In [15]:
```
pip install rpy2
```

```
Requirement already satisfied: rpy2 in /opt/anaconda3/lib/python3.13/site-pa
ckages (3.6.4)
Requirement already satisfied: rpy2-rinterface>=3.6.3 in /opt/anaconda3/lib/
python3.13/site-packages (from rpy2) (3.6.3)
Requirement already satisfied: rpy2-robjects>=3.6.3 in /opt/anaconda3/lib/py
thon3.13/site-packages (from rpy2) (3.6.3)
Requirement already satisfied: cffi>=1.15.1 in /opt/anaconda3/lib/python3.1
3/site-packages (from rpy2-rinterface>=3.6.3->rpy2) (1.17.1)
Requirement already satisfied: pycparser in /opt/anaconda3/lib/python3.13/si
te-packages (from cffi>=1.15.1->rpy2-rinterface>=3.6.3->rpy2) (2.21)
Requirement already satisfied: jinja2 in /opt/anaconda3/lib/python3.13/site-
packages (from rpy2-robjects>=3.6.3->rpy2) (3.1.6)
Requirement already satisfied: tzlocal in /opt/anaconda3/lib/python3.13/site
-packages (from rpy2-robjects>=3.6.3->rpy2) (5.3.1)
Requirement already satisfied: MarkupSafe>=2.0 in /opt/anaconda3/lib/python
3.13/site-packages (from jinja2->rpy2-robjects>=3.6.3->rpy2) (3.0.2)
Note: you may need to restart the kernel to use updated packages.
```

In [20]:
```python
import os
os.environ["R_HOME"] = "/Library/Frameworks/R.framework/Resources"
```

In [21]:
```python
print("R_HOME =", os.environ.get("R_HOME"))
```

```
R_HOME = /Library/Frameworks/R.framework/Resources
```

In [22]:
```python
import numpy as np
import itertools
from math import log, sqrt
from collections import defaultdict
from dataclasses import dataclass

# If you don't have SciPy, replace this with an approximation
from scipy.stats import norm


# ----------------------------------------------------------------------
# 1. Conditional independence test: Fisher-Z for Gaussian data
# ----------------------------------------------------------------------

def fisher_z_test(Z, i, j, cond, alpha):
    """
    Gaussian CI test using Fisher-Z.
    Z:   data matrix (n_samples, n_nodes)
    i,j: indices of variables
    cond: iterable of indices (conditioning set)
    alpha: significance level
    Returns: True if X_i ⊥ X_j | cond  (independent)
    """
    n, p = Z.shape
    var_idx = [i, j] + list(cond)
    sub = Z[:, var_idx]
    C = np.cov(sub, rowvar=False)
    # precision
```

```python
    try:
        K = np.linalg.inv(C)
    except np.linalg.LinAlgError:
        # fall back to pseudo-inverse
        K = np.linalg.pinv(C)
    # partial correlation between first two
    r = -K[0, 1] / np.sqrt(K[0, 0] * K[1, 1])
    r = max(min(r, 0.999999), -0.999999)
    z = 0.5 * log((1 + r) / (1 - r)) * sqrt(max(n - len(cond) - 3, 1))
    zcrit = norm.ppf(1 - alpha / 2.0)
    return abs(z) <= zcrit


# ---------------------------------------------------------------------
# 2. Dynamic PAG graph with homology (SVAR structure)
# ---------------------------------------------------------------------

@dataclass
class NodeInfo:
    var: int
    lag: int


class DynamicPAG:
    """
    Dynamic PAG segment for X_t, ..., X_{t-p}.
    Nodes are indexed 0..(k*(p+1)-1).
    Each node has (var, lag). lag=0 is time t (most recent).
    marks[i,j] is the endpoint mark at j for edge i--j:
        0  = circle (o)
        -1 = tail   (-)
        1  = arrow  (>)
    """
    def __init__(self, var_names, max_lag):
        self.var_names = list(var_names)
        self.k = len(var_names)
        self.p = max_lag
        self.n_nodes = self.k * (self.p + 1)

        # full complete graph with o-o edges
        self.adj = np.ones((self.n_nodes, self.n_nodes), dtype=bool)
        np.fill_diagonal(self.adj, False)
        self.marks = np.zeros((self.n_nodes, self.n_nodes), dtype=int)

        # separation sets: key = (min(i,j), max(i,j)), value = set of nodes
        self.sepset = {}

    # ----- node indexing / decoding -----

    def node_index(self, var, lag):
        return lag * self.k + var

    def decode_node(self, idx):
        lag = idx // self.k
        var = idx % self.k
        return NodeInfo(var=var, lag=lag)
```

```python
    def node_label(self, idx):
        info = self.decode_node(idx)
        return f"{self.var_names[info.var]}_lag{info.lag}"

    # ----- homology: pairs with same var pair + same lag difference -----

    def hom_pairs(self, i, j):
        info_i = self.decode_node(i)
        info_j = self.decode_node(j)
        d = info_i.lag - info_j.lag
        pairs = []
        for a in range(self.p + 1):
            b = a - d
            if 0 <= b <= self.p:
                m = self.node_index(info_i.var, a)
                n = self.node_index(info_j.var, b)
                pairs.append((m, n))
        return pairs

    # ----- adjacency restricted by time (adj_t) -----

    def neighbors(self, i):
        return [j for j in range(self.n_nodes) if self.adj[i, j]]

    def adj_t(self, i):
        info_i = self.decode_node(i)
        res = []
        for j in self.neighbors(i):
            info_j = self.decode_node(j)
            if info_j.lag >= info_i.lag:
                res.append(j)
        return res

    # ----- sepsets -----

    def set_sepset(self, i, j, S):
        if i > j:
            i, j = j, i
        self.sepset[(i, j)] = set(S)

    def get_sepset(self, i, j):
        if i > j:
            i, j = j, i
        return self.sepset.get((i, j), set())

    # ----- edge operations with homology -----

    def delete_edge_with_homology(self, i, j):
        for m, n in self.hom_pairs(i, j):
            if self.adj[m, n]:
                self.adj[m, n] = False
                self.adj[n, m] = False
                self.marks[m, n] = 0
                self.marks[n, m] = 0
```

```python
    def _orient_edge(self, i, j, mark_ij, mark_ji):
        """
        Set endpoint marks for edge i-j without touching homology.
        mark_ij is mark at j on edge from i to j.
        """
        if not self.adj[i, j]:
            return
        # simple consistency check: don't overwrite a hard arrow in opposite
        if self.marks[j, i] == 1 and mark_ij == 1:
            # would make i <-> j when there is already arrow at i
            pass
        self.marks[i, j] = mark_ij
        self.marks[j, i] = mark_ji

    def orient_with_homology(self, i, j, mark_ij, mark_ji):
        """
        Orient edge (i,j) and all homologous edges with same endpoint patter
        For example, for i *-> j we pass (1, -1) or (1, 0), etc.
        """
        for m, n in self.hom_pairs(i, j):
            if not self.adj[m, n]:
                continue
            self._orient_edge(m, n, mark_ij, mark_ji)

    def reset_all_to_oo(self):
        """Keep adjacency but set all marks to circle (o-o)."""
        self.marks[:, :] = 0

    # ----- helpers for collider / triangle checks -----

    def is_collider(self, a, b, c):
        """
        a ?-> b <-? c
        """
        if not (self.adj[a, b] and self.adj[b, c]):
            return False
        return self.marks[a, b] == 1 and self.marks[c, b] == 1

    def forms_triangle(self, a, b, c):
        return self.adj[a, b] and self.adj[b, c] and self.adj[a, c]


# ---------------------------------------------------------------------
# 3. Dynamic pds_s (time-restricted possible-d-sep)
# ---------------------------------------------------------------------

def pds_s(graph: DynamicPAG, i, j):
    """
    Time-restricted possible-d-sep set pds_s(X_i, X_j, P).
    This is a faithful adaptation of Zhang's pds definition,
    restricted to nodes whose lag <= max(lag(i), lag(j)).
    """
    info_i = graph.decode_node(i)
    info_j = graph.decode_node(j)
    maxlag = max(info_i.lag, info_j.lag)
```

```
    # BFS over paths satisfying collider-or-triangle condition
    result = set()
    queue = [[i]]
    visited_paths = set()

    def path_ok(path):
        # all internal triples (a,b,c) must be collider or triangle
        if len(path) < 3:
            return True
        for a, b, c in zip(path[:-2], path[1:-1], path[2:]):
            if not (graph.is_collider(a, b, c) or graph.forms_triangle(a, b,
                return False
        return True

    while queue:
        path = queue.pop(0)
        last = path[-1]
        for nb in graph.neighbors(last):
            if nb in path:
                continue
            new_path = path + [nb]
            key = tuple(new_path)
            if key in visited_paths:
                continue
            visited_paths.add(key)
            if not path_ok(new_path):
                continue

            info_nb = graph.decode_node(nb)
            if info_nb.lag <= maxlag:
                result.add(nb)
            queue.append(new_path)

    result.discard(i)
    result.discard(j)
    return result


# ----------------------------------------------------------------------
# 4. SVAR-FCI Algorithm 3.1
# ----------------------------------------------------------------------

class SVAR_FCI:
    def __init__(self, alpha=0.05, max_lag=2, verbose=False):
        self.alpha = alpha
        self.max_lag = max_lag
        self.verbose = verbose
        self.graph_ = None
        self.var_names_ = None
        self.Z_ = None  # lagged data

    # --- lagged data builder (X_t,...,X_{t-p}) ---

    def _build_lagged_matrix(self, X, var_names):
        """
        X: np.ndarray (T, k)
```

```
        Returns Z: (T-p, k*(p+1)), names: list[str]
        """
        T, k = X.shape
        p = self.max_lag
        rows = T - p
        Z = np.zeros((rows, k * (p + 1)))
        names = []
        for lag in range(p + 1):
            Z[:, lag * k:(lag + 1) * k] = X[p - lag:T - lag, :]
            for idx, name in enumerate(var_names):
                names.append(f"{name}_lag{lag}")
        return Z, names

    # --- independence wrapper ---

    def _indep(self, Z, i, j, S):
        return fisher_z_test(Z, i, j, S, self.alpha)

    # --- skeleton phase (Alg 3.1 lines 3–8) ---

    def _skeleton_phase(self, G: DynamicPAG, Z):
        if self.verbose:
            print("Skeleton phase (dynamic adj_t + homology)...")
        n = 0
        p = G.n_nodes
        changed = True
        while changed:
            changed = False
            if self.verbose:
                print(f"  Conditioning set size n={n}")
            for i in range(p):
                for j in range(i + 1, p):
                    if not G.adj[i, j]:
                        continue
                    # we test only with Xi as "left" node (as in Alg 3.1)
                    adj_i_t = [v for v in G.adj_t(i) if v != j]
                    if len(adj_i_t) < n:
                        continue
                    found_sep = False
                    for S in itertools.combinations(adj_i_t, n):
                        if self._indep(Z, i, j, S):
                            if self.verbose:
                                print(f"    indep({G.node_label(i)}, {G.node
                            G.delete_edge_with_homology(i, j)
                            G.set_sepset(i, j, S)
                            changed = True
                            found_sep = True
                            break
                    if found_sep:
                        continue
            n += 1

    # --- line 9: time orientation Xi_t *-> Xj_s if s > t ---

    # --- line 9: time orientation Xi_t *-> Xj_s if s > t (past -> future) -
```

```
    def _time_orientation(self, G: DynamicPAG):
        """
        Deterministic time-based orientation:

        We use lag = 0 for time t (most recent), lag = 1 for t-1, etc.
        So a larger lag means an *earlier* time.

        The SVAR-FCI rule Xi,t *-> Xj,s iff s > t (later in calendar time)
        therefore translates to:

            if lag(i) > lag(j):   # i is further in the past than j
                orient Xi,lag(i) *-> Xj,lag(j)

        i.e. edges always point from past -> future.
        """
        if self.verbose:
            print("Time orientation (past -> future: larger lag -> smaller l
        n = G.n_nodes
        for i in range(n):
            info_i = G.decode_node(i)
            for j in range(n):
                if not G.adj[i, j]:
                    continue
                info_j = G.decode_node(j)

                # i earlier in time than j  <=>  lag(i) > lag(j)
                if info_i.lag > info_j.lag:
                    # orient i o-> j (circle at i, arrow at j), with homolog
                    if G.marks[i, j] == 0 and G.marks[j, i] == 0:
                        G.orient_with_homology(i, j, 1, 0)


    # --- line 10: v-structures with homology ---

    def _orient_v_structures(self, G: DynamicPAG):
        if self.verbose:
            print("Orienting v-structures...")
        p = G.n_nodes
        for k in range(p):
            for i in range(p):
                if i == k or not G.adj[i, k]:
                    continue
                for j in range(i + 1, p):
                    if j == k or not G.adj[j, k]:
                        continue
                    if G.adj[i, j]:
                        continue  # shielded
                    S = G.get_sepset(i, j)
                    if k not in S:
                        # orient i *-> k <-* j
                        G.orient_with_homology(i, k, 1, -1)
                        G.orient_with_homology(j, k, 1, -1)

    # --- line 11: second deletion using dynamic pds_s + homology ---

    def _pds_deletion_phase(self, G: DynamicPAG, Z):
```

```python
        if self.verbose:
            print("pds_s deletion phase with homology...")
        n = 0
        p = G.n_nodes
        changed = True
        while changed:
            changed = False
            if self.verbose:
                print(f"  pds_s, conditioning size n={n}")
            for i in range(p):
                for j in range(i + 1, p):
                    if not G.adj[i, j]:
                        continue
                    # candidate conditioning sets from pds_s
                    P1 = pds_s(G, i, j)
                    P2 = pds_s(G, j, i)
                    P_union = list(P1.union(P2))
                    if len(P_union) < n:
                        continue
                    found_sep = False
                    for S in itertools.combinations(P_union, n):
                        if self._indep(Z, i, j, S):
                            if self.verbose:
                                print(f"    pds indep({G.node_label(i)}, {G.
                            G.delete_edge_with_homology(i, j)
                            G.set_sepset(i, j, S)
                            changed = True
                            found_sep = True
                            break
                    if found_sep:
                        continue
            n += 1

    # --- R1–R10 placeholder (needs full Zhang implementation) ---

        # --- R1–R10 (Zhang 2008) orientation rules on DynamicPAG ---
    # Assumptions:
    # - G.marks[i,j] is mark at endpoint j on edge (i,j):
    #       0  = circle (o)
    #      -1  = tail   (-)
    #       1  = arrow  (>)
    # - G.adj[i,j] is True iff there is an edge between i and j.
    # - G.orient_with_homology(i,j, mark_ij, mark_ji) orients (i,j) and
    #   all homologous edges with endpoint marks mark_ij at j, mark_ji at i.
    #
    # We implement R1–R4 and R8–R10 exactly in Zhang (2008), restricted
    # to MAGs without undirected edges (no selection bias), so R5–R7
    # are omitted as per Zhang’s remark.

    def _apply_R_rules(self, G: DynamicPAG):
        n = G.n_nodes

        def non_adjacent(a, b):
            return not G.adj[a, b]

        def is_arrow_into(child, parent):
```

```
        # edge parent *-> child  ⇔ mark at child on (parent, child) is 1
        return G.adj[parent, child] and G.marks[parent, child] == 1

    def is_circle_at(i, j):
        # circle at j on edge (i,j)
        return G.adj[i, j] and G.marks[i, j] == 0

    # --- R4 helpers: discriminating paths ---

    def is_discriminating_path(path, V, X, Y):
        """
        Check if 'path' (list of nodes [X,...,W,V,Y]) is a
        discriminating path for V between X and Y, in the sense of
        Zhang Def. 7 but adapted to the current PAG:
        - length >= 3 edges (>= 4 nodes)
        - V non-endpoint and adjacent to Y on path
        - X not adjacent to Y
        - every vertex between X and V is:
            * a collider on the path
            * a parent of Y (edge *-> Y).
        """
        if len(path) < 4:
            return False
        if path[0] != X or path[-1] != Y:
            return False
        if V not in path[1:-1]:
            return False
        if not non_adjacent(X, Y):
            return False

        V_idx = path.index(V)
        # V must be non-endpoint and directly before Y on the path
        if V_idx == 0 or V_idx == len(path) - 1:
            return False
        if path[V_idx + 1] != Y:
            return False

        # Every vertex between X and V must be a collider on the path
        # and must be a parent of Y (*-> Y).
        for idx in range(1, V_idx):
            v = path[idx]
            prev_v = path[idx - 1]
            next_v = path[idx + 1]
            # collider on the path
            if not G.is_collider(prev_v, v, next_v):
                return False
            # parent of Y: edge v *-> Y  ⇒ mark at Y on (v,Y) is 1
            if not is_arrow_into(Y, v):
                return False

        return True

    def all_simple_paths_unbounded(start, end):
        """
        Generate all simple paths from start to end (no repeated nodes).
        Use with care; graph is sparse after skeleton + pds, so it is
```

```
        manageable for typical macro VAR sizes.
        """
        stack = [(start, [start])]
        while stack:
            (v, path) = stack.pop()
            for w in G.neighbors(v):
                if w in path:
                    continue
                new_path = path + [w]
                if w == end:
                    yield new_path
                else:
                    stack.append((w, new_path))

    changed = True
    while changed:
        changed = False

        # ----------------- R1 -----------------
        # R1: If α *-> β o--* γ and α,γ nonadjacent, then orient β o--*
        for beta in range(n):
            for alpha in G.neighbors(beta):
                if not is_arrow_into(beta, alpha):
                    continue  # need α *-> β
                for gamma in G.neighbors(beta):
                    if gamma == alpha:
                        continue
                    # β o--* γ  -> circle at β on edge (β,γ)
                    if not is_circle_at(beta, gamma):
                        continue
                    if not non_adjacent(alpha, gamma):
                        continue
                    # orient β -> γ  (tail at β, arrow at γ)
                    if not (G.marks[beta, gamma] == 1 and G.marks[gamma,
                        G.orient_with_homology(beta, gamma, 1, -1)
                        changed = True

        # ----------------- R2 -----------------
        # R2: If α -> β *-> γ (or α *-> β -> γ) and α *-◦ γ, orient α *-
        for beta in range(n):
            for alpha in G.neighbors(beta):
                for gamma in G.neighbors(beta):
                    if gamma == alpha:
                        continue
                    # require β *-> γ and α *-> β (arrowheads into β and
                    if not (is_arrow_into(beta, alpha) and is_arrow_into
                        continue
                    if not G.adj[alpha, gamma]:
                        continue
                    # α *-◦ γ  => circle at γ on (α,γ)
                    if G.marks[gamma, alpha] != 0:
                        continue
                    # orient α o-> γ: arrow at γ, keep circle at α
                    if not (G.marks[alpha, gamma] == 1 and G.marks[gamma
                        G.orient_with_homology(alpha, gamma, 1, 0)
                        changed = True
```

```
        # ------------------ R3 ------------------
        # R3: If α *-> β <-* γ, α *-◦ θ ◦-* γ, α,γ nonadjacent,
        #     and θ *-◦ β, then orient θ *-◦ β as θ *-> β.
        for beta in range(n):
            # collider α *-> β <-* γ
            for alpha in G.neighbors(beta):
                if not is_arrow_into(beta, alpha):
                    continue
                for gamma in G.neighbors(beta):
                    if gamma == alpha:
                        continue
                    if not is_arrow_into(beta, gamma):
                        continue
                    if not non_adjacent(alpha, gamma):
                        continue
                    # search θ such that α *-◦ θ ◦-* γ and θ *-◦ β
                    for theta in range(n):
                        if theta in (alpha, beta, gamma):
                            continue
                        if not (G.adj[alpha, theta] and G.adj[theta, gam
                            continue
                        # circles at θ on α–θ and γ–θ
                        if G.marks[alpha, theta] != 0 or G.marks[gamma,
                            continue
                        # circle at β on θ–β
                        if G.marks[theta, beta] != 0:
                            continue
                        # orient θ o-> β: arrow at β, circle at θ
                        if not (G.marks[theta, beta] == 1 and G.marks[be
                            G.orient_with_homology(theta, beta, 1, 0)
                            changed = True

        # ------------------ R4 ------------------
        # R4: If u = <θ,...,α,β,γ> is a discriminating path between θ ar
        #     and β o--* γ, then:
        #     - if β ∈ Sepset(θ,γ), orient β o--* γ as β -> γ
        #     - else orient <α,β,γ> as α <-> β <-> γ.
        for beta in range(n):
            for theta in range(n):
                if theta == beta:
                    continue
                for gamma in range(n):
                    if gamma in (theta, beta):
                        continue
                    if not G.adj[beta, gamma]:
                        continue
                    # β o--* γ : circle at β on edge (β,γ)
                    if G.marks[beta, gamma] != 0:
                        continue
                    # search discriminating paths θ ... β γ
                    for path in all_simple_paths_unbounded(theta, gamma)
                        if beta not in path:
                            continue
                        if not is_discriminating_path(path, beta, theta,
                            continue
```

```
                    # found discriminating path
                    sepset = G.get_sepset(theta, gamma)
                    if beta in sepset:
                        # β -> γ
                        if not (G.marks[beta, gamma] == 1 and G.mark
                            G.orient_with_homology(beta, gamma, 1, -
                            changed = True
                    else:
                        # α <-> β <-> γ, where α is predecessor of β
                        b_idx = path.index(beta)
                        if b_idx == 0:
                            continue
                        alpha = path[b_idx - 1]
                        # α <-> β
                        if not (G.marks[alpha, beta] == 1 and G.mark
                            G.orient_with_homology(alpha, beta, 1, 1
                            changed = True
                        # β <-> γ
                        if not (G.marks[beta, gamma] == 1 and G.mark
                            G.orient_with_homology(beta, gamma, 1, 1
                            changed = True
                    break  # only need one discriminating path

        # ------------------ R5 ------------------
        # If α - β (undirected) and exists γ -> α with γ not adjacent to
        # orient β *-> α (arrowhead at α).
        for alpha in range(n):
            for beta in range(n):
                if alpha == beta:
                    continue
                # alpha — beta   (both tails)
                if not (G.adj[alpha, beta] and
                        G.marks[alpha, beta] == -1 and
                        G.marks[beta, alpha] == -1):
                    continue
                for gamma in range(n):
                    if gamma in (alpha, beta):
                        continue
                    # γ -> α  (arrowhead at α)
                    if not (G.adj[gamma, alpha] and G.marks[gamma, alpha
                        continue
                    # γ not adjacent to β
                    if G.adj[gamma, beta]:
                        continue
                    # orient β *-> α : tail at β, arrow at α
                    if not (G.marks[beta, alpha] == 1 and G.marks[alpha,
                        G.orient_with_homology(beta, alpha, 1, -1)
                        changed = True
                        break
                    # ------------------ R6 ------------------
        # If α - β and there exists a discriminating path for α between
        # some γ and β, orient β *-> α.
        for alpha in range(n):
            for beta in range(n):
                if alpha == beta:
                    continue
```

```
            # α — β
            if not (G.adj[alpha, beta] and
                    G.marks[alpha, beta] == -1 and
                    G.marks[beta, alpha] == -1):
                continue

            # search discriminating paths <γ, ..., α, β>
            for gamma in range(n):
                if gamma in (alpha, beta):
                    continue

                for path in all_simple_paths_unbounded(gamma, beta):
                    # path must end with ..., α, β
                    if len(path) < 3:
                        continue
                    if path[-2] != alpha:
                        continue

                    if is_discriminating_path(path, alpha, gamma, be
                        # orient β *-> α
                        if not (G.marks[beta, alpha] == 1 and G.mark
                            G.orient_with_homology(beta, alpha, 1, -
                            changed = True
                        break
                if changed:
                    break
                # ----------------- R7 -----------------
        # If α — β cannot be oriented by R5 or R6 but remains undirected
        # orient α <-> β (bidirected).
        for alpha in range(n):
            for beta in range(n):
                if alpha == beta:
                    continue
                # still undirected?
                if not (G.adj[alpha, beta] and
                        G.marks[alpha, beta] == -1 and
                        G.marks[beta, alpha] == -1):
                    continue
                # convert to bidirected α <-> β
                G.orient_with_homology(alpha, beta, 1, 1)
                changed = True


        # ----------------- R8 -----------------
        # R8: If α -> β -> γ or α -◦ β -> γ, and α ◦-> γ,
        #     orient α ◦-> γ as α -> γ.
        for alpha in range(n):
            for beta in G.neighbors(alpha):
                for gamma in G.neighbors(beta):
                    if gamma == alpha:
                        continue
                    # β -> γ
                    if not is_arrow_into(gamma, beta):
                        continue
                    # either α -> β or α -◦ β
                    cond1 = is_arrow_into(beta, alpha)  # α *-> β
```

```
            cond2 = (G.adj[alpha, beta] and
                     G.marks[alpha, beta] == 0 and  # circle at
                     G.marks[beta, alpha] == -1)   # tail at α
            if not (cond1 or cond2):
                continue
            # α ◦-> γ : circle at α, arrow at γ
            if not (G.adj[alpha, gamma] and
                    G.marks[alpha, gamma] == 1 and
                    G.marks[gamma, alpha] == 0):
                continue
            # orient α -> γ: tail at α, arrow at γ
            if not (G.marks[alpha, gamma] == 1 and G.marks[gamma
                G.orient_with_homology(alpha, gamma, 1, -1)
                changed = True

# -------- helpers for R9–R10: uncovered p.d. paths ----------

def is_uncovered(path):
    # every consecutive triple unshielded: Vi-1 and Vi+1 nonadja
    if len(path) < 3:
        return True
    for i in range(1, len(path) - 1):
        if G.adj[path[i - 1], path[i + 1]]:
            return False
    return True

def is_pd_edge(u, v):
    # edge u ?-? v is potentially directed from u to v
    # if there's no arrowhead into u along u->v
    return G.adj[u, v] and (G.marks[v, u] != 1)

def uncovered_pd_paths(start, end):
    stack = [(start, [start])]
    while stack:
        (v, path) = stack.pop()
        for w in G.neighbors(v):
            if w in path:
                continue
            if not is_pd_edge(v, w):
                continue
            new_path = path + [w]
            if not is_uncovered(new_path):
                continue
            if w == end:
                yield new_path
            else:
                stack.append((w, new_path))

# ----------------- R9 -----------------
# R9: If α ◦-> γ, and there is an uncovered p.d. path
#     p = <α, β, θ, ..., γ> from α to γ such that β and γ
#     are not adjacent, then orient α ◦-> γ as α -> γ.
for alpha in range(n):
    for gamma in range(n):
        if gamma == alpha:
            continue
```

```
            # α ◦-> γ
            if not (G.adj[alpha, gamma] and
                    G.marks[alpha, gamma] == 1 and
                    G.marks[gamma, alpha] == 0):
                continue
            for path in uncovered_pd_paths(alpha, gamma):
                if len(path) < 3:
                    continue
                beta = path[1]
                if not non_adjacent(beta, gamma):
                    continue
                # orient α -> γ
                if not (G.marks[alpha, gamma] == 1 and G.marks[gamma
                    G.orient_with_homology(alpha, gamma, 1, -1)
                    changed = True
                break  # one path is enough

    # ----------------- R10 -----------------
    # R10: Suppose α ◦-> γ, β -> γ <- θ,
    #       p1 is uncovered p.d. path α ... β,
    #       p2 is uncovered p.d. path α ... θ,
    #       let μ be neighbor of α on p1, ω neighbor of α on p2,
    #       if μ ≠ ω and μ,ω nonadjacent, orient α ◦-> γ as α -> γ.
    for alpha in range(n):
        for gamma in range(n):
            if gamma == alpha:
                continue
            # α ◦-> γ
            if not (G.adj[alpha, gamma] and
                    G.marks[alpha, gamma] == 1 and
                    G.marks[gamma, alpha] == 0):
                continue
            # nodes β, θ with β -> γ <- θ
            parents = [v for v in range(n) if is_arrow_into(gamma, v
            for beta in parents:
                for theta in parents:
                    if theta == beta or theta == alpha or beta == al
                        continue
                    # p1: uncovered p.d. from α to β
                    p1_list = list(uncovered_pd_paths(alpha, beta))
                    if not p1_list:
                        continue
                    # p2: uncovered p.d. from α to θ
                    p2_list = list(uncovered_pd_paths(alpha, theta))
                    if not p2_list:
                        continue
                    # take first such paths
                    p1 = p1_list[0]
                    p2 = p2_list[0]
                    if len(p1) < 2 or len(p2) < 2:
                        continue
                    mu = p1[1]
                    omega = p2[1]
                    if mu == omega:
                        continue
                    if not non_adjacent(mu, omega):
```

```
                            continue
                        # orient α -> γ
                        if not (G.marks[alpha, gamma] == 1 and G.marks[g
                            G.orient_with_homology(alpha, gamma, 1, -1)
                            changed = True
                        break  # β,θ found that trigger R10
                    if changed:
                        break
                # end loops over beta, theta
        # end while changed


    # --- main entry point ---

    def fit(self, X, var_names=None):
        """
        X: numpy array (T, k)
        var_names: list of length k
        """
        X = np.asarray(X)
        T, k = X.shape
        if var_names is None:
            var_names = [f"X{i}" for i in range(k)]
        self.var_names_ = var_names

        # build lagged matrix
        Z, lagged_names = self._build_lagged_matrix(X, var_names)
        self.Z_ = Z

        # init dynamic PAG
        G = DynamicPAG(lagged_names, self.max_lag)

        assert G.n_nodes == Z.shape[1], (
        f"ERROR: Graph has {G.n_nodes} nodes but Z has {Z.shape[1]} columns.
        f"Check R5–R7 indentation and DynamicPAG construction.")

        # Algorithm 3.1 steps
        # 1–2: done via initialization
        # 3–8: skeleton
        self._skeleton_phase(G, Z)

        # 9: time orientation
        self._time_orientation(G)

        # 10: v-structures
        self._orient_v_structures(G)

        # 11: pds_s deletion
        self._pds_deletion_phase(G, Z)

        # 12: reset o-o and repeat 9–10
        G.reset_all_to_oo()
        self._time_orientation(G)
        self._orient_v_structures(G)

        # 13: R1–R10
```

```
        self._apply_R_rules(G)

        self.graph_ = G
        return self
```

In [23]:
```
# ----------------------------------------------------------------------
# 5. ICF/BIC scoring via R ggm
# ----------------------------------------------------------------------

import rpy2.robjects as ro
from rpy2.robjects import numpy2ri
numpy2ri.activate()

# load ggm + helper once
ro.r('library(ggm)')
ro.r("""
icf_bic <- function(S, amat, n) {
  A <- AG(amat, showmat=FALSE)
  fit <- fitAncestralGraph(S, A, n.obs=n)
  return(list(
    loglik = fit$loglik,
    df     = fit$df,
    bic    = -2*fit$loglik + fit$df * log(n)
  ))
}
""")
icf_bic_R = ro.r["icf_bic"]


def pag_to_mag(self, G):
    """
    Convert a PAG into a single valid MAG consistent with:
    - all invariant arrowheads
    - all invariant tails
    - ancestral graph constraints
    - minimal additional orientation

    Returns an adjacency matrix with codes:
      0 = no edge
      1 = i -> j   (directed)
      2 = i <-> j  (bidirected / latent confounding)
      3 = i - j    (undirected adjacency)
    """

    p = G.num_nodes
    amat = np.zeros((p, p), dtype=int)

    def has_arrowhead(i, j):
        return G.marks[i, j] == 1     # arrowhead at j

    def has_tail(i, j):
        return G.marks[i, j] == -1    # tail at j (i *- j)

    for i in range(p):
        for j in range(i + 1, p):
            if not G.adj[i, j]:
```

```
                continue

            # Case 1: Fully oriented edges
            if has_tail(i, j) and has_arrowhead(j, i):
                # i *-> j
                amat[i, j] = 1  # i->j
                continue
            if has_tail(j, i) and has_arrowhead(i, j):
                # j *-> i
                amat[j, i] = 1
                continue

            # Case 2: Invariant arrowheads (one-ended orientation)
            if has_arrowhead(i, j) and not has_arrowhead(j, i):
                # i *--> j means j is not ancestor of i, so orient i <- j
                amat[j, i] = 1
                continue
            if has_arrowhead(j, i) and not has_arrowhead(i, j):
                amat[i, j] = 1
                continue

            # Case 3: Bidirected (latent confounding)
            # PAG: o-> or <-o or o-o could hide latent confounding.
            # If both endpoints uncertain (circles), assign <-> as safe.
            if G.marks[i, j] == 0 and G.marks[j, i] == 0:
                amat[i, j] = amat[j, i] = 2   # i <-> j
                continue

            # Case 4: Undirected edge ambiguous endpoints (tail-circle)
            # Use undirected as minimal encoding.
            amat[i, j] = amat[j, i] = 3

    return amat



def icf_bic_score(Z, G: DynamicPAG):
    """
    Compute ICF/BIC for one SVAR-FCI PAG.
    Z: lagged data (n, p_nodes)  (same Z used in SVAR_FCI)
    G: DynamicPAG
    """
    n = Z.shape[0]
    S = np.cov(Z, rowvar=False)
    amat = pag_to_mag(G)
    res = icf_bic_R(S, amat, n)
    return {
        "loglik": float(res[0][0]),
        "df": float(res[1][0]),
        "bic": float(res[2][0]),
    }
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
Cell In[23], line 5
      1 # ---------------------------------------------------------------------
--
      2 # 5. ICF/BIC scoring via R ggm
      3 # ---------------------------------------------------------------------
--
----> 5 import rpy2.robjects as ro
      6 from rpy2.robjects import numpy2ri
      7 numpy2ri.activate()

File /opt/anaconda3/lib/python3.13/site-packages/rpy2/robjects/__init__.py:2
2
     19 import rpy2.rinterface_lib.openrlib
     20 import rpy2.rlike.container as rlc
---> 22 from rpy2.robjects.robject import RObjectMixin, RObject
     23 import rpy2.robjects.functions
     24 from rpy2.robjects.environments import (Environment,
     25                                         local_context)

File /opt/anaconda3/lib/python3.13/site-packages/rpy2/robjects/robject.py:11
      7 import rpy2.rinterface_lib.callbacks
      9 from rpy2.robjects import conversion
---> 11 rpy2.rinterface.initr()
     14 def _add_warn_reticulate_hook():
     15     msg = """
     16     WARNING: The R package "reticulate" only fixed recently
     17     an issue that caused a segfault when used with rpy2:
   (...)
     20     the fix.
     21     """

File /opt/anaconda3/lib/python3.13/site-packages/rpy2/rinterface/__init__.p
y:1130, in initr(interactive, _want_setcallbacks, _c_stack_limit)
   1127     logger.info('R is already initialized. No need to initialize.')
   1128     return None
-> 1130 _setrenvvars(_ENVVAR_ACTION_MAP)
   1131 if embedded.is_r_externally_initialized():
   1132     embedded._setinitialized()

File /opt/anaconda3/lib/python3.13/site-packages/rpy2/rinterface/__init__.p
y:1222, in _setrenvvars(action_map)
   1220 def _setrenvvars(action_map: typing.Dict[str, _ENVVAR_ACTION]):
   1221     new_envvars = {}
-> 1222     for k, v in _getrenvvars():
   1223         if k in os.environ:
   1224             action = action_map[k]

File /opt/anaconda3/lib/python3.13/site-packages/rpy2/rinterface/__init__.p
y:1175, in _getrenvvars(baselinevars, r_home)
   1173     r_home = openrlib.R_HOME
   1174     if r_home is None:
-> 1175         raise RuntimeError('Unable to determine R_HOME.')
   1177 # Use a temporary file to write the environment variables. Windows
   1178 # has a file locking system that requires a slightly more complicate
```

```
d
   1179 # implementation than it would otherwise be on other OSes.
   1180 temp_fh = tempfile.NamedTemporaryFile(mode='w', delete=False, suffix
='.csv')

RuntimeError: Unable to determine R_HOME.
```

In [ ]:

```python
# ----------------------------------------------------------------------
# 6. Data-driven selection of alpha and p (Appendix)
# ----------------------------------------------------------------------

def select_alpha(
    X,
    var_names,
    p,
    alpha_grid=np.arange(0.01, 0.41, 0.01),
    verbose=False,
):
    """
    For fixed maximum lag p, choose alpha maximizing BIC(P_hat_alpha).
    """
    best_alpha = None
    best_bic = np.inf
    best_model = None
    best_score = None

    for alpha in alpha_grid:
        if verbose:
            print(f"alpha={alpha:.3f}")
        model = SVAR_FCI(alpha=alpha, max_lag=p, verbose=False)
        model.fit(X, var_names=var_names)
        score = icf_bic_score(model.Z_, model.graph_)
        if verbose:
            print("  BIC:", score["bic"])
        if score["bic"] < best_bic:
            best_bic = score["bic"]
            best_alpha = alpha
            best_model = model
            best_score = score

    return best_model, best_alpha, best_score


def select_p(
    X,
    var_names,
    alpha,
    p_grid,
    verbose=False,
):
    """
    For fixed alpha, choose p maximizing BIC(P_hat_p).
    """
    best_p = None
    best_bic = np.inf
    best_model = None
```

```
    best_score = None

    for p in p_grid:
        if verbose:
            print(f"p={p}")
        model = SVAR_FCI(alpha=alpha, max_lag=p, verbose=False)
        model.fit(X, var_names=var_names)
        score = icf_bic_score(model.Z_, model.graph_)
        if verbose:
            print("  BIC:", score["bic"])
        if score["bic"] < best_bic:
            best_bic = score["bic"]
            best_p = p
            best_model = model
            best_score = score

    return best_model, best_p, best_score


def select_alpha_and_p(
    X,
    var_names,
    alpha_grid=np.arange(0.01, 0.41, 0.01),
    p_grid=range(1, 5),
    verbose=False,
):
    """
    Joint search over alpha and p as described in the appendix.
    """
    best_alpha = None
    best_p = None
    best_bic = np.inf
    best_model = None
    best_score = None

    for alpha in alpha_grid:
        for p in p_grid:
            if verbose:
                print(f"alpha={alpha:.3f}, p={p}")
            model = SVAR_FCI(alpha=alpha, max_lag=p, verbose=False)
            model.fit(X, var_names=var_names)
            score = icf_bic_score(model.Z_, model.graph_)
            if verbose:
                print("  BIC:", score["bic"])
            if score["bic"] < best_bic:
                best_bic = score["bic"]
                best_alpha = alpha
                best_p = p
                best_model = model
                best_score = score

    return best_model, best_alpha, best_p, best_score
```

In [ ]:

```
import numpy as np

def simulate_svar_data(T=1000, seed=0):
```

```
    """
    Simulate 3-var VAR(2) with known causal DAG:
        X_{t-1} → Y_t
        Y_{t-1} → Z_t
        Z_{t-1} → X_t
    plus latent confounding:  X_t ↔ Z_t

    Returns:
        X : (T,3) matrix
        var_names : ["X","Y","Z"]
    """
    rng = np.random.default_rng(seed)

    A1 = np.array([
        [0.0, 0.0, 0.4],   # Z_{t-1} → X_t
        [0.3, 0.0, 0.0],   # X_{t-1} → Y_t
        [0.0, 0.2, 0.0]    # Y_{t-1} → Z_t
    ])
    A2 = np.zeros((3,3))   # no second lag effect, but keeps structure simpl

    # latent confounding: U influences X_t and Z_t
    T = T + 5
    X = np.zeros((T,3))
    for t in range(2, T):
        eta = rng.normal(size=3)
        latent = rng.normal()

        X[t] = (
            A1 @ X[t-1]
            + A2 @ X[t-2]
            + eta
            + np.array([latent, 0, latent]) * 0.5
        )

    return X[5:], ["X","Y","Z"]
```

In [ ]:

```
def test_svar_fci_once():
    X, names = simulate_svar_data(T=1200, seed=42)
    model = SVAR_FCI(alpha=0.05, max_lag=2, verbose=True)
    model.fit(X, var_names=names)

    G = model.graph_
    print("\n=== Learned PAG marks ===")
    print(G.marks)
    print("\n=== Learned adjacency ===")
    print(G.adj.astype(int))

    # Test MAP→MAG conversion
    amat = pag_to_mag(model, G)
    print("\n=== Converted MAG adjacency ===")
    print(amat)

    return model, amat

model, amat = test_svar_fci_once()
```

```
In [ ]:
```