_EPS, rmax=QTT_RMAX)
    return qtt_round(out, eps=QTT_EPS, rmax=QTT_RMAX)

```

```

def potential_nd(X):
    """Vectorized n-dimensional Morse PES evaluated at X[nsamples, N_OSC]."""
    if X.ndim != 2 or X.shape[1] != N_OSC:
        raise ValueError("X must have shape (nsamples, N_OSC)")

    V = torch.zeros(X.shape[0], dtype=X.dtype, device=X.device)
    s_values = []
    for d, p in enumerate(OSCILLATORS):
        s = 1.0 - torch.exp(-p["a"] * (X[:, d] - p["xe"]))
        s_values.append(s)
        V = V + p["De"] * s**2

    if COUPLING_MODEL == "morse_pair":
        kappa = torch.as_tensor(KAPPA, dtype=X.dtype, device=X.device)
        for i in range(N_OSC):
            for j in range(i + 1, N_OSC):
                if KAPPA[i, j] != 0.0:
                    V = V + kappa[i, j] * s_values[i] * s_values[j]
    elif COUPLING_MODEL == "custom":
        V = V + CUSTOM_COUPLING(X)
    elif COUPLING_MODEL != "none":
        raise ValueError("Unknown COUPLING_MODEL")

    return V


if COUPLING_MODEL in ("none", "morse_pair"):
    terms = []

    # Sum_d V_d(x_d)
    for d in range(N_OSC):
        factors = [V_1D_QTT[q] if q == d else ONES_1D_QTT[q] for q in range(N_OSC)]
        terms.append(embedded_product(factors))

    # Sum_{i<j} kappa_ij s_i s_j
    if COUPLING_MODEL == "morse_pair":
        for i in range(N_OSC):
            for j in range(i + 1, N_OSC):
                if KAPPA[i, j] == 0.0:
                    continue
                factors = []
                for q in range(N_OSC):
                    if q == i or q == j:
                        factors.append(S_1D_QTT[q])
                    else:
                        factors.append(ONES_1D_QTT[q])
                terms.append(qtt_scale(embedded_product(factors), KAPPA[i, j]))

    VQTT = add_terms_qtt(terms)
    VQTT_CROSS_INFO = {"method": "structured", "val_epss": [], "nsamples": 0}
    print("Built VQTT by structured QTT algebra (no TT-cross approximation).")

else:
    # TT-cross path for arbitrary custom many-body potentials.
    BIT_WEIGHTS = [
        2.0 ** torch.arange(L - 1, -1, -1, dtype=torch.get_default_dtype(), device=device)
        for L in BITS_PER_DIM
    ]
    BIT_OFFSETS = np.cumsum([0] + BITS_PER_DIM)

```

```
    def binary_samples_to_coordinates(B):
        B = B.to(dtype=torch.get_default_dtype(), device=device)
        coords = []
        for d in range(N_OSC):
            i0, i1 = BIT_OFFSETS[d], BIT_OFFSETS[d + 1]
            idx = B[:, i0:i1] @ BIT_WEIGHTS[d]
            p = OSCILLATORS[d]
            coords.append(p["xmin"] + DX[d] * idx)
        return torch.stack(coords, dim=1)

    def potential_from_binary_samples(B):
        return potential_nd(binary_samples_to_coordinates(B))

    BIT_DOMAIN = [
        torch.tensor([0.0, 1.0], dtype=torch.get_default_dtype(), device=device)
        for _ in range(TOTAL_BITS)
    ]

    print(f"Building custom VQTT over {TOTAL_BITS} binary modes by TT-cross...")
    VQTT, VQTT_CROSS_INFO = tn.cross(
        function=potential_from_binary_samples,
        domain=BIT_DOMAIN,
        function_arg="matrix",
        eps=CROSS_EPS,
        rmax=CROSS_RMAX,
        max_iter=CROSS_MAX_ITER,
        val_size=CROSS_VAL_SIZE,
        verbose=True,
        return_info=True
    )
    VQTT = qtt_round(VQTT, eps=QTT_EPS, rmax=QTT_RMAX)

Vtot_qtt = VQTT
qtt_summary("VQTT", VQTT)
val_epss = VQTT_CROSS_INFO.get("val_epss", [])
if len(val_epss):
    print(f"TT-cross validation relative error = {val_epss[-1]:.3e}")
if VQTT_CROSS_INFO.get("method") == "structured":
    print("Potential construction error is controlled only by QTT rounding.")
else:
    print("TT-cross samples =", VQTT_CROSS_INFO.get("nsamples", "n/a"))



# ===== Notebook cell 18 =====

# Full kinetic diagonal T(p1,...,pn) in QTT form

T_1D_QTT = [
    qtt_from_dense(T1D[d], [BITS_PER_DIM[d]], eps=1.0e-12, rmax=QTT_RMAX)
    for d in range(N_OSC)
]

def embed_1d_qtt(qtt_1d, target_dim):
    factors = [
        qtt_1d if d == target_dim else ONES_1D_QTT[d]
        for d in range(N_OSC)
    ]
    return tensor_product_many(factors)

kinetic_terms = [
```

```
    embed_1d_qtt(T_1D_QTT[d], d)
    for d in range(N_OSC)
]

Ttot_qtt = sum_qtts(kinetic_terms, eps=QTT_EPS, rmax=QTT_RMAX)
qtt_summary("T total", Ttot_qtt)



# ===== Notebook cell 20 =====

# QFT -> inverse-QFT round-trip validation on an n-dimensional product test state

test_factors = []
for d in range(N_OSC):
    x = XGRID[d]
    p = OSCILLATORS[d]
    width = 1.0 + 0.15 * d
    f = torch.exp(-0.5 * ((x - p["xe"]) / width)**2) * torch.cos((0.7 + 0.1*d) * x)
    test_factors.append(
        qtt_from_dense(f, [BITS_PER_DIM[d]], eps=QTT_EPS, rmax=QTT_RMAX)
    )

test_qtt = tensor_product_many(test_factors)
test_k_qtt = qtt_qft(test_qtt, BITS_PER_DIM, inverse=False, eps=QFT_EPS, rmax=QTT_RMAX)
test_back_qtt = qtt_qft(test_k_qtt, BITS_PER_DIM, inverse=True, eps=QFT_EPS, rmax=QTT_RMAX)

print(f"QFT round-trip relative QTT error = {qtt_relative_error(test_qtt, test_back_qtt):.3e}")
qtt_summary("test x", test_qtt)
qtt_summary("test k", test_k_qtt)



# ===== Notebook cell 22 =====

# QTT-native Hamiltonian action

def apply_H_qtt(psi_qtt, eps=QTT_EPS, qft_eps=QFT_EPS):
    Vpsi = qtt_hadamard(Vtot_qtt, psi_qtt, eps=eps, rmax=QTT_RMAX)

    psi_p = qtt_qft(
        psi_qtt, BITS_PER_DIM, inverse=False,
        eps=qft_eps, rmax=QTT_RMAX
    )

    Tpsi_p = qtt_hadamard(Ttot_qtt, psi_p, eps=eps, rmax=QTT_RMAX)

    Tpsi = qtt_qft(
        Tpsi_p, BITS_PER_DIM, inverse=True,
        eps=qft_eps, rmax=QTT_RMAX
    )

    return qtt_add(Vpsi, Tpsi, eps=eps, rmax=QTT_RMAX)


def qtt_energy(psi_qtt):
    """Rayleigh quotient. The common dvol factor cancels."""
    Hpsi = apply_H_qtt(psi_qtt)
    return (qtt_inner(psi_qtt, Hpsi) / qtt_inner(psi_qtt, psi_qtt)).real


```

```python
def qtt_normalize(psi_qtt):
    norm2 = dvol * qtt_inner(psi_qtt, psi_qtt).real
    norm = torch.sqrt(torch.clamp(norm2, min=1.0e-300))
    return qtt_scale(psi_qtt, 1.0 / norm)



# ===== Notebook cell 24 =====

# QTT-native block overlap, orthogonalization, and Rayleigh-Ritz

ORTHO_EIG_FLOOR = 1.0e-12

def qtt_overlap_matrix(states):
    M = len(states)
    S = torch.zeros((M, M), dtype=torch.complex128, device=device)

    for i in range(M):
        for j in range(i, M):
            value = dvol * qtt_inner(states[i], states[j])
            S[i, j] = value
            S[j, i] = value.conj()

    return 0.5 * (S + S.conj().T)


def rotate_qtt_block(states, C, eps=QTT_EPS, rmax=QTT_RMAX):
    """new_state_j = sum_i old_state_i * C[i,j]."""
    return [
        qtt_linear_combination(states, C[:, j], eps=eps, rmax=rmax)
        for j in range(C.shape[1])
    ]


def orthonormalize_qtt_block(states, passes=2):
    """Lowdin symmetric orthogonalization performed entirely with QTT overlaps."""
    states = list(states)

    for _ in range(passes):
        S = qtt_overlap_matrix(states)
        evals, U = torch.linalg.eigh(S)

        min_eval = float(evals.min().real.detach().cpu())
        if min_eval < ORTHO_EIG_FLOOR:
            print(f"Warning: small block-overlap eigenvalue {min_eval:.3e}; regularizing.")

        evals_safe = torch.clamp(evals.real, min=ORTHO_EIG_FLOOR)
        Sinvhalf = U @ torch.diag(evals_safe.rsqrt().to(torch.complex128)) @ U.conj().T
        states = rotate_qtt_block(states, Sinvhalf)

    return [qtt_normalize(state) for state in states]


def rayleigh_ritz_qtt(states):
    """QTT block orthogonalization + projected-Hamiltonian diagonalization."""
    states = orthonormalize_qtt_block(states, passes=2)
    M = len(states)

    Hstates = [apply_H_qtt(state) for state in states]
    Hsub = torch.zeros((M, M), dtype=torch.complex128, device=device)

```

```
    for i in range(M):
        for j in range(M):
            Hsub[i, j] = dvol * qtt_inner(states[i], Hstates[j])

    Hsub = 0.5 * (Hsub + Hsub.conj().T)
    evals, U = torch.linalg.eigh(Hsub)

    rotated = rotate_qtt_block(states, U)
    rotated = [qtt_normalize(state) for state in rotated]

    return rotated, evals.real



# ===== Notebook cell 26 =====

# Exact 1D Morse spectra and the k lowest product combinations

def morse_lambda(p):
    return np.sqrt(2.0 * p["mass"] * p["De"]) / (hbar * p["a"])

def exact_energy_1d(n, p, lam):
    s = lam - n - 0.5
    if s <= 0:
        raise ValueError(f"n={n} is not a bound level")
    return p["De"] - (hbar**2 * p["a"]**2 / (2.0 * p["mass"])) * s**2

def bound_energies_1d(p):
    lam = morse_lambda(p)
    values = []
    n = 0
    while lam - n - 0.5 > 0:
        values.append(exact_energy_1d(n, p, lam))
        n += 1
    return lam, values

LAMBDA = []
E1D_EXACT = []
for p in OSCILLATORS:
    lam, energies = bound_energies_1d(p)
    LAMBDA.append(lam)
    E1D_EXACT.append(energies)

def k_lowest_product_levels(energy_lists, k):
    """Return k smallest sums from sorted per-mode energy lists without full enumeration."""
    n = len(energy_lists)
    if any(len(e) == 0 for e in energy_lists):
        return []

    start = tuple([0] * n)
    start_E = sum(energy_lists[d][0] for d in range(n))
    heap = [(start_E, start)]
    visited = {start}
    out = []

    while heap and len(out) < k:
        E, label = heapq.heappop(heap)
        out.append((float(E), label))

        for d in range(n):
            if label[d] + 1 >= len(energy_lists[d]):
```

```
                continue
            new_label = list(label)
            new_label[d] += 1
            new_label = tuple(new_label)
            if new_label in visited:
                continue
            visited.add(new_label)
            new_E = sum(energy_lists[q][new_label[q]] for q in range(n))
            heapq.heappush(heap, (new_E, new_label))

    return out

target_levels = k_lowest_product_levels(E1D_EXACT, NSTATES)
if len(target_levels) < NSTATES:
    raise RuntimeError(f"Only {len(target_levels)} bound product states are available")

for d in range(N_OSC):
    print(f"mode {d}: lambda={LAMBDA[d]:.6f}, bound levels={len(E1D_EXACT[d])}")

print(f"\nLowest {NSTATES} uncoupled product levels:")
for k, (E, label) in enumerate(target_levels):
    print(f"  {k:2d}: n={label}, E={E:.12f}")

exact_energy_sorted = np.array([E for E, _ in target_levels], dtype=float)



# ===== Notebook cell 28 =====

# Exact 1D Morse states and n-dimensional product reference QTTs

def exact_state_1d(n, d):
    p = OSCILLATORS[d]
    lam = LAMBDA[d]
    x = XGRID[d].detach().cpu().numpy()
    s = lam - n - 0.5

    if s <= 0:
        raise ValueError(f"n={n} is not bound for mode {d}")

    z = 2.0 * lam * np.exp(-p["a"] * (x - p["xe"]))
    log_prefactor = s * np.log(z) - 0.5 * z
    prefactor = np.exp(np.maximum(log_prefactor, -745.0))
    psi = prefactor * eval_genlaguerre(n, 2.0 * s, z)
    psi = psi / np.sqrt(DX[d] * np.vdot(psi, psi).real)

    return torch.tensor(psi, dtype=torch.complex128, device=device)

_exact_1d_qtt_cache = {}

def exact_1d_qtt(n, d):
    key = (d, n)
    if key not in _exact_1d_qtt_cache:
        _exact_1d_qtt_cache[key] = qtt_from_dense(
            exact_state_1d(n, d),
            [BITS_PER_DIM[d]],
            eps=QTT_EPS,
            rmax=QTT_RMAX
        )
    return _exact_1d_qtt_cache[key]
```

```
def exact_product_state_qtt(label):
    factors = [exact_1d_qtt(label[d], d) for d in range(N_OSC)]
    return qtt_normalize(tensor_product_many(factors))



# ===== Notebook cell 30 =====

# Local 1D Fourier-grid Hamiltonian eigenstates used as numerical seeds

E1D_FGH = []
PSI1D_FGH = []

# We only need enough local states to cover the quantum numbers appearing
# in the requested low-energy product-state block.
max_needed_per_mode = [max(label[d] for _, label in target_levels) for d in range(N_OSC)]

for d in range(N_OSC):
    N = GRID_SHAPE[d]

    # F is the unitary discrete FFT matrix. The kinetic operator in the
    # coordinate representation is F^\dagger diag(T) F.
    eye = torch.eye(N, dtype=torch.complex128, device=device)
    F = torch.fft.fft(eye, dim=0, norm="ortho")
    Tmat = F.conj().T @ (T1D[d].to(torch.complex128)[:, None] * F)
    H1 = Tmat + torch.diag(V1D[d].to(torch.complex128))
    H1 = 0.5 * (H1 + H1.conj().T)

    evals, evecs = torch.linalg.eigh(H1)

    # torch.linalg.eigh normalizes sum_i |c_i|^2 = 1. Convert this to the
    # physical quadrature convention dx * sum_i |psi_i|^2 = 1.
    evecs = evecs / math.sqrt(DX[d])

    E1D_FGH.append(evals.real)
    PSI1D_FGH.append(evecs)

# Explicitly verify local grid quality against the independent analytical
# Morse spectrum before building the multidimensional block.
local_rows = []
for d in range(N_OSC):
    for n in range(max_needed_per_mode[d] + 1):
        E_exact = E1D_EXACT[d][n]
        E_fgh = float(E1D_FGH[d][n].detach().cpu())
        local_rows.append({
            "mode": d,
            "n": n,
            "E_exact": E_exact,
            "E_1D_FGH": E_fgh,
            "abs_error": abs(E_fgh - E_exact),
        })

print("1D grid-quality check (numerical FGH vs analytical Morse levels):")
display(pd.DataFrame(local_rows))

_seed_1d_qtt_cache = {}

def local_fgh_seed_1d_qtt(n, d):
    key = (d, n)
    if key not in _seed_1d_qtt_cache:
        vec = PSI1D_FGH[d][:, n]
```

```
        _seed_1d_qtt_cache[key] = qtt_from_dense(
            vec,
            [BITS_PER_DIM[d]],
            eps=QTT_EPS,
            rmax=QTT_RMAX
        )
    return _seed_1d_qtt_cache[key]

seed_labels = [label for _, label in target_levels]
states_qtt = []
for label in seed_labels:
    factors = [local_fgh_seed_1d_qtt(label[d], d) for d in range(N_OSC)]
    states_qtt.append(qtt_normalize(tensor_product_many(factors)))

print("\nLocal-FGH product seeds:")
for k, qtt in enumerate(states_qtt):
    qtt_summary(f"seed[{k}]", qtt)

print("\nInitial QTT Rayleigh-Ritz rotation...")
states_qtt, E0 = rayleigh_ritz_qtt(states_qtt)
print("Initial Ritz energies:")
for k, E in enumerate(E0.detach().cpu().numpy()):
    print(f"  {k:2d}: {E:.12f}")



# ===== Notebook cell 32 =====

# Split-operator propagators and one QTT imaginary-time step

def make_qtt_propagators(dtau):
    # For an uncoupled potential V=sum_d V_d, exp(-dtau V/2) factorizes
    # exactly. This removes TT-cross error from the default benchmark.
    if COUPLING_MODEL == "none":
        potential_factors = []
        for d in range(N_OSC):
            Uv_d = torch.exp(-0.5 * dtau * V1D[d])
            potential_factors.append(
                qtt_from_dense(Uv_d, [BITS_PER_DIM[d]], eps=QTT_EPS, rmax=QTT_RMAX)
            )
        Uv_qtt = tensor_product_many(potential_factors)
        info = {"method": "factorized", "val_epss": []}
    else:
        # Coupled V does not factorize, so apply the scalar map to VQTT by TT-cross.
        Uv_qtt, info = tn.cross(
            function=lambda v: torch.exp(-0.5 * dtau * v),
            tensors=[VQTT],
            eps=CROSS_EPS,
            rmax=CROSS_RMAX,
            max_iter=CROSS_MAX_ITER,
            val_size=CROSS_VAL_SIZE,
            verbose=False,
            return_info=True
        )
        Uv_qtt = qtt_round(Uv_qtt, eps=QTT_EPS, rmax=QTT_RMAX)

    # Kinetic propagator always factorizes because T=sum_d T_d.
    kinetic_factors = []
    for d in range(N_OSC):
        Ut_d = torch.exp(-dtau * T1D[d])
        kinetic_factors.append(
```

```
            qtt_from_dense(Ut_d, [BITS_PER_DIM[d]], eps=QTT_EPS, rmax=QTT_RMAX)
        )

    Ut_qtt = tensor_product_many(kinetic_factors)
    return Uv_qtt, Ut_qtt, info


def imaginary_time_step_qtt(psi_qtt, Uv_qtt, Ut_qtt):
    # Second-order Strang splitting:
    # exp(-dt H) = exp(-dt V/2) exp(-dt T) exp(-dt V/2) + O(dt^3)
    psi = qtt_hadamard(Uv_qtt, psi_qtt, eps=QTT_EPS, rmax=QTT_RMAX)
    psi = qtt_qft(psi, BITS_PER_DIM, inverse=False, eps=QFT_EPS, rmax=QTT_RMAX)
    psi = qtt_hadamard(Ut_qtt, psi, eps=QTT_EPS, rmax=QTT_RMAX)
    psi = qtt_qft(psi, BITS_PER_DIM, inverse=True, eps=QFT_EPS, rmax=QTT_RMAX)
    psi = qtt_hadamard(Uv_qtt, psi, eps=QTT_EPS, rmax=QTT_RMAX)
    return qtt_normalize(psi)



# ===== Notebook cell 34 =====

# n-state block imaginary-time propagation entirely in QTT form
#
# With local-FGH seeds the uncoupled benchmark starts extremely close to
# the target eigenstates, so we use small time steps primarily as a QTT
# propagation consistency test. Coupled problems need stronger filtering.
if COUPLING_MODEL == "none":
    schedule = [
        (5.0e-3, 3),
        (1.0e-3, 4),
        (2.0e-4, 5),
    ]
else:
    schedule = [
        (2.0e-2, 6),
        (5.0e-3, 8),
        (1.0e-3, 10),
        (2.0e-4, 10),
    ]

print(f"Starting {NSTATES}-state QTT imaginary-time propagation for {N_OSC} modes...\n")
history = []
global_step = 0
watch = sorted(set([0, min(1, NSTATES-1), NSTATES//2, NSTATES-1]))

for stage, (dtau, nsteps) in enumerate(schedule, start=1):
    Uv_qtt, Ut_qtt, info = make_qtt_propagators(dtau)
    val_epss = info.get("val_epss", [])
    cross_err = val_epss[-1] if len(val_epss) else np.nan
    prop_method = info.get("method", "tt-cross")

    print(
        f"Stage {stage}/{len(schedule)}: dtau={dtau:g}, steps={nsteps}, "
        f"Uv method={prop_method}, cross error={cross_err:.3e}"
    )

    for local_step in range(1, nsteps + 1):
        propagated = [
            imaginary_time_step_qtt(state, Uv_qtt, Ut_qtt)
            for state in states_qtt
        ]
```

```
        states_qtt, Esub = rayleigh_ritz_qtt(propagated)
        global_step += 1

        E_np = Esub.detach().cpu().numpy()
        max_rank_now = max(max(qtt_ranks(qtt)) for qtt in states_qtt)

        history.append({
            "global_step": global_step,
            "stage": stage,
            "local_step": local_step,
            "dtau": dtau,
            "max_rank": max_rank_now,
            **{f"E{k}": E_np[k] for k in range(NSTATES)},
        })

        report = ", ".join([f"E[{k}]={E_np[k]:.10f}" for k in watch])
        print(
            f"  step {local_step:3d}/{nsteps}; "
            f"max rank={max_rank_now:3d}; {report}"
        )

    print()

# Final cleanup rotation and independent Rayleigh quotients.
states_qtt, E_final_rr = rayleigh_ritz_qtt(states_qtt)
E_qtt_final = np.array([
    float(qtt_energy(state).detach().cpu())
    for state in states_qtt
])

order = np.argsort(E_qtt_final)
states_qtt = [states_qtt[i] for i in order]
E_qtt_final = E_qtt_final[order]

print("Final QTT energies:")
for k, E in enumerate(E_qtt_final):
    print(f"  {k:2d}: {E:.12f}")



# ===== Notebook cell 36 =====

# Optional dense validation

DENSE_VALIDATION_LIMIT = 200_000
DENSE_VALIDATION = TOTAL_POINTS <= DENSE_VALIDATION_LIMIT

E_dense_final = None

if DENSE_VALIDATION:
    print(f"Dense validation enabled ({TOTAL_POINTS:,} points <= {DENSE_VALIDATION_LIMIT:,}).")

    # Build dense coordinate-space potential only for this validation branch.
    meshes = torch.meshgrid(*XGRID, indexing="ij")
    X_dense = torch.stack([mesh.reshape(-1) for mesh in meshes], dim=1)
    Vdiag_dense = potential_nd(X_dense).reshape(GRID_SHAPE)

    # Dense kinetic diagonal by broadcasting the 1D T_d arrays.
    Tdiag_dense = torch.zeros(GRID_SHAPE, dtype=torch.get_default_dtype(), device=device)
    for d in range(N_OSC):
```

```
        shape = [1] * N_OSC
        shape[d] = GRID_SHAPE[d]
        Tdiag_dense = Tdiag_dense + T1D[d].reshape(shape)

    def apply_H_dense(Psi):
        squeeze = False
        if Psi.ndim == 1:
            Psi = Psi[:, None]
            squeeze = True

        M = Psi.shape[1]
        field = Psi.reshape(*GRID_SHAPE, M)
        fft_dims = tuple(range(N_OSC))
        field_p = torch.fft.fftn(field, dim=fft_dims, norm="ortho")

        Tfield = torch.fft.ifftn(
            Tdiag_dense[..., None] * field_p,
            dim=fft_dims,
            norm="ortho"
        )
        Hfield = Vdiag_dense[..., None] * field + Tfield
        out = Hfield.reshape(TOTAL_POINTS, M)
        return out[:, 0] if squeeze else out

    Psi_dense = torch.stack([
        qtt_to_dense(state, BITS_PER_DIM).reshape(-1)
        for state in states_qtt
    ], dim=1)

    HPsi_dense = apply_H_dense(Psi_dense)
    numerator = dvol * torch.sum(Psi_dense.conj() * HPsi_dense, dim=0)
    denominator = dvol * torch.sum(Psi_dense.conj() * Psi_dense, dim=0)
    E_dense_final = (numerator / denominator).real.detach().cpu().numpy()

    print("\nQTT vs dense-FGH expectation values:")
    dense_check = pd.DataFrame({
        "state": np.arange(NSTATES),
        "E_QTT": E_qtt_final,
        "E_dense_FGH": E_dense_final,
        "abs_difference": np.abs(E_qtt_final - E_dense_final),
    })
    display(dense_check)
else:
    print(
        f"Dense validation skipped: {TOTAL_POINTS:,} points > "
        f"{DENSE_VALIDATION_LIMIT:,}. The QTT calculation itself is unaffected."
    )



# ===== Notebook cell 38 =====

# Analyze final states using QTT overlaps and Hamiltonian residuals

def qtt_squared_overlap(a, b):
    ov = dvol * qtt_inner(a, b)
    na = dvol * qtt_inner(a, a).real
    nb = dvol * qtt_inner(b, b).real
    return float((torch.abs(ov)**2 / (na * nb)).real.detach().cpu())


```

```
def qtt_residual_norm(psi_qtt, energy):
    """Physical L2 norm ||H psi - E psi|| / ||psi||."""
    Hpsi = apply_H_qtt(psi_qtt)
    residual = qtt_add(
        Hpsi,
        qtt_scale(psi_qtt, -energy),
        eps=QTT_EPS,
        rmax=QTT_RMAX
    )
    num = dvol * qtt_inner(residual, residual).real
    den = dvol * qtt_inner(psi_qtt, psi_qtt).real
    return float(torch.sqrt(torch.clamp(num / den, min=0.0)).detach().cpu())


residuals_qtt = [qtt_residual_norm(state, E_qtt_final[j]) for j, state in enumerate(states_qtt)]

if COUPLING_MODEL == "none":
    reference_levels = target_levels
    reference_qtts = [exact_product_state_qtt(label) for _, label in reference_levels]

    overlap_matrix = np.zeros((NSTATES, NSTATES), dtype=float)
    for i in range(NSTATES):
        for j in range(NSTATES):
            overlap_matrix[i, j] = qtt_squared_overlap(reference_qtts[i], states_qtt[j])

    exact_idx, num_idx = linear_sum_assignment(-overlap_matrix)
    assigned = {int(i): int(j) for i, j in zip(exact_idx, num_idx)}

    rows = []
    for i, (E_exact, label) in enumerate(reference_levels):
        j = assigned[i]
        abs_err = abs(E_qtt_final[j] - E_exact)
        rows.append({
            "exact_rank": i,
            "numerical_index": j,
            "quantum_numbers": str(label),
            "E_exact": E_exact,
            "E_QTT": E_qtt_final[j],
            "abs_energy_error": abs_err,
            "rel_energy_error": abs_err / max(abs(E_exact), 1.0e-15),
            "fidelity": overlap_matrix[i, j],
            "residual_norm": residuals_qtt[j],
            "max_QTT_rank": max(qtt_ranks(states_qtt[j])),
            "QTT_coefficients": qtt_numcoef(states_qtt[j]),
        })

    results = pd.DataFrame(rows).sort_values("exact_rank").reset_index(drop=True)
    print("Uncoupled exact-state comparison:")
    display(results)

    print("\nSummary accuracy:")
    print(f"  max |E_QTT-E_exact| = {results['abs_energy_error'].max():.3e}")
    print(f"  min fidelity        = {results['fidelity'].min():.12f}")
    print(f"  max residual norm   = {results['residual_norm'].max():.3e}")

else:
    NREFERENCE = max(2 * NSTATES, NSTATES)
    reference_levels = k_lowest_product_levels(E1D_EXACT, NREFERENCE)
    reference_qtts = [exact_product_state_qtt(label) for _, label in reference_levels]

    overlap_matrix = np.zeros((len(reference_qtts), NSTATES), dtype=float)
```

```
    for i in range(len(reference_qtts)):
        for j in range(NSTATES):
            overlap_matrix[i, j] = qtt_squared_overlap(reference_qtts[i], states_qtt[j])

    rows = []
    for j in range(NSTATES):
        i_dom = int(np.argmax(overlap_matrix[:, j]))
        weights = overlap_matrix[:, j]
        captured = float(np.sum(weights))
        participation = (captured**2 / np.sum(weights**2)) if np.sum(weights**2) > 0 else np.nan

        rows.append({
            "numerical_state": j,
            "E_QTT": E_qtt_final[j],
            "residual_norm": residuals_qtt[j],
            "dominant_uncoupled_label": str(reference_levels[i_dom][1]),
            "dominant_product_weight": overlap_matrix[i_dom, j],
            "captured_weight_in_reference_set": captured,
            "effective_product_participation": participation,
            "max_QTT_rank": max(qtt_ranks(states_qtt[j])),
            "QTT_coefficients": qtt_numcoef(states_qtt[j]),
        })

    results = pd.DataFrame(rows)
    print("Coupled-state product-basis character:")
    display(results)

# Add dense-validation energies when available.
if E_dense_final is not None:
    dense_map = {int(i): float(E_dense_final[i]) for i in range(NSTATES)}
    if COUPLING_MODEL == "none":
        results["E_dense_FGH"] = [dense_map[int(j)] for j in results["numerical_index"]]
        results["QTT_minus_dense"] = results["E_QTT"] - results["E_dense_FGH"]
    else:
        results["E_dense_FGH"] = [dense_map[int(j)] for j in results["numerical_state"]]
        results["QTT_minus_dense"] = results["E_QTT"] - results["E_dense_FGH"]

results_file = OUTPUT_DIR / "n_morse_qtt_results.csv"
results.to_csv(results_file, index=False)
print("Saved:", results_file)



# ===== Notebook cell 40 =====

# QTT-native one-coordinate marginals

def qtt_marginal_1d(psi_qtt, target_dim):
    prob_qtt = qtt_hadamard(
        qtt_conjugate(psi_qtt),
        psi_qtt,
        eps=QTT_EPS,
        rmax=QTT_RMAX
    )

    cores = qtt_unfold_to_physical_tt(prob_qtt, BITS_PER_DIM)

    # Contract all coordinates to the left of target_dim.
    left = torch.ones((1,), dtype=cores[0].dtype, device=device)
    for d in range(target_dim):
        summed = cores[d].sum(dim=1)
```

```
        left = left @ summed

    # Contract all coordinates to the right of target_dim.
    right = torch.ones((1,), dtype=cores[0].dtype, device=device)
    for d in range(N_OSC - 1, target_dim, -1):
        summed = cores[d].sum(dim=1)
        right = summed @ right

    marginal = torch.einsum("a,aib,b->i", left, cores[target_dim], right).real

    # Integrate over all coordinates except target_dim.
    other_volume = dvol / DX[target_dim]
    marginal = marginal * other_volume

    # Remove tiny negative roundoff and renormalize.
    marginal = torch.clamp(marginal, min=0.0)
    norm = DX[target_dim] * marginal.sum()
    marginal = marginal / torch.clamp(norm, min=1.0e-300)

    return marginal

selected_states = sorted(set([0, min(1, NSTATES - 1), NSTATES - 1]))

for state_index in selected_states:
    plt.figure(figsize=(9, 5))
    for d in range(N_OSC):
        marginal = qtt_marginal_1d(states_qtt[state_index], d)
        plt.plot(
            XGRID[d].detach().cpu().numpy(),
            marginal.detach().cpu().numpy(),
            label=f"mode {d}"
        )

    plt.xlabel("coordinate")
    plt.ylabel(r"marginal probability $P_d(x_d)$")
    plt.title(f"State {state_index}: one-coordinate marginals")
    plt.grid(alpha=0.25)
    plt.legend()
    plt.tight_layout()
    plt.show()



# ===== Notebook cell 42 =====

# Convergence and final-spectrum diagnostics

history_df = pd.DataFrame(history)
history_file = OUTPUT_DIR / "n_morse_qtt_history.csv"
history_df.to_csv(history_file, index=False)
print("Saved:", history_file)

plt.figure(figsize=(10, 5))
for k in range(min(NSTATES, 6)):
    plt.plot(history_df["global_step"], history_df[f"E{k}"], marker="o", ms=3, label=f"E{k}")
plt.xlabel("QTT block iteration")
plt.ylabel("Ritz energy")
plt.title("Imaginary-time convergence of the lowest Ritz energies")
plt.grid(alpha=0.25)
plt.legend(ncol=2)
plt.tight_layout()
```

```
plt.show()

plt.figure(figsize=(10, 4))
plt.plot(history_df["global_step"], history_df["max_rank"], marker="o")
plt.xlabel("QTT block iteration")
plt.ylabel("maximum QTT rank")
plt.title("Rank growth during imaginary-time propagation")
plt.grid(alpha=0.25)
plt.tight_layout()
plt.show()

plt.figure(figsize=(10, 5))
plt.plot(np.arange(NSTATES), E_qtt_final, "o-", label="QTT")
if COUPLING_MODEL == "none":
    plt.plot(np.arange(NSTATES), exact_energy_sorted, "s--", label="exact uncoupled")
plt.xlabel("energy rank")
plt.ylabel("energy")
plt.title(f"Lowest {NSTATES} energies for {N_OSC} Morse modes")
plt.grid(alpha=0.25)
plt.legend()
plt.tight_layout()
plt.show()

plt.figure(figsize=(10, 6))
plt.imshow(overlap_matrix, origin="lower", aspect="auto")
plt.colorbar(label="squared overlap")
plt.xlabel("numerical state")
plt.ylabel("exact product state" if COUPLING_MODEL == "none" else "uncoupled product basis state")
plt.title("QTT overlap matrix")
plt.tight_layout()
plt.show()



# ===== Notebook cell 44 =====

# Side-by-side 2D numerical/analytical heat maps
#
# By default show every state in the requested low-energy block. Reduce
# N_HEATMAP_STATES if you want a shorter notebook output.
N_HEATMAP_STATES = min(NSTATES, 12)

if N_OSC == 2 and COUPLING_MODEL == "none":
    if 'assigned' not in globals():
        raise RuntimeError("Run the exact-state matching cell before the heat-map cell.")

    x1min, x1max = OSCILLATORS[0]["xmin"], OSCILLATORS[0]["xmax"]
    x2min, x2max = OSCILLATORS[1]["xmin"], OSCILLATORS[1]["xmax"]
    extent = [x2min, x2max, x1min, x1max]

    for i in range(N_HEATMAP_STATES):
        E_exact, label = reference_levels[i]
        j = assigned[i]

        psi_num = qtt_to_dense(states_qtt[j], BITS_PER_DIM)
        psi_exact = qtt_to_dense(reference_qtts[i], BITS_PER_DIM)

        # Phase-align the numerical state to the analytical reference.
        overlap = dvol * qtt_inner(reference_qtts[i], states_qtt[j])
        if torch.abs(overlap) > 0:
            phase_factor = torch.conj(overlap) / torch.abs(overlap)
```

```
            psi_num = psi_num * phase_factor

        psi_num_np = psi_num.detach().cpu().numpy()
        psi_exact_np = psi_exact.detach().cpu().numpy()

        num_real = np.real(psi_num_np)
        exact_real = np.real(psi_exact_np)
        num_prob = np.abs(psi_num_np)**2
        exact_prob = np.abs(psi_exact_np)**2

        amp_max = max(np.max(np.abs(num_real)), np.max(np.abs(exact_real)))
        prob_max = max(np.max(num_prob), np.max(exact_prob))

        fidelity = overlap_matrix[i, j]
        E_num = E_qtt_final[j]
        E_err = abs(E_num - E_exact)

        fig, axes = plt.subplots(2, 2, figsize=(11, 9), constrained_layout=True)

        im00 = axes[0, 0].imshow(
            num_real, origin="lower", aspect="auto", extent=extent,
            cmap="RdBu_r", vmin=-amp_max, vmax=amp_max
        )
        axes[0, 0].set_title("QTT: phase-aligned Re $\\Psi$")
        axes[0, 0].set_ylabel(r"$x_1$")
        fig.colorbar(im00, ax=axes[0, 0], shrink=0.85)

        im01 = axes[0, 1].imshow(
            exact_real, origin="lower", aspect="auto", extent=extent,
            cmap="RdBu_r", vmin=-amp_max, vmax=amp_max
        )
        axes[0, 1].set_title("Analytical: Re $\\Psi$")
        fig.colorbar(im01, ax=axes[0, 1], shrink=0.85)

        im10 = axes[1, 0].imshow(
            num_prob, origin="lower", aspect="auto", extent=extent,
            cmap="viridis", vmin=0.0, vmax=prob_max
        )
        axes[1, 0].set_title(r"QTT: $|\Psi|^2$")
        axes[1, 0].set_xlabel(r"$x_2$")
        axes[1, 0].set_ylabel(r"$x_1$")
        fig.colorbar(im10, ax=axes[1, 0], shrink=0.85)

        im11 = axes[1, 1].imshow(
            exact_prob, origin="lower", aspect="auto", extent=extent,
            cmap="viridis", vmin=0.0, vmax=prob_max
        )
        axes[1, 1].set_title(r"Analytical: $|\Psi|^2$")
        axes[1, 1].set_xlabel(r"$x_2$")
        fig.colorbar(im11, ax=axes[1, 1], shrink=0.85)

        fig.suptitle(
            f"state rank {i}, n={label}   |   "
            f"E_exact={E_exact:.10f}, E_QTT={E_num:.10f}, "
            f"|dE|={E_err:.2e}, fidelity={fidelity:.10f}"
        )
        plt.show()
else:
    print(
        "Direct QTT/analytical 2D heat maps require N_OSC == 2 and "
        "COUPLING_MODEL == 'none'. For coupled or higher-dimensional systems, "
```

```
            "use marginals and product-basis projections instead."
        )
```