CMI-based recoverability versus Wilson-loop diagnostics in $\mathbb{Z}_2$ lattice gauge theory (2+1D)

Exact diagonalization benchmark on small open lattices

Lluis Eriksson
Independent Researcher
lluiseriksson@gmail.com

January 2026

Abstract

We provide a finite-size benchmark testing whether a CMI-based recoverability proxy correlates with Wilson-loop confinement diagnostics in $\mathbb{Z}_2$ lattice gauge theory in 2+1 dimensions. We compute ground states by sparse exact diagonalization on 2×2 and 2×3 open lattices for a standard link-qubit Hamiltonian with a Gauss-law penalty term (biasing the gauge-invariant sector, verified numerically by $\langle G_v \rangle \simeq 1$). We evaluate entropic quantities using pure-state Schmidt decompositions (SVD) without constructing reduced density matrices. For the benchmark we take $d_{\text{edge}} = 2$, hence $\Sigma_B(w) = |\partial B(w)| \log 2$. We also provide fully reproducible code (Appendix A) to rerun the experiment in Colab.

1 Confinement as an Information Horizon (Phase 4)

1.1 Confinement as an Information Horizon (gauge-first, conditional)

Scope and status. We state a *conditional* bridge from a spatial area law (with positive spatial string tension) to an operational "information horizon" quantified by conditional mutual information (CMI) and recoverability. All entropic quantities are defined for a *quantum state* on a spatial Hilbert space. Euclidean expectation values enter only as physical input hypotheses and are related to the quantum state via a transfer-matrix (OS) construction.

Regulator (finite local dimension). We assume a regulator yielding finite-dimensional local Hilbert spaces (e.g. a finite gauge group such as $\mathbb{Z}_2$, or an $SU(N)$ representation cutoff). Constants below may depend on this regulator. In the benchmark section we use link qubits and therefore take $d_{\text{edge}} = 2$ (so boundary prefactors are proportional to $\log 2$).

Set-up: state, loops, collar geometry, and recovery length

Quantum state from Euclidean data. Fix a reflection-positive transfer-matrix construction producing a Hamiltonian H on a spatial Hilbert space $\mathcal{H}_{\text{slice}}$. For $\beta_T > 0$ define the thermal state

$$\rho_{\beta_T} := \frac{e^{-\beta_T H}}{\operatorname{Tr}(e^{-\beta_T H})},$$

and let ρ_0 denote the ground-state limit (when it exists) $\rho_0 = \lim_{\beta_T \to \infty} \rho_{\beta_T}$. All hypotheses below are assumed to hold for the chosen state ρ (either ρ_{β_T} at fixed β_T, or ρ_0 if the ground-state limit exists); we do not address uniformity in β_T here.

Spatial Wilson loops and spatial string tension. Let $\mathcal{C}$ be a closed loop in a fixed spatial time-slice, and $W(\mathcal{C})$ the corresponding spatial Wilson loop observable. We define the *spatial string tension* $\sigma_{\rm str}$ by the large-loop area law

$$\sigma_{\rm str} := -\lim_{\text{Area}\to\infty} \frac{1}{\text{Area}} \log\langle W(\mathcal{C})\rangle,$$

where Area is the minimal spanning area (in plaquette units) in the spatial slice. On the very small lattices used for exact diagonalization, we use finite-size proxies (e.g. Creutz ratios) rather than the asymptotic definition above; we do not equate these proxies with $\sigma_{\rm str}$. We use the spatial area law as the flux-suppression input relevant for spatial walls; we do not equate $\sigma_{\rm str}$ with the strict finite-temperature confinement order parameter.

Tripartition and boundary scale. Let A and C be disjoint spatial regions separated by a collar $B(w)$ of thickness $w \in \mathbb{Z}_{\geq 0}$ (measured in lattice/graph-distance units). Let $\partial B(w)$ denote the boundary of $B(w)$ within the spatial slice (degrees of freedom in $B(w)$ adjacent to $B(w)^c$). We introduce the boundary prefactor

$$\Sigma_B(w) := |\partial B(w)| \log 2.$$

CMI and two operational recovery-length conventions. For the reduced state ρ_{ABC} induced by ρ on $A \cup B(w) \cup C$, define (in nats)

$$I(A:C \mid B(w)) := S(AB(w)) + S(B(w)C) - S(B(w)) - S(AB(w)C).$$

We use two operational conventions at tolerance $\varepsilon > 0$:

$$\xi_{\rm rec}^{\rm abs}(\varepsilon) := \min\{w :\ I(A:C \mid B(w)) \leq \varepsilon\}, \qquad \xi_{\rm rec}^{\rm norm}(\varepsilon) := \min\left\{w :\ \frac{I(A:C \mid B(w))}{\Sigma_B(w)} \leq \varepsilon\right\}.$$

Pre-saturation regime. We say that w is *pre-saturation* if $(A \cup B(w) \cup C) \neq \Lambda_{\rm slice}$. (For mixed thermal states, pure-state identities need not hold; saturation still limits the range of informative w on a finite slice.)

Hypotheses (explicit bridge inputs)

(H1) Spatial area law input. Assume there exist constants $K_W < \infty$ and $\sigma_{\rm str} > 0$ such that for the class of spatial loops considered,

$$\langle W(\mathcal{C})\rangle \ \leq\ K_W \exp\big(-\sigma_{\rm str}\,\text{Area}_{\min}(\mathcal{C})\big).$$

(H2) Wall geometry forces a minimal crossing area. Assume there exists a geometry constant $c_0 > 0$ such that for all pre-saturation w,

$$\text{Area}_{\min}(w) \ \geq\ c_0\, w.$$

In wall geometries, $\text{Area}_{\min}(w)$ typically grows linearly in w with a coefficient proportional to the wall cross-sectional area; we absorb this dependence into c_0.

Gauge-first bridge quantity: loop-restricted connected leakage. Let $\mathcal{W}_A(L_0)$ and $\mathcal{W}_C(L_0)$ denote fixed families of spatial Wilson-loop observables supported in A and C, respectively, with perimeter bounded by a regulator-dependent cutoff L_0. Define the loop-restricted connected leakage

$$\Delta_{\mathrm{loop}}(w) \;:=\; \sup_{W_A\in\mathcal{W}_A(L_0),\; W_C\in\mathcal{W}_C(L_0)} \left|\langle W_A W_C\rangle_\rho - \langle W_A\rangle_\rho \, \langle W_C\rangle_\rho\right|.$$

We emphasize that $\Delta_{\mathrm{loop}}(w)$ depends on w through the geometry (choice of regions separated by the collar), not through a literal conditioning operation on $B(w)$.

(H3$^{\mathrm{g}}$) Flux-tube dominance and exponential suppression of loop leakage. Assume there exist constants $K_\Delta < \infty$ and $\kappa > 0$ such that for all pre-saturation w,

$$\Delta_{\mathrm{loop}}(w) \;\le\; K_\Delta \, \exp\big(-\kappa\,\sigma_{\mathrm{str}}\,\mathrm{Area}_{\min}(w)\big) \;\le\; K_\Delta \, \exp\big(-\kappa\,\sigma_{\mathrm{str}}\,c_0\,w\big).$$

This assumption is motivated by the heuristic that dominant cross-wall communication processes in a confining regime are flux-tube mediated and inherit an area-law suppression controlled by σ_{str}.

(H4$^{\mathrm{QI}}$) Structural input: from loop leakage to CMI. Assume there exist constants $K_I < \infty$, $C_f < \infty$, an exponent $p \ge 1$, and a threshold $x_0 > 0$, and a nondecreasing function $f : [0,1] \to \mathbb{R}_+$ with $f(0) = 0$, such that for all pre-saturation w with $\Delta_{\mathrm{loop}}(w) \le x_0$,

$$I(A:C \mid B(w)) \;\le\; K_I\,\Sigma_B(w)\,f(\Delta_{\mathrm{loop}}(w)), \qquad f(x) \;\le\; C_f\,x^p \quad (0 \le x \le x_0).$$

The constants may depend on the regulator and on the considered state family (e.g. temperature and couplings), but not on w within the pre-saturation regime.

Main conditional proposition

Proposition 1.1 (Loop leakage suppression $\Rightarrow$ information horizon (conditional))**.** *Assume (H1)–(H4$^{\mathrm{QI}}$). Then there exist constants $K < \infty$ and $\alpha > 0$ such that for all pre-saturation w with $\Delta_{\mathrm{loop}}(w) \le x_0$,*

$$I(A:C \mid B(w)) \;\le\; K\,\Sigma_B(w)\,e^{-\alpha w}, \qquad \alpha := p\,\kappa\,\sigma_{\mathrm{str}}\,c_0,$$

where one may take $K := K_I\,C_f\,(K_\Delta)^p$. Consequently, for any $\varepsilon > 0$,

$$\xi_{\mathrm{rec}}^{\mathrm{norm}}(\varepsilon) \;\le\; \frac{1}{\alpha}\log\frac{K}{\varepsilon}, \qquad \xi_{\mathrm{rec}}^{\mathrm{abs}}(\varepsilon) \;\le\; \frac{1}{\alpha}\log\frac{K\,\Sigma_B^{\max}}{\varepsilon},$$

where $\Sigma_B^{\max} := \sup\{\Sigma_B(w) : w \in \mathbb{Z}_{\ge 0}$ and $(A \cup B(w) \cup C) \ne \Lambda_{\mathrm{slice}}\}$.

Proof sketch. By (H3$^{\mathrm{g}}$) and (H2), $\Delta_{\mathrm{loop}}(w) \le K_\Delta e^{-\kappa\sigma_{\mathrm{str}}c_0 w}$. By (H4$^{\mathrm{QI}}$) and $f(x) \le C_f x^p$ (valid when $\Delta_{\mathrm{loop}}(w) \le x_0$),

$$I(A:C \mid B(w)) \;\le\; K_I\,\Sigma_B(w)\,C_f\,(K_\Delta)^p\,e^{-p\kappa\sigma_{\mathrm{str}}c_0 w},$$

which is the claimed bound with $\alpha = p\kappa\sigma_{\mathrm{str}}c_0$. □

Operational corollary: recoverability from small CMI

On a finite lattice, small CMI implies approximate recoverability. Let F denote the Uhlmann fidelity $F(\rho,\sigma) := \|\sqrt{\rho}\sqrt{\sigma}\|_1$. There exists a channel $\mathcal{R}_{B\to BC}$ such that

$$I(A:C \mid B(w)) \;\geq\; -2\log F\big(\rho_{AB(w)C}, (\mathrm{id}\otimes\mathcal{R}_{B\to BC})(\rho_{AB(w)})\big)\,,$$

hence

$$F\big(\rho_{AB(w)C}, (\mathrm{id}\otimes\mathcal{R})(\rho_{AB(w)})\big) \;\geq\; \exp\!\left(-\frac{1}{2}\,I(A:C \mid B(w))\right).$$

In companion numerical benchmarks one may instantiate a concrete recovery map (e.g. a regularized Petz-type map) and report $-\log F$ as an operational recoverability error.

Remark: the roadmap "missing link" (uniformity under refinement)

Remark 1.1 (Uniform area-law input $\Rightarrow$ uniform information clustering (conditional)). If one could establish a regulator-uniform lower bound $\sigma_{\mathrm{str}}(a) \geq \sigma_\star > 0$ as the lattice spacing $a \downarrow 0$, and justify (H3$^{\mathrm{g}}$) and (H4$^{\mathrm{QI}}$) with constants not deteriorating uncontrollably as $a \downarrow 0$, then Proposition 1.1 would yield a uniform exponential information horizon:

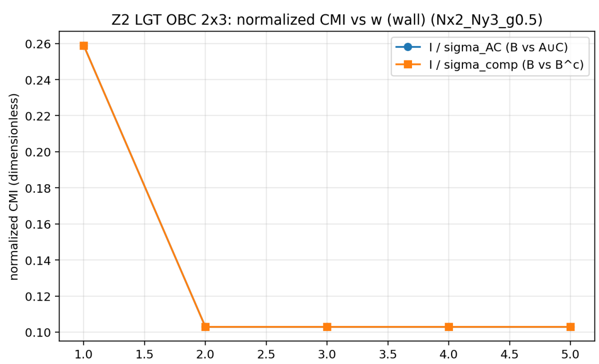
$$I(A:C \mid B(w)) \;\leq\; K_\star\,\Sigma_B(w)\,e^{-\alpha_\star w}, \qquad \alpha_\star = p_\star\kappa_\star c_{0,\star}\sigma_\star.$$

Such a bound is the quantitative "information clustering" input needed for iterative coarse-graining/recovery steps in a recovery-based renormalization strategy.

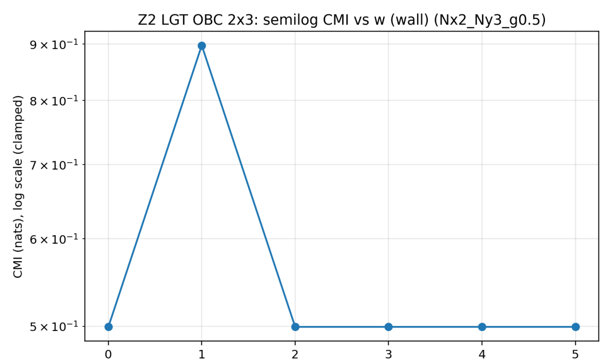
2 Exact diagonalization benchmark (wall geometry)

This section documents the specific ED benchmark instance whose plots are included below, using the wall collar construction on the 2×3 open lattice at $g = 0.5$.

2.1 Figures



(a) $I(A:C \mid B(w))/\Sigma_B(w)$ vs. w (wall).



(b) $I(A:C \mid B(w))$ vs. w (semilog).

Figure 1: Wall-collar ED benchmark on the 2×3 open lattice at $g = 0.5$.

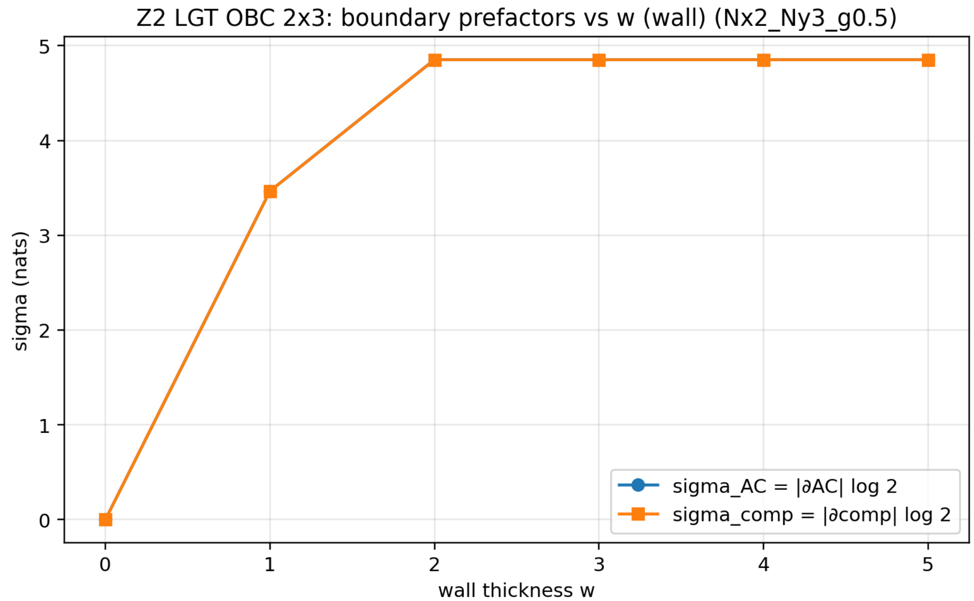


Figure 2: Boundary prefactors vs. w for the wall geometry on the 2×3 open lattice at $g = 0.5$.

2.2 Table (minimal summary of the same run)

In our implementation, $w = 0$ corresponds to $B(w) = \varnothing$, hence $\Sigma_B(0) = 0$ and the ratio I/Σ_B is undefined at $w = 0$.

Table 1: Wall geometry ED summary for 2×3 open lattice at $g = 0.5$ (from console output).

w	$\lvert\text{path}\rvert$	$\lvert B(w)\rvert$	$\lvert\partial_{\text{comp}} B(w)\rvert$	$I(A : C \mid B(w))$	$I/\Sigma_B(w)$
0	3	0	0	0.4992999	—
1	3	5	5	0.8974278	0.2589429
2	3	9	7	0.4992999	0.1029054

A Reproducibility code (Colab/Jupyter)

A.1 Script 1: ED + wall collar geometry (single-cell runnable)

To avoid transcription errors in the paper PDF, we provide the experiment as a single Python file. In Colab/Jupyter, create a file `z2_cmi_wall_geometry.py` with the contents below (or download it from the project repository), then run (the leading `!` is Jupyter/Colab shell syntax; in a terminal run `python z2_cmi_wall_geometry.py`):

```
!python z2_cmi_wall_geometry.py
```

For completeness, we include the full contents of `z2_cmi_wall_geometry.py` below.

```
# z2_cmi_wall_geometry.py
# Z2 gauge theory: CMI/entropy validator with WALL collar geometry + two boundary definitions.
#
# Key features:
# - LinearOperator ED (no sparse matrix build)
# - Pure-state entropies via Schmidt/SVD (no density matrices built)
# - Collar B(w) defined as thickened shortest-path "wall" between A and C
# - Two boundary sizes: B vs (AUC) and B vs B^c, both with sigma=|boundary| log 2
# - Writes CSV + PNGs
#
# Requires: numpy, scipy, matplotlib (standard in Colab)

import math, csv, time
from collections import deque
import numpy as np
import scipy.sparse.linalg as spla
import matplotlib.pyplot as plt

# -------------------------
# User settings (safe to edit only here)
# -------------------------
OUT_PREFIX = "z2_cmi_wall"
LAMBDA_GAUSS = 50.0
ENT_EPS = 1e-15

G_LIST = [0.5, 1.0, 2.0]
W_LIST = [0, 1, 2, 3, 4, 5] # wall thickness (graph steps around the wall path)
RUN_2x2 = True
RUN_2x3 = True

EIG_TOL = 1e-10
EIG_MAXITER = 30000

# ---------------------------
# Lattice indexing utilities
# ---------------------------

def idx_h(x, y, Nx, Ny):
    return y * Nx + x

def idx_v(x, y, Nx, Ny):
    Nh = Nx * (Ny + 1)
    return Nh + y * (Nx + 1) + x

def build_lattice_obc(Nx, Ny):
```

```python
    """
    Open boundary conditions on an Nx x Ny plaquette lattice (spatial).
    Qubits live on links.
    Returns: Nlinks, plaquettes, stars, adjacency
    """
    def vid(x, y):
        return y * (Nx + 1) + x

    link_endpoints = []

    # horizontals
    for y in range(Ny + 1):
        for x in range(Nx):
            link_endpoints.append((vid(x, y), vid(x + 1, y)))

    # verticals
    for y in range(Ny):
        for x in range(Nx + 1):
            link_endpoints.append((vid(x, y), vid(x, y + 1)))

    Nlinks = len(link_endpoints)

    # plaquettes
    plaquettes = []
    for y in range(Ny):
        for x in range(Nx):
            pl = [
                idx_h(x, y, Nx, Ny),
                idx_v(x + 1, y, Nx, Ny),
                idx_h(x, y + 1, Nx, Ny),
                idx_v(x, y, Nx, Ny),
            ]
            plaquettes.append(pl)

    Nv = (Nx + 1) * (Ny + 1)
    stars = [[] for _ in range(Nv)]
    for ell, (a, b) in enumerate(link_endpoints):
        stars[a].append(ell)
        stars[b].append(ell)

    # adjacency: links adjacent if share a vertex
    v2links = [[] for _ in range(Nv)]
    for ell, (a, b) in enumerate(link_endpoints):
        v2links[a].append(ell)
        v2links[b].append(ell)

    adjacency = [set() for _ in range(Nlinks)]
    for v in range(Nv):
        Ls = v2links[v]
        for i in Ls:
            for j in Ls:
                if i != j:
                    adjacency[i].add(j)

    return Nlinks, plaquettes, stars, adjacency

def mask_from_links(link_list):
    m = 0
    for ell in link_list:
```

```python
        m |= (1 << int(ell))
    return m

def parity_u64(x: int) -> int:
    return x.bit_count() & 1

# ----------------------------
# Regions: opposite-corner plaquette patches A,C (links)
# ----------------------------

def plaquette_patch_A_C(Nx, Ny):
    # SW plaquette boundary
    A = [idx_h(0,0,Nx,Ny), idx_v(1,0,Nx,Ny), idx_h(0,1,Nx,Ny), idx_v(0,0,Nx,Ny)]
    # NE plaquette boundary
    C = [idx_h(Nx-1,Ny-1,Nx,Ny), idx_v(Nx,Ny-1,Nx,Ny), idx_h(Nx-1,Ny,Nx,Ny), idx_v(Nx-1,Ny-1,Nx,Ny)]
    return sorted(set(A)), sorted(set(C))

# ----------------------------
# Wall collar geometry: shortest path + thickening
# ----------------------------

def shortest_path_links(adjacency, sources, targets):
    """Shortest path in the link graph from any source in sources to any target in targets."""
    n = len(adjacency)
    srcset = set(sources)
    tgtset = set(targets)

    prev = np.full(n, -1, dtype=np.int32)
    dist = np.full(n, -1, dtype=np.int32)
    q = deque()

    for s in srcset:
        dist[s] = 0
        q.append(s)

    meet = -1
    while q:
        u = q.popleft()
        if u in tgtset:
            meet = u
            break
        for v in adjacency[u]:
            if dist[v] < 0:
                dist[v] = dist[u] + 1
                prev[v] = u
                q.append(v)

    if meet < 0:
        return []

    # reconstruct
    path = []
    u = meet
    while u >= 0:
        path.append(int(u))
        u = int(prev[u])
    path.reverse()
```

```python
    return path

def buffer_around_set(adjacency, core, w, forbid=set()):
    """
␣␣␣␣All␣links␣within␣graph␣distance␣<=␣w␣from␣any␣link␣in␣core,␣excluding␣forbid.
␣␣␣␣NOTE:␣in␣this␣convention,␣w<=0␣returns␣empty,␣so␣B(0)=∅.
␣␣␣␣"""
    if w <= 0:
        return []
    n = len(adjacency)
    dist = np.full(n, -1, dtype=np.int32)
    q = deque()

    core = list(dict.fromkeys(core))
    for s in core:
        dist[s] = 0
        q.append(s)

    out = set()
    while q:
        u = q.popleft()
        if dist[u] > w:
            continue
        if u not in forbid:
            out.add(int(u))
        if dist[u] == w:
            continue
        for v in adjacency[u]:
            if dist[v] < 0:
                dist[v] = dist[u] + 1
                q.append(v)
    return sorted(out)

def B_wall_between_A_C(adjacency, A, C, w):
    """
␣␣␣␣Define␣B(w)␣as␣thickening␣of␣a␣shortest␣path␣between␣A␣and␣C.
␣␣␣␣Returns␣(B,␣path).
␣␣␣␣"""
    path = shortest_path_links(adjacency, A, C)
    forbid = set(A) | set(C)
    B = buffer_around_set(adjacency, path, w, forbid=forbid)
    return B, path

# Two boundary definitions to compare:
def boundary_links_vs_AC(adjacency, B, A, C):
    """∂_AC␣B␣:=␣{l␣in␣B␣:␣exists␣neighbor␣in␣AUC}."""
    ACset = set(A) | set(C)
    bd = set()
    for ell in B:
        for nbr in adjacency[ell]:
            if nbr in ACset:
                bd.add(ell)
                break
    return sorted(bd)

def boundary_links_vs_complement(adjacency, B):
    """∂_comp␣B␣:=␣{l␣in␣B␣:␣exists␣neighbor␣not␣in␣B}."""
    Bset = set(B)
    bd = set()
```

```python
    for ell in B:
        for nbr in adjacency[ell]:
            if nbr not in Bset:
                bd.add(ell)
                break
    return sorted(bd)

# ----------------------------
# LinearOperator Hamiltonian (no sparse matrix build)
# H(g) = -(1/g) sum_p prod_{l in ∂p} Z_l - g sum_l X_l + Lambda sum_v (I - G_v)
# G_v = prod_{l incident to v} X_l
# ----------------------------

def build_diag_z2(Nlinks, plaquettes, stars, g, Lambda):
    dim = 1 << Nlinks
    states = np.arange(dim, dtype=np.uint32 if Nlinks <= 32 else np.uint64)

    invg = 1.0 / float(g)
    diag = np.zeros(dim, dtype=np.float64)

    plaq_masks = [mask_from_links(pl) for pl in plaquettes]
    for mp in plaq_masks:
        par = np.fromiter((parity_u64(int(s) & mp) for s in states), count=dim, dtype=np.int8)
        eig = 1.0 - 2.0 * par # +1 even, -1 odd
        diag += -(invg) * eig

    if Lambda != 0.0:
        diag += float(Lambda) * len([st for st in stars if len(st) > 0])

    return diag

def make_H_linear_operator(Nlinks, plaquettes, stars, g, Lambda):
    dim = 1 << Nlinks
    diag = build_diag_z2(Nlinks, plaquettes, stars, g, Lambda)

    states = np.arange(dim, dtype=np.uint32 if Nlinks <= 32 else np.uint64)
    bit_masks = np.array([1 << ell for ell in range(Nlinks)], dtype=states.dtype)
    star_masks = np.array([mask_from_links(st) for st in stars if len(st) > 0], dtype=states.dtype)

    gf = float(g)
    Lf = float(Lambda)

    def matvec(x):
        y = diag * x
        for m in bit_masks:
            y += (-gf) * x[states ^ m]
        if Lf != 0.0:
            for mv in star_masks:
                y += (-Lf) * x[states ^ mv]
        return y

    return spla.LinearOperator((dim, dim), matvec=matvec, dtype=np.complex128)

def ground_state_linearop(Hop, tol=1e-10, maxiter=30000):
    vals, vecs = spla.eigsh(Hop, k=1, which="SA", tol=tol, maxiter=maxiter)
    psi = vecs[:, 0].astype(np.complex128)
    psi /= np.linalg.norm(psi)
```

```
    return float(vals[0]), psi

def expval_X_string(psi, mask, Nlinks):
    dim = 1 << Nlinks
    s = np.arange(dim, dtype=np.uint32 if Nlinks <= 32 else np.uint64)
    t = s ^ mask
    return float(np.vdot(psi, psi[t]).real)

def gauge_diagnostics(psi, stars, Nlinks):
    vals = []
    for st in stars:
        if len(st) == 0:
            continue
        mv = mask_from_links(st)
        vals.append(expval_X_string(psi, mv, Nlinks))
    return float(np.min(vals)), float(np.mean(vals))

# ----------------------------
# Pure-state entropy via Schmidt/SVD (NO density matrices)
# ----------------------------

def entropy_subset_pure(psi_t, subset, ent_eps=1e-15):
    """
␣␣␣␣psi_t:␣tensor␣view␣of␣psi␣with␣shape␣[2]*N.
␣␣␣␣subset:␣list␣of␣indices.
␣␣␣␣Computes␣S(subset)=S(complement)␣using␣the␣smaller␣side.
␣␣␣␣"""
    N = psi_t.ndim
    subset = sorted(set(subset))
    if len(subset) == 0 or len(subset) == N:
        return 0.0
    comp = [i for i in range(N) if i not in subset]

    # Use smaller side for SVD
    if len(subset) > len(comp):
        subset, comp = comp, subset

    perm = subset + comp
    psi_perm = np.transpose(psi_t, axes=perm)

    d_keep = 1 << len(subset)
    d_tr = 1 << len(comp)
    M = psi_perm.reshape(d_keep, d_tr)

    s = np.linalg.svd(M, compute_uv=False)
    p = (s.real**2) # singular values are real nonnegative
    p = p / np.sum(p)
    p = p[p > ent_eps]
    return float(-np.sum(p * np.log(p)))

# ----------------------------
# One lattice experiment: entropies + CMI vs wall thickness w
# ----------------------------

def compute_wall_series_pure(Nx, Ny, g, Lambda, w_list, ent_eps, out_prefix):
    Nlinks, plaquettes, stars, adjacency = build_lattice_obc(Nx, Ny)
    A, C = plaquette_patch_A_C(Nx, Ny)

    print(f"\n===␣Lattice␣{Nx}x{Ny}␣plaquettes:␣Nlinks={Nlinks},␣dim=2^{Nlinks}={1<<
```

```
        Nlinks} ===")
    print(f"g={g}, Lambda={Lambda}, w_list={w_list}")

    t0 = time.time()
    Hop = make_H_linear_operator(Nlinks, plaquettes, stars, g=g, Lambda=Lambda)
    E0, psi = ground_state_linearop(Hop, tol=EIG_TOL, maxiter=EIG_MAXITER)
    print(f"ED done in {time.time()-t0:.2f}s, E0={E0:.6f}")

    minG, meanG = gauge_diagnostics(psi, stars, Nlinks)
    print(f"Gauge check: min <Gv>={minG:.6f}, mean <Gv>={meanG:.6f} (should be ~1)")

    psi_t = psi.reshape([2]*Nlinks)

    series = []
    rows = []

    for w in w_list:
        B, path = B_wall_between_A_C(adjacency, A, C, w)

        AB = A + B
        BC = B + C
        ABC = A + B + C

        S_AB = entropy_subset_pure(psi_t, AB, ent_eps=ent_eps)
        S_BC = entropy_subset_pure(psi_t, BC, ent_eps=ent_eps)
        S_B = entropy_subset_pure(psi_t, B, ent_eps=ent_eps)
        S_ABC = entropy_subset_pure(psi_t, ABC, ent_eps=ent_eps)

        I = S_AB + S_BC - S_B - S_ABC

        bd_AC = boundary_links_vs_AC(adjacency, B, A, C)
        bd_comp = boundary_links_vs_complement(adjacency, B)

        sigma_AC = len(bd_AC) * math.log(2.0)
        sigma_comp = len(bd_comp) * math.log(2.0)

        Inorm_AC = (I / sigma_AC) if sigma_AC > 0 else float("nan")
        Inorm_comp = (I / sigma_comp) if sigma_comp > 0 else float("nan")

        series.append((w, len(path), len(B),
                       len(bd_AC), sigma_AC, Inorm_AC,
                       len(bd_comp), sigma_comp, Inorm_comp,
                       S_AB, S_BC, S_B, S_ABC, I))

        rows.append([
            Nx, Ny, Nlinks, g, Lambda, E0, minG, meanG,
            w, len(A), len(B), len(C),
            len(path), # wall core size
            len(bd_AC), sigma_AC, Inorm_AC,
            len(bd_comp), sigma_comp, Inorm_comp,
            S_AB, S_BC, S_B, S_ABC, I
        ])

        print(
            f" w={w:2d} |path|={len(path):2d} |B|={len(B):2d} "
            f"|∂AC|={len(bd_AC):2d} σAC={sigma_AC:.4f} I/σAC={Inorm_AC} "
            f"|∂comp|={len(bd_comp):2d} σcomp={sigma_comp:.4f} I/σcomp={Inorm_comp} "
            f"I={I:.6e}"
        )
```

```

    # plots
    ws = np.array([t[0] for t in series], dtype=float)
    S_AB = np.array([t[9] for t in series], dtype=float)
    S_BC = np.array([t[10] for t in series], dtype=float)
    S_B = np.array([t[11] for t in series], dtype=float)
    S_ABC = np.array([t[12] for t in series], dtype=float)
    Ivals = np.array([t[13] for t in series], dtype=float)

    sigAC = np.array([t[4] for t in series], dtype=float)
    sigComp = np.array([t[7] for t in series], dtype=float)
    InAC = np.array([t[5] for t in series], dtype=float)
    InComp = np.array([t[8] for t in series], dtype=float)

    tag = f"Nx{Nx}_Ny{Ny}_g{g}"

    plt.figure(figsize=(7.2,4.6))
    plt.plot(ws, S_AB, "o--", label="S(AB)")
    plt.plot(ws, S_BC, "s--", label="S(BC)")
    plt.plot(ws, S_B, "d--", label="S(B)")
    plt.plot(ws, S_ABC, "^--", label="S(ABC)")
    plt.grid(True, alpha=0.3)
    plt.xlabel("wall␣thickness␣w␣(graph␣steps␣around␣shortest␣A-C␣path)")
    plt.ylabel("entropy␣(nats)")
    plt.title(f"Z2␣LGT␣OBC␣{Nx}x{Ny}:␣entropy␣terms␣vs␣w␣(wall)␣({tag})")
    plt.legend()
    plt.tight_layout()
    plt.savefig(f"{out_prefix}_{tag}_entropies_wall.png", dpi=220)
    plt.close()

    plt.figure(figsize=(7.2,4.6))
    plt.plot(ws, Ivals, "o-", label="I(A:C|B)")
    plt.grid(True, alpha=0.3)
    plt.xlabel("wall␣thickness␣w")
    plt.ylabel("CMI␣(nats)")
    plt.title(f"Z2␣LGT␣OBC␣{Nx}x{Ny}:␣CMI␣vs␣w␣(wall)␣({tag})")
    plt.legend()
    plt.tight_layout()
    plt.savefig(f"{out_prefix}_{tag}_cmi_wall.png", dpi=220)
    plt.close()

    plt.figure(figsize=(7.2,4.6))
    plt.semilogy(ws, np.maximum(Ivals, 1e-16), "o-")
    plt.grid(True, which="both", alpha=0.3)
    plt.xlabel("wall␣thickness␣w")
    plt.ylabel("CMI␣(nats),␣log␣scale␣(clamped)")
    plt.title(f"Z2␣LGT␣OBC␣{Nx}x{Ny}:␣semilog␣CMI␣vs␣w␣(wall)␣({tag})")
    plt.tight_layout()
    plt.savefig(f"{out_prefix}_{tag}_cmi_wall_semilog.png", dpi=220)
    plt.close()

    plt.figure(figsize=(7.2,4.6))
    plt.plot(ws, InAC, "o-", label="I␣/␣sigma_AC␣(B␣vs␣AUC)")
    plt.plot(ws, InComp, "s-", label="I␣/␣sigma_comp␣(B␣vs␣B^c)")
    plt.grid(True, alpha=0.3)
    plt.xlabel("wall␣thickness␣w")
    plt.ylabel("normalized␣CMI␣(dimensionless)")
    plt.title(f"Z2␣LGT␣OBC␣{Nx}x{Ny}:␣normalized␣CMI␣vs␣w␣(wall)␣({tag})")
    plt.legend()
```

```
    plt.tight_layout()
    plt.savefig(f"{out_prefix}_{tag}_cmi_over_sigma_wall.png", dpi=220)
    plt.close()

    plt.figure(figsize=(7.2,4.6))
    plt.plot(ws, sigAC, "o-", label="sigma_AC␣=␣|∂AC|␣log␣2")
    plt.plot(ws, sigComp, "s-", label="sigma_comp␣=␣|∂comp|␣log␣2")
    plt.grid(True, alpha=0.3)
    plt.xlabel("wall␣thickness␣w")
    plt.ylabel("sigma␣(nats)")
    plt.title(f"Z2␣LGT␣OBC␣{Nx}x{Ny}:␣boundary␣prefactors␣vs␣w␣(wall)␣({tag})")
    plt.legend()
    plt.tight_layout()
    plt.savefig(f"{out_prefix}_{tag}_sigmas_wall.png", dpi=220)
    plt.close()

    return rows

def run_all():
    header = [
        "Nx","Ny","Nlinks","g","Lambda","E0","min<Gv>","mean<Gv>",
        "w","|A|","|B|","|C|","|path|",
        "|∂AC|","sigma_AC","I_over_sigma_AC",
        "|∂comp|","sigma_comp","I_over_sigma_comp",
        "S_AB","S_BC","S_B","S_ABC","I_CMI"
    ]

    rows_all = []
    for g in G_LIST:
        if RUN_2x2:
            rows_all += compute_wall_series_pure(2,2,g,LAMBDA_GAUSS,W_LIST,ENT_EPS,
                OUT_PREFIX)
        if RUN_2x3:
            rows_all += compute_wall_series_pure(2,3,g,LAMBDA_GAUSS,W_LIST,ENT_EPS,
                OUT_PREFIX)

    csv_path = f"{OUT_PREFIX}_results.csv"
    with open(csv_path, "w", newline="", encoding="utf-8") as f:
        wr = csv.writer(f)
        wr.writerow(header)
        wr.writerows(rows_all)

    print(f"\nWrote␣CSV:␣{csv_path}")
    print(f"Generated␣PNGs␣with␣prefix:␣{OUT_PREFIX}")
    return rows_all

rows = run_all()
rows[:3]
```

A.2 Script 2: convert CSV to a LaTeX table + zip for Overleaf

This helper script reads `z2_cmi_wall_results.csv`, writes `table_ed_wall.tex`, and produces a root-only zip for Overleaf. It requires `pandas` (available by default in Colab).

```
# make_table_and_zip.py (Colab/Jupyter)
import os, glob, zipfile
import pandas as pd

```

```python
CSV_PATH = "z2_cmi_wall_results.csv"
TABLE_TEX = "table_ed_wall.tex"
MAIN_TEX = "main.tex"
OUT_ZIP = "overleaf_root_upload.zip"

# --- 1) Make a small LaTeX table (filter example: Nx=2, Ny=3, g=0.5, w<=3)
df = pd.read_csv(CSV_PATH)
sdf = df[(df["Nx"]==2) & (df["Ny"]==3) & (df["g"]==0.5) & (df["w"]<=3)].sort_values("w")

cols = ["w","|path|","|B|","|∂comp|","sigma_comp","I_CMI","I_over_sigma_comp"]
cols = [c for c in cols if c in sdf.columns]

def fmt(x):
    try:
        return f"{float(x):.7g}"
    except Exception:
        return str(x)

lines = []
for _, r in sdf.iterrows():
    lines.append(" & ".join(fmt(r[c]) for c in cols) + r" \\")

table = r"""\begin{table}[ht]
\centering
\caption{Wall geometry summary (filtered from CSV).}
\begin{tabular}{%s}
\toprule
%s \\
\midrule
%s
\bottomrule
\end{tabular}
\end{table}
""" % ("r"*len(cols), " & ".join(cols), "\n".join(lines))

with open(TABLE_TEX, "w", encoding="utf-8") as f:
    f.write(table)

print("Wrote:", TABLE_TEX)

# --- 2) Zip root-only for Overleaf
pngs = sorted(glob.glob("z2_cmi_wall_*.png"))
pys = sorted(glob.glob("*.py"))
csvs = sorted(glob.glob("*.csv"))
texs = [MAIN_TEX, TABLE_TEX]

files = []
for fp in texs + pngs + pys + csvs:
    if os.path.isfile(fp):
        files.append(fp)

with zipfile.ZipFile(OUT_ZIP, "w", compression=zipfile.ZIP_DEFLATED) as z:
    for fp in sorted(set(files)):
        z.write(fp, arcname=os.path.basename(fp))

print("Wrote:", OUT_ZIP)
```

References

[1] J. B. Kogut, An introduction to lattice gauge theory and spin systems, *Rev. Mod. Phys.* **51**, 659–713 (1979).

[2] D. Petz, Sufficient subalgebras and the relative entropy of states of a von Neumann algebra, *Commun. Math. Phys.* **105**, 123–131 (1986).

[3] O. Fawzi and R. Renner, Quantum conditional mutual information and approximate Markov chains, *Commun. Math. Phys.* **340**, 575–611 (2015).