Universal Stochastic Predictor
Phase 3: Core Orchestration

Implementation Team

February 19, 2026

Índice

Capítulo 1

Phase 3: Core Orchestration Overview

Phase 3 implements the physical orchestration layer in `stochastic_predictor/core/`. This layer fuses heterogeneous kernel outputs using Wasserstein gradient flow (JKO) and entropic optimal transport (Sinkhorn) with volatility-coupled regularization.

1.1 Scope

Phase 3 covers:

- **Sinkhorn Regularization**: Volatility-coupled entropic regularization for stable optimal transport
- **Wasserstein Fusion**: JKO-weighted fusion of kernel predictions and confidence scores
- **Simplex Sanitization**: Enforced simplex constraints for kernel weights
- **Core API**: Exported fusion and Sinkhorn utilities via `core/__init__.py`

1.2 Design Principles

- **Zero-Heuristics Policy**: All parameters injected via `PredictorConfig`
- **JAX-Native**: Stateless functions compatible with JIT/vmap
- **Determinism**: Bit-exact reproducibility under configured XLA settings
- **Volatility Coupling**: Dynamic regularization tied to EWMA variance

Capítulo 2

Sinkhorn Module (core/sinkhorn.py)

2.1 Volatility-Coupled Regularization

The entropic regularization parameter adapts to local volatility according to the specification:

$$\varepsilon_t = \max\left(\varepsilon_{\min}, \varepsilon_0 \cdot (1 + \alpha \cdot \sigma_t)\right)$$

where $\sigma_t = \sqrt{\text{EMA variance}}$ and α is the coupling coefficient.

```python
def compute_sinkhorn_epsilon(ema_variance, config):
    sigma_t = jnp.sqrt(jnp.maximum(ema_variance, config.numerical_epsilon))
    epsilon_t = config.sinkhorn_epsilon_0 * (1.0 + config.sinkhorn_alpha * sigma_t)
    return jnp.maximum(config.sinkhorn_epsilon_min, epsilon_t)
```

2.2 Entropy-Regularized OT (Scan-Based)

The Sinkhorn iterations are implemented with `jax.lax.scan` to ensure predictable XLA lowering and to support per-iteration volatility coupling. The iteration count is controlled by `config.sinkhorn_max_iter`.

```python
def volatility_coupled_sinkhorn(source_weights, target_weights, cost_matrix, ema_variance
    , config):
    log_a = jnp.log(jnp.maximum(source_weights, config.numerical_epsilon))
    log_b = jnp.log(jnp.maximum(target_weights, config.numerical_epsilon))
    f0 = jnp.zeros_like(source_weights)
    g0 = jnp.zeros_like(target_weights)

    def sinkhorn_step(carry, _):
        f, g = carry
        eps = compute_sinkhorn_epsilon(ema_variance, config)
        f = _smin(cost_matrix - g[None, :], eps) + log_a
        g = _smin(cost_matrix.T - f[None, :], eps) + log_b
        return (f, g), None

    (f_final, g_final), _ = jax.lax.scan(
        sinkhorn_step, (f0, g0), None, length=config.sinkhorn_max_iter
    )

    epsilon_final = compute_sinkhorn_epsilon(ema_variance, config)
    transport = jnp.exp((f_final[:, None] + g_final[None, :] - cost_matrix) /
    epsilon_final)
    safe_transport = jnp.maximum(transport, config.numerical_epsilon)
    entropy_term = jnp.sum(safe_transport * (jnp.log(safe_transport) - 1.0))
    reg_ot_cost = jnp.sum(transport * cost_matrix) + epsilon_final * entropy_term
    row_err = jnp.max(jnp.abs(jnp.sum(transport, axis=1) - source_weights))
    col_err = jnp.max(jnp.abs(jnp.sum(transport, axis=0) - target_weights))
    converged = jnp.maximum(row_err, col_err) <= config.validation_simplex_atol
```

```
    return SinkhornResult(
        transport_matrix=transport,
        reg_ot_cost=reg_ot_cost,
        converged=jnp.asarray(converged),
        epsilon=jnp.asarray(epsilon_final),
    )
```

Capítulo 3

Fusion Module (core/fusion.py)

3.1 JKO-Weighted Fusion

The fusion step normalizes kernel confidences into a simplex and performs a JKO proximal update on weights:

$$\rho_{k+1} = \rho_k + \tau(\hat{\rho} - \rho_k)$$

```
def fuse_kernel_outputs(kernel_outputs, current_weights, ema_variance, config):
    predictions = jnp.array([ko.prediction for ko in kernel_outputs]).reshape(-1)
    confidences = jnp.array([ko.confidence for ko in kernel_outputs]).reshape(-1)
    target_weights = _normalize_confidences(confidences, config)

    cost_matrix = compute_cost_matrix(predictions, config)
    sinkhorn_result = volatility_coupled_sinkhorn(
        source_weights=current_weights,
        target_weights=target_weights,
        cost_matrix=cost_matrix,
        ema_variance=ema_variance,
        config=config,
    )

    updated_weights = _jko_update_weights(current_weights, target_weights, config)
    PredictionResult.validate_simplex(updated_weights, config.validation_simplex_atol)

    fused_prediction = jnp.sum(updated_weights * predictions)
    return FusionResult(
        fused_prediction=fused_prediction,
        updated_weights=updated_weights,
        free_energy=sinkhorn_result.reg_ot_cost,
        sinkhorn_converged=sinkhorn_result.converged,
        sinkhorn_epsilon=sinkhorn_result.epsilon,
        sinkhorn_transport=sinkhorn_result.transport_matrix,
    )
```

3.2 Simplex Sanitization

The simplex constraint is validated using the injected tolerance:

```
PredictionResult.validate_simplex(updated_weights, config.validation_simplex_atol)
```

Capítulo 4

Core Public API

```
from .fusion import FusionResult, fuse_kernel_outputs
from .sinkhorn import SinkhornResult, compute_sinkhorn_epsilon,
    volatility_coupled_sinkhorn
```

4.1 Compliance Checklist

- **Zero-Heuristics**: All parameters injected via config
- **Volatility Coupling**: Implemented per specification
- **Simplex Validation**: Config-driven tolerance enforced
- **JAX-Native**: Pure functions and stateless modules

Capítulo 5

Phase 3 Summary

Phase 3 delivers a concrete orchestration layer for Wasserstein fusion and JKO weight updates. The core layer is now physically present and ready for integration with the prediction pipeline.