QCD Lattice

Generated by Doxygen 1.13.2

Chapter 1

Namespace Index

1.1 Namespace List

Here is a list of all namespaces with brief descriptions:

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

Chapter 3

Namespace Documentation

3.1 Lattice Namespace Reference

Field Theory.

3.1.1 Detailed Description

Field Theory.

Lattice field theory simulation for a scalar field in (d+1) dimensions.

Implements Metropolis Monte-Carlo updates for a real scalar field, computes correlation functions, extracts effective mass via logarithmic and cosh fits. Includes thermalization, autocorrelation spacing, and multi-run averaging.

3.2 Lattice Field Theory Namespace Reference

Functions

- potential_at_site (phi_val)
- neighbor_index (idx, mu, shift=1)
- action_total (field)
- local_action_contribution (idx, field)
- metropolis_update_field (field, eps=0.5)
- measure_field_correlation_all_origins (field)
- compute_effective_mass (G, G_err=None)
- run_field_simulation (field, N_sweeps=2000, N_cor=10, eps=0.5, thermal_sweeps=500)
- run_multiple_simulations (num_runs, N_sweeps, N_cor, eps, thermal_sweeps)
- cosh_model (t, A, m)

Variables

- int d = 1
- float a = 1.0
- int L = 32
- float m = 1.0
- float lambda_ = 0.1
- int D = d + 1
- phi = np.zeros((L,) * D)
- float eps_tuned = 0.5
- G_mean = 0.5 * (G_mean + G_mean[::-1])
- G_err
- m_eff
- num_runs
- N_sweeps
- N_cor
- eps
- thermal_sweeps
- tdata = np.arange(L)
- int mask = G_mean > 5 * G_err
- A_fit
- m_fit
- p0
- G_plot = np.abs(G_mean)
- yerr
- fmt
- capsize

3.2.1 Function Documentation

3.2.1.1 action_total()

Lattice Field Theory.action_total (
field)

Full lattice action including kinetic nearest-neighbor term.

Definition at line 55 of file Lattice Field Theory.py.

```
00055 def action_total(field):
00056     """
00057     Full lattice action including kinetic nearest-neighbor term.
00058     """
00059     S = 0.0
00060     for idx in np.ndindex(*field.shape):
00061         phi_site = field[idx]
00062         S += potential_at_site(phi_site)
00063         for mu in range(field.ndim):
00064             neigh = neighbor_index(idx, mu, shift=1)
00065             diff = field[neigh] - phi_site
00066             S += 0.5 * (diff**2) / (a**2)
00067     return S
00068
00069
```

Here is the call graph for this function:

3.3 Monte Namespace Reference

Carlo integral.

3.3.1 Detailed Description

Carlo integral.

@mainpage Monte Carlo Simulation for the 1D Quantum Anharmonic Oscillator
@file monte_carlo_anharmonic.py
@brief Monte Carlo Euclidean path-integral estimator for the ground-state energy E .

@details
This script estimates the ground-state energy of a 1D quantum anharmonic oscillator with Euclidean action

S_E = Σ_j [m/(2a) (x_{j+1}-x_j)² + a V(x_j)],

where
V(x) = 1/2 x² + x .

Configurations of x( ) are sampled using local Metropolis updates with periodic boundary conditions. The two-point correlator

C( ) = x(0)x( ) e^(−E),

is accumulated and fitted to a single exponential in the plateau region to extract E .

3.4 Monte Carlo integral Namespace Reference

Functions

- potential_energy_V (position, lambda_parameter)
- local_action (x_prev, x_current, x_next, lambda_parameter)
- delta_action_change (x_path, j, x_new, lambda_parameter)
- run_monte_carlo (lambda_parameter)

Variables

- float particle_mass = 1.0
- float lattice_spacing_a = 0.1
- int number_of_sites = 100
- int total_monte_carlo_steps = 30000
- int thermalization_steps = 5000
- float proposal_step_size = 0.5
- list anharmonicity_values = [0.0, 0.1, 0.3, 0.5, 1.0]
- list estimated_ground_energies = [ ]
- correlation_function
- acceptance_rate
- float euclidean_times = np.arange(len(correlation_function)) ∗ lattice_spacing_a
- fit_slice = slice(1, 6)
- slope
- intercept
- estimated_E0 = -slope
- lw

3.4.1 Function Documentation

3.4.1.1 delta_action_change()

Monte Carlo integral.delta_action_change (
x_path,
j,
x_new,
lambda_parameter)

@brief Compute local change ΔS_E for a proposed update at site j.
@param x_path ndarray(float): full current path configuration.
@param j int: lattice site index for update.
@param x_new float: proposed new value for x[j].
@param lambda_parameter float: anharmonicity .
@return float: ΔS = S_new − S_old.
@details
Only the action terms involving sites {j−1, j, j+1} contribute to ΔS.
Indices wrap via periodic boundary conditions.

Definition at line 96 of file Monte Carlo integral.py.

```
00096 def delta_action_change(x_path, j, x_new, lambda_parameter):
00097     """
00098     @brief Compute local change ΔS_E for a proposed update at site j.
00099     @param x_path ndarray(float): full current path configuration.
00100     @param j int: lattice site index for update.
00101     @param x_new float: proposed new value for x[j].
00102     @param lambda_parameter float: anharmonicity .
00103     @return float: ΔS = S_new − S_old.
00104     @details
00105     Only the action terms involving sites {j−1, j, j+1} contribute to ΔS.
00106     Indices wrap via periodic boundary conditions.
00107     """
00108     j_minus = (j - 1) % number_of_sites
00109     j_plus = (j + 1) % number_of_sites
00110
00111     S_old = local_action(x_path[j_minus], x_path[j], x_path[j_plus], lambda_parameter)
00112     S_new = local_action(x_path[j_minus], x_new, x_path[j_plus], lambda_parameter)
00113
00114     return S_new - S_old
00115
00116
00117 # ==============================================================
00118 #         Monte Carlo Simulation for Given
00119 # ==============================================================
00120
```

Here is the call graph for this function: Here is the caller graph for this function:

3.4.1.2 local_action()

Monte Carlo integral.local_action (
x_prev,
x_current,
x_next,
lambda_parameter)

@brief Local contribution to discretized Euclidean action around site j.
@param x_prev float: x at site j−1.
@param x_current float: x at site j.
@param x_next float: x at site j+1.
@param lambda_parameter float: anharmonicity .
@return float: local action S_E(j).
@details
Uses symmetric discretized kinetic term:
S_kin(j) = m/(4a)[(x_{j+1}-x_j)² + (x_j - x_{j-1})²]
plus potential:
S_pot(j) = a V(x_j).
Periodic boundary conditions handled externally.

Definition at line 75 of file Monte Carlo integral.py.

```
00075 def local_action(x_prev, x_current, x_next, lambda_parameter):
00076     """
00077     @brief Local contribution to discretized Euclidean action around site j.
00078     @param x_prev float: x at site j−1.
00079     @param x_current float: x at site j.
00080     @param x_next float: x at site j+1.
00081     @param lambda_parameter float: anharmonicity  .
00082     @return float: local action S_E(j).
00083     @details
00084     Uses symmetric discretized kinetic term:
00085         S_kin(j) = m/(4a)[(x_{j+1}-x_j)² + (x_j - x_{j-1})²]
00086     plus potential:
00087         S_pot(j) = a V(x_j).
00088     Periodic boundary conditions handled externally.
00089     """
00090     S_kinetic_local = 0.5 * particle_mass / lattice_spacing_a * \
00091                       ((x_next - x_current)**2 + (x_current - x_prev)**2) / 2
00092     S_potential_local = lattice_spacing_a * potential_energy_V(x_current, lambda_parameter)
00093     return S_kinetic_local + S_potential_local
00094
00095
```

Here is the call graph for this function: Here is the caller graph for this function:

3.4.1.3 potential_energy_V()

Monte Carlo integral.potential_energy_V (
position,
lambda_parameter)

@brief Anharmonic potential energy V(x).
@param position float or ndarray: spatial coordinate(s) x.
@param lambda_parameter float: anharmonicity (→ 0 gives harmonic limit).
@return float or ndarray: potential energy V(x).
@details
Implements:
V(x) = 1/2 x² + x .

Definition at line 58 of file Monte Carlo integral.py.

```
00058 def potential_energy_V(position, lambda_parameter):
00059     """
00060     @brief Anharmonic potential energy V(x).
00061     @param position float or ndarray: spatial coordinate(s) x.
00062     @param lambda_parameter float: anharmonicity  ( → 0 gives harmonic limit).
00063     @return float or ndarray: potential energy V(x).
00064     @details
00065     Implements:
00066         V(x) = 1/2 x² +  x .
00067     """
00068     return 0.5 * position**2 + lambda_parameter * position**4
00069
00070
00071 # ================================================================
00072 #         Local Euclidean Action Contributions
00073 # ================================================================
00074
```

Here is the caller graph for this function:

3.4.1.4 run_monte_carlo()

Monte Carlo integral.run_monte_carlo (
lambda_parameter)

@brief Perform local Metropolis updates to sample Euclidean paths for fixed .
@param lambda_parameter float: anharmonicity value for this simulation.
@return tuple: (C_tau, acceptance_fraction)
- C_tau: ndarray(float) correlator C( ) for up to T/2
- acceptance_fraction: overall acceptance rate of updates
@details
- Initializes x( )=0 path
- Local updates at every site each sweep
- Observables recorded every 10 steps post-thermalization
- Correlator estimator:
C( ) = x_j x_{j+ } averaged over j and Monte Carlo samples

Definition at line 121 of file Monte Carlo integral.py.

```
00121 def run_monte_carlo(lambda_parameter):
00122     """
00123     @brief Perform local Metropolis updates to sample Euclidean paths for fixed .
00124     @param lambda_parameter float: anharmonicity value for this simulation.
00125     @return tuple: (C_tau, acceptance_fraction)
00126         - C_tau: ndarray(float) correlator C( ) for up to T/2
00127         - acceptance_fraction: overall acceptance rate of updates
00128     @details
00129     • Initializes x( )=0 path
00130     • Local updates at every site each sweep
00131     • Observables recorded every 10 steps post-thermalization
00132     • Correlator estimator:
00133             C( ) = x_j x_{j+ } averaged over j and Monte Carlo samples
00134     """
00135     x_path = np.zeros(number_of_sites)
00136     G_correlator = np.zeros(number_of_sites // 2)
00137     N_measure = 0
00138     accepted_updates = 0
00139 
00140     for monte_carlo_step in range(total_monte_carlo_steps):
00141         for j in range(number_of_sites):
00142             x_new = x_path[j] + np.random.uniform(-proposal_step_size, proposal_step_size)
00143             delta_S_local = delta_action_change(x_path, j, x_new, lambda_parameter)
00144 
00145             if delta_S_local < 0 or np.exp(-delta_S_local) > np.random.rand():
00146                 x_path[j] = x_new
00147                 accepted_updates += 1
00148 
00149         if monte_carlo_step >= thermalization_steps and monte_carlo_step % 10 == 0:
00150             for t_index in range(number_of_sites // 2):
00151                 G_correlator[t_index] += np.mean(x_path * np.roll(x_path, -t_index))
00152             N_measure += 1
00153 
00154     G_correlator /= N_measure
00155     acceptance_fraction = accepted_updates / (total_monte_carlo_steps * number_of_sites)
00156     return G_correlator, acceptance_fraction
00157 
00158 
00159 # ============================================================
00160 #         Main Loop — Extract Ground-State Energy
00161 # ============================================================
00162 
```

Here is the call graph for this function:

3.4.2 Variable Documentation

3.4.2.1 acceptance_rate

Monte Carlo integral.acceptance_rate

Definition at line 167 of file Monte Carlo integral.py.

3.4.2.2 anharmonicity_values

list Monte Carlo integral.anharmonicity_values = [0.0, 0.1, 0.3, 0.5, 1.0]

Definition at line 48 of file Monte Carlo integral.py.

3.4.2.3 correlation_function

Monte Carlo integral.correlation_function

Definition at line 167 of file Monte Carlo integral.py.

3.4.2.4 estimated_E0

Monte Carlo integral.estimated_E0 = -slope

Definition at line 175 of file Monte Carlo integral.py.

3.4.2.5 estimated_ground_energies

list Monte Carlo integral.estimated_ground_energies = []

Definition at line 163 of file Monte Carlo integral.py.

3.4.2.6 euclidean_times

float Monte Carlo integral.euclidean_times = np.arange(len(correlation_function)) * lattice_spacing_a

Definition at line 170 of file Monte Carlo integral.py.

3.4.2.7 fit_slice

Monte Carlo integral.fit_slice = slice(1, 6)

Definition at line 172 of file Monte Carlo integral.py.

3.4.2.8 intercept

Monte Carlo integral.intercept

Definition at line 173 of file Monte Carlo integral.py.

3.4.2.9 lattice_spacing_a

float Monte Carlo integral.lattice_spacing_a = 0.1

Definition at line 33 of file Monte Carlo integral.py.

3.4.2.10 lw

Monte Carlo integral.lw

Definition at line 186 of file Monte Carlo integral.py.

3.4.2.11 number_of_sites

int Monte Carlo integral.number_of_sites = 100

Definition at line 36 of file Monte Carlo integral.py.

3.4.2.12 particle_mass

float Monte Carlo integral.particle_mass = 1.0

Definition at line 30 of file Monte Carlo integral.py.

3.4.2.13 proposal_step_size

float Monte Carlo integral.proposal_step_size = 0.5

Definition at line 45 of file Monte Carlo integral.py.

3.4.2.14 slope

Monte Carlo integral.slope

Definition at line 173 of file Monte Carlo integral.py.

3.4.2.15 thermalization_steps

int Monte Carlo integral.thermalization_steps = 5000

Definition at line 42 of file Monte Carlo integral.py.

3.4.2.16 total_monte_carlo_steps

int Monte Carlo integral.total_monte_carlo_steps = 30000

Definition at line 39 of file Monte Carlo integral.py.

3.5 Path Namespace Reference

integral

3.5.1 Detailed Description

integral

@mainpage Numerical Euclidean Path Integral for the Harmonic Oscillator
@file path_integral.py
@brief Brute-force multidimensional Euclidean path integral evaluation.

@details
This demonstration numerically evaluates the diagonal propagator
K(x, x; T) = x| e^(−H T) |x
for a 1D harmonic oscillator using a discretized Euclidean action:

$$S\_E = \Sigma\_j \left[m/(2a) \, (x\_\{j+1\} - x\_j)^2 + a \, V(x\_j) \right],$$

with fixed endpoints x_0 = x_N = x_fixed and (N−1) internal lattice points
integrated over a finite domain. The integral is computed using
SciPy's multi-dimensional quadrature ('nquad'), which scales
exponentially with N and serves only as a pedagogical reference
(not an efficient Monte Carlo method).

3.6 Path integral Namespace Reference

Functions

- potential_V (x)
- S_lat (x_list, x_fixed, *args)
- integrand (*x_list)

Variables

- int N = 4
- int lattice_spacing_a = 1 / 2
- float particle_mass = 1.0
- int bound_limit = 5
- list bounds = [(-bound_limit, bound_limit)] * (N - 1)
- list propagator = [ ]
- tuple normalization_A = (particle_mass / (2 * math.pi * lattice_spacing_a)) ** (N / 2)
- list x_values = [i * 0.25 for i in range(-10, 11)]
- result
- error

3.6.1 Function Documentation

3.6.1.1 integrand()

Path integral.integrand (
*x_list)

@brief Integrand exp(−S_E[x]) for numerical quadrature.
@param x_list variadic float: internal lattice points.
@return float: value of exp(−S_E).
@details
This closure captures 'x_fixed' from the loop scope.

Definition at line 98 of file Path integral.py.

```
00098     def integrand(*x_list):
00099         """
00100         @brief Integrand exp(−S_E[x]) for numerical quadrature.
00101         @param x_list variadic float: internal lattice points.
00102         @return float: value of exp(−S_E).
00103         @details
00104         This closure captures `x_fixed` from the loop scope.
00105         """
00106         return math.exp(-S_lat(x_list, x_fixed))
00107
```

Here is the call graph for this function:

3.6.1.2 potential_V()

Path integral.potential_V (
x)

@brief Harmonic oscillator potential.
@param x float: position value.
@return float: potential V(x) = 1/2 x².

Definition at line 54 of file Path integral.py.

```
00054 def potential_V(x):
00055     """
00056     @brief Harmonic oscillator potential.
00057     @param x float: position value.
00058     @return float: potential V(x) = 1/2 x².
00059     """
00060     return 0.5 * x**2
00061
00062
00063 # ====================================================================
00064 #                        Euclidean Action
00065 # ====================================================================
00066
```

Here is the caller graph for this function:

3.6.1.3 S_lat()

Path integral.S_lat (
x_list,
x_fixed,
∗ args)

@brief Compute discretized Euclidean path action.
@param x_list list(float): internal coordinates, length (N−1).
@param x_fixed float: fixed boundary value x = x_N.
@return float: Euclidean action S_E for given path.
@details
Constructs full path:
x = [x_fixed, x , x , ..., x_{N−1}, x_fixed]
and applies:
S_E = Σ_j m/(2a)(x_{j+1} − x_j)² + a V(x_j)
without periodic BCs since endpoints are fixed.

Definition at line 67 of file Path integral.py.

```
00067 def S_lat(x_list, x_fixed, *args):
00068     """
00069     @brief Compute discretized Euclidean path action.
00070     @param x_list list(float): internal coordinates, length (N−1).
00071     @param x_fixed float: fixed boundary value x  = x_N.
00072     @return float: Euclidean action S_E for given path.
00073     @details
00074     Constructs full path:
00075         x = [x_fixed, x , x , ..., x_{N−1}, x_fixed]
00076     and applies:
00077         S_E = Σ_j m/(2a)(x_{j+1} − x_j)² + a V(x_j)
00078     without periodic BCs since endpoints are fixed.
00079     """
00080     x = [x_fixed] + list(x_list) + [x_fixed]
00081     Action_S = 0
00082     for j in range(0, N - 1):
00083         x_derivative = x[j + 1] - x[j]
00084         Action_S += (particle_mass / (2 * lattice_spacing_a)) * x_derivative**2 \
00085                 + lattice_spacing_a * potential_V(x[j])
00086     return Action_S
00087
00088
00089 # ================================================================
00090 #             Propagator Evaluation Loop
00091 # ================================================================
00092
```

Here is the call graph for this function: Here is the caller graph for this function:

3.6.2 Variable Documentation

3.6.2.1 bound_limit

int Path integral.bound_limit = 5

Definition at line 37 of file Path integral.py.

3.6.2.2 bounds

list Path integral.bounds = [(-bound_limit, bound_limit)] ∗ (N - 1)

Definition at line 40 of file Path integral.py.

3.6.2.3 error

Path integral.error

Definition at line 108 of file Path integral.py.

3.6.2.4 lattice_spacing_a

int Path integral.lattice_spacing_a = 1 / 2

Definition at line 31 of file Path integral.py.

3.6.2.5 N

int Path integral.N = 4

Definition at line 28 of file Path integral.py.

3.6.2.6 normalization_A

tuple Path integral.normalization_A = (particle_mass / (2 * math.pi * lattice_spacing_a)) ** (N / 2)

Definition at line 46 of file Path integral.py.

3.6.2.7 particle_mass

float Path integral.particle_mass = 1.0

Definition at line 34 of file Path integral.py.

3.6.2.8 propagator

list Path integral.propagator = []

Definition at line 43 of file Path integral.py.

3.6.2.9 result

Path integral.result

Definition at line 108 of file Path integral.py.

3.6.2.10 x_values

list Path integral.x_values = [i * 0.25 for i in range(-10, 11)]

Definition at line 93 of file Path integral.py.

3.7 QCD_Lattice_SU3 Namespace Reference

Functions

- x_neighbor (x, mu, shift=1)
- su3_matrices (M)
- su2_random_unitary (eps)
- embed_su2_into_su3 (R2, i, j)
- plaquette_matrix (x, mu, nu, link_sites)
- real_trace_plaquette (x, mu, nu, link_sites)
- plaquettes_touching_link (x, mu, link_sites)
- metropolis_update (link_sites, eps_sub=0.06)
- average_plaquette_su3 (link_sites)
- bootstrap_mean_std (values, nboot=300)
- tune_eps_su3 (matrix0, target=0.5, initial_eps=0.06, tries=10, test_sweeps=150)
- run_su3_simulation (link_sites, eps_sub=0.06, burn_in_sweeps=500, MC_sweeps=2000, N_correlator=5)
- init_links_identity (link_sites)
- randomize_links_small (link_sites, amplitude=0.02)
- measure_wilson_loop_RT (link_sites, R, T, spatial_direction=0, time_direction=None)
- su3_simulation_with_wilson_loops (link_sites, eps_sub=0.06, burn_in_sweeps=500, MC_sweeps=2000, N_correlator=5, max_R=None, max_T=None)
- gell_mann_matrices ()
- extract_gluon_field (U, g=1.0, a=1.0)
- field_strength_tensor (link_sites, x, mu, nu, g=1.0, a=1.0)
- measure_avg_A2_and_F2 (link_sites, g=1.0, a=1.0)

Variables

- int spatial_dims = 1
- int lattice_size_L = 8
- float beta = 6.0
- float eps_initial = 0.06
- int burn_in_sweeps = 500
- int MC_sweeps = 2000
- int MC_measure_interval = 5
- int n_boot = 300
- int D = spatial_dims + 1
- tuple x_shape = (lattice_size_L,) * D
- link_matrix = np.zeros((D,) + x_shape + (3, 3), dtype=np.complex128)
- amplitude
- eps_tuned = tune_eps_su3(link_matrix, initial_eps=eps_initial)
- samples
- plaq_mean
- plaq_err
- eps_sub
- N_correlator
- figsize
- marker
- linestyle
- bins
- alpha
- wilson_loops_samples

- R_values
- T_values
- max_R
- max_T
- avg_wilson_loops = np.mean(wilson_loops_samples, axis=0)
- V_R = np.zeros(len(R_values))
- list potentials = []
- W_T = avg_wilson_loops[i, j]
- W_Tp1 = avg_wilson_loops[i, j + 1]
- A2_avg
- F2_avg

3.7.1 Detailed Description

@mainpage SU(3) Lattice Gauge Theory (Wilson action) — Cabibbo–Marinari Implementation

@file QCD_Lattice_SU3.py
@brief SU(3) lattice gauge theory simulation using Cabibbo–Marinari SU(2) subgroup updates.
@details
This module implements a pragmatic SU(3) lattice gauge theory code based on the Wilson action,
using Cabibbo–Marinari updates (embedded SU(2) rotations) together with local ΔS computations
(only plaquettes touching a link are recomputed) and reprojection to SU(3) via SVD to maintain
unitarity and $\det U = 1$.

It contains:
- Local Metropolis updates applying a sequence of small SU(2) rotations embedded into SU(3).
- Efficient local plaquette recomputation for ΔS evaluations.
- Utilities for plaquette measurement, bootstrap error estimation, Wilson loops and static potential.
- Helpers to extract approximate gauge fields (A^a) and field-strength components F^{\mu\nu}_a
  from link matrices for diagnostic/classical analysis.

@section references Key references
- K. G. Wilson, "Confinement of quarks," Phys. Rev. D 10, 2445 (1974).
- N. Cabibbo and E. Marinari, "A new method for updating SU(N) matrices," Phys. Lett. B119 (1982).
- G. P. Lepage lecture notes for pragmatic algorithmic choices.

3.7.2 Function Documentation

3.7.2.1 average_plaquette_su3()

QCD_Lattice_SU3.average_plaquette_su3 (
link_sites)

@brief Compute the normalized average plaquette Re Tr P /3 over the lattice.
@param link_sites ndarray: link array.
@return float: average plaquette normalized by color factor (3).
@details The Wilson action density per plaquette is proportional to (1 - Re Tr P / 3).

Definition at line 239 of file QCD_Lattice_SU3.py.

```
00239 def average_plaquette_su3(link_sites):
00240     """
00241     @brief Compute the normalized average plaquette  Re Tr P /3 over the lattice.
00242     @param link_sites ndarray: link array.
00243     @return float: average plaquette normalized by color factor (3).
00244     @details The Wilson action density per plaquette is proportional to (1 - Re Tr P / 3).
00245     """
00246     total = 0.0
00247     count = 0
00248     for x in np.ndindex(*x_shape):
00249         for mu in range(D):
00250             for nu in range(mu + 1, D):
00251                 trace = real_trace_plaquette(x, mu, nu, link_sites)
00252                 total += trace
00253                 count += 1
00254     # Normalize by color dimension (Tr 1 = 3)
00255     return (total / count) / 3.0
00256
00257
```

Here is the call graph for this function: Here is the caller graph for this function:

3.7.2.2 bootstrap_mean_std()

QCD_Lattice_SU3.bootstrap_mean_std (
values,
nboot = 300)

@brief Estimate mean and bootstrap standard error for a 1D array of samples.
@param values array-like: measurement samples.
@param nboot int: number of bootstrap resamples.
@return tuple: (boot_mean, boot_std)
@details We resample with replacement and compute sample means for each bootstrap realization; the returned std is the bootstrap estimate of the error.

Definition at line 258 of file QCD_Lattice_SU3.py.

```
00258 def bootstrap_mean_std(values, nboot=300):
00259     """
00260     @brief Estimate mean and bootstrap standard error for a 1D array of samples.
00261     @param values array-like: measurement samples.
00262     @param nboot int: number of bootstrap resamples.
00263     @return tuple: (boot_mean, boot_std)
00264     @details We resample with replacement and compute sample means for each bootstrap
00265             realization; the returned std is the bootstrap estimate of the error.
00266     """
00267     vals = np.asarray(values)
00268     N = len(vals)
00269     boots = np.zeros(nboot)
00270     for i in range(nboot):
00271         inds = np.random.randint(0, N, size=N)
00272         boots[i] = np.mean(vals[inds])
00273     return boots.mean(), boots.std(ddof=1)
00274
00275
00276 # ---------- Tuner & Runner ----------
```

Here is the caller graph for this function:

3.7.2.3 embed_su2_into_su3()

QCD_Lattice_SU3.embed_su2_into_su3 (
R2,
i,
j)

@brief Embed a 2x2 SU(2) matrix into SU(3) acting on indices (i, j).
@param R2 ndarray: 2x2 SU(2) matrix.
@param i int: first SU(2) index (0..2).
@param j int: second SU(2) index (0..2), must satisfy i < j.
@return ndarray: 3x3 matrix equal to identity except the 2x2 block at (i,j) replaced by R2.
@details This is the standard Cabibbo–Marinari embedding that extends SU(2) subgroup rotations to SU(3) by acting non-trivially on a chosen 2D subspace.

Definition at line 116 of file QCD_Lattice_SU3.py.

```
00116 def embed_su2_into_su3(R2, i, j):
00117     """
00118     @brief Embed a 2x2 SU(2) matrix into SU(3) acting on indices (i, j).
00119     @param R2 ndarray: 2x2 SU(2) matrix.
00120     @param i int: first SU(2) index (0..2).
00121     @param j int: second SU(2) index (0..2), must satisfy i < j.
00122     @return ndarray: 3x3 matrix equal to identity except the 2x2 block at (i,j) replaced by R2.
00123     @details This is the standard Cabibbo–Marinari embedding that extends SU(2) subgroup rotations
00124             to SU(3) by acting non-trivially on a chosen 2D subspace.
00125     """
00126     R = np.eye(3, dtype=complex)
00127     R[i, i] = R2[0, 0]
00128     R[i, j] = R2[0, 1]
00129     R[j, i] = R2[1, 0]
00130     R[j, j] = R2[1, 1]
00131     return R
00132
00133
00134 # ---------- Plaquette helpers & local ΔS computation ----------
```

Here is the caller graph for this function:

3.7.2.4 extract_gluon_field()

QCD_Lattice_SU3.extract_gluon_field (
U,
g = 1.0,
a = 1.0)

@brief Extract approximate local gauge field components A^a from a single SU(3) link.
@param U ndarray: SU(3) link matrix.
@param g float: gauge coupling (default 1.0).
@param a float: lattice spacing (default 1.0).
@return ndarray: array shape (8,) containing A^a components (real).
@details For small lattice spacing we approximate U exp(i g a A) => A (U - U†)/(2 i g a).
We then project the traceless anti-Hermitian part onto the Gell-Mann basis.

Definition at line 530 of file QCD_Lattice_SU3.py.

```
00530 def extract_gluon_field(U, g=1.0, a=1.0):
00531     """
00532     @brief Extract approximate local gauge field components A^a from a single SU(3) link.
00533     @param U ndarray: SU(3) link matrix.
00534     @param g float: gauge coupling (default 1.0).
00535     @param a float: lattice spacing (default 1.0).
00536     @return ndarray: array shape (8,) containing A^a components (real).
00537     @details For small lattice spacing we approximate U   exp(i g a A) => A   (U - U†)/(2 i g a).
00538             We then project the traceless anti-Hermitian part onto the Gell-Mann basis.
00539     """
00540     difference = (U - U.conj().T) / (2j * g * a)
00541     difference -= np.trace(difference).real / 3.0 * np.eye(3)
00542     lambda_ = gell_mann_matrices()
00543     A_components = np.array([np.real(np.trace(difference @ lambda_a)) / 2.0 for lambda_a in lambda_])
00544     return A_components
00545
00546
```

Here is the call graph for this function: Here is the caller graph for this function:

3.7.2.5 field_strength_tensor()

QCD_Lattice_SU3.field_strength_tensor (
link_sites,
x,
mu,
nu,
g = 1.0,
a = 1.0)

@brief Compute the lattice field-strength components F_{mu,nu}^a at site x from the plaquette.
@param link_sites ndarray: link variables.
@param x tuple: lattice coordinate.
@param mu int: direction mu.
@param nu int: direction nu.
@param g float: gauge coupling.
@param a float: lattice spacing.
@return ndarray: shape (8,) F^a components (real).
@details Uses the anti-Hermitian traceless projection of the plaquette:
F ~ (P - P†)/(2 i g a^2) projected on Gell-Mann matrices.

Definition at line 547 of file QCD_Lattice_SU3.py.

```
00547 def field_strength_tensor(link_sites, x, mu, nu, g=1.0, a=1.0):
00548     """
00549     @brief Compute the lattice field-strength components F_{mu,nu}^a at site x from the plaquette.
00550     @param link_sites ndarray: link variables.
00551     @param x tuple: lattice coordinate.
00552     @param mu int: direction mu.
```

```
00553     @param nu int: direction nu.
00554     @param g float: gauge coupling.
00555     @param a float: lattice spacing.
00556     @return ndarray: shape (8,) F^a components (real).
00557     @details Uses the anti-Hermitian traceless projection of the plaquette:
00558             F ~ (P - P†)/(2 i g a^2) projected on Gell-Mann matrices.
00559     """
00560     P = plaquette_matrix(x, mu, nu, link_sites)
00561     difference = (P - P.conj().T) / (2j * g * a ** 2)
00562     difference -= np.trace(difference).real / 3.0 * np.eye(3)
00563     lambda_ = gell_mann_matrices()
00564     F_components = np.array([np.real(np.trace(difference @ lambda_a)) / 2.0 for lambda_a in lambda_])
00565     return F_components
00566
00567
```

Here is the call graph for this function: Here is the caller graph for this function:

3.7.2.6 gell_mann_matrices()

QCD_Lattice_SU3.gell_mann_matrices ()

@brief Return the eight Gell-Mann matrices ^a (3x3).
@return list: eight 3x3 numpy arrays forming a basis for su(3).
@details These are used to project Lie-algebra components from SU(3) link matrices.

Definition at line 512 of file QCD_Lattice_SU3.py.

```
00512 def gell_mann_matrices():
00513     """
00514     @brief Return the eight Gell-Mann matrices ^a (3x3).
00515     @return list: eight 3x3 numpy arrays forming a basis for su(3).
00516     @details These are used to project Lie-algebra components from SU(3) link matrices.
00517     """
00518     lambda_ = []
00519     lambda_.append(np.array([[0, 1, 0], [1, 0, 0], [0, 0, 0]], dtype=complex))
00520     lambda_.append(np.array([[0, -1j, 0], [1j, 0, 0], [0, 0, 0]], dtype=complex))
00521     lambda_.append(np.array([[1, 0, 0], [0, -1, 0], [0, 0, 0]], dtype=complex))
00522     lambda_.append(np.array([[0, 0, 1], [0, 0, 0], [1, 0, 0]], dtype=complex))
00523     lambda_.append(np.array([[0, 0, -1j], [0, 0, 0], [1j, 0, 0]], dtype=complex))
00524     lambda_.append(np.array([[0, 0, 0], [0, 0, 1], [0, 1, 0]], dtype=complex))
00525     lambda_.append(np.array([[0, 0, 0], [0, 0, -1j], [0, 1j, 0]], dtype=complex))
00526     lambda_.append((1 / np.sqrt(3)) * np.array([[1, 0, 0], [0, 1, 0], [0, 0, -2]], dtype=complex))
00527     return lambda_
00528
00529
```

Here is the caller graph for this function:

3.7.2.7 init_links_identity()

QCD_Lattice_SU3.init_links_identity (
link_sites)

@brief Initialize all links to the identity matrix.
@param link_sites ndarray: link array to initialize (modified in-place).

Definition at line 335 of file QCD_Lattice_SU3.py.

```
00335 def init_links_identity(link_sites):
00336     """
00337     @brief Initialize all links to the identity matrix.
00338     @param link_sites ndarray: link array to initialize (modified in-place).
00339     """
00340     for mu in range(D):
00341         for x in np.ndindex(*x_shape):
00342             link_sites[(mu,) + x] = np.eye(3, dtype=complex)
00343
00344
```

Here is the caller graph for this function:

3.7.2.8 measure_avg_A2_and_F2()

QCD_Lattice_SU3.measure_avg_A2_and_F2 (
link_sites,
g = 1.0,
a = 1.0)

@brief Compute averages A^2 and F^2 over the entire lattice as diagnostics.
@param link_sites ndarray: link configuration.
@param g float: gauge coupling.
@param a float: lattice spacing.
@return tuple: (A2_avg float, F2_avg float)
@details A^2 and F^2 are computed by summing squares of components and normalizing by counts.

Definition at line 568 of file QCD_Lattice_SU3.py.

```
00568 def measure_avg_A2_and_F2(link_sites, g=1.0, a=1.0):
00569     """
00570     @brief Compute averages A^2 and F^2 over the entire lattice as diagnostics.
00571     @param link_sites ndarray: link configuration.
00572     @param g float: gauge coupling.
00573     @param a float: lattice spacing.
00574     @return tuple: (A2_avg float, F2_avg float)
00575     @details A^2 and F^2 are computed by summing squares of components and normalizing by counts.
00576     """
00577     A2_sum = 0.0
00578     F2_sum = 0.0
00579     nA = 0
00580     nF = 0
00581     for x in np.ndindex(*x_shape):
00582         for mu in range(D):
00583             U = link_sites[(mu,) + x]
00584             A = extract_gluon_field(U, g=g, a=a)
00585             A2_sum += np.dot(A, A)
00586             nA += 1
00587         for mu in range(D):
00588             for nu in range(mu + 1, D):
00589                 F = field_strength_tensor(link_sites, x, mu, nu, g=g, a=a)
00590                 F2_sum += np.dot(F, F)
00591                 nF += 1
00592     return A2_sum / nA, F2_sum / nF
00593
00594
```

Here is the call graph for this function:

3.7.2.9 measure_wilson_loop_RT()

QCD_Lattice_SU3.measure_wilson_loop_RT (
link_sites,
R,
T,
spatial_direction = 0,
time_direction = None)

@brief Measure the average Wilson loop W(R,T) for rectangular loops of spatial size R and temporal extent T.
@param link_sites ndarray: link configuration.
@param R int: spatial extent (number of spatial steps).
@param T int: temporal extent (number of temporal steps).
@param spatial_direction int: spatial direction index used for the R side.
@param time_direction int or None: time direction index; defaults to D-1 (last axis).
@return float: average Re Tr[W(R,T)] / 3 over all possible loop origins.
@details
The loop path starts at each lattice site x and multiplies the link matrices along the rectangular contour.
Backward traversals multiply by Hermitian conjugate of the traversed link.

Definition at line 361 of file QCD_Lattice_SU3.py.

```
00361 def measure_wilson_loop_RT(link_sites, R, T, spatial_direction=0, time_direction=None):
00362     """
00363     @brief Measure the average Wilson loop W(R,T) for rectangular loops of spatial size R and temporal extent T.
00364     @param link_sites ndarray: link configuration.
00365     @param R int: spatial extent (number of spatial steps).
00366     @param T int: temporal extent (number of temporal steps).
00367     @param spatial_direction int: spatial direction index used for the R side.
00368     @param time_direction int or None: time direction index; defaults to D-1 (last axis).
00369     @return float: average Re Tr[W(R,T)] / 3 over all possible loop origins.
00370     @details
00371     The loop path starts at each lattice site x and multiplies the link matrices along the rectangular contour.
00372     Backward traversals multiply by Hermitian conjugate of the traversed link.
00373     """
00374     if time_direction is None:
00375         time_direction = D - 1
00376     total = 0.0
00377     count = 0
00378     for x in np.ndindex(*x_shape):
00379         current_x = x
00380         W = np.eye(3, dtype=complex)
00381         # R steps + spatial_direction
00382         for i in range(R):
00383             U = link_sites[(spatial_direction,) + current_x]
00384             W = W @ U
00385             current_x = x_neighbor(current_x, spatial_direction, 1)
00386         # T steps + time_direction
00387         for i in range(T):
00388             U = link_sites[(time_direction,) + current_x]
00389             W = W @ U
00390             current_x = x_neighbor(current_x, time_direction, 1)
00391         # R steps - spatial_direction (backwards)
00392         for i in range(R):
00393             current_x = x_neighbor(current_x, spatial_direction, -1)
00394             U = link_sites[(spatial_direction,) + current_x]
00395             W = W @ U.conj().T
00396         # T steps - time_direction (backwards)
00397         for i in range(T):
00398             current_x = x_neighbor(current_x, time_direction, -1)
00399             U = link_sites[(time_direction,) + current_x]
00400             W = W @ U.conj().T
00401         total += np.real(np.trace(W)) / 3.0
00402         count += 1
00403     return total / count
00404
00405
```

Here is the call graph for this function: Here is the caller graph for this function:

3.7.2.10 metropolis_update()

QCD_Lattice_SU3.metropolis_update (
link_sites,
eps_sub = 0.06)

@brief Perform a single Metropolis sweep over all links applying embedded SU(2) updates.
@param link_sites ndarray: link variable array (modified in-place).
@param eps_sub float: SU(2) proposal amplitude for each embedded sub-update.
@return tuple: (accepted int, proposals int)
@details
For each link U_mu(x) we cycle through the three SU(2) subgroups (0,1), (0,2), (1,2).
For each subgroup:

1. compute sum_old = Σ Re Tr(P) over plaquettes touching the link,
2. propose an SU(2) rotation R2, embed into SU(3) → R3,
3. set U_candidate = R3 @ U_old and reproject to SU(3),
4. compute sum_new and ΔS = - (/3) (sum_new - sum_old),
5. accept/reject with Metropolis probability.

Using only touching plaquettes makes ΔS computation local and efficient.

Definition at line 190 of file QCD_Lattice_SU3.py.

```
00190 def metropolis_update(link_sites, eps_sub=0.06):
00191     """
00192     @brief Perform a single Metropolis sweep over all links applying embedded SU(2) updates.
00193     @param link_sites ndarray: link variable array (modified in-place).
```

```
00194     @param eps_sub float: SU(2) proposal amplitude for each embedded sub-update.
00195     @return tuple: (accepted int, proposals int)
00196     @details
00197     For each link U_mu(x) we cycle through the three SU(2) subgroups (0,1), (0,2), (1,2).
00198     For each subgroup:
00199       1. compute sum_old = Σ Re Tr(P) over plaquettes touching the link,
00200       2. propose an SU(2) rotation R2, embed into SU(3) → R3,
00201       3. set U_candidate = R3 @ U_old and reproject to SU(3),
00202       4. compute sum_new and ΔS = - ( /3) (sum_new - sum_old),
00203       5. accept/reject with Metropolis probability.
00204     Using only touching plaquettes makes ΔS computation local and efficient.
00205     """
00206     accepted = 0
00207     proposals = 0
00208     su2_pairs = [(0, 1), (0, 2), (1, 2)]
00209     for mu in range(D):
00210         for x in np.ndindex(*x_shape):
00211             U_old = link_sites[(mu,) + x].copy()
00212             for (i, j) in su2_pairs:
00213                 plist = plaquettes_touching_link(x, mu, link_sites)
00214                 sum_old = sum(trace for (_meta, trace) in plist)
00215 
00216                 R2 = su2_random_unitary(eps_sub)
00217                 R3 = embed_su2_into_su3(R2, i, j)
00218                 link_sites[(mu,) + x] = R3 @ U_old
00219                 # Reproject to SU(3) to correct numerical drift
00220                 link_sites[(mu,) + x] = su3_matrices(link_sites[(mu,) + x])
00221 
00222                 new_p_list = plaquettes_touching_link(x, mu, link_sites)
00223                 sum_new = sum(trace for (_meta, trace) in new_p_list)
00224 
00225                 dS = - (beta / 3.0) * (sum_new - sum_old)
00226                 proposals += 1
00227                 # Metropolis acceptance: accept if dS <= 0 or with probability exp(-dS)
00228                 if dS > 0 and np.exp(-dS) < np.random.rand():
00229                     # reject: revert this subgroup update (resume next subgroup from U_old)
00230                     link_sites[(mu,) + x] = U_old.copy()
00231                 else:
00232                     # accept: update U_old so subsequent subgroup multiplications act on accepted matrix
00233                     U_old = link_sites[(mu,) + x].copy()
00234                     accepted += 1
00235     return accepted, proposals
00236 
00237 
00238 # ---------- Observables ----------
```

Here is the call graph for this function: Here is the caller graph for this function:

3.7.2.11 plaquette_matrix()

QCD_Lattice_SU3.plaquette_matrix (
x,
mu,
nu,
link_sites)

@brief Construct the plaquette matrix U_mu(x) U_nu(x+mu) U_mu^\dagger(x+nu) U_nu^\dagger(x).
@param x tuple: lattice coordinate.
@param mu int: direction index mu.
@param nu int: direction index nu.
@param link_sites ndarray: link variable array.
@return ndarray: 3x3 plaquette matrix P_{mu,nu}(x).

Definition at line 135 of file QCD_Lattice_SU3.py.

```
00135 def plaquette_matrix(x, mu, nu, link_sites):
00136     """
00137     @brief Construct the plaquette matrix U_mu(x) U_nu(x+mu) U_mu^\dagger(x+nu) U_nu^\dagger(x).
00138     @param x tuple: lattice coordinate.
00139     @param mu int: direction index mu.
00140     @param nu int: direction index nu.
00141     @param link_sites ndarray: link variable array.
00142     @return ndarray: 3x3 plaquette matrix P_{mu,nu}(x).
00143     """
00144     x_plus_mu = x_neighbor(x, mu, 1)
```

```
00145     x_plus_nu = x_neighbor(x, nu, 1)
00146     U_mu = link_sites[(mu,) + x]
00147     U_nu_xmu = link_sites[(nu,) + x_plus_mu]
00148     U_mu_xnu = link_sites[(mu,) + x_plus_nu]
00149     U_nu = link_sites[(nu,) + x]
00150     P = U_mu @ U_nu_xmu @ U_mu_xnu.conj().T @ U_nu.conj().T
00151     return P
00152
00153
```

Here is the call graph for this function: Here is the caller graph for this function:

3.7.2.12 plaquettes_touching_link()

```
QCD_Lattice_SU3.plaquettes_touching_link (
                x,
                mu,
                link_sites)
```

@brief List plaquettes that include the link at (mu, x).
@param x tuple: lattice coordinate of the starting site of the link.
@param mu int: link direction.
@param link_sites ndarray: array of link matrices.
@return list: entries [((x_plaq, mu, nu), real_trace), ...] for all plaquettes touching the link.
@details
For each nu != mu, the link (mu,x) sits in two elementary plaquettes:
- the plaquette at x in the (mu,nu) plane,
- the plaquette at x - e_nu in the (mu,nu) plane.
Only these plaquettes are required to compute the local change in action when U_mu(x) is updated.

Definition at line 164 of file QCD_Lattice_SU3.py.

```
00164 def plaquettes_touching_link(x, mu, link_sites):
00165     """
00166     @brief List plaquettes that include the link at (mu, x).
00167     @param x tuple: lattice coordinate of the starting site of the link.
00168     @param mu int: link direction.
00169     @param link_sites ndarray: array of link matrices.
00170     @return list: entries [((x_plaq, mu, nu), real_trace), ...] for all plaquettes touching the link.
00171     @details
00172     For each nu != mu, the link (mu,x) sits in two elementary plaquettes:
00173       - the plaquette at x in the (mu,nu) plane,
00174       - the plaquette at x - e_nu in the (mu,nu) plane.
00175     Only these plaquettes are required to compute the local change in action when U_mu(x) is updated.
00176     """
00177     p_list = []
00178     for nu in range(D):
00179         if nu == mu:
00180             continue
00181         trace_1 = real_trace_plaquette(x, mu, nu, link_sites)
00182         p_list.append(((x, mu, nu), trace_1))
00183         x_minus_nu = x_neighbor(x, nu, -1)
00184         trace_2 = real_trace_plaquette(x_minus_nu, mu, nu, link_sites)
00185         p_list.append(((x_minus_nu, mu, nu), trace_2))
00186     return p_list
00187
00188
00189 # ---------- Local update: Metropolis with embedded SU(2) updates ----------
```

Here is the call graph for this function: Here is the caller graph for this function:

3.7.2.13 randomize_links_small()

```
QCD_Lattice_SU3.randomize_links_small (
                link_sites,
                amplitude = 0.02)
```

@brief Apply small random SU(3) rotations (via embedded SU(2)) to each link for breaking symmetry.
@param link_sites ndarray: link array (modified in-place).
@param amplitude float: small rotation amplitude used for initial randomization.
@details Useful to seed the Markov chain with a slightly randomized starting configuration.

Definition at line 345 of file QCD_Lattice_SU3.py.

```
00345 def randomize_links_small(link_sites, amplitude=0.02):
00346     """
00347     @brief Apply small random SU(3) rotations (via embedded SU(2)) to each link for breaking symmetry.
00348     @param link_sites ndarray: link array (modified in-place).
00349     @param amplitude float: small rotation amplitude used for initial randomization.
00350     @details Useful to seed the Markov chain with a slightly randomized starting configuration.
00351     """
00352     for mu in range(D):
00353         for x in np.ndindex(*x_shape):
00354             for (i, j) in [(0, 1), (0, 2), (1, 2)]:
00355                 R2 = su2_random_unitary(amplitude)
00356                 R3 = embed_su2_into_su3(R2, i, j)
00357                 link_sites[(mu,) + x] = su3_matrices(R3 @ link_sites[(mu,) + x])
00358 
00359 
00360 # ---------- Wilson loop helper ----------
```

Here is the call graph for this function:

3.7.2.14 real_trace_plaquette()

QCD_Lattice_SU3.real_trace_plaquette (
x,
mu,
nu,
link_sites)

@brief Compute the real part of the trace of the plaquette matrix.
@return float: Re Tr[P_{mu,nu}(x)].

Definition at line 154 of file QCD_Lattice_SU3.py.

```
00154 def real_trace_plaquette(x, mu, nu, link_sites):
00155     """
00156     @brief Compute the real part of the trace of the plaquette matrix.
00157     @return float: Re Tr[P_{mu,nu}(x)].
00158     """
00159     P = plaquette_matrix(x, mu, nu, link_sites)
00160     trace = np.trace(P)
00161     return float(np.real(trace))
00162 
00163 
```

Here is the call graph for this function: Here is the caller graph for this function:

3.7.2.15 run_su3_simulation()

QCD_Lattice_SU3.run_su3_simulation (
link_sites,
eps_sub = 0.06,
burn_in_sweeps = 500,
MC_sweeps = 2000,
N_correlator = 5)

@brief Run SU(3) Metropolis simulation collecting plaquette samples.
@param link_sites ndarray: initial link configuration (modified in-place).
@param eps_sub float: SU(2) subgroup proposal amplitude.
@param burn_in_sweeps int: thermalization sweeps.
@param MC_sweeps int: measurement sweeps.
@param N_correlator int: interval between stored measurements.
@return tuple: (plaquette_samples ndarray, mean_plaquette float, error_plaquette float)
@details After burn-in we perform MC_sweeps sweeps and measure the average plaquette every
N_correlator sweeps. Bootstrap error estimation is applied to the set of plaquette samples.

Definition at line 306 of file QCD_Lattice_SU3.py.

```
00306 def run_su3_simulation(link_sites, eps_sub=0.06, burn_in_sweeps=500, MC_sweeps=2000, N_correlator=5):
00307     """
00308     @brief Run SU(3) Metropolis simulation collecting plaquette samples.
00309     @param link_sites ndarray: initial link configuration (modified in-place).
00310     @param eps_sub float: SU(2) subgroup proposal amplitude.
00311     @param burn_in_sweeps int: thermalization sweeps.
00312     @param MC_sweeps int: measurement sweeps.
00313     @param N_correlator int: interval between stored measurements.
00314     @return tuple: (plaquette_samples ndarray, mean_plaquette float, error_plaquette float)
00315     @details After burn-in we perform MC_sweeps sweeps and measure the average plaquette every
00316             N_correlator sweeps. Bootstrap error estimation is applied to the set of plaquette samples.
00317     """
00318     accepted = proposed = 0
00319     for i in range(burn_in_sweeps):
00320         a, p = metropolis_update(link_sites, eps_sub=eps_sub)
00321         accepted += a; proposed += p
00322     plaquette_samples = []
00323     accepted = proposed = 0
00324     for sweep in range(MC_sweeps):
00325         a, p = metropolis_update(link_sites, eps_sub=eps_sub)
00326         accepted += a; proposed += p
00327         if sweep % N_correlator == 0:
00328             plaquette = average_plaquette_su3(link_sites)
00329             plaquette_samples.append(plaquette)
00330     mean_plaquette, error_plaquette = bootstrap_mean_std(plaquette_samples, nboot=n_boot)
00331     return np.array(plaquette_samples), mean_plaquette, error_plaquette
00332
00333
00334 # ---------- Initialization helpers ----------
```

Here is the call graph for this function:

3.7.2.16 su2_random_unitary()

QCD_Lattice_SU3.su2_random_unitary (
eps)

@brief Generate a small random SU(2) rotation matrix using Gaussian parameters.
@param eps float: amplitude controlling rotation angle scale (a = eps * |r|).
@return ndarray: 2x2 complex SU(2) matrix.
@details
The parametrization uses R = cos(a) I + i sin(a) n · where n is a unit 3-vector
and _i are the Pauli matrices. We project via SVD to correct numerical drift and
ensure exact unitarity, then enforce det=1.

Definition at line 86 of file QCD_Lattice_SU3.py.

```
00086 def su2_random_unitary(eps):
00087     """
00088     @brief Generate a small random SU(2) rotation matrix using Gaussian parameters.
00089     @param eps float: amplitude controlling rotation angle scale (a = eps * |r|).
00090     @return ndarray: 2x2 complex SU(2) matrix.
00091     @details
00092     The parametrization uses R = cos(a) I + i sin(a) n · where n is a unit 3-vector
00093     and _i are the Pauli matrices. We project via SVD to correct numerical drift and
00094     ensure exact unitarity, then enforce det=1.
00095     """
00096     r = np.random.normal(size=3)
00097     r_norm = np.linalg.norm(r)
00098     if r_norm == 0:
00099         return np.eye(2, dtype=complex)
00100     a = eps * r_norm
```

```
00101     n = r / r_norm
00102     # Pauli matrices
00103     sigma1 = np.array([[0.0, 1.0], [1.0, 0.0]], dtype=complex)
00104     sigma2 = np.array([[0.0, -1j], [1j, 0.0]], dtype=complex)
00105     sigma3 = np.array([[1.0, 0.0], [0.0, -1.0]], dtype=complex)
00106     ndotsigma = n[0] * sigma1 + n[1] * sigma2 + n[2] * sigma3
00107     R = np.cos(a) * np.eye(2, dtype=complex) + 1j * np.sin(a) * ndotsigma
00108     # Project R to exact SU(2) via SVD/polar projection and fix determinant
00109     U, s, Vh = LA.svd(R)
00110     R_projection = U @ Vh
00111     det = LA.det(R_projection)
00112     R_projection /= (det ** 0.5)
00113     return R_projection
00114
00115
```

Here is the caller graph for this function:

3.7.2.17 su3_matrices()

```
QCD_Lattice_SU3.su3_matrices (
                M)
```

@brief Project a general complex 3x3 matrix to SU(3) via unitary polar/SVD projection.
@param M (ndarray): 3x3 complex matrix (candidate link).
@return ndarray: Unitary 3x3 matrix with det = 1 (projection of M into SU(3)).
@details
We perform an SVD: M = U S V^H and set U_proj = U V^H (closest unitary in Frobenius norm).
A global phase is then removed to enforce det(U_proj) = 1. If the projection yields
a near-singular matrix we add a tiny perturbation as fallback.

Definition at line 62 of file QCD_Lattice_SU3.py.

```
00062 def su3_matrices(M):
00063     """
00064     @brief Project a general complex 3x3 matrix to SU(3) via unitary polar/SVD projection.
00065     @param M (ndarray): 3x3 complex matrix (candidate link).
00066     @return ndarray: Unitary 3x3 matrix with det = 1 (projection of M into SU(3)).
00067     @details
00068     We perform an SVD: M = U S V^H and set U_proj = U V^H (closest unitary in Frobenius norm).
00069     A global phase is then removed to enforce det(U_proj) = 1. If the projection yields
00070     a near-singular matrix we add a tiny perturbation as fallback.
00071     """
00072     U, s, Vh = LA.svd(M)
00073     U_projection = U @ Vh
00074     determinant = LA.det(U_projection)
00075     if determinant == 0 or np.isnan(determinant):
00076         # Numerical fallback: small perturbation then reproject
00077         U_projection = U_projection + 1e-12 * np.eye(3, dtype=complex)
00078         determinant = LA.det(U_projection)
00079     # Remove global phase to ensure unit determinant
00080     phase = determinant ** (1.0 / 3.0)
00081     U_projection /= phase
00082     return U_projection
00083
00084
00085 # ---------- SU(2) small updater (embedded in SU(3)) ----------
```

Here is the caller graph for this function:

3.7.2.18 su3_simulation_with_wilson_loops()

```
QCD_Lattice_SU3.su3_simulation_with_wilson_loops (
                link_sites,
                eps_sub = 0.06,
                burn_in_sweeps = 500,
                MC_sweeps = 2000,
                N_correlator = 5,
                max_R = None,
                max_T = None)
```

@brief Run full SU(3) simulation storing Wilson loop matrices for each measurement.
@param link_sites ndarray: initial link configuration (modified in-place).
@param eps_sub float: SU(2) subgroup proposal amplitude.
@param burn_in_sweeps int: thermalization sweeps.
@param MC_sweeps int: measurement sweeps.
@param N_correlator int: interval between stored measurements.
@param max_R int or None: maximum spatial size to measure (defaults to L/2).
@param max_T int or None: maximum temporal size to measure (defaults to L/2).
@return tuple: (wilson_loops_samples ndarray [n_meas, n_R, n_T], R_values ndarray, T_values ndarray)
@details
Measures a grid of Wilson loops W(R,T) for R in [1..max_R], T in [1..max_T] at each stored configuration.

Definition at line 406 of file QCD_Lattice_SU3.py.

```
00406 def su3_simulation_with_wilson_loops(link_sites, eps_sub=0.06, burn_in_sweeps=500, MC_sweeps=2000,
      N_correlator=5, max_R=None, max_T=None):
00407     """
00408     @brief Run full SU(3) simulation storing Wilson loop matrices for each measurement.
00409     @param link_sites ndarray: initial link configuration (modified in-place).
00410     @param eps_sub float: SU(2) subgroup proposal amplitude.
00411     @param burn_in_sweeps int: thermalization sweeps.
00412     @param MC_sweeps int: measurement sweeps.
00413     @param N_correlator int: interval between stored measurements.
00414     @param max_R int or None: maximum spatial size to measure (defaults to L/2).
00415     @param max_T int or None: maximum temporal size to measure (defaults to L/2).
00416     @return tuple: (wilson_loops_samples ndarray [n_meas, n_R, n_T], R_values ndarray, T_values ndarray)
00417     @details
00418     Measures a grid of Wilson loops W(R,T) for R in [1..max_R], T in [1..max_T] at each stored configuration.
00419     """
00420     if max_R is None:
00421         max_R = lattice_size_L // 2
00422     if max_T is None:
00423         max_T = lattice_size_L // 2
00424     R_values = np.arange(1, max_R + 1)
00425     T_values = np.arange(1, max_T + 1)
00426     n_R = len(R_values)
00427     n_T = len(T_values)
00428     # Thermalize
00429     for i in range(burn_in_sweeps):
00430         metropolis_update(link_sites, eps_sub=eps_sub)
00431     wilson_loops_samples = []
00432     for sweep in range(MC_sweeps):
00433         metropolis_update(link_sites, eps_sub=eps_sub)
00434         if sweep % N_correlator == 0:
00435             W_sample = np.zeros((n_R, n_T))
00436             for i, R in enumerate(R_values):
00437                 for j, T in enumerate(T_values):
00438                     W_sample[i, j] = measure_wilson_loop_RT(link_sites, R, T)
00439             wilson_loops_samples.append(W_sample)
00440     wilson_loops_samples = np.array(wilson_loops_samples)
00441     return wilson_loops_samples, R_values, T_values
00442
00443
00444 # ============================================================
00445 # ================ PLAQUETTE CALCULATION SECTION ==============
00446 # ============================================================
```

Here is the call graph for this function:

3.7.2.19 tune_eps_su3()

QCD_Lattice_SU3.tune_eps_su3 (
matrix0,
target = 0.5,
initial_eps = 0.06,
tries = 10,
test_sweeps = 150)

@brief Tune the SU(2) proposal amplitude eps so that acceptance fraction ~ target.
@param matrix0 ndarray: initial link matrix copy for tuning (will be copied internally).
@param target float: desired acceptance fraction (e.g., 0.5).
@param initial_eps float: starting amplitude.
@param tries int: maximum adjustment attempts.
@param test_sweeps int: sweeps per tuning test.
@return float: tuned eps value.
@details We perform a small number of sweeps and adjust eps multiplicatively to move acceptance fraction towards target. This is a heuristic tuner used before a production run.

Definition at line 277 of file QCD_Lattice_SU3.py.

```
00277 def tune_eps_su3(matrix0, target=0.5, initial_eps=0.06, tries=10, test_sweeps=150):
00278     """
00279     @brief Tune the SU(2) proposal amplitude eps so that acceptance fraction ~ target.
00280     @param matrix0 ndarray: initial link matrix copy for tuning (will be copied internally).
00281     @param target float: desired acceptance fraction (e.g., 0.5).
00282     @param initial_eps float: starting amplitude.
00283     @param tries int: maximum adjustment attempts.
00284     @param test_sweeps int: sweeps per tuning test.
00285     @return float: tuned eps value.
00286     @details We perform a small number of sweeps and adjust eps multiplicatively to move acceptance
00287              fraction towards target. This is a heuristic tuner used before a production run.
00288     """
00289     eps = initial_eps
00290     for attempt in range(tries):
00291         matrix_copy = matrix0.copy()
00292         # quick thermalize copy
00293         for i in range(50):
00294             metropolis_update(matrix_copy, eps_sub=eps)
00295         accepted = proposed = 0
00296         for i in range(test_sweeps):
00297             a, p = metropolis_update(matrix_copy, eps_sub=eps)
00298             accepted += a; proposed += p
00299         fraction = accepted / proposed if proposed > 0 else 0.0
00300         if abs(fraction - target) < 0.05:
00301             break
00302         eps *= 1.2 if fraction > target else 0.8
00303     return eps
00304
00305
```

Here is the call graph for this function:

3.7.2.20 x_neighbor()

QCD_Lattice_SU3.x_neighbor (
x,
mu,
shift = 1)

@brief Periodic lattice neighbor coordinate.
@param x tuple: Lattice coordinate (length D).
@param mu int: Direction index (0..D-1).
@param shift int: Integer shift (positive forward, negative backward).
@return tuple: New lattice coordinate (with periodic wrap).
@details Implements periodic boundary conditions: (x_mu + shift) mod L.

Definition at line 48 of file QCD_Lattice_SU3.py.

```
00048 def x_neighbor(x, mu, shift=1):
00049     """
00050     @brief Periodic lattice neighbor coordinate.
00051     @param x tuple: Lattice coordinate (length D).
00052     @param mu int: Direction index (0..D-1).
00053     @param shift int: Integer shift (positive forward, negative backward).
00054     @return tuple: New lattice coordinate (with periodic wrap).
00055     @details Implements periodic boundary conditions: (x_mu + shift) mod L.
00056     """
00057     x_new = list(x)
00058     x_new[mu] = (x_new[mu] + shift) % lattice_size_L
00059     return tuple(x_new)
00060
00061
```

Here is the caller graph for this function:

3.7.3 Variable Documentation

3.7.3.1 A2_avg

QCD_Lattice_SU3.A2_avg

Definition at line 595 of file QCD_Lattice_SU3.py.

3.7.3.2 alpha

QCD_Lattice_SU3.alpha

Definition at line 468 of file QCD_Lattice_SU3.py.

3.7.3.3 amplitude

QCD_Lattice_SU3.amplitude

Definition at line 448 of file QCD_Lattice_SU3.py.

3.7.3.4 avg_wilson_loops

QCD_Lattice_SU3.avg_wilson_loops = np.mean(wilson_loops_samples, axis=0)

Definition at line 484 of file QCD_Lattice_SU3.py.

3.7.3.5 beta

float QCD_Lattice_SU3.beta = 6.0

Definition at line 33 of file QCD_Lattice_SU3.py.

3.7.3.6 bins

QCD_Lattice_SU3.bins

Definition at line 468 of file QCD_Lattice_SU3.py.

3.7.3.7 burn_in_sweeps

QCD_Lattice_SU3.burn_in_sweeps = 500

Definition at line 35 of file QCD_Lattice_SU3.py.

3.7.3.8 D

int QCD_Lattice_SU3.D = spatial_dims + 1

Definition at line 41 of file QCD_Lattice_SU3.py.

3.7.3.9 eps_initial

float QCD_Lattice_SU3.eps_initial = 0.06

Definition at line 34 of file QCD_Lattice_SU3.py.

3.7.3.10 eps_sub

QCD_Lattice_SU3.eps_sub

Definition at line 452 of file QCD_Lattice_SU3.py.

3.7.3.11 eps_tuned

QCD_Lattice_SU3.eps_tuned = tune_eps_su3(link_matrix, initial_eps=eps_initial)

Definition at line 450 of file QCD_Lattice_SU3.py.

3.7.3.12 F2_avg

QCD_Lattice_SU3.F2_avg

Definition at line 595 of file QCD_Lattice_SU3.py.

3.7.3.13 figsize

QCD_Lattice_SU3.figsize

Definition at line 458 of file QCD_Lattice_SU3.py.

3.7.3.14 lattice_size_L

int QCD_Lattice_SU3.lattice_size_L = 8

Definition at line 32 of file QCD_Lattice_SU3.py.

3.7.3.15 linestyle

QCD_Lattice_SU3.linestyle

Definition at line 459 of file QCD_Lattice_SU3.py.

3.7.3.16 link_matrix

QCD_Lattice_SU3.link_matrix = np.zeros((D,) + x_shape + (3, 3), dtype=np.complex128)

Definition at line 44 of file QCD_Lattice_SU3.py.

3.7.3.17 marker

QCD_Lattice_SU3.marker

Definition at line 459 of file QCD_Lattice_SU3.py.

3.7.3.18 max_R

QCD_Lattice_SU3.max_R

Definition at line 482 of file QCD_Lattice_SU3.py.

3.7.3.19 max_T

QCD_Lattice_SU3.max_T

Definition at line 482 of file QCD_Lattice_SU3.py.

3.7.3.20 MC_measure_interval

int QCD_Lattice_SU3.MC_measure_interval = 5

Definition at line 37 of file QCD_Lattice_SU3.py.

3.7.3.21 MC_sweeps

QCD_Lattice_SU3.MC_sweeps = 2000

Definition at line 36 of file QCD_Lattice_SU3.py.

3.7.3.22 n_boot

int QCD_Lattice_SU3.n_boot = 300

Definition at line 38 of file QCD_Lattice_SU3.py.

3.7.3.23 N_correlator

QCD_Lattice_SU3.N_correlator

Definition at line 453 of file QCD_Lattice_SU3.py.

3.7.3.24 plaq_err

QCD_Lattice_SU3.plaq_err

Definition at line 451 of file QCD_Lattice_SU3.py.

3.7.3.25 plaq_mean

QCD_Lattice_SU3.plaq_mean

Definition at line 451 of file QCD_Lattice_SU3.py.

3.7.3.26 potentials

list QCD_Lattice_SU3.potentials = []

Definition at line 489 of file QCD_Lattice_SU3.py.

3.7.3.27 R_values

QCD_Lattice_SU3.R_values

Definition at line 479 of file QCD_Lattice_SU3.py.

3.7.3.28 samples

QCD_Lattice_SU3.samples

Definition at line 451 of file QCD_Lattice_SU3.py.

3.7.3.29 spatial_dims

int QCD_Lattice_SU3.spatial_dims = 1

Definition at line 31 of file QCD_Lattice_SU3.py.

3.7.3.30 T_values

QCD_Lattice_SU3.T_values

Definition at line 479 of file QCD_Lattice_SU3.py.

3.7.3.31 V_R

QCD_Lattice_SU3.V_R = np.zeros(len(R_values))

Definition at line 487 of file QCD_Lattice_SU3.py.

3.7.3.32 W_T

QCD_Lattice_SU3.W_T = avg_wilson_loops[i, j]

Definition at line 491 of file QCD_Lattice_SU3.py.

3.7.3.33 W_Tp1

QCD_Lattice_SU3.W_Tp1 = avg_wilson_loops[i, j + 1]

Definition at line 492 of file QCD_Lattice_SU3.py.

3.7.3.34 wilson_loops_samples

QCD_Lattice_SU3.wilson_loops_samples

Definition at line 479 of file QCD_Lattice_SU3.py.

3.7.3.35 x_shape

tuple QCD_Lattice_SU3.x_shape = (lattice_size_L,) * D

Definition at line 42 of file QCD_Lattice_SU3.py.

Chapter 4

File Documentation

4.1 Lattice Field Theory.py File Reference

Namespaces

- namespace Lattice Field Theory
- namespace Lattice

  Field Theory.

Functions

- Lattice Field Theory.potential_at_site (phi_val)
- Lattice Field Theory.neighbor_index (idx, mu, shift=1)
- Lattice Field Theory.action_total (field)
- Lattice Field Theory.local_action_contribution (idx, field)
- Lattice Field Theory.metropolis_update_field (field, eps=0.5)
- Lattice Field Theory.measure_field_correlation_all_origins (field)
- Lattice Field Theory.compute_effective_mass (G, G_err=None)
- Lattice Field Theory.run_field_simulation (field, N_sweeps=2000, N_cor=10, eps=0.5, thermal_sweeps=500)
- Lattice Field Theory.run_multiple_simulations (num_runs, N_sweeps, N_cor, eps, thermal_sweeps)
- Lattice Field Theory.cosh_model (t, A, m)

Variables

- int Lattice Field Theory.d = 1
- float Lattice Field Theory.a = 1.0
- int Lattice Field Theory.L = 32
- float Lattice Field Theory.m = 1.0
- float Lattice Field Theory.lambda_ = 0.1
- int Lattice Field Theory.D = d + 1
- Lattice Field Theory.phi = np.zeros((L,) * D)
- float Lattice Field Theory.eps_tuned = 0.5
- Lattice Field Theory.G_mean = 0.5 * (G_mean + G_mean[::-1])
- Lattice Field Theory.G_err
- Lattice Field Theory.m_eff
- Lattice Field Theory.num_runs

- Lattice Field Theory.N_sweeps
- Lattice Field Theory.N_cor
- Lattice Field Theory.eps
- Lattice Field Theory.thermal_sweeps
- Lattice Field Theory.tdata = np.arange(L)
- int Lattice Field Theory.mask = G_mean > 5 * G_err
- Lattice Field Theory.A_fit
- Lattice Field Theory.m_fit
- Lattice Field Theory.p0
- Lattice Field Theory.G_plot = np.abs(G_mean)
- Lattice Field Theory.yerr
- Lattice Field Theory.fmt
- Lattice Field Theory.capsize

4.2 Lattice Field Theory.py

Go to the documentation of this file.

```
00001 """
00002 Lattice field theory simulation for a scalar    field in (d+1) dimensions.
00003
00004 Implements Metropolis Monte-Carlo updates for a real scalar field, computes
00005 correlation functions, extracts effective mass via logarithmic and cosh fits.
00006 Includes thermalization, autocorrelation spacing, and multi-run averaging.
00007 """
00008
00009 import numpy as np
00010 from scipy.optimize import curve_fit
00011 import itertools
00012 import matplotlib.pyplot as plt
00013
00014
00015 # --- Parameters ---
00016 d = 1
00017 """Spatial dimensions."""
00018
00019 a = 1.0
00020 """Lattice spacing."""
00021
00022 L = 32
00023 """Sites per spatial and temporal direction."""
00024
00025 m = 1.0
00026 """Bare mass parameter."""
00027
00028 lambda_ = 0.1
00029 """  interaction coupling. Set 0 for free-field tests."""
00030
00031 D = d + 1
00032 """Spacetime dimensions."""
00033
00034 phi = np.zeros((L,) * D)
00035 """Scalar field array; last axis = Euclidean time."""
00036
00037
00038 # --- Local physics functions ---
00039 def potential_at_site(phi_val):
00040     """
00041     Compute potential energy: V( ) = ½ m² ² +  /4   .
00042     """
00043     return 0.5 * m**2 * phi_val**2 + (lambda_/4.0) * phi_val**4
00044
00045
00046 def neighbor_index(idx, mu, shift=1):
00047     """
00048     Periodic-BC neighbor index shift by ±1 in direction  .
00049     """
00050     new = list(idx)
00051     new[mu] = (new[mu] + shift) % L
00052     return tuple(new)
00053
00054
00055 def action_total(field):
00056     """
00057     Full lattice action including kinetic nearest-neighbor term.
```

```
00058     """
00059     S = 0.0
00060     for idx in np.ndindex(*field.shape):
00061         phi_site = field[idx]
00062         S += potential_at_site(phi_site)
00063         for mu in range(field.ndim):
00064             neigh = neighbor_index(idx, mu, shift=1)
00065             diff = field[neigh] - phi_site
00066             S += 0.5 * (diff**2) / (a**2)
00067     return S
00068
00069
00070 def local_action_contribution(idx, field):
00071     """
00072     Local action contribution at a given lattice site incl. neighbors.
00073     Used for fast ΔS in Metropolis updates.
00074     """
00075     phi_site = field[idx]
00076     S = potential_at_site(phi_site)
00077     for mu in range(field.ndim):
00078         neigh_f = neighbor_index(idx, mu, shift=1)
00079         neigh_b = neighbor_index(idx, mu, shift=-1)
00080         S += 0.5 * (field[neigh_f] - phi_site)**2 / (a**2)
00081         S += 0.5 * (field[neigh_b] - phi_site)**2 / (a**2)
00082     return S
00083
00084
00085 # --- Metropolis updates ---
00086 def metropolis_update_field(field, eps=0.5):
00087     """
00088     Perform one Metropolis sweep; return accepted/proposed counts.
00089     """
00090     accepted = 0
00091     proposals = 0
00092     for idx in np.ndindex(*field.shape):
00093         old_val = field[idx]
00094         old_loc = local_action_contribution(idx, field)
00095
00096         new_val = old_val + np.random.uniform(-eps, eps)
00097         field[idx] = new_val
00098         new_loc = local_action_contribution(idx, field)
00099
00100         dS = new_loc - old_loc
00101         proposals += 1
00102         if dS > 0 and np.exp(-dS) < np.random.rand():
00103             field[idx] = old_val
00104         else:
00105             accepted += 1
00106     return accepted, proposals
00107
00108
00109 # --- Measurements ---
00110 def measure_field_correlation_all_origins(field):
00111     """
00112     Compute averaged two-point function G(Δt) over all origins.
00113     """
00114     Lt = field.shape[-1]
00115     spatial = field.shape[:-1]
00116     G = np.zeros(Lt)
00117     for dt in range(Lt):
00118         corr = 0.0
00119         count = 0
00120         for t0 in range(Lt):
00121             t1 = (t0 + dt) % Lt
00122             for idx in np.ndindex(*spatial):
00123                 corr += field[idx + (t0,)] * field[idx + (t1,)]
00124                 count += 1
00125         G[dt] = corr / count
00126     return G
00127
00128
00129 def compute_effective_mass(G, G_err=None):
00130     """
00131     Effective mass via m_eff(t) = log(G(t)/G(t+1)).
00132     Masked if too noisy.
00133     """
00134     m_eff = np.full(len(G) - 1, np.nan)
00135     for t in range(len(G) - 1):
00136         if G[t] > 0 and G[t+1] > 0:
00137             if G_err is not None and (G[t] < 2*G_err[t] or G[t+1] < 2*G_err[t+1]):
00138                 continue
00139             m_eff[t] = np.log(G[t] / G[t+1])
00140     return m_eff
00141
00142
00143 # --- Simulation driver ---
00144 def run_field_simulation(field, N_sweeps=2000, N_cor=10, eps=0.5, thermal_sweeps=500):
```

```
00145     """
00146     Run simulation: thermalize, measure correlations, return G and m_eff.
00147     """
00148     accepted = proposed = 0
00149     for _ in range(thermal_sweeps):
00150         a, p = metropolis_update_field(field, eps)
00151         accepted += a; proposed += p
00152     print(f"Post-thermalization acceptance: {accepted/proposed:.3f}")
00153 
00154     measurements = []
00155     accepted = proposed = 0
00156     for sweep in range(N_sweeps):
00157         a, p = metropolis_update_field(field, eps)
00158         accepted += a; proposed += p
00159         if sweep % N_cor == 0:
00160             measurements.append(measure_field_correlation_all_origins(field))
00161     print(f"Measurement acceptance: {accepted/proposed:.3f}")
00162 
00163     meas_arr = np.array(measurements)
00164     G_mean = np.mean(meas_arr, axis=0)
00165     G_err = np.std(meas_arr, axis=0, ddof=1)
00166     m_eff = compute_effective_mass(G_mean, G_err)
00167     return meas_arr, G_mean, G_err, m_eff
00168 
00169 
00170 def run_multiple_simulations(num_runs, N_sweeps, N_cor, eps, thermal_sweeps):
00171     """
00172     Run multiple independent simulations and average observed correlators.
00173     """
00174     all_meas = []
00175     for _ in range(num_runs):
00176         field = 0.01 * np.random.randn(*(L,) * D)
00177         meas_arr, *_ = run_field_simulation(field, N_sweeps, N_cor, eps, thermal_sweeps)
00178         all_meas.append(meas_arr)
00179 
00180     all_meas = np.vstack(all_meas)
00181     G_mean = np.mean(all_meas, axis=0)
00182     G_err = np.std(all_meas, axis=0, ddof=1) / np.sqrt(all_meas.shape[0])
00183     m_eff = compute_effective_mass(G_mean, G_err)
00184     return G_mean, G_err, m_eff
00185 
00186 
00187 # --- Analysis & visualization ---
00188 eps_tuned = 0.5
00189 G_mean, G_err, m_eff = run_multiple_simulations(
00190     num_runs=5, N_sweeps=4000, N_cor=20, eps=eps_tuned, thermal_sweeps=1000
00191 )
00192 
00193 print("G_mean:", G_mean)
00194 
00195 def cosh_model(t, A, m):
00196     return A * (np.exp(-m*t) + np.exp(-m*(L - t)))
00197 
00198 G_mean = 0.5 * (G_mean + G_mean[::-1])  # symmetrize
00199 
00200 tdata = np.arange(L)
00201 mask = G_mean > 5 * G_err
00202 A_fit, m_fit = curve_fit(cosh_model, tdata[mask], G_mean[mask], p0=[0.2, 1.0])[0]
00203 print(f"Fitted mass m = {m_fit:.3f}")
00204 
00205 G_plot = np.abs(G_mean)
00206 plt.errorbar(tdata, G_plot, yerr=G_err, fmt='o', capsize=3)
00207 plt.yscale('log')
00208 plt.ylim(1e-5, 1e0)
00209 plt.xlabel('Euclidean time t')
00210 plt.ylabel('G(t)')
00211 plt.title('Two-point correlation function')
00212 plt.grid(True)
00213 plt.show()
00214 
00215 plt.plot(np.arange(len(m_eff)), m_eff, 'o-')
00216 plt.xlabel('t')
00217 plt.ylabel('m_eff(t)')
00218 plt.title('Effective mass vs time')
00219 plt.grid(True)
00220 plt.show()
```

4.3 Monte Carlo integral.py File Reference

Namespaces

- namespace Monte Carlo integral

- namespace Monte
  Carlo integral.

Functions

- Monte Carlo integral.potential_energy_V (position, lambda_parameter)
- Monte Carlo integral.local_action (x_prev, x_current, x_next, lambda_parameter)
- Monte Carlo integral.delta_action_change (x_path, j, x_new, lambda_parameter)
- Monte Carlo integral.run_monte_carlo (lambda_parameter)

Variables

- float Monte Carlo integral.particle_mass = 1.0
- float Monte Carlo integral.lattice_spacing_a = 0.1
- int Monte Carlo integral.number_of_sites = 100
- int Monte Carlo integral.total_monte_carlo_steps = 30000
- int Monte Carlo integral.thermalization_steps = 5000
- float Monte Carlo integral.proposal_step_size = 0.5
- list Monte Carlo integral.anharmonicity_values = [0.0, 0.1, 0.3, 0.5, 1.0]
- list Monte Carlo integral.estimated_ground_energies = [ ]
- Monte Carlo integral.correlation_function
- Monte Carlo integral.acceptance_rate
- float Monte Carlo integral.euclidean_times = np.arange(len(correlation_function)) * lattice_spacing_a
- Monte Carlo integral.fit_slice = slice(1, 6)
- Monte Carlo integral.slope
- Monte Carlo integral.intercept
- Monte Carlo integral.estimated_E0 = -slope
- Monte Carlo integral.lw

4.4 Monte Carlo integral.py

Go to the documentation of this file.

```
00001 """
00002 @mainpage Monte Carlo Simulation for the 1D Quantum Anharmonic Oscillator
00003 @file monte_carlo_anharmonic.py
00004 @brief Monte Carlo Euclidean path-integral estimator for the ground-state energy E .
00005
00006 @details
00007 This script estimates the ground-state energy of a 1D quantum anharmonic oscillator
00008 with Euclidean action
00009
00010     S_E = Σ_j [ m/(2a) (x_{j+1}-x_j)² + a V(x_j) ],
00011
00012 where
00013     V(x) = 1/2 x² +   x .
00014
00015 Configurations of x( ) are sampled using local Metropolis updates with periodic
00016 boundary conditions. The two-point correlator
00017
00018     C( ) =  x(0)x( )    e^(−E   ),
00019
00020 is accumulated and fitted to a single exponential in the plateau region to extract E .
00021 """
00022
00023 import numpy as np
00024 import matplotlib.pyplot as plt
00025
00026 # ============================================================
00027 #                Physical and Simulation Parameters
00028 # ============================================================
00029
00030 particle_mass = 1.0
```

```
00031 #: float: mass of particle (in natural units   = 1).
00032
00033 lattice_spacing_a = 0.1
00034 #: float: Euclidean lattice spacing a = Δ .
00035
00036 number_of_sites = 100
00037 #: int: number of lattice sites (total imaginary time extent T = N · a).
00038
00039 total_monte_carlo_steps = 30000
00040 #: int: total Metropolis sweeps (equilibration + measurement).
00041
00042 thermalization_steps = 5000
00043 #: int: number of initial sweeps discarded before measurements.
00044
00045 proposal_step_size = 0.5
00046 #: float: amplitude of uniform proposal displacement for local updates.
00047
00048 anharmonicity_values = [0.0, 0.1, 0.3, 0.5, 1.0]
00049 #: list(float): list of anharmonicity parameters   to simulate.
00050
00051 np.random.seed(42)  # reproducibility
00052
00053
00054 # ================================================================
00055 #               Potential Energy Function
00056 # ================================================================
00057
00058 def potential_energy_V(position, lambda_parameter):
00059     """
00060     @brief Anharmonic potential energy V(x).
00061     @param position float or ndarray: spatial coordinate(s) x.
00062     @param lambda_parameter float: anharmonicity   ( → 0 gives harmonic limit).
00063     @return float or ndarray: potential energy V(x).
00064     @details
00065     Implements:
00066         V(x) = 1/2 x² +   x .
00067     """
00068     return 0.5 * position**2 + lambda_parameter * position**4
00069
00070
00071 # ================================================================
00072 #         Local Euclidean Action Contributions
00073 # ================================================================
00074
00075 def local_action(x_prev, x_current, x_next, lambda_parameter):
00076     """
00077     @brief Local contribution to discretized Euclidean action around site j.
00078     @param x_prev float: x at site j−1.
00079     @param x_current float: x at site j.
00080     @param x_next float: x at site j+1.
00081     @param lambda_parameter float: anharmonicity  .
00082     @return float: local action S_E(j).
00083     @details
00084     Uses symmetric discretized kinetic term:
00085         S_kin(j) = m/(4a)[(x_{j+1}-x_j)² + (x_j - x_{j-1})²]
00086     plus potential:
00087         S_pot(j) = a V(x_j).
00088     Periodic boundary conditions handled externally.
00089     """
00090     S_kinetic_local = 0.5 * particle_mass / lattice_spacing_a * \
00091                       ((x_next - x_current)**2 + (x_current - x_prev)**2) / 2
00092     S_potential_local = lattice_spacing_a * potential_energy_V(x_current, lambda_parameter)
00093     return S_kinetic_local + S_potential_local
00094
00095
00096 def delta_action_change(x_path, j, x_new, lambda_parameter):
00097     """
00098     @brief Compute local change ΔS_E for a proposed update at site j.
00099     @param x_path ndarray(float): full current path configuration.
00100     @param j int: lattice site index for update.
00101     @param x_new float: proposed new value for x[j].
00102     @param lambda_parameter float: anharmonicity  .
00103     @return float: ΔS = S_new − S_old.
00104     @details
00105     Only the action terms involving sites {j−1, j, j+1} contribute to ΔS.
00106     Indices wrap via periodic boundary conditions.
00107     """
00108     j_minus = (j - 1) % number_of_sites
00109     j_plus = (j + 1) % number_of_sites
00110
00111     S_old = local_action(x_path[j_minus], x_path[j], x_path[j_plus], lambda_parameter)
00112     S_new = local_action(x_path[j_minus], x_new, x_path[j_plus], lambda_parameter)
00113
00114     return S_new - S_old
00115
00116
00117 # ================================================================
```

```
00118 #         Monte Carlo Simulation for Given
00119 # ======================================================================
00120
00121 def run_monte_carlo(lambda_parameter):
00122     """
00123     @brief Perform local Metropolis updates to sample Euclidean paths for fixed  .
00124     @param lambda_parameter float: anharmonicity value for this simulation.
00125     @return tuple: (C_tau, acceptance_fraction)
00126         - C_tau: ndarray(float) correlator C( ) for   up to T/2
00127         - acceptance_fraction: overall acceptance rate of updates
00128     @details
00129     • Initializes x( )=0 path
00130     • Local updates at every site each sweep
00131     • Observables recorded every 10 steps post-thermalization
00132     • Correlator estimator:
00133             C( ) =   x_j x_{j+ }   averaged over j and Monte Carlo samples
00134     """
00135     x_path = np.zeros(number_of_sites)
00136     G_correlator = np.zeros(number_of_sites // 2)
00137     N_measure = 0
00138     accepted_updates = 0
00139
00140     for monte_carlo_step in range(total_monte_carlo_steps):
00141         for j in range(number_of_sites):
00142             x_new = x_path[j] + np.random.uniform(-proposal_step_size, proposal_step_size)
00143             delta_S_local = delta_action_change(x_path, j, x_new, lambda_parameter)
00144
00145             if delta_S_local < 0 or np.exp(-delta_S_local) > np.random.rand():
00146                 x_path[j] = x_new
00147                 accepted_updates += 1
00148
00149         if monte_carlo_step >= thermalization_steps and monte_carlo_step % 10 == 0:
00150             for t_index in range(number_of_sites // 2):
00151                 G_correlator[t_index] += np.mean(x_path * np.roll(x_path, -t_index))
00152             N_measure += 1
00153
00154     G_correlator /= N_measure
00155     acceptance_fraction = accepted_updates / (total_monte_carlo_steps * number_of_sites)
00156     return G_correlator, acceptance_fraction
00157
00158
00159 # ======================================================================
00160 #         Main Loop — Extract Ground-State Energy
00161 # ======================================================================
00162
00163 estimated_ground_energies = []
00164 #: list(float): extracted ground-state energies E ( ) from exponential fits.
00165
00166 for lambda_parameter in anharmonicity_values:
00167     correlation_function, acceptance_rate = run_monte_carlo(lambda_parameter)
00168
00169     correlation_function /= correlation_function[0]
00170     euclidean_times = np.arange(len(correlation_function)) * lattice_spacing_a
00171
00172     fit_slice = slice(1, 6)  # small   region where log(C) approx linear
00173     slope, intercept = np.polyfit(euclidean_times[fit_slice],
00174                                   np.log(correlation_function[fit_slice]), 1)
00175     estimated_E0 = -slope
00176     estimated_ground_energies.append(estimated_E0)
00177
00178     print(f"  = {lambda_parameter:.2f} | Estimated E0 = {estimated_E0:.4f} "
00179           f"| Acceptance = {acceptance_rate:.3f}")
00180
00181
00182 # ======================================================================
00183 #                   Visualization
00184 # ======================================================================
00185
00186 plt.plot(anharmonicity_values, estimated_ground_energies, 'o-', lw=2)
00187 plt.xlabel("Anharmonicity Parameter  ")
00188 plt.ylabel("Estimated Ground-State Energy $E_0$")
00189 plt.title("Quantum Anharmonic Oscillator: Ground-State Energy vs  ")
00190 plt.grid(True)
00191 plt.show()
```

4.5 Path integral.py File Reference

Namespaces

- namespace Path integral
- namespace Path

  integral

Functions

- Path integral.potential_V (x)
- Path integral.S_lat (x_list, x_fixed, *args)
- Path integral.integrand (*x_list)

Variables

- int Path integral.N = 4
- int Path integral.lattice_spacing_a = 1 / 2
- float Path integral.particle_mass = 1.0
- int Path integral.bound_limit = 5
- list Path integral.bounds = [(-bound_limit, bound_limit)] * (N - 1)
- list Path integral.propagator = []
- tuple Path integral.normalization_A = (particle_mass / (2 * math.pi * lattice_spacing_a)) ** (N / 2)
- list Path integral.x_values = [i * 0.25 for i in range(-10, 11)]
- Path integral.result
- Path integral.error

4.6 Path integral.py

Go to the documentation of this file.

```
00001 """
00002 @mainpage Numerical Euclidean Path Integral for the Harmonic Oscillator
00003 @file path_integral.py
00004 @brief Brute-force multidimensional Euclidean path integral evaluation.
00005
00006 @details
00007 This demonstration numerically evaluates the diagonal propagator
00008 K(x, x; T) = ⟨x| e^(−H T) |x⟩
00009 for a 1D harmonic oscillator using a discretized Euclidean action:
00010
00011     S_E = Σ_j [ m/(2a) (x_{j+1} − x_j)² + a V(x_j) ],
00012
00013 with fixed endpoints x_0 = x_N = x_fixed and (N−1) internal lattice points
00014 integrated over a finite domain. The integral is computed using
00015 SciPy's multi-dimensional quadrature (`nquad`), which scales
00016 exponentially with N and serves only as a pedagogical reference
00017 (not an efficient Monte Carlo method).
00018 """
00019
00020 import math
00021 import matplotlib.pyplot as plt
00022 from scipy.integrate import nquad
00023
00024 # ============================================================
00025 #             Physical and Discretization Parameters
00026 # ============================================================
00027
00028 N = 4
00029 #: int: number of lattice sites including fixed endpoints (integration dimension = N−1).
00030
00031 lattice_spacing_a = 1 / 2
00032 #: float: Euclidean time spacing, total extent T = N · a.
00033
00034 particle_mass = 1.0
00035 #: float: mass of particle (natural units ℏ = 1).
00036
00037 bound_limit = 5
00038 #: float: |x| integration domain bound for intermediate positions.
00039
00040 bounds = [(-bound_limit, bound_limit)] * (N - 1)
00041 #: list(tuple): integration bounds for each of the (N−1) intermediate coordinates.
00042
00043 propagator = []
00044 #: list(float): evaluated propagator K(x, x; T) at each fixed endpoint x.
00045
00046 normalization_A = (particle_mass / (2 * math.pi * lattice_spacing_a)) ** (N / 2)
```

```
00047 #: float: Gaussian normalization prefactor from discretized measure.
00048
00049
00050 # ==================================================================
00051 #                        Potential Energy
00052 # ==================================================================
00053
00054 def potential_V(x):
00055     """
00056     @brief Harmonic oscillator potential.
00057     @param x float: position value.
00058     @return float: potential V(x) = 1/2 x².
00059     """
00060     return 0.5 * x**2
00061
00062
00063 # ==================================================================
00064 #                         Euclidean Action
00065 # ==================================================================
00066
00067 def S_lat(x_list, x_fixed, *args):
00068     """
00069     @brief Compute discretized Euclidean path action.
00070     @param x_list list(float): internal coordinates, length (N−1).
00071     @param x_fixed float: fixed boundary value x  = x_N.
00072     @return float: Euclidean action S_E for given path.
00073     @details
00074     Constructs full path:
00075         x = [x_fixed, x , x , ..., x_{N−1}, x_fixed]
00076     and applies:
00077         S_E = Σ_j m/(2a)(x_{j+1} − x_j)² + a V(x_j)
00078     without periodic BCs since endpoints are fixed.
00079     """
00080     x = [x_fixed] + list(x_list) + [x_fixed]
00081     Action_S = 0
00082     for j in range(0, N - 1):
00083         x_derivative = x[j + 1] - x[j]
00084         Action_S += (particle_mass / (2 * lattice_spacing_a)) * x_derivative**2 \
00085                 + lattice_spacing_a * potential_V(x[j])
00086     return Action_S
00087
00088
00089 # ==================================================================
00090 #              Propagator Evaluation Loop
00091 # ==================================================================
00092
00093 x_values = [i * 0.25 for i in range(-10, 11)]
00094 #: list(float): fixed endpoint values used to evaluate K(x, x; T).
00095
00096 for x_fixed in x_values:
00097
00098     def integrand(*x_list):
00099         """
00100         @brief Integrand exp(−S_E[x]) for numerical quadrature.
00101         @param x_list variadic float: internal lattice points.
00102         @return float: value of exp(−S_E).
00103         @details
00104         This closure captures `x_fixed` from the loop scope.
00105         """
00106         return math.exp(-S_lat(x_list, x_fixed))
00107
00108     result, error = nquad(integrand, bounds)
00109     propagator.append(normalization_A * result)
00110
00111
00112 # ==================================================================
00113 #                        Visualization
00114 # ==================================================================
00115
00116 plt.plot(x_values, propagator)
00117 plt.xlabel("Fixed endpoint position x")
00118 plt.ylabel("Normalized propagator K(x, x; T)")
00119 plt.title(f"Numerical Path Integral (N={N})")
00120 plt.grid(True)
00121 plt.show()
```

4.7 QCD_Lattice_SU3.py File Reference

Namespaces

- namespace QCD_Lattice_SU3

Functions

- QCD_Lattice_SU3.x_neighbor (x, mu, shift=1)
- QCD_Lattice_SU3.su3_matrices (M)
- QCD_Lattice_SU3.su2_random_unitary (eps)
- QCD_Lattice_SU3.embed_su2_into_su3 (R2, i, j)
- QCD_Lattice_SU3.plaquette_matrix (x, mu, nu, link_sites)
- QCD_Lattice_SU3.real_trace_plaquette (x, mu, nu, link_sites)
- QCD_Lattice_SU3.plaquettes_touching_link (x, mu, link_sites)
- QCD_Lattice_SU3.metropolis_update (link_sites, eps_sub=0.06)
- QCD_Lattice_SU3.average_plaquette_su3 (link_sites)
- QCD_Lattice_SU3.bootstrap_mean_std (values, nboot=300)
- QCD_Lattice_SU3.tune_eps_su3 (matrix0, target=0.5, initial_eps=0.06, tries=10, test_↩ sweeps=150)
- QCD_Lattice_SU3.run_su3_simulation (link_sites, eps_sub=0.06, burn_in_sweeps=500, MC_sweeps=2000, N_correlator=5)
- QCD_Lattice_SU3.init_links_identity (link_sites)
- QCD_Lattice_SU3.randomize_links_small (link_sites, amplitude=0.02)
- QCD_Lattice_SU3.measure_wilson_loop_RT (link_sites, R, T, spatial_direction=0, time_↩ direction=None)
- QCD_Lattice_SU3.su3_simulation_with_wilson_loops (link_sites, eps_sub=0.06, burn_in_sweeps=500, MC_sweeps=2000, N_correlator=5, max_R=None, max_T=None)
- QCD_Lattice_SU3.gell_mann_matrices ()
- QCD_Lattice_SU3.extract_gluon_field (U, g=1.0, a=1.0)
- QCD_Lattice_SU3.field_strength_tensor (link_sites, x, mu, nu, g=1.0, a=1.0)
- QCD_Lattice_SU3.measure_avg_A2_and_F2 (link_sites, g=1.0, a=1.0)

Variables

- int QCD_Lattice_SU3.spatial_dims = 1
- int QCD_Lattice_SU3.lattice_size_L = 8
- float QCD_Lattice_SU3.beta = 6.0
- float QCD_Lattice_SU3.eps_initial = 0.06
- int QCD_Lattice_SU3.burn_in_sweeps = 500
- int QCD_Lattice_SU3.MC_sweeps = 2000
- int QCD_Lattice_SU3.MC_measure_interval = 5
- int QCD_Lattice_SU3.n_boot = 300
- int QCD_Lattice_SU3.D = spatial_dims + 1
- tuple QCD_Lattice_SU3.x_shape = (lattice_size_L,) ∗ D
- QCD_Lattice_SU3.link_matrix = np.zeros((D,) + x_shape + (3, 3), dtype=np.complex128)
- QCD_Lattice_SU3.amplitude
- QCD_Lattice_SU3.eps_tuned = tune_eps_su3(link_matrix, initial_eps=eps_initial)
- QCD_Lattice_SU3.samples
- QCD_Lattice_SU3.plaq_mean
- QCD_Lattice_SU3.plaq_err
- QCD_Lattice_SU3.eps_sub
- QCD_Lattice_SU3.N_correlator
- QCD_Lattice_SU3.figsize
- QCD_Lattice_SU3.marker
- QCD_Lattice_SU3.linestyle
- QCD_Lattice_SU3.bins
- QCD_Lattice_SU3.alpha
- QCD_Lattice_SU3.wilson_loops_samples
- QCD_Lattice_SU3.R_values

- QCD_Lattice_SU3.T_values
- QCD_Lattice_SU3.max_R
- QCD_Lattice_SU3.max_T
- QCD_Lattice_SU3.avg_wilson_loops = np.mean(wilson_loops_samples, axis=0)
- QCD_Lattice_SU3.V_R = np.zeros(len(R_values))
- list QCD_Lattice_SU3.potentials = [ ]
- QCD_Lattice_SU3.W_T = avg_wilson_loops[i, j]
- QCD_Lattice_SU3.W_Tp1 = avg_wilson_loops[i, j + 1]
- QCD_Lattice_SU3.A2_avg
- QCD_Lattice_SU3.F2_avg

4.8 QCD_Lattice_SU3.py

Go to the documentation of this file.

```
00001 """
00002 @mainpage SU(3) Lattice Gauge Theory (Wilson action) — Cabibbo–Marinari Implementation
00003
00004 @file QCD_Lattice_SU3.py
00005 @brief SU(3) lattice gauge theory simulation using Cabibbo–Marinari SU(2) subgroup updates.
00006 @details
00007 This module implements a pragmatic SU(3) lattice gauge theory code based on the Wilson action,
00008 using Cabibbo–Marinari updates (embedded SU(2) rotations) together with local ΔS computations
00009 (only plaquettes touching a link are recomputed) and reprojection to SU(3) via SVD to maintain
00010 unitarity and $\det U = 1$.
00011
00012 It contains:
00013  - Local Metropolis updates applying a sequence of small SU(2) rotations embedded into SU(3).
00014  - Efficient local plaquette recomputation for ΔS evaluations.
00015  - Utilities for plaquette measurement, bootstrap error estimation, Wilson loops and static potential.
00016  - Helpers to extract approximate gauge fields (A^a) and field-strength components F^{\mu\nu}_a
00017    from link matrices for diagnostic/classical analysis.
00018
00019 @section references Key references
00020  - K. G. Wilson, "Confinement of quarks," Phys. Rev. D 10, 2445 (1974).
00021  - N. Cabibbo and E. Marinari, "A new method for updating SU(N) matrices," Phys. Lett. B119 (1982).
00022  - G. P. Lepage lecture notes for pragmatic algorithmic choices.
00023 """
00024
00025 import numpy as np
00026 from numpy import linalg as LA
00027 import matplotlib.pyplot as plt
00028
00029
00030 # ---------------- Parameters (global simulation controls) ----------------
00031 spatial_dims = 1            #: int: number of spatial dimensions (d)
00032 lattice_size_L = 8          #: int: lattice extent per dimension (L)
00033 beta = 6.0                  #: float: gauge coupling parameter (Wilson action:  = 6/g²)
00034 eps_initial = 0.06          #: float: initial SU(2) rotation amplitude for subgroup updates
00035 burn_in_sweeps = 500        #: int: number of thermalization sweeps
00036 MC_sweeps = 2000            #: int: number of Monte Carlo sweeps for measurements
00037 MC_measure_interval = 5     #: int: sweeps between stored measurements (decorrelation interval)
00038 n_boot = 300                #: int: bootstrap samples for error estimation
00039
00040 # Derived geometry / storage
00041 D = spatial_dims + 1        #: int: total spacetime dimensions (d + 1)
00042 x_shape = (lattice_size_L,) * D
00043 # link_matrix: shape (D, L, L, ..., 3, 3) storing SU(3) link matrices for each direction mu and site x
00044 link_matrix = np.zeros((D,) + x_shape + (3, 3), dtype=np.complex128)
00045
00046
00047 # ---------------- Utilities ----------------
00048 def x_neighbor(x, mu, shift=1):
00049     """
00050     @brief Periodic lattice neighbor coordinate.
00051     @param x tuple: Lattice coordinate (length D).
00052     @param mu int: Direction index (0..D-1).
00053     @param shift int: Integer shift (positive forward, negative backward).
00054     @return tuple: New lattice coordinate (with periodic wrap).
00055     @details Implements periodic boundary conditions: (x_mu + shift) mod L.
00056     """
00057     x_new = list(x)
00058     x_new[mu] = (x_new[mu] + shift) % lattice_size_L
00059     return tuple(x_new)
00060
00061
```

```
00062 def su3_matrices(M):
00063     """
00064     @brief Project a general complex 3x3 matrix to SU(3) via unitary polar/SVD projection.
00065     @param M (ndarray): 3x3 complex matrix (candidate link).
00066     @return ndarray: Unitary 3x3 matrix with det = 1 (projection of M into SU(3)).
00067     @details
00068     We perform an SVD: M = U S V^H and set U_proj = U V^H (closest unitary in Frobenius norm).
00069     A global phase is then removed to enforce det(U_proj) = 1. If the projection yields
00070     a near-singular matrix we add a tiny perturbation as fallback.
00071     """
00072     U, s, Vh = LA.svd(M)
00073     U_projection = U @ Vh
00074     determinant = LA.det(U_projection)
00075     if determinant == 0 or np.isnan(determinant):
00076         # Numerical fallback: small perturbation then reproject
00077         U_projection = U_projection + 1e-12 * np.eye(3, dtype=complex)
00078         determinant = LA.det(U_projection)
00079     # Remove global phase to ensure unit determinant
00080     phase = determinant ** (1.0 / 3.0)
00081     U_projection /= phase
00082     return U_projection
00083 
00084 
00085 # ---------- SU(2) small updater (embedded in SU(3)) ----------
00086 def su2_random_unitary(eps):
00087     """
00088     @brief Generate a small random SU(2) rotation matrix using Gaussian parameters.
00089     @param eps float: amplitude controlling rotation angle scale (a = eps * |r|).
00090     @return ndarray: 2x2 complex SU(2) matrix.
00091     @details
00092     The parametrization uses R = cos(a) I + i sin(a) n ·  where n is a unit 3-vector
00093     and  _i are the Pauli matrices. We project via SVD to correct numerical drift and
00094     ensure exact unitarity, then enforce det=1.
00095     """
00096     r = np.random.normal(size=3)
00097     r_norm = np.linalg.norm(r)
00098     if r_norm == 0:
00099         return np.eye(2, dtype=complex)
00100     a = eps * r_norm
00101     n = r / r_norm
00102     # Pauli matrices
00103     sigma1 = np.array([[0.0, 1.0], [1.0, 0.0]], dtype=complex)
00104     sigma2 = np.array([[0.0, -1j], [1j, 0.0]], dtype=complex)
00105     sigma3 = np.array([[1.0, 0.0], [0.0, -1.0]], dtype=complex)
00106     ndotsigma = n[0] * sigma1 + n[1] * sigma2 + n[2] * sigma3
00107     R = np.cos(a) * np.eye(2, dtype=complex) + 1j * np.sin(a) * ndotsigma
00108     # Project R to exact SU(2) via SVD/polar projection and fix determinant
00109     U, s, Vh = LA.svd(R)
00110     R_projection = U @ Vh
00111     det = LA.det(R_projection)
00112     R_projection /= (det ** 0.5)
00113     return R_projection
00114 
00115 
00116 def embed_su2_into_su3(R2, i, j):
00117     """
00118     @brief Embed a 2x2 SU(2) matrix into SU(3) acting on indices (i, j).
00119     @param R2 ndarray: 2x2 SU(2) matrix.
00120     @param i int: first SU(2) index (0..2).
00121     @param j int: second SU(2) index (0..2), must satisfy i < j.
00122     @return ndarray: 3x3 matrix equal to identity except the 2x2 block at (i,j) replaced by R2.
00123     @details This is the standard Cabibbo–Marinari embedding that extends SU(2) subgroup rotations
00124              to SU(3) by acting non-trivially on a chosen 2D subspace.
00125     """
00126     R = np.eye(3, dtype=complex)
00127     R[i, i] = R2[0, 0]
00128     R[i, j] = R2[0, 1]
00129     R[j, i] = R2[1, 0]
00130     R[j, j] = R2[1, 1]
00131     return R
00132 
00133 
00134 # ---------- Plaquette helpers & local ΔS computation ----------
00135 def plaquette_matrix(x, mu, nu, link_sites):
00136     """
00137     @brief Construct the plaquette matrix U_mu(x) U_nu(x+mu) U_mu^\dagger(x+nu) U_nu^\dagger(x).
00138     @param x tuple: lattice coordinate.
00139     @param mu int: direction index mu.
00140     @param nu int: direction index nu.
00141     @param link_sites ndarray: link variable array.
00142     @return ndarray: 3x3 plaquette matrix P_{mu,nu}(x).
00143     """
00144     x_plus_mu = x_neighbor(x, mu, 1)
00145     x_plus_nu = x_neighbor(x, nu, 1)
00146     U_mu = link_sites[(mu,) + x]
00147     U_nu_xmu = link_sites[(nu,) + x_plus_mu]
00148     U_mu_xnu = link_sites[(mu,) + x_plus_nu]
```

```
00149     U_nu = link_sites[(nu,) + x]
00150     P = U_mu @ U_nu_xmu @ U_mu_xnu.conj().T @ U_nu.conj().T
00151     return P
00152
00153
00154 def real_trace_plaquette(x, mu, nu, link_sites):
00155     """
00156     @brief Compute the real part of the trace of the plaquette matrix.
00157     @return float: Re Tr[P_{mu,nu}(x)].
00158     """
00159     P = plaquette_matrix(x, mu, nu, link_sites)
00160     trace = np.trace(P)
00161     return float(np.real(trace))
00162
00163
00164 def plaquettes_touching_link(x, mu, link_sites):
00165     """
00166     @brief List plaquettes that include the link at (mu, x).
00167     @param x tuple: lattice coordinate of the starting site of the link.
00168     @param mu int: link direction.
00169     @param link_sites ndarray: array of link matrices.
00170     @return list: entries [((x_plaq, mu, nu), real_trace), ...] for all plaquettes touching the link.
00171     @details
00172     For each nu != mu, the link (mu,x) sits in two elementary plaquettes:
00173       - the plaquette at x in the (mu,nu) plane,
00174       - the plaquette at x - e_nu in the (mu,nu) plane.
00175     Only these plaquettes are required to compute the local change in action when U_mu(x) is updated.
00176     """
00177     p_list = []
00178     for nu in range(D):
00179         if nu == mu:
00180             continue
00181         trace_1 = real_trace_plaquette(x, mu, nu, link_sites)
00182         p_list.append(((x, mu, nu), trace_1))
00183         x_minus_nu = x_neighbor(x, nu, -1)
00184         trace_2 = real_trace_plaquette(x_minus_nu, mu, nu, link_sites)
00185         p_list.append(((x_minus_nu, mu, nu), trace_2))
00186     return p_list
00187
00188
00189 # ---------- Local update: Metropolis with embedded SU(2) updates ----------
00190 def metropolis_update(link_sites, eps_sub=0.06):
00191     """
00192     @brief Perform a single Metropolis sweep over all links applying embedded SU(2) updates.
00193     @param link_sites ndarray: link variable array (modified in-place).
00194     @param eps_sub float: SU(2) proposal amplitude for each embedded sub-update.
00195     @return tuple: (accepted int, proposals int)
00196     @details
00197     For each link U_mu(x) we cycle through the three SU(2) subgroups (0,1), (0,2), (1,2).
00198     For each subgroup:
00199       1. compute sum_old = Σ Re Tr(P) over plaquettes touching the link,
00200       2. propose an SU(2) rotation R2, embed into SU(3) → R3,
00201       3. set U_candidate = R3 @ U_old and reproject to SU(3),
00202       4. compute sum_new and ΔS = - ( /3) (sum_new - sum_old),
00203       5. accept/reject with Metropolis probability.
00204     Using only touching plaquettes makes ΔS computation local and efficient.
00205     """
00206     accepted = 0
00207     proposals = 0
00208     su2_pairs = [(0, 1), (0, 2), (1, 2)]
00209     for mu in range(D):
00210         for x in np.ndindex(*x_shape):
00211             U_old = link_sites[(mu,) + x].copy()
00212             for (i, j) in su2_pairs:
00213                 plist = plaquettes_touching_link(x, mu, link_sites)
00214                 sum_old = sum(trace for (_meta, trace) in plist)
00215
00216                 R2 = su2_random_unitary(eps_sub)
00217                 R3 = embed_su2_into_su3(R2, i, j)
00218                 link_sites[(mu,) + x] = R3 @ U_old
00219                 # Reproject to SU(3) to correct numerical drift
00220                 link_sites[(mu,) + x] = su3_matrices(link_sites[(mu,) + x])
00221
00222                 new_p_list = plaquettes_touching_link(x, mu, link_sites)
00223                 sum_new = sum(trace for (_meta, trace) in new_p_list)
00224
00225                 dS = - (beta / 3.0) * (sum_new - sum_old)
00226                 proposals += 1
00227                 # Metropolis acceptance: accept if dS <= 0 or with probability exp(-dS)
00228                 if dS > 0 and np.exp(-dS) < np.random.rand():
00229                     # reject: revert this subgroup update (resume next subgroup from U_old)
00230                     link_sites[(mu,) + x] = U_old.copy()
00231                 else:
00232                     # accept: update U_old so subsequent subgroup multiplications act on accepted matrix
00233                     U_old = link_sites[(mu,) + x].copy()
00234                     accepted += 1
00235     return accepted, proposals
```

```
00236
00237
00238 # ---------- Observables ----------
00239 def average_plaquette_su3(link_sites):
00240     """
00241     @brief Compute the normalized average plaquette  Re Tr P /3 over the lattice.
00242     @param link_sites ndarray: link array.
00243     @return float: average plaquette normalized by color factor (3).
00244     @details The Wilson action density per plaquette is proportional to (1 - Re Tr P / 3).
00245     """
00246     total = 0.0
00247     count = 0
00248     for x in np.ndindex(*x_shape):
00249         for mu in range(D):
00250             for nu in range(mu + 1, D):
00251                 trace = real_trace_plaquette(x, mu, nu, link_sites)
00252                 total += trace
00253                 count += 1
00254     # Normalize by color dimension (Tr 1 = 3)
00255     return (total / count) / 3.0
00256
00257
00258 def bootstrap_mean_std(values, nboot=300):
00259     """
00260     @brief Estimate mean and bootstrap standard error for a 1D array of samples.
00261     @param values array-like: measurement samples.
00262     @param nboot int: number of bootstrap resamples.
00263     @return tuple: (boot_mean, boot_std)
00264     @details We resample with replacement and compute sample means for each bootstrap
00265              realization; the returned std is the bootstrap estimate of the error.
00266     """
00267     vals = np.asarray(values)
00268     N = len(vals)
00269     boots = np.zeros(nboot)
00270     for i in range(nboot):
00271         inds = np.random.randint(0, N, size=N)
00272         boots[i] = np.mean(vals[inds])
00273     return boots.mean(), boots.std(ddof=1)
00274
00275
00276 # ---------- Tuner & Runner ----------
00277 def tune_eps_su3(matrix0, target=0.5, initial_eps=0.06, tries=10, test_sweeps=150):
00278     """
00279     @brief Tune the SU(2) proposal amplitude eps so that acceptance fraction ~ target.
00280     @param matrix0 ndarray: initial link matrix copy for tuning (will be copied internally).
00281     @param target float: desired acceptance fraction (e.g., 0.5).
00282     @param initial_eps float: starting amplitude.
00283     @param tries int: maximum adjustment attempts.
00284     @param test_sweeps int: sweeps per tuning test.
00285     @return float: tuned eps value.
00286     @details We perform a small number of sweeps and adjust eps multiplicatively to move acceptance
00287              fraction towards target. This is a heuristic tuner used before a production run.
00288     """
00289     eps = initial_eps
00290     for attempt in range(tries):
00291         matrix_copy = matrix0.copy()
00292         # quick thermalize copy
00293         for i in range(50):
00294             metropolis_update(matrix_copy, eps_sub=eps)
00295         accepted = proposed = 0
00296         for i in range(test_sweeps):
00297             a, p = metropolis_update(matrix_copy, eps_sub=eps)
00298             accepted += a; proposed += p
00299         fraction = accepted / proposed if proposed > 0 else 0.0
00300         if abs(fraction - target) < 0.05:
00301             break
00302         eps *= 1.2 if fraction > target else 0.8
00303     return eps
00304
00305
00306 def run_su3_simulation(link_sites, eps_sub=0.06, burn_in_sweeps=500, MC_sweeps=2000, N_correlator=5):
00307     """
00308     @brief Run SU(3) Metropolis simulation collecting plaquette samples.
00309     @param link_sites ndarray: initial link configuration (modified in-place).
00310     @param eps_sub float: SU(2) subgroup proposal amplitude.
00311     @param burn_in_sweeps int: thermalization sweeps.
00312     @param MC_sweeps int: measurement sweeps.
00313     @param N_correlator int: interval between stored measurements.
00314     @return tuple: (plaquette_samples ndarray, mean_plaquette float, error_plaquette float)
00315     @details After burn-in we perform MC_sweeps sweeps and measure the average plaquette every
00316              N_correlator sweeps. Bootstrap error estimation is applied to the set of plaquette samples.
00317     """
00318     accepted = proposed = 0
00319     for i in range(burn_in_sweeps):
00320         a, p = metropolis_update(link_sites, eps_sub=eps_sub)
00321         accepted += a; proposed += p
00322     plaquette_samples = []
```

```
00323     accepted = proposed = 0
00324     for sweep in range(MC_sweeps):
00325         a, p = metropolis_update(link_sites, eps_sub=eps_sub)
00326         accepted += a; proposed += p
00327         if sweep % N_correlator == 0:
00328             plaquette = average_plaquette_su3(link_sites)
00329             plaquette_samples.append(plaquette)
00330     mean_plaquette, error_plaquette = bootstrap_mean_std(plaquette_samples, nboot=n_boot)
00331     return np.array(plaquette_samples), mean_plaquette, error_plaquette
00332
00333
00334 # ---------- Initialization helpers ----------
00335 def init_links_identity(link_sites):
00336     """
00337     @brief Initialize all links to the identity matrix.
00338     @param link_sites ndarray: link array to initialize (modified in-place).
00339     """
00340     for mu in range(D):
00341         for x in np.ndindex(*x_shape):
00342             link_sites[(mu,) + x] = np.eye(3, dtype=complex)
00343
00344
00345 def randomize_links_small(link_sites, amplitude=0.02):
00346     """
00347     @brief Apply small random SU(3) rotations (via embedded SU(2)) to each link for breaking symmetry.
00348     @param link_sites ndarray: link array (modified in-place).
00349     @param amplitude float: small rotation amplitude used for initial randomization.
00350     @details Useful to seed the Markov chain with a slightly randomized starting configuration.
00351     """
00352     for mu in range(D):
00353         for x in np.ndindex(*x_shape):
00354             for (i, j) in [(0, 1), (0, 2), (1, 2)]:
00355                 R2 = su2_random_unitary(amplitude)
00356                 R3 = embed_su2_into_su3(R2, i, j)
00357                 link_sites[(mu,) + x] = su3_matrices(R3 @ link_sites[(mu,) + x])
00358
00359
00360 # ---------- Wilson loop helper ----------
00361 def measure_wilson_loop_RT(link_sites, R, T, spatial_direction=0, time_direction=None):
00362     """
00363     @brief Measure the average Wilson loop W(R,T) for rectangular loops of spatial size R and temporal extent T.
00364     @param link_sites ndarray: link configuration.
00365     @param R int: spatial extent (number of spatial steps).
00366     @param T int: temporal extent (number of temporal steps).
00367     @param spatial_direction int: spatial direction index used for the R side.
00368     @param time_direction int or None: time direction index; defaults to D-1 (last axis).
00369     @return float: average Re Tr[W(R,T)] / 3 over all possible loop origins.
00370     @details
00371     The loop path starts at each lattice site x and multiplies the link matrices along the rectangular contour.
00372     Backward traversals multiply by Hermitian conjugate of the traversed link.
00373     """
00374     if time_direction is None:
00375         time_direction = D - 1
00376     total = 0.0
00377     count = 0
00378     for x in np.ndindex(*x_shape):
00379         current_x = x
00380         W = np.eye(3, dtype=complex)
00381         # R steps + spatial_direction
00382         for i in range(R):
00383             U = link_sites[(spatial_direction,) + current_x]
00384             W = W @ U
00385             current_x = x_neighbor(current_x, spatial_direction, 1)
00386         # T steps + time_direction
00387         for i in range(T):
00388             U = link_sites[(time_direction,) + current_x]
00389             W = W @ U
00390             current_x = x_neighbor(current_x, time_direction, 1)
00391         # R steps - spatial_direction (backwards)
00392         for i in range(R):
00393             current_x = x_neighbor(current_x, spatial_direction, -1)
00394             U = link_sites[(spatial_direction,) + current_x]
00395             W = W @ U.conj().T
00396         # T steps - time_direction (backwards)
00397         for i in range(T):
00398             current_x = x_neighbor(current_x, time_direction, -1)
00399             U = link_sites[(time_direction,) + current_x]
00400             W = W @ U.conj().T
00401         total += np.real(np.trace(W)) / 3.0
00402         count += 1
00403     return total / count
00404
00405
00406 def su3_simulation_with_wilson_loops(link_sites, eps_sub=0.06, burn_in_sweeps=500, MC_sweeps=2000,
      N_correlator=5, max_R=None, max_T=None):
00407     """
00408     @brief Run full SU(3) simulation storing Wilson loop matrices for each measurement.
```

```
00409     @param link_sites ndarray: initial link configuration (modified in-place).
00410     @param eps_sub float: SU(2) subgroup proposal amplitude.
00411     @param burn_in_sweeps int: thermalization sweeps.
00412     @param MC_sweeps int: measurement sweeps.
00413     @param N_correlator int: interval between stored measurements.
00414     @param max_R int or None: maximum spatial size to measure (defaults to L/2).
00415     @param max_T int or None: maximum temporal size to measure (defaults to L/2).
00416     @return tuple: (wilson_loops_samples ndarray [n_meas, n_R, n_T], R_values ndarray, T_values ndarray)
00417     @details
00418     Measures a grid of Wilson loops W(R,T) for R in [1..max_R], T in [1..max_T] at each stored configuration.
00419     """
00420     if max_R is None:
00421         max_R = lattice_size_L // 2
00422     if max_T is None:
00423         max_T = lattice_size_L // 2
00424     R_values = np.arange(1, max_R + 1)
00425     T_values = np.arange(1, max_T + 1)
00426     n_R = len(R_values)
00427     n_T = len(T_values)
00428     # Thermalize
00429     for i in range(burn_in_sweeps):
00430         metropolis_update(link_sites, eps_sub=eps_sub)
00431     wilson_loops_samples = []
00432     for sweep in range(MC_sweeps):
00433         metropolis_update(link_sites, eps_sub=eps_sub)
00434         if sweep % N_correlator == 0:
00435             W_sample = np.zeros((n_R, n_T))
00436             for i, R in enumerate(R_values):
00437                 for j, T in enumerate(T_values):
00438                     W_sample[i, j] = measure_wilson_loop_RT(link_sites, R, T)
00439             wilson_loops_samples.append(W_sample)
00440     wilson_loops_samples = np.array(wilson_loops_samples)
00441     return wilson_loops_samples, R_values, T_values
00442
00443
00444 # ============================================================
00445 # ================ PLAQUETTE CALCULATION SECTION ==============
00446 # ============================================================
00447 init_links_identity(link_matrix)
00448 randomize_links_small(link_matrix, amplitude=0.02)
00449
00450 eps_tuned = tune_eps_su3(link_matrix, initial_eps=eps_initial)
00451 samples, plaq_mean, plaq_err = run_su3_simulation(
00452     link_matrix, eps_sub=eps_tuned, burn_in_sweeps=burn_in_sweeps,
00453     MC_sweeps=MC_sweeps, N_correlator=MC_measure_interval)
00454
00455 print(f"Average plaquette = {plaq_mean:.6f} ± {plaq_err:.6f}")
00456
00457 # Plot plaquette history
00458 plt.figure(figsize=(8, 5))
00459 plt.plot(np.arange(len(samples)), samples, marker='o', linestyle='-')
00460 plt.xlabel('Measurement Index')
00461 plt.ylabel('Plaquette Value (Re Tr P / 3)')
00462 plt.title('Plaquette History')
00463 plt.tight_layout()
00464 plt.show()
00465
00466 # Plot histogram of plaquette samples
00467 plt.figure(figsize=(8, 5))
00468 plt.hist(samples, bins=30, alpha=0.75)
00469 plt.xlabel('Plaquette Value')
00470 plt.ylabel('Frequency')
00471 plt.title('Histogram of Plaquette Samples')
00472 plt.tight_layout()
00473 plt.show()
00474
00475
00476 # ============================================================
00477 # ================ WILSON LOOP CALCULATION SECTION ============
00478 # ============================================================
00479 wilson_loops_samples, R_values, T_values = su3_simulation_with_wilson_loops(
00480     link_matrix, eps_sub=eps_tuned, burn_in_sweeps=burn_in_sweeps,
00481     MC_sweeps=MC_sweeps, N_correlator=MC_measure_interval,
00482     max_R=lattice_size_L // 2, max_T=lattice_size_L // 2)
00483
00484 avg_wilson_loops = np.mean(wilson_loops_samples, axis=0)  # shape (n_R, n_T)
00485
00486 # Extract static potential V(R) via effective mass from Wilson loops
00487 V_R = np.zeros(len(R_values))
00488 for i in range(len(R_values)):
00489     potentials = []
00490     for j in range(len(T_values) - 1):
00491         W_T = avg_wilson_loops[i, j]
00492         W_Tp1 = avg_wilson_loops[i, j + 1]
00493         if W_T > 0 and W_Tp1 > 0:
00494             potentials.append(-np.log(W_Tp1 / W_T))
00495     V_R[i] = np.mean(potentials) if potentials else np.nan
```

```
00496
00497 print("Wilson loop and static potential calculation complete.")
00498 print("R values:", R_values)
00499 print("V(R):", V_R)
00500
00501 plt.figure(figsize=(8, 5))
00502 plt.plot(R_values, V_R, marker='o', linestyle='-')
00503 plt.xlabel('Spatial Separation R')
00504 plt.ylabel('Static Quark-Antiquark Potential V(R)')
00505 plt.title('Static Potential from Averaged Wilson Loops')
00506 plt.grid(True)
00507 plt.tight_layout()
00508 plt.show()
00509
00510
00511 # ---------- Classical gluon-field diagnostics ----------
00512 def gell_mann_matrices():
00513     """
00514     @brief Return the eight Gell-Mann matrices ^a (3x3).
00515     @return list: eight 3x3 numpy arrays forming a basis for su(3).
00516     @details These are used to project Lie-algebra components from SU(3) link matrices.
00517     """
00518     lambda_ = []
00519     lambda_.append(np.array([[0, 1, 0], [1, 0, 0], [0, 0, 0]], dtype=complex))
00520     lambda_.append(np.array([[0, -1j, 0], [1j, 0, 0], [0, 0, 0]], dtype=complex))
00521     lambda_.append(np.array([[1, 0, 0], [0, -1, 0], [0, 0, 0]], dtype=complex))
00522     lambda_.append(np.array([[0, 0, 1], [0, 0, 0], [1, 0, 0]], dtype=complex))
00523     lambda_.append(np.array([[0, 0, -1j], [0, 0, 0], [1j, 0, 0]], dtype=complex))
00524     lambda_.append(np.array([[0, 0, 0], [0, 0, 1], [0, 1, 0]], dtype=complex))
00525     lambda_.append(np.array([[0, 0, 0], [0, 0, -1j], [0, 1j, 0]], dtype=complex))
00526     lambda_.append((1 / np.sqrt(3)) * np.array([[1, 0, 0], [0, 1, 0], [0, 0, -2]], dtype=complex))
00527     return lambda_
00528
00529
00530 def extract_gluon_field(U, g=1.0, a=1.0):
00531     """
00532     @brief Extract approximate local gauge field components A^a from a single SU(3) link.
00533     @param U ndarray: SU(3) link matrix.
00534     @param g float: gauge coupling (default 1.0).
00535     @param a float: lattice spacing (default 1.0).
00536     @return ndarray: array shape (8,) containing A^a components (real).
00537     @details For small lattice spacing we approximate U   exp(i g a A) => A   (U - U†)/(2 i g a).
00538              We then project the traceless anti-Hermitian part onto the Gell-Mann basis.
00539     """
00540     difference = (U - U.conj().T) / (2j * g * a)
00541     difference -= np.trace(difference).real / 3.0 * np.eye(3)
00542     lambda_ = gell_mann_matrices()
00543     A_components = np.array([np.real(np.trace(difference @ lambda_a)) / 2.0 for lambda_a in lambda_])
00544     return A_components
00545
00546
00547 def field_strength_tensor(link_sites, x, mu, nu, g=1.0, a=1.0):
00548     """
00549     @brief Compute the lattice field-strength components F_{mu,nu}^a at site x from the plaquette.
00550     @param link_sites ndarray: link variables.
00551     @param x tuple: lattice coordinate.
00552     @param mu int: direction mu.
00553     @param nu int: direction nu.
00554     @param g float: gauge coupling.
00555     @param a float: lattice spacing.
00556     @return ndarray: shape (8,) F^a components (real).
00557     @details Uses the anti-Hermitian traceless projection of the plaquette:
00558              F ~ (P - P†)/(2 i g a^2) projected on Gell-Mann matrices.
00559     """
00560     P = plaquette_matrix(x, mu, nu, link_sites)
00561     difference = (P - P.conj().T) / (2j * g * a ** 2)
00562     difference -= np.trace(difference).real / 3.0 * np.eye(3)
00563     lambda_ = gell_mann_matrices()
00564     F_components = np.array([np.real(np.trace(difference @ lambda_a)) / 2.0 for lambda_a in lambda_])
00565     return F_components
00566
00567
00568 def measure_avg_A2_and_F2(link_sites, g=1.0, a=1.0):
00569     """
00570     @brief Compute averages  A^2  and  F^2  over the entire lattice as diagnostics.
00571     @param link_sites ndarray: link configuration.
00572     @param g float: gauge coupling.
00573     @param a float: lattice spacing.
00574     @return tuple: (A2_avg float, F2_avg float)
00575     @details A^2 and F^2 are computed by summing squares of components and normalizing by counts.
00576     """
00577     A2_sum = 0.0
00578     F2_sum = 0.0
00579     nA = 0
00580     nF = 0
00581     for x in np.ndindex(*x_shape):
00582         for mu in range(D):
```

```
00583             U = link_sites[(mu,) + x]
00584             A = extract_gluon_field(U, g=g, a=a)
00585             A2_sum += np.dot(A, A)
00586             nA += 1
00587         for mu in range(D):
00588             for nu in range(mu + 1, D):
00589                 F = field_strength_tensor(link_sites, x, mu, nu, g=g, a=a)
00590                 F2_sum += np.dot(F, F)
00591                 nF += 1
00592     return A2_sum / nA, F2_sum / nF
00593
00594
00595 A2_avg, F2_avg = measure_avg_A2_and_F2(link_matrix)
00596 print(f" A² = {A2_avg:.6e},  F² = {F2_avg:.6e}")
```