A Pedagogical Tutorial on Collocation Methods for Computing Quantum Eigenstates

Victor S. Batista

Abstract

Collocation methods determine eigenstates by enforcing the Schrödinger equation at selected coordinate-space points rather than by evaluating all Hamiltonian matrix elements through numerical quadrature. This tutorial derives square and rectangular collocation, explains the role of the pseudoinverse and point selection, and develops a complete Gaussian-basis implementation for the one-dimensional Morse oscillator. It then extends the method to a two-dimensional sum of Morse potentials using Christoffel weights and a product-contracted basis. The point-selection stage is treated in two ways: the usual greedy column-pivoted QR construction and a regularized log-determinant beam search that retains several competing partial point sets. The two methods are compared using identical candidate points, weights, basis functions, and rectangular-collocation equations. Numerical results are benchmarked against analytic separable Morse solutions and are then extended to a coupled two-dimensional potential.

1 Why collocation?

Consider the stationary Schrödinger equation

$$\hat{H}\psi_n(x) = E_n\psi_n(x), \qquad \hat{H} = -\frac{\hbar^2}{2m}\frac{d^2}{dx^2} + V(x). \tag{1}$$

In a conventional variational calculation, the wavefunction is expanded as

$$\psi_n(x) = \sum_{j=1}^{N_b} c_{jn} b_j(x), \tag{2}$$

and the equation is projected onto every basis function. This gives

$$\boldsymbol{H}\boldsymbol{c}_n = E_n \boldsymbol{S}\boldsymbol{c}_n, \tag{3}$$

where

$$H_{ij} = \langle b_i|\hat{H}|b_j\rangle, \tag{4}$$
$$S_{ij} = \langle b_i|b_j\rangle. \tag{5}$$

For a general multidimensional potential, the potential matrix requires quadrature:

$$V_{ij} = \int b_i^*(\boldsymbol{x})V(\boldsymbol{x})b_j(\boldsymbol{x})\,d\boldsymbol{x} \approx \sum_{k=1}^{N_p} w_k b_i^*(\boldsymbol{x}_k)V(\boldsymbol{x}_k)b_j(\boldsymbol{x}_k). \tag{6}$$

The number of quadrature points can grow approximately as

$$N_p \sim n_{\mathrm{1D}}^d, \tag{7}$$

which becomes prohibitive as the dimensionality d increases.

Collocation changes the question. Instead of integrating the residual against test functions, it requires the residual to vanish at selected points:

$$\left[\hat{H}\psi_n(x) - E_n\psi_n(x)\right]_{x=x_i} = 0. \tag{8}$$

The potential then appears only through the values $V(x_i)$.

2 From the differential equation to a collocation matrix

Insert the basis expansion into the Schrödinger equation and evaluate it at the collocation points x_i, $i = 1, \ldots, N_p$:

$$\sum_{j=1}^{N_b} \left[-\frac{\hbar^2}{2m} b_j''(x_i) + V(x_i) b_j(x_i)\right] c_{jn} = E_n \sum_{j=1}^{N_b} b_j(x_i) c_{jn}. \tag{9}$$

Define

$$B_{ij} = b_j(x_i), \tag{10}$$

$$T_{ij} = -\frac{\hbar^2}{2m} b_j''(x_i), \tag{11}$$

$$V_{ik} = V(x_i)\delta_{ik}. \tag{12}$$

The collocation equation is

$$(\boldsymbol{T} + \boldsymbol{V}\boldsymbol{B})\boldsymbol{c}_n = E_n \boldsymbol{B}\boldsymbol{c}_n. \tag{13}$$

Both $\boldsymbol{B}$ and $\boldsymbol{T}$ have dimensions $N_p \times N_b$.

2.1 Square collocation

When $N_p = N_b$ and $\boldsymbol{B}$ is invertible, multiplying Eq. (13) by $\boldsymbol{B}^{-1}$ gives

$$\boldsymbol{B}^{-1}(\boldsymbol{T} + \boldsymbol{V}\boldsymbol{B})\boldsymbol{c}_n = E_n \boldsymbol{c}_n. \tag{14}$$

This is an ordinary eigenvalue problem.

2.2 Rectangular collocation

Usually it is advantageous to enforce the equation at more points than there are basis functions:

$$N_p > N_b. \tag{15}$$

The system is then overdetermined. A simple regular rectangular-collocation projection uses the Moore–Penrose pseudoinverse:

$$\boldsymbol{M} = \boldsymbol{B}^{+}, \qquad \boldsymbol{B}^{+} = (\boldsymbol{B}^{\mathsf{T}}\boldsymbol{B})^{-1}\boldsymbol{B}^{\mathsf{T}}, \tag{16}$$

assuming full column rank. Multiplication by $\boldsymbol{B}^{+}$ gives

$$\underbrace{\boldsymbol{B}^{+}(\boldsymbol{T} + \boldsymbol{V}\boldsymbol{B})}_{\boldsymbol{A}_{\mathrm{RC}}} \boldsymbol{c}_n = E_n \boldsymbol{c}_n. \tag{17}$$

The matrix $\boldsymbol{A}_{\mathrm{RC}}$ has dimensions $N_b \times N_b$.

A useful interpretation is that Eq. (17) finds the coefficient vector whose Schrödinger residual is as small as possible in a discrete least-squares sense. The extra collocation points suppress solutions that satisfy the equation only accidentally at a small set of locations.

3 Weighted and chopped variants

If some regions of coordinate space should contribute more strongly than others, introduce a diagonal weight matrix

$$\boldsymbol{W} = \mathrm{diag}(w_1, \ldots, w_{N_p}). \tag{18}$$

Weighted least squares gives

$$\boldsymbol{M}_{\mathrm{W}} = (\boldsymbol{B}^{\mathsf{T}}\boldsymbol{W}\boldsymbol{B})^{-1}\boldsymbol{B}^{\mathsf{T}}\boldsymbol{W}. \tag{19}$$

The corresponding effective Hamiltonian is

$$\boldsymbol{A}_{\mathrm{WRC}} = \boldsymbol{M}_{\mathrm{W}}(\boldsymbol{T} + \boldsymbol{V}\boldsymbol{B}). \tag{20}$$

A chopped method constructs an initially large candidate point set and retains a numerically informative subset. Pivoted QR factorization, Fekete-like points, and Christoffel-weighted sampling are common ways to avoid nearly redundant points. These refinements are especially valuable for high-dimensional product bases, but the regular pseudoinverse method is sufficient for the one-dimensional demonstration below.

4 Gaussian basis functions

We use Gaussian radial basis functions centered at c_j:

$$b_j(x) = \exp[-\beta(x - c_j)^2]. \tag{21}$$

Their second derivatives are analytic:

$$b_j''(x) = \left[4\beta^2(x - c_j)^2 - 2\beta\right] b_j(x). \tag{22}$$

Therefore,

$$T_{ij} = -\frac{\hbar^2}{2m}\left[4\beta^2(x_i - c_j)^2 - 2\beta\right] e^{-\beta(x_i - c_j)^2}. \tag{23}$$

No kinetic-energy quadrature is required, and the potential is evaluated only as $V(x_i)$.

The exponent β controls the overlap between neighboring Gaussians. For uniformly spaced centers with separation Δc, it is convenient to set

$$\beta = \frac{s}{(\Delta c)^2}, \tag{24}$$

where s is a dimensionless shape parameter. Very large s gives narrow, poorly overlapping functions; very small s makes the basis nearly linearly dependent.

5 The Morse oscillator

The Morse potential is

$$V(x) = D_e\left[1 - e^{-a(x - x_e)}\right]^2. \tag{25}$$

Its minimum is zero at $x = x_e$, and its dissociation limit is D_e.

Define

$$\lambda = \frac{\sqrt{2mD_e}}{\hbar a}. \tag{26}$$

The bound-state energies are

$$E_n = D_e - \frac{\hbar^2 a^2}{2m}\left(\lambda - n - \frac{1}{2}\right)^2, \qquad n < \lambda - \frac{1}{2}. \tag{27}$$

Equivalently,

$$E_n = \hbar\omega_e\left(n+\frac{1}{2}\right) - \frac{[\hbar\omega_e(n+\frac{1}{2})]^2}{4D_e}, \qquad \omega_e = a\sqrt{\frac{2D_e}{m}}. \tag{28}$$

Introduce

$$y = 2\lambda e^{-a(x-x_e)}. \tag{29}$$

Apart from normalization, the analytic eigenfunction is

$$\psi_n(x) \propto e^{-y/2} y^{\lambda-n-\frac{1}{2}} L_n^{(2\lambda-2n-1)}(y), \tag{30}$$

where $L_n^{(\alpha)}$ is an associated Laguerre polynomial.

6 Numerical algorithm

The demonstration follows these steps:

1. Choose N_b Gaussian centers c_j.
2. Choose $N_p > N_b$ collocation points x_i.
3. Build $\boldsymbol{B}$, $\boldsymbol{T}$, and $\boldsymbol{H}_{\text{col}} = \boldsymbol{T} + \boldsymbol{V}\boldsymbol{B}$.
4. Compute $\boldsymbol{B}^+$ with a singular-value decomposition.
5. Diagonalize $\boldsymbol{A}_{\text{RC}} = \boldsymbol{B}^+\boldsymbol{H}_{\text{col}}$.
6. Reconstruct $\psi_n(x) = \sum_j c_{jn} b_j(x)$ on a dense plotting grid.
7. Normalize each numerical state and align its arbitrary global sign with the analytic solution.

The effective collocation matrix need not be exactly symmetric, even though the underlying Hamiltonian is Hermitian. Well-converged physical eigenvalues should nevertheless have negligible imaginary parts. Large imaginary components usually indicate a poorly conditioned basis, insufficient coordinate range, or unsuitable point selection.

7 Complete Python test: 1D-Morse Potential

The accompanying file `morse_rectangular_collocation.py` uses dimensionless parameters

$$\hbar = m = a = 1, \qquad D_e = 20, \qquad x_e = 0. \tag{31}$$

It uses $N_b = 70$ Gaussian functions and $N_p = 140$ collocation points.

```
#!/usr/bin/env python3
"""Rectangular collocation for the one-dimensional Morse oscillator."""
from __future__ import annotations
import numpy as np
import matplotlib.pyplot as plt
from scipy.linalg import eig
from scipy.special import eval_genlaguerre
from scipy.integrate import simpson

HBAR = 1.0
MASS = 1.0
D_E = 20.0
A = 1.0
X_E = 0.0
```

```
X_MIN = -2.5
X_MAX = 8.0
N_BASIS = 70
N_POINTS = 140
N_STATES_TO_PLOT = 4
SHAPE = 0.50
PINV_RCOND = 1.0e-12

def morse_potential(x):
    return D_E * (1.0 - np.exp(-A * (x - X_E))) ** 2

def morse_lambda():
    return np.sqrt(2.0 * MASS * D_E) / (HBAR * A)

def exact_energy(n):
    lam = morse_lambda()
    return D_E - (HBAR**2 * A**2 / (2.0 * MASS)) * (lam - n - 0.5) ** 2

def exact_wavefunction(n, x):
    lam = morse_lambda()
    y = 2.0 * lam * np.exp(-A * (x - X_E))
    alpha = 2.0 * lam - 2.0 * n - 1.0
    psi = np.exp(-0.5 * y) * y ** (lam - n - 0.5)
    psi *= eval_genlaguerre(n, alpha, y)
    return psi / np.sqrt(simpson(psi * psi, x=x))

def gaussian_matrices(points, centers, beta):
    displacement = points[:, None] - centers[None, :]
    basis = np.exp(-beta * displacement**2)
    second_derivative = (4.0 * beta**2 * displacement**2 - 2.0 * beta) * basis
    kinetic = -(HBAR**2 / (2.0 * MASS)) * second_derivative
    return basis, kinetic

def solve_rectangular_collocation():
    centers = np.linspace(X_MIN, X_MAX, N_BASIS)
    points = np.linspace(X_MIN, X_MAX, N_POINTS)
    spacing = centers[1] - centers[0]
    beta = SHAPE / spacing**2
    B, T = gaussian_matrices(points, centers, beta)
    H_col = T + morse_potential(points)[:, None] * B
    B_pinv = np.linalg.pinv(B, rcond=PINV_RCOND)
    A_eff = B_pinv @ H_col
    eigenvalues, eigenvectors = eig(A_eff)
    real_mask = np.abs(eigenvalues.imag) < 1.0e-7
    eigenvalues = eigenvalues.real[real_mask]
    eigenvectors = eigenvectors[:, real_mask].real
    order = np.argsort(eigenvalues)
    return eigenvalues[order], eigenvectors[:, order], centers, beta

def reconstruct_state(coefficients, x_dense, centers, beta):
    displacement = x_dense[:, None] - centers[None, :]
    B_dense = np.exp(-beta * displacement**2)
    psi = B_dense @ coefficients
    return psi / np.sqrt(simpson(psi * psi, x=x_dense))

def main():
    eigenvalues, eigenvectors, centers, beta = solve_rectangular_collocation()
    x_dense = np.linspace(X_MIN, X_MAX, 3000)
```

```
    n_plot = min(N_STATES_TO_PLOT, int(np.ceil(morse_lambda() - 0.5)))
    rows, numerical_states, exact_states = [], [], []
    for n in range(n_plot):
        psi_num = reconstruct_state(eigenvectors[:, n], x_dense, centers, beta)
        psi_exact = exact_wavefunction(n, x_dense)
        overlap = simpson(psi_num * psi_exact, x=x_dense)
        if overlap < 0.0:
            psi_num *= -1.0
            overlap *= -1.0
        l2_error = np.sqrt(simpson((psi_num - psi_exact) ** 2, x=x_dense))
        rows.append((n, eigenvalues[n], exact_energy(n), abs(eigenvalues[n]-
            exact_energy(n)), overlap, l2_error))
        numerical_states.append(psi_num)
        exact_states.append(psi_exact)

    lines = [" n E_collocation E_exact |Delta E| overlap L2 error\n"]
    for row in rows:
        lines.append(f"{row[0]:2d} {row[1]:16.10f} {row[2]:16.10f} {row[3]:14.6e} {row
            [4]:12.8f} {row[5]:12.6e}\n")
    table_text = ''.join(lines)
    print(table_text)
    with open('morse_energy_table.txt','w',encoding='utf-8') as f:
        f.write(table_text)

    fig, axes = plt.subplots(n_plot, 1, figsize=(8.0, 2.45*n_plot), sharex=True)
    if n_plot == 1:
        axes=[axes]
    for n, ax in enumerate(axes):
        ax.plot(x_dense, numerical_states[n], label='Rectangular collocation')
        ax.plot(x_dense, exact_states[n], '--', label='Analytic Morse state')
        ax.axhline(0.0, linewidth=0.7)
        ax.set_ylabel(rf'$\psi_{{{n}}}(x)$')
        ax.set_title(rf'$n={n}$: $E_{{RC}}={eigenvalues[n]:.8f}$, $E_{{exact}}={
            exact_energy(n):.8f}$')
        ax.legend(); ax.grid(alpha=0.25)
    axes[-1].set_xlabel(r'$x$')
    fig.suptitle('Morse-oscillator eigenstate amplitudes:\nrectangular collocation
        versus analytic solutions')
    fig.tight_layout()
    fig.savefig('morse_eigenstate_comparison.png', dpi=220)
    plt.close(fig)

if __name__ == '__main__':
    main()
```

8 Expected comparison

Run

`python morse_rectangular_collocation.py`

from the directory containing the script. It creates

- `morse_eigenstate_comparison.png`, and
- `morse_energy_table.txt`.

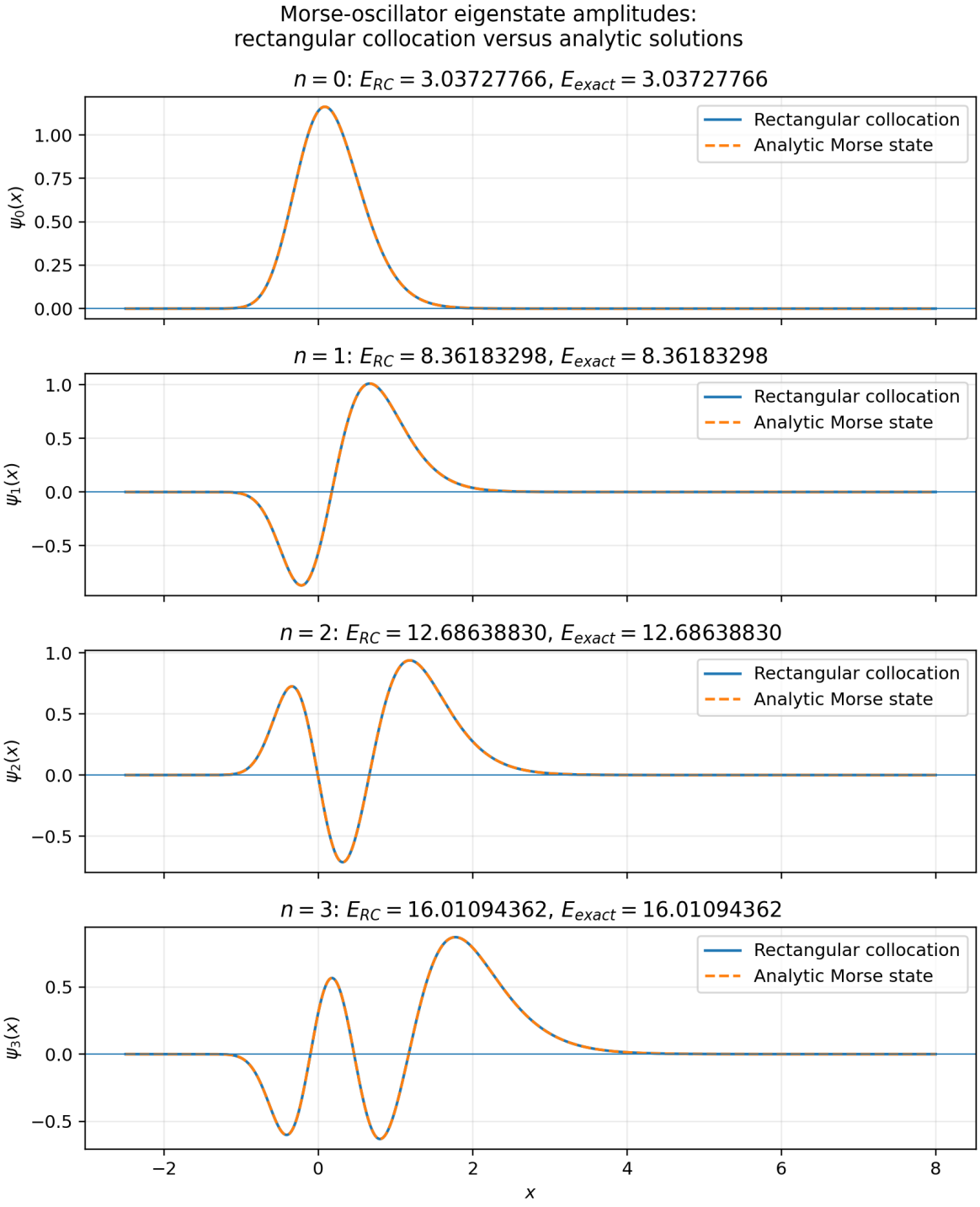

Figure 1: The first four Morse eigenstate amplitudes obtained by regular rectangular collocation, compared with the analytic Laguerre-polynomial solutions. The global sign of each numerical eigenvector is aligned with the corresponding analytic state before plotting.

When the figure is present in the same directory as this document, it can be included as follows:

For the chosen parameters,

$$\lambda = \sqrt{40} \approx 6.3249, \tag{32}$$

and the first exact energies are approximately

$$E_0 = 3.03727766, \tag{33}$$
$$E_1 = 8.36183298, \tag{34}$$
$$E_2 = 12.68638830, \tag{35}$$
$$E_3 = 16.01094362. \tag{36}$$

The supplied discretization reproduces these values closely.

9 Convergence checks

A reliable collocation result should be tested against:

1. increasing N_b and N_p;
2. changing the ratio N_p/N_b;
3. enlarging the coordinate interval;
4. varying the Gaussian shape parameter;
5. monitoring the singular values and condition number of $\boldsymbol{B}$;
6. checking the residual
$$r_n = \frac{\|(\boldsymbol{T}+\boldsymbol{V}\boldsymbol{B})\boldsymbol{c}_n - E_n\boldsymbol{B}\boldsymbol{c}_n\|_2}{\|\boldsymbol{B}\boldsymbol{c}_n\|_2}; \tag{37}$$
7. confirming that physical eigenvalues have negligible imaginary parts.

Low-lying states converge first because they are localized near the potential minimum (Fig. 1). States near dissociation extend farther into the large-x region and therefore require a larger coordinate interval and often more collocation points.

10 Two-dimensional implementation with Fekete points, Christoffel weights, and a product-contracted basis

The one-dimensional example demonstrates the basic rectangular-collocation idea, as shown in Fig. 1, but the principal motivation for weighted and chopped collocation is the rapid growth of tensor-product grids in several dimensions. This section develops the complete two-dimensional implementation supplied in `morse_2d_fekete_product_contracted_colab.ipynb`.

10.1 Two-dimensional Hamiltonian

Consider two coordinates x and y with the Hamiltonian

$$\hat{H} = -\frac{\hbar^2}{2m_x}\frac{\partial^2}{\partial x^2} - \frac{\hbar^2}{2m_y}\frac{\partial^2}{\partial y^2} + V_x(x) + V_y(y), \tag{38}$$

where

$$V_x(x) = D_x \left[1 - e^{-a_x(x-x_{e,x})}\right]^2, \tag{39}$$

$$V_y(y) = D_y \left[1 - e^{-a_y(y-y_{e,y})}\right]^2. \tag{40}$$

Because the potential is separable, the exact eigenfunctions and energies factorize:

$$\Psi^{\text{exact}}_{n_x n_y}(x, y) = \psi^{(x)}_{n_x}(x)\psi^{(y)}_{n_y}(y), \tag{41}$$

$$E^{\text{exact}}_{n_x n_y} = E^{(x)}_{n_x} + E^{(y)}_{n_y}. \tag{42}$$

This makes the model an especially useful benchmark: it is genuinely two-dimensional, but every computed level has a known analytic target.

10.2 Why not use the full primitive tensor product?

Suppose the primitive one-dimensional Gaussian bases contain $N^{(x)}_{\text{prim}}$ and $N^{(y)}_{\text{prim}}$ functions. The direct-product basis contains

$$N^{(2D)}_{\text{prim}} = N^{(x)}_{\text{prim}} N^{(y)}_{\text{prim}} \tag{43}$$

functions. Even a moderate choice of 42 primitive functions in each coordinate produces

$$42 \times 42 = 1764 \tag{44}$$

two-dimensional basis functions. The tensor grid of collocation points grows similarly.

The product-contracted strategy first solves each one-dimensional problem and retains only the low-energy eigenfunctions that are relevant to the desired two-dimensional spectrum. In the current beam-search/QR comparison notebook, seven numerical functions are retained in each coordinate, so the final product-contracted basis contains

$$N_{\text{PC}} = N^{(x)}_{\text{PC}} N^{(y)}_{\text{PC}} = 7 \times 7 = 49 \tag{45}$$

functions.

10.3 Step 1: solve the one-dimensional primitive problems

For each coordinate $q \in \{x, y\}$, use primitive Gaussian functions

$$g^{(q)}_j(q) = \exp\left[-\beta_q (q - c^{(q)}_j)^2\right]. \tag{46}$$

The one-dimensional rectangular-collocation equation is

$$\left(\boldsymbol{T}^{(q)} + \boldsymbol{V}^{(q)}\boldsymbol{B}^{(q)}\right)\boldsymbol{u}^{(q)}_\alpha = \varepsilon^{(q)}_\alpha \boldsymbol{B}^{(q)}\boldsymbol{u}^{(q)}_\alpha. \tag{47}$$

Multiplication by the pseudoinverse gives

$$\left(\boldsymbol{B}^{(q)}\right)^+ \left(\boldsymbol{T}^{(q)} + \boldsymbol{V}^{(q)}\boldsymbol{B}^{(q)}\right)\boldsymbol{u}^{(q)}_\alpha = \varepsilon^{(q)}_\alpha \boldsymbol{u}^{(q)}_\alpha. \tag{48}$$

The lowest $N^{(q)}_{\text{PC}}$ eigenvectors are retained.

10.4 Step 2: construct orthonormal contracted functions

Each retained contracted function is a linear combination of primitive Gaussians:

$$\phi_\alpha^{(x)}(x) = \sum_{j=1}^{N_{\text{prim}}^{(x)}} U_{j\alpha}^{(x)} g_j^{(x)}(x), \tag{49}$$

with an analogous expression for y.

Because rectangular collocation does not automatically return vectors that are orthonormal under the continuous coordinate-space inner product, the notebook reconstructs the retained functions on a dense grid and forms

$$S_{\alpha\gamma}^{(x)} = \int \phi_\alpha^{(x)}(x)\phi_\gamma^{(x)}(x)\, dx. \tag{50}$$

A symmetric orthogonalization is then applied:

$$\widetilde{\boldsymbol{U}}^{(x)} = \boldsymbol{U}^{(x)} \left(\boldsymbol{S}^{(x)}\right)^{-1/2}. \tag{51}$$

This is important because the Christoffel function has its standard meaning only when the basis is orthonormal, or at least consistently normalized.

10.5 Step 3: form the product-contracted basis

The two-dimensional basis functions are

$$\Phi_{\alpha\beta}(x,y) = \phi_\alpha^{(x)}(x)\phi_\beta^{(y)}(y). \tag{52}$$

For each collocation point (x_i, y_i),

$$B_{i,(\alpha\beta)} = \Phi_{\alpha\beta}(x_i, y_i). \tag{53}$$

The kinetic-energy action also factorizes:

$$\hat{T}\Phi_{\alpha\beta} = \left(\hat{T}_x\phi_\alpha^{(x)}\right)\phi_\beta^{(y)} + \phi_\alpha^{(x)}\left(\hat{T}_y\phi_\beta^{(y)}\right). \tag{54}$$

Equation (54) is evaluated directly at the selected collocation points. No two-dimensional kinetic-energy integrals are needed.

In matrix language, the product structure is

$$\boldsymbol{B}_{\text{TP}} = \boldsymbol{B}^{(x)} \otimes \boldsymbol{B}^{(y)} \tag{55}$$

on a complete tensor grid. The notebook avoids explicitly constructing a large Kronecker matrix. Instead, it evaluates the products row by row only at the candidate or selected points.

10.6 Step 4: compute the Christoffel function

For an orthonormal product-contracted basis $\{\Phi_\mu(x,y)\}_{\mu=1}^{N_{\text{PC}}}$, define

$$K(x,y) = \sum_{\mu=1}^{N_{\text{PC}}} |\Phi_\mu(x,y)|^2. \tag{56}$$

The Christoffel weight is

$$w(x,y) = \frac{1}{K(x,y)}. \tag{57}$$

For the product basis in Eq. (52),

$$K(x,y) = \sum_{\alpha,\beta} \left|\phi_\alpha^{(x)}(x)\right|^2 \left|\phi_\beta^{(y)}(y)\right|^2 \tag{58}$$

$$= \left[\sum_{\alpha} \left|\phi_\alpha^{(x)}(x)\right|^2\right] \left[\sum_{\beta} \left|\phi_\beta^{(y)}(y)\right|^2\right] \tag{59}$$

$$= K_x(x) K_y(y). \tag{60}$$

The Christoffel function measures the local collective magnitude of the basis. Where many basis functions are simultaneously large, K is large and the weight $w = 1/K$ is small. The weighting therefore prevents densely represented regions from dominating the discrete least-squares problem.

Below is a drop-in replacement for Sec. 10.7, using the notation and structure of the tutorial.

10.7 Step 5: select approximate Fekete points

The candidate tensor grid introduced above usually contains many more points than are needed for the final collocation calculation. The goal of the present step is to select a much smaller subset of points that still captures the linear information contained in the product-contracted basis.

Begin with the candidate set

$$\mathcal{X}_c = \{(x_i, y_i)\}_{i=1}^{N_c}, \tag{61}$$

where N_c is much larger than the number N_{PC} of product-contracted basis functions. Evaluating all basis functions at all candidate points gives the matrix

$$B_c \in \mathbb{R}^{N_c \times N_{\mathrm{PC}}}, \qquad (B_c)_{i\mu} = \Phi_\mu(x_i, y_i). \tag{62}$$

Each row of B_c corresponds to one candidate point, while each column corresponds to one basis function. Explicitly,

$$B_c = \begin{pmatrix} \Phi_1(x_1,y_1) & \Phi_2(x_1,y_1) & \cdots & \Phi_{N_{\mathrm{PC}}}(x_1,y_1) \\ \Phi_1(x_2,y_2) & \Phi_2(x_2,y_2) & \cdots & \Phi_{N_{\mathrm{PC}}}(x_2,y_2) \\ \vdots & \vdots & & \vdots \\ \Phi_1(x_{N_c},y_{N_c}) & \Phi_2(x_{N_c},y_{N_c}) & \cdots & \Phi_{N_{\mathrm{PC}}}(x_{N_c},y_{N_c}) \end{pmatrix}. \tag{63}$$

10.7.1 The interpolation matrix

To understand the point-selection problem, first suppose that exactly N_{PC} candidate points are retained. Denote their indices by

$$i_1, i_2, \ldots, i_{N_{\mathrm{PC}}}. \tag{64}$$

Taking the corresponding rows of B_c produces the square matrix

$$B_s = \begin{pmatrix} \Phi_1(x_{i_1},y_{i_1}) & \cdots & \Phi_{N_{\mathrm{PC}}}(x_{i_1},y_{i_1}) \\ \vdots & & \vdots \\ \Phi_1(x_{i_{N_{\mathrm{PC}}}},y_{i_{N_{\mathrm{PC}}}}) & \cdots & \Phi_{N_{\mathrm{PC}}}(x_{i_{N_{\mathrm{PC}}}},y_{i_{N_{\mathrm{PC}}}}) \end{pmatrix}. \tag{65}$$

This matrix is called the *interpolation matrix* associated with the selected points.

The name follows directly from its action. Any function in the product-contracted space can be written as

$$f(x,y) = \sum_{\mu=1}^{N_{\mathrm{PC}}} c_\mu \Phi_\mu(x,y). \tag{66}$$

Evaluating this function at the selected points gives

$$\begin{pmatrix} f(x_{i_1}, y_{i_1}) \\ \vdots \\ f(x_{i_{N_{\mathrm{PC}}}}, y_{i_{N_{\mathrm{PC}}}}) \end{pmatrix} = B_s \begin{pmatrix} c_1 \\ \vdots \\ c_{N_{\mathrm{PC}}} \end{pmatrix}. \tag{67}$$

Thus, B_s maps the basis coefficients to the values of the represented function at the selected points.

If B_s is nonsingular, the coefficients can be recovered uniquely from these sampled values:

$$\mathbf{c} = B_s^{-1}\mathbf{f}_s. \tag{68}$$

The selected points therefore distinguish every function in the finite basis space. If B_s is singular, two different coefficient vectors can produce the same sampled values, and unique interpolation is impossible. If B_s is nearly singular, the interpolation is formally unique but highly sensitive to numerical errors.

A good set of interpolation points should consequently produce a matrix B_s that is far from singular.

10.7.2 Exact Fekete points

For a square interpolation matrix, the determinant provides a measure of the linear independence of its rows:

$$\left|\det B_s\right|. \tag{69}$$

Geometrically, this quantity is the volume of the parallelepiped spanned by the row vectors of B_s. If the rows are linearly dependent, this volume is zero. [1] If they are nearly dependent, the

[1] To see why the determinant measures a volume, let the rows of the square interpolation matrix $B_s \in \mathbb{R}^{N_{\mathrm{PC}} \times N_{\mathrm{PC}}}$ be denoted by

$$\mathbf{r}_1^T, \mathbf{r}_2^T, \ldots, \mathbf{r}_{N_{\mathrm{PC}}}^T. \tag{70}$$

These vectors span the parallelepiped

$$\mathcal{P} = \left\{ \sum_{i=1}^{N_{\mathrm{PC}}} t_i \mathbf{r}_i \ : \ 0 \le t_i \le 1 \right\}. \tag{71}$$

In two dimensions, two vectors span a parallelogram. Its area is the length of the first vector multiplied by the component of the second vector perpendicular to the first.

To show that the determinant is the area of the parallelogram, let $\mathbf{u} = (u_1, u_2)$ and $\mathbf{v} = (v_1, v_2)$. A unit vector perpendicular to $\mathbf{u}$, obtained by rotating $\mathbf{u}$ counterclockwise through 90°, is $\widehat{\mathbf{n}} = (-u_2, u_1)/\|\mathbf{u}\|$. The signed component of $\mathbf{v}$ perpendicular to $\mathbf{u}$ is therefore

$$\mathbf{v} \cdot \widehat{\mathbf{n}} = \frac{-u_2 v_1 + u_1 v_2}{\|\mathbf{u}\|} = \frac{u_1 v_2 - v_1 u_2}{\|\mathbf{u}\|}.$$

Multiplying this perpendicular component by the base length $\|\mathbf{u}\|$ gives the signed area of the parallelogram:

$$\|\mathbf{u}\| \, (\mathbf{v} \cdot \widehat{\mathbf{n}}) = u_1 v_2 - v_1 u_2.$$

The sign records the orientation of the ordered pair $(\mathbf{u}, \mathbf{v})$; consequently, the ordinary nonnegative area is

$$\|\mathbf{u}\| \, |\mathbf{v} \cdot \widehat{\mathbf{n}}| = \left|u_1 v_2 - v_1 u_2\right|.$$

In three dimensions, three vectors span a parallelepiped whose volume is the base area multiplied by the perpendicular height. The same geometric idea extends to higher dimensions. A convenient algebraic expression for this volume uses the Gram matrix of the row vectors. The Gram matrix is

$$G_{ij} = \mathbf{r}_i \cdot \mathbf{r}_j. \tag{72}$$

Since the rows of B_s are the vectors $\mathbf{r}_i^T$, the complete Gram matrix is

$$G = B_s B_s^T. \tag{73}$$

volume is small. A relatively large determinant indicates that the selected points probe the basis in substantially different ways.

Exact Fekete points are defined as the subset of N_{PC} candidate points that maximizes the determinant magnitude:

$$\mathcal{X}_{\mathrm{Fekete}} = \underset{\substack{\mathcal{X}_s \subset \mathcal{X}_c \\ |\mathcal{X}_s| = N_{\mathrm{PC}}}}{\arg\max} \left|\det B_s\right|. \tag{81}$$

The definition is simple, but direct computation is generally impractical. The number of possible subsets is

$$\binom{N_c}{N_{\mathrm{PC}}}, \tag{82}$$

which becomes enormous even for moderate values of N_c and N_{PC}. An exhaustive determinant search is therefore replaced by a computationally efficient greedy approximation.

10.7.3 Christoffel scaling

Before selecting points, the candidate basis matrix is scaled using the Christoffel weights:

$$B_{w,c} = W_c^{1/2} B_c, \qquad (W_c)_{ii} = w(x_i, y_i). \tag{83}$$

Since

$$w(x_i, y_i) = \frac{1}{K(x_i, y_i)}, \qquad K(x_i, y_i) = \sum_{\mu=1}^{N_{\mathrm{PC}}} \left|\Phi_\mu(x_i, y_i)\right|^2, \tag{84}$$

the squared Euclidean norm of row i of $B_{w,c}$ is

$$\left\|(B_{w,c})_{i,:}\right\|_2^2 = w(x_i, y_i) \sum_{\mu=1}^{N_{\mathrm{PC}}} \left|\Phi_\mu(x_i, y_i)\right|^2 \tag{85}$$

$$= w(x_i, y_i) K(x_i, y_i) \tag{86}$$

$$= 1. \tag{87}$$

The squared volume of the parallelepiped spanned by a collection of vectors is equal to the determinant of their Gram matrix:

$$\mathrm{Vol}(\mathcal{P})^2 = \det G. \tag{74}$$

Therefore,

$$\mathrm{Vol}(\mathcal{P})^2 = \det\left(B_s B_s^T\right) \tag{75}$$

$$= \det(B_s)\det(B_s^T) \tag{76}$$

$$= \det(B_s)^2, \tag{77}$$

where

$$\det(B_s^T) = \det(B_s). \tag{78}$$

Taking the nonnegative square root gives

$$\mathrm{Vol}(\mathcal{P}) = \left|\det B_s\right|. \tag{79}$$

The absolute value is required because a determinant is an *oriented* volume. Interchanging two row vectors reverses their orientation and changes the sign of the determinant, but it does not change the geometric volume of the parallelepiped.

Equation (79) also explains the connection between the determinant and linear independence. If the row vectors are linearly dependent, they lie in a lower-dimensional subspace, so the parallelepiped is flattened in at least one direction. Its volume is then zero, and

$$\det B_s = 0. \tag{80}$$

If the rows are nearly linearly dependent, the parallelepiped is very thin, and $|\det B_s|$ is small. Selecting points that make $|\det B_s|$ large therefore favors interpolation rows that probe distinct and complementary directions in the product-contracted basis space.

Thus, Christoffel scaling places the candidate rows on a common norm scale.

Without this normalization, a point could be selected simply because all basis functions happen to have large amplitudes there. After normalization, the point-selection procedure emphasizes whether a candidate row contributes a new linearly independent direction, rather than merely whether that row has a large magnitude.

10.7.4 Why the candidate matrix is transposed

The weighted candidate matrix has dimensions

$$B_{w,c} \in \mathbb{R}^{N_c \times N_{\mathrm{PC}}}. \tag{88}$$

Its rows correspond to candidate points. Standard column-pivoted QR factorization, however, selects and reorders *columns*. To make candidate points the objects being selected, the matrix is transposed:

$$B_{w,c}^T \in \mathbb{R}^{N_{\mathrm{PC}} \times N_c}. \tag{89}$$

Column i of $B_{w,c}^T$ is

$$\mathbf{a}_i = \sqrt{w_i} \begin{pmatrix} \Phi_1(x_i, y_i) \\ \Phi_2(x_i, y_i) \\ \vdots \\ \Phi_{N_{\mathrm{PC}}}(x_i, y_i) \end{pmatrix}. \tag{90}$$

It contains the weighted values of every product-contracted basis function at candidate point (x_i, y_i). Selecting columns of $B_{w,c}^T$ is therefore exactly equivalent to selecting candidate points.

10.7.5 Column-pivoted QR factorization

The column-pivoted QR factorization is written as

$$B_{w,c}^T P = QR. \tag{91}$$

Here, Q has orthonormal columns, R is upper trapezoidal, and P is a permutation matrix. The permutation rearranges the columns of $B_{w,c}^T$, and hence rearranges the candidate points:

$$B_{w,c}^T P = \begin{pmatrix} \mathbf{a}_{p_1} & \mathbf{a}_{p_2} & \cdots & \mathbf{a}_{p_{N_c}} \end{pmatrix}. \tag{92}$$

The ordered indices

$$p_1, p_2, \ldots, p_{N_c} \tag{93}$$

rank the candidate points according to the greedy QR selection.

The logic of the pivoting can be understood sequentially. At the first step, the algorithm selects a column with large norm. After one or more columns have been chosen, each remaining candidate column is separated into two parts:

$$\mathbf{a}_j = Q_k Q_k^T \mathbf{a}_j + \left(I - Q_k Q_k^T\right) \mathbf{a}_j, \tag{94}$$

where Q_k spans the columns selected during the first k steps. The first term lies in the space already represented by the selected points. The second term is the part that cannot yet be represented.

The next pivot is chosen approximately according to

$$p_{k+1} = \underset{j \text{ not yet selected}}{\arg\max} \left\| \left(I - Q_k Q_k^T\right) \mathbf{a}_j \right\|_2. \tag{95}$$

In words, the next point is the one whose weighted basis-value vector has the largest component orthogonal to the information already collected.

This procedure discourages the selection of redundant points. If two nearby candidate points produce nearly parallel basis-value vectors, then after one is selected, the orthogonal remainder of the other is small. The second point will therefore tend to appear later in the pivot ordering. A point whose basis-value vector points in a substantially new direction will have a larger orthogonal remainder and will be selected earlier.

Column-pivoted QR therefore builds a set of points whose interpolation rows are as linearly independent as possible at each greedy step.

10.7.6 Connection with approximate Fekete points

Column-pivoted QR does not solve the global determinant-maximization problem in Eq. (81). It instead makes a sequence of local, greedy choices. Nevertheless, the diagonal entries of R quantify the new independent component added at each step, and for the first N_{PC} selected columns,

$$|\det R_{1:N_{\mathrm{PC}},1:N_{\mathrm{PC}}}| = \prod_{k=1}^{N_{\mathrm{PC}}} |R_{kk}|. \tag{96}$$

Because each pivot attempts to keep the next diagonal factor large, the procedure tends to produce a square interpolation matrix with a relatively large determinant and good numerical rank. The resulting points are therefore called *approximate Fekete points* or *Fekete-like points*.

In the notebook, the selection is implemented by

```
Q, R, pivots = qr(
    Bw_candidate.T,
    pivoting=True,
    mode="economic"
)

selected_indices = pivots[:N_SELECTED]
selected_xy = candidate_xy[selected_indices]
```

The transpose in `Bw_candidate.T` is essential. Without the transpose, column pivoting would rank the basis functions rather than the candidate points.

If exactly N_{PC} points are retained, the selected interpolation matrix is square. In the present calculation, however, slightly more points are retained:

$$N_p > N_{\mathrm{PC}}. \tag{97}$$

The first N_{PC} pivots provide a Fekete-like square core. The additional pivots add points that continue to contribute new information, producing a rectangular sampling matrix with dimensions

$$B_s \in \mathbb{R}^{N_p \times N_{\mathrm{PC}}}. \tag{98}$$

The final choice

$$N_p = 2N_{\mathrm{PC}} \tag{99}$$

in the current comparison notebook deliberately uses a factor-of-two rectangular oversampling. This makes the comparison between point-selection methods less sensitive to the particular square core selected during the first N_{PC} steps.

10.8 Step 6: weighted rectangular collocation on the selected set

At the selected points, build

$$\boldsymbol{B}_{i\mu} = \Phi_\mu(x_i, y_i), \tag{100}$$

$$\boldsymbol{T}_{i\mu} = \left[\hat{T}\Phi_\mu(x, y)\right]_{(x_i,y_i)}, \tag{101}$$

$$\boldsymbol{H}_{\mathrm{col}} = \boldsymbol{T} + \boldsymbol{V}\boldsymbol{B}, \tag{102}$$

where

$$V_{ii} = V_x(x_i) + V_y(y_i). \tag{103}$$

Using the selected Christoffel weights,

$$\boldsymbol{W} = \mathrm{diag}(w_1, \ldots, w_{N_p}), \tag{104}$$

the weighted left inverse is

$$\boldsymbol{M}_W = \left(\boldsymbol{B}^{\mathsf{T}}\boldsymbol{W}\boldsymbol{B}\right)^{-1}\boldsymbol{B}^{\mathsf{T}}\boldsymbol{W}. \tag{105}$$

The final product-contracted eigenproblem is

$$\boldsymbol{M}_W\boldsymbol{H}_{\mathrm{col}}\boldsymbol{c}_n = E_n\boldsymbol{c}_n. \tag{106}$$

10.9 How the Colab implementation maps onto the equations

The notebook is organized so that each mathematical object has a direct code counterpart:

Mathematical object	Notebook variable or function
Primitive Gaussian matrix $\boldsymbol{B}^{(q)}$	`gaussian_basis_and_kinetic`
One-dimensional solutions	`solve_1d`
Orthonormal contracted coefficients	`contracted_coefficients`
Product basis values	`rowwise_product`
Product kinetic action	`rowwise_product_kinetic`
Candidate Christoffel function	`K_candidate`
Candidate Christoffel weights	`w_candidate`
Pivoted-QR Fekete ordering	`qr(..., pivoting=True)`
Selected coordinates	`selected_xy`
Weighted left inverse	`Mweighted`
Final effective Hamiltonian	`Apc`

The product basis is assembled with the NumPy expression

```
np.einsum("ia,ib->iab", Px, Py)
```

which forms all products $\phi_\alpha^{(x)}(x_i)\phi_\beta^{(y)}(y_i)$ for each point i without explicitly building a full tensor grid matrix.

The pivoted-QR step is

```
Q, R, pivots = qr(Bw_candidate.T, pivoting=True)
selected_indices = pivots[:N_SELECTED]
```

The transpose is essential: its columns correspond to candidate points, so column pivoting selects points rather than basis functions.

10.10 Outputs generated by the notebook

The Colab implementation produces:

- a scatter plot of the full candidate grid and the selected Fekete-like points;
- a contour map of $\log_{10} w(x, y)$ with the selected points superimposed;
- a table containing every selected coordinate, its Christoffel function, and its Christoffel weight;
- a comparison of the weighted product-contracted energies with the exact sums in Eq. (42);
- contour plots of the first four two-dimensional eigenstate amplitudes;
- the numerical-minus-analytic ground-state amplitude difference.

The CSV file `fekete_points_and_christoffel_weights.csv` makes the selected point set reusable in later calculations.

10.11 Extension to a coupled potential

The product-contracted basis and selected points are not restricted to a separable potential. For

$$V(x, y) = V_x(x) + V_y(y) + V_{\text{coupling}}(x, y), \tag{107}$$

the kinetic matrix, Christoffel weights, and Fekete selection remain unchanged. Only the pointwise potential vector changes:

$$V_{ii} = V_x(x_i) + V_y(y_i) + V_{\text{coupling}}(x_i, y_i). \tag{108}$$

This is one of the central advantages of collocation: once the basis and points are available, changing the potential requires only evaluating the new potential at those points.

10.12 Convergence diagnostics for the two-dimensional calculation

The following quantities should be monitored:

1. the number of primitive functions in each coordinate;
2. the number of retained contracted functions;
3. the density and coordinate range of the candidate grid;
4. the oversampling ratio N_p/N_{PC};
5. the condition number
$$\kappa\left(\boldsymbol{W}^{1/2}\boldsymbol{B}\right); \tag{109}$$
6. the weighted collocation residual
$$r_n = \frac{\left\|\boldsymbol{W}^{1/2}\left(\boldsymbol{H}_{\text{col}}\boldsymbol{c}_n - E_n\boldsymbol{B}\boldsymbol{c}_n\right)\right\|_2}{\left\|\boldsymbol{W}^{1/2}\boldsymbol{B}\boldsymbol{c}_n\right\|_2}; \tag{110}$$
7. stability of the low-energy spectrum when the selected point set is enlarged.

The separable model is particularly useful for diagnosing errors. If the one-dimensional contracted bases are converged but the two-dimensional energies are inaccurate, the problem usually lies in the candidate grid, the Fekete selection, the weighting, or numerical conditioning rather than in the underlying one-dimensional representation.

11 Beam-search refinement of the greedy QR point selection

Column-pivoted QR is an efficient approximation to Fekete-point selection, but it is intrinsically greedy: once a pivot is chosen, that decision is never revisited. The current comparison notebook therefore adds a beam-search alternative and evaluates it against pivoted QR using exactly the same Christoffel-scaled candidate matrix

$$\boldsymbol{B}_w = \boldsymbol{W}_c^{1/2}\boldsymbol{B}_c. \tag{111}$$

For an orthonormal contracted basis the Christoffel scaling makes each candidate row have unit norm up to numerical error,

$$\|(\boldsymbol{B}_w)_{i,:}\|_2^2 = w_i \sum_{\mu} |\Phi_\mu(\boldsymbol{x}_i)|^2 \simeq 1. \tag{112}$$

Thus the selection problem is primarily about finding complementary *directions* in basis-value space rather than simply choosing points where the basis amplitudes are largest.

11.1 Regularized log-determinant objective

For a partial selected set S, define

$$\boldsymbol{G}_S = \delta\boldsymbol{I} + \boldsymbol{B}_{w,S}^{\mathsf{T}}\boldsymbol{B}_{w,S}, \qquad \delta > 0, \tag{113}$$

and score it by

$$J(S) = \log\det \boldsymbol{G}_S - N_{\mathrm{PC}} \log\delta. \tag{114}$$

The subtraction only fixes $J(\varnothing) = 0$. When $|S| = N_{\mathrm{PC}}$ and $\delta \to 0$, this objective approaches the logarithm of the squared weighted interpolation determinant. Unlike the determinant of a square interpolation matrix, Eq. (114) remains meaningful for $|S| > N_{\mathrm{PC}}$, so the same criterion can rank both the Fekete-like core and the extra points used for rectangular oversampling.

If candidate row $\boldsymbol{a}_j^{\mathsf{T}}$ is added to S, then

$$\boldsymbol{G}_{S\cup\{j\}} = \boldsymbol{G}_S + \boldsymbol{a}_j\boldsymbol{a}_j^{\mathsf{T}}. \tag{115}$$

The matrix-determinant lemma gives the marginal gain

$$\Delta_j(S) = \log\left(1 + \boldsymbol{a}_j^{\mathsf{T}}\boldsymbol{G}_S^{-1}\boldsymbol{a}_j\right). \tag{116}$$

A redundant point has small gain, whereas a point that strengthens a weakly represented basis-space direction has large gain.

11.2 Beam-search algorithm and its relationship to QR

At every depth, the implementation expands each retained partial set through its best candidate additions, pools the children, removes duplicate subsets, and keeps only the highest-scoring sets. In the current notebook

$$b = 10, \qquad r = 14, \qquad \delta = 10^{-8}, \tag{117}$$

where b is the beam width and r is the branch width. The pivoted-QR order is supplied only as a deterministic tie-breaker. This detail matters because Christoffel scaling makes the singleton scores nearly degenerate. The beam search is therefore not initialized from a completely unrelated ordering, but after the first ties are broken it is free to retain and compare multiple alternative point sets. The final beam and QR sets share only five of 98 points, confirming that the tie-break does not force the two procedures onto the same trajectory.

The exact implementation used in the notebook is reproduced below.

Listing 1: Beam-search selection used in the comparison notebook.

```
from dataclasses import dataclass


@dataclass
class BeamState:
    """One partial point set in the beam."""

    selected: tuple[int, ...]
    information: np.ndarray
    score: float


def regularized_logdet_score(
    weighted_candidate_matrix,
    selected_indices,
    ridge=1.0e-8,
):
    """Return the regularized D-optimal score for one selected set."""
    X = np.asarray(weighted_candidate_matrix, dtype=float)
    indices = np.asarray(selected_indices, dtype=int)

    information = ridge * np.eye(X.shape[1])
    if indices.size:
        X_selected = X[indices]
        information += X_selected.T @ X_selected

    sign, logdet = np.linalg.slogdet(information)
    if sign <= 0:
        return -np.inf

    return logdet - X.shape[1] * np.log(ridge)


def beam_select_points(
    weighted_candidate_matrix,
    n_selected,
    *,
    beam_width=12,
    branch_width=16,
    ridge=1.0e-8,
    tie_break_order=None,
    verbose=False,
):
    """
    Select candidate rows by regularized log-determinant beam search.

    Parameters
    ----------
    weighted_candidate_matrix : ndarray, shape (n_candidates, n_basis)
        Christoffel-weighted candidate basis matrix.
    n_selected : int
        Number of candidate points to retain.
    beam_width : int
        Number of partial point sets retained at each depth.
    branch_width : int
        Number of candidate additions explored from each beam state.
    ridge : float
```

```
    Positive regularization in delta I + X_S.T @ X_S.
tie_break_order : array-like, optional
    Candidate ordering used only when marginal gains are tied. Passing
    the pivoted-QR ordering gives a reproducible hybrid initialization.
verbose : bool
    Print progress while the beam is constructed.

Returns
-------
selected_indices : ndarray
    Candidate indices in their beam-search selection order.
best_score : float
    Final regularized log-determinant score.
"""
X = np.asarray(weighted_candidate_matrix, dtype=float)

if X.ndim != 2:
    raise ValueError("weighted_candidate_matrix must be two-dimensional.")
if not np.all(np.isfinite(X)):
    raise ValueError("weighted_candidate_matrix contains non-finite values.")

n_candidates, n_basis = X.shape

if not 1 <= n_selected <= n_candidates:
    raise ValueError("n_selected must lie between 1 and n_candidates.")
if beam_width < 1 or branch_width < 1:
    raise ValueError("beam_width and branch_width must be positive.")
if ridge <= 0.0:
    raise ValueError("ridge must be positive.")

if tie_break_order is None:
    tie_break_order = np.arange(n_candidates)
else:
    tie_break_order = np.asarray(tie_break_order, dtype=int)
    if (
        tie_break_order.shape != (n_candidates,)
        or not np.array_equal(
            np.sort(tie_break_order), np.arange(n_candidates)
        )
    ):
        raise ValueError(
            "tie_break_order must be a permutation of all candidate indices."
        )

tie_rank = np.empty(n_candidates, dtype=int)
tie_rank[tie_break_order] = np.arange(n_candidates)

base_logdet = n_basis * np.log(ridge)
initial_information = ridge * np.eye(n_basis)
beam = [BeamState((), initial_information, 0.0)]

for depth in range(n_selected):
    # Different selection orders can lead to the same subset. Keying by
    # the sorted subset prevents those duplicates from occupying the beam.
    children = {}

    for state in beam:
        # For every candidate a_i, compute a_i^T G_S^{-1} a_i.
```

```
        solved = np.linalg.solve(state.information, X.T).T
        leverage = np.einsum("ij,ij->i", solved, X)
        leverage = np.maximum(leverage, 0.0)

        if state.selected:
            leverage[np.asarray(state.selected, dtype=int)] = -np.inf

        available = np.flatnonzero(np.isfinite(leverage))
        n_branches = min(branch_width, available.size)
        marginal_gains = np.log1p(leverage[available])

        # Primary key: decreasing marginal gain.
        # Secondary key: the supplied deterministic candidate ordering.
        local_order = np.lexsort(
            (tie_rank[available], -marginal_gains)
        )[:n_branches]

        for candidate in available[local_order]:
            candidate = int(candidate)
            basis_vector = X[candidate]
            new_information = (
                state.information
                + np.outer(basis_vector, basis_vector)
            )

            sign, logdet = np.linalg.slogdet(new_information)
            if sign <= 0:
                continue

            new_selected = state.selected + (candidate,)
            new_score = logdet - base_logdet
            subset_key = tuple(sorted(new_selected))

            child = BeamState(
                selected=new_selected,
                information=new_information,
                score=new_score,
            )
            previous = children.get(subset_key)

            if previous is None or child.score > previous.score:
                children[subset_key] = child

    if not children:
        raise RuntimeError("Beam search produced no valid child states.")

    beam = sorted(
        children.values(),
        key=lambda state: (
            -state.score,
            tuple(tie_rank[i] for i in state.selected),
        ),
    )[:beam_width]

    if verbose and (
        depth == 0
        or (depth + 1) % 10 == 0
        or depth + 1 == n_selected
```

```
        ):
            print(
                f"Beam depth {depth + 1:3d}/{n_selected}: "
                f"best score = {beam[0].score:.8f}"
            )

    best_state = beam[0]
    return np.asarray(best_state.selected, dtype=int), best_state.score
```

The QR baseline is much shorter:

Listing 2: Pivoted-QR baseline evaluated on the same weighted candidate matrix.

```
Bw_candidate = np.sqrt(w_candidate)[:, None] * B_candidate

# Pivoted QR provides a deterministic baseline and a reproducible tie-break
# order for the beam search.
Q, R, pivots = qr(Bw_candidate.T, pivoting=True, mode="economic")
qr_indices = np.asarray(pivots[:N_SELECTED], dtype=int)
qr_score = regularized_logdet_score(
    Bw_candidate,
    qr_indices,
    ridge=BEAM_RIDGE,
)
qr_condition = np.linalg.cond(Bw_candidate[qr_indices])

# Beam search is evaluated regardless of which method is selected for the
# original downstream aliases, so every comparison is available in one run.
beam_indices, beam_score = beam_select_points(
    Bw_candidate,
    N_SELECTED,
    beam_width=BEAM_WIDTH,
    branch_width=BEAM_BRANCH_WIDTH,
    ridge=BEAM_RIDGE,
    tie_break_order=pivots,
    verbose=True,
)
beam_condition = np.linalg.cond(Bw_candidate[beam_indices])
```

The beam implementation above stores the regularized information matrix and uses dense linear solves to evaluate Eq. (116). It is written for clarity rather than minimum asymptotic cost. A faster implementation can store $\boldsymbol{G}_S^{-1}$ and update it by Sherman–Morrison, but that optimization is not used for the numerical results reported below.

12 Updated Colab implementation and numerical setup

The revised notebook `beam_search_qr_comparison.ipynb` solves the same two-dimensional Morse benchmark with both point-selection procedures in parallel. The calculation uses

$$N_{\mathrm{prim}}^{(x)} = N_{\mathrm{prim}}^{(y)} = 70, \tag{118}$$

$$N_{\mathrm{col}}^{(x)} = N_{\mathrm{col}}^{(y)} = 210, \tag{119}$$

$$N_{\mathrm{PC}}^{(x)} = N_{\mathrm{PC}}^{(y)} = 7, \tag{120}$$

$$N_{\mathrm{PC}} = 49, \tag{121}$$

$$N_c = 45 \times 45 = 2025, \tag{122}$$

$$N_p = 98 = 2N_{\mathrm{PC}}. \tag{123}$$

The coordinate intervals are $-2.8 \le x \le 11$ and $-3.0 \le y \le 11$, with Gaussian shape parameter $s = 0.35$ and pseudoinverse cutoff 10^{-11}. The contraction reduces the primitive $70^2 = 4900$ product space to 49 product-contracted functions, a factor of 100.

The seven retained one-dimensional numerical levels are

$$\boldsymbol{\varepsilon}^{(x)} = (3.037278, 8.361833, 12.686388, 16.010944, 18.335499, 19.660060, 20.064906), \tag{124}$$

$$\boldsymbol{\varepsilon}^{(y)} = (2.237508, 6.170650, 9.381292, 11.869434, 13.635075, 14.678246, 15.061985). \tag{125}$$

There is an important benchmark caveat. For the chosen Morse parameters, $\lambda_x - 1/2 = 5.8246$ and $\lambda_y - 1/2 = 5.9438$, so the physical bound quantum numbers are $n = 0, \ldots, 5$ in both coordinates. The seventh retained numerical functions lie slightly above the dissociation limits ($D_x = 20$ and $D_y = 15$). Therefore, table entries whose analytic label contains $n_x = 6$ or $n_y = 6$ use the algebraic Morse energy formula outside its physical bound-state domain. They are useful as numerical diagnostics of the finite basis, but should not be interpreted as exact bound-state benchmarks.

13 Beam search versus pivoted QR: numerical results

13.1 Point-set quality and spatial distribution

The two algorithms select strikingly different subsets of the same 2025-point candidate grid:

Method	$J(S)$	$\kappa(\boldsymbol{B}_w)$	Shared points	Jaccard index
Beam search	935.662318	1.458270	5/98	0.026178
Pivoted QR	911.795241	7.358227	5/98	0.026178

The beam objective exceeds the QR value by 23.8671 log units. Because this is a log determinant of the regularized information matrix, the corresponding regularized determinant ratio is approximately $\exp(23.8671) \simeq 2.3 \times 10^{10}$. More directly relevant to the collocation solve, the condition number of the weighted sampling matrix is reduced by a factor of about 5.05.

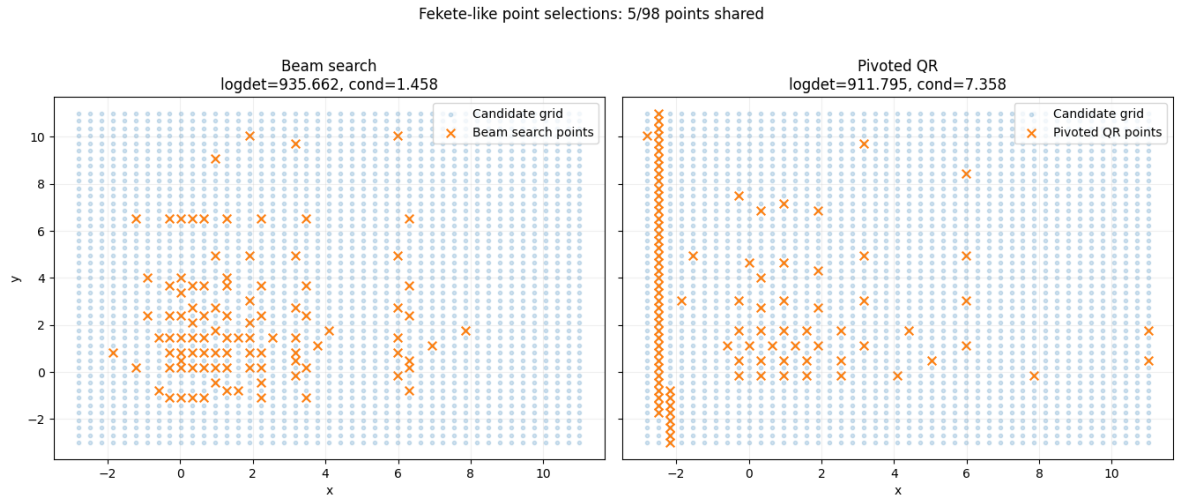

Figure 2: Side-by-side comparison of the 98 Fekete-like points selected by beam search and pivoted QR from the same 45×45 candidate grid. Only five points are shared. The numerical titles report the regularized log-determinant score and the condition number of the Christoffel-weighted sampling matrix. The beam result is not simply a small perturbation of the greedy QR ordering; it identifies a substantially different global distribution with much better conditioning.

Figure 2 is particularly informative because both panels use the same candidate grid and axis limits. QR follows one locally orthogonalizing trajectory, whereas beam search can preserve alternatives whose value becomes apparent only after several additional points have been selected. The low overlap between the final sets is therefore expected rather than pathological: many individual points are nearly interchangeable locally, but their collective geometry controls the final determinant and condition number.

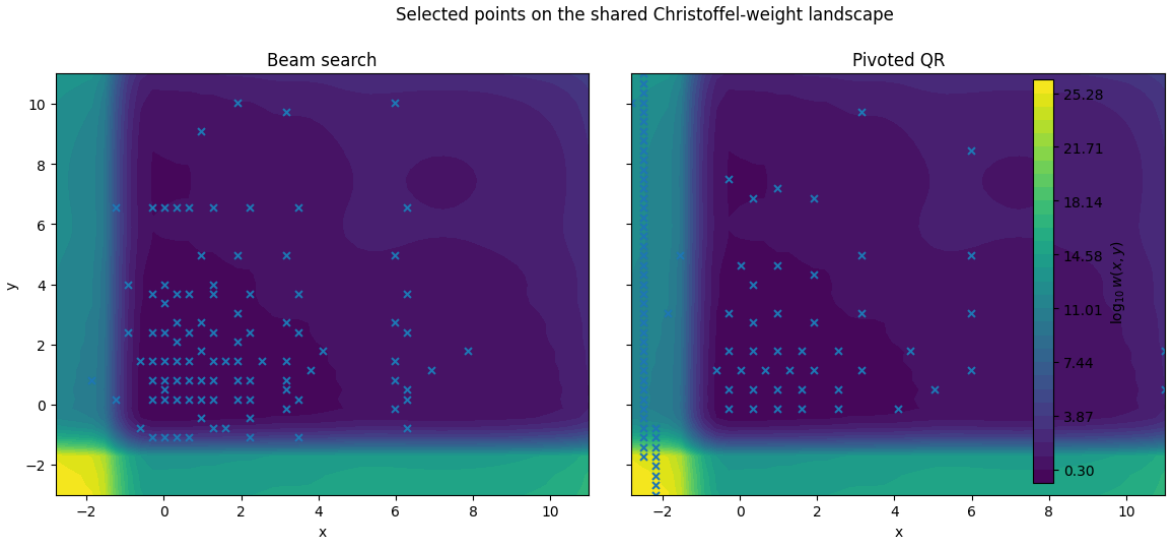

Figure 3: Beam-search and pivoted-QR selections superimposed on the same $\log_{10} w(x, y)$ Christoffel-weight landscape. A shared contour normalization makes the two panels directly comparable. Both methods sample the dynamically relevant central region and the more weakly represented tails, but they distribute points differently within those regions.

The candidate Christoffel function spans $6.70 \times 10^{-27} \leq K \leq 3.902$, corresponding to $0.256 \leq w \leq 1.49 \times 10^{26}$. Figure 3 shows why raw spatial density is not the correct criterion for judging a point set: the weights compensate regions in which the contracted basis is naturally large or small. The objective acts on the weighted row directions, not on Euclidean point spacing alone.

13.2 Conditioning of the weighted collocation solve

After the points are selected, both methods use the identical weighted rectangular-collocation equations. The resulting conditioning is

Method	$\kappa(\boldsymbol{W}^{1/2}\boldsymbol{B})$	$\kappa(\boldsymbol{B}^{\mathsf{T}}\boldsymbol{W}\boldsymbol{B})$
Beam search	1.458270	2.126551
Pivoted QR	7.358227	54.143509

The normal-equation condition number is about 25.5 times smaller for beam search. The numbers also illustrate the familiar squaring of the condition number when normal equations are formed. The beam-selected matrix is close to an isometry after Christoffel scaling, so the current normal-equation solve is numerically benign. For less favorable problems, the direct weighted least-squares form

$$\min_{\boldsymbol{X}} \left\| \boldsymbol{W}^{1/2}\boldsymbol{B}\boldsymbol{X} - \boldsymbol{W}^{1/2}\boldsymbol{H}_{\mathrm{col}} \right\|_F \tag{126}$$

remains preferable because it avoids explicitly squaring the condition number.

13.3 Energy accuracy for the separable benchmark

For the first 25 sorted levels, the aggregate absolute errors are

Method	Mean absolute error	Maximum absolute error
Beam search	0.141282	0.812722
Pivoted QR	0.394179	1.569761

Thus the beam search lowers the mean absolute error by about 64.2% and the maximum error by about 48.2% for this particular calculation. The improvement is not uniform state by state, which is expected because the collocation projection is nonvariational and the two point sets emphasize different basis-space directions. The important observation is that the better global conditioning is accompanied by a clear reduction in the aggregate spectral error.

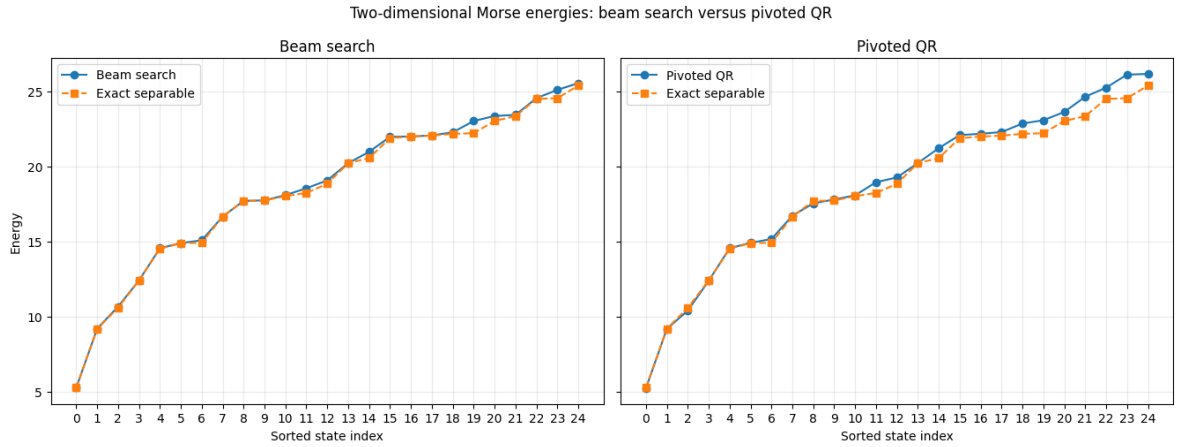

Figure 4: First 25 two-dimensional energies for beam search and pivoted QR, each compared with the same sorted separable Morse reference spectrum. The low states are close for both methods, while the beam-selected spectrum tracks the reference more consistently as the state index increases.

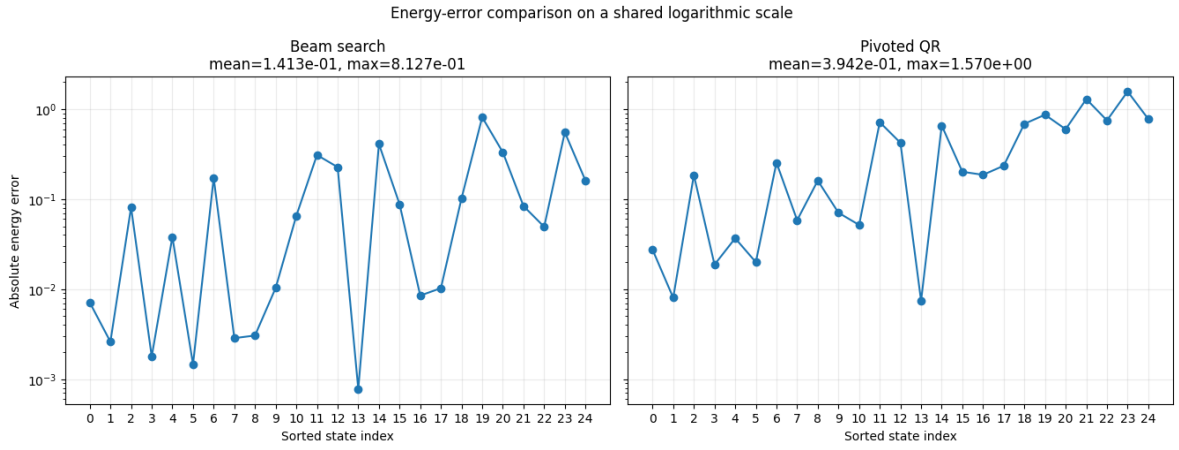

Figure 5: Absolute energy errors on a common logarithmic scale. Beam search gives the smaller mean and maximum errors over the 25-state comparison, although individual states can favor either selection. Entries involving $n_x = 6$ or $n_y = 6$ should be treated as finite-basis diagnostics rather than exact bound-state comparisons.

The full numerical comparison is listed below. The explicit (n_x, n_y) labels make the high-state caveat visible rather than hiding it in the sorted index.

Table 1: First 25 sorted separable levels and collocation energies for both point-selection methods. $|\Delta E| = |E_{\text{coll}} - E_{\text{exact}}|$.

k	(n_x, n_y)	E_{exact}	E_{beam}	$\lvert\Delta E\rvert_{\text{beam}}$	E_{QR}	$\lvert\Delta E\rvert_{\text{QR}}$
0	(0,0)	5.274786	5.281902	0.007116	5.247334	0.027452
1	(0,1)	9.207928	9.210533	0.002605	9.199898	0.008030
2	(1,0)	10.599341	10.680483	0.081142	10.414522	0.184820
3	(0,2)	12.418570	12.420353	0.001784	12.399841	0.018728
4	(1,1)	14.532483	14.570464	0.037981	14.569258	0.036774
5	(0,3)	14.906711	14.908175	0.001463	14.926681	0.019969
6	(2,0)	14.923897	15.096327	0.172431	15.178302	0.254405
7	(0,4)	16.672353	16.675212	0.002859	16.730740	0.058387
8	(0,5)	17.715495	17.718550	0.003055	17.554757	0.160738
9	(1,2)	17.743125	17.753434	0.010309	17.813765	0.070640
10	(0,6)	18.036136	18.101351	0.065214	18.087852	0.051716
11	(3,0)	18.248452	18.557837	0.309385	18.963848	0.715396
12	(2,1)	18.857038	19.084008	0.226970	19.279707	0.422668
13	(1,3)	20.231267	20.232044	0.000777	20.238681	0.007415
14	(4,0)	20.573007	20.982602	0.409595	21.229761	0.656754
15	(5,0)	21.897563	21.985464	0.087902	22.098947	0.201385
16	(1,4)	21.996908	22.005403	0.008495	22.183256	0.186348
17	(2,2)	22.067680	22.077941	0.010261	22.302560	0.234880
18	(3,1)	22.181594	22.283863	0.102269	22.867896	0.686303
19	(6,0)	22.222118	23.034840	0.812722	23.088500	0.866382
20	(1,5)	23.040050	23.371134	0.331084	23.638025	0.597975
21	(1,6)	23.360692	23.444430	0.083739	24.652465	1.291773
22	(4,1)	24.506149	24.555468	0.049319	25.255355	0.749206
23	(2,3)	24.555822	25.108759	0.552937	26.125583	1.569761
24	(3,2)	25.392235	25.552862	0.160627	26.168808	0.776573

Several trends are worth separating. For the lowest states, where the exact Morse functions are compact, beam errors are already very small: 7.1×10^{-3} for the ground state, 2.6×10^{-3} for $(0, 1)$, and 1.8×10^{-3} for $(0, 2)$. The main residual errors occur for states with greater excitation along x, including $(3, 0)$, $(4, 0)$, and the continuum-like $(6, 0)$ entry. This pattern indicates that point selection is not the only source of error: the finite coordinate range and the contracted representation of diffuse high-energy functions also matter.

13.4 Ground-state wavefunction

The analytic ground state is separable, so it provides a direct wavefunction-level diagnostic in addition to the energy comparison:

Method	Ground-state energy error	Overlap	L^2 error
Beam search	+0.007116	0.999989	0.004671
Pivoted QR	-0.027452	0.999853	0.017118

The beam-selected ground state reduces the L^2 error by about 72.7% and improves the absolute energy error by nearly a factor of four. Both overlaps are high, so the difference is not a qualitative change of state identity; it is a reduction of a small but spatially coherent projection error.

13.5 Representative excited-state amplitudes

The notebook reconstructs a sequence of beam-search eigenstates and compares them with the corresponding analytic product functions. The lowest state is nearly indistinguishable from the

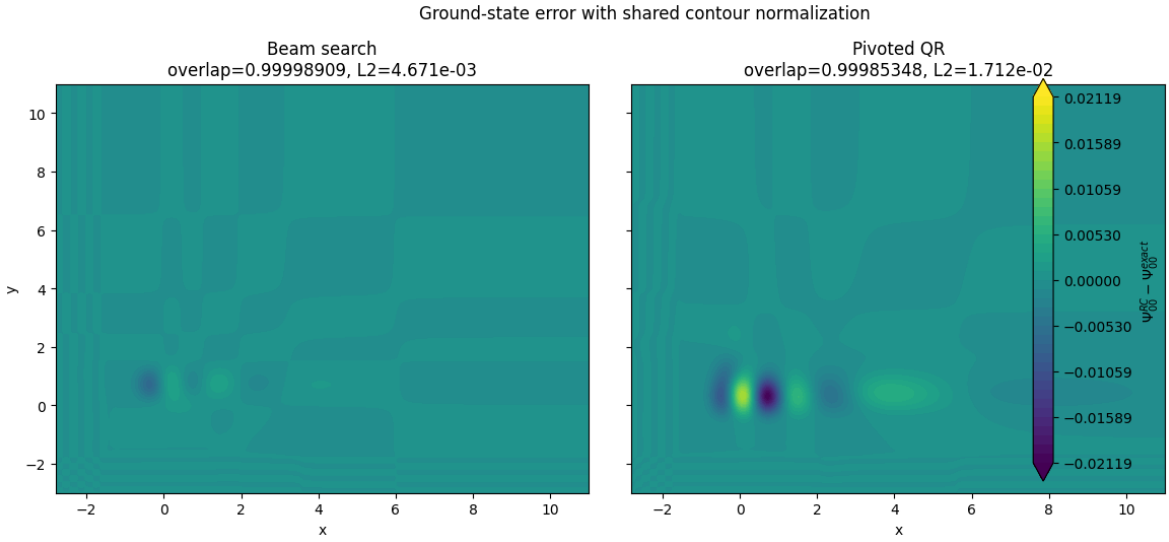

Figure 6: Numerical-minus-analytic ground-state amplitude for beam search and pivoted QR using one shared contour normalization. The beam-selected residual is visibly smaller, consistent with the overlap and L^2 diagnostics.

exact result, while higher excitations make finite-domain and contraction errors increasingly visible. Representative examples are shown in Figs. 7–9.

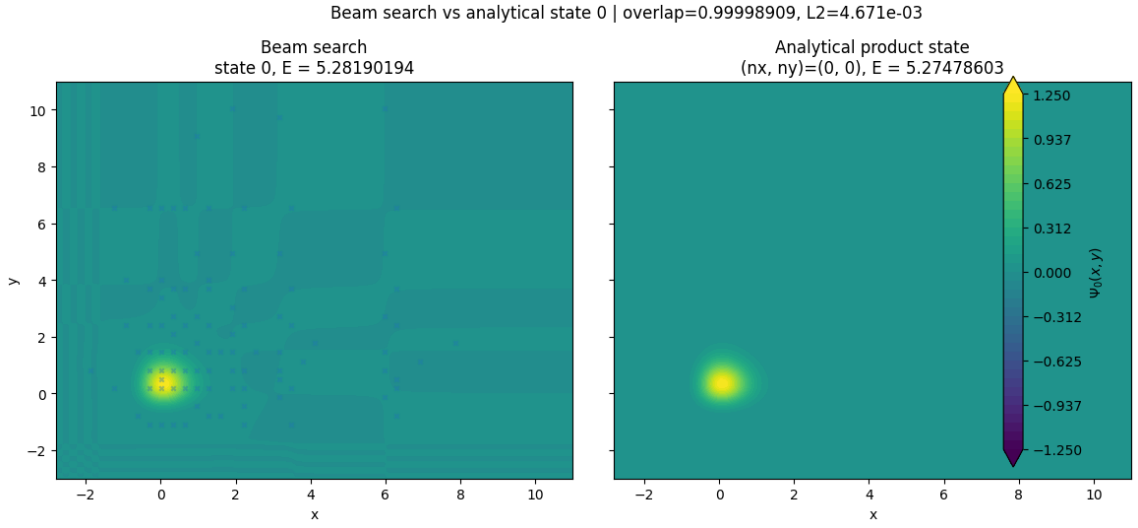

Figure 7: Beam-search ground-state amplitude compared with the analytic product state.

14 Extension to a coupled Morse potential

The notebook finally replaces the separable potential by

$$V(x, y) = V_x(x) + V_y(y) + kxy, \qquad k = 1.5. \tag{127}$$

The product-contracted basis, candidate grid, Christoffel weights, and selected point sets are left unchanged; only the pointwise potential values entering $\boldsymbol{H}_{\mathrm{col}}$ are replaced. This illustrates an important practical advantage of collocation: a new potential does not require re-evaluating multidimensional potential-energy matrix elements.

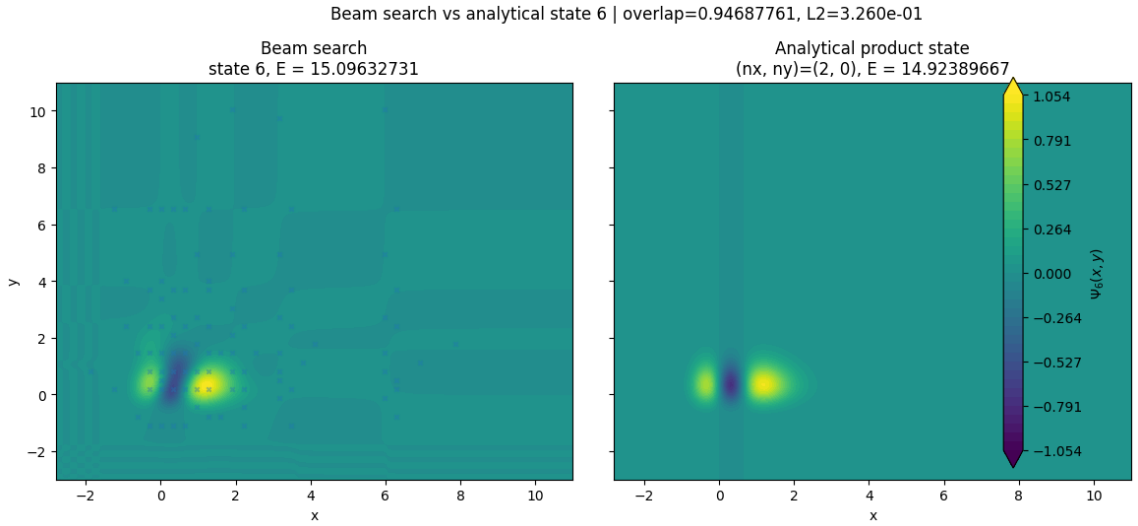

Figure 8: Representative intermediate excitation. The nodal topology is reproduced while the quantitative error is larger than for the lowest states.

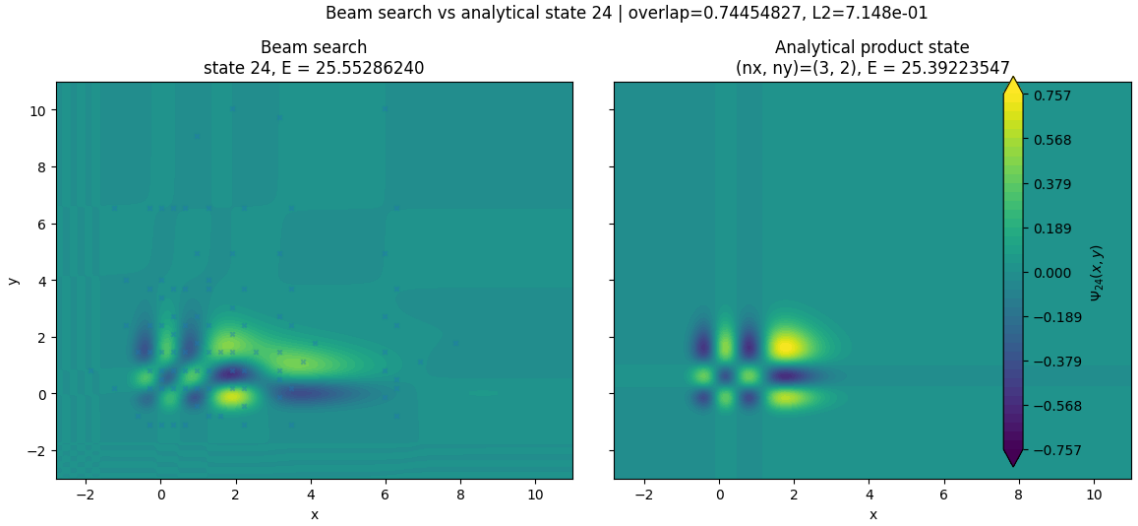

Figure 9: High member of the 25-state comparison. This state probes the most diffuse part of the retained contracted space and therefore provides a more stringent test of the finite coordinate domain and point set.

14.1 Beam-search and QR spectra for the coupled potential

The first 20 energies are

State	Beam search	Pivoted QR	Beam − QR
0	5.356595	5.302898	+0.053697
1	9.317071	9.297579	+0.019491
2	10.942794	10.649002	+0.293792
3	12.526730	12.506986	+0.019744
4	14.960463	15.045239	-0.084777
5	15.142845	15.091245	+0.051600
6	15.544182	15.732982	-0.188801
7	16.824885	16.867300	-0.042415
8	17.612018	17.003340	+0.608678
9	17.998712	18.271011	-0.272299
10	18.282297	18.664767	-0.382470
11	19.295802	19.898433	-0.602631
12	19.964470	20.376907	-0.412437
13	20.815714	21.050393	-0.234680
14	21.843146	22.823444	-0.980299
15	22.655139	23.185144	-0.530005
16	23.037749	24.016649	-0.978901
17	23.307918	24.234107	-0.926189
18	23.880660	24.528591	-0.647930
19	23.997719	25.686399	-1.688680

The same point-set conditioning carries over to the coupled calculation: $\kappa(\boldsymbol{B}^{\mathsf{T}}\boldsymbol{W}\boldsymbol{B}) = 2.126551$ for beam search and 54.143509 for pivoted QR. The two spectra remain close at low energy but separate more strongly for higher states, reaching a difference of about 1.69 for state 19.

Unlike the separable case, there is no analytic reference spectrum in this notebook for the coupled Hamiltonian. Consequently, Fig. 10 *cannot by itself establish that the beam energies are more accurate.* Its role is different: it shows that a better-conditioned point set reduces one source of numerical fragility, while the remaining beam–QR differences provide an empirical measure of point-selection sensitivity. A definitive accuracy test for the coupled problem would require an independent converged reference, for example a dense DVR, a direct contracted-basis Hamiltonian, or systematic convergence with N_p and N_{PC}.

14.2 How the coupling changes the beam-search eigenstates

The notebook also matches coupled and uncoupled beam-search states by maximizing the total absolute overlap with a one-to-one Hungarian assignment. The first 20 assignments are

j_k	j_0	E_k	E_0	$E_k - E_0$	$\lvert\langle\Psi_k\rvert\Psi_0\rangle\rvert$	L^2
0	0	5.356595	5.281902	+0.0747	0.9994	0.0359
1	1	9.317071	9.210533	+0.1065	0.9948	0.1020
2	2	10.942794	10.680483	+0.2623	0.9934	0.1152
3	3	12.526730	12.420353	+0.1064	0.9904	0.1385
4	5	14.960463	14.908175	+0.0523	0.7761	0.6691
5	4	15.142845	14.570464	+0.5724	0.7254	0.7411
6	6	15.544182	15.096327	+0.4479	0.9632	0.2714

j_k	j_0	E_k	E_0	$E_k - E_0$	$\lvert\langle\Psi_k\lvert\Psi_0\rangle\rvert$	L^2
7	7	16.824885	16.675212	+0.1497	0.9605	0.2811
8	8	17.612018	17.718550	-0.1065	0.7956	0.6394
9	10	17.998712	18.101351	-0.1026	0.7086	0.7634
10	9	18.282297	17.753434	+0.5289	0.8393	0.5668
11	11	19.295802	18.557837	+0.7380	0.9232	0.3919
12	12	19.964470	19.084008	+0.8805	0.9283	0.3787
13	13	20.815714	20.232044	+0.5837	0.9010	0.4450
14	14	21.843146	20.982602	+0.8605	0.8681	0.5137
15	17	22.655139	22.077941	+0.5772	0.6842	0.7948
16	15	23.037749	21.985464	+1.0523	0.7421	0.7181
17	16	23.307918	22.005403	+1.3025	0.6931	0.7835
18	18	23.880660	22.283863	+1.5968	0.8014	0.6302
19	19	23.997719	23.034840	+0.9629	0.7311	0.7334

The lowest four states remain adiabatically recognizable: their overlaps are 0.990–0.999 and their energy shifts are only 0.075–0.262. Stronger mixing appears near states 4 and 5, which map onto uncoupled states 5 and 4 rather than preserving the energy ordering. Their overlaps fall to 0.776 and 0.725. At still higher energy, several overlaps lie between roughly 0.68 and 0.93, showing that the kxy term produces substantial state mixing rather than a uniform perturbative energy shift.

Representative state maps in the accompanying bundle show the same trend: the ground state changes only weakly, whereas intermediate and high states can reorganize their nodal structure through mixing of nearby uncoupled product states.

15 Interpretation and convergence priorities

The comparison isolates a useful distinction between *greedy local independence* and *global set quality*. Pivoted QR is excellent when a fast, deterministic ordering is required. Beam search is more expensive, but in this example its ability to preserve alternative partial sets yields a larger D-optimal score, a nearly orthogonal weighted sampling matrix, smaller aggregate separable-spectrum errors, and a more accurate ground-state wavefunction. These are mutually consistent diagnostics rather than four independent claims.

The results also identify the next numerical tests that matter most:

1. repeat the comparison for several beam widths and branch widths to separate a genuine beam-search advantage from a single hyperparameter choice;
2. vary N_p/N_{PC} and the candidate-grid density;
3. restore a strictly bound contracted benchmark ($N_{\mathrm{PC}}^{(x)} = N_{\mathrm{PC}}^{(y)} = 6$) when quoting exact Morse errors, or replace the high-state reference by a converged continuum-capable discretization;
4. use a direct weighted least-squares solve instead of normal equations;
5. compare with the direct Kronecker-sum Hamiltonian in the same contracted basis for the separable potential;
6. obtain an independent converged reference for the coupled potential.

These checks would determine how much of the observed improvement comes from point-set optimization itself and how much is limited by basis contraction, finite-domain effects, or the treatment of near-continuum states.

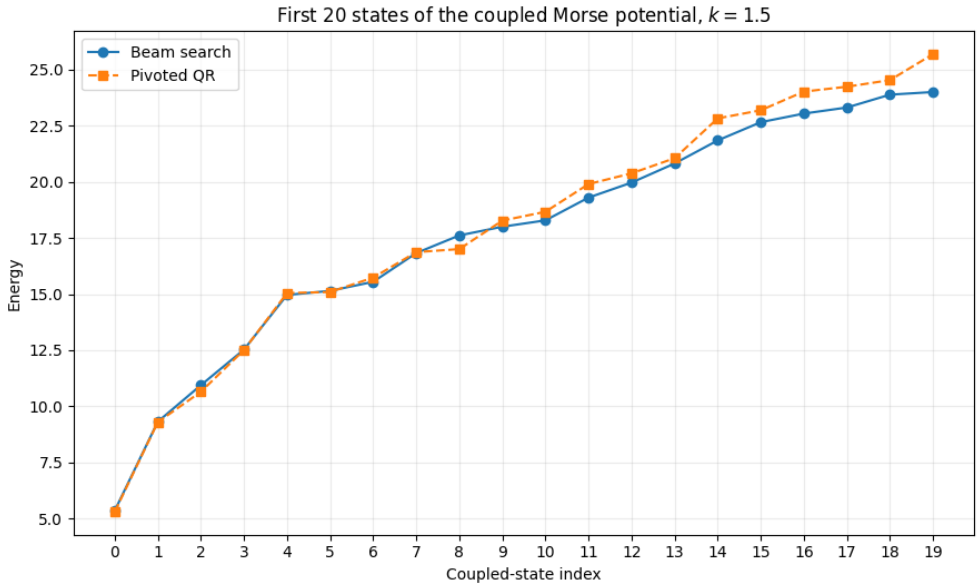

Figure 10: First 20 energies for the coupled Morse potential with $k = 1.5$ using the beam-search and pivoted-QR point sets. The plot quantifies the sensitivity of the nonseparable calculation to the collocation-point selection.

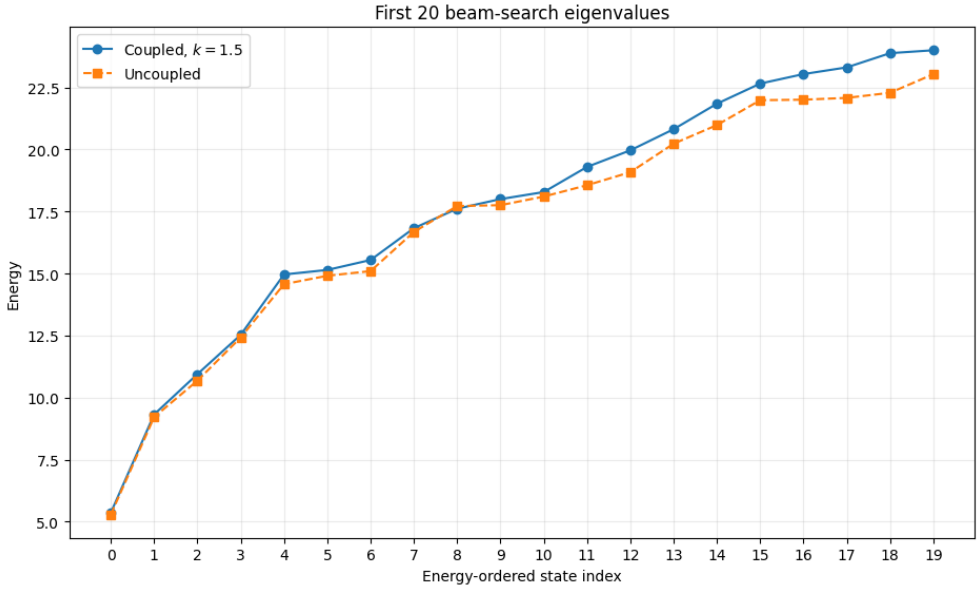

Figure 11: First 20 beam-search energies for the coupled and uncoupled potentials. The increasing separation at higher energy accompanies stronger wavefunction mixing in the overlap analysis.

16 Main lesson

Collocation replaces multidimensional potential-energy integration by pointwise evaluation of the differential equation. Product contraction reduces the basis dimension, Christoffel scaling balances the sampling geometry, and point selection determines how efficiently that reduced space is interrogated. Pivoted QR provides a fast greedy approximation to Fekete selection. The beam search studied here attacks the same weighted design problem with a broader combinatorial search. For the present two-dimensional Morse benchmark, that extra search produces a substantially better-conditioned sampling matrix and smaller separable-spectrum and ground-state errors. The coupled example shows how the same points and basis can be reused immediately for a new potential, while also emphasizing that conditioning alone is not a substitute for an independent accuracy benchmark.

A Complete Python implementation

The complete code exported from the supplied notebook is included as `code/beam_search_qr_comparison.py` and reproduced here for reference.

Listing 3: Complete beam-search versus pivoted-QR comparison code.

```
# Exported from beam_search_qr_comparison.ipynb

# %% [cell 1]
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from scipy.linalg import eig, qr
from scipy.special import eval_genlaguerre
from scipy.integrate import simpson

np.set_printoptions(precision=6, suppress=True)

# %% [cell 3]
HBAR = 1.0

MASS_X, D_X, A_X, XE_X = 1.0, 20.0, 1.0, 0.0
MASS_Y, D_Y, A_Y, XE_Y = 1.0, 15.0, 0.85, 0.25

# Primitive one-dimensional Gaussian bases
N_PRIM_X = 70
N_PRIM_Y = 70

# Primitive one-dimensional collocation grids
N_COL_X = 3 * N_PRIM_X
N_COL_Y = 3 * N_PRIM_Y

# Product-contracted basis
N_PC_X = 7
N_PC_Y = 7
N_PC_TOTAL = N_PC_X * N_PC_Y

# Candidate grid for Fekete-point selection
N_CAND_X = 45
N_CAND_Y = 45

# Oversampled Fekete-like point set
```

```
N_SELECTED = int(np.ceil(2.0 * N_PC_TOTAL))

# Point-selection method: "beam" or "qr"
POINT_SELECTION_METHOD = "beam"

# Beam-search controls
BEAM_WIDTH = 10
BEAM_BRANCH_WIDTH = 14
BEAM_RIDGE = 1.0e-8

# Coordinate ranges
X_MIN, X_MAX = -2.8, 11.0
Y_MIN, Y_MAX = -3.0, 11.0

# Gaussian width
GAUSSIAN_SHAPE = 0.35

# Pseudoinverse cutoff
PINV_RCOND = 1.0e-11

print("Primitive tensor-product size:", N_PRIM_X * N_PRIM_Y)
print("Product-contracted size:", N_PC_TOTAL)
print("Candidate points:", N_CAND_X * N_CAND_Y)
print("Selected points:", N_SELECTED)
print("Point-selection method:", POINT_SELECTION_METHOD)

# %% [cell 5]
def morse_potential(q, D, a, qe):
    return D * (1.0 - np.exp(-a * (q - qe)))**2

def morse_lambda(mass, D, a):
    return np.sqrt(2.0 * mass * D) / (HBAR * a)

def exact_morse_energy(n, mass, D, a):
    lam = morse_lambda(mass, D, a)
    return D - (HBAR**2 * a**2 / (2.0 * mass)) * (lam - n - 0.5)**2

def exact_morse_wavefunction(n, q, mass, D, a, qe):
    lam = morse_lambda(mass, D, a)
    z = 2.0 * lam * np.exp(-a * (q - qe))
    alpha = 2.0 * lam - 2.0 * n - 1.0
    psi = np.exp(-0.5 * z) * z**(lam - n - 0.5)
    psi *= eval_genlaguerre(n, alpha, z)
    return psi / np.sqrt(simpson(psi**2, x=q))

# %% [cell 7]
def gaussian_basis_and_kinetic(points, centers, beta, mass):
    d = points[:, None] - centers[None, :]
    B = np.exp(-beta * d**2)
    d2B = (4.0 * beta**2 * d**2 - 2.0 * beta) * B
    T = -(HBAR**2 / (2.0 * mass)) * d2B
    return B, T

def solve_1d(qmin, qmax, nprim, ncol, mass, D, a, qe):
    centers = np.linspace(qmin, qmax, nprim)
    points = np.linspace(qmin, qmax, ncol)
    beta = GAUSSIAN_SHAPE / (centers[1] - centers[0])**2
```

```
    B, T = gaussian_basis_and_kinetic(points, centers, beta, mass)
    Hcol = T + morse_potential(points, D, a, qe)[:, None] * B
    Aeff = np.linalg.pinv(B, rcond=PINV_RCOND) @ Hcol

    values, vectors = eig(Aeff)
    mask = np.abs(values.imag) < 1.0e-8
    values = values.real[mask]
    vectors = vectors[:, mask].real
    order = np.argsort(values)

    return {
        "energies": values[order],
        "coefficients": vectors[:, order],
        "centers": centers,
        "beta": beta,
        "points": points,
        "B": B,
        "T": T,
        "Hcol": Hcol,
    }

sol_x = solve_1d(X_MIN, X_MAX, N_PRIM_X, N_COL_X,
                 MASS_X, D_X, A_X, XE_X)
sol_y = solve_1d(Y_MIN, Y_MAX, N_PRIM_Y, N_COL_Y,
                 MASS_Y, D_Y, A_Y, XE_Y)

print("x energies:", sol_x["energies"][:N_PC_X])
print("y energies:", sol_y["energies"][:N_PC_Y])

# %% [cell 9]
def contracted_coefficients(solution, nkeep, mass, qmin, qmax):
    qdense = np.linspace(qmin, qmax, 3000)
    Bdense, _ = gaussian_basis_and_kinetic(
        qdense, solution["centers"], solution["beta"], mass
    )
    C = solution["coefficients"][:, :nkeep].copy()
    Phi = Bdense @ C

    S = np.empty((nkeep, nkeep))
    for i in range(nkeep):
        for j in range(nkeep):
            S[i, j] = simpson(Phi[:, i] * Phi[:, j], x=qdense)

    vals, vecs = np.linalg.eigh(S)
    Sinvhalf = vecs @ np.diag(vals**-0.5) @ vecs.T
    return C @ Sinvhalf

C_X = contracted_coefficients(sol_x, N_PC_X, MASS_X, X_MIN, X_MAX)
C_Y = contracted_coefficients(sol_y, N_PC_Y, MASS_Y, Y_MIN, Y_MAX)

# %% [cell 11]
def evaluate_contracted_1d(q, solution, C, mass):
    Bp, Tp = gaussian_basis_and_kinetic(
        np.asarray(q), solution["centers"], solution["beta"], mass
    )
    return Bp @ C, Tp @ C

def rowwise_product(Px, Py):
```

```
    return np.einsum("ia,ib->iab", Px, Py).reshape(
        Px.shape[0], Px.shape[1] * Py.shape[1]
    )

def rowwise_product_kinetic(Tx, Px, Ty, Py):
    return (
        np.einsum("ia,ib->iab", Tx, Py)
        + np.einsum("ia,ib->iab", Px, Ty)
    ).reshape(Px.shape[0], Px.shape[1] * Py.shape[1])

# %% [cell 13]
xc = np.linspace(X_MIN, X_MAX, N_CAND_X)
yc = np.linspace(Y_MIN, Y_MAX, N_CAND_Y)
XX, YY = np.meshgrid(xc, yc, indexing="ij")
candidate_xy = np.column_stack([XX.ravel(), YY.ravel()])

Px, Tx = evaluate_contracted_1d(candidate_xy[:, 0], sol_x, C_X, MASS_X)
Py, Ty = evaluate_contracted_1d(candidate_xy[:, 1], sol_y, C_Y, MASS_Y)

B_candidate = rowwise_product(Px, Py)
K_candidate = np.sum(B_candidate**2, axis=1)
w_candidate = 1.0 / K_candidate

print("K range:", K_candidate.min(), K_candidate.max())
print("w range:", w_candidate.min(), w_candidate.max())

# %% [cell 16]
from dataclasses import dataclass


@dataclass
class BeamState:
    """One partial point set in the beam."""

    selected: tuple[int, ...]
    information: np.ndarray
    score: float


def regularized_logdet_score(
    weighted_candidate_matrix,
    selected_indices,
    ridge=1.0e-8,
):
    """Return the regularized D-optimal score for one selected set."""
    X = np.asarray(weighted_candidate_matrix, dtype=float)
    indices = np.asarray(selected_indices, dtype=int)

    information = ridge * np.eye(X.shape[1])
    if indices.size:
        X_selected = X[indices]
        information += X_selected.T @ X_selected

    sign, logdet = np.linalg.slogdet(information)
    if sign <= 0:
        return -np.inf

    return logdet - X.shape[1] * np.log(ridge)
```

```python
def beam_select_points(
    weighted_candidate_matrix,
    n_selected,
    *,
    beam_width=12,
    branch_width=16,
    ridge=1.0e-8,
    tie_break_order=None,
    verbose=False,
):
    """
    Select candidate rows by regularized log-determinant beam search.

    Parameters
    ----------
    weighted_candidate_matrix : ndarray, shape (n_candidates, n_basis)
        Christoffel-weighted candidate basis matrix.
    n_selected : int
        Number of candidate points to retain.
    beam_width : int
        Number of partial point sets retained at each depth.
    branch_width : int
        Number of candidate additions explored from each beam state.
    ridge : float
        Positive regularization in delta I + X_S.T @ X_S.
    tie_break_order : array-like, optional
        Candidate ordering used only when marginal gains are tied. Passing
        the pivoted-QR ordering gives a reproducible hybrid initialization.
    verbose : bool
        Print progress while the beam is constructed.

    Returns
    -------
    selected_indices : ndarray
        Candidate indices in their beam-search selection order.
    best_score : float
        Final regularized log-determinant score.
    """
    X = np.asarray(weighted_candidate_matrix, dtype=float)

    if X.ndim != 2:
        raise ValueError("weighted_candidate_matrix must be two-dimensional.")
    if not np.all(np.isfinite(X)):
        raise ValueError("weighted_candidate_matrix contains non-finite values.")

    n_candidates, n_basis = X.shape

    if not 1 <= n_selected <= n_candidates:
        raise ValueError("n_selected must lie between 1 and n_candidates.")
    if beam_width < 1 or branch_width < 1:
        raise ValueError("beam_width and branch_width must be positive.")
    if ridge <= 0.0:
        raise ValueError("ridge must be positive.")

    if tie_break_order is None:
        tie_break_order = np.arange(n_candidates)
```

```
    else:
        tie_break_order = np.asarray(tie_break_order, dtype=int)
        if (
            tie_break_order.shape != (n_candidates,)
            or not np.array_equal(
                np.sort(tie_break_order), np.arange(n_candidates)
            )
        ):
            raise ValueError(
                "tie_break_order must be a permutation of all candidate indices."
            )

    tie_rank = np.empty(n_candidates, dtype=int)
    tie_rank[tie_break_order] = np.arange(n_candidates)

    base_logdet = n_basis * np.log(ridge)
    initial_information = ridge * np.eye(n_basis)
    beam = [BeamState((), initial_information, 0.0)]

    for depth in range(n_selected):
        # Different selection orders can lead to the same subset. Keying by
        # the sorted subset prevents those duplicates from occupying the beam.
        children = {}

        for state in beam:
            # For every candidate a_i, compute a_i^T G_S^{-1} a_i.
            solved = np.linalg.solve(state.information, X.T).T
            leverage = np.einsum("ij,ij->i", solved, X)
            leverage = np.maximum(leverage, 0.0)

            if state.selected:
                leverage[np.asarray(state.selected, dtype=int)] = -np.inf

            available = np.flatnonzero(np.isfinite(leverage))
            n_branches = min(branch_width, available.size)
            marginal_gains = np.log1p(leverage[available])

            # Primary key: decreasing marginal gain.
            # Secondary key: the supplied deterministic candidate ordering.
            local_order = np.lexsort(
                (tie_rank[available], -marginal_gains)
            )[:n_branches]

            for candidate in available[local_order]:
                candidate = int(candidate)
                basis_vector = X[candidate]
                new_information = (
                    state.information
                    + np.outer(basis_vector, basis_vector)
                )

                sign, logdet = np.linalg.slogdet(new_information)
                if sign <= 0:
                    continue

                new_selected = state.selected + (candidate,)
                new_score = logdet - base_logdet
                subset_key = tuple(sorted(new_selected))
```

```
                child = BeamState(
                    selected=new_selected,
                    information=new_information,
                    score=new_score,
                )
                previous = children.get(subset_key)

                if previous is None or child.score > previous.score:
                    children[subset_key] = child

        if not children:
            raise RuntimeError("Beam search produced no valid child states.")

        beam = sorted(
            children.values(),
            key=lambda state: (
                -state.score,
                tuple(tie_rank[i] for i in state.selected),
            ),
        )[:beam_width]

        if verbose and (
            depth == 0
            or (depth + 1) % 10 == 0
            or depth + 1 == n_selected
        ):
            print(
                f"Beam depth {depth + 1:3d}/{n_selected}: "
                f"best score = {beam[0].score:.8f}"
            )

    best_state = beam[0]
    return np.asarray(best_state.selected, dtype=int), best_state.score

# %% [cell 18]
Bw_candidate = np.sqrt(w_candidate)[:, None] * B_candidate

# Pivoted QR provides a deterministic baseline and a reproducible tie-break
# order for the beam search.
Q, R, pivots = qr(Bw_candidate.T, pivoting=True, mode="economic")
qr_indices = np.asarray(pivots[:N_SELECTED], dtype=int)
qr_score = regularized_logdet_score(
    Bw_candidate,
    qr_indices,
    ridge=BEAM_RIDGE,
)
qr_condition = np.linalg.cond(Bw_candidate[qr_indices])

# Beam search is evaluated regardless of which method is selected for the
# original downstream aliases, so every comparison is available in one run.
beam_indices, beam_score = beam_select_points(
    Bw_candidate,
    N_SELECTED,
    beam_width=BEAM_WIDTH,
    branch_width=BEAM_BRANCH_WIDTH,
    ridge=BEAM_RIDGE,
    tie_break_order=pivots,
```

```
    verbose=True,
)
beam_condition = np.linalg.cond(Bw_candidate[beam_indices])

selection_indices = {
    "Beam search": beam_indices,
    "Pivoted QR": qr_indices,
}
selection_scores = {
    "Beam search": beam_score,
    "Pivoted QR": qr_score,
}
selection_conditions = {
    "Beam search": beam_condition,
    "Pivoted QR": qr_condition,
}

method = POINT_SELECTION_METHOD.strip().lower()
if method == "beam":
    active_method_label = "Beam search"
elif method == "qr":
    active_method_label = "Pivoted QR"
else:
    raise ValueError(
        'POINT_SELECTION_METHOD must be either "beam" or "qr".'
    )

selected_indices = selection_indices[active_method_label].copy()
selection_score = selection_scores[active_method_label]
selection_label = f"{active_method_label} Fekete-like points"
selected_xy = candidate_xy[selected_indices]
selected_weights = w_candidate[selected_indices]
selected_K = K_candidate[selected_indices]
selected_condition = selection_conditions[active_method_label]

intersection_size = np.intersect1d(beam_indices, qr_indices).size
union_size = np.union1d(beam_indices, qr_indices).size

selection_summary = pd.DataFrame({
    "method": ["Beam search", "Pivoted QR"],
    "logdet_score": [beam_score, qr_score],
    "condition_number": [beam_condition, qr_condition],
    "shared_points": [intersection_size, intersection_size],
    "fraction_shared": [
        intersection_size / N_SELECTED,
        intersection_size / N_SELECTED,
    ],
    "Jaccard_index": [
        intersection_size / union_size,
        intersection_size / union_size,
    ],
})

display(selection_summary)

selection_point_tables = []
for label, indices in selection_indices.items():
    xy = candidate_xy[indices]
```

```
    selection_point_tables.append(pd.DataFrame({
        "method": label,
        "point": np.arange(N_SELECTED),
        "candidate_index": indices,
        "x": xy[:, 0],
        "y": xy[:, 1],
        "Christoffel_K": K_candidate[indices],
        "Christoffel_weight": w_candidate[indices],
    }))
selection_points_comparison = pd.concat(
    selection_point_tables,
    ignore_index=True,
)

fekete_table = selection_points_comparison[
    selection_points_comparison["method"] == active_method_label
].copy()
fekete_table["selection_method"] = method

display(fekete_table.head(15))
print(
    f"Active method: {active_method_label}; "
    f"condition number = {selected_condition:.8g}"
)
print(
    f"Beam/QR overlap: {intersection_size}/{N_SELECTED} points "
    f"({intersection_size / N_SELECTED:.1%})"
)

# %% [cell 20]
fig, axes = plt.subplots(
    1,
    2,
    figsize=(13.5, 5.6),
    sharex=True,
    sharey=True,
)

for ax, label in zip(axes, ["Beam search", "Pivoted QR"]):
    indices = selection_indices[label]
    xy = candidate_xy[indices]
    ax.scatter(
        candidate_xy[:, 0],
        candidate_xy[:, 1],
        s=9,
        alpha=0.20,
        label="Candidate grid",
    )
    ax.scatter(
        xy[:, 0],
        xy[:, 1],
        s=48,
        marker="x",
        label=f"{label} points",
    )
    ax.set_xlabel("x")
    ax.set_title(
        f"{label}\n"
```

```
        f"logdet={selection_scores[label]:.3f}, "
        f"cond={selection_conditions[label]:.3f}"
    )
    ax.grid(alpha=0.20)
    ax.legend(loc="upper right")

axes[0].set_ylabel("y")
fig.suptitle(
    f"Fekete-like point selections: {intersection_size}/{N_SELECTED} "
    "points shared",
    y=1.02,
)
fig.tight_layout()
fig.savefig("beam_vs_qr_fekete_points.png", dpi=220, bbox_inches="tight")
plt.show()

# %% [cell 21]
weight_grid = w_candidate.reshape(N_CAND_X, N_CAND_Y)
log_weight_grid = np.log10(weight_grid)
shared_levels = np.linspace(
    np.nanmin(log_weight_grid),
    np.nanmax(log_weight_grid),
    31,
)

fig, axes = plt.subplots(
    1,
    2,
    figsize=(13.5, 5.6),
    sharex=True,
    sharey=True,
)

for ax, label in zip(axes, ["Beam search", "Pivoted QR"]):
    indices = selection_indices[label]
    xy = candidate_xy[indices]
    cf = ax.contourf(
        XX,
        YY,
        log_weight_grid,
        levels=shared_levels,
    )
    ax.scatter(xy[:, 0], xy[:, 1], s=25, marker="x")
    ax.set_xlabel("x")
    ax.set_title(label)

axes[0].set_ylabel("y")
fig.colorbar(
    cf,
    ax=axes,
    label=r"$\log_{10} w(x,y)$",
    shrink=0.92,
    pad=0.03,
)
fig.suptitle("Selected points on the shared Christoffel-weight landscape")
fig.subplots_adjust(left=0.07, right=0.88, bottom=0.12, top=0.86, wspace=0.08)
fig.savefig(
    "beam_vs_qr_christoffel_weights.png",
```

```
    dpi=220,
    bbox_inches="tight",
)
plt.show()

# %% [cell 23]
def solve_weighted_collocation(point_indices):
    """Solve the rectangular collocation problem for one point set."""
    point_indices = np.asarray(point_indices, dtype=int)
    xy = candidate_xy[point_indices]
    weights = w_candidate[point_indices]

    xsel_local = xy[:, 0]
    ysel_local = xy[:, 1]

    Px_local, Tx_local = evaluate_contracted_1d(
        xsel_local, sol_x, C_X, MASS_X
    )
    Py_local, Ty_local = evaluate_contracted_1d(
        ysel_local, sol_y, C_Y, MASS_Y
    )

    Bsel_local = rowwise_product(Px_local, Py_local)
    Tsel_local = rowwise_product_kinetic(
        Tx_local, Px_local, Ty_local, Py_local
    )

    Vsel_local = (
        morse_potential(xsel_local, D_X, A_X, XE_X)
        + morse_potential(ysel_local, D_Y, A_Y, XE_Y)
    )
    Hsel_local = Tsel_local + Vsel_local[:, None] * Bsel_local

    # Avoid forming the dense diagonal W explicitly.
    gram = Bsel_local.T @ (weights[:, None] * Bsel_local)
    weighted_rhs = Bsel_local.T * weights[None, :]
    Mweighted_local = np.linalg.solve(gram, weighted_rhs)
    Apc_local = Mweighted_local @ Hsel_local

    energies, coefficients = eig(Apc_local)
    real_mask = np.abs(energies.imag) < 1.0e-8
    energies = energies.real[real_mask]
    coefficients = coefficients[:, real_mask].real
    order_local = np.argsort(energies)

    return {
        "indices": point_indices,
        "xy": xy,
        "weights": weights,
        "Bsel": Bsel_local,
        "Hsel": Hsel_local,
        "gram": gram,
        "gram_condition": np.linalg.cond(gram),
        "weighted_matrix_condition": np.linalg.cond(
            Bw_candidate[point_indices]
        ),
        "Apc": Apc_local,
        "energies": energies[order_local],
```

```
        "coefficients": coefficients[:, order_local],
    }


collocation_results = {
    label: solve_weighted_collocation(indices)
    for label, indices in selection_indices.items()
}

active_result = collocation_results[active_method_label]
xsel = active_result["xy"][:, 0]
ysel = active_result["xy"][:, 1]
Bsel = active_result["Bsel"]
Hsel = active_result["Hsel"]
Apc = active_result["Apc"]
E2d = active_result["energies"]
C2d = active_result["coefficients"]

collocation_condition_summary = pd.DataFrame({
    "method": ["Beam search", "Pivoted QR"],
    "weighted_matrix_condition": [
        collocation_results["Beam search"]["weighted_matrix_condition"],
        collocation_results["Pivoted QR"]["weighted_matrix_condition"],
    ],
    "normal_matrix_condition": [
        collocation_results["Beam search"]["gram_condition"],
        collocation_results["Pivoted QR"]["gram_condition"],
    ],
})

display(collocation_condition_summary)
for label in ["Beam search", "Pivoted QR"]:
    print(f"{label:12s} first 12 energies:")
    print(collocation_results[label]["energies"][:12])

# %% [cell 25]
exact_levels = []
for nx in range(N_PC_X):
    for ny in range(N_PC_Y):
        e = (
            exact_morse_energy(nx, MASS_X, D_X, A_X)
            + exact_morse_energy(ny, MASS_Y, D_Y, A_Y)
        )
        exact_levels.append((e, nx, ny))
exact_levels.sort()

ncompare = 25
rows = []
for label in ["Beam search", "Pivoted QR"]:
    energies = collocation_results[label]["energies"]
    for k in range(ncompare):
        e_exact, nx, ny = exact_levels[k]
        rows.append({
            "method": label,
            "state": k,
            "nx": nx,
            "ny": ny,
            "E_collocation": energies[k],
```

```
            "E_exact": e_exact,
            "signed_error": energies[k] - e_exact,
            "absolute_error": abs(energies[k] - e_exact),
        })

energy_comparison_all = pd.DataFrame(rows)
energy_comparison = energy_comparison_all[
    energy_comparison_all["method"] == active_method_label
].drop(columns="method").reset_index(drop=True)

energy_error_summary = (
    energy_comparison_all.groupby("method", sort=False)["absolute_error"]
    .agg(["mean", "max"])
    .rename(columns={"mean": "mean_absolute_error", "max": "max_absolute_error"})
    .reset_index()
)

display(energy_comparison_all)
display(energy_error_summary)

# %% [cell 26]
fig, axes = plt.subplots(
    1,
    2,
    figsize=(13.5, 5.2),
    sharex=True,
    sharey=True,
)

for ax, label in zip(axes, ["Beam search", "Pivoted QR"]):
    subset = energy_comparison_all[
        energy_comparison_all["method"] == label
    ]
    k = subset["state"]
    ax.plot(k, subset["E_collocation"], "o-", label=label)
    ax.plot(k, subset["E_exact"], "s--", label="Exact separable")
    ax.set_xlabel("Sorted state index")
    ax.set_title(label)
    ax.set_xticks(k)
    ax.grid(alpha=0.25)
    ax.legend()

axes[0].set_ylabel("Energy")
fig.suptitle("Two-dimensional Morse energies: beam search versus pivoted QR")
fig.tight_layout()
fig.savefig(
    "beam_vs_qr_energy_spectra.png",
    dpi=220,
    bbox_inches="tight",
)
plt.show()

fig, axes = plt.subplots(
    1,
    2,
    figsize=(13.5, 5.2),
    sharex=True,
    sharey=True,
```

```
)

positive_floor = np.finfo(float).tiny
for ax, label in zip(axes, ["Beam search", "Pivoted QR"]):
    subset = energy_comparison_all[
        energy_comparison_all["method"] == label
    ]
    errors = np.maximum(subset["absolute_error"].to_numpy(), positive_floor)
    ax.semilogy(subset["state"], errors, "o-")
    ax.set_xlabel("Sorted state index")
    ax.set_title(
        f"{label}\n"
        f"mean={errors.mean():.3e}, max={errors.max():.3e}"
    )
    ax.set_xticks(subset["state"])
    ax.grid(alpha=0.25)

axes[0].set_ylabel("Absolute energy error")
fig.suptitle("Energy-error comparison on a shared logarithmic scale")
fig.tight_layout()
fig.savefig(
    "beam_vs_qr_energy_errors.png",
    dpi=220,
    bbox_inches="tight",
)
plt.show()

# %% [cell 28]
def evaluate_state_grid(coefficients, xgrid, ygrid):
    Px, _ = evaluate_contracted_1d(xgrid, sol_x, C_X, MASS_X)
    Py, _ = evaluate_contracted_1d(ygrid, sol_y, C_Y, MASS_Y)
    Cmat = coefficients.reshape(N_PC_X, N_PC_Y)
    return Px @ Cmat @ Py.T


def exact_product_state(nx, ny, xgrid, ygrid):
    psi_x = exact_morse_wavefunction(nx, xgrid, MASS_X, D_X, A_X, XE_X)
    psi_y = exact_morse_wavefunction(ny, ygrid, MASS_Y, D_Y, A_Y, XE_Y)
    return np.outer(psi_x, psi_y)


def normalize_on_grid(psi, xgrid, ygrid):
    norm = simpson(simpson(np.abs(psi)**2, x=ygrid, axis=1), x=xgrid)
    return psi / np.sqrt(norm)


# Plotting grid
xplot = np.linspace(X_MIN, X_MAX, 180)
yplot = np.linspace(Y_MIN, Y_MAX, 180)
XXp, YYp = np.meshgrid(xplot, yplot, indexing="ij")

# Use only the beam-search result
beam = collocation_results["Beam search"]
beam_coeffs = beam["coefficients"]
beam_energies = beam["energies"]
beam_xy = beam["xy"]

# Choose which states to plot
```

```
STATES_TO_PLOT = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20, 24]

for state in STATES_TO_PLOT:
    if state >= len(beam_energies):
        print(f"Skipping state {state}: only {len(beam_energies)} states are available.
            ")
        continue

    # Numerical beam-search state
    psi_num = evaluate_state_grid(beam_coeffs[:, state], xplot, yplot)
    psi_num = normalize_on_grid(psi_num, xplot, yplot)

    # Matching analytical state from the sorted exact spectrum
    e_exact, nx, ny = exact_levels[state]
    psi_exact = exact_product_state(nx, ny, xplot, yplot)
    psi_exact = normalize_on_grid(psi_exact, xplot, yplot)

    # Fix the arbitrary global sign of the numerical eigenvector
    overlap = simpson(
        simpson(psi_num * psi_exact, x=yplot, axis=1),
        x=xplot,
    )
    if overlap < 0:
        psi_num *= -1
        overlap *= -1

    l2_error = np.sqrt(
        simpson(
            simpson((psi_num - psi_exact)**2, x=yplot, axis=1),
            x=xplot,
        )
    )

    # Shared contour scale for fair visual comparison
    max_amp = max(np.max(np.abs(psi_num)), np.max(np.abs(psi_exact)))
    levels = np.linspace(-max_amp, max_amp, 41)

    fig, axes = plt.subplots(
        1, 2,
        figsize=(13.5, 5.6),
        sharex=True,
        sharey=True,
    )

    # Left: beam-search state
    cf = axes[0].contourf(
        XXp, YYp, psi_num,
        levels=levels,
        extend="both",
    )
    axes[0].scatter(
        beam_xy[:, 0], beam_xy[:, 1],
        s=10, marker="x", alpha=0.4,
    )
    axes[0].set_title(
        f"Beam search\nstate {state}, E = {beam_energies[state]:.8f}"
    )
    axes[0].set_xlabel("x")
```

```
    axes[0].set_ylabel("y")

    # Right: analytical product state
    axes[1].contourf(
        XXp, YYp, psi_exact,
        levels=levels,
        extend="both",
    )
    axes[1].set_title(
        f"Analytical product state\n(nx, ny)=({nx}, {ny}), E = {e_exact:.8f}"
    )
    axes[1].set_xlabel("x")

    fig.colorbar(
        cf,
        ax=axes,
        label=rf"$\Psi_{{{state}}}(x,y)$",
        shrink=0.92,
        pad=0.03,
    )

    fig.suptitle(
        f"Beam search vs analytical state {state} | overlap={overlap:.8f}, L2={l2_error
            :.3e}"
    )
    fig.subplots_adjust(left=0.07, right=0.88, bottom=0.12, top=0.84, wspace=0.08)

    plt.savefig(
        f"beam_vs_exact_eigenstate_{state}.png",
        dpi=220,
        bbox_inches="tight",
    )
    plt.show()

# %% [cell 30]
psi_x0 = exact_morse_wavefunction(0, xplot, MASS_X, D_X, A_X, XE_X)
psi_y0 = exact_morse_wavefunction(0, yplot, MASS_Y, D_Y, A_Y, XE_Y)
psi_exact = np.outer(psi_x0, psi_y0)

ground_state_results = {}
for label in ["Beam search", "Pivoted QR"]:
    coefficients = collocation_results[label]["coefficients"][:, 0]
    psi = evaluate_state_grid(coefficients, xplot, yplot)
    psi /= np.sqrt(
        simpson(simpson(psi**2, x=yplot, axis=1), x=xplot)
    )

    overlap_local = simpson(
        simpson(psi * psi_exact, x=yplot, axis=1),
        x=xplot,
    )
    if overlap_local < 0:
        psi *= -1
        overlap_local *= -1

    l2_local = np.sqrt(
        simpson(
            simpson((psi - psi_exact)**2, x=yplot, axis=1),
```

```
            x=xplot,
        )
    )

    ground_state_results[label] = {
        "psi": psi,
        "difference": psi - psi_exact,
        "overlap": overlap_local,
        "l2_error": l2_local,
    }

# Preserve the original active-method aliases.
psi_num = ground_state_results[active_method_label]["psi"]
overlap = ground_state_results[active_method_label]["overlap"]
l2_error = ground_state_results[active_method_label]["l2_error"]

ground_state_comparison = pd.DataFrame({
    "method": ["Beam search", "Pivoted QR"],
    "overlap": [
        ground_state_results["Beam search"]["overlap"],
        ground_state_results["Pivoted QR"]["overlap"],
    ],
    "L2_error": [
        ground_state_results["Beam search"]["l2_error"],
        ground_state_results["Pivoted QR"]["l2_error"],
    ],
})

display(ground_state_comparison)

# %% [cell 31]
differences = [
    ground_state_results[label]["difference"]
    for label in ["Beam search", "Pivoted QR"]
]
max_abs_difference = max(np.max(np.abs(diff)) for diff in differences)
shared_difference_levels = np.linspace(
    -max_abs_difference,
    max_abs_difference,
    41,
)

fig, axes = plt.subplots(
    1,
    2,
    figsize=(13.5, 5.6),
    sharex=True,
    sharey=True,
)

for ax, label, difference in zip(
    axes,
    ["Beam search", "Pivoted QR"],
    differences,
):
    cf = ax.contourf(
        XXp,
        YYp,
```

```
        difference,
        levels=shared_difference_levels,
        extend="both",
    )
    result = ground_state_results[label]
    ax.set_xlabel("x")
    ax.set_title(
        f"{label}\n"
        f"overlap={result['overlap']:.8f}, "
        f"L2={result['l2_error']:.3e}"
    )

axes[0].set_ylabel("y")
fig.colorbar(
    cf,
    ax=axes,
    label=r"$\Psi_{00}^{RC}-\Psi_{00}^{exact}$",
    shrink=0.92,
    pad=0.03,
)
fig.suptitle("Ground-state error with shared contour normalization")
fig.subplots_adjust(left=0.07, right=0.88, bottom=0.12, top=0.84, wspace=0.08)
fig.savefig(
    "beam_vs_qr_ground_state_difference.png",
    dpi=220,
    bbox_inches="tight",
)
plt.show()

# %% [cell 33]
# Active-method outputs retained for backward compatibility.
fekete_table.to_csv(
    "fekete_points_and_christoffel_weights.csv",
    index=False,
)
energy_comparison.to_csv(
    "morse_2d_energy_comparison.csv",
    index=False,
)

# New direct-comparison outputs.
selection_summary.to_csv(
    "beam_vs_qr_selection_summary.csv",
    index=False,
)
selection_points_comparison.to_csv(
    "beam_vs_qr_selected_points.csv",
    index=False,
)
collocation_condition_summary.to_csv(
    "beam_vs_qr_condition_summary.csv",
    index=False,
)
energy_comparison_all.to_csv(
    "beam_vs_qr_energy_comparison.csv",
    index=False,
)
ground_state_comparison.to_csv(
```

```
    "beam_vs_qr_ground_state_comparison.csv",
    index=False,
)

try:
    from google.colab import files
except ImportError:
    files = None

# In Colab, uncomment any desired download:
# files.download("beam_vs_qr_selection_summary.csv")
# files.download("beam_vs_qr_selected_points.csv")
# files.download("beam_vs_qr_condition_summary.csv")
# files.download("beam_vs_qr_energy_comparison.csv")
# files.download("beam_vs_qr_ground_state_comparison.csv")
# files.download("beam_vs_qr_fekete_points.png")
# files.download("beam_vs_qr_christoffel_weights.png")
# files.download("beam_vs_qr_energy_spectra.png")
# files.download("beam_vs_qr_energy_errors.png")
# files.download("beam_vs_qr_ground_state_difference.png")

# %% [cell 35]
# ================================================================
# First 20 eigenstates of the coupled 2D Morse potential
#
# V(x,y) = Vx(x) + Vy(y) + K_COUPLING * x * y
# ================================================================

K_COUPLING = 1.5 # Coupling strength; modify as needed
N_STATES_COUPLED = 20

if N_PC_TOTAL < N_STATES_COUPLED:
    raise ValueError(
        f"The product-contracted basis contains only {N_PC_TOTAL} functions. "
        f"Increase N_PC_X and/or N_PC_Y to compute "
        f"{N_STATES_COUPLED} states."
    )


def coupled_morse_potential(x, y, coupling):
    """
    Coupled two-dimensional Morse potential:

        V(x,y) = Vx(x) + Vy(y) + coupling*x*y
    """
    vx = morse_potential(x, D_X, A_X, XE_X)
    vy = morse_potential(y, D_Y, A_Y, XE_Y)

    return vx + vy + coupling * x * y


def solve_coupled_weighted_collocation(
    point_indices,
    coupling=K_COUPLING,
    n_states=N_STATES_COUPLED,
):
    """
    Construct and diagonalize the weighted rectangular-collocation
```

```
    Hamiltonian for the coupled Morse potential.
    """
    point_indices = np.asarray(point_indices, dtype=int)

    # Selected Fekete-like points and Christoffel weights
    xy = candidate_xy[point_indices]
    weights = w_candidate[point_indices]

    xsel_local = xy[:, 0]
    ysel_local = xy[:, 1]

    # Evaluate the same product-contracted basis used previously
    Px_local, Tx_local = evaluate_contracted_1d(
        xsel_local,
        sol_x,
        C_X,
        MASS_X,
    )

    Py_local, Ty_local = evaluate_contracted_1d(
        ysel_local,
        sol_y,
        C_Y,
        MASS_Y,
    )

    # Product-contracted basis matrix
    Bsel_local = rowwise_product(Px_local, Py_local)

    # Kinetic-energy action on the product basis
    Tsel_local = rowwise_product_kinetic(
        Tx_local,
        Px_local,
        Ty_local,
        Py_local,
    )

    # Full coupled potential evaluated at the selected points
    Vsel_local = coupled_morse_potential(
        xsel_local,
        ysel_local,
        coupling,
    )

    # Collocation representation of H Phi
    Hsel_local = (
        Tsel_local
        + Vsel_local[:, None] * Bsel_local
    )

    # Weighted rectangular pseudoinverse:
    #
    # M = (B^T W B)^(-1) B^T W
    #
    gram_local = (
        Bsel_local.T
        @ (weights[:, None] * Bsel_local)
    )
```

```
    weighted_rhs_local = (
        Bsel_local.T * weights[None, :]
    )

    Mweighted_local = np.linalg.solve(
        gram_local,
        weighted_rhs_local,
    )

    # Effective Hamiltonian in the product-contracted basis
    Apc_local = Mweighted_local @ Hsel_local

    # The collocation matrix is generally not exactly symmetric.
    eigenvalues, eigenvectors = eig(Apc_local)

    # Retain numerically real states
    real_mask = np.abs(eigenvalues.imag) < 1.0e-8

    eigenvalues = eigenvalues.real[real_mask]
    eigenvectors = eigenvectors[:, real_mask].real

    # Sort from lowest to highest energy
    order = np.argsort(eigenvalues)

    eigenvalues = eigenvalues[order]
    eigenvectors = eigenvectors[:, order]

    if len(eigenvalues) < n_states:
        raise RuntimeError(
            f"Only {len(eigenvalues)} numerically real eigenvalues "
            f"were found, but {n_states} were requested."
        )

    return {
        "coupling": coupling,
        "indices": point_indices,
        "xy": xy,
        "weights": weights,
        "potential": Vsel_local,
        "Bsel": Bsel_local,
        "Tsel": Tsel_local,
        "Hsel": Hsel_local,
        "gram": gram_local,
        "gram_condition": np.linalg.cond(gram_local),
        "Apc": Apc_local,
        "all_energies": eigenvalues,
        "all_coefficients": eigenvectors,
        "energies": eigenvalues[:n_states],
        "coefficients": eigenvectors[:, :n_states],
    }


# Solve with both existing point-selection methods
coupled_results = {
    label: solve_coupled_weighted_collocation(
        point_indices=indices,
        coupling=K_COUPLING,
```

```
        n_states=N_STATES_COUPLED,
    )
    for label, indices in selection_indices.items()
}


# Convenient aliases for the beam-search solution
coupled_beam_result = coupled_results["Beam search"]

E_coupled_beam = coupled_beam_result["energies"]
C_coupled_beam = coupled_beam_result["coefficients"]

xsel_coupled_beam = coupled_beam_result["xy"][:, 0]
ysel_coupled_beam = coupled_beam_result["xy"][:, 1]


# Convenient aliases for the pivoted-QR solution
coupled_qr_result = coupled_results["Pivoted QR"]

E_coupled_qr = coupled_qr_result["energies"]
C_coupled_qr = coupled_qr_result["coefficients"]


# Tabulate the first 25 energies
coupled_energy_table = pd.DataFrame({
    "state": np.arange(N_STATES_COUPLED),
    "beam_search_energy": E_coupled_beam,
    "pivoted_QR_energy": E_coupled_qr,
    "beam_minus_QR": E_coupled_beam - E_coupled_qr,
})

display(coupled_energy_table)

print(f"Coupling strength k = {K_COUPLING}")
print(
    "Beam-search normal-matrix condition number:",
    f"{coupled_beam_result['gram_condition']:.6e}",
)
print(
    "Pivoted-QR normal-matrix condition number:",
    f"{coupled_qr_result['gram_condition']:.6e}",
)


# Plot the first 25 coupled-state energies
fig, ax = plt.subplots(figsize=(9, 5.5))

state_indices = np.arange(N_STATES_COUPLED)

ax.plot(
    state_indices,
    E_coupled_beam,
    marker="o",
    label="Beam search",
)

ax.plot(
    state_indices,
```

```
    E_coupled_qr,
    marker="s",
    linestyle="--",
    label="Pivoted QR",
)

ax.set_xlabel("Coupled-state index")
ax.set_ylabel("Energy")
ax.set_title(
    rf"First {N_STATES_COUPLED} states of the coupled Morse potential, "
    rf"$k={K_COUPLING}$"
)
ax.set_xticks(state_indices)
ax.grid(alpha=0.25)
ax.legend()

plt.tight_layout()
plt.savefig(
    "coupled_morse_first_25_energies.png",
    dpi=220,
    bbox_inches="tight",
)
plt.show()

# %% [cell 36]
# ============================================================
# Coupled versus uncoupled beam-search eigenstates
#
# Coupled potential:
# V(x,y) = Vx(x) + Vy(y) + K_COUPLING*x*y
#
# Uncoupled potential:
# V0(x,y) = Vx(x) + Vy(y)
# ============================================================

from scipy.optimize import linear_sum_assignment

N_STATES_COMPARE = 25
PLOT_GRID_SIZE = 180


# ------------------------------------------------------------
# Numerical integration utilities
# ------------------------------------------------------------

def grid_inner_product(psi_a, psi_b, xgrid, ygrid):
    """
    Continuous inner product evaluated on the plotting grid:

        <psi_a|psi_b> = integral psi_a*(x,y) psi_b(x,y) dx dy
    """
    integrand = np.conjugate(psi_a) * psi_b

    return simpson(
        simpson(integrand, x=ygrid, axis=1),
        x=xgrid,
    )
```

```
def normalize_grid_state(psi, xgrid, ygrid):
    """Normalize a two-dimensional wavefunction on the plotting grid."""
    norm2 = np.real(
        grid_inner_product(
            psi,
            psi,
            xgrid,
            ygrid,
        )
    )

    if norm2 <= 0.0:
        raise ValueError(
            "The wavefunction has a nonpositive numerical norm."
        )

    return psi / np.sqrt(norm2)


def grid_l2_error(psi_a, psi_b, xgrid, ygrid):
    """Compute the L2 distance between two normalized states."""
    difference = psi_a - psi_b

    error2 = np.real(
        grid_inner_product(
            difference,
            difference,
            xgrid,
            ygrid,
        )
    )

    return np.sqrt(max(error2, 0.0))


# ------------------------------------------------------------
# Extract beam-search solutions
# ------------------------------------------------------------

uncoupled_beam = collocation_results["Beam search"]
coupled_beam = coupled_results["Beam search"]

E_uncoupled_all = np.asarray(
    uncoupled_beam["energies"],
    dtype=float,
)

C_uncoupled_all = np.asarray(
    uncoupled_beam["coefficients"],
)

E_coupled_all = np.asarray(
    coupled_beam["energies"],
    dtype=float,
)

C_coupled_all = np.asarray(
```

```
    coupled_beam["coefficients"],
)


n_states = min(
    N_STATES_COMPARE,
    len(E_uncoupled_all),
    len(E_coupled_all),
)

if n_states < N_STATES_COMPARE:
    print(
        f"Only {n_states} states are available for comparison; "
        f"{N_STATES_COMPARE} were requested."
    )

E_uncoupled = E_uncoupled_all[:n_states]
C_uncoupled = C_uncoupled_all[:, :n_states]

E_coupled = E_coupled_all[:n_states]
C_coupled = C_coupled_all[:, :n_states]


# ------------------------------------------------------------
# Evaluation grid
# ------------------------------------------------------------

xplot_compare = np.linspace(
    X_MIN,
    X_MAX,
    PLOT_GRID_SIZE,
)

yplot_compare = np.linspace(
    Y_MIN,
    Y_MAX,
    PLOT_GRID_SIZE,
)

XX_compare, YY_compare = np.meshgrid(
    xplot_compare,
    yplot_compare,
    indexing="ij",
)


# ------------------------------------------------------------
# Evaluate and normalize all states
# ------------------------------------------------------------

psi_uncoupled = []
psi_coupled = []

for state in range(n_states):

    psi0 = evaluate_state_grid(
        C_uncoupled[:, state],
        xplot_compare,
```

```
        yplot_compare,
    )

    psik = evaluate_state_grid(
        C_coupled[:, state],
        xplot_compare,
        yplot_compare,
    )

    psi0 = normalize_grid_state(
        psi0,
        xplot_compare,
        yplot_compare,
    )

    psik = normalize_grid_state(
        psik,
        xplot_compare,
        yplot_compare,
    )

    psi_uncoupled.append(psi0)
    psi_coupled.append(psik)

psi_uncoupled = np.asarray(psi_uncoupled)
psi_coupled = np.asarray(psi_coupled)


# ------------------------------------------------------------
# Construct the coupled/uncoupled overlap matrix
# ------------------------------------------------------------

overlap_matrix = np.zeros(
    (n_states, n_states),
    dtype=float,
)

for coupled_state in range(n_states):
    for uncoupled_state in range(n_states):

        overlap_value = grid_inner_product(
            psi_coupled[coupled_state],
            psi_uncoupled[uncoupled_state],
            xplot_compare,
            yplot_compare,
        )

        overlap_matrix[
            coupled_state,
            uncoupled_state
        ] = np.real(overlap_value)


# ------------------------------------------------------------
# One-to-one state assignment by maximum total overlap
#
# The Hungarian algorithm prevents several coupled states from
# being assigned to the same uncoupled state.
```

```
# ------------------------------------------------------------

coupled_indices, matched_uncoupled_indices = (
    linear_sum_assignment(-np.abs(overlap_matrix))
)

state_mapping = {
    int(coupled_state): int(uncoupled_state)
    for coupled_state, uncoupled_state in zip(
        coupled_indices,
        matched_uncoupled_indices,
    )
}


# ------------------------------------------------------------
# Align signs and construct the eigenvalue table
# ------------------------------------------------------------

comparison_rows = []
matched_state_data = {}

for coupled_state in range(n_states):

    uncoupled_state = state_mapping[coupled_state]

    psik = psi_coupled[coupled_state].copy()
    psi0 = psi_uncoupled[uncoupled_state].copy()

    signed_overlap = grid_inner_product(
        psik,
        psi0,
        xplot_compare,
        yplot_compare,
    )

    signed_overlap = float(np.real(signed_overlap))

    # Eigenvector signs are arbitrary.
    if signed_overlap < 0.0:
        psik *= -1.0
        signed_overlap *= -1.0

    l2_error = grid_l2_error(
        psik,
        psi0,
        xplot_compare,
        yplot_compare,
    )

    coupled_energy = E_coupled[coupled_state]
    uncoupled_energy = E_uncoupled[uncoupled_state]

    comparison_rows.append({
        "coupled_state": coupled_state,
        "matched_uncoupled_state": uncoupled_state,
        "E_coupled": coupled_energy,
        "E_uncoupled": uncoupled_energy,
```

```
        "delta_E": coupled_energy - uncoupled_energy,
        "absolute_overlap": signed_overlap,
        "L2_difference": l2_error,
    })

    matched_state_data[coupled_state] = {
        "uncoupled_state": uncoupled_state,
        "psi_coupled": psik,
        "psi_uncoupled": psi0,
        "overlap": signed_overlap,
        "l2_error": l2_error,
    }


coupled_uncoupled_table = pd.DataFrame(comparison_rows)

coupled_uncoupled_table = coupled_uncoupled_table.sort_values(
    "coupled_state"
).reset_index(drop=True)


# ------------------------------------------------------------
# Report the eigenvalues
# ------------------------------------------------------------

print(
    f"First {n_states} coupled states for "
    f"k = {K_COUPLING}"
)

display(
    coupled_uncoupled_table.style.format({
        "E_coupled": "{:.10f}",
        "E_uncoupled": "{:.10f}",
        "delta_E": "{:+.6e}",
        "absolute_overlap": "{:.8f}",
        "L2_difference": "{:.6e}",
    })
)

print("\nPlain-text eigenvalue table:\n")

print(
    coupled_uncoupled_table.to_string(
        index=False,
        formatters={
            "E_coupled": lambda value: f"{value:.10f}",
            "E_uncoupled": lambda value: f"{value:.10f}",
            "delta_E": lambda value: f"{value:+.6e}",
            "absolute_overlap": lambda value: f"{value:.8f}",
            "L2_difference": lambda value: f"{value:.6e}",
        },
    )
)


# Save the numerical results
coupled_uncoupled_table.to_csv(
```

```
    "coupled_vs_uncoupled_first_25_eigenvalues.csv",
    index=False,
)


# ------------------------------------------------------------
# Plot both energy spectra
# ------------------------------------------------------------

fig, ax = plt.subplots(figsize=(9.5, 5.8))

states = np.arange(n_states)

ax.plot(
    states,
    E_coupled,
    marker="o",
    label=rf"Coupled, $k={K_COUPLING}$",
)

ax.plot(
    states,
    E_uncoupled,
    marker="s",
    linestyle="--",
    label="Uncoupled",
)

ax.set_xlabel("Energy-ordered state index")
ax.set_ylabel("Energy")
ax.set_title(
    f"First {n_states} beam-search eigenvalues"
)
ax.set_xticks(states)
ax.grid(alpha=0.25)
ax.legend()

plt.tight_layout()

plt.savefig(
    "coupled_vs_uncoupled_energy_spectrum.png",
    dpi=220,
    bbox_inches="tight",
)

plt.show()


# ------------------------------------------------------------
# Plot each coupled state beside its matched uncoupled state
# ------------------------------------------------------------

for coupled_state in range(n_states):

    result = matched_state_data[coupled_state]

    uncoupled_state = result["uncoupled_state"]
    psik = result["psi_coupled"]
```

```
psi0 = result["psi_uncoupled"]

overlap_value = result["overlap"]
l2_error = result["l2_error"]

# Shared contour normalization
maximum_amplitude = max(
    np.max(np.abs(psik)),
    np.max(np.abs(psi0)),
)

if maximum_amplitude == 0.0:
    maximum_amplitude = 1.0

contour_levels = np.linspace(
    -maximum_amplitude,
    maximum_amplitude,
    41,
)

fig, axes = plt.subplots(
    1,
    2,
    figsize=(13.5, 5.7),
    sharex=True,
    sharey=True,
)

# Coupled state
contour = axes[0].contourf(
    XX_compare,
    YY_compare,
    psik,
    levels=contour_levels,
    extend="both",
)

axes[0].scatter(
    coupled_beam["xy"][:, 0],
    coupled_beam["xy"][:, 1],
    s=9,
    marker="x",
    alpha=0.35,
)

axes[0].set_title(
    "Coupled beam-search state\n"
    rf"$j={coupled_state}$, "
    rf"$E_j={E_coupled[coupled_state]:.8f}$"
)

axes[0].set_xlabel("x")
axes[0].set_ylabel("y")

# Matched uncoupled state
axes[1].contourf(
    XX_compare,
    YY_compare,
```

```
    psi0,
    levels=contour_levels,
    extend="both",
)

axes[1].scatter(
    uncoupled_beam["xy"][:, 0],
    uncoupled_beam["xy"][:, 1],
    s=9,
    marker="x",
    alpha=0.35,
)

axes[1].set_title(
    "Matched uncoupled state\n"
    rf"$j_0={uncoupled_state}$, "
    rf"$E_{{j_0}}={E_uncoupled[uncoupled_state]:.8f}$"
)

axes[1].set_xlabel("x")

fig.colorbar(
    contour,
    ax=axes,
    label=r"$\Psi(x,y)$",
    shrink=0.92,
    pad=0.03,
)

fig.suptitle(
    rf"Coupled state {coupled_state} versus uncoupled state "
    rf"{uncoupled_state}: "
    rf"$|\langle\Psi_k|\Psi_0\rangle|={overlap_value:.6f}$, "
    rf"$L^2={l2_error:.3e}$"
)

fig.subplots_adjust(
    left=0.07,
    right=0.88,
    bottom=0.12,
    top=0.82,
    wspace=0.08,
)

plt.savefig(
    f"coupled_vs_uncoupled_state_{coupled_state:02d}.png",
    dpi=220,
    bbox_inches="tight",
)

plt.show()
```

B Additional state-resolved figures

The bundle contains all state-resolved diagnostic figures produced by the notebook: beam-versus-exact separable states 0–10, 15, 20, and 24, and coupled-versus-uncoupled beam-search states

0–19. They are kept as individual files so that the main tutorial does not become dominated by repetitive full-page contour plots.