DMRG for Quantum Eigenstates

Binary tensorization, TT-cross, and matrix-free MPOs

Victor S. Batista

Abstract

This tutorial develops a sequence of controlled tensor-network calculations for vibrational eigenstates of Morse oscillators, following the accompanying notebook `DMRG_Morse_TTcross.ipynb`. We begin with a one-dimensional Morse oscillator on a uniform grid with $N_x = 2^L$ points and convert the grid index into an L-bit artificial chain. This binary tensorization maps the wavefunction to a matrix-product state (MPS) and the Hamiltonian to a matrix-product operator (MPO). A small $L = 8$ problem is first solved by dense finite-difference diagonalization and by two-site DMRG, so that analytic Morse error, grid-discretization error, and MPS/DMRG error can be separated cleanly. We then remove the dense Hamiltonian: the diagonal Morse potential is constructed by TT-cross from adaptive black-box function evaluations, whereas the finite-difference kinetic energy is represented exactly with binary shift MPOs. Excited states are obtained with matrix-free MPS-projector deflation. Finally, the same construction is applied directly to a combined two-coordinate Hamiltonian on 18 binary sites ($512^2 = 262\,144$ grid configurations), using a fourth-order kinetic stencil, tighter TT-cross/DMRG controls, repeated DMRG restarts, and a Rayleigh–Ritz refinement in the span of the computed MPS states. The two-coordinate example is intentionally discussed together with its remaining failure at the twentieth targeted level, illustrating an important practical lesson: improving discretization and subspace refinement does not by itself guarantee that sequential deflation has found every low-energy eigenstate.

Contents

1 Why use DMRG for a one-dimensional oscillator?

For a single one-dimensional vibrational coordinate, direct diagonalization, a discrete-variable representation, or a finite-difference eigensolver is normally simpler than DMRG. The density-matrix renormalization group was introduced by White [1, 2] and is especially transparent when formulated variationally in the modern MPS language [3]. The purpose of the present calculation is therefore not computational efficiency. Instead, the Morse oscillator provides an unusually transparent environment in which to learn the tensor-network methodology because three independent descriptions can be compared:

1. the analytic Morse spectrum;
2. exact diagonalization of the finite-difference grid Hamiltonian;
3. DMRG optimization of an MPS representation of the same grid problem.

The calculation therefore separates two distinct sources of numerical error. The difference between the analytic and finite-grid spectra measures the *coordinate discretization error*, whereas the difference between the finite-grid and DMRG results measures the *tensor-network optimization and truncation error*.

The important conceptual step is to convert a single coordinate into a tensor train on which DMRG can act. This is done by binary tensorization of the coordinate grid, closely related to the quantized tensor-train (QTT) construction [4, 5]. Later, we show that TT-cross can replace the stored potential vector, binary-shift MPOs replace the dense kinetic matrix, and MPS projectors replace dense outer products in excited-state deflation. The final two-coordinate calculation therefore uses the same tensorized language from Hamiltonian construction through eigenstate refinement.

2 The Morse oscillator

2.1 Hamiltonian

The one-dimensional model system is described by the Morse Hamiltonian, originally introduced as an analytically solvable model of molecular vibrations [6],

$$H = -\frac{\hbar^2}{2m}\frac{d^2}{dx^2} + D_\mathrm{e}\left[1 - e^{-a(x-x_\mathrm{e})}\right]^2, \tag{1}$$

with $x_\mathrm{e} = 0$. The parameters are

$$\hbar = 1, \qquad m = 1, \qquad D_\mathrm{e} = 12, \qquad a = 0.8. \tag{2}$$

The potential minimum is zero and the dissociation threshold is $D_\mathrm{e} = 12$.

The coordinate interval is

$$x \in [-2.5, 10.0] \tag{3}$$

and is discretized with

$$N_x = 256 = 2^8 \tag{4}$$

points. Thus $L = 8$ binary sites will be required later. The grid spacing is

$$\Delta x = \frac{x_{\max} - x_{\min}}{N_x - 1} = 0.049020. \tag{5}$$

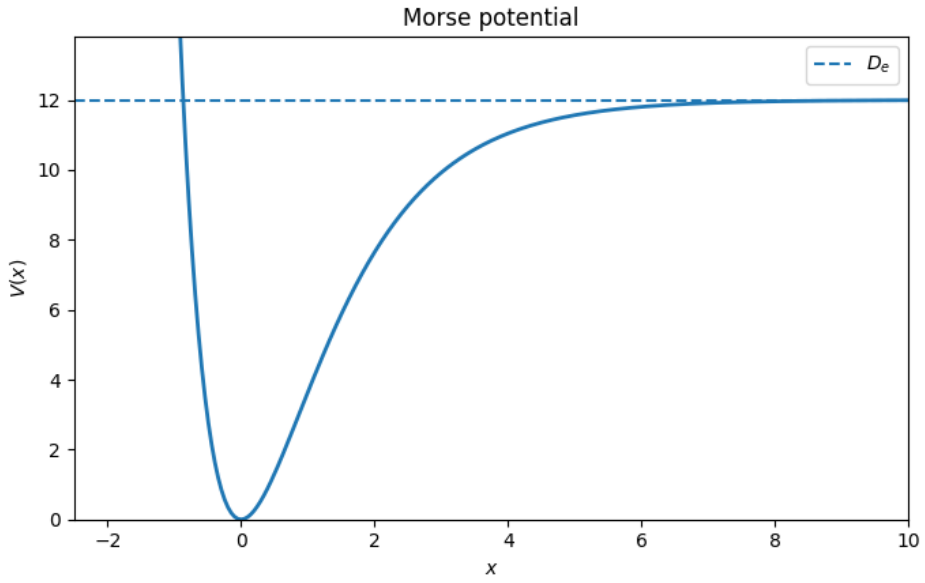

Figure 1: **Morse potential used in the numerical benchmark.** The minimum is at $x_e = 0$, while the dashed horizontal line marks the dissociation threshold $D_e = 12$. The asymmetric potential is steeply repulsive for $x < 0$ and approaches D_e from below for large positive x.

2.2 Analytic spectrum

For the convention in Eq. (1), define

$$\omega_e = a\sqrt{\frac{2D_e}{m}} \tag{6}$$

and

$$\lambda = \frac{\sqrt{2mD_e}}{\hbar a}. \tag{7}$$

The bound-state energies are

$$E_n = \hbar\omega_e\left(n + \frac{1}{2}\right) - \frac{\hbar^2 a^2}{2m}\left(n + \frac{1}{2}\right)^2. \tag{8}$$

Bound states satisfy

$$n + \frac{1}{2} < \lambda. \tag{9}$$

For the parameters above,

$$\lambda = 6.123724, \tag{10}$$

so $n = 0, \ldots, 5$: the model contains six analytic bound states.

Equation (8) already displays the anharmonicity of the Morse oscillator. Unlike a harmonic oscillator, the spacing between adjacent levels decreases with increasing n, and the highest levels accumulate near the dissociation threshold.

3 Finite-difference discretization

On a uniform grid $x_j = x_{\min} + j\Delta x$, the second derivative is approximated by the standard centered finite-difference formula [7],

$$\left.\frac{d^2\psi}{dx^2}\right|_{x_j} \simeq \frac{\psi_{j+1} - 2\psi_j + \psi_{j-1}}{\Delta x^2}. \tag{11}$$

Substitution into the kinetic-energy operator gives

$$T_{jj} = \frac{\hbar^2}{m\Delta x^2}, \tag{12}$$

$$T_{j,j\pm1} = -\frac{\hbar^2}{2m\Delta x^2}, \tag{13}$$

with all other elements zero. The full matrix is therefore

$$H_{jj'} = T_{jj'} + V(x_j)\delta_{jj'}. \tag{14}$$

The implementation is compact:

Listing 1: Construction of the finite-difference Morse Hamiltonian.

```
def build_morse_hamiltonian(x, De, a, m=1.0, hbar=1.0, xe=0.0):
    dx = x[1] - x[0]
    N = len(x)

    V = De * (1.0 - np.exp(-a * (x - xe)))**2

    main = np.full(N, hbar**2 / (m * dx**2)) + V
    off = np.full(N - 1, -hbar**2 / (2.0 * m * dx**2))

    H = np.diag(main)
    H += np.diag(off, k=1)
    H += np.diag(off, k=-1)

    return H, V
```

The resulting matrix has shape 256×256, and the notebook verifies a Hermiticity error of exactly zero to machine precision.

3.1 Finite-grid benchmark

Direct diagonalization of Eq. (14) provides the benchmark listed in Table 1. The errors are of order 10^{-3}–10^{-2}, except for the highest bound state, and arise from the finite coordinate box and the second-order finite-difference approximation.

These values are important because DMRG should reproduce the *finite-grid* spectrum, not the continuum analytic spectrum exactly.

4 Binary tensorization: turning coordinates into MPS chains

4.1 Binary representation of the grid index

DMRG is naturally formulated for a tensor-product Hilbert space. A single coordinate grid instead begins as a vector space of dimension N_x. The bridge between the two descriptions is the choice

$$N_x = 2^L. \tag{15}$$

Table 1: **Analytic and finite-grid Morse energies.** The final column is $E_n^{\mathrm{grid}} - E_n^{\mathrm{ana}}$.

n	E_n^{ana}	E_n^{grid}	Difference
0	1.879591794	1.878560736	-1.031×10^{-3}
1	5.158775383	5.155056237	-3.719×10^{-3}
2	7.797958971	7.791393954	-6.565×10^{-3}
3	9.797142560	9.789375168	-7.767×10^{-3}
4	11.156326148	11.149821257	-6.505×10^{-3}
5	11.875509736	11.876096855	$+5.871 \times 10^{-4}$

Every grid index can then be expressed as an L-bit binary number,

$$j = (b_0 b_1 \dots b_{L-1})_2, \qquad b_\ell \in \{0, 1\}. \tag{16}$$

Consequently,

$$\mathbb{C}^{N_x} = \mathbb{C}^{2^L} \simeq \underbrace{\mathbb{C}^2 \otimes \mathbb{C}^2 \otimes \cdots \otimes \mathbb{C}^2}_{L \text{ factors}}. \tag{17}$$

A grid wavefunction

$$|\psi\rangle = \sum_{j=0}^{N_x - 1} c_j \, |j\rangle \tag{18}$$

can therefore be rewritten as

$$|\psi\rangle = \sum_{b_0, \dots, b_{L-1}} \Psi_{b_0 \dots b_{L-1}} \, |b_0\rangle \otimes \cdots \otimes |b_{L-1}\rangle \, . \tag{19}$$

The eight sites in the present calculation are *not physical spins*. They are the bits of the coordinate-grid index. This distinction is central to interpreting the calculation correctly.

4.2 Matrix-product-state factorization

The order-L coefficient tensor can be factorized as an MPS [3, 8],

$$\Psi_{b_0 b_1 \cdots b_{L-1}} = A^{[0] b_0} A^{[1] b_1} \cdots A^{[L-1] b_{L-1}}. \tag{20}$$

For open boundary conditions the first and last matrices have one trivial bond index, while the internal bonds have dimensions $\chi_1, \chi_2, \dots$. The largest retained dimension is usually denoted by Schmidt rank χ.

For a completely arbitrary state on $L = 8$ two-level sites, the largest Schmidt rank occurs at the central cut and cannot exceed

$$2^{L/2} = 16. \tag{21}$$

Thus $\chi = 16$ is formally sufficient to represent any state in the present 256-dimensional space. The useful observation, however, is that the low Morse eigenstates require substantially smaller ranks.

5 Representing the Hamiltonian as an MPO

The operator analogue of an MPS is a matrix-product operator [3, 8],

$$H_{b_0 \dots b_{L-1}, \, b'_0 \dots b'_{L-1}} = W^{[0] b_0 b'_0} W^{[1] b_1 b'_1} \cdots W^{[L-1] b_{L-1} b'_{L-1}}. \tag{22}$$

For pedagogical reasons, the first part of the notebook constructs the finite-grid Hamiltonian explicitly as a dense 256×256 matrix and then compresses it into an MPO, using `MatrixProductOperator.from_dense` as explained in Sec. A.1. Later, the notebook demonstrates how to construct the MPO directly, without building first the dense matrix.

Listing 2: Dense-to-MPO conversion used in the tutorial.

```
H_mpo = qtn.MatrixProductOperator.from_dense(
    H,
    dims=[2] * L,
    cutoff=1e-12,
)
```

Note that building the dense $N_x \times N_x$ matrix is *not* a scalable strategy. For large-scale implementations we will build directly compressed MPO representations of the finite-difference kinetic operator and the diagonal Morse potential directly.

6 Two-site DMRG for the ground state

In its variational MPS formulation, DMRG minimizes the Rayleigh quotient [1–3]

$$\mathcal{E}[\Psi] = \frac{\langle \Psi | H | \Psi \rangle}{\langle \Psi | \Psi \rangle} \tag{23}$$

within the set of MPSs allowed by the chosen bond dimensions.

In a two-site DMRG sweep, two adjacent MPS tensors are combined into a local two-site tensor. All remaining tensors are held fixed, reducing the global variational problem to an effective eigenvalue problem for that two-site tensor. After optimization, a singular-value decomposition separates the two sites again. Small singular values can be discarded, thereby controlling the MPS bond dimension and the computational cost. The sweep is then moved along the chain and repeated in the opposite direction.

The notebook wraps the `quimb` two-site solver `DMRG2` [9], explained in Sec. A.2, as follows:

Listing 3: Ground-state DMRG driver.

```
def run_dmrg(mpo, bond_dims=(4, 8, 16), cutoff=1e-10,
             tol=1e-10, max_sweeps=12, verbosity=1):

    p0 = qtn.MPS_rand_state(
        L=mpo.L,
        bond_dim=2,
        phys_dim=2,
        normalize=True,
    )

    dmrg = qtn.DMRG2(
        mpo,
        bond_dims=list(bond_dims),
        cutoffs=cutoff,
        p0=p0,
    )

    converged = dmrg.solve(
        tol=tol,
        max_sweeps=max_sweeps,
        verbosity=verbosity,
    )

```

```
    psi = dmrg.state.copy()
    vec = mps_to_vector(psi)
    return converged, dmrg, psi, vec
```

The sequence

$$\chi = 4 \longrightarrow 8 \longrightarrow 16 \tag{24}$$

allows the variational space to grow during successive sweeps.

6.1 A useful convergence distinction

With the strict setting `tol` $= 10^{-10}$ and at most 12 sweeps, the notebook returns `converged = False`. This flag should not be confused with failure to obtain an accurate state. The final ground-state diagnostics are

$$E_0^{\mathrm{DMRG}} = 1.87856075684, \tag{25}$$

$$E_0^{\mathrm{grid}} = 1.87856073594, \tag{26}$$

$$E_0^{\mathrm{DMRG}} - E_0^{\mathrm{grid}} = 2.09 \times 10^{-8}, \tag{27}$$

$$|\langle \psi_0^{\mathrm{grid}} | \psi_0^{\mathrm{DMRG}} \rangle|^2 = 0.999999999668. \tag{28}$$

The final MPS has maximum bond dimension 5. Thus the requested stopping criterion was not reached within the sweep limit, but the physical state is already extremely accurate.

7 Excited states by deflation

Ordinary ground-state DMRG minimizes Eq. (23). To obtain successive excited states we use deflation, as follows. After finding states $|\psi_0\rangle, \ldots, |\psi_{k-1}\rangle$, the deflation method defines

$$H^{(k)} = H + \mu \sum_{r=0}^{k-1} |\psi_r\rangle \langle \psi_r| . \tag{29}$$

The positive penalty μ shifts the already found states upward in energy. If the shift is larger than the relevant level spacings, the ground state of $H^{(k)}$ is the next physical state.

The notebook uses

$$\mu = 4D_{\mathrm{e}} = 48. \tag{30}$$

After each DMRG run, the newly obtained vector is explicitly Gram–Schmidt orthogonalized against all previously stored states and renormalized. Finally, its physical energy is evaluated using the original Hamiltonian H, not the penalized Hamiltonian.

A condensed form of the central loop is

Listing 4: Excited-state deflation used in the notebook.

```
for k in range(n_states):
    Hdef = H.copy()

    for psi_old in states:
        Hdef += penalty * np.outer(
            psi_old, psi_old.conj()
        )

    Hdef_mpo = qtn.MatrixProductOperator.from_dense(
        Hdef, dims=[2] * L, cutoff=mpo_cutoff
    )

```

```
    converged, dmrg, psi_mps, psi = run_dmrg(
        Hdef_mpo, bond_dims=bond_dims,
        cutoff=dmrg_cutoff, tol=tol,
        max_sweeps=max_sweeps
    )

    for psi_old in states:
        psi -= np.vdot(psi_old, psi) * psi_old

    psi /= np.linalg.norm(psi)
    E = np.vdot(psi, H @ psi).real
```

This construction is deliberately simple and transparent. Because each deflated dense Hamiltonian is rebuilt and refactorized as an MPO, it is not a production-scale excited-state DMRG method. For larger problems, orthogonality constraints are normally incorporated directly into the variational sweep.

8 One-oscillator DMRG results

8.1 Energies and state fidelities

The first five DMRG states are compared with exact finite-grid eigenvectors in Table 2. The fidelity is

$$F_n = \left| \langle \psi_n^{\mathrm{grid}} | \psi_n^{\mathrm{DMRG}} \rangle \right|^2 . \tag{31}$$

Table 2: **DMRG validation for the first five Morse eigenstates.** The energy error is measured relative to exact diagonalization of the same finite-grid Hamiltonian.

n	E_n^{grid}	E_n^{DMRG}	Error	F_n
0	1.878560736	1.878560757	2.062×10^{-8}	0.9999999997
1	5.155056237	5.155056830	5.928×10^{-7}	0.9999998779
2	7.791393954	7.791394384	4.301×10^{-7}	0.9999998450
3	9.789375168	9.789378946	3.778×10^{-6}	0.9999983751
4	11.149821257	11.149829781	8.525×10^{-6}	0.9999944195

The trend is instructive. The DMRG error grows modestly for the higher states, as expected for repeated deflation and for wavefunctions that extend farther into the anharmonic region, but even for $n = 4$ the error remains only 8.5×10^{-6} and the fidelity remains 0.9999944.

8.2 Wavefunctions

The matrix eigenvectors are normalized according to

$$\sum_j |c_j|^2 = 1. \tag{32}$$

To display an approximation to a continuum-normalized wavefunction, the notebook plots

$$\psi(x_j) \simeq \frac{c_j}{\sqrt{\Delta x}}, \tag{33}$$

for which

$$\sum_j |\psi(x_j)|^2 \Delta x \simeq 1. \tag{34}$$

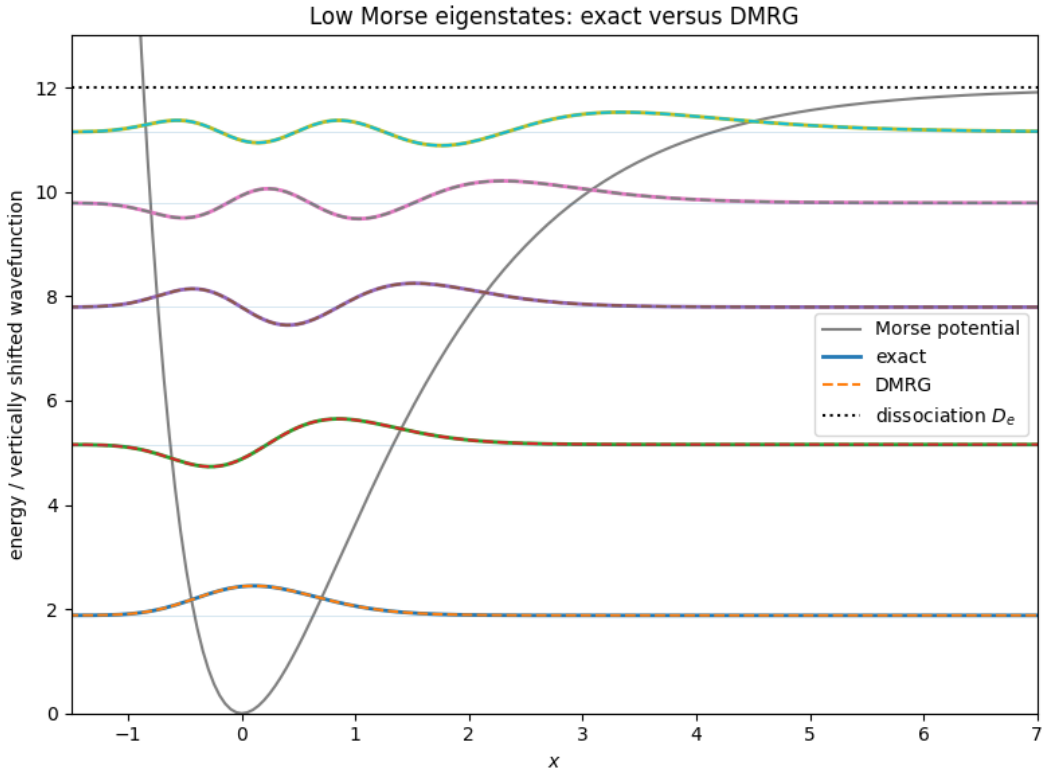

Figure 2: **First five Morse eigenstates from exact grid diagonalization and DMRG.** The wavefunctions are vertically displaced by their corresponding energies and superimposed on the Morse potential. Solid curves denote exact finite-grid eigenstates and dashed curves denote DMRG states. Their near-complete overlap is consistent with the fidelities in Table 2. The dotted horizontal line marks the dissociation threshold D_e.

Because an eigenvector has an arbitrary global sign, the DMRG vector is sign-aligned with the exact grid vector before plotting.

Figure 2 also illustrates the physical progression of the Morse eigenstates. Higher states explore larger positive displacements, where the potential becomes shallower and approaches dissociation. This is the coordinate-space manifestation of the decreasing anharmonic level spacing.

9 Bond-dimension convergence

The central approximation in an MPS calculation is the restriction of the Schmidt rank across each cut. The notebook therefore repeats the ground-state calculation at fixed maximum bond dimensions $\chi = 2, 4, 8, 16$.

Table 3: **Ground-state convergence with MPS bond dimension.**

Requested χ	Energy	Error vs. grid	Fidelity	Final max. bond
2	2.2344359645	3.559×10^{-1}	0.9956743723	2
4	1.8786166718	5.594×10^{-5}	0.9999993390	4
8	1.8785607726	3.668×10^{-8}	0.9999999988	7
16	1.8785607727	3.677×10^{-8}	0.9999999984	6

The improvement from $\chi = 2$ to $\chi = 8$ spans more than seven orders of magnitude in the energy error. Increasing the requested maximum from 8 to 16 does not materially improve the result in this stochastic finite-sweep run; the optimized states terminate at maximum bond dimensions 7 and 6, respectively. The nearly identical final errors are therefore better interpreted as residual optimization tolerance rather than as a limitation of the available variational bond dimension.

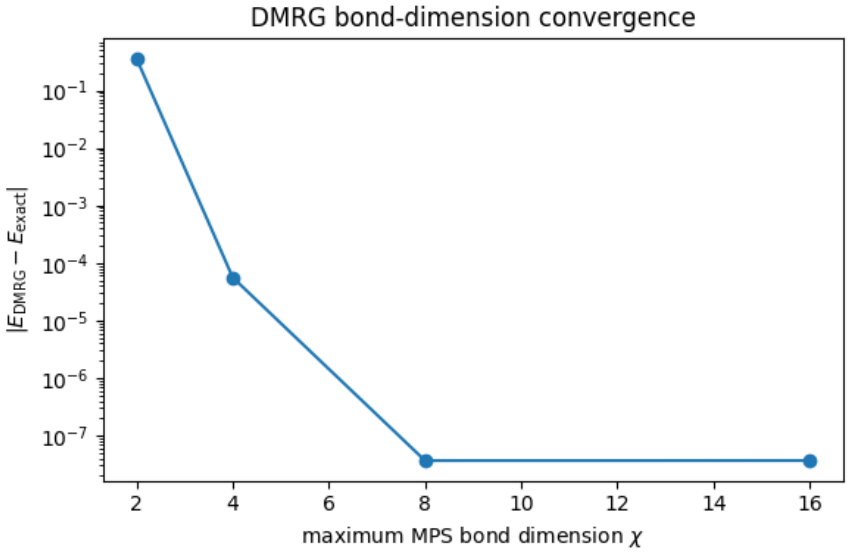

Figure 3: **Ground-state energy error versus MPS bond dimension.** The error drops rapidly as the variational space is enlarged from $\chi = 2$ to $\chi = 8$. The comparable errors at requested $\chi = 8$ and 16 are consistent with optimized MPSs that terminate at bond dimensions 7 and 6 rather than saturating the requested maxima.

This convergence test is one of the clearest demonstrations of what DMRG is achieving in

this artificial binary chain: the exact 256-component ground state does not need the maximum possible rank 16. A much lower-rank MPS is already sufficient.

10 From dense validation to matrix-free TT-cross DMRG

The dense $L = 8$ calculation is an excellent validation problem, but it is not yet a scalable tensor-network calculation. The call

```
H_mpo = qtn.MatrixProductOperator.from_dense(
    H,
    dims=[2] * L,
    cutoff=1e-12,
)
```

starts from all $N_x^2 = 4^L$ matrix elements. The notebook therefore next rebuilds the same Hamiltonian without ever forming the coordinate-space matrix,

$$H_{\mathrm{MPO}} = T_{\mathrm{MPO}} + V_{\mathrm{MPO}}. \tag{35}$$

10.1 TT-cross for the diagonal Morse potential

TT-cross extends matrix cross approximation to the tensor-train format and constructs a low-rank tensor from adaptively selected entries [10, 11]. Appendix B explains the implementation in detail. For $N_x = 2^L$,

$$j = \sum_{\ell=0}^{L-1} 2^{L-1-\ell} b_\ell, \qquad b_\ell \in \{0, 1\}, \tag{36}$$

so

$$x(b_0, \ldots, b_{L-1}) = x_{\min} + \Delta x \sum_{\ell=0}^{L-1} 2^{L-1-\ell} b_\ell. \tag{37}$$

TT-cross receives $V(x)$ only as a black-box function of these bits. It adaptively samples selected multi-indices and constructs

$$V_{b_0 \cdots b_{L-1}} \simeq G^{[0]}(b_0) G^{[1]}(b_1) \cdots G^{[L-1]}(b_{L-1}). \tag{38}$$

For a diagonal operator, each TT core is promoted directly to an MPO core,

$$W^{[\ell]}_{\alpha\beta,s,s'} = G^{[\ell]}_{\alpha,s,\beta} \delta_{ss'}. \tag{39}$$

Thus the TT ranks of the sampled potential become the MPO bond dimensions of V_{MPO}.

10.2 Exact binary-shift MPO for the kinetic energy

For the second-order finite-difference operator (Sec. B.9),

$$T = \frac{\hbar^2}{m\Delta x^2} I - \frac{\hbar^2}{2m\Delta x^2} (S_+ + S_-), \tag{40}$$

where

$$S_+ = \sum_{j=0}^{N_x - 2} |j+1\rangle \langle j|, \qquad S_- = S_+^\dagger. \tag{41}$$

On the binary chain S_+ is simply increment-by-one. With $\sigma_+ = |1\rangle\langle 0|$ and $\sigma_- = |0\rangle\langle 1|$,

$$S_+ = \sum_{k=0}^{L-1} \left(\bigotimes_{\ell<k} I_\ell \right) \otimes \sigma_+^{(k)} \otimes \left(\bigotimes_{\ell>k} \sigma_-^{(\ell)} \right). \tag{42}$$

Every term is a product operator and is assembled directly as an MPO (Sec. B.9). Therefore TT-cross is used where it is advantageous—the smooth diagonal potential—while the sparse analytic kinetic operator is represented exactly.

10.3 The $L = 8$ calculation as a matrix-free unit test

The matrix-free construction is first applied to the same $L = 8$, $N_x = 256$ problem used for dense validation. The dense eigensystem is consulted only afterward. In the reported notebook run,

$$\text{TT ranks}(V) = (1, 2, 4, 4, 4, 4, 4, 2, 1), \tag{43}$$

$$\chi_{\text{MPO}}(V) = 4, \tag{44}$$

$$\chi_{\text{MPO}}(T) = 17, \tag{45}$$

$$\chi_{\text{MPO}}(H) = 21. \tag{46}$$

TT-cross sampled 362 configurations and reported val_eps $= 7.59 \times 10^{-16}$. The ground-state calculation gave

$$E_0^{\text{mf}} = 1.87856075694, \tag{47}$$

only 2.10×10^{-8} above the exact finite-grid value, with fidelity 0.99999999965 and final MPS bond dimension 5. As in the dense-to-MPO calculation, `dmrg.solve` returned `False` for the strict stopping condition even though the physical error was already extremely small.

10.4 Matrix-free excited states

An MPS

$$\psi_{s_0 \cdots s_{L-1}} = A^{[0]s_0} \cdots A^{[L-1]s_{L-1}} \tag{48}$$

gives its projector directly as an MPO,

$$P^{[\ell]}_{(a,a'),(b,b'),s,t} = A^{[\ell]s}_{ab} \left(A^{[\ell]t}_{a'b'} \right)^* . \tag{49}$$

The projector bond dimension is at most χ^2, so the deflated Hamiltonian can remain matrix-free throughout.

Table 4: **Matrix-free TT-cross/DMRG validation for the first five Morse states.** The exact values are from the independent dense-grid benchmark and are not used by the matrix-free algorithm.

n	E_n^{grid}	E_n^{mf}	Error	Fidelity
0	1.878560736	1.878560754	1.809×10^{-8}	0.9999999997
1	5.155056237	5.155056733	4.960×10^{-7}	0.9999998970
2	7.791393954	7.791394433	4.792×10^{-7}	0.9999998627
3	9.789375168	9.789377680	2.512×10^{-6}	0.9999987244
4	11.149821257	11.149830853	9.596×10^{-6}	0.9999935593

10.5 What "matrix-free" means

The scalable path contains only a TT/QTT potential, binary-shift MPOs, the combined Hamiltonian MPO, MPS eigenstates, and MPS-projector MPOs. The optional $L = 20$ notebook cell,

$$N_x = 2^{20} = 1,048,576 \tag{50}$$

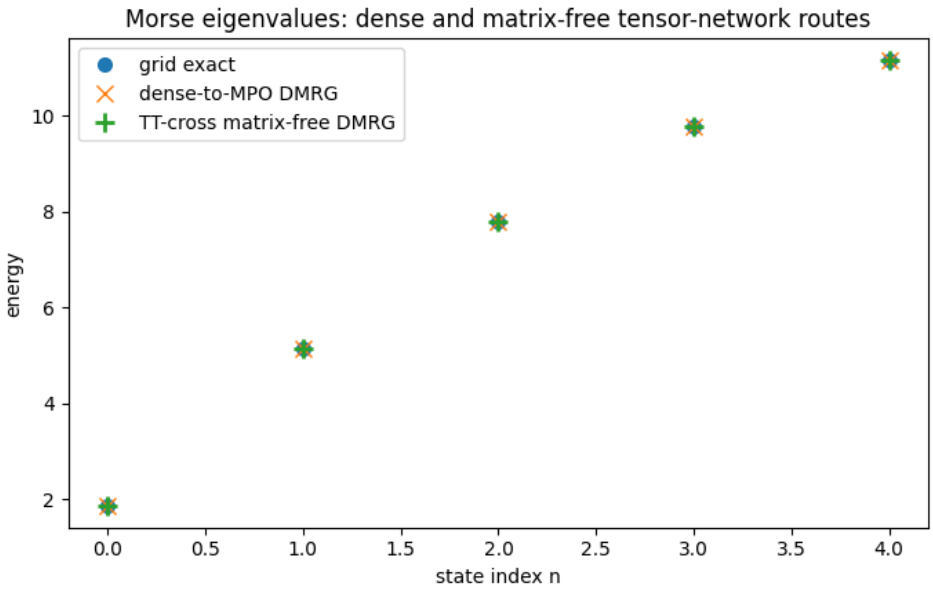

Figure 4: **One-dimensional Morse eigenvalues obtained by the three validation routes.** Exact finite-grid diagonalization, DMRG after dense-to-MPO conversion, and TT-cross matrix-free DMRG are visually indistinguishable on the scale of the spectrum.

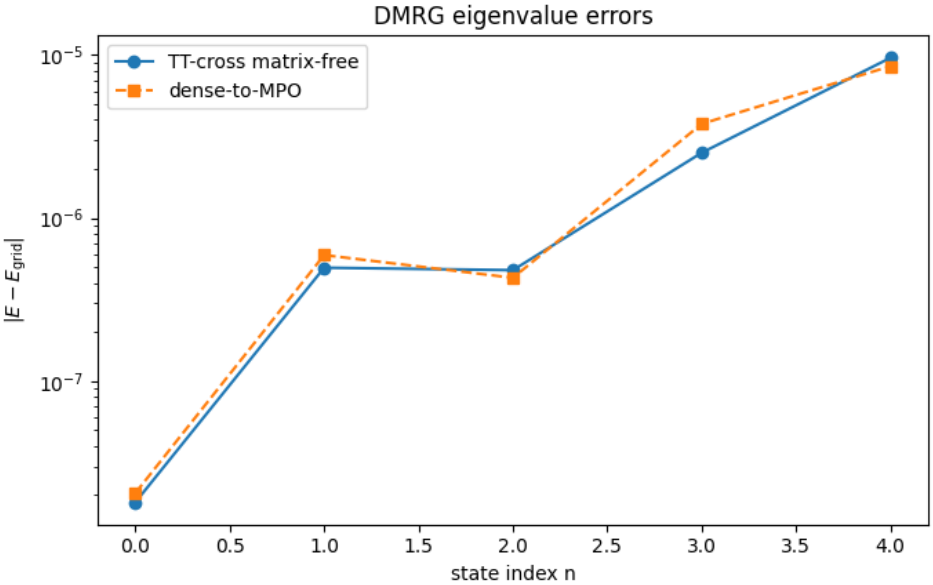

Figure 5: **DMRG errors relative to the exact finite-grid eigenvalues.** The matrix-free route reaches essentially the same accuracy as the dense-to-MPO route for the first five states.

is disabled by default, but its construction does not require either an N_x-component potential vector or an $N_x \times N_x$ Hamiltonian.

There are consequently two independent approximation layers:

$$\boxed{\text{operator compression (TT-cross)} \quad \neq \quad \text{state compression (DMRG/MPS).}} \tag{51}$$

They must be converged separately.

11 Direct two-coordinate TT-cross DMRG

The final calculation solves the combined Hamiltonian

$$H^{(2)} = -\frac{\hbar^2}{2m_1}\frac{\partial^2}{\partial x_1^2} - \frac{\hbar^2}{2m_2}\frac{\partial^2}{\partial x_2^2} + V_1(x_1) + V_2(x_2) \tag{52}$$

directly as one MPO. The model is mathematically separable, but the numerical calculation does *not* diagonalize separate one-dimensional grid Hamiltonians and does not construct the numerical spectrum by summing their eigenvalues. Separability is used only to obtain an external analytic benchmark.

11.1 Why the Morse parameters are changed

For the earlier choice $a_1 = 0.80$, $a_2 = 0.82$, the lowest dissociation channel enters before the twentieth bound–bound Morse sum. The notebook therefore uses slightly wider and still different wells,

$$D_{\mathrm{e},1} = 12.0, \qquad a_1 = 0.50, \tag{53}$$

$$D_{\mathrm{e},2} = 11.5, \qquad a_2 = 0.52, \tag{54}$$

with $m_1 = m_2 = \hbar = 1$, $x_{\mathrm{e},1} = x_{\mathrm{e},2} = 0$, and $x_1, x_2 \in [-3, 12]$. Each coordinate uses

$$L = 9, \qquad N_x = 512, \tag{55}$$

so the combined MPS contains 18 binary sites and represents $512^2 = 262,144$ coordinate configurations.

The analytic product energy is

$$E^{\mathrm{ana}}_{n_1 n_2} = E^{(1)}_{n_1} + E^{(2)}_{n_2}. \tag{56}$$

The first continuum threshold is

$$E_{\mathrm{cont}} = \min\left[D_{\mathrm{e},1} + E^{(2)}_0, D_{\mathrm{e},2} + E^{(1)}_0\right] = 12.693494871. \tag{57}$$

The twentieth analytic level is 11.877430379, leaving a 0.816064493 gap below the continuum. Thus a failure in the first-20 comparison cannot be attributed to continuum-state interleaving.

11.2 Fourth-order matrix-free kinetic energy

We now construct the two-coordinate Hamiltonian directly in tensor-network form, without first assembling the corresponding dense matrix. Each coordinate is discretized using L binary variables. The first L bits encode the grid index of coordinate x_1, while the remaining L bits encode the grid index of coordinate x_2:

$$j_1 = (b_0 \dots b_{L-1})_2, \qquad j_2 = (b_L \dots b_{2L-1})_2. \tag{58}$$

Thus, a single bit string $(b_0, \ldots, b_{2L-1})$ identifies one point (x_{1,j_1}, x_{2,j_2}) on the two-dimensional product grid. In the present example, $L = 9$, so the full configuration space is represented by 18 bits.

The potential energy is supplied to TT-cross as a black-box function,

$$V_{\mathrm{tot}}(x_1, x_2) = V_1(x_1) + V_2(x_2). \tag{59}$$

Rather than evaluating this function on all 2^{18} grid configurations, TT-cross queries only a small set of adaptively selected bit strings and constructs an MPO approximation to V_{tot} from those samples.

The kinetic-energy operator is constructed analytically. For each coordinate,

$$T_i = -\frac{\hbar^2}{2m_i}\frac{d^2}{dx_i^2}, \qquad i = 1, 2. \tag{60}$$

We approximate the second derivative using the fourth-order accurate five-point stencil,

$$\frac{d^2\psi_j}{dx^2} \simeq \frac{-\psi_{j+2} + 16\psi_{j+1} - 30\psi_j + 16\psi_{j-1} - \psi_{j-2}}{12\Delta x^2}, \tag{61}$$

whose truncation error is $\mathcal{O}(\Delta x^4)$. Unlike the usual three-point stencil, this approximation couples each grid point not only to its nearest neighbors, $j \pm 1$, but also to its next-nearest neighbors, $j \pm 2$.

Substituting Eq. (61) into the kinetic-energy operator gives

$$\begin{aligned} T_i = {} & \frac{5\hbar^2}{4m_i\Delta x_i^2} I - \frac{2\hbar^2}{3m_i\Delta x_i^2}\left(S_{i,+} + S_{i,-}\right) \\ & + \frac{\hbar^2}{24m_i\Delta x_i^2}\left(S_{i,+}^{(2)} + S_{i,-}^{(2)}\right), \end{aligned} \tag{62}$$

where I is the identity operator. The operators $S_{i,+}$ and $S_{i,-}$ shift the grid index of coordinate i by one point,

$$j_i \longmapsto j_i \pm 1, \tag{63}$$

whereas $S_{i,+}^{(2)}$ and $S_{i,-}^{(2)}$ implement the corresponding two-point shifts,

$$j_i \longmapsto j_i \pm 2. \tag{64}$$

Because the grid indices are stored in binary form, these shift operators can be constructed directly as MPOs implementing binary addition and subtraction. In particular, the ± 2 terms required by the fourth-order stencil are built analytically in the same way. Consequently, the kinetic-energy MPO is obtained without ever forming its dense finite-difference matrix.

For the run reported here, the two coordinates use the same grid spacing,

$$\Delta x_1 = \Delta x_2 = 0.0293542074. \tag{65}$$

TT-cross constructs the potential with MPO bond dimension

$$\chi_{\mathrm{MPO}}(V_{\mathrm{tot}}) = 4, \tag{66}$$

after sampling only 1042 configurations out of the full $2^{18} = 262144$ configuration space. The reported validation error is

$$\mathrm{val\_eps} = 2.51 \times 10^{-15}. \tag{67}$$

After combining the potential and kinetic-energy contributions, the resulting two-coordinate Hamiltonian has MPO bond dimension

$$\chi_{\mathrm{MPO}}(H^{(2)}) = 73. \tag{68}$$

These results illustrate the main advantage of the matrix-free construction: both the black-box potential and the finite-difference kinetic energy can be represented directly as MPOs, avoiding the storage and compression of the full dense Hamiltonian.

11.3 Deflation, restarts, and Rayleigh–Ritz refinement

For each of 20 targets the notebook uses

$$\chi_{\text{schedule}} = (8, 16, 32, 48, 64), \qquad \epsilon_{\text{MPS}} = 10^{-12}, \tag{69}$$

two random restarts, a six-sweep warm-up, and up to 22 refinement sweeps. The local sparse eigensolver is deliberately not oversolved during warm-up; an `ArpackNoConvergence` exception triggers continuation from the current MPS with a larger Krylov space.

After each accepted state, the code adds the matrix-free projector penalty

$$2\max(D_{\text{e},1}, D_{\text{e},2})\,|\psi_k\rangle\,\langle\psi_k|\,, \tag{70}$$

and compresses the deflated MPO. After all targets are found, it constructs

$$S_{ij} = \langle\psi_i|\psi_j\rangle\,, \qquad H_{ij} = \langle\psi_i|\,H^{(2)}_{\text{MPO}}\,|\psi_j\rangle \tag{71}$$

with the original undeflated Hamiltonian and solves the Rayleigh–Ritz problem in the 20-MPS span, i.e. a projected eigenvalue problem in the computed variational subspace [12]. The overlap matrix is well conditioned:

$$\lambda_{\min}(S) = 0.9869, \qquad \max_{i\neq j}|S_{ij}| = 9.15\times10^{-3}. \tag{72}$$

11.4 First 20 levels and the unresolved final target

Table 5: **Direct two-coordinate spectrum from the notebook.** The final column is $E_{\text{Ritz}} - E_{\text{ana}}$.

Rank	(n_1, n_2)	E_{ana}	E_{raw}	E_{Ritz}	$E_{\text{Ritz}} - E_{\text{ana}}$	conv.	bond
0	(0,0)	2.4066110675	2.4066108471	2.4066108471	-2.204×10^{-7}	T	6
1	(1,0)	4.6061008102	4.6069507917	4.6065043603	$+4.036\times10^{-4}$	T	9
2	(0,1)	4.6300434596	4.6306609109	4.6304063180	$+3.629\times10^{-4}$	T	8
3	(2,0)	6.5555905530	6.5558359254	6.5558696377	$+2.791\times10^{-4}$	T	11
4	(0,2)	6.5830758517	6.5843952841	6.5838343051	$+7.585\times10^{-4}$	T	9
5	(1,1)	6.8295332024	6.8300369818	6.8299806033	$+4.474\times10^{-4}$	T	13
6	(3,0)	8.2550802958	8.2558705623	8.2557752843	$+6.950\times10^{-4}$	T	14
7	(0,3)	8.2657082438	8.2668804495	8.2669713670	$+1.263\times10^{-3}$	T	10
8	(2,1)	8.7790229451	8.7795298733	8.7794868147	$+4.639\times10^{-4}$	T	15
9	(1,2)	8.7825655945	8.7831184621	8.7830933842	$+5.278\times10^{-4}$	T	14
10	(0,4)	9.6779406359	9.6787524230	9.6788637541	$+9.231\times10^{-4}$	T	11
11	(4,0)	9.7045700386	9.7063934540	9.7064883238	$+1.918\times10^{-3}$	T	16
12	(1,3)	10.4651979866	10.4668226747	10.4665980049	$+1.400\times10^{-3}$	T	16
13	(3,1)	10.4785126879	10.4798610182	10.4799168789	$+1.404\times10^{-3}$	F	21
14	(2,2)	10.7320553373	10.7326398921	10.7326667828	$+6.114\times10^{-4}$	T	15
15	(0,5)	10.8197730281	10.8241598615	10.8209677174	$+1.195\times10^{-3}$	T	12
16	(5,0)	10.9040597814	10.9093920215	10.9100566830	$+5.997\times10^{-3}$	T	15
17	(0,6)	11.6912054202	11.6990195239	11.7012591799	$+1.005\times10^{-2}$	T	12
18	(6,0)	11.8535495242	11.8801787635	11.8804206344	$+2.687\times10^{-2}$	T	15
19	(1,4)	11.8774303787	12.2976644796	12.2994741143	$+4.220\times10^{-1}$	T	14

For ranks 0–15, the refined errors are mostly 10^{-4}–10^{-3}. They rise to 5.997×10^{-3}, 1.005×10^{-2}, and 2.687×10^{-2} for ranks 16–18. The final rank has the much larger error

$$\left|E_{19}^{\text{Ritz}} - E_{19}^{\text{ana}}\right| = 0.4220437. \tag{73}$$

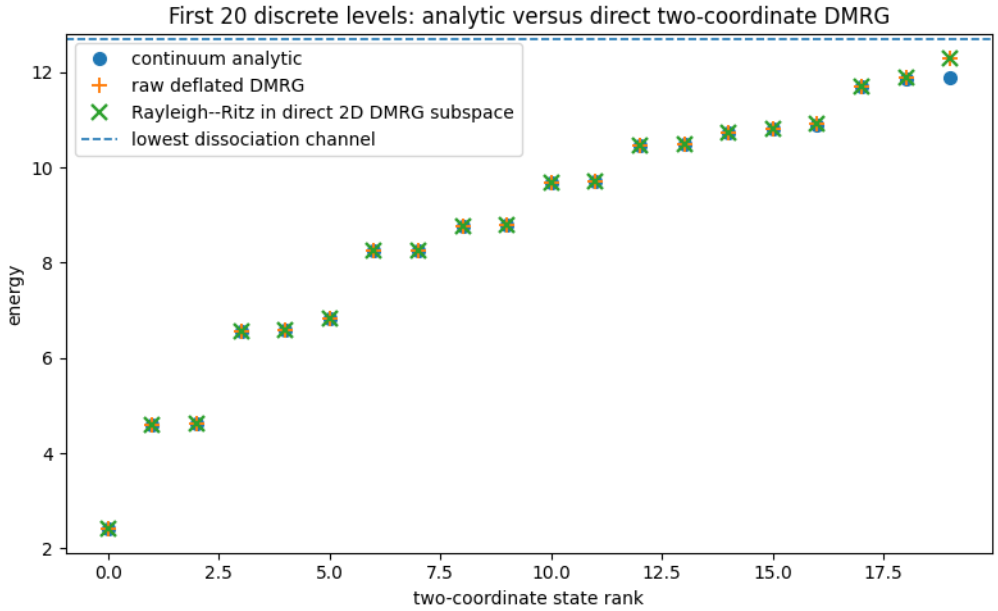

Figure 6: **First 20 analytic product levels and the direct two-coordinate DMRG spectrum.** The dashed line is the lowest dissociation channel. All 20 analytic reference levels lie below it.

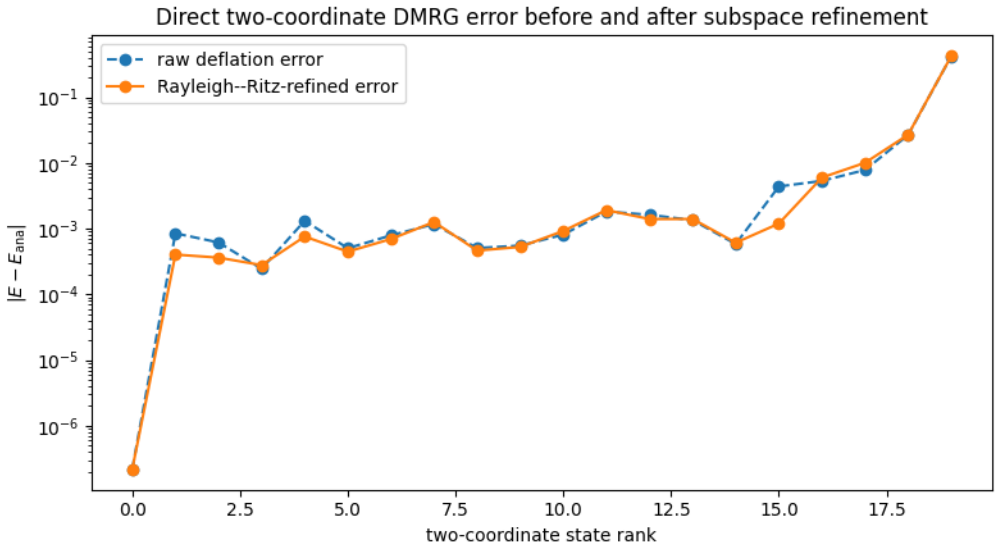

Figure 7: **Direct two-coordinate energy errors before and after Rayleigh–Ritz refinement.** The refinement improves combinations already represented in the MPS span but cannot recover the unresolved final low-energy state.

Consequently the notebook reports

$$\max|\Delta E| = 4.220437 \times 10^{-1}, \tag{74}$$

$$\mathrm{RMS}(\Delta E) = 9.460285 \times 10^{-2}, \tag{75}$$

$$\mathrm{mean}|\Delta E| = 2.388093 \times 10^{-2}. \tag{76}$$

This result should not be described as a uniformly converged first-20 spectrum. The continuum check has removed one possible explanation, and the well-conditioned overlap matrix shows that Rayleigh–Ritz itself is not numerically singular. The remaining interpretation supported by the notebook is that sequential excited-state targeting has not adequately represented one of the low-energy states in the 20-MPS subspace. Rayleigh–Ritz can recombine the states it is given; it cannot create a missing basis vector.

The notebook suggests increasing L_{pair} from 9 to 10 when a *systematic* residual shift persists after clean spectral convergence. For the present isolated high-rank failure, a complementary convergence test is to target more than 20 deflated states and retain the lowest 20 after Rayleigh–Ritz, or to increase the restart count before attributing the error to spatial discretization.

12 Reading the notebook as an algorithm

The notebook is best understood as three nested calculations of increasing generality.

1. **Dense $L = 8$ validation.** Construct and exactly diagonalize the second-order finite-difference matrix; then tensorize the same problem, convert it to an MPO, and validate DMRG energies, fidelities, deflation, and bond-dimension convergence.

2. **Matrix-free reconstruction of the same one-dimensional problem.** Replace dense-to-MPO conversion by TT-cross for the potential and exact binary-shift MPOs for the kinetic energy. Replace dense projector deflation by direct MPS-projector MPOs. The earlier dense eigensystem is used only as an external unit test.

3. **Direct combined two-coordinate calculation.** Concatenate two $L = 9$ binary coordinate blocks, sample $V_1(x_1) + V_2(x_2)$ by TT-cross on the full 18-bit domain, build fourth-order ± 1 and ± 2 kinetic shifts as MPOs, and target 20 states from the single combined Hamiltonian. Repeated DMRG restarts and MPO-projector deflation are followed by a Rayleigh–Ritz diagonalization of the original Hamiltonian in the computed MPS span.

The order is pedagogically useful: the dense problem establishes correctness, the matrix-free one-dimensional problem establishes scalable operator construction, and the two-coordinate calculation exposes the additional difficulty of obtaining a complete excited-state subspace.

13 Numerical lessons and limitations

Keep the benchmark hierarchy explicit. The analytic Morse spectrum tests the physical continuum problem against the finite grid; exact grid diagonalization tests DMRG; TT-cross introduces a separate operator-representation error.

Operator compression and state compression are independent. TT-cross controls the Hamiltonian representation, while DMRG controls the MPS eigenstate representation. Increasing the DMRG bond dimension cannot repair an inaccurate TT-cross potential.

A Boolean convergence flag is not a physical accuracy measure. The one-dimensional DMRG runs can report `converged=False` while energies and fidelities are already excellent. Conversely, most two-coordinate targets report `converged=True`, yet the twentieth physical level is not captured correctly.

Excited-state deflation is the fragile step. MPS-projector deflation keeps the calculation matrix-free, but projector errors accumulate and the deflated MPO grows. Restarts and overlap rejection improve robustness without guaranteeing that every low-energy state is found.

Rayleigh–Ritz fixes nonorthogonality, not missing states. The 20-state overlap matrix is well conditioned, yet the final refined level remains wrong by about 0.422. A subspace method cannot recover an eigenvector that is absent from the subspace.

Continuum placement must be checked before interpreting high excited states. The adjusted two-coordinate parameters place all 20 analytic reference levels below the first dissociation channel. This prevents continuum-box states from being mistaken for DMRG failures.

Higher-order finite differences address a different error source. The fourth-order stencil and finer $L = 9$ grid reduce spatial-discretization error but do not solve excited-state targeting errors.

The two-coordinate benchmark is direct but still uncoupled. The code solves the combined 18-site MPO rather than adding numerical one-dimensional spectra, but $V_1(x_1)+V_2(x_2)$ remains separable. A coupled potential is the next test of inter-coordinate entanglement.

14 Suggested next step: a genuinely coupled vibrational problem

The notebook now already performs a direct two-coordinate DMRG calculation, but the Hamiltonian remains separable. A minimal genuinely correlated extension is

$$H_{\text{coupled}} = H_1 + H_2 + \kappa x_1 x_2. \tag{77}$$

The direct uncoupled two-coordinate calculation provides a controlled $\kappa \to 0$ reference. As κ is increased, one can monitor

1. convergence of the lowest eigenvalues with MPS bond dimension;
2. the Schmidt spectrum between the two oscillator coordinate blocks;
3. the entanglement entropy;
4. overlaps with the uncoupled product basis;
5. the cost and accuracy of direct MPO construction relative to dense conversion.

Such a model retains the pedagogical simplicity of Morse coordinates while introducing exactly the feature for which DMRG is designed: nontrivial correlations in a tensor-product Hilbert space.

A How the `quimb` tensor-network routines work

The numerical implementation developed in this tutorial relies on the `quimb` tensor-network library [9], in particular three central routines from the `quimb.tensor` module:

$$\boxed{\texttt{MatrixProductOperator.from\_dense} \longrightarrow \texttt{DMRG2} \longrightarrow \texttt{dmrg.solve}.} \tag{78}$$

These compact Python commands hide several important tensor-network operations. The purpose of this appendix is to explain what each routine does, how its arguments should be interpreted, and how the corresponding numerical operations relate to the MPS and MPO formalism introduced in the main text.

The overall computational sequence is

$$H_{\text{dense}} \longrightarrow H_{\text{MPO}} \longrightarrow |\Psi_{\text{MPS}}\rangle \longrightarrow \text{variational DMRG optimization.} \tag{79}$$

A.1 `MatrixProductOperator.from_dense`: converting a dense Hamiltonian into an MPO

The finite-difference Morse Hamiltonian is initially represented by the ordinary dense matrix

$$H \in \mathbb{R}^{256\times 256}. \tag{80}$$

Because

$$256 = 2^8, \tag{81}$$

the 256-dimensional coordinate-grid Hilbert space can be written as

$$\mathcal{H} \simeq \underbrace{\mathbb{C}^2 \otimes \mathbb{C}^2 \otimes \cdots \otimes \mathbb{C}^2}_{L=8 \text{ factors}}. \tag{82}$$

The conversion performed in the notebook is

```
H_mpo = qtn.MatrixProductOperator.from_dense(
    H,
    dims=[2] * L,
    cutoff=1e-12,
)
```

The first argument,

```
H
```

is the dense operator to be converted.

The argument

```
dims=[2] * L
```

specifies how the 256-dimensional index is to be factorized. For $L = 8$, this is equivalent to

$$\texttt{dims} = [2, 2, 2, 2, 2, 2, 2, 2], \tag{83}$$

and therefore

$$\prod_{\ell=0}^{L-1} d_\ell = 2^8 = 256. \tag{84}$$

A conventional matrix element

$$H_{j,j'} \tag{85}$$

is consequently reinterpreted as the higher-rank tensor

$$H_{b_0 b_1 \cdots b_{L-1},\, b'_0 b'_1 \cdots b'_{L-1}}, \tag{86}$$

where

$$j = (b_0 b_1 \cdots b_{L-1})_2, \tag{87}$$
$$j' = (b'_0 b'_1 \cdots b'_{L-1})_2. \tag{88}$$

Each binary variable satisfies

$$b_\ell, b'_\ell \in \{0, 1\}. \tag{89}$$

Thus, each artificial site of the MPO has two physical indices: one associated with the output index of the operator and one associated with its input index.

The tensor in Eq. (86) is factorized into an MPO of the form

$$H_{\mathbf{b},\mathbf{b}'} = \sum_{\alpha_1,\ldots,\alpha_{L-1}} W^{[0]}_{b_0 b'_0,\alpha_1} W^{[1]}_{\alpha_1,b_1 b'_1,\alpha_2} \times \cdots W^{[L-1]}_{\alpha_{L-1},b_{L-1} b'_{L-1}}. \tag{90}$$

The indices

$$\alpha_1, \ldots, \alpha_{L-1} \tag{91}$$

are the internal MPO bond indices. Their dimensions determine how much operator information must be transmitted between neighboring binary sites.

Schematically, a bulk MPO tensor has dimensions

$$D_{\ell-1} \times 2 \times 2 \times D_\ell, \tag{92}$$

where $D_{\ell-1}$ and D_ℓ are MPO bond dimensions.

The decomposition can be understood in terms of successive matrix factorizations. At each partition, the operator tensor is reshaped into a matrix and decomposed schematically as

$$M = USV^\dagger. \tag{93}$$

The resulting factors are reorganized into neighboring MPO tensors, and the procedure is repeated along the binary chain.

The option

```
cutoff=1e-12
```

controls the tensor compression performed during this decomposition. Very small singular components can be discarded according to the splitting criterion used internally by `quimb`. Thus,

$$H_{\text{dense}} \simeq H_{\text{MPO}}. \tag{94}$$

For the Morse Hamiltonian used in this tutorial, the resulting MPO has maximum bond dimension

$$D_{\max} = 5. \tag{95}$$

The original 256×256 Hamiltonian therefore possesses a compact MPO representation after binary tensorization.

It is important, however, to distinguish MPO compression from a genuinely scalable construction. In the present tutorial the full dense Hamiltonian is formed *before* calling `MatrixProductOperator.from_dense`. The calculation therefore demonstrates the MPO representation clearly, but it does not yet avoid the memory cost of constructing H_{dense}. In a large-scale calculation, the kinetic and potential contributions should instead be constructed directly as MPOs.

A.2 `qtn.DMRG2`: initializing a two-site DMRG calculation

After the Hamiltonian has been converted to an MPO, the DMRG calculation is initialized with

```
dmrg = qtn.DMRG2(
    mpo,
    bond_dims=list(bond_dims),
    cutoffs=cutoff,
    p0=p0,
)
```

The call to `DMRG2` constructs the DMRG object and stores the Hamiltonian, the initial MPS, the allowed bond dimensions, and the MPS truncation parameters. The actual sequence of optimization sweeps is performed later by `dmrg.solve()`.

A.2.1 Why is it called `DMRG2`?

The suffix "2" indicates a *two-site* DMRG algorithm.

Consider an MPS written schematically as

$$\Psi_{s_0 s_1 \cdots s_{L-1}} = A^{[0]s_0} A^{[1]s_1} \cdots A^{[L-1]s_{L-1}}. \tag{96}$$

During one local step of two-site DMRG, two neighboring tensors, $A^{[i]}$ and $A^{[i+1]}$, are temporarily combined into a single two-site tensor,

$$\Theta^{s_i s_{i+1}}_{\alpha\beta}. \tag{97}$$

All tensors outside these two sites are held fixed. Their contraction with the MPO forms the left and right environments. The full variational problem is thereby reduced to a local effective eigenvalue equation,

$$H_{\text{eff}}\, \boldsymbol{\theta} = E_{\text{loc}}\, \boldsymbol{\theta}, \tag{98}$$

where $\boldsymbol{\theta}$ is the vectorized two-site tensor.

The lowest eigenvector of H_{eff} supplies the optimized local tensor.

The optimized tensor is then reshaped as

$$\Theta_{(\alpha s_i),(s_{i+1}\beta)} \tag{99}$$

and factorized through an SVD,

$$\Theta = USV^\dagger. \tag{100}$$

The matrices U and $V^\dagger$, together with the singular values in S, are used to reconstruct two separate neighboring MPS tensors.

This splitting step is also where the MPS bond dimension can be truncated. The procedure is repeated successively along the chain.

A.2.2 The initial MPS `p0`

Before constructing the DMRG object, the notebook generates an initial random state:

```
p0 = qtn.MPS_rand_state(
    L=mpo.L,
    bond_dim=2,
    phys_dim=2,
    normalize=True,
)
```

For the Morse calculation,

$$L = 8, \qquad d = 2, \qquad \chi_{\text{initial}} = 2. \tag{101}$$

Here $d = 2$ is the local physical dimension of each binary site, while χ_{initial} is the initial MPS bond dimension.

The random MPS is only a starting guess:

$$|\Psi^{(0)}\rangle = |p_0\rangle\,. \tag{102}$$

During the DMRG sweeps, its tensor elements are varied to minimize the energy.

A.2.3 The argument `bond_dims`

The tutorial uses

```
bond_dims=(4, 8, 16)
```

which is supplied to `DMRG2` as

```
bond_dims=[4, 8, 16]
```

This specifies a sequence of maximum allowed MPS bond dimensions for successive sweeps. Schematically,

$$\text{first sweep}: \quad \chi_{\max} = 4, \tag{103}$$

$$\text{second sweep}: \quad \chi_{\max} = 8, \tag{104}$$

$$\text{third and later sweeps}: \quad \chi_{\max} = 16. \tag{105}$$

Once the supplied sequence has been exhausted, the final value is reused for later sweeps.

The quantity $\chi_{\max}$ is only an upper bound. It does not require the optimized state to use that entire bond dimension. If the singular-value spectrum is sufficiently compressible, the actual MPS bond dimension can be considerably smaller.

For example, output of the form

```
max_bond=(5/16)
```

means that a bond dimension as large as 16 was allowed, whereas the actual optimized MPS required only a maximum bond dimension of 5.

This behavior is precisely what one hopes for in a tensor-network calculation: the allowed variational space can be large while the optimized state remains compact.

A.2.4 The argument `cutoffs`

The DMRG object is constructed with

```
cutoffs=1e-10
```

This cutoff acts during the SVD in Eq. (100), when the optimized two-site tensor is split back into two MPS tensors. Small singular-value contributions can be discarded.

Thus,

$$\texttt{cutoffs} = 10^{-10} \tag{106}$$

controls the compression of the *MPS during DMRG*.

This is distinct from

$$\texttt{cutoff} = 10^{-12} \tag{107}$$

used by `MatrixProductOperator.from_dense`, which controls the compression of the *Hamiltonian MPO*.

A.2.5 Which eigenvalue does DMRG target?

The two-site DMRG implementation targets the lowest-energy state by default. The underlying DMRG machinery uses the choice

$$\texttt{which='SA'}, \tag{108}$$

where "SA" denotes the smallest algebraic eigenvalue.

For a Hermitian Hamiltonian, this corresponds to the ground-state variational problem

$$E_0 = \min_{\Psi} \frac{\langle \Psi | H | \Psi \rangle}{\langle \Psi | \Psi \rangle}. \tag{109}$$

This also explains the deflation strategy used in the main text. Instead of requesting the second or third eigenstate directly, previously found states are shifted upward in energy so that the next desired physical state becomes the ground state of the modified Hamiltonian.

A.3 `dmrg.solve`: performing the optimization sweeps

Once the DMRG object has been initialized, the actual variational optimization is started with

```
converged = dmrg.solve(
    tol=tol,
    max_sweeps=max_sweeps,
    verbosity=verbosity,
)
```

In the notebook,

$$\texttt{tol} = 10^{-10}, \qquad \texttt{max\_sweeps} = 12. \tag{110}$$

The method `solve()` repeatedly performs DMRG sweeps over the MPS. At each bond, it

1. constructs the effective two-site Hamiltonian H_{eff};
2. solves its lowest-eigenvalue problem;
3. inserts the optimized two-site state;
4. splits it using an SVD;
5. truncates the MPS bond if necessary;
6. updates the tensor-network environment;
7. moves to the neighboring bond.

For an L-site chain, a sweep contains $L-1$ two-site optimization positions. In the present case,

$$L = 8, \tag{111}$$

so each directional pass contains

$$L - 1 = 7 \tag{112}$$

two-site optimization steps.

A sweep may be pictured schematically as

$$(0,1) \longrightarrow (1,2) \longrightarrow \cdots \longrightarrow (6,7), \tag{113}$$

followed by a sweep in the opposite direction.

A.3.1 Reading the printed DMRG output

A typical line from the calculation has the form

```
2, R, max_bond=(4/8), cutoff:1e-10
```

This can be interpreted as follows.

The leading integer identifies the sweep number. The symbol

$$\texttt{R} \tag{114}$$

denotes the current sweep direction. The expression

$$\texttt{max\_bond=(4/8)} \tag{115}$$

means that the current MPS has maximum bond dimension 4 while a maximum bond dimension of 8 is permitted during that sweep.

Finally,

$$\texttt{cutoff:1e-10} \tag{116}$$

reports the MPS compression cutoff used during the two-site splitting.

A.3.2 The convergence tolerance `tol`

The argument

```
tol=1e-10
```

controls convergence of the DMRG sweeps.

The criterion is based on the absolute change in energy between successive completed sweeps. Schematically, convergence requires

$$\left|E^{(s)} - E^{(s-1)}\right| < \epsilon_E, \tag{117}$$

where

$$\epsilon_E = \texttt{tol}. \tag{118}$$

Thus, in the tutorial,

$$\epsilon_E = 10^{-10}. \tag{119}$$

This tolerance should not be confused with either the MPO compression cutoff or the MPS singular-value truncation cutoff.

The calculation stops when either the energy criterion in Eq. (117) is satisfied or the maximum number of sweeps,

$$\texttt{max\_sweeps} = 12, \tag{120}$$

has been reached.

The variable

```
converged
```

returned by `solve()` is therefore Boolean:

$$\texttt{True} \quad \text{or} \quad \texttt{False}. \tag{121}$$

It answers the specific numerical question:

> Did the change in energy between successive DMRG sweeps fall below the requested value of `tol` before the maximum number of sweeps was reached?

It does not by itself measure the absolute error in the energy or wavefunction.

A.3.3 Why can `converged=False` still correspond to an accurate state?

For the ground-state calculation in the notebook, the final diagnostics are approximately

$$E_0^{\mathrm{DMRG}} = 1.8785607568, \tag{122}$$

$$E_0^{\mathrm{grid}} = 1.8785607359, \tag{123}$$

$$\left|E_0^{\mathrm{DMRG}} - E_0^{\mathrm{grid}}\right| \simeq 2.1 \times 10^{-8}, \tag{124}$$

$$F_0 = \left|\langle \psi_0^{\mathrm{grid}} | \psi_0^{\mathrm{DMRG}} \rangle\right|^2 \simeq 0.9999999997. \tag{125}$$

Nevertheless, the call to `solve()` can return

$$\texttt{False}. \tag{126}$$

There is no contradiction. The requested convergence threshold,

$$10^{-10}, \tag{127}$$

is substantially smaller than the residual sweep-to-sweep energy variation. The algorithm therefore correctly reports that this particularly strict stopping criterion was not reached within the permitted number of sweeps.

At the same time, direct comparison with exact diagonalization shows that the resulting MPS already represents the ground state with extremely high accuracy.

This illustrates an important practical lesson: DMRG convergence should be assessed using several diagnostics, including

$$\text{energy stability}, \qquad \text{bond-dimension convergence}, \qquad \text{truncation sensitivity}, \tag{128}$$

and, when possible,

$$\text{comparison with an independent benchmark}. \tag{129}$$

A.4 Accessing the optimized MPS and its energy

After `dmrg.solve()` has completed, the optimized MPS is stored in

```
dmrg.state
```

and the DMRG energy is available as

```
dmrg.energy
```

The notebook makes a copy of the optimized MPS through

```
psi = dmrg.state.copy()
```

For validation against the dense coordinate-grid result, the MPS is contracted into an ordinary vector:

```
def mps_to_vector(psi):
    v = np.asarray(psi.to_dense()).reshape(-1)
    v = v / np.linalg.norm(v)
    return v
```

Conceptually,

$$A^{[0]} A^{[1]} \cdots A^{[L-1]} \longrightarrow \Psi_{b_0 b_1 \cdots b_{L-1}} \longrightarrow \psi_j. \tag{130}$$

For the eight-site binary representation this produces

$$2^8 = 256 \tag{131}$$

coefficients, exactly matching the original coordinate grid.

The notebook additionally evaluates the energy using the original dense Hamiltonian:

```
E0_dmrg = np.vdot(psi0, H @ psi0).real
```

which computes

$$E_{\text{check}} = \frac{\langle \psi_{\text{DMRG}} | \, H_{\text{dense}} \, | \psi_{\text{DMRG}} \rangle}{\langle \psi_{\text{DMRG}} | \psi_{\text{DMRG}} \rangle}. \tag{132}$$

Agreement between this value and `dmrg.energy` provides an additional check that the compressed MPO and the original dense Hamiltonian represent the same physical operator to the required accuracy.

A.5 Three numerical tolerances with different meanings

The implementation contains three parameters that are numerically small but control fundamentally different approximations. They should not be confused.

Table 6: **Distinct numerical accuracy controls used in the tutorial.**

Parameter	Value	Meaning
`from_dense cutoff`	10^{-12}	Controls compression of the dense Hamiltonian when it is converted into an MPO.
`DMRG2 cutoffs`	10^{-10}	Controls singular-value truncation when the optimized two-site tensor is split back into two MPS tensors.
`solve tol`	10^{-10}	Controls convergence of the DMRG sweeps through the change in energy between successive sweeps.

The computational hierarchy can therefore be summarized as

$$\boxed{\begin{array}{c} H_{\text{dense}} \\ \downarrow \quad \text{MPO decomposition cutoff} \\ H_{\text{MPO}} \\ \downarrow \quad \text{DMRG2 optimization and MPS truncation} \\ |\Psi_{\text{MPS}}\rangle \\ \downarrow \quad \text{sweep-to-sweep energy tolerance} \\ E_0, \ |\Psi_0\rangle \end{array}} \tag{133}$$

In other words, the first cutoff controls how accurately the *operator* is represented, the second controls how accurately the *wavefunction* is retained during local tensor splitting, and the third controls when the iterative DMRG optimization is considered complete.

A.6 Summary of the complete DMRG call

The central implementation used in the notebook can now be read line by line:

```
def run_dmrg(mpo, bond_dims=(4, 8, 16),
             cutoff=1e-10,
             tol=1e-10,
             max_sweeps=12,
             verbosity=1):

    p0 = qtn.MPS_rand_state(
        L=mpo.L,
        bond_dim=2,
        phys_dim=2,
        normalize=True,
    )

    dmrg = qtn.DMRG2(
        mpo,
        bond_dims=list(bond_dims),
        cutoffs=cutoff,
        p0=p0,
    )

    converged = dmrg.solve(
        tol=tol,
        max_sweeps=max_sweeps,
        verbosity=verbosity,
    )

    psi = dmrg.state.copy()
    vec = mps_to_vector(psi)

    return converged, dmrg, psi, vec
```

The sequence is therefore:

1. construct a random normalized MPS;
2. associate that MPS with the Hamiltonian MPO;
3. permit its bond dimension to grow through the sequence $4 \to 8 \to 16$;
4. optimize pairs of neighboring tensors with two-site DMRG;
5. compress the MPS after each local optimization;
6. continue sweeping until the energy-change criterion is satisfied or the maximum number of sweeps is reached;
7. return the optimized MPS and its energy;
8. contract the MPS back to the original 256-component grid representation for direct validation.

This makes explicit that the Python interface is only a compact wrapper around the variational tensor-network procedure developed in the main text.

B Matrix-free Hamiltonian construction with TT-cross

Section 10 presents the matrix-free route as part of the main notebook workflow. This appendix explains the construction at the implementation level. The diagonal Morse potential is approximated by TT-cross using selected black-box function evaluations, whereas the finite-difference kinetic operator is assembled analytically from binary increment and decrement MPOs.

The contrast with the dense validation path is

$$N_x = 2^L \quad \Longrightarrow \quad H_{\text{dense}} \text{ contains } 4^L \text{ entries}, \tag{134}$$

whereas the matrix-free route stores TT/MPO cores whose cost is governed by their internal ranks. The same routines are generalized in Section 11 to two coordinate blocks and a fourth-order finite-difference stencil.

B.1 From tensor trains to quantized tensor trains

Consider first a vector sampled on

$$N_x = 2^L \tag{135}$$

grid points,

$$v_j, \qquad j = 0, \ldots, 2^L - 1. \tag{136}$$

The binary representation

$$j = \sum_{\ell=0}^{L-1} 2^{L-1-\ell} b_\ell, \qquad b_\ell \in \{0, 1\}, \tag{137}$$

allows the same vector to be regarded as an order-L tensor,

$$v_j \longleftrightarrow \mathcal{V}_{b_0 b_1 \cdots b_{L-1}}, \tag{138}$$

whose dimensions are

$$2 \times 2 \times \cdots \times 2. \tag{139}$$

A tensor train represents this tensor as

$$\mathcal{V}_{b_0 b_1 \cdots b_{L-1}} = \sum_{\alpha_1, \ldots, \alpha_{L-1}} G^{[0]}_{1, b_0, \alpha_1} G^{[1]}_{\alpha_1, b_1, \alpha_2} \cdots G^{[L-1]}_{\alpha_{L-1}, b_{L-1}, 1}. \tag{140}$$

Each core therefore has dimensions

$$G^{[\ell]} \in \mathbb{R}^{r_{\ell-1} \times 2 \times r_\ell}, \tag{141}$$

with

$$r_0 = r_L = 1. \tag{142}$$

Because the tensorization is obtained by repeatedly splitting a single large physical index into binary indices, this representation is often called a *quantized tensor train*, or QTT [4, 5]. The binary MPS representation used throughout this tutorial is therefore mathematically the same tensor structure as a QTT representation.

B.2 Why an ordinary TT decomposition is not enough

Suppose that all 2^L values of $\mathcal{V}$ were already known. One could then apply a sequence of singular-value decompositions to obtain Eq. (140). This procedure is generally called TT-SVD and is one of the basic constructive algorithms underlying the TT format [13].

The difficulty is that TT-SVD requires access to the complete tensor. Thus, even though the final TT representation may contain only

$$O(Lr^2) \tag{143}$$

parameters, one would first have to evaluate and store all 2^L grid values. TT-cross removes this requirement.

B.3 The basic idea of cross approximation

The idea originates from low-rank matrix approximation and its extension to TT-cross [10]. Consider a matrix A that is approximately of rank r. Rather than performing an SVD of every matrix element, one can select informative rows and columns and form the cross approximation, [1]

$$A \simeq A(:, J)\,[A(I, J)]^{-1}\,A(I, :), \tag{144}$$

where I and J contain selected row and column indices. Thus the complete matrix can be approximated using only a subset of its entries.

The quality of the approximation depends strongly on the choice of $A(I, J)$. Cross algorithms therefore seek a well-conditioned submatrix, often described as one having large or approximately maximal *volume*,

$$\mathrm{vol}\,[A(I, J)] = |\det A(I, J)|\,. \tag{145}$$

TT-cross extends this principle from matrices to multidimensional tensors. Rather than evaluating

$$\mathcal{V}_{b_0 \cdots b_{L-1}} \tag{146}$$

for every one of the 2^L bit strings, the algorithm adaptively chooses selected combinations of left and right multi-indices and evaluates only the corresponding tensor fibers. Schematically, at site k it samples objects of the form

$$\mathcal{V}\,(I_{<k}, b_k, I_{>k})\,, \tag{147}$$

where $I_{<k}$ and $I_{>k}$ denote selected combinations of binary indices to the left and right of site k. The selected index sets are then updated as the algorithm sweeps through the tensor.

The resulting procedure has an important property: the function that defines the tensor need only be evaluated at the selected points. The complete tensor never needs to exist.

B.4 Cost of TT-cross

Assume, for simplicity, that every tensor mode has dimension n and that all TT ranks are bounded by

$$r_k \le r. \tag{148}$$

A TT representation contains approximately

$$O(Lnr^2) \tag{149}$$

parameters. A standard TT-cross construction similarly requires a number of sampled tensor entries that grows polynomially with L, n, and the TT rank rather than exponentially with L. A commonly quoted estimate is

$$N_{\text{samples}} \sim O(Lnr^2), \tag{150}$$

with linear-algebra work of approximately

$$O(Lnr^3), \tag{151}$$

for bounded ranks.

[1] For an exact rank-r matrix A, choose r linearly independent columns indexed by J and r rows indexed by I such that $A(I, J)$ is nonsingular. Since the selected columns span the column space of A, one may write $A = A(:, J)X$. Restricting this relation to the rows I gives $A(I, :) = A(I, J)X$, and therefore $X = [A(I, J)]^{-1}A(I, :)$. Substitution yields $A = A(:, J)[A(I, J)]^{-1}A(I, :)$. For a matrix that is only approximately rank r, the same argument leads to the cross approximation $A \simeq A(:, J)[A(I, J)]^{-1}A(I, :)$. The accuracy and numerical stability depend on selecting informative rows and columns such that the cross matrix $A(I, J)$ is well conditioned.

In the present binary representation,

$$n = 2, \tag{152}$$

so the estimates reduce to

$$N_{\text{samples}} \sim O(Lr^2), \tag{153}$$
$$\text{work} \sim O(Lr^3). \tag{154}$$

The exponential dependence 2^L is therefore replaced by a polynomial dependence on L, provided the TT ranks remain moderate. This final qualification is essential: TT-cross is useful only when the target tensor is compressible in TT form.

B.5 Applying TT-cross to the Morse potential

We consider the Morse oscillator,

$$V(x) = D_{\text{e}} \left[1 - e^{-a(x-x_{\text{e}})}\right]^2, \tag{155}$$

on the finite grid,

$$x_j = x_{\min} + j\Delta x, \tag{156}$$

where

$$\Delta x = \frac{x_{\max} - x_{\min}}{2^L - 1}. \tag{157}$$

Using Eq. (137), the potential may be regarded directly as a function of the binary variables:

$$\mathcal{V}(b_0, \ldots, b_{L-1}) = D_{\text{e}} \left[1 - \exp\left(- a \left[x_{\min} + \Delta x \sum_{\ell=0}^{L-1} 2^{L-1-\ell} b_\ell - x_{\text{e}} \right] \right)\right]^2. \tag{158}$$

Equation (158) is particularly well suited to TT-cross: given any binary string, its value can be evaluated immediately without constructing the entire coordinate grid.

B.6 Implementation with `tntorch.cross`

A convenient Python implementation of TT-cross is provided by the `tntorch` package [14]. A Colab installation is

```
!pip -q install tntorch quimb
```

and the required imports are

```
import numpy as np
import torch
import tntorch as tn
import quimb.tensor as qtn

from quimb.tensor.tensor_builder import MPO_product_operator
```

We first specify the physical problem:

```
hbar = 1.0
m = 1.0

De = 12.0
a = 0.8
xe = 0.0
```

```

L = 20
Nx = 2**L

xmin = -2.5
xmax = 10.0

dx = (xmax - xmin) / (Nx - 1)
```

Notice that for $L = 20$,

$$N_x = 1,048,576 \tag{159}$$

but no vector containing one million potential values is created.

The binary domain consists simply of $b_\ell \in \{0, 1\}$:

```
domain = [
    torch.tensor([0.0, 1.0], dtype=torch.float64)
    for _ in range(L)
]
```

The binary weights appearing in Eq. (137) are

```
weights = torch.tensor(
    [2**(L - 1 - k) for k in range(L)],
    dtype=torch.float64,
)
```

We then define a function that evaluates the Morse potential only at the binary configurations requested by TT-cross:

```
def morse_from_bits(B):
    """
    B has shape (P, L), where P is the number
    of points requested by TT-cross.
    """

    j = B @ weights
    x = xmin + dx * j

    return De * (
        1.0 - torch.exp(-a * (x - xe))
    )**2
```

The crucial point is that this function accepts only the selected binary configurations requested by the cross algorithm. It does not evaluate $V(x_j)$ for every j.

The TT approximation is generated with

```
V_tt, info = tn.cross(
    function=morse_from_bits,
    domain=domain,
    function_arg="matrix",
    eps=1e-10,
    rmax=32,
    max_iter=30,
    val_size=1000,
    verbose=True,
    return_info=True,
)
```

The principal parameters have the following meanings:

`eps` Target validation accuracy of the cross approximation.

`rmax` Maximum TT rank permitted during adaptive rank growth.

`max_iter` Maximum number of cross iterations.

`val_size` Number of sampled configurations used to estimate or monitor the approximation error.

`return_info=True` Return diagnostic information in addition to the TT tensor.

The result `V_tt` contains TT cores

$$G^{[k]} \in \mathbb{R}^{r_{k-1} \times 2 \times r_k}. \tag{160}$$

Importantly, no array of length 2^L has been constructed.

B.7 Turning the TT potential into a diagonal MPO

The potential operator is diagonal in coordinate space:

$$\hat{V} = \sum_j V_j \left|j\right\rangle \left\langle j\right|. \tag{161}$$

In binary notation,

$$V_{\mathbf{b},\mathbf{b}'} = \mathcal{V}_{\mathbf{b}} \delta_{\mathbf{b},\mathbf{b}'}. \tag{162}$$

Therefore each TT core can be converted directly into an MPO core by adding a second physical index:

$$W^{[k]}_{\alpha\beta,s,s'} = G^{[k]}_{\alpha,s,\beta} \delta_{s,s'}. \tag{163}$$

This transformation is exact and does not increase the TT/MPO bond ranks.

The corresponding Python routine is

```
def tt_to_diagonal_mpo(tt):
    """
    Convert a tntorch TT vector into a quimb
    diagonal MatrixProductOperator.
    """

    arrays = []
    cores = tt.cores
    Ltt = len(cores)

    for k, core_torch in enumerate(cores):

        G = core_torch.detach().cpu().numpy()
        rleft, d, rright = G.shape

        W = np.zeros(
            (rleft, rright, d, d),
            dtype=G.dtype,
        )

        for s in range(d):
            W[:, :, s, s] = G[:, s, :]

        # quimb omits the absent outer bond index
        # on the first and last MPO tensors.
```

```
        if k == 0:
            W = W[0, :, :, :]
        elif k == Ltt - 1:
            W = W[:, 0, :, :]

        arrays.append(W)

    return qtn.MatrixProductOperator(
        arrays,
        shape="lrud",
    )
```

The potential MPO is then obtained directly as

```
V_mpo = tt_to_diagonal_mpo(V_tt)

print("Potential MPO max bond:",
      V_mpo.max_bond())
```

The sequence

$$V(b_0, \ldots, b_{L-1}) \overset{\text{TT-cross}}{\longrightarrow} G^{[0]} \cdots G^{[L-1]} \longrightarrow V_{\text{MPO}} \tag{164}$$

has never required the dense potential vector.

B.8 Why not apply TT-cross directly to the complete Hamiltonian?

One could define a black-box function for arbitrary matrix elements,

$$\begin{aligned} H_{jj'} = & \left[\frac{\hbar^2}{m\Delta x^2} + V_j\right]\delta_{jj'} \\ & - \frac{\hbar^2}{2m\Delta x^2}\left(\delta_{j,j'+1} + \delta_{j,j'-1}\right), \end{aligned} \tag{165}$$

and then apply TT-cross to the resulting $2L$-index tensor

$$H_{b_0 \ldots b_{L-1}, b'_0 \ldots b'_{L-1}}. \tag{166}$$

Although this is mathematically possible, it is generally not the best strategy for the present Hamiltonian. Almost every matrix element is zero. The nonzero entries occupy only the main diagonal and its two neighboring diagonals. A generic cross algorithm sampling the N_x^2 matrix domain therefore encounters a highly sparse pattern.

In contrast, the potential is a smooth function of x, making it highly suitable for TT-cross, while the finite-difference kinetic operator has a known algebraic structure that can be represented exactly. The hybrid construction

$$\boxed{\text{TT-cross potential} + \text{exact kinetic MPO}} \tag{167}$$

is therefore preferable.

B.9 Exact matrix-free MPO for the kinetic energy

The finite-difference kinetic operator is

$$T = \frac{\hbar^2}{m\Delta x^2} I - \frac{\hbar^2}{2m\Delta x^2}\left(S_+ + S_-\right), \tag{168}$$

where S_+ increments a binary integer by one and S_- decrements it, as follows:

$$S_+ = \sum_{j=0}^{N_x-2} |j+1\rangle\langle j|, \tag{169}$$

$$S_- = S_+^\dagger. \tag{170}$$

Defining

$$\sigma_+ = |1\rangle\langle 0| = \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}, \tag{171}$$

and

$$\sigma_- = |0\rangle\langle 1| = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}, \tag{172}$$

we obtain

$$S_+ = \sum_{k=0}^{L-1} \left[\bigotimes_{\ell<k} I_\ell\right] \otimes \sigma_+^{(k)} \otimes \left[\bigotimes_{\ell>k} \sigma_-^{(\ell)}\right]. \tag{173}$$

Similarly,

$$S_- = \sum_{k=0}^{L-1} \left[\bigotimes_{\ell<k} I_\ell\right] \otimes \sigma_-^{(k)} \otimes \left[\bigotimes_{\ell>k} \sigma_+^{(\ell)}\right]. \tag{174}$$

These expressions implement the usual binary carry operation exactly. They also enforce the open finite-difference boundaries: there is no wraparound increment from $2^L - 1$ to zero and no decrement from zero to $2^L - 1$.

For example, incrementing

$$0111 \tag{175}$$

gives

$$0111 \longrightarrow 1000. \tag{176}$$

The leftmost zero is changed by σ_+, while all trailing ones are changed into zeros by σ_-.

B.10 Building the shift MPOs in `quimb`

The local matrices are

```
I2 = np.eye(2)

sp = np.array([
    [0.0, 0.0],
    [1.0, 0.0],
])

sm = np.array([
    [0.0, 1.0],
    [0.0, 0.0],
])
```

Each term in Eqs. (173) and (174) is a product operator and therefore has MPO bond dimension one. The following routine constructs their sums:

```
def add_mpos(terms):
    result = terms[0]
    for term in terms[1:]:
        result = result + term
    return result
```

```
```

```
def build_binary_shift_mpos(L):

    plus_terms = []
    minus_terms = []

    for k in range(L):

        # |j+1><j|
        ops_plus = (
            [I2 for _ in range(k)]
            + [sp]
            + [sm for _ in range(L - k - 1)]
        )
        plus_terms.append(
            MPO_product_operator(ops_plus)
        )

        # |j-1><j|
        ops_minus = (
            [I2 for _ in range(k)]
            + [sm]
            + [sp for _ in range(L - k - 1)]
        )
        minus_terms.append(
            MPO_product_operator(ops_minus)
        )

    Splus = add_mpos(plus_terms)
    Sminus = add_mpos(minus_terms)

    return Splus, Sminus
```

We then construct the identity MPO,

```
I_mpo = MPO_product_operator(
    [I2 for _ in range(L)]
)
```

and the two shift operators,

```
Splus, Sminus = build_binary_shift_mpos(L)
```

The kinetic-energy MPO follows directly from Eq. (168):

```
c0 = hbar**2 / (m * dx**2)

c1 = -hbar**2 / (
    2.0 * m * dx**2
)

T_mpo = (
    c0 * I_mpo
    + c1 * Splus
    + c1 * Sminus
)
```

At no stage has the tridiagonal $2^L \times 2^L$ kinetic matrix been constructed.

B.11 Combining kinetic and potential MPOs

The complete Hamiltonian is now simply

```
H_mpo = T_mpo + V_mpo
```

corresponding to

$$H_{\mathrm{MPO}} = T_{\mathrm{MPO}} + V_{\mathrm{MPO}}. \tag{177}$$

The complete matrix-free workflow is therefore

$$\boxed{\begin{array}{lcll} V(b_0,\ldots,b_{L-1}) & \xrightarrow{\text{TT-cross}} & V_{\mathrm{TT}} & \longrightarrow\ V_{\mathrm{MPO}} \\ \text{binary shifts} & \xrightarrow{\text{exact}} & S_{\pm} & \longrightarrow\ T_{\mathrm{MPO}} \\ & & V_{\mathrm{MPO}} + T_{\mathrm{MPO}} & =\ H_{\mathrm{MPO}}. \end{array}} \tag{178}$$

This H_{MPO} can be passed directly to the DMRG routine developed in the previous appendix.

B.12 Running DMRG without constructing a dense Hamiltonian

For example:

```
p0 = qtn.MPS_rand_state(
    L=L,
    bond_dim=2,
    phys_dim=2,
    normalize=True,
)

dmrg = qtn.DMRG2(
    H_mpo,
    bond_dims=[4, 8, 16, 32],
    cutoffs=1e-10,
    p0=p0,
)

converged = dmrg.solve(
    tol=1e-9,
    max_sweeps=20,
    verbosity=1,
)

print("Converged:", converged)
print("Ground-state energy:", dmrg.energy)
print("Final MPS max bond:",
      dmrg.state.max_bond())
```

The important difference from the original tutorial is what is absent:

```
# No dense Hamiltonian:
# H = np.zeros((Nx, Nx))

# No dense diagonalization:
# E_exact, U_exact = eigh(H)

# No dense-to-MPO conversion:
# H_mpo = qtn.MatrixProductOperator.from_dense(H, ...)
```

Thus the total Hilbert-space dimension, 2^L, may be extremely large without requiring storage proportional to 4^L.

B.13 Complete matrix-free implementation

For convenience, the essential pieces can be combined into a single self-contained implementation:

```
import numpy as np
import torch
import tntorch as tn
import quimb.tensor as qtn

from quimb.tensor.tensor_builder import (
    MPO_product_operator,
)

# ------------------------------------------------------------
# Physical parameters
# ------------------------------------------------------------

hbar = 1.0
m = 1.0

De = 12.0
a = 0.8
xe = 0.0

L = 20
Nx = 2**L

xmin = -2.5
xmax = 10.0

dx = (xmax - xmin) / (Nx - 1)

# ------------------------------------------------------------
# 1. TT-cross representation of the Morse potential
# ------------------------------------------------------------

domain = [
    torch.tensor(
        [0.0, 1.0],
        dtype=torch.float64,
    )
    for _ in range(L)
]

weights = torch.tensor(
    [2**(L - 1 - k) for k in range(L)],
    dtype=torch.float64,
)


def morse_from_bits(B):

    j = B @ weights
    x = xmin + dx * j

```

```
    return De * (
        1.0
        - torch.exp(-a * (x - xe))
    )**2


V_tt, info = tn.cross(
    function=morse_from_bits,
    domain=domain,
    function_arg="matrix",
    eps=1e-10,
    rmax=32,
    max_iter=30,
    val_size=1000,
    verbose=True,
    return_info=True,
)

# ------------------------------------------------------------
# 2. Convert the TT vector to a diagonal quimb MPO
# ------------------------------------------------------------

def tt_to_diagonal_mpo(tt):

    arrays = []
    cores = tt.cores

    for k, core_torch in enumerate(cores):

        G = core_torch.detach().cpu().numpy()
        rleft, d, rright = G.shape

        W = np.zeros(
            (rleft, rright, d, d),
            dtype=G.dtype,
        )

        for s in range(d):
            W[:, :, s, s] = G[:, s, :]

        if k == 0:
            W = W[0, :, :, :]
        elif k == len(cores) - 1:
            W = W[:, 0, :, :]

        arrays.append(W)

    return qtn.MatrixProductOperator(
        arrays,
        shape="lrud",
    )


V_mpo = tt_to_diagonal_mpo(V_tt)

# ------------------------------------------------------------
# 3. Exact binary finite-difference kinetic MPO
# ------------------------------------------------------------
```

```
110
111 I2 = np.eye(2)
112
113 sp = np.array([
114     [0.0, 0.0],
115     [1.0, 0.0],
116 ])
117
118 sm = np.array([
119     [0.0, 1.0],
120     [0.0, 0.0],
121 ])
122
123
124 def add_mpos(terms):
125
126     result = terms[0]
127
128     for term in terms[1:]:
129         result = result + term
130
131     return result
132
133
134 def build_binary_shift_mpos(L):
135
136     plus_terms = []
137     minus_terms = []
138
139     for k in range(L):
140
141         ops_plus = (
142             [I2 for _ in range(k)]
143             + [sp]
144             + [sm for _ in range(L-k-1)]
145         )
146
147         plus_terms.append(
148             MPO_product_operator(ops_plus)
149         )
150
151         ops_minus = (
152             [I2 for _ in range(k)]
153             + [sm]
154             + [sp for _ in range(L-k-1)]
155         )
156
157         minus_terms.append(
158             MPO_product_operator(ops_minus)
159         )
160
161     return (
162         add_mpos(plus_terms),
163         add_mpos(minus_terms),
164     )
165
166
167 Splus, Sminus = build_binary_shift_mpos(L)
```

```
168
169 I_mpo = MPO_product_operator(
170     [I2 for _ in range(L)]
171 )
172
173 c0 = hbar**2 / (m * dx**2)
174
175 c1 = -hbar**2 / (
176     2.0 * m * dx**2
177 )
178
179 T_mpo = (
180     c0 * I_mpo
181     + c1 * Splus
182     + c1 * Sminus
183 )
184
185 # ------------------------------------------------------------
186 # 4. Matrix-free Hamiltonian
187 # ------------------------------------------------------------
188
189 H_mpo = T_mpo + V_mpo
190
191 print("Nx =", Nx)
192 print("Potential MPO max bond =",
193       V_mpo.max_bond())
194 print("Kinetic MPO max bond =",
195       T_mpo.max_bond())
196 print("Hamiltonian MPO max bond =",
197       H_mpo.max_bond())
198
199 # ------------------------------------------------------------
200 # 5. DMRG
201 # ------------------------------------------------------------
202
203 p0 = qtn.MPS_rand_state(
204     L=L,
205     bond_dim=2,
206     phys_dim=2,
207     normalize=True,
208 )
209
210 dmrg = qtn.DMRG2(
211     H_mpo,
212     bond_dims=[4, 8, 16, 32],
213     cutoffs=1e-10,
214     p0=p0,
215 )
216
217 converged = dmrg.solve(
218     tol=1e-9,
219     max_sweeps=20,
220     verbosity=1,
221 )
222
223 print()
224 print("Converged =", converged)
225 print("Energy =", dmrg.energy)
```

```
print(
    "Final MPS max bond =",
    dmrg.state.max_bond(),
)
```

B.14 What has been gained?

The original dense workflow requires storage of an

$$N_x \times N_x = 2^L \times 2^L \tag{179}$$

matrix and therefore scales as

$$O(4^L). \tag{180}$$

The matrix-free construction stores instead

1. the TT cores of the potential, requiring approximately $O(Lr_V^2)$ parameters for binary local dimension two;
2. the finite-bond-dimension kinetic MPO;
3. the MPS used by DMRG.

Thus the central numerical objects scale approximately linearly with the number of binary sites L, apart from their dependence on TT/MPO/MPS bond ranks, provided those ranks remain moderate.

The transformation is conceptually important:

$$\boxed{\text{large coordinate grid} \quad \not\Rightarrow \quad \text{large dense matrix.}} \tag{181}$$

The coordinate grid can be exponentially large in L, while the Hamiltonian and its low-energy eigenstates can remain compact tensor networks.

B.15 Accuracy controls in the matrix-free calculation

The matrix-free approach introduces several independent convergence parameters.

Table 7: **Numerical convergence parameters in the matrix-free TT-cross plus DMRG calculation.**

Parameter	Example	Meaning
TT-cross `eps`	10^{-10}	Controls the requested validation accuracy of the TT approximation to the potential.
TT-cross `rmax`	32	Maximum rank allowed for the potential TT.
DMRG `cutoffs`	10^{-10}	Controls truncation of singular values when the MPS is split during two-site DMRG.
DMRG bond dimension	$4, 8, 16, 32$	Controls the size of the variational MPS manifold.
DMRG `tol`	10^{-9}	Controls the required change in energy between successive DMRG sweeps.

These parameters should be converged independently. In particular, increasing the MPS bond dimension cannot correct an inaccurate potential MPO. The TT-cross approximation should therefore first be converged with respect to `eps` and `rmax`, after which the DMRG calculation can be converged with respect to MPS bond dimension and sweep tolerance.

B.16 Recommended validation strategy

For the small $L = 8$ problem used earlier in the tutorial, one should retain the dense calculation as a unit test. A useful development strategy is therefore

1. construct H_{dense} and the matrix-free H_{MPO} for $L = 8$;
2. verify their agreement;
3. compare DMRG energies obtained from the two Hamiltonian representations;
4. increase L;
5. remove all dense objects from the production calculation.

Thus the small dense problem is not discarded. Instead, it becomes a controlled validation benchmark for the scalable matrix-free implementation.

B.17 Perspective

TT-cross and DMRG solve two different compression problems. TT-cross addresses the question

> Can the *Hamiltonian or one of its functional components* be constructed in low-rank tensor form without evaluating every grid point?

DMRG addresses the separate question

> Can a low-energy *eigenstate* of that Hamiltonian be represented and optimized as a low-rank MPS?

The complete matrix-free calculation therefore contains two levels of tensor compression:

$$\boxed{\begin{array}{lll} V(x) & \xrightarrow{\text{TT-cross}} & V_{\text{MPO}} \\ V_{\text{MPO}} + T_{\text{MPO}} & \xrightarrow{\text{DMRG}} & |\Psi\rangle_{\text{MPS}}\,. \end{array}} \tag{182}$$

TT-cross compresses the *operator representation*, whereas DMRG compresses and optimizes the *eigenstate*. This combination provides a natural route from the small pedagogical Morse oscillator studied in this tutorial to grid sizes for which construction of the full Hamiltonian is impossible.

B.18 Further reading on TT and TT-cross

The tensor-train representation is developed systematically by Oseledets [13], while TT-cross is introduced by Oseledets and Tyrtyshnikov [10]. Binary quantization and QTT representations are discussed in Refs. [4, 5], and adaptive TT interpolation closely related to the cross implementation used here is described by Savostyanov and Oseledets [11]. For the software used in the accompanying implementation, see the published `quimb` and `tntorch` descriptions [9, 14].

C Complete Python implementation

The complete sequence of nonempty code cells from `DMRG_Morse_TTcross.ipynb` is supplied as `DMRG_Morse_TTcross_complete.py` in the accompanying bundle. Colab shell-installation lines are retained as comments so the extracted file is valid Python:

Listing 5: Complete code extracted from DMRG_Morse_TTcross.ipynb.

```
"""Code extracted from DMRG_Morse_TTcross.ipynb.
Colab shell installation commands are retained as comments.
Notebook cells are separated by banners.
"""


# ============================================================================
# Notebook code cell 2
# ============================================================================
# Colab / fresh Python environment
# Colab shell: !pip -q install quimb autoray cotengra

# ============================================================================
# Notebook code cell 3
# ============================================================================

import numpy as np
import matplotlib.pyplot as plt
from scipy.linalg import eigh
from scipy.sparse.linalg import ArpackNoConvergence
import quimb.tensor as qtn

np.set_printoptions(precision=10, suppress=True)

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

# Physical parameters
hbar = 1.0
m = 1.0
De = 12.0
a = 0.8
xe = 0.0

# Binary grid: Nx = 2**L
L = 8
Nx = 2**L

# Coordinate box
xmin = -2.5
xmax = 10.0
x = np.linspace(xmin, xmax, Nx)
dx = x[1] - x[0]

# Morse potential
V = De * (1.0 - np.exp(-a * (x - xe)))**2

print(f"L = {L} binary sites")
print(f"Nx = {Nx} grid points")
print(f"dx = {dx:.6f}")
print(f"V(+infinity) -> De = {De}")

# ============================================================================
# Notebook code cell 6
# ============================================================================

```

```
plt.figure(figsize=(7, 4.5))
plt.plot(x, V, lw=2)
plt.axhline(De, ls="--", label=r"$D_e$")
plt.xlim(xmin, xmax)
plt.ylim(0, 1.15 * De)
plt.xlabel(r"$x$")
plt.ylabel(r"$V(x)$")
plt.title("Morse potential")
plt.legend()
plt.tight_layout()
plt.show()

# ============================================================================
# Notebook code cell 8
# ============================================================================
def build_morse_hamiltonian(x, De, a, m=1.0, hbar=1.0, xe=0.0):
    dx = x[1] - x[0]
    N = len(x)

    V = De * (1.0 - np.exp(-a * (x - xe)))**2

    main = np.full(N, hbar**2 / (m * dx**2)) + V
    off = np.full(N - 1, -hbar**2 / (2.0 * m * dx**2))

    H = np.diag(main)
    H += np.diag(off, k=1)
    H += np.diag(off, k=-1)

    return H, V

H, V = build_morse_hamiltonian(x, De, a, m=m, hbar=hbar, xe=xe)

print("Hermiticity error =", np.linalg.norm(H - H.T))
print("Hamiltonian shape =", H.shape)

# ============================================================================
# Notebook code cell 10
# ============================================================================

E_exact, U_exact = eigh(H)

# Analytic Morse bound-state energies
lam = np.sqrt(2.0 * m * De) / (hbar * a)
omega_e = a * np.sqrt(2.0 * De / m)

n_values = []
E_analytic = []

n = 0
while n + 0.5 < lam:
    En = (
        hbar * omega_e * (n + 0.5)
        - (hbar**2 * a**2 / (2.0 * m)) * (n + 0.5)**2
    )
    n_values.append(n)
    E_analytic.append(En)
    n += 1

```

```
E_analytic = np.array(E_analytic)

print(f"lambda = {lam:.6f}")
print(f"Number of analytic bound states = {len(E_analytic)}")
print()
print(" n analytic grid-exact difference")
for n, Ea in zip(n_values, E_analytic):
    Eg = E_exact[n]
    print(f"{n:2d} {Ea:14.9f} {Eg:14.9f} {Eg-Ea:+.3e}")

# ============================================================================
# Notebook code cell 13
# ============================================================================
H_mpo = qtn.MatrixProductOperator.from_dense(
    H,
    dims=[2] * L,
    cutoff=1e-12,
)

print(H_mpo)
print("MPO maximum bond dimension =", H_mpo.max_bond())

# ============================================================================
# Notebook code cell 15
# ============================================================================
def mps_to_vector(psi):
    # Convert a quimb MPS to a normalized 1D NumPy vector.
    v = np.asarray(psi.to_dense()).reshape(-1)
    v = v / np.linalg.norm(v)
    return v

def run_dmrg(mpo, bond_dims=(4, 8, 16), cutoff=1e-10,
             tol=1e-10, max_sweeps=12, verbosity=1):
    # Two-site DMRG ground-state calculation.
    p0 = qtn.MPS_rand_state(
        L=mpo.L,
        bond_dim=2,
        phys_dim=2,
        normalize=True,
    )

    dmrg = qtn.DMRG2(
        mpo,
        bond_dims=list(bond_dims),
        cutoffs=cutoff,
        p0=p0,
    )

    converged = dmrg.solve(
        tol=tol,
        max_sweeps=max_sweeps,
        verbosity=verbosity,
    )

    psi = dmrg.state.copy()
    vec = mps_to_vector(psi)

    return converged, dmrg, psi, vec
```

```
174
175 conv0, dmrg0, psi0_mps, psi0 = run_dmrg(H_mpo)
176
177 E0_dmrg = np.vdot(psi0, H @ psi0).real
178 fid0 = abs(np.vdot(U_exact[:, 0], psi0))**2
179
180 print()
181 print("Converged =", conv0)
182 print("DMRG internal energy =", dmrg0.energy)
183 print("Dense-check energy =", E0_dmrg)
184 print("Exact grid energy =", E_exact[0])
185 print("Energy error =", E0_dmrg - E_exact[0])
186 print("Fidelity with exact gs =", fid0)
187 print("Final MPS max bond =", psi0_mps.max_bond())
188
189 # ============================================================================
190 # Notebook code cell 17
191 # ============================================================================
192
193 def dmrg_low_states_by_deflation(
194     H,
195     L,
196     n_states=4,
197     penalty=None,
198     bond_dims=(4, 8, 16),
199     mpo_cutoff=1e-12,
200     dmrg_cutoff=1e-10,
201     tol=1e-10,
202     max_sweeps=12,
203     verbosity=0,
204 ):
205     if penalty is None:
206         penalty = 4.0 * De
207
208     states = []
209     energies = []
210     dmrg_objects = []
211
212     for k in range(n_states):
213         Hdef = H.copy()
214
215         for psi_old in states:
216             Hdef += penalty * np.outer(psi_old, psi_old.conj())
217
218         Hdef_mpo = qtn.MatrixProductOperator.from_dense(
219             Hdef,
220             dims=[2] * L,
221             cutoff=mpo_cutoff,
222         )
223
224         converged, dmrg, psi_mps, psi = run_dmrg(
225             Hdef_mpo,
226             bond_dims=bond_dims,
227             cutoff=dmrg_cutoff,
228             tol=tol,
229             max_sweeps=max_sweeps,
230             verbosity=verbosity,
231         )
```

```
232
233         # Numerical cleanup: explicitly orthogonalize against previous states.
234         for psi_old in states:
235             psi -= np.vdot(psi_old, psi) * psi_old
236
237         psi /= np.linalg.norm(psi)
238
239         E = np.vdot(psi, H @ psi).real
240
241         states.append(psi)
242         energies.append(E)
243         dmrg_objects.append(dmrg)
244
245         print(
246             f"state {k}: "
247             f"E = {E:.10f}, "
248             f"exact = {E_exact[k]:.10f}, "
249             f"error = {E-E_exact[k]:+.3e}, "
250             f"converged = {converged}"
251         )
252
253     return np.array(energies), states, dmrg_objects
254
255 n_target = 5
256
257 E_dmrg, psi_dmrg, dmrg_runs = dmrg_low_states_by_deflation(
258     H,
259     L,
260     n_states=n_target,
261     penalty=4.0 * De,
262     bond_dims=(4, 8, 16),
263     verbosity=0,
264 )
265
266 # ============================================================================
267 # Notebook code cell 19
268 # ============================================================================
269
270 print(" n E_exact E_DMRG error fidelity")
271 for n in range(n_target):
272     ov = np.vdot(U_exact[:, n], psi_dmrg[n])
273     fid = abs(ov)**2
274     err = E_dmrg[n] - E_exact[n]
275     print(
276         f"{n:2d} {E_exact[n]:14.9f} {E_dmrg[n]:14.9f} "
277         f"{err:+.3e} {fid:.10f}"
278     )
279
280 # ============================================================================
281 # Notebook code cell 21
282 # ============================================================================
283
284 plt.figure(figsize=(8, 6))
285
286 plt.plot(x, V, "k-", lw=1.5, alpha=0.5, label="Morse potential")
287 scale = 0.55
288
289 for n in range(n_target):
```

```
    psi_e = U_exact[:, n].copy()
    psi_d = psi_dmrg[n].copy()

    if np.vdot(psi_e, psi_d).real < 0:
        psi_d *= -1.0

    psi_e_cont = psi_e / np.sqrt(dx)
    psi_d_cont = psi_d / np.sqrt(dx)

    plt.plot(
        x,
        E_exact[n] + scale * psi_e_cont,
        lw=2,
        label="exact" if n == 0 else None,
    )
    plt.plot(
        x,
        E_dmrg[n] + scale * psi_d_cont,
        "--",
        lw=1.5,
        label="DMRG" if n == 0 else None,
    )
    plt.axhline(E_exact[n], lw=0.5, alpha=0.25)

plt.axhline(De, color="k", ls=":", label=r"dissociation $D_e$")
plt.xlim(-1.5, 7.0)
plt.ylim(0, De + 1.0)
plt.xlabel(r"$x$")
plt.ylabel("energy / vertically shifted wavefunction")
plt.title("Low Morse eigenstates: exact versus DMRG")
plt.legend()
plt.tight_layout()
plt.show()

# ============================================================================
# Notebook code cell 23
# ============================================================================

chis = [2, 4, 8, 16]
conv_data = []

for chi in chis:
    converged, dmrg, psi_mps, psi = run_dmrg(
        H_mpo,
        bond_dims=(chi,),
        cutoff=1e-12,
        tol=1e-10,
        max_sweeps=16,
        verbosity=0,
    )

    E = np.vdot(psi, H @ psi).real
    fid = abs(np.vdot(U_exact[:, 0], psi))**2

    conv_data.append((chi, E, E - E_exact[0], fid, psi_mps.max_bond()))

print(" chi energy error fidelity final bond")
for chi, E, err, fid, mb in conv_data:
```

```
    print(f"{chi:4d} {E:14.10f} {err:+.3e} {fid:.10f} {mb:5d}")

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

chis_arr = np.array([row[0] for row in conv_data])
errors = np.abs(np.array([row[2] for row in conv_data]))

plt.figure(figsize=(6, 4))
plt.semilogy(chis_arr, errors, "o-")
plt.xlabel(r"maximum MPS bond dimension $\chi$")
plt.ylabel(r"$|E_{\rm DMRG}-E_{\rm exact}|$")
plt.title("DMRG bond-dimension convergence")
plt.tight_layout()
plt.show()

# ============================================================================
# Notebook code cell 28
# ============================================================================
# TT-cross dependency for this section.
# quimb is already installed above, but including it here makes the section
# easier to copy into a fresh Colab notebook.
# Colab shell: !pip -q install tntorch

# ============================================================================
# Notebook code cell 29
# ============================================================================
import torch
import tntorch as tn

from quimb.tensor.tensor_builder import MPO_product_operator

torch.set_default_dtype(torch.float64)

# ============================================================================
# Notebook code cell 31
# ============================================================================
def build_morse_potential_tt(
    L,
    xmin,
    xmax,
    De,
    a,
    xe=0.0,
    eps=1e-10,
    rmax=32,
    max_iter=30,
    val_size=1000,
    verbose=True,
):
    """
    Construct V(x) as a QTT/TT using only black-box function evaluations.

    No vector of length 2**L is formed.
    """
    Nx = 2**L
    dx = (xmax - xmin) / (Nx - 1)
```

```
406
407     # Binary domain b_l in {0, 1}.
408     domain = [
409         torch.tensor([0.0, 1.0], dtype=torch.float64)
410         for _ in range(L)
411     ]
412
413     # MSB-first convention, matching NumPy reshape / quimb dims=[2]*L.
414     weights = torch.tensor(
415         [2**(L - 1 - k) for k in range(L)],
416         dtype=torch.float64,
417     )
418
419     def morse_from_bits(B):
420         # B has shape (P, L), where P is chosen adaptively by TT-cross.
421         j = B @ weights
422         xvals = xmin + dx * j
423         return De * (1.0 - torch.exp(-a * (xvals - xe)))**2
424
425     V_tt, info = tn.cross(
426         function=morse_from_bits,
427         domain=domain,
428         function_arg="matrix",
429         eps=eps,
430         rmax=rmax,
431         max_iter=max_iter,
432         val_size=val_size,
433         verbose=verbose,
434         return_info=True,
435     )
436
437     return V_tt, info, dx
438
439
440 def tt_to_diagonal_mpo(tt):
441     """
442     Convert a tntorch TT vector to a diagonal quimb MPO.
443
444     If G[a, s, b] is a TT core, construct
445
446         W[(a,a'), (b,b'), s, t]
447             = G[a,s,b] delta_{s,t}
448
449     for the diagonal operator. Since only one TT is needed for a diagonal
450     function, the bond ranks are unchanged.
451     """
452     arrays = []
453     cores = tt.cores
454     Ltt = len(cores)
455
456     for k, core_torch in enumerate(cores):
457         G = core_torch.detach().cpu().numpy()
458         rleft, d, rright = G.shape
459
460         W = np.zeros((rleft, rright, d, d), dtype=G.dtype)
461
462         for s in range(d):
463             W[:, :, s, s] = G[:, s, :]
```

```
464
465         # Open-boundary quimb MPO end tensors omit the absent outer bond.
466         if k == 0:
467             W = W[0, :, :, :]
468         elif k == Ltt - 1:
469             W = W[:, 0, :, :]
470
471         arrays.append(W)
472
473     return qtn.MatrixProductOperator(arrays, shape="lrud")
474
475 # ============================================================================
476 # Notebook code cell 33
477 # ============================================================================
478 I2_mf = np.eye(2)
479
480 # |1><0| and |0><1|
481 sp_mf = np.array([
482     [0.0, 0.0],
483     [1.0, 0.0],
484 ])
485
486 sm_mf = np.array([
487     [0.0, 1.0],
488     [0.0, 0.0],
489 ])
490
491
492 def add_mpos(terms):
493     out = terms[0]
494     for term in terms[1:]:
495         out = out + term
496     return out
497
498
499 def build_binary_shift_mpos(L):
500     plus_terms = []
501     minus_terms = []
502
503     for k in range(L):
504         # Increment:
505         # unchanged more-significant bits,
506         # 0 -> 1 at the carry-termination site,
507         # all less-significant 1 bits -> 0.
508         ops_plus = (
509             [I2_mf for _ in range(k)]
510             + [sp_mf]
511             + [sm_mf for _ in range(L - k - 1)]
512         )
513         plus_terms.append(MPO_product_operator(ops_plus))
514
515         # Decrement is the Hermitian-conjugate carry rule.
516         ops_minus = (
517             [I2_mf for _ in range(k)]
518             + [sm_mf]
519             + [sp_mf for _ in range(L - k - 1)]
520         )
521         minus_terms.append(MPO_product_operator(ops_minus))
```

```
522
523     return add_mpos(plus_terms), add_mpos(minus_terms)
524
525
526 def build_morse_hamiltonian_mpo(
527     L,
528     xmin,
529     xmax,
530     De,
531     a,
532     m=1.0,
533     hbar=1.0,
534     xe=0.0,
535     cross_eps=1e-10,
536     cross_rmax=32,
537     cross_max_iter=30,
538     cross_val_size=1000,
539     cross_verbose=True,
540 ):
541     """
542     Build H = T + V directly as an MPO.
543
544     Dense N x N matrices and dense N-component potential vectors
545     are never constructed.
546     """
547     V_tt, cross_info, dx = build_morse_potential_tt(
548         L=L,
549         xmin=xmin,
550         xmax=xmax,
551         De=De,
552         a=a,
553         xe=xe,
554         eps=cross_eps,
555         rmax=cross_rmax,
556         max_iter=cross_max_iter,
557         val_size=cross_val_size,
558         verbose=cross_verbose,
559     )
560
561     V_mpo = tt_to_diagonal_mpo(V_tt)
562
563     Splus, Sminus = build_binary_shift_mpos(L)
564     I_mpo = MPO_product_operator([I2_mf for _ in range(L)])
565
566     c0 = hbar**2 / (m * dx**2)
567     c1 = -hbar**2 / (2.0 * m * dx**2)
568
569     T_mpo = c0 * I_mpo + c1 * Splus + c1 * Sminus
570     H_mpo = T_mpo + V_mpo
571
572     return H_mpo, T_mpo, V_mpo, V_tt, cross_info, dx
573
574 # ============================================================================
575 # Notebook code cell 35
576 # ============================================================================
577 # Use the same small problem as the dense benchmark.
578 L_mf = L
579
```

```
H_mpo_mf, T_mpo_mf, V_mpo_mf, V_tt_mf, cross_info_mf, dx_mf = (
    build_morse_hamiltonian_mpo(
        L=L_mf,
        xmin=xmin,
        xmax=xmax,
        De=De,
        a=a,
        m=m,
        hbar=hbar,
        xe=xe,
        cross_eps=1e-10,
        cross_rmax=32,
        cross_verbose=False,
    )
)

print(f"Matrix-free grid size = {2**L_mf}")
print(f"dx = {dx_mf:.8f}")
print(f"TT ranks of V = {V_tt_mf.ranks_tt}")
print(f"Potential MPO max bond = {V_mpo_mf.max_bond()}")
print(f"Kinetic MPO max bond = {T_mpo_mf.max_bond()}")
print(f"Full Hamiltonian MPO bond = {H_mpo_mf.max_bond()}")

for key in ("val_eps", "nsamples", "evals", "iterations"):
    if key in cross_info_mf:
        print(f"TT-cross {key:12s} = {cross_info_mf[key]}")

p0_mf = qtn.MPS_rand_state(
    L=L_mf,
    bond_dim=2,
    phys_dim=2,
    normalize=True,
)

dmrg_mf = qtn.DMRG2(
    H_mpo_mf,
    bond_dims=[4, 8, 16],
    cutoffs=1e-10,
    p0=p0_mf,
)

conv_mf = dmrg_mf.solve(
    tol=1e-10,
    max_sweeps=12,
    verbosity=1,
)

psi0_mf = dmrg_mf.state.copy()
E0_mf = float(np.real(dmrg_mf.energy))

# Validation only: compare to the dense benchmark already computed above.
psi0_mf_dense = mps_to_vector(psi0_mf)
fid0_mf = abs(np.vdot(U_exact[:, 0], psi0_mf_dense))**2

print()
print("Matrix-free DMRG converged =", conv_mf)
print("Matrix-free DMRG energy =", E0_mf)
print("Dense-grid exact energy =", E_exact[0])
```

```
print("Energy error =", E0_mf - E_exact[0])
print("Ground-state fidelity =", fid0_mf)
print("Final MPS max bond =", psi0_mf.max_bond())

# ============================================================================
# Notebook code cell 37
# ============================================================================
def mps_projector_mpo(psi):
    """
    Build the exact MPO |psi><psi| directly from an open-boundary MPS.
    No dense state vector or dense outer product is formed.
    """
    p = psi.copy()
    p.normalize()
    p.permute_arrays("lrp")

    arrays = []
    Lp = p.L

    for k, Araw in enumerate(p.arrays):
        A = np.asarray(Araw)

        # Put every tensor into explicit (left, right, physical) form.
        if k == 0:
            # (right, physical) -> (1, right, physical)
            A = A[None, :, :]
        elif k == Lp - 1:
            # (left, physical) -> (left, 1, physical)
            A = A[:, None, :]

        ldim, rdim, d = A.shape

        # P[(a,a'),(b,b'),s,t] = A[a,b,s] conj(A[a',b',t])
        W = np.einsum(
            "abs,ABt->aAbBst",
            A,
            A.conj(),
            optimize=True,
        )
        W = W.reshape(
            ldim * ldim,
            rdim * rdim,
            d,
            d,
        )

        if k == 0:
            W = W[0, :, :, :]
        elif k == Lp - 1:
            W = W[:, 0, :, :]

        arrays.append(W)

    return qtn.MatrixProductOperator(arrays, shape="lrud")


def mps_mpo_expectation(psi, mpo):
    """
```

```
    Evaluate <psi|MPO|psi> as a tensor-network contraction.
    """
    ket = psi.copy()
    bra = ket.H
    op = mpo.copy()

    # Align physical indices exactly as in the quimb MPO/MPS examples.
    ket.align_(op, bra)

    value = (bra & op & ket) ^ ...
    return float(np.real_if_close(value))


def matrix_free_low_states(
    H_mpo,
    n_states=5,
    penalty=48.0,
    bond_dims=(4, 8, 16),
    cutoff=1e-10,
    tol=1e-10,
    max_sweeps=12,
    verbosity=0,
):
    """
    Low-lying states by DMRG plus MPO-projector deflation.

    Every Hamiltonian and projector remains an MPO.
    """
    states = []
    energies = []
    runs = []

    H_def = H_mpo.copy()

    for k in range(n_states):
        p0 = qtn.MPS_rand_state(
            L=H_mpo.L,
            bond_dim=2,
            phys_dim=2,
            normalize=True,
        )

        dmrg = qtn.DMRG2(
            H_def,
            bond_dims=list(bond_dims),
            cutoffs=cutoff,
            p0=p0,
        )

        converged = dmrg.solve(
            tol=tol,
            max_sweeps=max_sweeps,
            verbosity=verbosity,
        )

        psi = dmrg.state.copy()
        psi.normalize()

```

```
        # Physical energy with the ORIGINAL (undeflated) MPO.
        E = mps_mpo_expectation(psi, H_mpo)

        states.append(psi)
        energies.append(E)
        runs.append(dmrg)

        print(
            f"state {k}: E = {E:.10f}, "
            f"converged = {converged}, "
            f"MPS bond = {psi.max_bond()}"
        )

        # Shift this state upward before targeting the next state.
        P = mps_projector_mpo(psi)
        H_def = H_def + penalty * P

    return np.array(energies), states, runs


n_states_mf = 5

E_mf_states, psi_mf_states, dmrg_mf_runs = matrix_free_low_states(
    H_mpo_mf,
    n_states=n_states_mf,
    penalty=4.0 * De,
    bond_dims=(4, 8, 16),
    cutoff=1e-10,
    tol=1e-10,
    max_sweeps=12,
    verbosity=0,
)

print()
print(" n E_exact E_matrix-free error fidelity")
for n in range(n_states_mf):
    vec = mps_to_vector(psi_mf_states[n]) # validation only at L=8
    fid = abs(np.vdot(U_exact[:, n], vec))**2
    err = E_mf_states[n] - E_exact[n]

    print(
        f"{n:2d} {E_exact[n]:14.9f} "
        f"{E_mf_states[n]:14.9f} "
        f"{err:+.3e} {fid:.10f}"
    )

# ============================================================================
# Notebook code cell 38
# ============================================================================
# Compare the dense-MPO and TT-cross matrix-free eigenvalues.
nst = n_states_mf
idx = np.arange(nst)

plt.figure(figsize=(7, 4.5))
plt.plot(idx, E_exact[:nst], "o", ms=7, label="grid exact")
plt.plot(idx, E_dmrg[:nst], "x", ms=8, label="dense-to-MPO DMRG")
plt.plot(idx, E_mf_states[:nst], "+", ms=10, mew=2,
         label="TT-cross matrix-free DMRG")
```

```
plt.xlabel("state index n")
plt.ylabel("energy")
plt.title("Morse eigenvalues: dense and matrix-free tensor-network routes")
plt.legend()
plt.tight_layout()
plt.show()

plt.figure(figsize=(7, 4.5))
plt.semilogy(
    idx,
    np.abs(E_mf_states[:nst] - E_exact[:nst]),
    "o-",
    label="TT-cross matrix-free",
)
plt.semilogy(
    idx,
    np.abs(E_dmrg[:nst] - E_exact[:nst]),
    "s--",
    label="dense-to-MPO",
)
plt.xlabel("state index n")
plt.ylabel(r"$|E-E_{\rm grid}|$")
plt.title("DMRG eigenvalue errors")
plt.legend()
plt.tight_layout()
plt.show()

# ============================================================================
# Notebook code cell 40
# ============================================================================
RUN_LARGE = False

if RUN_LARGE:
    L_large = 20

    H_large, T_large, V_large, Vtt_large, info_large, dx_large = (
        build_morse_hamiltonian_mpo(
            L=L_large,
            xmin=xmin,
            xmax=xmax,
            De=De,
            a=a,
            m=m,
            hbar=hbar,
            xe=xe,
            cross_eps=1e-10,
            cross_rmax=32,
            cross_verbose=True,
        )
    )

    print(f"Nx = {2**L_large:,}")
    print("TT ranks V =", Vtt_large.ranks_tt)
    print("V MPO max bond =", V_large.max_bond())
    print("T MPO max bond =", T_large.max_bond())
    print("H MPO max bond =", H_large.max_bond())

    p0_large = qtn.MPS_rand_state(
```

```
        L=L_large,
        bond_dim=2,
        phys_dim=2,
        normalize=True,
    )

    dmrg_large = qtn.DMRG2(
        H_large,
        bond_dims=[4, 8, 16, 32],
        cutoffs=1e-10,
        p0=p0_large,
    )

    conv_large = dmrg_large.solve(
        tol=1e-9,
        max_sweeps=20,
        verbosity=1,
    )

    print("Converged =", conv_large)
    print("Ground-state energy =", dmrg_large.energy)
    print("Final MPS max bond =", dmrg_large.state.max_bond())
else:
    print("Large-grid run skipped. Set RUN_LARGE = True to execute it.")

# ============================================================================
# Notebook code cell 43
# ============================================================================
# Parameters for the high-accuracy direct two-coordinate calculation
L_pair = 9
Nx_pair = 2**L_pair

# The bound states considered below are already negligible at these boundaries.
xmin_pair = -3.0
xmax_pair = 12.0

# Slightly different Morse wells. The smaller a values keep the first
# 20 bound-bound product levels below the first dissociation channel.
De1, a1, xe1 = 12.0, 0.50, 0.0
De2, a2, xe2 = 11.5, 0.52, 0.0

m1_pair = 1.0
m2_pair = 1.0
hbar_pair = 1.0

n_pair_compare = 20

print(f'Binary sites per coordinate = {L_pair}')
print(f'Grid points per coordinate = {Nx_pair}')
print(f'Combined binary sites = {2 * L_pair}')
print(f'Two-coordinate dimension = {Nx_pair**2:,}')

# ============================================================================
# Notebook code cell 45
# ============================================================================
def analytic_one_morse_energy(n, De, a, m=1.0, hbar=1.0):
    q = n + 0.5
    return (
```

```
        hbar * a * np.sqrt(2.0 * De / m) * q
        - (hbar**2 * a**2 / (2.0 * m)) * q**2
    )


def analytic_two_morse_energy(
    n1,
    n2,
    De1,
    a1,
    De2,
    a2,
    m1=1.0,
    m2=1.0,
    hbar=1.0,
):
    return (
        analytic_one_morse_energy(n1, De1, a1, m=m1, hbar=hbar)
        + analytic_one_morse_energy(n2, De2, a2, m=m2, hbar=hbar)
    )


lambda1 = np.sqrt(2.0 * m1_pair * De1) / (hbar_pair * a1)
lambda2 = np.sqrt(2.0 * m2_pair * De2) / (hbar_pair * a2)

n1_bound = [n for n in range(100) if n + 0.5 < lambda1]
n2_bound = [n for n in range(100) if n + 0.5 < lambda2]

E1_bound_analytic = np.array([
    analytic_one_morse_energy(n, De1, a1, m=m1_pair, hbar=hbar_pair)
    for n in n1_bound
])
E2_bound_analytic = np.array([
    analytic_one_morse_energy(n, De2, a2, m=m2_pair, hbar=hbar_pair)
    for n in n2_bound
])

analytic_levels = []
for n1 in n1_bound:
    for n2 in n2_bound:
        E = analytic_two_morse_energy(
            n1,
            n2,
            De1,
            a1,
            De2,
            a2,
            m1=m1_pair,
            m2=m2_pair,
            hbar=hbar_pair,
        )
        analytic_levels.append((E, n1, n2))

analytic_levels.sort(key=lambda item: (item[0], item[1], item[2]))
analytic_levels = analytic_levels[:n_pair_compare]

E2D_analytic = np.array([item[0] for item in analytic_levels])
qn2D_analytic = [(item[1], item[2]) for item in analytic_levels]
```

```

continuum_threshold = min(
    De1 + E2_bound_analytic[0],
    De2 + E1_bound_analytic[0],
)
continuum_gap_after_20 = continuum_threshold - E2D_analytic[-1]

print(f'lambda_1 = {lambda1:.8f}; number of bound n1 states = {len(n1_bound)}')
print(f'lambda_2 = {lambda2:.8f}; number of bound n2 states = {len(n2_bound)}')
print(f'Lowest two-coordinate continuum threshold = {continuum_threshold:.9f}')
print(f'20th analytic level = {E2D_analytic[-1]:.9f}')
print(f'Gap between continuum and level 20 = {continuum_gap_after_20:.9f}')

assert continuum_gap_after_20 > 0.0, (
    'The 20th analytic bound-bound level is above the first dissociation '
    'threshold. Change the Morse parameters before doing a rank-by-rank '
    'first-20 comparison.'
)

print()
print('rank (n1,n2) E_analytic')
print('-' * 38)
for k, ((n1, n2), E) in enumerate(zip(qn2D_analytic, E2D_analytic)):
    print(f'{k:3d} ({n1},{n2}) {E:14.9f}')

# ============================================================================
# Notebook code cell 47
# ============================================================================
def build_two_morse_potential_tt(
    L1,
    L2,
    xmin1,
    xmax1,
    xmin2,
    xmax2,
    De1,
    a1,
    De2,
    a2,
    xe1=0.0,
    xe2=0.0,
    eps=1e-12,
    rmax=48,
    max_iter=50,
    val_size=2000,
    verbose=False,
):
    """TT-cross of the combined potential V1(x1) + V2(x2)."""
    Ltot = L1 + L2
    Nx1 = 2**L1
    Nx2 = 2**L2
    dx1 = (xmax1 - xmin1) / (Nx1 - 1)
    dx2 = (xmax2 - xmin2) / (Nx2 - 1)

    domain = [
        torch.tensor([0.0, 1.0], dtype=torch.float64)
        for _ in range(Ltot)
    ]
```

```
    weights1 = torch.tensor(
        [2**(L1 - 1 - k) for k in range(L1)],
        dtype=torch.float64,
    )
    weights2 = torch.tensor(
        [2**(L2 - 1 - k) for k in range(L2)],
        dtype=torch.float64,
    )

    def total_potential_from_bits(B):
        B1 = B[:, :L1]
        B2 = B[:, L1:]

        j1 = B1 @ weights1
        j2 = B2 @ weights2

        x1vals = xmin1 + dx1 * j1
        x2vals = xmin2 + dx2 * j2

        V1vals = De1 * (1.0 - torch.exp(-a1 * (x1vals - xe1)))**2
        V2vals = De2 * (1.0 - torch.exp(-a2 * (x2vals - xe2)))**2

        return V1vals + V2vals

    Vtot_tt, info = tn.cross(
        function=total_potential_from_bits,
        domain=domain,
        function_arg='matrix',
        eps=eps,
        rmax=rmax,
        max_iter=max_iter,
        val_size=val_size,
        verbose=verbose,
        return_info=True,
    )

    return Vtot_tt, info, dx1, dx2


def build_block_shift_mpos(L_total, start, L_block):
    """Shift one MSB-first binary coordinate block by +/-1."""
    plus_terms = []
    minus_terms = []

    n_before = start
    n_after = L_total - start - L_block

    for k in range(L_block):
        ops_plus = (
            [I2_mf for _ in range(n_before)]
            + [I2_mf for _ in range(k)]
            + [sp_mf]
            + [sm_mf for _ in range(L_block - k - 1)]
            + [I2_mf for _ in range(n_after)]
        )
        ops_minus = (
            [I2_mf for _ in range(n_before)]
```

```
            + [I2_mf for _ in range(k)]
            + [sm_mf]
            + [sp_mf for _ in range(L_block - k - 1)]
            + [I2_mf for _ in range(n_after)]
        )

        plus_terms.append(MPO_product_operator(ops_plus))
        minus_terms.append(MPO_product_operator(ops_minus))

    return add_mpos(plus_terms), add_mpos(minus_terms)


def build_block_shift_by_two_mpos(L_total, start, L_block):
    """
    Shift one MSB-first binary coordinate block by +/-2.

    Because the final bit of a coordinate block is the least-significant bit,
    j -> j +/- 2 leaves that bit unchanged and increments/decrements the
    first L_block-1 bits. The omitted LSB is automatically filled by I.
    """
    if L_block < 2:
        raise ValueError('A +/-2 binary shift requires at least two bits.')

    return build_block_shift_mpos(
        L_total=L_total,
        start=start,
        L_block=L_block - 1,
    )


def build_two_morse_hamiltonian_mpo(
    L1,
    L2,
    xmin1,
    xmax1,
    xmin2,
    xmax2,
    De1,
    a1,
    De2,
    a2,
    m1=1.0,
    m2=1.0,
    hbar=1.0,
    xe1=0.0,
    xe2=0.0,
    stencil_order=4,
    cross_eps=1e-12,
    cross_rmax=48,
    cross_max_iter=50,
    cross_val_size=2000,
    cross_verbose=False,
):
    """
    Build the full two-coordinate finite-difference Hamiltonian directly
    as an MPO. No one-dimensional eigensystem is constructed or used.
    """
    Ltot = L1 + L2
```

```
    Vtot_tt, cross_info, dx1, dx2 = build_two_morse_potential_tt(
        L1=L1,
        L2=L2,
        xmin1=xmin1,
        xmax1=xmax1,
        xmin2=xmin2,
        xmax2=xmax2,
        De1=De1,
        a1=a1,
        De2=De2,
        a2=a2,
        xe1=xe1,
        xe2=xe2,
        eps=cross_eps,
        rmax=cross_rmax,
        max_iter=cross_max_iter,
        val_size=cross_val_size,
        verbose=cross_verbose,
    )

    Vtot_mpo = tt_to_diagonal_mpo(Vtot_tt)

    S1plus, S1minus = build_block_shift_mpos(
        L_total=Ltot,
        start=0,
        L_block=L1,
    )
    S2plus, S2minus = build_block_shift_mpos(
        L_total=Ltot,
        start=L1,
        L_block=L2,
    )

    I_all = MPO_product_operator([I2_mf for _ in range(Ltot)])

    if stencil_order == 2:
        c01 = hbar**2 / (m1 * dx1**2)
        c11 = -hbar**2 / (2.0 * m1 * dx1**2)
        c02 = hbar**2 / (m2 * dx2**2)
        c12 = -hbar**2 / (2.0 * m2 * dx2**2)

        Htot_mpo = (
            (c01 + c02) * I_all
            + c11 * (S1plus + S1minus)
            + c12 * (S2plus + S2minus)
            + Vtot_mpo
        )

    elif stencil_order == 4:
        S1plus2, S1minus2 = build_block_shift_by_two_mpos(
            L_total=Ltot,
            start=0,
            L_block=L1,
        )
        S2plus2, S2minus2 = build_block_shift_by_two_mpos(
            L_total=Ltot,
            start=L1,
```

```
            L_block=L2,
        )

        c01 = 5.0 * hbar**2 / (4.0 * m1 * dx1**2)
        c11 = -2.0 * hbar**2 / (3.0 * m1 * dx1**2)
        c21 = hbar**2 / (24.0 * m1 * dx1**2)

        c02 = 5.0 * hbar**2 / (4.0 * m2 * dx2**2)
        c12 = -2.0 * hbar**2 / (3.0 * m2 * dx2**2)
        c22 = hbar**2 / (24.0 * m2 * dx2**2)

        Htot_mpo = (
            (c01 + c02) * I_all
            + c11 * (S1plus + S1minus)
            + c21 * (S1plus2 + S1minus2)
            + c12 * (S2plus + S2minus)
            + c22 * (S2plus2 + S2minus2)
            + Vtot_mpo
        )

    else:
        raise ValueError('stencil_order must be 2 or 4.')

    return Htot_mpo, Vtot_mpo, Vtot_tt, cross_info, dx1, dx2


(
    H2D_pair_mpo,
    V2D_pair_mpo,
    V2D_pair_tt,
    info2D_pair,
    dx1_2D,
    dx2_2D,
) = build_two_morse_hamiltonian_mpo(
    L1=L_pair,
    L2=L_pair,
    xmin1=xmin_pair,
    xmax1=xmax_pair,
    xmin2=xmin_pair,
    xmax2=xmax_pair,
    De1=De1,
    a1=a1,
    De2=De2,
    a2=a2,
    m1=m1_pair,
    m2=m2_pair,
    hbar=hbar_pair,
    xe1=xe1,
    xe2=xe2,
    stencil_order=4,
    cross_eps=1e-12,
    cross_rmax=48,
    cross_max_iter=50,
    cross_val_size=2000,
    cross_verbose=False,
)

print(f'dx_1 = {dx1_2D:.10f}')
```

```
print(f'dx_2 = {dx2_2D:.10f}')
print(f'Combined binary sites = {H2D_pair_mpo.L}')
print(f'Two-coordinate dimension = {2**H2D_pair_mpo.L:,}')
print(f'TT ranks of V_total = {V2D_pair_tt.ranks_tt}')
print(f'Potential MPO max bond = {V2D_pair_mpo.max_bond()}')
print(f'Full Hamiltonian max bond = {H2D_pair_mpo.max_bond()}')
print('Kinetic stencil order = 4')

for key in ('val_eps', 'nsamples', 'evals', 'iterations'):
    if key in info2D_pair:
        print(f'TT-cross {key:12s} = {info2D_pair[key]}')

# ============================================================================
# Notebook code cell 49
# ============================================================================
def _configure_local_eigensolver(dmrg, local_tol, ncv, maxiter):
    """Configure Quimb's sparse local eigensolver for robust DMRG sweeps."""
    dmrg.opts['local_eig_backend'] = 'SCIPY'
    dmrg.opts['local_eig_tol'] = local_tol
    dmrg.opts['local_eig_ncv'] = ncv
    dmrg.opts['local_eig_maxiter'] = maxiter


def _solve_dmrg_stage_with_retry(
    dmrg,
    solve_tol,
    max_sweeps,
    local_tol,
    ncv,
    maxiter,
    fallback_local_tol=1e-5,
    fallback_ncv=48,
    fallback_maxiter=30000,
    verbosity=0,
    stage_name='DMRG',
):
    """Run one DMRG stage and recover automatically from ARPACK failure."""
    _configure_local_eigensolver(dmrg, local_tol, ncv, maxiter)
    try:
        return dmrg.solve(
            tol=solve_tol,
            max_sweeps=max_sweeps,
            verbosity=verbosity,
        )
    except ArpackNoConvergence as err:
        nconv = 0 if err.eigenvalues is None else len(err.eigenvalues)
        print(
            f' {stage_name}: ARPACK did not converge '
            f'({nconv} local eigenpair(s) returned).'
        )
        print(
            ' Resuming from the current MPS with '
            f'ncv={fallback_ncv}, local_tol={fallback_local_tol:.1e}, '
            f'maxiter={fallback_maxiter}.'
        )
        _configure_local_eigensolver(
            dmrg,
            fallback_local_tol,
```

```
            fallback_ncv,
            fallback_maxiter,
        )
        return dmrg.solve(
            tol=solve_tol,
            max_sweeps=max_sweeps,
            verbosity=verbosity,
        )


def _mps_overlap(psi_left, psi_right):
    """Return <psi_left|psi_right> as a scalar."""
    value = psi_left.H @ psi_right
    try:
        value = value.item()
    except AttributeError:
        pass
    return complex(value)


def mps_mpo_matrix_element(psi_left, mpo, psi_right):
    """Return <psi_left|MPO|psi_right> by tensor-network contraction."""
    bra = psi_left.H
    op = mpo.copy()
    ket = psi_right.copy()
    ket.align_(op, bra)
    value = (bra & op & ket) ^ ...
    try:
        value = value.item()
    except AttributeError:
        pass
    return complex(value)


def rayleigh_ritz_from_mps_subspace(states, H_mpo, overlap_cutoff=1e-10):
    """Rayleigh--Ritz refinement in the span of the direct 2D DMRG states."""
    n = len(states)
    S = np.zeros((n, n), dtype=np.complex128)
    Hs = np.zeros((n, n), dtype=np.complex128)

    for i in range(n):
        for j in range(i, n):
            sij = _mps_overlap(states[i], states[j])
            hij = mps_mpo_matrix_element(states[i], H_mpo, states[j])
            S[i, j] = sij
            S[j, i] = np.conjugate(sij)
            Hs[i, j] = hij
            Hs[j, i] = np.conjugate(hij)

    S = 0.5 * (S + S.conj().T)
    Hs = 0.5 * (Hs + Hs.conj().T)

    svals, U = np.linalg.eigh(S)
    smax = np.max(svals)
    keep = svals > overlap_cutoff * smax

    if np.count_nonzero(keep) < n:
        print(
```

```
            'Rayleigh--Ritz warning: numerical DMRG-subspace rank = '
            f'{np.count_nonzero(keep)}/{n}.'
        )

    X = U[:, keep] / np.sqrt(svals[keep])[None, :]
    Horth = X.conj().T @ Hs @ X
    Horth = 0.5 * (Horth + Horth.conj().T)
    E_ritz, Corth = np.linalg.eigh(Horth)
    C = X @ Corth

    return np.real_if_close(E_ritz).astype(float), S, Hs, C, svals


def direct_two_coordinate_low_states(
    H_mpo,
    n_states=20,
    penalty=24.0,
    bond_dims=(8, 16, 32, 48, 64),
    cutoff=1e-12,
    tol=1e-10,
    warmup_local_eig_tol=1e-4,
    refine_local_eig_tol=1e-6,
    local_eig_ncv=24,
    local_eig_maxiter=12000,
    fallback_local_eig_tol=1e-5,
    fallback_local_eig_ncv=48,
    fallback_local_eig_maxiter=30000,
    warmup_sweeps=6,
    refine_sweeps=22,
    n_restarts=2,
    init_bond_dim=4,
    deflation_max_bond=256,
    deflation_cutoff=1e-13,
    overlap_accept_tol=1e-6,
    verbosity=0,
):
    """Low-lying spectrum from the one combined two-coordinate MPO."""
    states = []
    energies = []
    converged_flags = []
    final_bonds = []
    selected_deflated_energies = []
    accepted_max_overlap2 = []
    H_def = H_mpo.copy()

    for k in range(n_states):
        best = None
        best_key = None

        for restart in range(n_restarts):
            p0 = qtn.MPS_rand_state(
                L=H_mpo.L,
                bond_dim=init_bond_dim,
                phys_dim=2,
                normalize=True,
            )
            dmrg = qtn.DMRG2(
                H_def,
```

```
                bond_dims=list(bond_dims),
                cutoffs=cutoff,
                p0=p0,
            )

            warmup_tol = max(1e-7, 1000.0 * tol)
            _solve_dmrg_stage_with_retry(
                dmrg,
                solve_tol=warmup_tol,
                max_sweeps=warmup_sweeps,
                local_tol=warmup_local_eig_tol,
                ncv=local_eig_ncv,
                maxiter=local_eig_maxiter,
                fallback_local_tol=fallback_local_eig_tol,
                fallback_ncv=fallback_local_eig_ncv,
                fallback_maxiter=fallback_local_eig_maxiter,
                verbosity=verbosity,
                stage_name=f'state {k}, restart {restart + 1}, warm-up',
            )

            converged = _solve_dmrg_stage_with_retry(
                dmrg,
                solve_tol=tol,
                max_sweeps=refine_sweeps,
                local_tol=refine_local_eig_tol,
                ncv=local_eig_ncv,
                maxiter=local_eig_maxiter,
                fallback_local_tol=fallback_local_eig_tol,
                fallback_ncv=fallback_local_eig_ncv,
                fallback_maxiter=fallback_local_eig_maxiter,
                verbosity=verbosity,
                stage_name=f'state {k}, restart {restart + 1}, refinement',
            )

            psi = dmrg.state.copy()
            psi.normalize()
            E_def = float(np.real(dmrg.energy))
            E_phys = mps_mpo_expectation(psi, H_mpo)

            if states:
                max_overlap2 = float(max(
                    abs(_mps_overlap(phi, psi))**2 for phi in states
                ))
            else:
                max_overlap2 = 0.0

            candidate = {
                'E_def': E_def,
                'E_phys': E_phys,
                'psi': psi,
                'converged': bool(converged),
                'bond': psi.max_bond(),
                'restart': restart,
                'max_overlap2': max_overlap2,
            }
            candidate_key = (
                candidate['max_overlap2'] > overlap_accept_tol,
                candidate['E_def'],
```

```
            )
            if (best is None) or (candidate_key < best_key):
                best = candidate
                best_key = candidate_key

        psi = best['psi']
        states.append(psi)
        energies.append(best['E_phys'])
        converged_flags.append(best['converged'])
        final_bonds.append(best['bond'])
        selected_deflated_energies.append(best['E_def'])
        accepted_max_overlap2.append(best['max_overlap2'])

        print(
            f"state {k:2d}: E_raw = {best['E_phys']:14.10f}, "
            f"converged = {best['converged']}, MPS bond = {best['bond']}, "
            f"max previous overlap^2 = {best['max_overlap2']:.2e}, "
            f"restart = {best['restart'] + 1}/{n_restarts}"
        )

        if k < n_states - 1:
            projector = mps_projector_mpo(psi)
            H_def = H_def + penalty * projector
            H_def.compress(
                max_bond=deflation_max_bond,
                cutoff=deflation_cutoff,
            )
            print(
                ' deflated MPO max bond after compression = '
                f'{H_def.max_bond()}'
            )

    energies = np.asarray(energies)
    order = np.argsort(energies)
    energies = energies[order]
    states = [states[i] for i in order]
    converged_flags = np.asarray(converged_flags, dtype=bool)[order]
    final_bonds = np.asarray(final_bonds, dtype=int)[order]
    selected_deflated_energies = np.asarray(selected_deflated_energies)[order]
    accepted_max_overlap2 = np.asarray(accepted_max_overlap2)[order]

    return (
        energies,
        states,
        converged_flags,
        final_bonds,
        selected_deflated_energies,
        accepted_max_overlap2,
    )


(
    E2D_raw,
    psi2D_direct,
    conv2D_direct,
    bonds2D_direct,
    E2D_deflated_selected,
    max_overlap2_2D,
```

```
) = direct_two_coordinate_low_states(
    H2D_pair_mpo,
    n_states=n_pair_compare,
    penalty=2.0 * max(De1, De2),
    bond_dims=(8, 16, 32, 48, 64),
    cutoff=1e-12,
    tol=1e-10,
    warmup_local_eig_tol=1e-4,
    refine_local_eig_tol=1e-6,
    local_eig_ncv=24,
    local_eig_maxiter=12000,
    fallback_local_eig_tol=1e-5,
    fallback_local_eig_ncv=48,
    fallback_local_eig_maxiter=30000,
    warmup_sweeps=6,
    refine_sweeps=22,
    n_restarts=2,
    init_bond_dim=4,
    deflation_max_bond=256,
    deflation_cutoff=1e-13,
    overlap_accept_tol=1e-6,
    verbosity=0,
)

print('\nBuilding the 20 x 20 projected Hamiltonian for Rayleigh--Ritz refinement...')
(
    E2D_direct,
    S2D_subspace,
    H2D_subspace,
    C2D_ritz,
    S2D_eigvals,
) = rayleigh_ritz_from_mps_subspace(
    psi2D_direct,
    H2D_pair_mpo,
    overlap_cutoff=1e-10,
)

print(f'Rayleigh--Ritz subspace dimension = {len(E2D_direct)}')
print(f'Smallest overlap-matrix eigenvalue = {np.min(S2D_eigvals):.3e}')
offdiag_S = S2D_subspace - np.diag(np.diag(S2D_subspace))
print(f'Max off-diagonal |S_ij| = {np.max(np.abs(offdiag_S)):.3e}')

# ============================================================================
# Notebook code cell 51
# ============================================================================
ranks_pair = np.arange(n_pair_compare)
raw_error_2D = E2D_raw - E2D_analytic
energy_error_2D = E2D_direct - E2D_analytic
abs_error_2D = np.abs(energy_error_2D)

print(
    'rank analytic label E_analytic E_raw-DMRG '
    'E_Ritz-direct Ritz-analytic converged MPS bond'
)
print('-' * 128)
for k, (n1, n2) in enumerate(qn2D_analytic):
    print(
        f'{k:3d} ({n1},{n2}) '
```

```
        f'{E2D_analytic[k]:14.10f} '
        f'{E2D_raw[k]:14.10f} '
        f'{E2D_direct[k]:14.10f} '
        f'{energy_error_2D[k]:+.3e} '
        f'{str(conv2D_direct[k]):>5s} '
        f'{bonds2D_direct[k]:3d}'
    )

print()
print(f'Maximum |raw DMRG - analytic| = {np.max(np.abs(raw_error_2D)):.6e}')
print(f'Maximum |Ritz 2D - analytic| = {np.max(abs_error_2D):.6e}')
print(f'RMS |Ritz 2D - analytic| = {np.sqrt(np.mean(energy_error_2D**2)):.6e}')
print(f'Mean |Ritz 2D - analytic| = {np.mean(abs_error_2D):.6e}')
print(f'Max accepted previous overlap^2 = {np.max(max_overlap2_2D):.6e}')
print(f'Continuum gap above level 20 = {continuum_gap_after_20:.6e}')

plt.figure(figsize=(8, 5))
plt.plot(ranks_pair, E2D_analytic, 'o', ms=7, label='continuum analytic')
plt.plot(ranks_pair, E2D_raw, '+', ms=8, mew=1.4, label='raw deflated DMRG')
plt.plot(
    ranks_pair,
    E2D_direct,
    'x',
    ms=8,
    mew=1.7,
    label='Rayleigh--Ritz in direct 2D DMRG subspace',
)
plt.axhline(
    continuum_threshold,
    ls='--',
    lw=1.2,
    label='lowest dissociation channel',
)
plt.xlabel('two-coordinate state rank')
plt.ylabel('energy')
plt.title('First 20 discrete levels: analytic versus direct two-coordinate DMRG')
plt.legend()
plt.tight_layout()
plt.show()

plt.figure(figsize=(8, 4.5))
plt.semilogy(
    ranks_pair,
    np.maximum(np.abs(raw_error_2D), 1e-16),
    'o--',
    label='raw deflation error',
)
plt.semilogy(
    ranks_pair,
    np.maximum(abs_error_2D, 1e-16),
    'o-',
    label='Rayleigh--Ritz-refined error',
)
plt.xlabel('two-coordinate state rank')
plt.ylabel(r'$|E-E_{\rm ana}|$')
plt.title('Direct two-coordinate DMRG error before and after subspace refinement')
plt.legend()
plt.tight_layout()
```

```
plt.show()
```

References

[1] Steven R. White. Density matrix formulation for quantum renormalization groups. *Physical Review Letters*, 69(19):2863–2866, 1992.

[2] Steven R. White. Density-matrix algorithms for quantum renormalization groups. *Physical Review B*, 48(14):10345–10356, 1993.

[3] Ulrich Schollwöck. The density-matrix renormalization group in the age of matrix product states. *Annals of Physics*, 326(1):96–192, 2011.

[4] I. V. Oseledets. Approximation of $2^d \times 2^d$ matrices using tensor decomposition. *SIAM Journal on Matrix Analysis and Applications*, 31(4):2130–2145, 2010.

[5] Boris N. Khoromskij. $O(d \log N)$-quantics approximation of N-d tensors in high-dimensional numerical modeling. *Constructive Approximation*, 34(2):257–280, 2011.

[6] Philip M. Morse. Diatomic molecules according to the wave mechanics. II. vibrational levels. *Physical Review*, 34(1):57–64, 1929.

[7] Bengt Fornberg. Generation of finite difference formulas on arbitrarily spaced grids. *Mathematics of Computation*, 51(184):699–706, 1988.

[8] Román Orús. A practical introduction to tensor networks: Matrix product states and projected entangled pair states. *Annals of Physics*, 349:117–158, 2014.

[9] Johnnie Gray. quimb: A python package for quantum information and many-body calculations. *Journal of Open Source Software*, 3(29):819, 2018.

[10] Ivan Oseledets and Eugene Tyrtyshnikov. TT-cross approximation for multidimensional arrays. *Linear Algebra and its Applications*, 432(1):70–88, 2010.

[11] Dmitry Savostyanov and Ivan Oseledets. Fast adaptive interpolation of multi-dimensional arrays in tensor train format. In *2011 7th International Workshop on Multidimensional (nD) Systems*, pages 1–8. IEEE, 2011.

[12] Yousef Saad. *Numerical Methods for Large Eigenvalue Problems*, volume 66 of *Classics in Applied Mathematics*. Society for Industrial and Applied Mathematics, Philadelphia, revised edition, 2011.

[13] I. V. Oseledets. Tensor-train decomposition. *SIAM Journal on Scientific Computing*, 33(5):2295–2317, 2011.

[14] Mikhail Usvyatsov, Rafael Ballester-Ripoll, and Konrad Schindler. tntorch: Tensor network learning with PyTorch. *Journal of Machine Learning Research*, 23(208):1–6, 2022.