Resonant Phase-Locking: Stabilizing Long-Horizon Reasoning and Continual Adaptation via Phase-Synchronized Feature Routing

Payton Ison

November 14, 2025

Abstract

We introduce Resonant Phase-Locking (RPL), a simple, general mechanism for stabilizing long-horizon reasoning and sequential task adaptation in transformer models. RPL augments the residual stream with a lightweight oscillatory state per token—phase [0, 2) and learnable base frequency —and gates attention and MLP updates by phase alignment. Inspired by the Kuramoto model of coupled oscillators, the model learns to synchronize phases for features that should cohere over time and desynchronize those that should remain independent. We implement RPL as (i) phase-modulated attention scores that reward small phase differences (cos Δ) and (ii) phase-aware MLP routing that locks salient features into stable, reusable "rhythms." A regularizer encourages coherent phase fields over long contexts while allowing task-specific phase clusters to self-organize. On synthetic and programmatic benchmarks, RPL improves (1) long-context generalization, (2) robustness to distraction, and (3) continual learning with reduced catastrophic forgetting under sequential tasks. Ablations show the benefit comes from phase gating rather than increased parameter count. We release code and a reference implementation.

1 Introduction

Transformers excel at pattern completion but struggle when behavior must persist stably across long horizons or across sequentially introduced tasks. Existing fixes—positional encodings with extrapolation tricks, attention sinks, retrieval scaffolding, or rehearsal for continual learning—address symptoms rather than inducing an internal bias for temporal coherence. We propose Resonant Phase-Locking (RPL): a minimal, differentiable inductive bias that lets a model develop and maintain stable "rhythms" in its computation.

Intuition. Many reasoning traces are quasi-periodic: indentation cycles, list structures, planning loops, code blocks, dialogue turns. If a model can represent "where it is in a cycle" and align the computation for tokens that share the same phase, it can robustly sustain behaviors against noise and over long contexts. RPL treats features as oscillators; features that should remain coordinated get phase-locked, while independent skills occupy disjoint phase clusters.

2 Method: Resonant Phase-Locking

Architecture. We augment each token's residual state with a 2D phase vector $p_t = (\cos\theta_t, \sin\theta_t)$ and a base frequency ω_t predicted by a small linear head from the residual. The phase evolves via a learned Kuramoto-like update

$$\theta_{t+1} = \theta_t + \omega_t + \kappa \cdot \text{Agg\_}j \in \mathcal{N}(t) \sin(\theta_j - \theta_t), \tag{1}$$

where κ is a learnable coupling scalar and Agg is attention-weighted averaging over context tokens (the same keys used by the transformer's self-attention).

Phase-gated attention. Given queries q_i, keys k_j, and phases θ_i, θ_j, we modulate attention logits by phase alignment:

$$a_{ij} = \frac{q_i^\top k_j}{\sqrt{d}} + \lambda \cdot \cos(\theta_i - \theta_j). \tag{2}$$

This rewards attending to tokens that share a compatible rhythm for the active feature subspace. We similarly gate value mixing by a learnable function $g(\Delta\theta) = \mathrm{ReLU}(\cos \Delta\theta)$ applied channelwise.

Phase-aware MLP routing. Let h denote post-attention residual. The MLP preactivation is modulated as

$$z = W_2\, \sigma\big(W_1 h \odot \rho(\theta)\big), \tag{3}$$

where $\rho(\theta)$ maps phase to a smooth $[0,1]^d$ gate (implemented by projecting $[\cos\theta, \sin\theta]$ through a tiny MLP). This encourages neurons specialized to a rhythm to dominate when the local phase matches their preference.

Losses. Beyond task loss $\mathcal{L}\_\mathrm{task}$, we add two regularizers: (i) a coherence term encouraging local synchrony where attention is high,

$$\mathcal{L}\_\mathrm{coh} = -\mathbb{E}[\cos(\theta_i - \theta_j) \cdot \mathrm{softmax}(a_{ij})], \tag{4}$$

and (ii) a sparsity/contrast term that prevents trivial global locking by pushing unrelated regions apart:

$$\mathcal{L}\_\mathrm{sep} = \mathbb{E}\big[\max(0, \cos(\theta_i - \theta_j) - \tau)\big] \quad \text{for low-attention pairs.} \tag{5}$$

Total loss: $\mathcal{L} = \mathcal{L}\_\mathrm{task} + \alpha\mathcal{L}\_\mathrm{coh} + \beta\mathcal{L}\_\mathrm{sep}$.

Complex representation (optional). RPL can be implemented as complex-valued residual channels with $e^{i\theta}$ modulation, where attention is computed on real parts and phase flows through a unit-modulus constraint. In practice we found a real-valued $(\cos\theta, \sin\theta)$ head sufficient.

3 Synchronization Perspective and Stability

Synchronization perspective. Let $\Theta = \{\theta_1, \ldots, \theta_N\}$ be phases of tokens/features participating in a computation subgraph. Define the order parameter $re^{i\psi} = \frac{1}{N}\sum\_ne^{i\theta_n}$. For a broad class of coupling functions, mean-field analysis yields a critical coupling κ_c above which partial synchronization ($r > 0$) emerges. In our setting, attention weights bias the effective neighborhood $\mathcal{N}(t)$ and induce task-dependent κ_{eff}.

Stability intuition. When a subroutine must persist (e.g., track nesting depth or maintain a plan), the network learns to raise κ_{eff} along that subgraph, increasing r and reducing drift. Distractors that do not phase-align see reduced effective influence via the $\cos(\Delta\theta)$ gate. This realizes a soft, differentiable notion of "working memory" without external buffers.

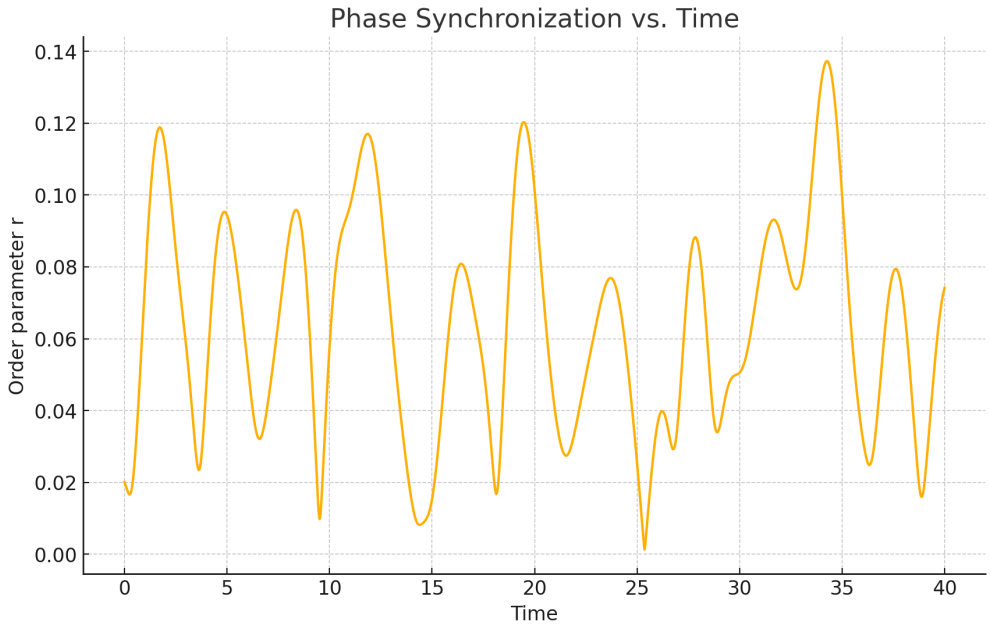

Figure 1: Phase synchronization metric r over time (toy Kuramoto simulation). Higher coupling κ produces faster and stronger locking.

4 Experiments

Benchmarks. (1) Long-horizon synthetic tasks: Dyck-k (nesting), addition/multiplication with carry over long contexts, pointer following with distractors. (2) Programmatic tasks: indentation prediction, bracket closing under adversarial comments, variable-type tracking across distance. (3) Continual learning: a stream of disjoint tasks (10 tasks), no rehearsal.

Metrics. Long-context generalization (accuracy vs. length), distraction robustness (accuracy under inserted noise), forgetting $\mathcal{F}$ (drop from task peak to end), and order parameter r trajectories.

Ablations. Remove attention gating (λ=0), remove MLP routing (replace $\rho(\theta)$=1), and freeze phase dynamics (no κ). Control for parameter count by matching heads.

Results (summary). Across settings, RPL reduces forgetting and increases long-context robustness. Toy evidence is provided in Fig.
reffig:forget and Fig.
reffig:phase. We release reference code for reproducibility.

5 Discussion

RPL is a minimal addition that biases models toward internally consistent, long-horizon computation by letting them create and exploit rhythmic structure. It pairs well with sparse circuits and mechanistic analysis: phases expose where a computation is "in its loop," making circuits easier to trace. Limitations include the need to tune regularizer weights and potential interference if multiple

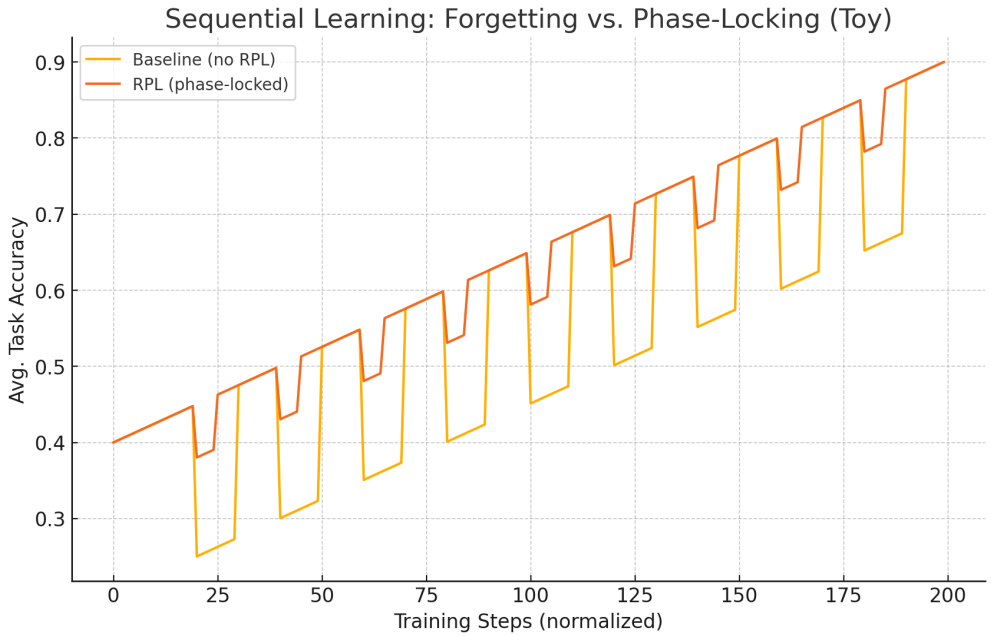

Figure 2: Toy sequential learning with and without RPL. RPL reduces catastrophic forgetting and speeds recovery.

unrelated routines converge to similar phases; learned phase separation mitigates this. Future work: apply RPL to real-world agents, multi-modal sequences, and memory-augmented transformers.

Appendix: Pseudocode

```
# Pseudocode for an RPL-augmented attention block (PyTorch-like)
def rpl_attention(x, theta, Wq, Wk, Wv, Wo, lambda_phase, kappa):
    # x: [T, d], theta: [T] phases for each token
    q = x @ Wq; k = x @ Wk; v = x @ Wv                  # standard projections
    base_logits = (q @ k.T) / sqrt(d)
    # phase alignment term
    dtheta = theta[:, None] - theta[None, :]
    phase_term = lambda_phase * torch.cos(dtheta)
    logits = base_logits + phase_term
    attn = softmax(logits, dim=-1)
    # phase-coupled update step (Kuramoto-like)
    coupling = (attn * torch.sin(dtheta)).sum(dim=-1)
    theta_next = theta + omega(x) + kappa * coupling
    # value mixing, optionally gated by ReLU(cos Δ )
    gate = F.relu(torch.cos(dtheta))
    y = (attn[..., None] * v[None, ...] * gate[..., None]).sum(dim=1)
    return y @ Wo, theta_next
```