

# **Guía de Implementación en Python de Predictores Estocásticos Universales**

Consortio de Desarrollo de Meta-Predicción Adaptativa

18 de febrero de 2026

# Índice

<b>1</b>	<b>Entorno y Stack Tecnológico</b>	<b>2</b>
1.1	Selección de Librerías . . . . .	2
1.2	Gestión de Precisión Numérica Global . . . . .	2
<b>2</b>	<b>Módulo 1: Motor de Identificación (SIA)</b>	<b>3</b>
2.1	Estimación WTMM con Callback Asíncrono . . . . .	3
2.2	Cálculo de Peso de Malliavin (Integral de Skorokhod) . . . . .	6
<b>3</b>	<b>Módulo 2: Núcleos de Predicción</b>	<b>9</b>
3.1	Rama A: Procesos de Lévy y Monte Carlo Vectorizado . . . . .	9
3.2	Rama B: Solvers DGM-PDE con Equinox . . . . .	9
3.2.1	Monitoreo de Entropía DGM (Detección de Mode Collapse) . . . . .	11
<b>4</b>	<b>Módulo 3: Orquestador JKO (Jordan-Kinderlehrer-Otto)</b>	<b>14</b>
4.1	Sinkhorn Estabilizado en Log-Domain con OTT-JAX . . . . .	14
4.2	Integración con Optax para Aprendizaje de Metaparametros . . . . .	16
4.3	Rama C: Esquema IMEX con Solver de Punto Fijo Robusto . . . . .	16
4.4	Rama D: Log-Signatures con Signax . . . . .	17
<b>5</b>	<b>Integración y Pipeline</b>	<b>19</b>
5.1	Clase Maestra PredictionEngine . . . . .	19
<b>6</b>	<b>Meta-Optimización: Walk-Forward y Bayesian Tuning</b>	<b>26</b>
6.1	Validación Rolling Walk-Forward . . . . .	26
6.2	Optimización Bayesiana con Optuna . . . . .	27
<b>7</b>	<b>Sistema de Telemetría y Flags de Estado</b>	<b>29</b>
7.1	Estructura de Telemetría . . . . .	29
7.2	Interpretación de Flags . . . . .	32

# Capítulo 1

## Entorno y Stack Tecnológico

Esta guía traduce las especificaciones algorítmicas del tratado universal a un ecosistema de producción en Python de alto rendimiento.

### 1.1 Selección de Librerías

Para equilibrar expresividad matemática y eficiencia computacional (C++/CUDA backend), se prescribe el siguiente stack:

- **JAX**: Para computación numérica acelerada (XLA), vectorización automática (`vmap`) y diferenciación automática (`grad`, `jacfwd`) requerida por Malliavin y JKO.
- **Equinox / DiffraX**: Frameworks sobre JAX para redes neuronales y solvers de ecuaciones diferenciales estocásticas (SDEs), respectivamente.
- **Signax**: (Nativo en JAX) para el cálculo diferencial y ultra-rápido de Signatures y Log-Signatures en GPU, manteniendo el grafo computacional intacto.
- **PyWavelets**: Para la Transformada Wavelet Continua (en CPU, con callback asíncrono) en el módulo SIA.
- **OTT-JAX (Optimal Transport Tools)**: Implementación robusta y diferenciable de Sinkhorn-Knopp.

### 1.2 Gestión de Precisión Numérica Global

Las operaciones en Rama D (Signatures) y el cálculo de derivadas de Malliavin son altamente sensibles al error de redondeo. JAX usa por defecto `float32`, lo que puede inducir derivas numéricas acumulativas en series temporales largas ( $N > 10000$ ). Se recomienda activar globalmente:

```
1 import jax
2 jax.config.update('jax_enable_x64', True)
```

Esta activación duplica la precisión a `float64`, preservando:

- Estabilidad numérica en exponentes de Hölder (derivadas de  $\tau(q)$ )
- Precisión en integrales de Skorokhod (sensibles a cuantización)
- Convergencia del algoritmo de Sinkhorn bajo condiciones extremas ( $\epsilon \rightarrow 0$ )

**Costo y Compensación:** El overhead es  $\sim 2\times$  en tiempo de compilación XLA y VRAM consumida, pero es esencial para la confiabilidad en producción cuando se procesan trayectorias de assets volátiles.

## Capítulo 2

# Módulo 1: Motor de Identificación (SIA)

### 2.1 Estimación WTMM con Callback Asíncrono

Uso de `jax.pure_callback` para invocar código CPU (PyWavelets) sin romper el grafo JIT ni la diferenciabilidad del resto del pipeline.

```
1 import pywt
2 import jax
3 import jax.numpy as jnp
4 from jax import jit, vmap
5 import numpy as np
6
7 class WTMM_Estimator:
8     def __init__(self, n_scales=40, j_min=1.0, j_max=6.0):
9         # Implementacion Fiel: Escalas Diadicas Densas
10        #  $a_j = 2^{\{j/v\}}$  para analisis multifractal preciso
11        # Usamos 10 voces por octava (densidad estandar en WTMM)
12
13        powers = jnp.linspace(j_min, j_max, num=n_scales)
14        self.scales = jnp.power(2.0, powers)
15        self.wavelet = 'gaus1'
16
17    def compute_cwt_safe(self, signal):
18        """
19        Wrapper seguro para llamar a PyWavelets desde una funcion JIT.
20        """
21        result_shape = (len(self.scales), signal.shape[0])
22
23        def _cwt_cpu(s, sc):
24            # Esta funcion corre en CPU puro con arrays de Numpy
25            coefs, _ = pywt.cwt(np.array(s), np.array(sc), 'gaus1')
26            return coefs.astype(np.float32)
27
28        # pure_callback permite inyectar valores externos en el grafo
29        coefs = jax.pure_callback(_cwt_cpu,
30                                jax.ShapeDtypeStruct(result_shape, jnp.float32),
31                                signal, self.scales)
32
33        return coefs
34
35    @staticmethod
36    @jit
37    def find_modulus_maxima(cwt_coefs):
38        # ... (Idem implementacion anterior) ...
39        c = jnp.abs(cwt_coefs)
40        left = jnp.roll(c, 1, axis=1)
```

```

40     right = jnp.roll(c, -1, axis=1)
41     is_local_max = (c > left) & (c > right)
42     return c * is_local_max
43
44 @staticmethod
45 @jit
46 def trace_skeletons_and_compute_tau(maxima_coeffs, scales, q_moments=jnp.array([-2.0,
47     -1.0, 1.0, 2.0]), C_influence=1.5):
48     """
49     Implementacion Fiel del Algoritmo 2 (Guia Universal): Enlace de Maximos y Funcion
50     de Particion.
51     """
52     # maxima_coeffs: Array [n_scales, time_steps] con los modulos en los maximos (0
53     en el resto).
54
55     # PASO 2: Enlace de Maximos (Tracking Vectorizado)
56     # Inicializamos las lineas activas en la escala mas gruesa (J)
57     active_lines = jnp.where(maxima_coeffs[-1] > 0, maxima_coeffs[-1], 0.0)
58
59     def link_scale(prev_active_lines, current_scale_data):
60         curr_maxima, curr_a = current_scale_data
61
62         # Solucion Estructural para XLA: Broadcasting en lugar de reduce_window
63         dinamico
64         # reduce_window requiere tamaños estaticos, pero el radio depende de curr_a (
65         Tracer).
66         # Usamos una mascara de distancia global.
67
68         # 1. Crear matriz de distancias relativas
69         n_time = prev_active_lines.shape[0]
70         indices = jnp.arange(n_time)
71         # Matriz (N, N): dist_matrix[i, j] = |i - j|
72         dist_matrix = jnp.abs(indices[:, None] - indices[None, :])
73
74         # 2. Definir el radio dinamico de influencia
75         radius = jnp.ceil(C_influence * curr_a)
76
77         # 3. Mascara de influencia (N, N)
78         # mask[i, j] es True si j influye en i (esta dentro del radio)
79         influence_mask = dist_matrix <= radius
80
81         # 4. Dilatacion segura con XLA (Max-Pool via Masked Reduction)
82         # Para cada punto i, calculamos el maximo de prev_active_lines[j] donde mask[
83         i,j] es True.
84         # Rellenamos con -inf donde no hay influencia para que el maximo funcione
85         masked_values = jnp.where(influence_mask, prev_active_lines[None, :], -jnp.
86         inf)
87         dilated_prev = jnp.max(masked_values, axis=1)
88
89         # Interseccion Logica (AND suave):
90         # Un maximo actual sobrevive SOLO si cae dentro del cono dilatado de un
91         ancestro
92         # Y ademas es un maximo local valido
93         linked_maxima = jnp.where((curr_maxima > 0) & (dilated_prev > 0), curr_maxima
94         , 0.0)
95
96         # Actualizamos lineas activas: propagamos la magnitud
97         return linked_maxima, linked_maxima
98
99     # Escaneo hacia arriba (escalas mas finas) usando jax.lax.scan
100     _, skeletons = jax.lax.scan(
101         link_scale,

```

```

94         active_lines,
95         (maxima_coeffs[: -1][::-1], scales[: -1][::-1])
96     )
97
98     # Reordenamos las escalas a su orden original
99     skeletons = jnp.vstack([skeletons[: -1], active_lines])
100
101     # PASO 3: Funcion de Particion Z(q, a)
102     def compute_zq(q):
103         # Filtrar ceros para evitar NaNs en potencias negativas
104         safe_skeletons = jnp.where(skeletons > 1e-8, skeletons, jnp.nan)
105         return jnp.nansum(safe_skeletons ** q, axis=1)
106
107     Z_q_a = vmap(compute_zq)(q_moments) # Forma: [n_q, n_scales]
108
109     # PASO 4: Exponentes tau(q) mediante regresion lineal (log Z vs log a)
110     log_a = jnp.log(scales)
111     log_Z = jnp.log(Z_q_a + 1e-8)
112
113     a_mean = jnp.mean(log_a)
114     def compute_slope(lz):
115         return jnp.sum((log_a - a_mean) * (lz - jnp.mean(lz))) / jnp.sum((log_a -
124 a_mean)**2)
116
117     tau_q = vmap(compute_slope)(log_Z)
118
119     # Espectro de Legendre: D(h) = min_q (q*h - tau(q))
120     # El Holder local 'h' se extrae de las derivadas de tau(q)
121     h_estimates = jnp.gradient(tau_q, q_moments)
122
123     # Retornamos el Holder minimo (la singularidad mas fuerte) para el Circuit
124     Breaker
125     return jnp.min(h_estimates)
126
127 def estimate_holder_exponent(self, signal, besov_c=1.5):
128     # Pipeline completo fiel a la Guia Universal
129     coefs = self.compute_cwt_safe(signal)
130     maxima = self.find_modulus_maxima(coefs)
131     # Inyectar el parametro de influencia de Besov calibrable
132     h_min = self.trace_skeletons_and_compute_tau(maxima, self.scales, C_influence=
133     besov_c)
134
135     # Retornar array escalar para consistencia
136     return jnp.array([h_min])
137
138 def compute_cwt_windowed(self, signal, window_size=1024):
139     """
140     Variante optimizada en memoria para buffers grandes (N_buf > 1024).
141     Usa jax.lax.reduce_window para max-pooling en lugar de construir
142     una matriz de distancias de tamaño O(N^2).
143
144     Aplicable cuando el buffer excede GPU VRAM disponible y la precision
145     multifractal puede relajarse ligeramente en favor de la estabilidad.
146     """
147     if signal.shape[0] <= window_size:
148         # Para buffers pequenos, usar com implantacao normal
149         return self.compute_cwt_safe(signal)
150
151     # Dividir en ventanas con overlap para mantener continuidad
152     stride = window_size // 2 # 50% overlap
153     coefs_chunks = []
154
155     for i in range(0, signal.shape[0] - window_size, stride):

```

```

154         chunk = signal[i:i + window_size]
155         coef_chunk = self.compute_cwt_safe(chunk)
156
157         # Tomar solo la parte 'nueva' para evitar duplicados
158         if i > 0:
159             coef_chunk = coef_chunk[:, stride:]
160             coefs_chunks.append(coef_chunk)
161
162         # Concatenar: forma [n_scales, n_time_processado]
163         coefs_full = jnp.concatenate(coefs_chunks, axis=1)
164         return coefs_full

```

## 2.2 Cálculo de Peso de Malliavin (Integral de Skorokhod)

Aumentamos el estado de la SDE para computar simultáneamente la integral estocástica requerida para las Griegas en payoffs discontinuos.

```

1 import jax
2 import jax.numpy as jnp
3 import diffrax
4
5 class MalliavinCalculator:
6     def __init__(self, drift, diffusion, inv_diffusion_fn):
7         self.drift = drift
8         self.diffusion = diffusion
9         self.inv_diffusion = inv_diffusion_fn
10
11     def solve_malliavin_system(self, x0, t_span, key):
12         """
13         Resuelve:
14         1. Estado:  $dX_t = b(X)dt + \sigma(X)dW_t$ 
15         2. Tangente:  $dY_t = b'(X)Y dt + \sigma'(X)Y dW_t$ 
16         3. Peso (Integral):  $dP_t = (\sigma^{-1}(X) Y)^T dW_t$ 
17         """
18         y0 = jnp.eye(x0.shape[0])
19         p0 = jnp.zeros(x0.shape[0]) # Malliavin weight accumulator
20
21         def vector_field(t, state, args):
22             x, y, p = state
23
24             # 1. Drift terminos
25             bx = self.drift(t, x)
26             db_dx = jax.jacfwd(lambda _x: self.drift(t, _x))(x)
27             by = db_dx @ y
28             bp = jnp.zeros_like(p) # Integral estocastica no tiene drift en Ito estandar
29
30             # 2. Diffusion terminos
31             sx = self.diffusion(t, x)
32             ds_dx = jax.jacfwd(lambda _x: self.diffusion(t, _x))(x)
33             sy = ds_dx @ y
34
35             # Termino del peso de Malliavin (Bismut-Elworthy-Li):
36             #  $dP_t = (\sigma^{-1}(X) Y \nabla b(X))^T dW_t$ 
37             #  $sp = (\sigma_{inv} @ Y @ drift\_jacobian).T$ 
38
39             s_inv = self.inv_diffusion(t, x)
40             # Correccion Teorica: Incluir Jacobiano del Drift en el integrando
41             #  $db\_dx @ y$  es la deformacion local del flujo determinista
42             # Multiplicamos por sigma inversa para convertirlo en ruido
43
44             # Nota: La formula exacta puede variar segun si buscamos Delta o Vega.
45             # Aqui asumimos la variacion estandar respecto a x0 via flujo tangente Y_t.

```

```

46         # Para Delta puro: weight ~ int (sigma^-1 Y_t)^T dW_t es la formula estandar
simplificada
47         # Pero el tratado exige la formulacion completa que acopla drift y difusion.
48
49         # sp = (s_inv @ y).T # Anterior (Incompleto)
50         sp = (s_inv @ (db_dx @ y)).T # Corregido (Con Drift Sensitivity)
51
52         return (bx, by, bp), (sx, sy, sp)
53
54     # Terminos para SDE Solver
55     # Malliavin requiere esquema fuerte 1.0 (Milstein) si difusion no es constante.
56     # DiffraX no tiene Milstein directo simple para ruido multidimensional general
sin conmutatividad.
57     # Pero podemos usar un esquema Runge-Kutta estocastico de orden fuerte 1.0 o 1.5.
58
59     # Usamos Heun estocastico (Trapezoidal) que converge mas fuerte que Euler
60     # 0 idealmente diffraX.ItoMilstein() si el ruido es escalar o conmutativo.
61     # Para maxima precision general: SRK1 (Strong Order 1.0)
62
63 class CoupledMalliavinTerm(diffraX.AbstractTerm):
64     """
65     Termino personalizado para Ecuaciones Diferenciales Estocasticas Acopladas (Malliavin
).
66     Maneja el producto tensorial explicito entre el estado PyTree ((D), (D,D), (D))
y el ruido browniano vectorial (D).
67     """
68
69     def __init__(self, diffusion_fn, brownian_path):
70         self.diffusion = diffusion_fn
71         self.control = brownian_path
72
73     def vf(self, t, y, args):
74         return self.diffusion(t, y, args)
75
76     def contr(self, t0, t1):
77         return self.control.evaluate(t0, t1)
78
79     def prod(self, vf, control):
80         # Esta es la logica critica que fallaba en ControlTerm estandar.
81         # vf es el output de diffusion_fn: (sx, sy, sp)
82         # control es el incremento browniano: dW (vector D)
83
84         sx, sy, sp = vf
85
86         # 1. Estado Primal (X_t): sx @ dW
87         dx_diff = jnp.dot(sx, control)
88
89         # 2. Estado Tangente (Y_t): Contraction (D,D,D) * (D) -> (D,D)
90         # sy_ijk * dW_k
91         dy_diff = jnp.einsum('ijk,k->ij', sy, control)
92
93         # 3. Peso Malliavin (P_t): Dot product (D) * (D) -> Scalar
94         dp_diff = jnp.dot(sp, control)
95
96         return (dx_diff, dy_diff, dp_diff)
97
98     def is_in_place(self, *args, **kwargs):
99         return False
100
101     # --- FIN DE LA CLASE ANIDADA ---
102
103     # Retomamos el flujo de solve_malliavin_system CON LA INDENTACION CORRECTA
104     # VirtualBrownianTree requiere 'shape' para definir la dimension del ruido W_t
105     # Asumimos ruido de misma dimension que el estado (Difusion Cuadrada)

```



```

106 brownian = diffrax.VirtualBrownianTree(t_span[0], t_span[1], tol=1e-3, shape=x0.shape
    , key=key)
107
108 # Definimos MultiTerm con nuestro termino personalizado
109 drift = diffrax.ODETerm(lambda t, s, a: vector_field(t, s, a)[0])
110
111 def diffusion_fn_wrapper(t, state, args):
112     return vector_field(t, state, args)[1]
113
114 diffusion = CoupledMalliavinTerm(diffusion_fn_wrapper, brownian)
115
116 terms = diffrax.MultiTerm(drift, diffusion)
117
118 # Solver seguro para ruido multidimensional general (Strong Order 0.5)
119 # Para Malliavin, Euler es suficiente si el paso es pequeño (dt=0.01)
120 solver = diffrax.Euler()
121 sol = diffrax.diffeqsolve(terms, solver, t0=t_span[0], t1=t_span[1],
122                          dt0=0.01, y0=(x0, y0, p0))
123
124 x_T = sol.ys[0][-1]
125 weight_integral = sol.ys[2][-1]
126
127 return x_T, weight_integral
128
129 def compute_delta(self, x0, t_span, key, payoff_fn):
130     #  $E[f(X_T) * Weight * (1/T)]$ 
131     x_T, integral = self.solve_malliavin_system(x0, t_span, key)
132     T = t_span[1] - t_span[0]
133     malliavin_weight = integral / T
134
135     return payoff_fn(x_T) * malliavin_weight

```

## Capítulo 3

# Módulo 2: Núcleos de Predicción

### 3.1 Rama A: Procesos de Lévy y Monte Carlo Vectorizado

Implementación del algoritmo de Chambers-Mallows-Stuck para simulación estable.

```
1 import jax.numpy as jnp
2 from jax import random, vmap
3
4 def simulate_stable_levy(key, alpha, beta, gamma, delta, n_samples):
5     """
6     Generador Vectorizado de Variables Alpha-Estables
7     Algoritmo: Chambers-Mallows-Stuck (1976)
8     """
9     k1, k2 = random.split(key)
10
11     # Variables auxiliares uniformes y exponenciales
12     phi = random.uniform(k1, shape=(n_samples,), minval=-jnp.pi/2, maxval=jnp.pi/2)
13     w = random.exponential(k2, shape=(n_samples,))
14
15     # Terminos S1, S2 segun parametrizacion (alpha != 1)
16     # Ver Predictor_Estocastico_Implementacion.tex Eq (3.4)
17
18     s_alpha_beta = (1 + (beta * jnp.tan(jnp.pi * alpha / 2))**2)**(1 / (2 * alpha))
19     b_alpha_beta = jnp.arctan(beta * jnp.tan(jnp.pi * alpha / 2)) / alpha
20
21     term1 = s_alpha_beta * (jnp.sin(alpha * (phi + b_alpha_beta))) / ((jnp.cos(phi))**(1/alpha))
22     term2 = ((jnp.cos(phi - alpha * (phi + b_alpha_beta))) / w)**((1 - alpha) / alpha)
23
24     z = term1 * term2
25
26     return gamma * z + delta
```

### 3.2 Rama B: Solvers DGM-PDE con Equinox

Uso de Deep Galerkin Method para resolver la ecuación HJB en alta dimensión.

```
1 import equinox as eqx
2 import diffrax
3
4 class DGM_HJB_Solver(eqx.Module):
5     # Red simple para V(t,x)
6     mlp: eqx.nn.MLP
7
8     def __init__(self, in_size, key):
9         self.mlp = eqx.nn.MLP(in_size, 1, width_size=64, depth=4, key=key, activation=jax.nn.tanh)
```

```

10
11 def __call__(self, t, x):
12     # Concatenar tiempo y espacio
13     t = jnp.array([t]) if jnp.ndim(t) == 0 else t
14     tx = jnp.concatenate([t, x])
15     return self.mlp(tx)[0]
16
17 def loss_hjb(model, t_batch, x_batch, hamiltonian_fn, terminal_cond_fn, boundary_cond_fn,
18     T, x_term_batch=None, t_bound_batch=None, x_bound_batch=None):
19     """
20     Computa la Loss Total DGM:  $L = L_{\text{interior}} + L_{\text{terminal}} + L_{\text{boundary}}$ 
21     Algoritmo 5 (Guia Universal) - Version Vectorizada (vmap)
22     """
23     # 1. Loss Interior (Residual PDE)
24     # Definimos el residual para UN solo punto (t, x) para que jax.grad funcione (retorno
25     # escalar)
26     def compute_single_residual(t_val, x_val):
27         # t_val: scalar, x_val: vector (dim espacial)
28
29         # Derivadas automaticas respecto al tiempo
30         v_t = jax.grad(lambda _t: model(_t, x_val))(_t_val)
31
32         # Derivadas automaticas respecto al espacio
33         v_x = jax.grad(lambda _x: model(t_val, _x))(x_val)
34         v_xx = jax.hessian(lambda _x: model(t_val, _x))(x_val)
35
36         # Residual HJB
37         return v_t + hamiltonian_fn(x_val, v_x, v_xx)
38
39     # Vectorizamos sobre el batch de entrenamiento usando vmap
40     # t_batch: [Batch], x_batch: [Batch, D]
41     residuals = vmap(compute_single_residual)(t_batch, x_batch)
42     loss_interior = jnp.mean(residuals**2)
43
44     # 2. Loss Terminal (Condicion de Contorno Temporal)
45     #  $V(T, x) = g(x)$ 
46
47     x_term = x_batch if x_term_batch is None else x_term_batch
48
49     # Vectorizacion simple de la inferencia
50     v_terminal_pred = vmap(lambda x: model(T, x))(x_term)
51     v_terminal_target = vmap(terminal_cond_fn)(x_term)
52
53     loss_terminal = jnp.mean((v_terminal_pred - v_terminal_target)**2)
54
55     # 3. Loss Frontera (Condicion de Contorno Espacial)
56     #  $V(t, x_b) = h(t, x_b)$ 
57
58     if x_bound_batch is not None and t_bound_batch is not None:
59         v_bound_pred = vmap(lambda t, x: model(t, x))(t_bound_batch, x_bound_batch)
60         v_bound_target = vmap(boundary_cond_fn)(t_bound_batch, x_bound_batch)
61         loss_boundary = jnp.mean((v_bound_pred - v_bound_target)**2)
62     else:
63         loss_boundary = 0.0
64
65     return loss_interior + loss_terminal + loss_boundary

```

### 3.2.1 Monitoreo de Entropía DGM (Detección de Mode Collapse)

Durante el entrenamiento de la red neuronal DGM, existe el riesgo de que la red colapse a soluciones triviales (constantes) que satisfacen formalmente la PDE pero carecen de contenido informativo. Esta subsección implementa el **Principio de Conservación de Entropía de Solución** del documento de Teoría.

```
1 def compute_entropy_dgm(model, t, x_samples, num_bins=50):
2     """
3     Calcula la entropia diferencial de la solucion DGM en tiempo t
4     sobre una distribucion de puntos espaciales x_samples.
5
6      $H[V_{\theta}] = - \int p(v) \log p(v) dv$ 
7
8     Args:
9         model: Red DGM ( $V_{\theta}$ )
10        t: Tiempo de evaluacion (scalar)
11        x_samples: Puntos de muestreo espacial [N, D]
12        num_bins: Numero de bins para estimacion de densidad
13
14    Returns:
15        entropy: Entropia diferencial estimada (scalar)
16    """
17    # Evaluar la red en los puntos de muestreo
18    v_values = vmap(lambda x: model(t, x))(x_samples) # [N]
19
20    # Estimar densidad mediante histograma normalizado
21    # Para estimacion mas precisa, usar KDE (Kernel Density Estimation)
22    # pero histograma es mas eficiente para monitoreo en linea
23
24    hist, bin_edges = jnp.histogram(v_values, bins=num_bins, density=True)
25
26    # Ancho de bins para normalizacion
27    bin_width = bin_edges[1] - bin_edges[0]
28
29    # Probabilidades normalizadas
30    #  $p_i = \text{hist}_i * \text{bin\_width}$  (para que  $\sum(p_i) = 1$ )
31    probs = hist * bin_width
32
33    # Entropia:  $H = - \sum p_i \log(p_i)$ 
34    # Evitar  $\log(0)$  agregando epsilon
35    log_probs = jnp.log(probs + 1e-10)
36    entropy = -jnp.sum(probs * log_probs)
37
38    return entropy
39
40 def check_mode_collapse(model, t_eval, x_samples,
41                         terminal_entropy, gamma=0.5):
42     """
43     Verifica si la red DGM ha colapsado a solucion trivial.
44
45     Criterio del Teorema de Conservacion de Entropia:
46      $H[V_{\theta}(t)] \geq \gamma * H[g]$ 
47
48     Args:
49         model: Red DGM
50         t_eval: Tiempos de evaluacion [T_eval]
51         x_samples: Puntos espaciales [N, D]
52         terminal_entropy:  $H[g]$  - entropia de condicion terminal
53         gamma: Factor de retencion [0.5, 1.0]
54
55     Returns:
56         collapsed: bool - True si mode collapse detectado
```

```

57     avg_entropy: Entropia promedio temporal
58     """
59     # Calcular entropia en cada tiempo
60     entropies = jnp.array([
61         compute_entropy_dgm(model, t, x_samples)
62         for t in t_eval
63     ])
64
65     # Entropia promedio temporal
66     avg_entropy = jnp.mean(entropies)
67
68     # Criterio de colapso
69     threshold = gamma * terminal_entropy
70     collapsed = avg_entropy < threshold
71
72     return collapsed, avg_entropy
73
74 # Ejemplo de uso en loop de entrenamiento
75 def train_dgm_with_entropy_monitoring(model, optimizer,
76                                     train_data, gamma=0.5):
77     """
78     Entrenamiento DGM con monitoreo de entropia para prevenir
79     mode collapse.
80     """
81     # Calcular entropia terminal (baseline)
82     # g(x) es la condicion terminal (payoff function)
83     x_terminal_samples = train_data['x_terminal']
84     g_values = vmap(terminal_cond_fn)(x_terminal_samples)
85
86     hist_term, edges_term = jnp.histogram(g_values, bins=50, density=True)
87     bin_w = edges_term[1] - edges_term[0]
88     probs_term = hist_term * bin_w
89     H_terminal = -jnp.sum(probs_term * jnp.log(probs_term + 1e-10))
90
91     opt_state = optimizer.init(model)
92
93     for epoch in range(num_epochs):
94         # Loss estandar DGM
95         loss, grads = jax.value_and_grad(loss_hjb)(
96             model, t_batch, x_batch,
97             hamiltonian_fn, terminal_cond_fn,
98             boundary_cond_fn, T
99         )
100
101         # Actualizar parametros
102         updates, opt_state = optimizer.update(grads, opt_state)
103         model = optax.apply_updates(model, updates)
104
105         # Monitoreo de entropia cada K epochs
106         if epoch % 10 == 0:
107             t_eval = jnp.linspace(0, 0.9*T, 20) # Evaluar hasta 90% del tiempo
108             x_eval = x_batch # Reusar batch de entrenamiento
109
110             collapsed, avg_H = check_mode_collapse(
111                 model, t_eval, x_eval, H_terminal, gamma
112             )
113
114             if collapsed:
115                 print(f"WARNING: Mode collapse detected at epoch {epoch}")
116                 print(f"  Avg entropy: {avg_H:.4f}")
117                 print(f"  Threshold: {gamma*H_terminal:.4f}")
118
119                 # Accion correctiva: reinicializar red o ajustar hiperparametros

```

```
120         # En produccion: reducir peso rho_B -> 0 en orquestador
121         break
122
123     return model
```

**Nota Teórica:** Este monitoreo implementa el Teorema de Conservación de Entropía (Sección 3.2 del documento de Teoría). Una solución colapsada tiene  $H[V_\theta] \rightarrow -\infty$  (distribución delta), violando el criterio  $H_{avg} \geq \gamma \cdot H[g]$ . En el orquestador JKO, si se detecta colapso persistente ( $> 10$  pasos consecutivos), se debe reducir  $\rho_B \rightarrow 0$  hasta re-entrenar la red DGM.

## Capítulo 4

# Módulo 3: Orquestador JKO (Jordan-Kinderlehrer-Otto)

### 4.1 Sinkhorn Estabilizado en Log-Domain con OTT-JAX

Implementación numéricamente robusta del algoritmo de transporte óptimo entrópico.

```
1 import jax.numpy as jnp
2 from ott.geometry import geometry
3 from ott.problems.linear import linear_problem
4 from ott.solvers.linear import sinkhorn
5
6 class JKO_Discreto:
7     def __init__(self, epsilon=1e-2):
8         self.epsilon = epsilon # Regularizacion entropica
9
10    def solve_ot_step(self, weights_prev, gradients_energy, tau=0.1):
11        """
12        Resuelve un paso del esquema JKO:
13         $\rho_{k+1} = \operatorname{argmin}_{\rho} \{ \text{Energy}(\rho) + (1/2\tau)W_2^2(\rho, \rho_k) \}$ 
14
15        Usamos la formulacion proximal:
16         $\rho_{k+1} = P \#_{\epsilon} (\rho_k * \exp(-\tau * \text{grad}_E))$ 
17        Donde P es el mapa de transporte entropico.
18        """
19
20        # 1. Paso explicito (Gradient Descent en espacio de probabilidad)
21        log(rho_target) = log(rho_prev) - tau * grad_E
22        log_weights_target = jnp.log(weights_prev + 1e-8) - tau * gradients_energy
23        weights_target = jax.nn.softmax(log_weights_target)
24
25        # 2. Proyeccion Wasserstein (Sinkhorn)
26        # En JKO puro, minimizamos  $W_2^2$ . Aqui usamos Sinkhorn para encontrar
27        # la proyeccion mas cercana en geometria de transporte si hay restricciones.
28        # Si no hay restricciones espaciales complejas, el paso softmax es suficiente
29        # para la version "Mean Field".
30        # Para rigor completo, definimos una geometria entre los "modelos" (nucleos)
31
32        # Asumimos que la "distancia" entre modelos es uniforme (todos equidistantes)
33        # 0 definimos una matriz de covarianza entre predictores
34        # Costo: penaliza moverse lejos de la diagonal. Costo 0 en diagonal.
35        C = 1.0 - jnp.eye(len(weights_prev))
36        geom = geometry.Geometry(cost_matrix=C, epsilon=self.epsilon)
37
38        prob = linear_problem.LinearProblem(geom, a=weights_prev, b=weights_target)
39        solver = sinkhorn.Sinkhorn(lse_mode=True)
40        out = solver(prob)
```

```

41     # El plan de transporte optimo P (out.matrix) es el acoplamiento pi(x, y).
42     # Por definicion de Transporte Optimo (Kantorovich), las marginales de P son 'a'
43     y 'b'.
44     # En el esquema JKO, buscamos la proyeccion de la distribucion actual en la
45     direccion del gradiente.
46     # La nueva distribucion rho_{k+1} es efectivamente la marginal 'b' (
47     weights_target) ajustada
48     # por la regularizacion de Sinkhorn si no convergio perfectamente,
49     # o mas precisamente, la marginal proyeccion de la masa transportada.
50
51     # Correccion Matematica:
52     # out.matrix es la matriz de acoplamiento pi_{ij}.
53     # La masa total que llega al destino j es la suma sobre i de pi_{ij}.
54     # No debemos multiplicar por weights_prev nuevamente, pues pi_{ij} ya contiene la
55     masa.
56
57     transported_weights = jnp.sum(out.matrix, axis=0)
58
59     # Asegurar normalizacion (por si hay pequenas fugas numericas)
60     transported_weights = transported_weights / jnp.sum(transported_weights)
61
62     return transported_weights
63
64 def solve_ot_step_with_entropy_annealing(self, weights_prev, gradients_energy, tau
65 =0.1, max_iter_sinkhorn=100):
66     """
67     Variante robusta del algoritmo JKO con Recocido Entrópico (Entropy Annealing).
68
69     Bajo condiciones de mercado extremas (ej. gaps de precio, vol explota), el
70     algoritmo
71     de Sinkhorn puede divergir si epsilon es muy pequeno. Este metodo implementa 3
72     intentos:
73
74     1. Intenta convergencia con epsilon nominal
75     2. Si falla, duplica epsilon (entropy annealing)
76     3. Si sigue fallando, retorna Safe Weight Fallback: pesos uniformes [0.25, 0.25,
77     0.25, 0.25]
78     con flag InferenciaDegradada activado
79
80     Teoria: El recocido entropico intercambia precision por robustez. Si todo falla,
81     la distribucion uniforme es el estado de maxima incertidumbre (Jaynes). Mejor que
82     crash.
83     """
84
85     # 1. Paso exponencial (Gradient Descent en espacio de probabilidad)
86     log_weights_target = jnp.log(weights_prev + 1e-8) - tau * gradients_energy
87     weights_target = jax.nn.softmax(log_weights_target)
88
89     C = 1.0 - jnp.eye(len(weights_prev))
90
91     # Intento 1: epsilon nominal
92     geom = geometry.Geometry(cost_matrix=C, epsilon=self.epsilon)
93     prob = linear_problem.LinearProblem(geom, a=weights_prev, b=weights_target)
94     solver = sinkhorn.Sinkhorn(lse_mode=True, max_iterations=max_iter_sinkhorn)
95     out = solver(prob)
96
97     # Diagnosticar convergencia: Si n_iterations >= max_iter_sinkhorn, reintentar
98     converged_nominal = out.n_iterations < max_iter_sinkhorn
99
100     def try_annealed():
101         """Plan B: Epsilon annealing (2x) para convergencia robusta."""
102         epsilon_annealed = self.epsilon * 2.0

```



```

95         geom_anneal = geometry.Geometry(cost_matrix=C, epsilon=epsilon_annealed)
96         prob_anneal = linear_problem.LinearProblem(geom_anneal, a=weights_prev, b=
weights_target)
97         solver_anneal = sinkhorn.Sinkhorn(lse_mode=True, max_iterations=
max_iter_sinkhorn)
98         out_anneal = solver_anneal(prob_anneal)
99         converged_anneal = out_anneal.n_iterations < max_iter_sinkhorn
100         return out_anneal, converged_anneal
101
102     def safe_uniform_fallback():
103         """Plan C: Fallback uniforme ante fallo catastrófico de Sinkhorn.
104         Retorna pesos uniformes [0.25, 0.25, 0.25, 0.25] (máxima entropía).
105         El flag InferenciaDegradada se activa en el orquestador principal.
106         """
107         n_branches = len(weights_prev)
108         uniform_weights = jnp.ones(n_branches) / n_branches
109         return uniform_weights, False # False = degraded mode
110
111     # Ejecutar lógica de fallback
112     if converged_nominal:
113         transported_weights = jnp.sum(out.matrix, axis=0)
114         transported_weights = transported_weights / jnp.sum(transported_weights)
115         converged_final = True
116     else:
117         out_anneal, converged_anneal = try_annealed()
118         if converged_anneal:
119             transported_weights = jnp.sum(out_anneal.matrix, axis=0)
120             transported_weights = transported_weights / jnp.sum(transported_weights)
121             converged_final = True
122         else:
123             # Ultimo recurso: pesos uniformes
124             transported_weights, converged_final = safe_uniform_fallback()
125
126     return transported_weights, converged_final

```

## 4.2 Integración con Optax para Aprendizaje de Metapámetros

Optimización de los hiperparámetros del orquestador (tasas de aprendizaje, regularización).

```

1 import optax
2
3 def make_optimizer(learning_rate):
4     # Optimizador AdamW con weight decay para regularizacion
5     optimizer = optax.adamw(learning_rate, weight_decay=1e-4)
6     return optimizer
7
8 def update_metaparameters(params, grads, opt_state, optimizer):
9     """
10     Actualiza los parametros del orquestador (ej. pesos de atencion, tasas)
11     usando gradientes calculados via backprop kroz del tiempo.
12     """
13     updates, new_opt_state = optimizer.update(grads, opt_state, params)
14     new_params = optax.apply_updates(params, updates)
15     return new_params, new_opt_state

```

## 4.3 Rama C: Esquema IMEX con Solver de Punto Fijo Robusto

Splitting Implícito-Explícito utilizando jaxopt para garantizar convergencia en la parte rígida.

```

1 import jaxopt
2 import jax.numpy as jnp

```

```

3 from jax.scipy.signal import convolve
4
5 def compute_jump_fft(u, kernel_fft):
6     """
7     Evalua la integral de salto (convolucion compensada) usando el Teorema de Convolucion
8     via FFT.
9     Operador no-local:  $L u(x) = \int (u(x+y) - u(x)) \nu(dy)$ 
10    =  $(u * \nu)(x) - u(x) * \lambda$ 
11    """
12    # 1. Transformada al dominio de la frecuencia
13    u_fft = jnp.fft.fft(u)
14
15    # 2. Producto punto a punto (convolucion circular)
16    # kernel_fft debe ser pre-calculado para eficiencia
17    conv_fft = u_fft * kernel_fft
18
19    # 3. Transformada inversa (retornar parte real)
20    convolution_term = jnp.real(jnp.fft.ifft(conv_fft))
21
22    # 4. Compensacion de Levy (Conservacion de Masa)
23    # El termino  $-(\lambda * u)$  es necesario porque la masa salta FUERA de x.
24    #  $\lambda$  (intensidad total) es la suma del kernel = kernel_fft[0] (Componente DC)
25
26    lambda_intensity = jnp.real(kernel_fft[0])
27    jump_integral = convolution_term - lambda_intensity * u
28
29    return jump_integral
30
31 def imex_step(x_curr, dt, drift_stiff, jump_kernel_fft, diffusion, key):
32     """
33     Esquema IMEX de 1er orden para PIDE con Saltos (Levy).
34     Parte Implicita: Difusion + Drift Local Stiff
35     Parte Explicita: Integral de Saltos (No-Local) via FFT
36     """
37     noise = random.normal(key, x_curr.shape) * jnp.sqrt(dt)
38
39     # 1. Evaluar termino no-local (integral de salto) explicitamente
40     # drift_nonstiff (Jumps) =  $\lambda * \int (u(y) - u(x)) \nu(dy) \sim$  Convolucion
41     jump_term = compute_jump_fft(x_curr, jump_kernel_fft)
42
43     # Parte explicita total
44     explicit_part = x_curr + dt * jump_term + diffusion(x_curr) * noise
45
46     # 2. Solver Implicito para Drift Stiff (Reaccion/Difusion Local)
47     #  $y - dt * f_I(y) = \text{explicit\_part}$ 
48     def fixed_point_op(y, _):
49         return explicit_part + dt * drift_stiff(y)
50
51     # Solver robusto (Anderson Acceleration converge mas rapido que Picard simple)
52     solver = jaxopt.AndersonAcceleration(fixed_point_op, maxiter=10, tol=1e-5)
53
54     # Iniciar con x_curr como guess
55     x_new, state = solver.run(x_curr, None)
56
57     return x_new

```

## 4.4 Rama D: Log-Signatures con Signax

Cálculo de características topológicas de rutas rugosas.

```

1 import signax # Libreria JAX para signatures
2

```

```

3 def compute_features(path, depth=3):
4     # path: [Batch, Time, Channels]
5     # Usamos signax nativo de JAX para calcular log-signatures
6     # Esto permite backprop a traves de la signature si fuera necesario
7
8     signature = signax.signature(path, depth)
9     log_sig = signax.logsignature(path, depth)
10
11     return log_sig

```

## Capítulo 5

# Integración y Pipeline

### 5.1 Clase Maestra PredictionEngine

Coordinación asíncrona de los módulos.

```
1 class UniversalPredictor:
2     def __init__(self, epsilon=1e-2, tau=0.1, holder_threshold=0.1,
3                 signature_depth=3, signal_buffer_size=128,
4                 cusum_k=0.5, cusum_h=5.0, error_alpha=0.05,
5                 besov_c=1.5): # Parametro de influencia de Besov
6         self.sia = WTMM_Estimator()
7
8         # Guardar parametro Besov para inyectarlo en cada step
9         self.besov_c = besov_c
10
11        # Buffer circular Estatico (Performance Hack XLA)
12        # Usamos JAX arrays estaticos para evitar recompilacion en cada paso
13        # Inicializamos con ceros; CWT ignorara el inicio si usamos padding correcto o
14        # controlamos el indice.
15        self.max_buffer_size = signal_buffer_size
16        self.signal_buffer = jnp.zeros(signal_buffer_size)
17        self.buffer_idx = 0
18
19        self.min_buffer_size = 32 # Minimo necesario para una wavelet de escala media
20
21        # Inyectar profundidad de signature en Kernel D (Topologico)
22        # Asumimos que KernelD acepta 'depth' en su constructor
23        self.kernels = [
24            KernelA(),
25            KernelB(),
26            KernelC(),
27            KernelD(depth=signature_depth)
28        ]
29
30        self.orchestrator = JKO_Discreto(epsilon=epsilon)
31        self.prev_weights = jnp.ones(4) / 4.0
32        self.tau = tau
33        self.holder_threshold = holder_threshold
34
35        # Estado CUSUM para deteccion de cambio de regimen
36        # Incluimos rastreo de varianza (EMA) para estandarizacion dinamica
37        self.cusum_state = {
38            'g_plus': 0.0,
39            'g_minus': 0.0,
40            'threshold': cusum_h,
41            'slack_k': cusum_k, # Parametro de deriva calibrable
42            'alpha_var': error_alpha, # Memoria de volatilidad calibrable
43            'error_sq_ema': 0.1, # Varianza inicial estimada
```

```

43         'n_obs': 0
44     }
45
46     def fit(self, historical_data):
47         """
48         Calibracion o Warm-up del modelo online usando datos historicos.
49         Procesa la serie temporal para estabilizar pesos JKO y estados internos.
50         """
51         # 0. Calibrar Nucleos Individuales (Ramas A, B, C, D)
52         # Cada kernel ajusta sus parametros internos (ej. redes neuronales DGM,
parametros Levy)
53         # Esto es critico para que las predicciones base no sean ruido aleatorio.
54         for kernel in self.kernels:
55             # Asumimos contrato de interfaz: kernel.fit(data)
56             if hasattr(kernel, 'fit'):
57                 kernel.fit(historical_data)
58
59         # Simulamos el paso del tiempo para actualizar pesos y CUSUM
60         # Asumimos que historical_data es un array [Time, Features]
61         # Y que la target es la propia serie (autoregresivo) o parte de ella
62
63         # Reset de estados por seguridad
64         self.prev_weights = jnp.ones(4) / 4.0
65         self.cusum_state['g_plus'] = 0.0
66         self.cusum_state['g_minus'] = 0.0
67
68         # Bucle de warm-up (sin guardar predicciones)
69         # En JAX puro, esto deberia ser un scan, pero mantenemos loop python
70         # por legibilidad en esta guia, dado que fit() se llama pocas veces.
71
72         for t in range(len(historical_data)):
73             current_obs = historical_data[t]
74
75             # Correction Causal (Time-Shift Bug):
76             # En validacion One-Step-Ahead, la observacion que ACABA de llegar (
current_obs)
77             # es el target real para evaluar la prediccion que hicimos en el paso
anterior (t-1).
78             # Por tanto, previous_target = current_obs.
79
80             # Step actualiza pesos y CUSUM internamente evaluando error = current_obs -
last_pred
81             _, _ = self.step(current_obs, previous_target=current_obs)
82
83     def predict(self, test_data):
84         """
85         Generacion de pronosticos secuenciales fuera de muestra.
86         """
87         predictions = []
88
89         for t in range(len(test_data)):
90             current_obs = test_data[t]
91
92             # Predecir paso t (usando error del paso t-1 evaluado contra current_obs)
93             # El argumento previous_target se usa dentro de step() para calcular el error
94             # de la prediccion anterior. Ese target es la observacion actual.
95
96             pred, _ = self.step(current_obs, previous_target=current_obs)
97             predictions.append(pred)
98
99         return jnp.array(predictions)
100
101     def _check_regime_change(self, prediction_error):

```

```

102     # Implementacion del Algoritmo 3 (CUSUM Secuencial) con Residuos Estandarizados
103     # Correccion Dimensionalidad: Reducir error vectorial a escalar (Norma L2)
104     # para evitar ValueError en decisiones booleanas.
105
106     # Calculamos la magnitud del error (escalar) conservando el signo si es 1D,
107     # o usando la norma si es multivariado.
108     # Para deteccion general de "falta de ajuste", usamos el error cuadratico medio
instantaneo.
109
110     error_sq = jnp.sum(prediction_error**2)
111     error_norm = jnp.sqrt(error_sq)
112
113     # 1. Actualizar estimacion de volatilidad del error (EMA escalar)
114     alpha_ema = self.cusum_state['alpha_var'] # Memoria calibrada por Optuna
115     current_var = self.cusum_state['error_sq_ema']
116     new_var = (1 - alpha_ema) * current_var + alpha_ema * error_sq
117
118     # Estandarizacion:  $s_t \sim (e^2 / \sigma^2) - 1$  (Chi-squared check)
119     # 0 mas simple para CUSUM de media en magnitud:  $s_t = (|e| - \mu_e) / \sigma_e$ 
120     # Asumimos que bajo regimen normal, error_norm tiene media correlacionada con
sqrt(var).
121
122     sigma_t = jnp.sqrt(new_var + 1e-8)
123
124     # Score estandarizado: Cuantas desviaciones estandar nos alejamos
125     s_standardized = (error_norm / sigma_t)
126     # Restamos el bias esperado (1.0 bajo asuncion normal aprox) para centrar en 0
127     s_centered = s_standardized - 1.0
128
129     # Guardar estado actualizado
130     self.cusum_state['error_sq_ema'] = new_var
131     self.cusum_state['n_obs'] += 1
132
133     # 2. Logica CUSUM Unilateral (solo nos importa si el error crece)
134     k = self.cusum_state['slack_k'] # Slack calibrado por Optuna
135     h = self.cusum_state['threshold']
136
137     # Solo monitoreamos g_plus (aumento de error) para detectar ruptura de modelo
138     self.cusum_state['g_plus'] = jnp.maximum(0.0, self.cusum_state['g_plus'] +
s_centered - k)
139     # g_minus no es relevante para deteccion de error (error disminuyendo es bueno)
140     self.cusum_state['g_minus'] = 0.0
141
142     alarm = self.cusum_state['g_plus'] > h
143     return alarm
144
145 def _check_regime_change_with_kurtosis(self, prediction_error, window_size=252):
146     """
147     Version mejorada de CUSUM con ajuste adaptativo del umbral basado en curtosis.
148
149     Implementa el Lema de Umbral Adaptativo del documento de Teoria:
150      $h_t = k * \sigma_t * (1 + \ln(\kappa_t / 3))$ 
151
152     Args:
153         prediction_error: Error de prediccion (scalar o vector)
154         window_size: Tamano de ventana para calculo de curtosis (default 252 ~ 1 ano)
155
156     Returns:
157         alarm: bool - True si cambio de regimen detectado
158         kurtosis: float - Curtosis empirica actual
159     """
160     # Conversion a escalar
161     error_sq = jnp.sum(prediction_error**2)

```

```

162     error_norm = jnp.sqrt(error_sq)
163
164     # 1. Actualizar buffer de errores para curtosis
165     # Inicializar buffer si no existe
166     if not hasattr(self, 'error_buffer'):
167         self.error_buffer = jnp.zeros(window_size)
168         self.error_buffer_idx = 0
169
170     # Actualizar buffer circular
171     self.error_buffer = self.error_buffer.at[self.error_buffer_idx].set(error_norm)
172     self.error_buffer_idx = (self.error_buffer_idx + 1) % window_size
173
174     # 2. Calcular curtosis empirica (cuarto momento estandarizado)
175     # Solo calcular cuando tengamos suficientes observaciones
176     valid_errors = jnp.where(
177         jnp.arange(window_size) < min(self.cusum_state['n_obs'], window_size),
178         self.error_buffer,
179         jnp.nan
180     )
181
182     # Media y varianza del buffer (sin NaNs)
183     mu_e = jnp.nanmean(valid_errors)
184     sigma_e_sq = jnp.nanvar(valid_errors)
185     sigma_e = jnp.sqrt(sigma_e_sq + 1e-8)
186
187     # Cuarto momento
188     m_4 = jnp.nanmean((valid_errors - mu_e)**4)
189
190     # Curtosis: kappa = E[(e - mu)^4] / sigma^4
191     kappa_t = m_4 / (sigma_e**4 + 1e-10)
192
193     # Durante warm-up, asumir curtosis Gaussiana
194     kappa_t = jnp.where(
195         self.cusum_state['n_obs'] < window_size,
196         3.0, # Curtosis Gaussiana
197         kappa_t
198     )
199
200     # 3. Calcular umbral adaptativo
201     # h_t = k * sigma * (1 + ln(kappa/3))
202     k = self.cusum_state['slack_k']
203
204     # Actualizar varianza con EMA (para consistencia con version original)
205     alpha_ema = self.cusum_state['alpha_var']
206     current_var = self.cusum_state['error_sq_ema']
207     new_var = (1 - alpha_ema) * current_var + alpha_ema * error_sq
208     sigma_t = jnp.sqrt(new_var + 1e-8)
209
210     # Umbral adaptativo con curtosis
211     # ln(kappa/3) puede ser negativo si kappa < 3 (sub-Gaussiano, muy raro en
    finanzas)
212     # Clampeamos para evitar umbrales negativos
213     kurtosis_adjustment = jnp.log(jnp.maximum(kappa_t, 1.0) / 3.0)
214     h_adaptive = k * sigma_t * (1.0 + kurtosis_adjustment)
215
216     # 4. Estadística CUSUM estandarizada
217     s_standardized = error_norm / sigma_t
218     s_centered = s_standardized - 1.0
219
220     # 5. Actualizar acumulador CUSUM
221     self.cusum_state['g_plus'] = jnp.maximum(
222         0.0,
223         self.cusum_state['g_plus'] + s_centered - k

```

```

224     )
225
226     # Actualizar estado
227     self.cusum_state['error_sq_ema'] = new_var
228     self.cusum_state['n_obs'] += 1
229
230     # 6. Deteccion con umbral adaptativo
231     alarm = self.cusum_state['g_plus'] > h_adaptive
232
233     return alarm, kappa_t, h_adaptive

```

### Ejemplo de Uso e Interpretacion de Curtosis:

```

1  # En el metodo step() del PredictionEngine, reemplazar:
2  # regime_changed = self._check_regime_change(last_error)
3  # por:
4  regime_changed, kurtosis, h_adaptive = self._check_regime_change_with_kurtosis(last_error
5  )
6  # Logueo de telemetria
7  if kurtosis > 5.0:
8      print(f"High volatility regime detected: kurtosis = {kurtosis:.2f}")
9  if kurtosis > 15.0:
10     print(f"Crisis regime: kurtosis = {kurtosis:.2f}, adaptive threshold = {h_adaptive:.4
11     f}")
12 if kurtosis > 20.0:
13     print(f"WARNING: Extreme fat tails - residual model may be invalid")
14
15 # Interpretacion:
16 # - kappa 3: Regimen Gaussiano (mercado normal)
17 # - kappa [5, 10]: Volatilidad financiera estandar
18 # - kappa [10, 15]: Alta volatilidad (eventos outlier frecuentes)
19 # - kappa > 15: Regimen de crisis (colas extremadamente pesadas)
20 # - kappa > 20: Falla del modelo de residuos - considerar cambio de arquitectura

```

**Nota Teorica:** Esta implementacion refleja el **Lema de Umbral Adaptativo** (Seccion 2.3 del documento de Teoria). El ajuste logaritmico  $\ln(\kappa_t/3)$  permite que el umbral CUSUM se expanda automaticamente en regimenes de colas pesadas, evitando falsos positivos mientras mantiene sensibilidad a cambios estructurales genuinos. La formulacion esta derivada de la Desigualdad de Markov de cuarto orden y tiene consistencia asintotica probada.

```

1
2  def step(self, new_data, previous_target=None):
3      # 0. Actualizacion del Buffer Circular Estatico (Fix XLA Recompile)
4      # Convertimos new_data a escalar representativo
5      scalar_obs = new_data[0] if hasattr(new_data, '__len__') and len(new_data) > 0
6      else new_data
7
8      # Desplazamiento ciclico eficiente (Roll)
9      # self.signal_buffer = [x_1, x_2, ..., x_N] -> [x_2, ..., x_N, new_x]
10
11     # Ojo: jnp.roll devuelve un nuevo array (immutable). Debemos reemplazar la
12     referencia.
13     # Si buffer_idx < max_size, estamos en llenado inicial.
14     # Pero para XLA estatico, siempre mantenemos el array full size.
15
16     self.signal_buffer = jnp.roll(self.signal_buffer, shift=-1)
17     self.signal_buffer = self.signal_buffer.at[-1].set(float(scalar_obs))
18
19     # Incremento contador de observaciones validas (hasta saturar)
20     new_count = self.buffer_idx + 1
21     self.buffer_idx = min(new_count, self.max_buffer_size) # Clamp al maximo int
22     standard de python

```



```

21     # 1. Identificacion (Singularidad Holderiana)
22     # Siempre pasamos el buffer COMPLETO de tamaño fijo a la función JIT.
23     # WTM calculara sobre todo el buffer con el parametro de Besov ajustado.
24
25     meta_state_h = self.sia.estimate_holder_exponent(self.signal_buffer, besov_c=self
26     .besov_c)
27
28     # Logica condicional fuera de JAX puro (o con where) para el warm-up
29     if self.buffer_idx < self.min_buffer_size:
30         meta_state_h = jnp.array([0.5]) # Default browniano durante arranque
31
32     # 1.1 Detección de Cambio de Régimen (CUSUM)
33     # Calcular error de la predicción anterior si existe target
34     last_error = 0.0
35
36     # Inicializar last_pred si no existe (startup)
37     if not hasattr(self, 'last_pred'):
38         self.last_pred = jnp.zeros_like(new_data)
39
40     if previous_target is not None:
41         # Necesitamos haber guardado la predicción anterior (self.last_pred)
42         # last_pred debe tener la misma forma que target
43         last_error = previous_target - self.last_pred
44     else:
45         # Si no hay target previo (burn-in inicial), el error es cero vector
46         last_error = jnp.zeros_like(new_data)
47
48     # Validación dimensional: CUSUM internamente reduce a escalar
49     # Retorna un booleano único (True/False)
50     regime_changed = self._check_regime_change(last_error)
51
52     # 2. Circuit Breaker (Robustez)
53     loss_type = 'mse' # Default
54
55     if jnp.min(meta_state_h) < self.holder_threshold: # Singularidad detectada (Crash
56     /Salto) con Umbral Dinámico
57         # Forzar peso a signatures (Kernel D) con epsilon de seguridad
58         epsilon = 1e-8
59         weights = jnp.array([epsilon, epsilon, epsilon, 1.0])
60         weights = weights / jnp.sum(weights)
61
62         loss_type = 'huber' # Activar robustez para el siguiente paso
63
64     elif regime_changed:
65         # Reinicio Entropico (Softmax Uniforme)
66         # El cambio estructural invalida la historia de pesos
67         weights = jnp.ones(len(self.kernels)) / len(self.kernels)
68
69         # Reset CUSUM
70         self.cusum_state['g_plus'] = 0.0
71         self.cusum_state['g_minus'] = 0.0
72
73     else:
74         # Calcular gradientes reales basados en el error anterior
75         energy_grads = jnp.zeros(len(self.kernels))
76
77         if previous_target is not None and hasattr(self, 'last_kernel_preds'):
78             # Recuperar volatilidad reciente del error para escalar la robustez
79             # Esto hace que el parametro delta sea adaptativo y universal (scale-
80             invariant)
81             current_volatility = jnp.sqrt(self.cusum_state['error_sq_ema'] + 1e-8)
82
83             # Definir función de pérdida local para JAX autograd

```

```

81         def loss_objective(w):
82             pred = jnp.dot(w, self.last_kernel_preds)
83             diff = pred - previous_target
84
85             if self.last_loss_type == 'huber':
86                 # Huber Loss robusta: delta depende de la escala de volatilidad
87                 # Usamos 1.35 * sigma para eficiencia del 95% en distribucion
88
89                 normal
90
91                 delta = 1.35 * current_volatility
92
93                 abs_diff = jnp.abs(diff)
94                 is_small = abs_diff <= delta
95                 loss_elements = jnp.where(is_small, 0.5 * diff**2, delta * (
96                     abs_diff - 0.5 * delta))
97                 return jnp.sum(loss_elements) # Retornar Energia Escalar Total
98             else:
99                 return 0.5 * jnp.sum(diff**2) # MSE Escalar Total
100
101             # Obtener gradiente de la Energia (Loss) respecto a los pesos (rho)
102             energy_grads = jax.grad(loss_objective)(self.prev_weights)
103
104             # Resolver flujo JKO con gradientes reales
105             # IMPORTANTE: Inyectar el parametro 'tau' optimizado por Optuna
106             # De lo contrario, se usaria el default (0.1), ignorando el aprendizaje de
107             metaparametros.
108             weights = self.orchestrator.solve_ot_step(self.prev_weights, energy_grads,
109                 tau=self.tau)
110
111             # 3. Prediccion Ponderada
112             # Calcular predicciones individuales para usarlas en el siguiente gradiente
113             current_kernel_preds = jnp.array([k.predict(new_data) for k in self.kernels])
114             final_pred = jnp.dot(weights, current_kernel_preds)
115
116             # Actualizar estado
117             self.prev_weights = weights
118             self.last_pred = final_pred
119             self.last_kernel_preds = current_kernel_preds # Guardar componentes para autograd
120             self.last_loss_type = loss_type # Recordar si estabamos en modo robusto
121
122             return final_pred, loss_type

```

## Capítulo 6

# Meta-Optimización: Walk-Forward y Bayesian Tuning

Implementación de los protocolos de gobernanza para hiperparámetros no diferenciables.

### 6.1 Validación Rolling Walk-Forward

Implementación vectorizada del esquema de validación causal con ventana deslizante.

```
1 class WalkForwardValidator:
2     def __init__(self, model_factory, metric_fn, window_size, horizon, max_memory=None):
3         self.model_factory = model_factory # Funcion: params -> Model
4         self.metric_fn = metric_fn
5         self.window_size = window_size
6         self.horizon = horizon
7         self.max_memory = max_memory # W_max para Rolling Window
8
9     def run(self, data, hyperparams):
10         """
11         Ejecuta el protocolo de validacion sin data leakage, vectorizado con jax.vmap
12         para evaluar múltiples horizontes en paralelo.
13         data: serie temporal completa [T, Features]
14         """
15         # Recolectar todas las ventanas de entrenamiento/validacion en arrays
16         t = self.window_size
17         train_windows = []
18         test_windows = []
19
20         while t + self.horizon + 1 <= len(data):
21             start_idx = 0
22             if self.max_memory is not None:
23                 start_idx = max(0, t - self.max_memory)
24             train_windows.append(data[start_idx:t])
25             test_windows.append(data[t : t + self.horizon + 1])
26             t += self.horizon
27
28         # Vectorizar: procesar todas las ventanas en paralelo usando jax.vmap
29         def compute_error_for_window(train_data, test_data):
30             """Función escalar que procesa una sola ventana de validación."""
31             # 1. Instanciar modelo
32             model = self.model_factory(hyperparams)
33             model.fit(train_data)
34
35             # 2. Extraer inputs/targets de ventana de test
36             input_data = test_data[:-1]
37             target_data = test_data[1:]
38
```

```

39         # 3. Predecir
40         preds = model.predict(input_data)
41
42         # 4. Evaluar error
43         if len(preds) > 0:
44             error = self.metric_fn(preds, target_data)
45         else:
46             error = jnp.inf
47
48         return error
49
50     # Crear función vectorizada (vmap sobre el eje de ventanas)
51     # Nota: En JAX radicalmente compilable, esto permite XLA fusionar operaciones
52     # - Si los modelos son stateless (pytree), vmap compila múltiples horizontes en
53     # paralelo
54     # - Reduce tiempo de validación de  $O(n\_windows)$  a  $O(1)$  en GPU/TPU
55     errors_vectorized = jax.vmap(compute_error_for_window, in_axes=(0, 0))(
56         jnp.array(train_windows),
57         jnp.array(test_windows)
58     )
59     return jnp.mean(errors_vectorized)

```

## 6.2 Optimización Bayesiana con Optuna

Uso del estimador TPE (Tree-structured Parzen Estimator) para buscar heurísticas óptimas.

```

1 import optuna
2
3 def objective(trial):
4     # Definir espacio de búsqueda (AHORA TOTALMENTE AUTO-APRENDIDO)
5     hyperparams = {
6         # Discretización
7         'signature_depth': trial.suggest_int('depth', 3, 5),
8
9         # Regularización
10        'sinkhorn_epsilon': trial.suggest_float('epsilon', 1e-3, 1e-1, log=True),
11        'jko_tau': trial.suggest_float('tau', 0.01, 1.0),
12
13        # Umbrales y Heurísticas Estocásticas (Conectados al Constructor)
14        'cusum_h': trial.suggest_float('cusum_h', 2.0, 5.0), # Umbral de disparo
15        'cusum_slack': trial.suggest_float('cusum_slack', 0.1, 1.0), # Tolerancia a la
16        # deriva
17        'error_alpha': trial.suggest_float('error_alpha', 0.01, 0.2, log=True), # Memoria
18        # de volatilidad
19        'besov_c': trial.suggest_float('besov_c', 1.0, 3.0), # Cono de influencia WTMM
20        'holder_threshold': trial.suggest_float('h_min', 0.3, 0.6)
21    }
22
23    # Validación Causal
24    # Definir factoria: hyperparams -> UniversalPredictor Wrapper
25    def model_factory(hp):
26        # Inyección TOTAL de Hiperparámetros Evolutivos hacia el Constructor
27        model = UniversalPredictor(
28            epsilon=hp['sinkhorn_epsilon'],
29            tau=hp['jko_tau'],
30            holder_threshold=hp['holder_threshold'],
31            signature_depth=hp['signature_depth'],
32            cusum_h=hp['cusum_h'], # <- Conexión restaurada
33            cusum_k=hp['cusum_slack'], # <- Conexión restaurada (Deriva/Slack)
34            error_alpha=hp['error_alpha'], # <- Conexión restaurada (Memoria Volatilidad)
35        )

```

```

33         besov_c=hp['besov_c']           # <- Conexion restaurada (Cono de Besov)
34     )
35     return model
36
37     # Definir metrica robusta (MAE/MSE)
38     def metric_fn(preds, targets):
39         return np.mean(np.abs(preds - targets))
40
41     # Instanciar validador con ventana de 252 dias (trading year) y horizonte 1 dia
42     # Usamos la "fabrica" actualizada que inyecta todos los metaparametros
43     validator = WalkForwardValidator(
44         model_factory=model_factory,
45         metric_fn=metric_fn,
46         window_size=252,
47         horizon=1,
48         max_memory=504
49     )
50
51     # Ejecutar validacion (data debe ser visible en el scope o pasado como argumento
52     # global)
53     mean_error = validator.run(historical_data, hyperparams)
54
55     return mean_error
56
57 def run_meta_optimization(n_trials=50):
58     study = optuna.create_study(direction='minimize')
59     study.optimize(objective, n_trials=n_trials)
60
61     print("Best Personality:", study.best_params)
62     return study.best_params

```

## Capítulo 7

# Sistema de Telemetría y Flags de Estado

Para garantizar la observabilidad completa del sistema en producción, se requiere una estructura de telemetría que exponga métricas críticas y flags de operación. Esta sección implementa la especificación de  $\mathbb{S}_{risk}$  del documento I/O.

### 7.1 Estructura de Telemetría

```
1 from dataclasses import dataclass
2 from typing import Dict
3
4 @dataclass
5 class PredictorTelemetry:
6     """
7     Estructura de telemetria completa del predictor universal.
8     Alineada con la especificacion  $\mathbb{S}_{risk}$  del documento I/O.
9     """
10    # Metricas de singularidad
11    holder_exponent: float          #  $H_t$ 
12    cusum_drift: float              #  $G^+$ 
13    distance_to_alarm: float        #  $h - G^+$ 
14
15    # Metricas avanzadas (nuevas)
16    kurtosis: float                 #  ${}_t$  - Curtosis empirica
17    dgm_entropy: float              #  $H_{DGM}$  - Entropia del predictor DGM
18    adaptive_threshold: float       #  $h_t$  adaptativo
19
20    # Pesos y energia
21    kernel_weights: jnp.ndarray     # (4 elementos)
22    free_energy: float              #  $F []$ 
23
24    # Flags de operacion
25    degraded_inference_mode: bool   # TTL violation
26    emergency_mode: bool            #  $H_t < H_{min}$  (singularidad critica)
27    regime_change_detected: bool    # CUSUM alarm
28    mode_collapse_warning: bool     #  $H_{DGM} < \cdot H[g]$ 
29
30    # Estado del solver
31    sinkhorn_converged: bool
32    loss_type: str                  # 'mse' | 'huber'
33
34 class TelemetryLogger:
35     """
36     Logger de telemetria para monitoreo en produccion.
```

```

37 """
38 def __init__(self, gamma_entropy=0.5, ttl_max_steps=100):
39     self.gamma = gamma_entropy
40     self.ttl_max = ttl_max_steps
41     self.ttl_counter = 0
42     self.mode_collapse_counter = 0
43     self.terminal_entropy_baseline = None
44
45 def log_state(self,
46               holder: float,
47               cusum_g_plus: float,
48               cusum_h: float,
49               kurtosis: float,
50               dgm_entropy: float,
51               weights: jnp.ndarray,
52               free_energy: float,
53               regime_changed: bool,
54               sinkhorn_ok: bool,
55               loss_type: str) -> PredictorTelemetry:
56     """
57     Construye estructura de telemetria a partir del estado interno.
58     """
59     # Calcular flags
60     emergency = holder < 0.4 # H_min threshold
61
62     # TTL counter (simulado - en produccion usar timestamps reales)
63     if regime_changed or emergency:
64         self.ttl_counter = 0 # Reset en eventos criticos
65     else:
66         self.ttl_counter += 1
67
68     degraded = self.ttl_counter > self.ttl_max
69
70     # Mode collapse warning
71     if self.terminal_entropy_baseline is not None:
72         threshold = self.gamma * self.terminal_entropy_baseline
73         mode_collapse = dgm_entropy < threshold
74
75         if mode_collapse:
76             self.mode_collapse_counter += 1
77         else:
78             self.mode_collapse_counter = 0
79
80     # Alarma persistente (>10 pasos consecutivos)
81     mode_collapse_warning = self.mode_collapse_counter > 10
82     else:
83         mode_collapse_warning = False
84
85     # Construir telemetria
86     telemetry = PredictorTelemetry(
87         holder_exponent=holder,
88         cusum_drift=cusum_g_plus,
89         distance_to_alarm=cusum_h - cusum_g_plus,
90         kurtosis=kurtosis,
91         dgm_entropy=dgm_entropy,
92         adaptive_threshold=cusum_h,
93         kernel_weights=weights,
94         free_energy=free_energy,
95         degraded_inference_mode=degraded,
96         emergency_mode=emergency,
97         regime_change_detected=regime_changed,
98         mode_collapse_warning=mode_collapse_warning,
99         sinkhorn_converged=sinkhorn_ok,

```

```

100         loss_type=loss_type
101     )
102
103     return telemetry
104
105     def set_terminal_entropy_baseline(self, H_terminal: float):
106         """
107         Establece baseline de entropia para deteccion de mode collapse.
108         Debe llamarse una vez durante inicializacion con H[g].
109         """
110         self.terminal_entropy_baseline = H_terminal
111
112 # Ejemplo de integracion en PredictionEngine
113 class UniversalPredictorWithTelemetry(UniversalPredictor):
114     """
115     Version extendida del predictor con telemetria completa.
116     """
117     def __init__(self, *args, **kwargs):
118         super().__init__(*args, **kwargs)
119         self.telemetry_logger = TelemetryLogger()
120
121     def step_with_telemetry(self, new_data, previous_target=None):
122         """
123         Version extendida de step() que retorna telemetria.
124         """
125         # Ejecutar prediccion normal
126         pred, loss_type = self.step(new_data, previous_target)
127
128         # Calcular metricas avanzadas
129         # (Asumir que _check_regime_change_with_kurtosis fue usado)
130         kurtosis = getattr(self, 'last_kurtosis', 3.0)
131
132         # Calcular entropia DGM (solo si Rama B activa)
133         if self.prev_weights[1] > 0.05: # rho_B > 5%
134             # En produccion, evaluar entropy sobre batch de puntos
135             dgm_entropy = 1.0 # Placeholder - reemplazar con compute_entropy_dgm()
136         else:
137             dgm_entropy = float('nan') # No aplicable
138
139         # Calcular energia libre (funcional de Wasserstein)
140         free_energy = -jnp.sum(self.prev_weights * jnp.log(self.prev_weights + 1e-10))
141
142         # Loguear estado
143         telemetry = self.telemetry_logger.log_state(
144             holder=float(self.sia.estimate_holder_exponent(self.signal_buffer)[0]),
145             cusum_g_plus=float(self.cusum_state['g_plus']),
146             cusum_h=float(getattr(self, 'last_h_adaptive', self.cusum_state['threshold'])),
147             kurtosis=kurtosis,
148             dgm_entropy=dgm_entropy,
149             weights=self.prev_weights,
150             free_energy=free_energy,
151             regime_changed=bool(self.cusum_state['g_plus'] > self.cusum_state['threshold']),
152             sinkhorn_ok=True, # Placeholder
153             loss_type=loss_type
154         )
155
156     return pred, telemetry

```

### Ejemplo de Uso en Producción:

```

1 # Inicializar predictor con telemetria
2 predictor = UniversalPredictorWithTelemetry(

```



```

3     epsilon=0.01, tau=0.1, holder_threshold=0.45
4 )
5
6 # Establecer baseline de entropia (calculado una vez al inicio)
7 predictor.telemetry_logger.set_terminal_entropy_baseline(H_terminal=2.5)
8
9 # Loop de inferencia
10 for obs in live_market_data:
11     pred, telemetry = predictor.step_with_telemetry(
12         obs.price, previous_target=obs.target
13     )
14
15     # Monitoreo de flags criticos
16     if telemetry.degraded_inference_mode:
17         logger.warning(f"DEGRADED MODE: TTL exceeded {predictor.telemetry_logger.ttl_max}
18             steps")
19         logger.warning("Consider reducing position size - weights are stale")
20
21     if telemetry.emergency_mode:
22         logger.critical(f"EMERGENCY: Singularity detected (H={telemetry.holder_exponent
23             :.3f})")
24         logger.critical("Forcing Kernel D (signatures) with Huber loss")
25
26     if telemetry.mode_collapse_warning:
27         logger.error("MODE COLLAPSE: DGM entropy below threshold")
28         logger.error(f"   H_DGM = {telemetry.dgm_entropy:.3f}")
29         logger.error(f"   Threshold = {predictor.telemetry_logger.gamma * predictor.
30             telemetry_logger.terminal_entropy_baseline:.3f}")
31         logger.error("Action: Reducing rho_B -> 0 until retraining")
32
33     if telemetry.kurtosis > 15.0:
34         logger.warning(f"Crisis regime: kurtosis = {telemetry.kurtosis:.2f}")
35         logger.info(f"Adaptive CUSUM threshold = {telemetry.adaptive_threshold:.4f}")
36
37     # Telemetria para dashboard
38     metrics_buffer.append({
39         'timestamp': obs.timestamp,
40         'prediction': pred,
41         'holder': telemetry.holder_exponent,
42         'kurtosis': telemetry.kurtosis,
43         'cusum_distance': telemetry.distance_to_alarm,
44         'weights': telemetry.kernel_weights.tolist(),
45         'emergency': telemetry.emergency_mode,
46         'degraded': telemetry.degraded_inference_mode
47     })

```

## 7.2 Interpretación de Flags

- **degraded\_inference\_mode**: Se activa cuando el sistema no recibe señales frescas ( $y_{target}$ ) dentro del límite TTL. Los pesos  $\rho$  se congelan y las predicciones continúan pero con confianza degradada. Recuperación con histéresis:  $TTL < 0.8 \cdot \Delta_{max}$ .
- **emergency\_mode**: Singularidad crítica detectada ( $H_t < H_{min}$ ). Fuerza  $w_D \rightarrow 1.0$  (Kernel D de signatures) y activa pérdida de Huber robusta. Indica evento de mercado extremo (flash crash, circuit breaker).
- **regime\_change\_detected**: CUSUM detecta cambio estructural ( $G^+ > h_t$ ). Reinicia entropía a distribución uniforme y resetea acumuladores. Indica cambio de paradigma de mercado.
- **mode\_collapse\_warning**: DGM ha colapsado a solución trivial ( $H_{DGM} < \gamma \cdot H[g]$  durante  $> 10$  pasos). Requiere re-entrenamiento de la red. Mientras tanto, reducir  $\rho_B \rightarrow 0$ .