

# **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
<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) . . . . .	5
<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
<b>4</b>	<b>Módulo 3: Orquestador JKO (Jordan-Kinderlehrer-Otto)</b>	<b>11</b>
4.1	Sinkhorn Estabilizado en Log-Domain con OTT-JAX . . . . .	11
4.2	Integración con Optax para Aprendizaje de Metapámetros . . . . .	12
4.3	Rama C: Esquema IMEX con Solver de Punto Fijo Robusto . . . . .	12
4.4	Rama D: Log-Signatures con Signax . . . . .	13
<b>5</b>	<b>Integración y Pipeline</b>	<b>14</b>
5.1	Clase Maestra PredictionEngine . . . . .	14
<b>6</b>	<b>Meta-Optimización: Walk-Forward y Bayesian Tuning</b>	<b>19</b>
6.1	Validación Rolling Walk-Forward . . . . .	19
6.2	Optimización Bayesiana con Optuna . . . . .	20

# 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.

## 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 -
126 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])

```

## 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  $x_0$  via flujo tangente  $Y_t$ .
46         # Para Delta puro:  $weight \sim \int (\sigma^{-1} Y_t)^T dW_t$  es la formula estandar
47         # simplificada
48         # Pero el tratado exige la formulacion completa que acopla drift y difusion.
49
50         #  $sp = (s\_inv @ y).T$  # Anterior (Incompleto)
51         #  $sp = (s\_inv @ (db\_dx @ y)).T$  # Corregido (Con Drift Sensitivity)
52
53         return (bx, by, bp), (sx, sy, sp)
54
55         # Terminos para SDE Solver
56         # Malliavin requiere esquema fuerte 1.0 (Milstein) si difusion no es constante.
57         # DiffraX no tiene Milstein directo simple para ruido multidimensional general
58         # sin conmutatividad.
59         # Pero podemos usar un esquema Runge-Kutta estocastico de orden fuerte 1.0 o 1.5.
60
61         # Usamos Heun estocastico (Trapezoidal) que converge mas fuerte que Euler
62         # 0 idealmente diffraX.ItoMilstein() si el ruido es escalar o conmutativo.
63         # Para maxima precision general: SRK1 (Strong Order 1.0)
64
65     class CoupledMalliavinTerm(diffraX.AbstractTerm):
66         """
67         Termino personalizado para Ecuaciones Diferenciales Estocasticas Acopladas (Malliavin).
68         Maneja el producto tensorial explicito entre el estado PyTree ((D), (D,D), (D))
69         y el ruido browniano vectorial (D).
70         """
71         def __init__(self, diffusion_fn, brownian_path):
72             self.diffusion = diffusion_fn
73             self.control = brownian_path

```

```

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

```



```
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

```

## 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

```

## 4.2 Integración con Optax para Aprendizaje de Metaparametros

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)

```

```

13
14 # 2. Producto punto a punto (convolucion circular)
15 # kernel_fft debe ser pre-calculado para eficiencia
16 conv_fft = u_fft * kernel_fft
17
18 # 3. Transformada inversa (retornar parte real)
19 convolution_term = jnp.real(jnp.fft.ifft(conv_fft))
20
21 # 4. Compensacion de Levy (Conservacion de Masa)
22 # El termino -(lambda * u) es necesario porque la masa salta FUERA de x.
23 # lambda (intensidad total) es la suma del kernel = kernel_fft[0] (Componente DC)
24
25 lambda_intensity = jnp.real(kernel_fft[0])
26 jump_integral = convolution_term - lambda_intensity * u
27
28 return jump_integral
29
30 def imex_step(x_curr, dt, drift_stiff, jump_kernel_fft, diffusion, key):
31     """
32     Esquema IMEX de 1er orden para PIDE con Saltos (Levy).
33     Parte Implicita: Difusion + Drift Local Stiff
34     Parte Explicita: Integral de Saltos (No-Local) via FFT
35     """
36     noise = random.normal(key, x_curr.shape) * jnp.sqrt(dt)
37
38     # 1. Evaluar termino no-local (integral de salto) explicitamente
39     # drift_nonstiff (Jumps) = lambda * int (u(y) - u(x)) nu(dy) ~ Convolucion
40     jump_term = compute_jump_fft(x_curr, jump_kernel_fft)
41
42     # Parte explicita total
43     explicit_part = x_curr + dt * jump_term + diffusion(x_curr) * noise
44
45     # 2. Solver Implicito para Drift Stiff (Reaccion/Difusion Local)
46     # y - dt*f_I(y) = explicit_part
47     def fixed_point_op(y, _):
48         return explicit_part + dt * drift_stiff(y)
49
50     # Solver robusto (Anderson Acceleration converge mas rapido que Picard simple)
51     solver = jaxopt.AndersonAcceleration(fixed_point_op, maxiter=10, tol=1e-5)
52
53     # Iniciar con x_curr como guess
54     x_new, state = solver.run(x_curr, None)
55
56     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 step(self, new_data, previous_target=None):
146     # 0. Actualizacion del Buffer Circular Estatico (Fix XLA Recompile)
147     # Convertimos new_data a escalar representativo
148     scalar_obs = new_data[0] if hasattr(new_data, '__len__') and len(new_data) > 0
else new_data
149
150     # Desplazamiento ciclico eficiente (Roll)
151     # self.signal_buffer = [x_1, x_2, ..., x_N] -> [x_2, ..., x_N, new_x]
152
153     # Ojo: jnp.roll devuelve un nuevo array (immutable). Debemos reemplazar la
referencia.
154     # Si buffer_idx < max_size, estamos en llenado inicial.
155     # Pero para XLA estatico, siempre mantenemos el array full size.
156
157     self.signal_buffer = jnp.roll(self.signal_buffer, shift=-1)
158     self.signal_buffer = self.signal_buffer.at[-1].set(float(scalar_obs))
159

```

```

160     # Incremento contador de observaciones validas (hasta saturar)
161     new_count = self.buffer_idx + 1
162     self.buffer_idx = min(new_count, self.max_buffer_size) # Clamp al maximo int
standard de python
163
164     # 1. Identificacion (Singularidad Holderiana)
165     # Siempre pasamos el buffer COMPLETO de tamano fijo a la funcion JIT.
166     # WTMM calculara sobre todo el buffer con el parametro de Besov ajustado.
167
168     meta_state_h = self.sia.estimate_holder_exponent(self.signal_buffer, besov_c=self
.besov_c)
169
170     # Logica condicional fuera de JAX puro (o con where) para el warm-up
171     if self.buffer_idx < self.min_buffer_size:
172         meta_state_h = jnp.array([0.5]) # Default browniano durante arranque
173
174     # 1.1 Deteccion de Cambio de Regimen (CUSUM)
175     # Calcular error de la prediccion anterior si existe target
176     last_error = 0.0
177
178     # Inicializar last_pred si no existe (startup)
179     if not hasattr(self, 'last_pred'):
180         self.last_pred = jnp.zeros_like(new_data)
181
182     if previous_target is not None:
183         # Necesitamos haber guardado la prediccion anterior (self.last_pred)
184         # last_pred debe tener la misma forma que target
185         last_error = previous_target - self.last_pred
186     else:
187         # Si no hay target previo (burn-in inicial), el error es cero vector
188         last_error = jnp.zeros_like(new_data)
189
190     # Validacion dimensional: CUSUM internamente reduce a escalar
191     # Retorna un booleano unico (True/False)
192     regime_changed = self._check_regime_change(last_error)
193
194     # 2. Circuit Breaker (Robustez)
195     loss_type = 'mse' # Default
196
197     if jnp.min(meta_state_h) < self.holder_threshold: # Singularidad detectada (Crash
/Salto) con Umbral Dinamico
198         # Forzar peso a signatures (Kernel D) con epsilon de seguridad
199         epsilon = 1e-8
200         weights = jnp.array([epsilon, epsilon, epsilon, 1.0])
201         weights = weights / jnp.sum(weights)
202
203         loss_type = 'huber' # Activar robustez para el siguiente paso
204
205     elif regime_changed:
206         # Reinicio Entropico (Softmax Uniforme)
207         # El cambio estructural invalida la historia de pesos
208         weights = jnp.ones(len(self.kernels)) / len(self.kernels)
209
210         # Reset CUSUM
211         self.cusum_state['g_plus'] = 0.0
212         self.cusum_state['g_minus'] = 0.0
213
214     else:
215         # Calcular gradientes reales basados en el error anterior
216         energy_grads = jnp.zeros(len(self.kernels))
217
218         if previous_target is not None and hasattr(self, 'last_kernel_preds'):
219             # Recuperar volatilidad reciente del error para escalar la robustez

```

```

220         # Esto hace que el parametro delta sea adaptativo y universal (scale-
invariant)
221         current_volatility = jnp.sqrt(self.cusum_state['error_sq_ema'] + 1e-8)
222
223         # Definir funcion de perdida local para JAX autograd
224         def loss_objective(w):
225             pred = jnp.dot(w, self.last_kernel_preds)
226             diff = pred - previous_target
227
228             if self.last_loss_type == 'huber':
229                 # Huber Loss robusta: delta depende de la escala de volatilidad
230                 # Usamos 1.35 * sigma para eficiencia del 95% en distribucion
normal
231                 delta = 1.35 * current_volatility
232
233                 abs_diff = jnp.abs(diff)
234                 is_small = abs_diff <= delta
235                 loss_elements = jnp.where(is_small, 0.5 * diff**2, delta * (
abs_diff - 0.5 * delta))
236                 return jnp.sum(loss_elements) # Retornar Energia Escalar Total
237             else:
238                 return 0.5 * jnp.sum(diff**2) # MSE Escalar Total
239
240             # Obtener gradiente de la Energia (Loss) respecto a los pesos (rho)
241             energy_grads = jax.grad(loss_objective)(self.prev_weights)
242
243             # Resolver flujo JKO con gradientes reales
244             # IMPORTANTE: Inyectar el parametro 'tau' optimizado por Optuna
245             # De lo contrario, se usaria el default (0.1), ignorando el aprendizaje de
metaparametros.
246             weights = self.orchestrator.solve_ot_step(self.prev_weights, energy_grads,
tau=self.tau)
247
248             # 3. Prediccion Ponderada
249             # Calcular predicciones individuales para usarlas en el siguiente gradiente
250             current_kernel_preds = jnp.array([k.predict(new_data) for k in self.kernels])
251             final_pred = jnp.dot(weights, current_kernel_preds)
252
253             # Actualizar estado
254             self.prev_weights = weights
255             self.last_pred = final_pred
256             self.last_kernel_preds = current_kernel_preds # Guardar componentes para autograd
257             self.last_loss_type = loss_type # Recordar si estabamos en modo robusto
258
259             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.
12         data: serie temporal completa [T, Features]
13         """
14         t = self.window_size
15         errors = []
16
17         # Ajuste de Horizonte Estructural:
18         # Para validar H pasos adelante, necesitamos H+1 datos reales.
19         # El dato extra al final es el target real para la ultima prediccion.
20         while t + self.horizon + 1 <= len(data):
21             # 1. Definir ventanas (Rolling si max_memory esta definido)
22             start_idx = 0
23             if self.max_memory is not None:
24                 start_idx = max(0, t - self.max_memory)
25
26             train_data = data[start_idx:t]
27
28             # Extraemos una ventana de prueba extendida (H+1)
29             # test_data incluye: [Obs_t, Obs_t+1, ..., Obs_t+H]
30             # Esto nos permite:
31             # - Alimentar [Obs_t, ..., Obs_t+H-1] al modelo para generar predicciones
32             # - Usar [Obs_t+1, ..., Obs_t+H] como vector de verdad (Ground Truth)
33
34             test_window_full = data[t : t + self.horizon + 1]
35             input_data = test_window_full[:-1] # Lo que ve el modelo
36             target_data = test_window_full[1:] # La realidad futura
37
38             # 2. Instanciar y Entrenar (Reset del estado)
```

```

39     model = self.model_factory(hyperparams)
40
41     # Ejecucion del entrenamiento (Warm-up / Fitting)
42     model.fit(train_data)
43
44     # 3. Inferencia Fuera de Muestra
45     # El modelo recibe N inputs y genera N predicciones one-step-ahead
46     preds = model.predict(input_data)
47
48     # Correction Data Leakage (Off-by-One) SOLUCIONADA:
49     # preds[i] es la prediccion hecha en t+i para t+i+1
50     # target_data[i] es el valor real en t+i+1
51     # Ahora ambos arrays tienen longitud exacta self.horizon (incluso si H=1)
52
53     valid_preds = preds
54     valid_targets = target_data
55
56     if len(valid_preds) > 0:
57         error = self.metric_fn(valid_preds, valid_targets)
58         errors.append(error)
59
60     # 4. Avanzar
61     t += self.horizon
62
63     return np.mean(errors)

```

## 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 busqueda (AHORA TOTALMENTE AUTO-APRENDIDO)
5     hyperparams = {
6         # Discretizacion
7         'signature_depth': trial.suggest_int('depth', 3, 5),
8
9         # Regularizacion
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    # Validacion Causal
24    # Definir factoria: hyperparams -> UniversalPredictor Wrapper
25    def model_factory(hp):
26        # Inyeccion TOTAL de Hiperparametros 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'], # <- Conexion restaurada

```

```

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

```