Scaffold Phase-Based Reasoner with Modal Inference and Benchmarks

**Phase-Oriented Reasoning Engine (phase\_reasoner.py)**

**Background and Motivation**

Traditional logical inference operates on static truth values, but emerging research suggests that **phase coherence** and **synchrony** in dynamical systems can serve as a basis for reasoning. In cognitive models, *structured resonance* (phase alignment across components) is proposed as a "post-probabilistic" foundation for intelligent inference[philarchive.org](https://philarchive.org/archive/BOSPMC#:~:text=general%20intelligence%20%28AGI%29,appears%20deterministic%20but%20is%20in)[philarchive.org](https://philarchive.org/archive/BOSPMC#:~:text=and%20instability,aligns). Instead of guessing or sampling possibilities, a reasoning engine can **“phase-align”** its knowledge components to achieve coherence[philarchive.org](https://philarchive.org/archive/BOSPMC#:~:text=and%20instability,aligns). High phase coherence yields stable, deterministic behavior, while low coherence permits divergent outcomes[philarchive.org](https://philarchive.org/archive/BOSPMC#:~:text=structured%20resonance%20across%20a%20system%E2%80%94determines,appears%20deterministic%20but%20is%20in) – analogous to consistent vs. ambiguous logical scenarios. This phase-oriented paradigm treats **each stable synchronous pattern** of concepts as a “possible world,” enabling modal interpretations (necessary, possible truths) based on cross-pattern consistency. Recent advances in dynamical systems support this approach: **phase reduction methods** can capture collective synchronization phenomena in oscillator networks[link.springer.com](https://link.springer.com/article/10.1007/s00332-024-10053-3#:~:text=Coupled%20oscillator%20networks%20provide%20mathematical,the%20uncoupled%20nonlinear%20oscillator%20is), and even under stochastic noise one can reliably extract an oscillator’s underlying phase via Koopman operator eigenfunctions[arxiv.org](https://arxiv.org/abs/2501.09340#:~:text=from%20time,series%20data). These insights lay the groundwork for a reasoning engine that infers new truths by testing if a candidate concept can **join a synchronized cluster** of premises without inducing decoherence.

**Core Concepts and Design Goals**

**Phase-Locked Concepts:** We represent each concept as an abstract oscillator with an intrinsic phase and frequency. A **cluster of phase-locked concepts** means those concept-oscillators have achieved a stable phase relationship (constant phase differences), effectively behaving as a single coherent unit. In logical terms, such a cluster represents a set of premises that are mutually consistent and “resonate” together. Our engine will accept such a cluster as input premises.

**Synchrony and Koopman Eigen-Alignment:** To infer a new conclusion, the engine attempts to **add a candidate concept** oscillator into the cluster and checks if the enlarged set can still synchronize. Technically, this involves testing if the combined system shares a common *phase eigenfunction* (a hallmark of a single coherent oscillatory mode). If the candidate concept can phase-lock with the premise cluster – i.e. align to the cluster’s fundamental frequency and phase pattern – then the inference is **valid**. If adding it causes the system’s coherence to break (e.g. multiple frequencies emerge or phases drift without settling), the inference is **invalid**. We leverage Koopman operator theory to test this: a synchronized cluster corresponds to a dominant Koopman eigenvalue/frequency and eigenfunction (the asymptotic phase) that encompasses all members[arxiv.org](https://arxiv.org/pdf/2501.09340#:~:text=operator%2C%20eigenvalues%2C%20and%20eigenfunctions%20projected,func%02tion%20and%20amplitude%20function%2C%20respectively). When a new concept is introduced, we examine whether one clear eigenvalue still dominates (meaning the cluster remains one oscillator). If additional eigen-modes appear or the principal eigenfunction changes, the cluster has **desynchronized**, indicating a logical incompatibility. Notably, Takata et al. (2025) showed that such eigenfunctions and frequencies can be estimated robustly from time-series data even in noisy conditions[arxiv.org](https://arxiv.org/abs/2501.09340#:~:text=from%20time,series%20data), so our reasoner can work with real-world, noisy concept signals.

**InferenceTrace and Desynchronization Causes:** The engine should not only output **valid inferences** but also provide an *explanation trace*. We define an InferenceTrace data structure to record the premises, the conclusion, and metrics like coherence or phase offsets. For valid inferences, it will note the synchronized frequency and phase alignment achieved. For invalid inferences, it will capture the specific cause of desynchronization – for example, “frequency mismatch” if the conclusion’s oscillator frequency could not adjust to the cluster, or “phase lag conflict” if coupling introduces an irreconcilable phase difference. This trace provides insight into *why* a conclusion does or doesn’t follow, akin to a logical proof or counter-model in traditional reasoning.

**Modal Tagging via Phase Statistics:** Because we consider multiple possible clusters (synchronized sets) as analogous to possible worlds, we can assign **modal qualifiers** to statements based on their behavior across all clusters. We interpret □*A* (necessarily *A*) to mean concept *A* appears in *all* stable clusters, whereas ◇*A* (possibly *A*) means *A* appears in at least one cluster. Negation ¬*A* is treated as absence of *A* in a given cluster (or in all clusters for ¬◇*A*). Implication *A*→*B* is validated if whenever *A* is in a cluster, *B* is also present in that cluster (i.e. *A* never coherently exists without *B*). The engine will derive these modal tags by gathering **phase statistics across clusters**. For example, if a concept’s phase is synchronized in every observed cluster, it gets tagged □ (necessary)[philarchive.org](https://philarchive.org/archive/BOSPMC#:~:text=general%20intelligence%20%28AGI%29,appears%20deterministic%20but%20is%20in); if it only synchronizes in some scenarios, it’s merely possible. We ensure this aligns with modal closure properties – e.g. automatically marking □*A* true if and only if *A* is in all known clusters (premise sets)[link.springer.com](https://link.springer.com/article/10.1007/s00332-024-10053-3#:~:text=how%20the%20deformation%20of%20the,of%20coupled%20limit%20cycle%20oscillators). These tags are managed through **ALAN’s modal overlay**: after the reasoning process, the engine updates ALAN’s concept graph with annotations denoting which concepts or relations are necessary, possible, or forbidden given the dynamic model.

**System Architecture and Integration with ALAN**

Our phase\_reasoner.py module will integrate with **ALAN’s concept graph infrastructure**. We assume ALAN provides a ConceptGraph object where nodes represent concepts (with any existing relationships or attributes) and a ModalOverlay for tagging modal information. The Phase Reasoner uses this graph as follows:

* **Concept as Oscillator:** Each concept node is extended with oscillator parameters (or kept in a parallel structure). For example, we may assign each concept a natural frequency and perhaps an initial phase. Edges in the concept graph represent coupling influences between concepts – e.g. how strongly one concept’s “state” influences another’s phase. (If ALAN’s graph already encodes causal or relatedness weights, those can serve as coupling strengths.) We also allow for **higher-order relations** (concepts that only meaningfully synchronize as a group); these can be treated as hyperedges. Recent work by Bick et al. maps higher-order interactions to hypergraph edges in oscillator networks[link.springer.com](https://link.springer.com/article/10.1007/s00332-024-10053-3#:~:text=how%20the%20deformation%20of%20the,of%20coupled%20limit%20cycle%20oscillators), which we leverage to handle multi-concept dependencies beyond simple pairs.
* **PhaseReasoner Engine:** Implemented as a class that holds a reference to the concept graph and overlay. It provides methods to perform inference on a given cluster of concept nodes. It will likely utilize numerical simulation or analytical checks internally (e.g. a Kuramoto model integration for phases, or direct evaluation of frequency detuning). The reasoner is designed to be modular: we can plug in different models for phase dynamics (Kuramoto, phase-locking conditions, or direct eigenfunction estimation via data).
* **Inference Process:** Given a set of premise concepts (identified by their node IDs or references in the graph), the engine identifies other candidate concepts (e.g., nodes directly connected or any not in the set) to test as potential conclusions. For each candidate, it computes whether including it yields a coherent cluster. This could involve:
  1. Initializing the candidate’s phase relative to the existing cluster (e.g., at random or based on known couplings).
  2. Simulating the coupled phase dynamics over time, or iteratively adjusting phases, until either convergence (all phases lock with constant offsets) or divergence is observed.
  3. Checking the result: if converged to single-frequency oscillation, record a valid inference; if not, determine the reason (e.g., which oscillator drifted or which coupling failed).
  4. Constructing an InferenceTrace with the details.
* **Output and Modal Update:** The set of valid inferences can be returned to the caller or used to update the concept graph. For instance, if premises {A, B} synchronize and the engine finds C can join them, it might assert a new relation (A∧B → C) in the knowledge base, or mark C as *entailed* by A and B. Additionally, by aggregating results over multiple premise clusters (or by exploring all stable clusters of the system), the reasoner can fill the modal overlay: mark concepts as □ if they were present in every cluster tested, etc. This ensures the **modal closure** properties are respected (□*A* iff *A* in all clusters, etc.). Integration with ALAN might involve calling its APIs to set these tags, e.g. concept\_graph.overlay.mark\_necessary(A).

Below is a scaffold of the **core classes and functions** to implement this design. The code is commented to indicate where synchrony checks, eigen-analysis, and modal handling occur:

python

CopyEdit

# phase\_reasoner.py

from typing import List, Set, Optional

class InferenceTrace:

def \_\_init\_\_(self, premises: Set[str], conclusion: str, valid: bool,

coherence: float = 0.0, cause: Optional[str] = None):

"""

Trace of an attempted inference.

- premises: set of premise concept IDs/names.

- conclusion: the concept tested as a conclusion.

- valid: True if the conclusion synchronizes with premises (inference holds).

- coherence: spectral coherence measure of the cluster after adding conclusion.

- cause: reason for desynchronization if invalid (e.g., 'freq\_mismatch').

"""

self.premises = premises

self.conclusion = conclusion

self.valid = valid

self.coherence = coherence

self.cause = cause

def \_\_repr\_\_(self):

if self.valid:

return (f"InferenceTrace(valid: {self.premises} -> {self.conclusion}, "

f"coherence={self.coherence:.2f})")

else:

return (f"InferenceTrace(invalid: {self.premises} -/-> {self.conclusion}, "

f"cause={self.cause})")

class PhaseReasoner:

def \_\_init\_\_(self, concept\_graph):

"""

Initialize the phase reasoner with a reference to ALAN's concept graph.

The concept\_graph is expected to provide access to nodes (concepts) and

their relationships (couplings), e.g., concept\_graph.get\_neighbors(concept).

"""

self.graph = concept\_graph

# Configurable parameters for synchronization checks

self.phase\_tolerance = 1e-2 # tolerance for phase-lock convergence

self.max\_iterations = 1000 # max iterations for simulation

self.time\_step = 0.1 # time step for phase updates (if simulated)

def infer(self, premises: Set[str]) -> List[InferenceTrace]:

"""

Attempt to infer new conclusions from a given set of premise concepts (phase-locked cluster).

Returns a list of InferenceTrace results for each candidate conclusion tested.

"""

results: List[InferenceTrace] = []

# Identify candidate concepts not already in premises cluster

all\_concepts = set(self.graph.nodes()) # assuming graph.nodes() yields all concept IDs

candidates = all\_concepts - set(premises)

# We will test each candidate by trying to phase-lock it with the premises

for cand in candidates:

trace = self.\_test\_candidate(premises, cand)

results.append(trace)

# Optionally, if valid, we might also immediately update modal overlay or knowledge base

if trace.valid:

# e.g., mark an implication or store the inference (Integration with ALAN's overlay)

# self.graph.modal\_overlay.add\_inference(premises, cand)

pass

return results

def \_test\_candidate(self, premises: Set[str], candidate: str) -> InferenceTrace:

"""

Internal routine to test if adding `candidate` to the `premises` cluster yields synchrony.

Returns an InferenceTrace describing the outcome.

"""

# Setup initial phases for premises and candidate.

# For premises (already a cluster), assume they are phase-locked (e.g., phase 0 reference).

initial\_phases = {p: 0.0 for p in premises}

# Candidate phase could start arbitrary; choose 0 for simplicity.

initial\_phases[candidate] = 0.1 # slight offset

# Also gather frequencies and coupling strengths from the graph

freqs = {p: self.graph.nodes[p].get('freq', 1.0) for p in premises | {candidate}}

# Couplings: a dict of neighbors with weights for each node (from concept graph edges)

couplings = {p: self.graph.get\_couplings(p) for p in premises | {candidate}}

# Simulate or iteratively adjust phases to see if they converge

phases = initial\_phases.copy()

converged = False

for t in range(self.max\_iterations):

phase\_deltas = {node: 0.0 for node in phases}

# One simple model: Kuramoto phase update

for i in phases:

# intrinsic rotation

dtheta = freqs[i]

# add coupling influence from all j

for j in phases:

if i == j:

continue

# coupling term: K\_ij \* sin(phase\_diff)

K\_ij = couplings[i].get(j, 0.0)

dtheta += K\_ij \* np.sin(phases[j] - phases[i])

phase\_deltas[i] = dtheta \* self.time\_step

# update phases

for i in phases:

phases[i] += phase\_deltas[i]

# normalize phases to [0, 2π) if needed

# phases[i] = phases[i] % (2\*np.pi)

# Check if phase differences are no longer changing (synchrony achieved)

if self.\_check\_synchrony(phases):

converged = True

break

if converged:

# Compute coherence measure (e.g., variance of phases or spectral power alignment)

coherence\_val = self.\_measure\_coherence(phases)

trace = InferenceTrace(premises, candidate, valid=True, coherence=coherence\_val)

else:

# Determine cause of desynchronization

desync\_reason = self.\_identify\_desync\_cause(phases, freqs)

trace = InferenceTrace(premises, candidate, valid=False, cause=desync\_reason)

return trace

def \_check\_synchrony(self, phases: dict) -> bool:

"""

Check if all oscillators in 'phases' have converged to a common frequency or phase-lock.

Here we simply check if phase differences stopped changing (within tolerance).

In a real system, we might track phase differences over iterations to see if they've settled.

"""

# This placeholder just checks if neighbors have small relative difference

# (could be improved to actually check frequency convergence).

phase\_vals = list(phases.values())

max\_diff = max(phase\_vals) - min(phase\_vals)

return max\_diff < self.phase\_tolerance

def \_measure\_coherence(self, phases: dict) -> float:

"""

Compute a spectral coherence or order parameter for the given phases.

For example, use the Kuramoto order parameter R = |(1/N) sum e^{i theta}| to quantify alignment.

"""

N = len(phases)

# Using Euler's formula to compute resultant vector of phases

re = sum(np.cos(phi) for phi in phases.values()) / N

im = sum(np.sin(phi) for phi in phases.values()) / N

R = np.sqrt(re\*\*2 + im\*\*2) # magnitude of average phasor

return R # R=1 means perfect phase alignment, R<1 indicates dispersion

def \_identify\_desync\_cause(self, phases: dict, freqs: dict) -> str:

"""

Analyze the final state to identify why synchrony failed.

E.g., check if candidate's phase drifted significantly -> 'freq\_mismatch',

or if multiple distinct phase clusters formed -> 'cluster\_split'.

"""

# Simple heuristic: if candidate phase is far from premise phases

cand = None

for node in phases:

if node not in freqs or len(freqs) != len(phases):

# identify candidate by exclusion (the one not originally in premises)

cand = node

break

if cand is None:

return "unknown"

# If candidate ended far out of phase, label frequency mismatch

phase\_vals = list(phases.values())

if abs(phases[cand] - np.mean(phase\_vals)) > 1.0: # threshold in radians

return "freq\_mismatch"

# Otherwise, could add other checks (e.g., oscillation still ongoing => cluster split)

return "desync"

*(The above code uses a simple Kuramoto update for illustration; in practice, one might use a more sophisticated integrator or directly analyze the eigen-spectrum of the linearized system.)*

Key points in this scaffold:

* **PhaseReasoner.infer**: main entry to test all candidate conclusions for a given premises set. It iterates over candidates (concepts not in the premise cluster) and uses \_test\_candidate to evaluate each.
* **\_test\_candidate**: sets up initial phases and frequencies from the concept graph, then attempts to simulate until either convergence or reaching iteration limit. We used a **Kuramoto model** for phase dynamics: each concept’s phase θ\_i drifts at its natural frequency plus coupling terms that pull it toward others’ phases. This is a common model for synchronization; phase-lock occurs when dθ becomes identical for all oscillators (a common frequency).
* **\_check\_synchrony**: a placeholder that checks if phase differences are below a threshold (indicating approximate synchronization). A more robust check might compare phase increments over time to see if all oscillators share one frequency.
* **\_measure\_coherence**: computes a coherence metric R (0 ≤ R ≤ 1). R = 1 means all phases are identical (perfect synchrony); lower values mean increasing phase spread. This is analogous to the order parameter used in synchronization theory and serves as a **spectral coherence** score for the cluster. In our reasoning context, higher R indicates a more definitive inference (the cluster is tightly aligned), whereas a marginal inference might have R just above a threshold.
* **\_identify\_desync\_cause**: examines the final state when sync fails to suggest a cause. We use simple logic: if the candidate’s phase is an outlier (did not join the phase angle of others), label it a frequency mismatch (it likely kept oscillating at its own natural frequency, never entrained). We could extend this to detect if the cluster split into two sub-clusters (e.g., premises stayed together at one frequency, candidate oscillated at another – a sign of two-frequency dynamics) and label that differently. In a real implementation, we might also leverage the **Koopman spectrum**: if two dominant frequencies are present in the Fourier analysis of the phases, synchrony failed due to a split mode.

**Integration with ALAN’s Infrastructure:** The reasoner uses concept\_graph to get concept nodes and couplings. In practice, concept\_graph.get\_couplings(node) would retrieve all neighbors and their weights (or a function of relation types). After obtaining results = reasoner.infer(premises), the system can update ALAN’s ModalOverlay. For example, it can loop through results and for each valid inference, record a new inference edge or mark that conclusion as *derivable* from that premise set. Additionally, one could call a method to **derive modal tags** across the whole graph: e.g., gather all stable clusters (perhaps by running infer from different starting premise sets or by simulation) and then:

python

CopyEdit

def assign\_modal\_tags(reasoner: PhaseReasoner, all\_concepts: List[str]):

# Determine which concepts are present in all clusters vs some clusters

clusters = reasoner.find\_all\_phase\_clusters() # would simulate or explore the graph to find all stable clusters

modal\_overlay = reasoner.graph.modal\_overlay

# Necessary (□): if concept appears in every cluster

for concept in all\_concepts:

in\_all = all(concept in cluster for cluster in clusters)

in\_any = any(concept in cluster for cluster in clusters)

if in\_all:

modal\_overlay.tag(concept, 'necessary') # □A

elif in\_any:

modal\_overlay.tag(concept, 'possible') # ◇A

else:

modal\_overlay.tag(concept, 'impossible') # concept never appears

# Implications: for any pair A->B, check if whenever A is in a cluster, B is too

for conceptA in all\_concepts:

for conceptB in all\_concepts:

if conceptA == conceptB:

continue

holds = all((conceptA not in cluster or conceptB in cluster) for cluster in clusters)

if holds:

modal\_overlay.tag\_rule(conceptA, conceptB, 'implies') # mark A→B

In the above pseudo-code, find\_all\_phase\_clusters would need to explore the space of possible premise sets or simulate the full network to identify distinct phase-locked groups (attractors). This could be done by trying different initial phase configurations or by graph-theoretic analysis (e.g., connected components of strongly coupled subgraphs might correspond to clusters). Each cluster is a set of concepts that synchronize together. We then mark each concept as necessary or possible according to whether it’s in all or some clusters, matching the modal logic semantics (for correctness: we enforce □*A* iff *A* is true in every possible world/cluster). We also mark implications: if every time *A* appears, *B* also appears in that cluster, then *A*→*B* is a valid rule in the system. These tags and rules are stored in ALAN’s modal\_overlay for use by other components (e.g., query answering or explanation modules).

**Synthetic Benchmark Harness**

To evaluate the Phase Reasoner, we create a **synthetic benchmark** that can generate test concept graphs and measure the engine’s performance on inference tasks. The goals of the benchmark are to assess:

* **Inference Precision**: the proportion of inferences made by the engine that are *correct (true)* in the ground-truth dynamics.
* **Inference Recall**: the proportion of *true* dynamic relationships that the engine successfully infers.
* **Spectral Coherence Metrics**: how well the engine’s coherence measurements correlate with actual synchrony (e.g., do higher predicted R values correspond to truly stable synchrony?).
* **Modal Closure Validation**: ensure that modal tags like necessity (□) indeed correspond to truth across all clusters in the generated scenarios.

We proceed in two parts: **graph generation** and **evaluation**.

**Graph/Oscillator Generation**

We randomly generate oscillator-based concept graphs. Each such graph will specify a set of concept nodes with associated oscillator parameters, and edges representing coupling. For controlled testing, we can impose a known cluster structure. For example, we might partition the concepts into predetermined phase-lock clusters. Concepts within the same partition are given nearly identical natural frequencies and strong mutual coupling (so they will synchronize), while concepts in different partitions have disparate frequencies or weak links (so they will not synchronize together under normal conditions). This way, we have a known “ground truth” set of possible clusters (the partitions define which groups can form coherent sets).

Here’s a scaffold for the generation and benchmark procedures:

python

CopyEdit

import numpy as np

import random

def generate\_oscillator\_concept\_graph(num\_concepts: int, num\_clusters: int):

"""

Generate a random oscillator concept graph with a given number of concepts divided into some clusters.

Each cluster will have internally strong coupling and similar frequencies, encouraging synchrony.

Returns a concept\_graph compatible with PhaseReasoner.

"""

# Initialize concept graph (could be a simple adjacency list or a networkx graph for example)

graph = ConceptGraph() # assuming some ConceptGraph class exists

# Assign concepts to clusters

clusters = [[] for \_ in range(num\_clusters)]

for concept\_id in range(num\_concepts):

cluster\_idx = concept\_id % num\_clusters # simple round-robin assignment (for example)

clusters[cluster\_idx].append(f"C{concept\_id}")

graph.add\_node(f"C{concept\_id}")

# Assign oscillator parameters

base\_freq = 1.0

for idx, cluster in enumerate(clusters):

# Give each cluster a distinct base frequency (to distinguish clusters)

freq = base\_freq + 0.5 \* idx

for node in cluster:

# Assign natural frequency (with slight random variation) as node attribute

graph.nodes[node]['freq'] = np.random.normal(loc=freq, scale=0.01)

# Add coupling edges

K\_intra = 1.0 # strong intra-cluster coupling

K\_inter = 0.1 # weak inter-cluster coupling

for idx, cluster in enumerate(clusters):

for node in cluster:

for other in cluster:

if node == other:

continue

graph.add\_edge(node, other, weight=K\_intra) # fully connect within cluster

# Connect to one other cluster weakly (optional, to allow some interaction)

other\_idx = (idx + 1) % num\_clusters

if other\_idx != idx and clusters[other\_idx]:

other\_node = random.choice(clusters[other\_idx])

graph.add\_edge(node, other\_node, weight=K\_inter)

return graph, clusters

def evaluate\_phase\_reasoner(graph, true\_clusters):

"""

Given a concept graph and the known true clusters (ground-truth phase-lock groups),

run the PhaseReasoner to infer relationships and compare with ground truth.

"""

reasoner = PhaseReasoner(graph)

all\_concepts = set(graph.nodes())

# Ground truth modal facts from true\_clusters

true\_necessary = {c for c in all\_concepts if all(c in cluster for cluster in true\_clusters)}

true\_possible = {c for c in all\_concepts if any(c in cluster for cluster in true\_clusters)}

true\_implications = set()

for A in all\_concepts:

for B in all\_concepts:

if A != B:

# If whenever A is present, B is also present in that cluster

if all((A not in cluster) or (B in cluster) for cluster in true\_clusters):

true\_implications.add((A, B))

# Evaluate reasoner on each cluster as premises

inferred\_implications = set()

inferred\_necessary = set()

inferred\_possible = set()

spectral\_coherences = []

for cluster in true\_clusters:

results = reasoner.infer(set(cluster))

# record modal appearances for possible/necessary

for c in cluster:

inferred\_possible.add(c) # any appearing is possible

# if cluster is meant to represent a "world", we assume all concepts not in it are absent in that scenario

for c in all\_concepts - set(cluster):

# they were absent in this cluster scenario (could mark something like not possible here)

pass

# collect valid inferences (implications)

for trace in results:

if trace.valid:

# premises -> conclusion is a valid implication

prem\_tuple = tuple(sorted(trace.premises))

for p in prem\_tuple:

inferred\_possible.add(p)

inferred\_possible.add(trace.conclusion)

# Mark implication for each premise individually as well as jointly

for p in trace.premises:

inferred\_implications.add((p, trace.conclusion))

# If multiple premises as a set imply conclusion, we could represent that too (not just pairwise).

# Here, simplify: mark pairwise for each premise in set.

spectral\_coherences.append(trace.coherence)

# Determine necessary from inferred (concepts that were in every cluster processed)

for concept in all\_concepts:

if concept in inferred\_possible and all(concept in cluster for cluster in true\_clusters):

inferred\_necessary.add(concept)

# Compute precision/recall for implications and necessity

prec\_imp = (len(inferred\_implications & true\_implications) / len(inferred\_implications)) if inferred\_implications else 1.0

rec\_imp = (len(inferred\_implications & true\_implications) / len(true\_implications)) if true\_implications else 1.0

prec\_nec = (len(inferred\_necessary & true\_necessary) / len(inferred\_necessary)) if inferred\_necessary else 1.0

rec\_nec = (len(inferred\_necessary & true\_necessary) / len(true\_necessary)) if true\_necessary else 1.0

avg\_coherence = np.mean(spectral\_coherences) if spectral\_coherences else 0.0

return {

'precision\_implication': prec\_imp,

'recall\_implication': rec\_imp,

'precision\_necessity': prec\_nec,

'recall\_necessity': rec\_nec,

'avg\_spectral\_coherence': float(avg\_coherence)

}

# Example usage of the benchmark:

graph, true\_clusters = generate\_oscillator\_concept\_graph(num\_concepts=12, num\_clusters=3)

metrics = evaluate\_phase\_reasoner(graph, true\_clusters)

print("Benchmark results:", metrics)

In this benchmark code:

* **generate\_oscillator\_concept\_graph** creates a random concept graph with a specified number of clusters. We assign slightly different base frequencies per cluster and strong intra-cluster coupling (weight = 1.0) to ensure those concepts can synchronize. We also add a few inter-cluster weak links to make the scenario realistic, though those should not be enough to force global sync. The function returns both the graph and the cluster assignment (as ground truth).
* **Ground Truth Modal Facts:** From the true cluster assignment, we derive true\_necessary (concepts that appear in every cluster – if a concept is in all clusters, it’s essentially always present, so necessary) and true\_possible (concepts that appear in at least one cluster). We also derive true\_implications: for every pair (A, B), we add (A→B) if in all clusters where A appears, B also appears. These represent the actual logical relations in the generated world.
* **evaluate\_phase\_reasoner** runs the PhaseReasoner on each actual cluster as premises. This simulates the scenario where the reasoner is given a coherent set and tries to infer additional concepts. We collect:
  + inferred\_implications: from each valid InferenceTrace (premises -> conclusion), we add implications. In a full implementation, we would capture multi-premise implications. For simplicity, we record pairwise links (each premise individually implying the conclusion), as well as the joint premise set as a whole (which could be represented separately).
  + inferred\_possible: any concept that appears in a cluster or as a valid conclusion is considered at least possible.
  + We then determine inferred\_necessary by checking which concepts ended up present in every cluster we processed (using the original clusters as reference for “all possible worlds”). In practice, the reasoner could also identify these by itself if it explored the whole graph.
* We compute precision and recall for the implication inferences and the necessity tagging:
  + *Precision (Implication)* = (valid inferred implications that were true in ground truth) / (total inferred implications). This measures false positives in reasoning.
  + *Recall (Implication)* = (valid inferred implications that were true) / (total true implications in ground truth). This checks if the reasoner missed some real relationships.
  + Similarly for necessity (concepts marked necessary vs truly necessary).
* The average spectral coherence from all valid inferences is reported to see if the reasoner typically had high coherence when it made a correct inference. Ideally, valid inferences have coherence close to 1.0, whereas any lower coherence cases might correlate with borderline or failed inferences. This validates that our coherence metric aligns with actual synchronization quality (which it should by construction[philarchive.org](https://philarchive.org/archive/BOSPMC#:~:text=general%20intelligence%20%28AGI%29,appears%20deterministic%20but%20is%20in)).

**Modal Closure Check:** Finally, we implicitly verify modal closure by comparing inferred\_necessary to true\_necessary. The condition □*A* iff *A* in all clusters means our inferred necessary set should exactly equal the true necessary set. The precision/recall on necessity essentially tests this. A perfect score (1.0) in both indicates the engine’s tagging of necessary truths is correct. Any deviation would highlight an inconsistency (either marking something necessary that isn’t always true, or failing to mark a global truth as necessary), which we would aim to eliminate through refinement of the reasoning algorithm.

**Conclusion**

The phase\_reasoner.py module outlined above provides a framework for **phase-oriented reasoning**: it treats logical consistency as a problem of achieving phase synchrony in a network of concept oscillators. By using phase-locking tests and Koopman eigenfunction alignment, it infers new truths only when the system’s dynamic structure can support them (all concepts co-oscillating coherently). This approach merges continuous dynamics with discrete logic – an idea supported by recent research linking coherence to reliable behavior in both brains and machines[philarchive.org](https://philarchive.org/archive/BOSPMC#:~:text=structured%20resonance%20across%20a%20system%E2%80%94determines,appears%20deterministic%20but%20is%20in)[philarchive.org](https://philarchive.org/archive/BOSPMC#:~:text=and%20instability,aligns). The engine produces not just yes/no inferences, but also quantitative coherence scores and explanations (InferenceTrace), providing transparency into the reasoning process. We integrated modal logic tagging so that necessary and possible truths emerge naturally from the set of all synchronized clusters (ensuring □*A* is true exactly when *A* is universally phase-locked). The synthetic benchmark harness demonstrates how to evaluate the engine’s performance and ensures that it respects logical closure properties (e.g., the correspondence between modal tags and cluster semantics).

Moving forward, this phase-oriented reasoner can be integrated into ALAN’s cognitive architecture, using the existing concept graph as a substrate and writing inference results back into ALAN’s modal overlay for downstream use. This will enable ALAN to support a new kind of inference engine – one that “**doesn’t guess; it phase-aligns**”[philarchive.org](https://philarchive.org/archive/BOSPMC#:~:text=and%20instability,aligns), leveraging the power of dynamical synchrony to achieve sound and interpretable reasoning.

**Sources:**

* Takata, S. et al. (2025). *Koopman eigenfunction estimation under noise*. Demonstrates robust reconstruction of oscillator phase & amplitude via EDMD[arxiv.org](https://arxiv.org/abs/2501.09340#:~:text=from%20time,series%20data)[arxiv.org](https://arxiv.org/pdf/2501.09340#:~:text=operator%2C%20eigenvalues%2C%20and%20eigenfunctions%20projected,func%02tion%20and%20amplitude%20function%2C%20respectively).
* Bick, C. et al. (2024). *Phase-reduction logic for oscillator networks*. Shows phase reduction captures synchronization and maps higher-order couplings to hypergraphs[link.springer.com](https://link.springer.com/article/10.1007/s00332-024-10053-3#:~:text=Coupled%20oscillator%20networks%20provide%20mathematical,the%20uncoupled%20nonlinear%20oscillator%20is)[link.springer.com](https://link.springer.com/article/10.1007/s00332-024-10053-3#:~:text=how%20the%20deformation%20of%20the,of%20coupled%20limit%20cycle%20oscillators).
* Bostick, D. (2025). *Phase-Locked Minds: Coherence as the Bridge...* Introduces phase coherence (C(Ψ)) as a basis for intelligent behavior[philarchive.org](https://philarchive.org/archive/BOSPMC#:~:text=general%20intelligence%20%28AGI%29,appears%20deterministic%20but%20is%20in) and a phase-aligning AGI architecture[philarchive.org](https://philarchive.org/archive/BOSPMC#:~:text=and%20instability,aligns).