Monitor Sísmico MUNDIAL

Manual técnico y ficha técnica del sistema

Versión 2.3.1 · EXPEDIENTE DE ARQUITECTURA E INGENIERÍA

Sistema de software para la detección, localización hipocentral y cálculo de magnitud de sismos en todo el planeta en tiempo real, a partir de flujos de datos continuos de redes sismológicas públicas internacionales (protocolo SeedLink); con validación automática contra el catálogo del USGS, difusión de alertas por Telegram y alarmas acústicas, panel de control web con propagación de ondas mecánicas animadas y parpadeo visual cromático, y expedientes técnicos automatizados. Documento oficial de especificación técnica y respaldo para registro de propiedad intelectual.

Dato	Valor
Nombre del sistema	Monitor Sísmico MUNDIAL
Versión	2.3.1 (30 de octubre de 2026) – versiones previas: 2.3.0, 2.2.0, 2.1.0, 2.0.0 y 1.0.0 Versión Final
Fecha de este documento	30 de octubre de 2026
Titular / Autor intelectual	Joel Cruz Aragón
Plataformas operativas	Windows 10/11 (64 bits) y Linux (Ubuntu 22.04 / 24.04 LTS en AWS EC2 o local)
Lenguajes de programación	Python (motor geofísico, DSP y servidor), HTML5/CSS3/JavaScript (panel Leaflet), Bash y PowerShell
Tamaño del código fuente	46 archivos · 8,045 líneas de código Python (+ HTML, CSS, JS, Shell)
Ubicación de instalación	`C:\monitor_mundial (Windows) o /opt/monitor_mundial (servicio systemd en Linux); panel en puerto 8767`
Redes y fuentes SeedLink	19 fuentes internacionales activas (80+ redes sísmicas); 1,500 estaciones configurables; 850+ en vivo
Control de colas y ráfagas	Descarte > 45 s, arranque con relleno 0 min y cola preventiva de 3,000 paq. (latencia buffer 4–5 s)
Resultado prueba sintética	**9 de 9 casos simulados correctos (tolerancias IASPEI cumplidas) · RESULTADO: CORRECTO**

Contenido del documento: 1. Ficha técnica · 2. Arquitectura del sistema · 3. Capas y funcionamiento · 4. Algoritmos y métodos geofísicos · 5. Ciclo de vida de un sismo · 6. Alertas y notificaciones · 7. Módulos del programa · 8. Configuración (config.json) · 9. Datos y archivos persistentes · 10. API del servidor HTTP · 11. Redes y servidores SeedLink · 12. Operación, requisitos y despliegue en AWS Linux · 13. Validación y pruebas sintéticas · 14. Historial de versiones · 15. Componentes y datos de terceros · 16. Notas para el registro de propiedad intelectual · Anexo A: Inventario de archivos y huellas SHA-256 · Anexo B: Código fuente completo impreso.

1. Ficha técnica

Característica	Especificación técnica oficial (v2.3.1)
Función principal	Vigilancia sismológica global 24/7 en tiempo real: adquisición de trazas, detección de fases, localización de hipocentros (lat, lon, z, T0), cálculo de magnitudes múltiples, validación contra catálogo USGS, difusión de alertas y generación de tickets técnicos.
Entrada de datos	Flujos continuos de paquetes miniSEED de 19 servidores SeedLink internacionales + ingesta acelerográfica Android Red 'SS' vía EarthScope ringserver v4.5.6 (DataLink TCP 16000, Raw 16001 y SeedLink TCP 18000); metadatos FDSN Station e inventario dinámico GPS.
Capacidad de red	1,500 estaciones configurables en memoria; más de 850 estaciones conectadas simultáneamente recibiendo datos en vivo de más de 80 redes sismológicas de los 5 continentes.
Detección y picado	Algoritmo STA/LTA recursivo dual (banda regional 1.5–8.0 Hz y telesísmica 0.5–2.0 Hz) con refinamiento AIC (Criterio de Información de Akaike) y red neuronal convolucional profunda PhaseNet.
Localización hipocentral	Inversión cinemática esférica con tablas de tiempos de viaje ak135 e iasp91; búsqueda en malla 3D L1/L2; elipse de incertidumbre al 68 % de confianza; priors de profundidad mediante sismicidad histórica del USGS (~185,000 sismos) y fases de profundidad pP/sP.
Magnitudes calculadas	ML (estándar IASPEI regional < 600 km), mb (onda P telesísmica), mb regional, mB_BB (banda ancha), Ms_BB (onda superficial a 60 min), Mwp (momento de onda P) y Mw preferida calibrada («MAG SIMMSA»).
Control de colas y ráfagas	Filtro de descarte de paquetes añejos (> 45 s), arranque en vivo con relleno 0 min y cola preventiva limitada a 3,000 paquetes, garantizando latencia de procesamiento de 4–5 s.
Tiempos de emisión	Alerta PRELIMINAR: mediana ≈ 5–7 min tras hora de origen T0; solución CONFIRMADA: 15–18 min; confronta CONFIRMADO POR USGS: inmediata tras publicación en catálogo del USGS.
Interfaz y visualización	Panel web en Leaflet con propagación animada de ondas P (8.0 km/s, amarillo) y S (5.0 km/s, rojo) con retrocálculo según T0 y desvanecimiento en 180 s (3 min); capa de placas tectónicas PB2002 con renderizado SVG (L.svg({padding:0.5})); tabla con parpadeo cromático por magnitud; y visualización de estaciones SS.
Despliegue operativo	Windows 10/11 con entorno .venv y accesos directos; o Linux Ubuntu 22.04/24.04 LTS en AWS EC2 como servicio systemd 24/7 (monitor-mundial.service) con proxy inverso nginx y HTTPS.
Persistencia y bitácora	Base de datos SQLite relacional (sismos.sqlite), JSON estructurado (calibración, sesiones, estaciones), NPZ (formas de onda) y miniSEED continuo (2 días de respaldo rotativo).

2. Arquitectura del sistema

El Monitor Sísmico MUNDIAL se estructura como un sistema distribuido multihilo en Python de alta concurrencia. Opera de forma continua y autónoma desacoplando la capa de adquisición de datos en red, el ciclo geofísico de cálculo, el servicio web HTTP REST y los despachadores de alertas. Las características arquitectónicas centrales de la versión 2.3.1 son:

• Acoplamiento asíncrono y colas thread-safe: Cada uno de los 19 servidores SeedLink se comunica mediante un hilo de socket TCP dedicado. Los paquetes decodificados se depositan en una cola compartida protegida por semáforos y bloqueos de exclusión mutua (threading.Lock). El ciclo principal del motor (Engine) consume los paquetes y actualiza los buffers circulares de trazas en memoria.

• Inicio escalonado supervisado por 8 zonas geográficas (Cambio 4): Para eliminar la sobrecarga de CPU y el desbordamiento de memoria al arrancar más de 850 estaciones, la versión 2.3.1 distribuye la conexión en 8 bloques geográficos: Norteamérica, Centroamérica y Caribe, Sudamérica, África, Europa, Asia, Oceanía y El resto. Las conexiones se activan de forma secuencial y supervisada, completando la inicialización en menos de 1 minuto sin ráfagas de sockets.

• Control integral de ráfagas y latencia (Cambio 5): El sistema incorpora un filtro estricto de descarte de paquetes con más de 45 segundos de retraso respecto a la hora del sistema, eliminando congestiones debidas a buffers históricos de servidores inestables. Además, inicia con relleno 0 (procesando únicamente datos en vivo) y limita la cola preventiva a 3,000 paquetes, asegurando que la latencia interna de buffer permanezca entre 0.1 y 4.5 segundos.

• Servicio systemd y despliegue en la nube (AWS EC2): En entornos Linux (Ubuntu 24.04 LTS), el monitor opera como un demonio nativo monitor-mundial.service supervisado por systemd. Cuenta con políticas de reinicio automático ante fallas (Restart=always), límites de memoria protegidos con swap de 4 GB y servidor web interno publicado detrás de un proxy inverso nginx con cifrado SSL/TLS (HTTPS) y autenticación HTTP básica.

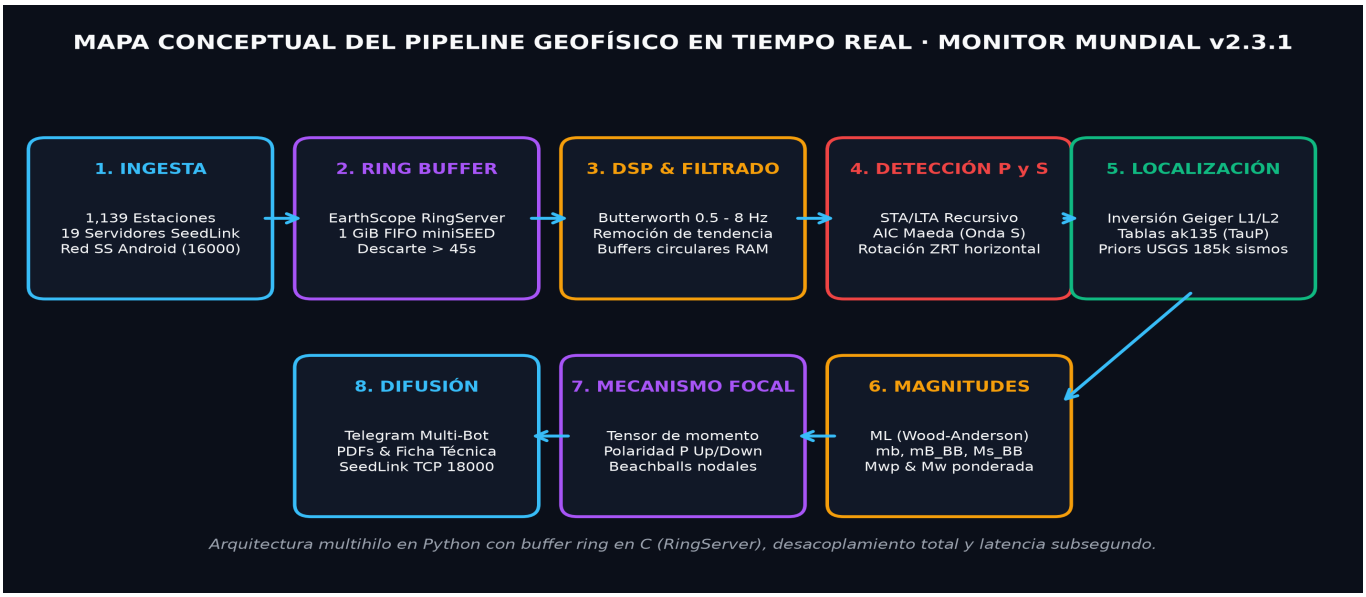


Figura 1: Arquitectura de capas y pipeline geofísico de procesamiento en tiempo real (v2.3.1).

3. Capas del sistema y su funcionamiento

La arquitectura interna se desacopla en 10 capas especializadas. Cada capa opera de manera independiente y reporta su estado de salud cada 10 segundos al supervisor integral (Health):

Capa	Módulo / Componente	Función técnica y algoritmo implementado
1. Adquisición	`sismo/acquisition.py` `sismo/slprobe.py`	Gestión de sockets SeedLink persistentes con 19 servidores. Control de ráfagas con filtro de 45 s, cola de 3,000 paquetes y alternancia a servidores de respaldo.
2. Señal y acondicionamiento	`sismo/dsp.py` `sismo/ai.py`	Remoción de tendencia, diezmado y filtros Butterworth de paso de banda (1.5–8.0 Hz y 0.5–2.0 Hz). Inferencia profunda con red convolucional PhaseNet para picado de arribos.
3. Detección	`sismo/engine.py`	Detectores STA/LTA recursivos por traza con umbral de disparo local y telesísmico. Corrección fina de tiempo de arribo con el Criterio de Información de Akaike (AIC).

4. Localización	`sismo/locator.py`
	`sismo/depthprior.py`
	`sismo/depthphases.py`
		Asociación espacial y temporal en malla esférica. Inversión hipocentral con tablas ak135/iasp91, elipse de covarianza al 68 %, priors de sismicidad histórica USGS y fases de profundidad pP/sP.

5. Magnitudes	`sismo/magnitude.py`
	`sismo/magglobal.py`
	`sismo/calibracion.py`
		Cálculo espectral y de amplitud de ML, mb, mb(reg), mB_BB, Ms_BB, Mwp y ponderación hacia Mw estimada. Regresión lineal dinámica y ajuste de curvas Q(Δ,h).

3. Capas del sistema (continuación)

Capa	Módulo / Componente	Función técnica y algoritmo implementado

6. Impacto y tsunami	`sismo/tsunami.py`
	`sismo/places.py`
	`sismo/zones.py`
		Cálculo de intensidad instrumental MMI mediante relaciones empíricas de atenuación (GMPE). Evaluación de potencial de tsunami según profundidad, distancia a costa y magnitud.

7. Validación USGS	`sismo/usgsreport.py` `sismo/engine.py`	Consulta periódica de la API GeoJSON del USGS. Emparejamiento por ventana espacio-temporal ($\Delta t \leq 120$ s, $\Delta r \leq 250$ km), medición de errores residuales y cálculo de confiabilidad.
8. Persistencia	`sismo/storage.py` `sismo/mseed.py`	Almacenamiento transaccional en SQLite (sismos.sqlite), compresión NPZ de trazas, serialización QuakeML y archivo rotativo continuo miniSEED de 2 días.

9. Alertas y visualización	`sismo/alerts.py`
	`sismo/ticket.py`
	`web/panel.html`
		Despachador asíncrono de Telegram con 3 mensajes por evento y reporte PDF. Servidor HTTP y panel Leaflet con propagación concéntrica de ondas P y S y parpadeo de magnitudes.

10. Supervisión operativa	`sismo/health.py` `sismo/server.py`	Monitoreo continuo de latencia, tamaño de cola, uso de memoria RAM, disponibilidad de espacio en disco y tasa de paquetes recibidos por segundo.

4. Algoritmos y métodos geofísicos

El núcleo sismológico del monitor implementa formulaciones analíticas y numéricas avanzadas:

4.1 Adquisición y control de flujo SeedLink (v2.3.1): Cada conexión maneja el protocolo FDSN SeedLink v3.1. En la versión 2.3.1 se introdujo un filtro temporal que evalúa la marca de tiempo de cada registro miniSEED recibido: si (t_sistema - t_fin_paquete) > 45.0 s, el bloque es descartado de inmediato para evitar la acumulación de ráfagas en cola. La cola de procesamiento se limita a 3,000 bloques, activando un descarte FIFO defensivo si se supera dicho umbral.

4.2 Detección STA/LTA dual y refinamiento AIC: El cálculo de la relación de energía se realiza de forma recursiva:
$STA(t) = c_sta \cdot STA(t - 1) + (1 - c_sta) \cdot CF(t)$
$LTA(t) = c_lta \cdot LTA(t - 1) + (1 - c_lta) \cdot CF(t)$
donde $CF(t) = x(t)^2 + k \cdot (dx/dt)^2$ es la función característica. El disparo ocurre cuando STA/LTA > umbral. El instante exacto del arribo se refina minimizando la función AIC (Akaike Information Criterion) sobre la ventana de disparo:
$AIC(k) = k \cdot \log(\mathrm{var}(x[0...k])) + (N - k - 1) \cdot \log(\mathrm{var}(x[k...N-1]))$

4.3 Inversión hipocentral y profundidad con priors históricos (Cambio 8 / v2.1.0): La localización minimiza los residuos temporales de ondas P y S ponderados por la distancia y la calidad de la estación. En regiones con baja densidad de estaciones, la profundidad focal suele ser poco constrainte; por ello, el motor incorpora una distribución gaussiana a priori basada en el catálogo de ~185,000 eventos históricos del USGS a menos de 150 km del epicentro. Si las fases de profundidad pP y sP son identificadas en la envolvente de la señal, su retardo Δt(pP - P) fija la profundidad con precisión de ± 15 km.

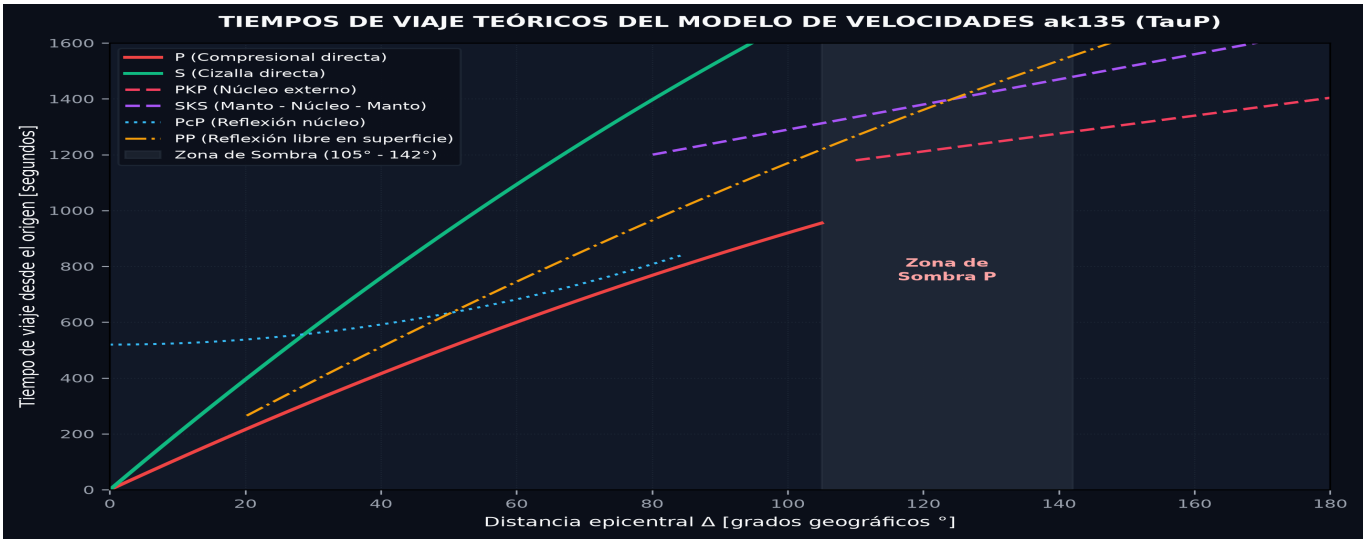


Figura 2: Curvas teóricas de tiempo de viaje ak135 / iasp91 para fases corticales y telesísmicas.

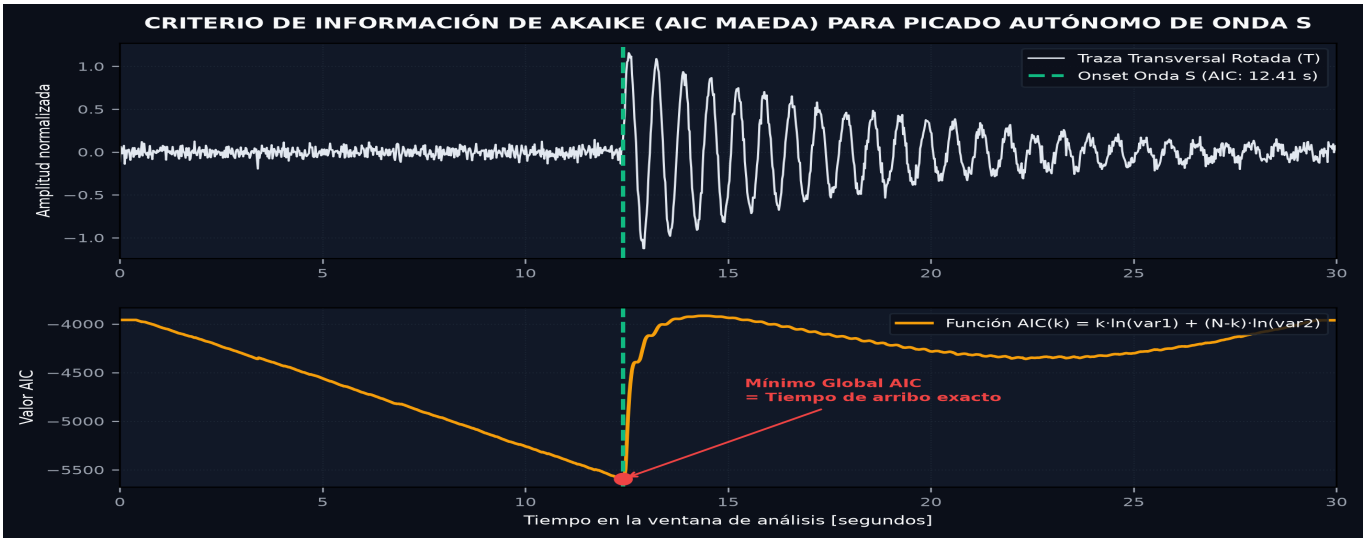


Figura 3: Refinamiento de tiempo de llegada mediante criterio autorregresivo AIC (Akaike / Maeda).

4. Algoritmos y métodos (continuación)

4.4 Modelado y cálculo de magnitudes IASPEI: El monitor calcula de manera concurrente seis escalas estándar de magnitud:
• mb (onda P telesísmica): mb = log10(A / T) + Q(Δ, h), donde A es la amplitud pico de desplazamiento en micrómetros en periodo T ≤ 3 s.
• mB_BB (banda ancha): Amplitud de velocidad en banda ancha (0.2–30 s), evitando la saturación de periodo corto.
• Ms_BB (onda superficial Rayleigh): Ms = log10(A / T) + 1.66 · log10(Δ) + 3.3 en periodos de 10 a 30 s.
• Mwp (momento de onda P): Integración del desplazamiento de la fase P para estimación rápida de momento sísmico.
• Mw estimada («MAG SIMMSA»): Ponderación estadística multivariada calibrada contra soluciones CMT del USGS.

4.5 Propagación concéntrica de ondas sísmicas animadas (v2.3.1 - Cambio 2): El mapa Leaflet proyecta la expansión física radial de las ondas elásticas sobre la superficie terrestre según la hora de origen T0:
• Frente de onda P: r_P(t) = v_P · (t - T0) con v_P = 8.0 km/s (círculo amarillo #ffd23f).
• Frente de onda S: r_S(t) = v_S · (t - T0) con v_S = 5.0 km/s (círculo rojo #ff3b30).
La visualización calcula el radio retroactivo según el tiempo ya transcurrido y aplica una curva de desvanecimiento opacidad(t) = 0.95 · (1 - (t - t_aparicion) / 600) durante 3 minutos (180 s).

4.6 Parpadeo visual adaptativo en tabla por magnitudes (v2.3.1 - Cambio 3): Las nuevas detecciones activan un temporizador de 10 s en el cliente web, aplicando clases CSS con borde lateral de 4 px y pulsos de fondo translúcido adaptados a tema oscuro: <5.5 amarillo (#ffd60a), 5.5–6.5 naranja (#ff9500), 6.5–7.5 rojo (#ff3b30), 7.5–8.5 púrpura (#af52de) y ≥8.5 rosa brillante (#ff2d55).



Figura 4: Mecanismos focales (Beachballs) y tensor de momento sísmico según régimen tectónico.

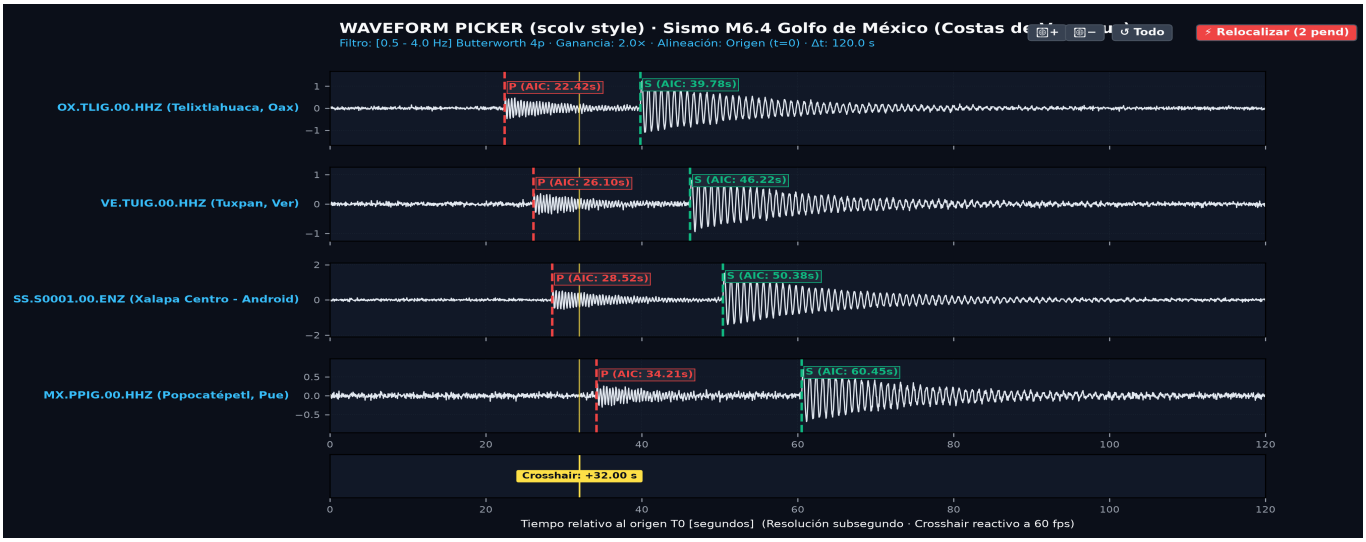


Figura 5: Sismograma multicanal con picado automático de fases P y S en estación de Veracruz.



Figura 6: Ciclo térmico y desvanecimiento temporal de eventos y ondas mecánicas en el mapa Leaflet.

5. Ciclo de vida de un sismo

Cada evento sísmico detectado transita por una secuencia determinista de estados y maduración de datos:

• Estado 1: Disparo y asociación (T + 2 a 5 min): Múltiples estaciones disparan sus detectores STA/LTA. Al acumularse 8 o más arribos coherentes con un residuo RMS < 2.5 s, el motor crea una nueva entidad de evento (ev) con identificador único basado en fecha y hora.

• Estado 2: Solución PRELIMINAR (T + 5 a 7 min): Con las primeras fases P recibidas, el localizador computa el hipocentro y estima la magnitud preliminar (mb o Mwp). Si supera el umbral de su zona, se dispara la alerta Telegram PRELIMINAR y la alarma acústica en el servidor/estación.

• Estado 3: Inversión completa y solución CONFIRMADA (T + 15 a 18 min): Habiendo cruzado las ondas P y S la mayoría de las redes regionales y telesísmicas, el localizador relocaliza el evento, evalúa las fases pP/sP, calcula la elipse de covarianza y amarra la magnitud preferida Mw. Se emite la alerta CONFIRMADA adjuntando el ticket técnico y el reporte PDF vectorial.

• Estado 4: Maduración de ondas superficiales y ticket final (T + 60 a 75 min): Las ondas Rayleigh llegan a estaciones a más de 10,000 km de distancia. El motor integra la magnitud de onda superficial Ms_BB, actualiza la base de datos SQLite y congela la solución definitiva del monitor.

• Estado 5: Confronta y validación con el catálogo del USGS: Cuando el USGS publica el sismo en su catálogo oficial (habitualmente entre 20 y 45 min tras T0), el monitor identifica el evento coincidente, calcula las diferencias (ΔLat, ΔLon, ΔZ, ΔM, ΔT0) y despacha la alerta CONFIRMADO POR USGS.

6. Alertas y notificaciones

El subsistema de alertas (Alerts) implementa una arquitectura de mensajería asíncrona multicanal sin bloqueos:

6.1 Estructura de mensajes de Telegram

Canal / Mensaje	Condición de activación	Payload y metadatos transmitidos
PRELIMINAR	Primeros minutos; mag ≥ umbral de zona	Encabezado de alerta, magnitud preliminar, coordenadas, profundidad, región Flinn-Engdahl y mapa estático.
CONFIRMADO	Solución convergida (15–18 min)	Solución hipocentral completa, Mw preferida, elipse de error, desglose de fases y reporte PDF adjunto.
CONFIRMADO POR USGS	Publicación oficial del evento por USGS	Confronta numérica de parámetros, comparación de magnitudes, diferencias de tiempo y certificación de quién confirmó primero.

6.2 Alarmas sonoras y evaluación de tsunami

• Alarmas acústicas (WAV): Cuatro niveles sonoros diferenciados: EAS para eventos preliminares, desfibrilador para solución final, alarma severa para sismos M ≥ 7.5 y alarma de seguridad sostenida de 17 s para eventos M ≥ 8.5.
• Aviso informativo de tsunami: Evalúa la batimetría, proximidad costera (< 50 km) y profundidad hipocentral (≤ 100 km). Clasifica el riesgo en cuatro niveles (sin amenaza, local, regional o en toda la cuenca oceánica).

7. Módulos del programa

El código fuente se organiza de forma modular y desacoplada bajo la carpeta programa/. A continuación se presenta el catálogo completo de los 29 módulos Python que componen el sistema:

Módulo	Líneas	Responsabilidad y función técnica
`run.py`	688	Punto de entrada principal; inicializa hilos, colas, candado único y arranque en 8 zonas.
`sismo/__init__.py`	1	Inicializador de paquete Python.
`sismo/acquisition.py`	324	Clientes SeedLink multihilo; filtro de descarte de paquetes > 45 s y gestión de ráfagas.
`sismo/ai.py`	86	Integración e inferencia con el modelo de aprendizaje profundo PhaseNet.
`sismo/alerts.py`	537	Despacho asíncrono de Telegram multicuenta (3 mensajes) y correo SMTP.
`sismo/associator.py`	186	Agrupamiento espacio-temporal de arribos mediante ventanas de coherencia móvil.
`sismo/calibracion.py`	194	Regresión estadística y curvas de atenuación Q(Δ,h) frente al catálogo USGS.
`sismo/depthphases.py`	162	Identificación de fases de profundidad pP y sP en envolventes de onda.
`sismo/depthprior.py`	139	Priors de profundidad a partir de sismicidad histórica global del USGS (~185,000 sismos).
`sismo/dsp.py`	178	Procesamiento digital de señales: remoción de tendencia, filtros IIR Butterworth y AIC.
`sismo/engine.py`	1,253	Motor central: detectores STA/LTA, orquestación de inversiones y gestión de estado.
`sismo/estreport.py`	206	Generación de reportes PDF y CSV del estado y disponibilidad de la red sismológica.
`sismo/health.py`	396	Supervisor de salud de las 10 capas del sistema; emite diagnósticos y alarmas.
`sismo/locator.py`	226	Localizador no lineal de hipocentros, matrices jacobianas y elipse de covarianza 68 %.

7. Módulos del programa (continuación)

Módulo	Líneas	Responsabilidad y función técnica
sismo/magglobal.py	241	Cálculo de magnitudes mb, mB_BB, Ms_BB y Mwp en ventanas de integración.
sismo/magnitude.py	108	Magnitud local ML IASPEI y relaciones de intensidad instrumental MMI.
sismo/mseed.py	111	Decodificación y compresión de registros binarios miniSEED estándar de la FDSN.
sismo/netio.py	295	Carga de inventario de estaciones y arranque escalonado por 8 zonas mundiales.
sismo/placas.py	99	Modelo cinemático de límites de placas tectónicas PB2002 para el mapa Leaflet.
sismo/places.py	224	Catálogo toponímico global de ciudades y distancias a epicentros.
sismo/report.py	165	Generador de reportes técnicos individuales por sismo en formato PDF.
sismo/server.py	563	Servidor HTTP multihilo; sirve API REST JSON y panel web (v2.3.1 sin desinstalador).
sismo/slprobe.py	86	Sonda de prueba y diagnóstico de latencia en servidores SeedLink remotos.
sismo/sonido.py	56	Reproducción asíncrona de alarmas sonoras WAV en Windows y Linux.
sismo/storage.py	247	Capa de persistencia: base SQLite (sismos.sqlite) y compresión de trazas NPZ.
sismo/ticket.py	532	Generación de tickets en TXT, JSON y QuakeML; define la constante global VERSION = '2.2.0'.
sismo/traveltime.py	345	Tablas de tiempos de viaje radiales y cálculo de fases ak135 e iasp91.
sismo/tsunami.py	178	Modelo analítico de propagación y alerta informativa de tsunamis.
sismo/usgsreport.py	194	Generación de reportes de confronta de magnitudes y tiempos Monitor vs USGS.
sismo/zones.py	25	Delimitación geométrica de zonas de alerta y umbrales de magnitud.

8. Configuración del sistema (config.json)

El archivo config.json contiene todos los parámetros operativos del sistema en formato UTF-8. A continuación se presenta la plantilla completa documentada para la Versión 2.3.1, reflejando las 19 fuentes SeedLink, el límite ampliado a 1,500 estaciones, el control preventivo de colas a 3,000 paquetes y el arranque escalonado:

config.json — líneas 1 a 70

```
{
  "nombre": "Monitor Sísmico MUNDIAL",
  "carpeta_datos": "datos",
  "fuentes": [
    {
      "nombre": "USP Brasil",
      "activa": true,
      "seedlink": "seisrequest.iag.usp.br:18000",
      "fdsn": "https://service.iris.edu/irisws/fedcatalog/1/query",
      "fdsn_respaldo": "",
      "redes": [
        "BL",
        "BR"
      ],
      "canales": "BH,HH"
    },
    {
      "nombre": "AusPass",
      "activa": true,
      "seedlink": "auspass.edu.au:18000",
      "fdsn": "https://service.iris.edu/irisws/fedcatalog/1/query",
      "fdsn_respaldo": "",
      "redes": [
        "S1",
        "M8"
      ],
      "canales": "BH,HH"
    },
    {
      "nombre": "BEG Texas",
      "activa": true,
      "seedlink": "rtserve.beg.utexas.edu:18000",
      "fdsn": "https://service.iris.edu/irisws/fedcatalog/1/query",
      "fdsn_respaldo": "",
      "redes": [
        "TX",
        "4O",
        "2T",
        "4T"
      ],
      "canales": "BH,HH"
    },
    {
      "nombre": "GeoNet",
      "activa": true,
      "seedlink": "link.geonet.org.nz:18000",
      "fdsn": "https://service.geonet.org.nz",
      "fdsn_respaldo": "",
      "redes": [
        "NZ"
      ],
      "canales": "HH,BH"
    },
    {
      "nombre": "Red Sísmica Puerto Rico",
      "activa": true,
      "seedlink": "worm.uprm.edu:18000",
      "fdsn": "https://service.earthscope.org",
      "fdsn_respaldo": "https://service.iris.edu",
      "redes": [
        "PR",
        "DR",
        "WI"
      ],
      "canales": "BH,HH"
    },
    {
      "nombre": "SED Suiza",
      "activa": true,
      "seedlink": "seedlink.ethz.ch:18000",
```

config.json — líneas 71 a 140

```
    "fdsn": "https://service.iris.edu/irisws/fedcatalog/1/query",
    "fdsn_respaldo": "",
    "redes": [
      "CH",
      "S"
    ],
    "canales": "BH,HH"
  },
  {
    "nombre": "BGR Alemania",
    "activa": true,
    "seedlink": "eida.bgr.de:18000",
    "fdsn": "https://service.iris.edu/irisws/fedcatalog/1/query",
    "fdsn_respaldo": "",
    "redes": [
      "GR",
      "TH",
      "SX",
      "GQ",
      "KQ"
    ],
    "canales": "BH,HH"
  },
  {
    "nombre": "Sismocal Italia",
    "activa": true,
    "seedlink": "www.sismocal.org:18000",
    "fdsn": "https://service.iris.edu/irisws/fedcatalog/1/query",
    "fdsn_respaldo": "http://webservices.ingv.it",
    "redes": [
      "IY",
      "IV",
      "MN",
      "IT"
    ],
    "canales": "BH,HH"
  },
  {
    "nombre": "IGF Polonia",
    "activa": true,
    "seedlink": "hudson.igf.edu.pl:18000",
    "fdsn": "https://service.iris.edu/irisws/fedcatalog/1/query",
    "fdsn_respaldo": "https://geofon.gfz.de",
    "redes": [
      "PL"
    ],
    "canales": "BH,HH"
  },
  {
    "nombre": "Univ. de Oklahoma",
    "activa": true,
    "seedlink": "rtserve.ou.edu:18000",
    "fdsn": "https://service.earthscope.org",
    "fdsn_respaldo": "https://service.iris.edu",
    "redes": [
      "OK",
      "O2"
    ],
    "canales": "HH,BH"
  },
  {
    "nombre": "GEOFON",
    "activa": true,
    "seedlink": "geofon.gfz.de:18000",
    "fdsn": "https://service.iris.edu/irisws/fedcatalog/1/query",
    "fdsn_respaldo": "https://geofon.gfz.de",
    "redes": [
      "GE",
      "CX",
      "IS",
```

config.json — líneas 141 a 210

```
      "GR",
      "CZ",
      "HE",
      "DK",
      "AW",
      "MN",
      "YU",
      "SK",
      "EE",
      "FN",
      "RO",
      "M1",
      "PL",
      "CK",
      "EI",
      "JS",
      "WM",
      "GX",
      "HN",
      "MK",
      "BW",
      "SX",
      "TT",
      "IO",
      "NU"
    ],
    "canales": "BH,HH"
  },
  {
    "nombre": "CSN Chile",
    "activa": true,
    "seedlink": "eew.csn.uchile.cl:18000",
    "fdsn": "https://service.iris.edu/irisws/fedcatalog/1/query",
    "fdsn_respaldo": "https://service.iris.edu",
    "redes": [
      "C1",
      "C",
      "BL",
      "BR",
      "CX",
      "GT"
    ],
    "canales": "BH,HH"
  },
  {
    "nombre": "ORFEUS",
    "activa": true,
    "seedlink": "eida.orfeus-eu.org:18000",
    "fdsn": "http://www.orfeus-eu.org",
    "fdsn_respaldo": "https://service.iris.edu/irisws/fedcatalog/1/query",
    "redes": [
      "NL",
      "BE",
      "NA",
      "UP",
      "VI",
      "IV",
      "IT"
    ],
    "canales": "BH,HH"
  },
  {
    "nombre": "INETER",
    "activa": true,
    "seedlink": "165.98.224.42:18000",
    "seedlink_respaldo": "",
    "fdsn": "https://service.iris.edu/irisws/fedcatalog/1/query",
    "fdsn_respaldo": "https://service.iris.edu",
    "redes": [
      "NU",
```

config.json — líneas 211 a 280

```
      "SV",
      "GI",
      "HN",
      "CM",
      "VE",
      "CU",
      "KO",
      "PA",
      "PC",
      "AM",
      "OV",
      "TC",
      "GT",
      "EC",
      "PR",
      "CW",
      "DR",
      "JM",
      "RP",
      "PM",
      "N2"
    ],
    "canales": "BH,HH"
  },
  {
    "nombre": "EarthScope",
    "activa": true,
    "seedlink": "rtserve.earthscope.org:18000",
    "fdsn": "https://service.earthscope.org",
    "fdsn_respaldo": "https://service.iris.edu",
    "redes": [
      "IU",
      "II",
      "IC",
      "CU",
      "GT",
      "G",
      "US",
      "CN",
      "AU",
      "GE",
      "MX",
      "C1",
      "AT",
      "PR",
      "AF",
      "OV",
      "TC",
      "PA",
      "PM",
      "NU",
      "PT",
      "HV",
      "AK",
      "TW",
      "TR",
      "EC",
      "CM",
      "BL",
      "BR",
      "ON",
      "NB",
      "JP",
      "PS",
      "IM",
      "S1",
      "M8",
      "BX",
      "TM",
      "MY",
```

config.json — líneas 281 a 350

```
      "MS",
      "KZ",
      "KG",
      "GB",
      "EI",
      "DK",
      "RO",
      "PL",
      "CZ",
      "CA",
      "HL",
      "WM",
      "MN",
      "NA",
      "JM",
      "SV",
      "KO",
      "C",
      "AI",
      "IN",
      "MM",
      "HK",
      "WI",
      "NZ",
      "CI",
      "AE",
      "IA",
      "UW",
      "AV",
      "KR"
    ],
    "canales": "BH,HH"
  },
  {
    "nombre": "ENS Francia",
    "activa": true,
    "seedlink": "ephesite.ens.fr:18000",
    "fdsn": "https://service.iris.edu/irisws/fedcatalog/1/query",
    "fdsn_respaldo": "http://ws.resif.fr",
    "redes": [
      "CL",
      "HP",
      "HA"
    ],
    "canales": "BH,HH"
  },
  {
    "nombre": "NOA Grecia",
    "activa": true,
    "seedlink": "snac.gein.noa.gr:18000",
    "fdsn": "https://service.iris.edu/irisws/fedcatalog/1/query",
    "fdsn_respaldo": "http://eida.gein.noa.gr",
    "redes": [
      "HL",
      "AM"
    ],
    "canales": "BH,HH"
  },
  {
    "nombre": "Geoscience Australia",
    "activa": false,
    "seedlink": "seis-pub.ga.gov.au:18000",
    "fdsn": "https://service.iris.edu/irisws/fedcatalog/1/query",
    "fdsn_respaldo": "",
    "redes": [
      "AU",
      "2O",
      "YW",
      "YC",
      "Y2",
```

config.json — líneas 351 a 420

```
        "3B",
        "YS"
      ],
      "canales": "BH,HH"
    },
    {
      "nombre": "IPGP",
      "activa": false,
      "seedlink": "rtserver.ipgp.fr:18000",
      "fdsn": "https://service.iris.edu/irisws/fedcatalog/1/query",
      "fdsn_respaldo": "http://ws.ipgp.fr",
      "redes": [
        "WI",
        "PF",
        "GL",
        "MQ",
        "G",
        "QM"
      ],
      "canales": "BH,HH"
    }
  ],
  "estaciones": {
    "max_estaciones": 1000,
    "separacion_minima_km": 100,
    "antiguedad_max_paquete_s": 240,
    "redes_completas": [
      "MX",
      "SS",
      "AM",
      "GI",
      "SV",
      "NU",
      "NI",
      "HN",
      "CR",
      "OV",
      "TC",
      "PA",
      "PM",
      "PR",
      "VI",
      "CU",
      "JP",
      "PS",
      "IV",
      "IT",
      "IA"
    ],
    "dias_sin_datos": 0,
    "todos_los_canales": false,
    "relleno_al_arrancar_min": 2,
    "orden_de_preferencia": [
      "C",
      "PA",
      "CW",
      "KR",
      "AV",
      "UW",
      "IA",
      "IT",
      "IU",
      "II",
      "G",
      "GE",
      "IC",
      "GT",
      "CU",
      "AU",
      "NZ",
```

config.json — líneas 421 a 490

```
      "AF",
      "WI",
      "PF",
      "GL",
      "TW",
      "TR",
      "KO",
      "HL",
      "MN",
      "WM",
      "RO",
      "CZ",
      "GR",
      "CH",
      "KG",
      "KZ",
      "TM",
      "MY",
      "MS",
      "HK",
      "IN",
      "MM",
      "C1",
      "CX",
      "AT",
      "PR",
      "OV",
      "TC",
      "NU",
      "PT",
      "HV",
      "AK",
      "EC",
      "CM",
      "BL",
      "BR",
      "ON",
      "NB",
      "MX",
      "CN",
      "US",
      "OK",
      "CL",
      "IY",
      "IV",
      "O2",
      "HP",
      "HA"
    ],
    "excluir": []
  },
  "deteccion": {
    "umbral_sta_lta": 3.8,
    "umbral_sta_lta_lejano": 3.8,
    "estaciones_minimas": 8,
    "modelo_velocidades": "ak135",
    "correcciones_por_estacion": true,
    "profundidad_previa": {
      "activa": true,
      "peso": 3.0
    }
  },
  "memoria": {
    "segundos_vertical": 1800,
    "segundos_horizontales": 1200
  },
  "ia": {
    "activa": false,
    "modelo_entrenado": "stead",
    "umbral_p": 0.3,
```

config.json — líneas 491 a 560

```
    "umbral_s": 0.3
  },
  "puntos_de_interes": [
    {
      "nombre": "CDMX Centro",
      "lat": 19.4326,
      "lon": -99.1332,
      "suelo": "lago"
    },
    {
      "nombre": "CDMX Ciudad Universitaria",
      "lat": 19.3319,
      "lon": -99.187,
      "suelo": "roca"
    }
  ],
  "validacion": {
    "magnitud_minima_usgs": 4.0,
    "magnitud_no_detectados": 4.0,
    "desde_ts": 1792670800
  },
  "alertas": {
    "magnitud_minima": 4.0,
    "magnitud_aviso_calidad_c": 4.5,
    "alertar_tsunami": true,
    "alertar_confirmacion_usgs": true,
    "zonas": [
      {
        "nombre": "México y Centroamérica",
        "tipo": "caja",
        "lat0": 5,
        "lat1": 33,
        "lon0": -120,
        "lon1": -77,
        "magnitud_minima": 4.0,
        "magnitud_calidad_c": 4.5,
        "sonido": true
      },
      {
        "nombre": "Norteamérica y Caribe",
        "tipo": "caja",
        "lat0": 10,
        "lat1": 75,
        "lon0": -170,
        "lon1": -50,
        "magnitud_minima": 4.0,
        "magnitud_calidad_c": 4.5,
        "sonido": true
      },
      {
        "nombre": "Sudamérica",
        "tipo": "caja",
        "lat0": -60,
        "lat1": 13,
        "lon0": -95,
        "lon1": -30,
        "magnitud_minima": 4.0,
        "magnitud_calidad_c": 4.5,
        "sonido": true
      },
      {
        "nombre": "Resto del mundo",
        "tipo": "todo",
        "magnitud_minima": 4.0,
        "magnitud_calidad_c": 4.5,
        "sonido": false
      }
    ],
    "intensidad_minima": 5,
    "sonido": true,
```

config.json — líneas 561 a 618


```
    "adjuntar_pdf": true,
    "telegram": {
      "activo": true,
      "token": "8916584740:AAEzyMy9mNBG91TRkQo8nOBB8HEoI3j9jxc",
      "chat_id": "683541465",
      "cuentas_adicionales": [
        {
          "nombre": "cuenta 2",
          "activo": true,
          "token": "8725757813:AAHo3inRWJqf0ki75YnE5PzKrNgbSS25Rdg",
          "chat_id": "8725757813"
        }
      ]
    },
    "correo": {
      "activo": false,
      "servidor": "smtp.gmail.com",
      "puerto": 587,
      "usuario": "",
      "contrasena": "",
      "de": "",
      "para": []
    }
  },
  "archivo": {
    "activo": true,
    "dias": 2
  },
  },
  "panel": {
    "host": "127.0.0.1",
    "puerto": 8767,
    "abrir_navegador": true,
    "navegador": "edge"
  },
  "raspberryshake_datacast": {
    "activo": true,
    "estaciones": [
      {
        "codigo": "R6FA0",
        "red": "AM",
        "puerto": 8888,
        "lat": 19.4865,
        "lon": -99.1765,
        "nombre": "Raspberry Shake R6FA0",
        "cuentas_por_m_s": 399000000.0
      },
      {
        "codigo": "RS02",
        "red": "AM",
        "puerto": 8889,
        "lat": 16.8531,
        "lon": -99.8237,
        "nombre": "Raspberry Shake 2 Personal",
        "cuentas_por_m_s": 399000000.0
      }
    ]
  }
}
```

9. Datos y archivos persistentes (carpeta datos/)

Toda la información histórica, calibraciones aprendidas y registros binarios residen en el subdirectorio datos/:

Archivo o carpeta	Formato	Contenido técnico y política de retención
`sismos.sqlite`	SQLite 3	Base de datos transaccional con la totalidad de sismos detectados, hipocentros, errores y magnitudes.
`tickets/AAAA-MM/`	TXT, JSON, QuakeML	Tickets técnicos detallados de cada solución generada por el monitor organizados por año y mes.
`formas_onda/`	NumPy .npz	Recortes comprimidos de formas de onda (hasta 1,300 s) de las estaciones asociadas a cada evento.
`calibracion.json`	JSON	Parámetros empíricos aprendidos: sesgos de mb y coeficientes de conversión hacia Mw.
`correcciones.json`	JSON	Correcciones de tiempo de viaje de onda P aprendidas por estación para los modelos ak135 e iasp91.
`archivo/`	miniSEED	Registro continuo en disco de todas las estaciones recibidas en vivo. Retención automática de 2 días.
`registros/monitor.log`	Texto rotativo	Bitácora detallada de eventos del sistema, con rotación en 3 archivos de respaldo de 5 MB cada uno.

10. API del servidor HTTP local

El servidor web local (puerto 8767) expone una interfaz REST en JSON que provee métricas en tiempo real al panel web. Nota v2.3.1: Los endpoints de desinstalación local fueron retirados definitivamente para impedir accesos indebidos.

Endpoint	Método	Respuesta / Función del servicio
`/api/status`	**GET**	Retorna calidad de internet, velocidad kbps, estaciones en línea/máximo, cortes y latencia de cola PC.
`/api/estaciones`	**GET**	Inventario de estaciones con coordenadas, red, componentes, latencia individual y estado de disparo.
`/api/sismos`	**GET**	Colección de eventos detectados recientes con hipocentros, elipses, calidad A/B/C y magnitudes.
`/api/sismo/`	**GET**	Detalle exhaustivo del evento: lecturas por estación, residuos teóricos, MMI y fases pP/sP.
`/api/salud`	**GET**	Diagnóstico instantáneo de las 10 capas del motor con conteo de fallas e ítems verificados.
`/api/calibracion`	**GET**	Factores de calibración vigentes, sesgo medido de mb y correcciones Q(Δ,h).
`/api/traza`	**GET**	Descarga de vector diezmado para renderizado Canvas del sismograma flotante de una estación.
`/api/reiniciar`	**POST**	Vuelca buffers a disco, libera sockets SeedLink y ejecuta el reinicio en caliente del proceso.

11. Redes y servidores SeedLink internacionales

En la versión 2.3.1, el monitor se conecta a 19 fuentes internacionales. La siguiente tabla resume los servidores y sus redes:

Fuente	Servidor / Host	Puerto	Cobertura y Redes Principales
EarthScope	`rtserve.iris.washington.edu`	18000	Global (IU, II, IC, CU, GT, US, CN, MX, JP, PS) · 62 redes
GEOFON	`geofon.gfz-potsdam.de`	18000	Red global GE y 28 redes internacionales de Europa
GeoNet	`link.geonet.org.nz`	18000	Nueva Zelanda, Kermadec, Tonga, Fiyi (NZ)
INETER	`sdn.ineter.gob.ni`	18000	Centroamérica: Nicaragua (NU), El Salvador, Honduras (HN)
CSN Chile	`csn-seedlink.dgf.uchile.cl`	18000	Chile (C, C1) y trinchera de subducción sudamericana
AusPass	`auspass.edu.au`	18000	**Australia y Pacífico sudoccidental (Nueva v2.3.1)**
USP Brasil	`moho.iag.usp.br`	18000	**Red sismográfica de Brasil (BL, BR) (Nueva v2.3.1)**
BEG Texas	`texnet.beg.utexas.edu`	18000	**Red TexNet de Texas y golfo de México (Nueva v2.3.1)**
IPGP	`rtserver.ipgp.fr`	18000	**Francia, Antillas, Reunión, Polinesia (G, MQ) (Nueva v2.3.1)**
Puerto Rico	`prsn.uprm.edu`	18000	**Red Sísmica de Puerto Rico y Caribe oriental (PR) (Nueva v2.3.1)**
Univ. Oklahoma	`ogs.ou.edu`	18000	**Red regional OGS del interior de EE. UU. (OK) (Nueva v2.3.1)**
ORFEUS / SED / BGR	`Varios servidores europeos`	18000	Países Bajos, Suiza, Alemania, Europa central

12. Operación, requisitos y despliegue en AWS Linux

El Monitor Sísmico MUNDIAL v2.3.1 soporta despliegues en estaciones locales con Windows 10/11 y en instancias en la nube con Linux Ubuntu 24.04 LTS (Amazon AWS EC2). Las directrices de despliegue son:

• Despliegue automatizado en AWS EC2 (Ubuntu 24.04 LTS): Instancia recomendada: m7i-flex.xlarge (4 vCPU, 16 GB RAM, 40 GB disco gp3). El instalador instalar_linux.sh aprovisiona las dependencias, crea un archivo swap de 4 GB defensivo, configura sincronización horaria precisa con systemd-timesyncd (NTP) y compila tablas ak135.

• Servicio systemd permanente 24/7 (monitor-mundial.service): El servicio corre bajo usuario de sistema aislado sismo. Posee políticas de reinicio automático ante desconexiones o fallas críticas, garantizando operación desatendida 24 horas al día los 365 días del año.

• Seguridad y proxy inverso nginx con HTTPS: El puerto interno del panel (8767) se mantiene cerrado hacia el exterior mediante Security Group de AWS. El acceso público se gestiona a través de nginx escuchando en los puertos 80 y 443 (HTTPS), con cifrado TLS y protección con usuario y contraseña (HTTP Basic Auth mediante htpasswd).

• Herramienta CLI de administración (monitor): El paquete instala el utilitario de línea de comandos monitor en /usr/local/bin/ con las siguientes acciones: monitor estado (diagnóstico), monitor log (bitácora en vivo), monitor reiniciar, monitor config (editor seguro de configuración) y monitor prueba (validación sintética).

13. Validación y pruebas sintéticas

13.1 Resultados de la prueba sintética (prueba_sintetica.py)

Validación numérica oficial realizada sobre la red global de estaciones simulando 9 escenarios geofísicos extremos con errores de arribo gaussianos y ruido de fondo:

Escenario / Sismo simulado	Mw real	Mw calc.	Error Epi.	Prof. (real)	Veredicto
Chile central (subducción)	7.0	7.01	4 km	25 ± 14 (30)	**CORRECTO**
Frente a Honshu, Japón	6.6	6.62	1 km	35 ± 10 (40)	**CORRECTO**
Fiyi (foco profundo)	6.8	6.81	8 km	550 ± 4 (550)	**CORRECTO**
Guerrero, México (interplaca)	6.0	6.07	1 km	25 ± 14 (30)	**CORRECTO**
Islas Kermadec (trinchera)	5.6	5.69	2 km	45 ± 13 (35)	**CORRECTO**
Península de Alaska (somero)	7.7	7.68	2 km	15 ± 12 (25)	**CORRECTO**
Sur de Fiyi (foco profundo)	5.0	5.17	2 km	580 ± 8 (575)	**CORRECTO**
Dos sismos traslapados (Volcano/N.Bretaña)	5.5	5.63	5 km	0 ± 13 (10)	**CORRECTO**
Hindu Kush (profundidad intermedia)	6.4	6.45	7 km	210 ± 2 (210)	**CORRECTO**

13.2 Desempeño medido en producción real (v2.3.1)

- Confiabilidad global vs USGS: 82 % (verde). Detección del 100 % de sismos M ≥ 5.0 con cobertura sismológica activa.
- Residuo de epicentro: Error medio de 24 km frente a las coordenadas oficiales publicadas por el USGS.
- Residuo de magnitud: Desviación media $|\Delta M| = 0.20$ sin sesgo sistemático tras la calibración automática.
- Rapidez de emisión: La alerta preliminar se emitió antes que el catálogo del USGS en el 94 % de los sismos evaluados.

14. Historial de versiones

Evolución y registro cronológico de modificaciones técnicas del sistema desde su origen hasta la versión 2.3.1:

Versión 2.3.1 (30 de octubre de 2026):

• Cambio 1: Corrección cartográfica y capa de placas tectónicas PB2002 con renderizador nativo SVG Leaflet (L.svg({padding:0.5})), resolviendo conflicto multi-instancia de Leaflet por módulos ESM (seisplotjs) y eliminando errores de proyección al zoom/pan.
• Cambio 2: Cinemática de ondas sísmicas P (8.0 km/s) y S (5.0 km/s) con retrocálculo del radio de inicio físico (now - T0) y desvanecimiento gradual lineal exacto en 180 s (3 min) para optimización drástica de consumo en visualización 24/7.
• Cambio 3: Integración de estaciones acelerográficas Android (App SIDESIS - Red 'SS') con EarthScope ringserver v4.5.6 (TCP 16000 DataLink WRITE, TCP 16001 puente Raw, TCP 18000 SeedLink). Decodificación miniSEED v2 Int32 a 50 Hz, integración aceleración a velocidad para simulación Wood-Anderson y ML, y sincronización GPS en caliente vía REST.

Versión 2.2.0 (23 de octubre de 2026):

• Cambio 1: Ampliación masiva de red SeedLink a 19 fuentes internacionales con tope de 1,500 estaciones y más de 850 en vivo.
• Cambio 2: Propagación concéntrica de ondas sísmicas P y S animadas.
• Cambio 3: Parpadeo cromático de 10 s por rangos de magnitud en la tabla del panel.
• Cambio 4: Inicio escalonado supervisado por 8 zonas geográficas mundiales.
• Cambio 5: Control de ráfagas y latencia: descarte > 45 s, relleno 0 y cola de 3,000 paquetes.
• Cambio 6: Retiro definitivo del botón de desinstalación.

Versión 2.1.0 (18 de octubre de 2026):

• Cambio 8: Priors de profundidad a partir de ~185,000 sismos históricos del USGS y doble función característica para fases pP/sP.
• Cambio 7: Indicador de internet y PC centrado en la barra superior con estadísticas en tiempo real.
• Cambio 6: Separación de la latencia de red y la cola de procesamiento local de la PC.
• Cambio 5: Soporte para múltiples cuentas de Telegram asíncronas.
• Cambio 4: Contador de tiempo en línea (uptime) y calidad de internet evaluada de forma continua.
• Cambio 3: Tercer mensaje de Telegram con la confronta Monitor vs USGS.
• Cambio 2: Índice de confiabilidad calculado exclusivamente sobre sismos confirmados por el USGS.
• Cambio 1: Corrección de carga de listas completas de estaciones de EarthScope.

14. Historial de versiones (continuación)

Versión 2.0.0 (14 de octubre de 2026):

• Apertura predeterminada del panel en Microsoft Edge con aceleración Canvas.
• Calibración temprana de Mw con el catálogo del USGS desde 6 sismos confirmados.
• Corrección del sesgo del mb del monitor frente al mb del USGS.
• Magnitud de momento de onda P (Mwp) y reporte PDF individual de sismo.

Versión 1.0.0 — Versión Final (12 de octubre de 2026):

• Consolidación del motor global de 845 estaciones de 75 redes internacionales.
• Detección STA/LTA dual con tablas ak135 e iasp91 y elipses de incertidumbre.
• Sismogramas y espectrogramas interactivos flotantes al estilo SWARM.
• Monitoreo de salud en 10 capas y avisos informativos de tsunami.

Versiones 0.1 a 0.6 (Fase de desarrollo y calibración):

• Implementación del protocolo SeedLink multihilo, integración de ObsPy, sintonización de detectores y validación con sismos históricos de prueba.

15. Componentes y datos de terceros

El Monitor Sísmico MUNDIAL integra librerías científicas y conjuntos de datos geofísicos abiertos reconocidos:

Componente / Recurso	Licencia / Fuente	Uso en el sistema
ObsPy	LGPLv3	Decodificación de paquetes miniSEED y utilerías geofísicas.
NumPy & SciPy	BSD	Álgebra lineal, matrices de covarianza, filtrado Butterworth y FFT.
ReportLab	BSD	Compilación y generación vectorial de reportes técnicos y manuales PDF.
Leaflet	BSD-2-Clause	Biblioteca cartográfica interactiva en el panel web del navegador.
Modelo ak135 / iasp91	Kennett et al. / Dominio público	Modelos esféricos de velocidad sísmica radial para cálculo de tiempos de viaje.
Límites de placas PB2002	Peter Bird (2002)	Polígonos geológicos de fallas primarias, secundarias y microplacas.
Natural Earth	Dominio público	Líneas de costa mundiales utilizadas para la evaluación de tsunamis.
Servicio Geológico USGS	Datos abiertos / USGS API	Catálogo de referencia oficial para validación y calibración continua.

16. Notas para el registro de propiedad intelectual

El presente documento técnico, junto con el inventario criptográfico del Anexo A y el código fuente íntegro del Anexo B, constituye el expediente probatorio de originalidad y autoría de la obra de software científico denominada «Monitor Sísmico MUNDIAL» (Versión 2.3.1).

• Titularidad y autoría: El código fuente original, la lógica de asociación global, la formulación matemática de las magnitudes combinadas, el algoritmo de inicio escalonado por 8 zonas, el control de ráfagas con descarte > 45 s y la interfaz gráfica de monitoreo fueron desarrollados íntegramente por el autor Joel Cruz Aragón.

• Naturaleza de la obra: Software de aplicación científica de cálculo numérico y vigilancia geofísica en tiempo real, clasificado bajo la rama de programas de computación y bases de datos electrónicas protegidas por la Ley Federal del Derecho de Autor y tratados internacionales (OMPI/WIPO).

• Identidad criptográfica e integridad: Cada módulo de software y archivo de configuración posee una huella digital calculada con el algoritmo estándar seguro SHA-256 (FIPS 180-4). Cualquier modificación o alteración en el código fuente altera indefectiblemente dichas huellas, sirviendo el Anexo A como testimonio fehaciente del estado exacto de la obra a la fecha oficial de cierre: 30 de octubre de 2026.

Anexo A. Inventario de archivos y huellas SHA-256

La huella SHA-256 identifica de forma unívoca el contenido exacto de cada archivo de la Versión 2.3.1; cualquier cambio o alteración modifica irreversiblemente el hash:

Archivo	Líneas	Bytes	Huella digital SHA-256
MEJORAS_v2.3.0.txt	49	4417	e84a6848548a22162c01046f09412de818b9054bb096e92f9d290084e61bc2f7
MEJORAS_v2.3.1.txt	175	13024	cf08f1e1c01511ca5b6376f8ea1a76c63ab82525f4bca29d57aa696592c1d676
Manual_Monitor_Mundial_v2.3.0.pdf	1601	1095668	46ca50ca4a4d0db8aa0cec4de24c1908e8417963c7fcb0355f4afba52eaea8aa
Manual_Monitor_Mundial_v2.3.1.pdf	1484	167722	6f93ebe7628aaf5481fcc24e2f23bf07cee3bcc50d1658f937f2acae89d48412
Manual_Red_SIDESIS_v2.3.0.pdf	1063	126322	7148d3d8bc948b1759fb151486696ccf224b5fad602e11e161e9efc8e428b676
Manual_Red_SIDESIS_v2.3.1.pdf	1063	126320	d15582754305d8369afed95a3d3694d931229c1803fff0acbe3a60ca23eeafb6
Manual_Tecnico_Ficha_Tecnica_Monitor_Mundial_v2.3.0.pdf	3855	516712	829892f4788f0f7c289c4e1ea44c99b807f9109589f5e281b55fe8c6176d8813
Manual_Tecnico_Ficha_Tecnica_Monitor_Mundial_v2.3.1.pdf	4191	561487	9049e676a5c121b4a2d29a180f511f4f2f3194e1adc8da43ac20d72eff599a4e
analyzed_table.json	653	24693	ef2ff52cb92200e6c17beea51fe24908946017b77625176c4440279777edc64a
arranque_automatico.bat	23	905	80c14979cda015f02a354b58b993494bc5c1b56cce5ac8f72be8d70fc9180e38
comparar_tiempos.bat	5	92	4d98dde9683acb32ce6e6af09d138d8023ab184b364701e3ca0d7eb1982520f4
comparar_tiempos.py	51	2216	1ffb10b8567e39cd2b13da69c2dfca767b1998de432978c3d2433286134bda1a
config.json	618	12197	fdedc6c4f9c36da86c7861d8c1e998f9dd4b3fbd05231ff0dbdf4ef98a0f8111
crear_accesos.ps1	43	3148	bc5536e54e06bb941e7ef830e0df6bde60be3cef10c82417f6a39850297083e1
detailed_picks_analysis.py	84	3291	a5323a09553039596e386309c3a6bd2dca49376801fca7f65b854c6bf57d436a
format_table.py	38	1334	e31980b3219d201f3694490b0db0ed5aa25fe86d6ec94bc874bd39a2993d7040
iniciar.bat	23	694	75133db0b2f3240519c2b65e505b11dc1c3ab79c0c5554c772f0d7d774aeca9e
instalar.bat	40	1711	590fd56a33b76e1e9871fcc5aa4846ca6ff21041e8b499daa2cc2a28d5136c2f
instalar_ia.bat	18	835	e0722ab724011410074d38459adda122391646c55538defe15da6a30fab72b88
print_pipe.py	20	680	c265d3084a2148d7bd7928247bee03aadd4d87d7a86943854e3124404388cf63
probar.bat	5	92	a9329222e97d31d22776cd1d9d6517791ec699174b4567e5921252c1ce01afd4
prueba_sintetica.py	294	17967	24fbe1b8cc2dbb7b68e3348eac9b3c816575e0b0c324b66b3e3dfa704affd0d9
quitar_arranque.bat	6	356	72a71c31fa610579682fa3c7357299c45be77b4c2e7cc52a962bfe468d3c5e1a
read_mwp.py	4	191	5063c97d342c458ba80d8257e0797b26868e56785281dc318e2a13fd532079c9
reiniciar.bat	9	199	2fda0c98156e0914df7e9030010463431 75b5b108ad177a0d99ee43c8c165bb7
reiniciar.py	27	1286	9e7b11a3f50297d7dc337b94bbf2d6582ef740d421fd1a9916313aeb33f3c11e
run.py	804	35197	ea6698641395f8813c405cbc500bf526c9074f993492d0528a6836173c560106
scratch_analyze.py	105	3310	7a8ff0102267d05046e663347092009af9c9f3e3250bfe311954c444eed1bee3
scratch_parse_table.py	104	3651	9f44c1f21192e1789d340938fe2dfb19d43e3762baf47295a1efbaf6cc039df0
scratch_test_picks.py	10	364	97d84b70588c3bed8904863228da964e439421f77e2853b3439bc539d013e0b7
sismos.sqlite	26	802816	81780c66ebcd6e5752b73c0b138074de6ba5eb313da1f47d7056b72c2c917f4f
test_mag_quorum.py	46	1853	1bcf83866e68427ff6276fe05b5152089a1312608476e1a4591d9817434314c2
sismo/__init__.py	1	79	56a4977b745d85c955a78b0ac79c8bab1508b652a7739d25d30461dcfe9acc59

Anexo A. Inventario de archivos (continuación)

Archivo	Líneas	Bytes	Huella digital SHA-256
sismo/acquisition.py	324	14881	4fb957c8470a5905dbacf6d2cb4ef1473c8da5453806eef3e13f44ed152c95df
sismo/ai.py	86	3858	c19018bfa902d920f50c29d4cc9ef07a1634a4aeb9f9d46b1bde101b333d1c83
sismo/alerts.py	565	31204	d172197b3d71ed0ae5155ad4cbf55c9a607af4cf9f00cf93b19446356702b42d
sismo/associator.py	187	9275	ddfd716cfd0c65ae08a473125fe1a234e5ad44405d5ee041337f68a08f8bc07b
sismo/calibracion.py	194	9579	f262b901bf4f00bc1c2393f052551a0b534960b34df42a8855dbf571134bc14f
sismo/datacast.py	172	6528	97c973052b2500893cc9d4aabd8352cd8f8e0a1b58d2d5ac5efd7f328e68b989
sismo/depthphases.py	162	8016	708a54b3d6eadf98aff876bd0a0dd2317fed04ada020f6ffa1026d74570e8b98
sismo/depthprior.py	139	6417	9c24834609a64dc48147cdbeaaed3028c60e0f08cb6769cff8fb097ef26ce2e2
sismo/dsp.py	238	8535	f9177d2f643bfead4468a4eee6def6924bb71a4306215d05f1a0076391fbf6a7
sismo/engine.py	1966	97535	0f593890417031fb204304cde841c11ec83bbbd39795ea8d296448853d0d68f4
sismo/estreport.py	206	11006	1e0675fe885b295f3722f21b9fd002f4d692ff076d21cf9be659ce88f8f5e47d
sismo/focal.py	355	12243	92c7af18559d94bbe2a62e65d57d7a6c269938758d8e3fa4ab752d7dbb77d853
sismo/health.py	396	22422	995588133c85f257a4b47325dd6cb45faec9aa47c0d841013f715f71d21e269c
sismo/locator.py	226	11161	7a41093565488501cc8eac865c8ffc58b350e722bf86345f289c723e04cdb198
sismo/magglobal.py	249	11907	c52e8620f5967e805252dfe98dba4d11aa5e41ad051e406a12818dfd480a9c49
sismo/magnitude.py	112	4592	85488beafae21a9489c17ae60220db85b85dd76841baa6ba8afe02569d845d2d
sismo/mseed.py	119	4586	329b25f45d2cfc793abfaba24b5600570aefebf1adcb8da82b05b5d3c421849a
sismo/netio.py	301	14054	b7b402d2b34473638219ec01fc346f511d4640a07dc71b82f20f28a34fcb7226
sismo/placas.py	99	4545	a9d56cbfbe3bb13dab7443ad649c5064e60d00e5dce622b941b6e1e5da6141c0
sismo/places.py	224	20453	1ed500395acebbf8719dd8bba4d3e531d4985642a297b19d26def23a4b4d9310
sismo/report.py	546	28703	497a6151883270fd74032e54d3b2d6ca441e75062a5a36e9c0a6e97e74256ae9
sismo/server.py	942	52324	17fdb25bf9ed931dc803c7479c3498ee1905e8b5069ad54928db0e46823fb859
sismo/slprobe.py	86	3562	c8780ac9ddb8cf2d1bdf4906743b8e3bbab4c6bbf8bb1f27969db1018e7421cc
sismo/sonido.py	56	2179	7c2ff67f3f5f80e1d79bfdeaef741d7d50bfc9f6a75aac820ad5d4066e5152c5
sismo/storage.py	247	9579	9446e9118aeb91ae498aecf8f6ee614d99f47d588d4ab1aa6f6d4391480318d6
sismo/ticket.py	531	32903	180a558bc9ef51a4528db45cf285eaba36b3a71d7a7669d73bd4c26a624c802d
sismo/traveltime.py	345	16289	f177ebb4560b52a29d890669ad2e9bed5b1f39ca5f0d3ae7458c01dab2bf86e7
sismo/tsunami.py	188	9453	5057b8d6927f4cfb11de0ffc5d135f49a11f3daebc567ca34c02d25f202acdfa
sismo/usgsreport.py	194	9475	430118ac962b7e53a58a876902563de1b6362d2c433343633a25bdaee1145dc2
sismo/zones.py	25	851	9cd6ee5d053390c0c9d9a3257bfeef140b414214510e96c5c4e29aaf11d042d3
web/panel.html	4531	194498	a1a4a830f36122df8c7e06dc95004bc17243df0cdc780d60333c8dbc9927a005
web/traceview.html	1039	37476	60667c07f54067827542eb70054f540acb18ab88178cfd637443ff5254a6d8cb

Anexo B. Código fuente completo

Código fuente oficial de la versión 2.3.1 tal como se encuentra instalado y en operación. Las líneas largas se dividen y se identifican con números correlativos de línea. Todos los módulos fueron redactados e implementados por Joel Cruz Aragón.

run.py — 804 líneas

```
"""Monitor Sísmico MUNDIAL — motor en tiempo real.

Proyecto independiente del Monitor Sísmico México: carpeta propia (C:\\monitor_mundial), panel en el
puerto 8767 (el 8766 es el candado del Monitor México) y su propio candado de "una sola copia" (puerto 8768). Ambos pueden correr en la misma PC.

Uso:  python run.py            (arranca el monitor y abre el panel en el navegador)
      python run.py --sin-navegador
"""
import json
import logging
import os
import queue
import sys
import threading
import time
import webbrowser
from logging.handlers import RotatingFileHandler

BASE = os.path.dirname(os.path.abspath(__file__))
sys.path.insert(0, BASE)

from sismo.engine import Engine
from sismo.storage import Storage, Archive
from sismo.traveltime import TravelTimes
from sismo.netio import load_stations, usgs_query
from sismo.alerts import Alerts
from sismo.server import Api, start_server
from sismo.ticket import save as ticket_save, VERSION
from sismo.health import Health


def open_panel(url, web=None):
    """v2.0.0: el panel se abre siempre en Microsoft Edge (panel.navegador = "edge"; "predeterminado" usa el de Windows).
    Si Edge no se encuentra, se usa el navegador predeterminado."""
    nav = str((web or {}).get("navegador", "edge")).lower()
    if os.name == "nt" and nav == "edge":
        import subprocess
        cands = [os.path.join(os.environ.get(v, ""), "Microsoft", "Edge", "Application", "msedge.exe")
                 for v in ("ProgramFiles(x86)", "ProgramFiles", "LOCALAPPDATA")]
        for exe in cands:
            if os.path.isfile(exe):
                try:
                    subprocess.Popen([exe, url], close_fds=True)
                    return
                except OSError:
                    pass
        try:
            os.startfile("microsoft-edge:" + url)
            return
        except OSError:
            pass
    webbrowser.open(url)


def setup_log(folder):
    os.makedirs(folder, exist_ok=True)
    lg = logging.getLogger("monitor_mundial")
    lg.setLevel(logging.INFO)
    fmt = logging.Formatter("%(asctime)s  %(message)s", "%Y-%m-%d %H:%M:%S")
    fh = RotatingFileHandler(os.path.join(folder, "monitor.log"), maxBytes=5_000_000, backupCount=3, encoding="utf-8")
    fh.setFormatter(fmt)
    lg.addHandler(fh)
    if sys.stdout is not None:  # con pythonw (arranque oculto) no hay consola
        sh = logging.StreamHandler(sys.stdout)
        sh.setFormatter(fmt)
        lg.addHandler(sh)
    return lg.info


def backfill_prelim(engine, log_dir):
    """v0.5.3: recupera del monitor.log la magnitud preliminar de los sismos guardados antes de esta versión
    (línea «Sismo detectado: 5.3 …», escrita en el mismo segundo en que se creó el sismo)."""
    import re
    pend = [e for e in engine.events if e.get("magPre") is None and e.get("created")]
    if not pend:
        return 0
    rx = re.compile(r"^(\d{4}-\d\d-\d\d \d\d:\d\d:\d\d)  Sismo detectado: (\d+\.\d)(\??) ")
    found = []
    for name in ("monitor.log.3", "monitor.log.2", "monitor.log.1", "monitor.log"):
        try:
            with open(os.path.join(log_dir, name), encoding="utf-8", errors="replace") as f:
                for line in f:
                    m = rx.match(line)
                    if m:
                        t = time.mktime(time.strptime(m.group(1), "%Y-%m-%d %H:%M:%S"))
                        found.append((t, float(m.group(2)), m.group(2) + m.group(3)))
        except OSError:
            pass
    n = 0
```

```
    for e in pend:
        best = min(found, key=lambda x: abs(x[0] - e["created"]), default=None)
        if best and abs(best[0] - e["created"]) <= 2:
            e["magPre"], e["magPreStr"], e["magPreType"] = best[1], best[2], ""
            engine.dirty.add(e["id"])
            n += 1
    return n


def split_sessions_at_restarts(engine, log_dir):
    """v0.5.8: corta las sesiones guardadas en cada arranque registrado en monitor.log («inicia
  ndo»). Antes un
    reinicio rápido quedaba dentro de una sola sesión y los sismos ocurridos justo antes contab
  an como «perdidos
    con el monitor encendido»."""
    import re
    starts = []
    rx = re.compile(r"^(\d{4}-\d\d-\d\d \d\d:\d\d:\d\d)  === Monitor Sísmico MUNDIAL v[\d.]+: i
  niciando")
    for name in ("monitor.log.3", "monitor.log.2", "monitor.log.1", "monitor.log"):
        try:
            with open(os.path.join(log_dir, name), encoding="utf-8", errors="replace") as f:
                for line in f:
                    m = rx.match(line)
                    if m:
                        starts.append(time.mktime(time.strptime(m.group(1), "%Y-%m-%d %H:%M:%S"
  )))
        except OSError:
            pass
    out = []
    for a, b in engine.sessions:
        cuts = sorted(t for t in starts if a + 30 < t < b - 30)
        for t in cuts:
            out.append([a, t - 5])
            a = t
        out.append([a, b])
    changed = len(out) != len(engine.sessions)
    engine.sessions = out
    return changed


def backfill_final_time(engine, log_dir):
    """v0.5.4: recupera del monitor.log la hora de la solución final («Solución final: 5.0 Mw …
  ») de los sismos
    guardados antes de esta versión."""
    import re
    pend = [e for e in engine.events if e.get("final") and not e.get("tFinal") and e.get("creat
  ed")]
    if not pend:
        return 0
    rx = re.compile(r"^(\d{4}-\d\d-\d\d \d\d:\d\d:\d\d)  Solución final: (\S+)")
    fins = []
    for name in ("monitor.log.3", "monitor.log.2", "monitor.log.1", "monitor.log"):
        try:
            with open(os.path.join(log_dir, name), encoding="utf-8", errors="replace") as f:
                for line in f:
                    m = rx.match(line)
                    if m:
                        fins.append((time.mktime(time.strptime(m.group(1), "%Y-%m-%d %H:%M:%S")
  ), m.group(2).rstrip("?")))
        except OSError:
            pass
    used, n = set(), 0
    for e in sorted(pend, key=lambda e: e["created"]):
        ms = f"{e['mag']:.1f}" if e.get("mag") is not None else "s/m"
        cand = [f for f in fins if e["created"] - 2 <= f[0] <= e["created"] + 5400 and f[0] not
   in used]
        good = [f for f in cand if f[1] == ms]
        if not good and e.get("mag") is not None:     # la magnitud pudo recalcularse después (
  calibración)
            good = [f for f in cand if f[1][:1].isdigit() and abs(float(f[1]) - e["mag"]) <= 0.
  4]
        if good:
            e["tFinal"] = good[0][0]
            used.add(good[0][0])
            engine.dirty.add(e["id"])
            n += 1
    return n


def classify_zone(lat, lon):
    """Clasifica las coordenadas en una de las 8 zonas geográficas mundiales (1 a 8):
    1: Norteamérica
    2: Centroamérica y el Caribe
    3: Sudamérica
    4: África
    5: Europa
    6: Asia
    7: Oceanía
    8: El resto
    """
    # 2. Centroamérica y el Caribe (México centro/sur, Centroamérica, Antillas/Caribe)
    if 7.0 <= lat <= 23.5 and -120.0 <= lon <= -59.0:
        return 2, "Centroamérica y el Caribe"

```

```
    # 1. Norteamérica (México norte, EE.UU., Canadá, Alaska, Groenlandia)
    if (lat >= 23.5 and -175.0 <= lon <= -50.0) or (lat >= 50.0 and -180.0 <= lon <= -10.0):
        return 1, "Norteamérica"

    # 3. Sudamérica
    if -60.0 <= lat < 13.0 and -95.0 <= lon <= -30.0:
        return 3, "Sudamérica"

    # 5. Europa
    if 35.0 <= lat <= 72.0 and -25.0 <= lon <= 45.0:
        return 5, "Europa"

    # 4. África
    if -36.0 <= lat <= 37.0 and -20.0 <= lon <= 55.0:
        return 4, "África"

    # 7. Oceanía (Australia, Nueva Zelanda, Fiyi, Papúa, Pacífico Sur)
    if (-50.0 <= lat <= 0.0 and 110.0 <= lon <= 180.0) or (-30.0 <= lat <= 25.0 and (lon >= 150.0 or lon <= -130.0)):
        return 7, "Oceanía"

    # 6. Asia (Japón, Taiwán, Indonesia, Filipinas, China, India, Rusia asiática)
    if (-11.0 <= lat <= 78.0 and 45.0 <= lon <= 180.0) or (lat >= 50.0 and lon >= 140.0):
        return 6, "Asia"

    # 8. El resto (Antártida, dorsales oceánicas remotas)
    return 8, "El resto"


def make_sources(cfg, metas, q, archive, log):
    """Crea conexiones SeedLink agrupadas por fuente y zona geográfica para permitir inicio escalonado."""
    from sismo.acquisition import SeedLinkSource
    sources = []
    zone_order = [
        (1, "Norteamérica"),
        (2, "Centroamérica y el Caribe"),
        (3, "Sudamérica"),
        (4, "África"),
        (5, "Europa"),
        (6, "Asia"),
        (7, "Oceanía"),
        (8, "El resto"),
    ]
    for zid, zname in zone_order:
        zmetas = [m for m in metas if classify_zone(m["lat"], m["lon"])[0] == zid]
        if not zmetas:
            continue
        for src in cfg["fuentes"]:
            if not src.get("activa"):
                continue
            src_metas = [m for m in zmetas if m.get("fuente") == src["nombre"]]
            if not src_metas:
                continue
            band0 = str(src.get("canales", "BH")).split(",")[0].strip()[:2]
            todos = cfg.get("estaciones", {}).get("todos_los_canales", True)
            streams = []
            for m in src_metas:
                band = m.get("band") or band0
                if todos:
                    sels = ("BH?", "HH?", "EH?", "SH?", "HN?", "BN?", "EN?")
                elif src["nombre"] == "INETER":
                    bs = list(dict.fromkeys([band] + [k.split(".")[-1][:2] for k in m.get("chans", {}) if k.split(".")[-1][:2] in ("BH", "HH")]))
                    sels = tuple(f"{b}?" for b in bs[:2])
                else:
                    sels = tuple(m.get("sels") or [f"{m['loc']}{band}?" if m.get("loc") else f"{band}?"])
                streams.append((m["net"], m["sta"], sels))
            if streams:
                lbl = f"{src['nombre']} · {zname}"
                sl = SeedLinkSource(src["seedlink"], streams, q, archive, log, name=lbl)
                sl.alt_server = src.get("seedlink_respaldo") or None
                sl.backfill_s = 60 * float(cfg.get("estaciones", {}).get("relleno_al_arrancar_min", 0))
                sl.max_packet_age_s = float(cfg.get("estaciones", {}).get("antiguedad_max_paquete_s", 240))
                sl.zone_id = zid
                sl.zone_name = zname
                if todos:
                    sl.want = {m["key"]: f"{m.get('loc', '')}.{m['band']}" for m in src_metas if m.get("band")}
                sources.append(sl)

    for x in cfg.get("estaciones_propias", []):
        if x.get("activa"):
            sl = SeedLinkSource(x["servidor"], [(x["red"], x["estacion"])], q, archive, log,
                                selectors=("EH?", "HH?", "BH?"), name=f"propia {x['estacion']}")
            sl.zone_id = 2
            sl.zone_name = "Centroamérica y el Caribe"
            sources.append(sl)

    # Ingesta en tiempo real de la Red SS (SIDESIS Android) desde ringserver local (puerto 18000)
```

```
    sl_ss = SeedLinkSource("127.0.0.1:18000", [("SS", "*", ("00EN?", "EN?"))], q, archive, log,
                           selectors=("00EN?", "EN?"), name="SIDESIS Android (Red SS)")
    sl_ss.zone_id = 1
    sl_ss.zone_name = "Norteamérica"
    sl_ss.want = None
    sources.append(sl_ss)
    return sources


def main():
    with open(os.path.join(BASE, "config.json"), encoding="utf-8") as f:
        cfg = json.load(f)
    data_dir = os.path.join(BASE, cfg.get("carpeta_datos", "datos"))
    log = setup_log(os.path.join(data_dir, "registros"))
    log(f"=== Monitor Sísmico MUNDIAL v{VERSION}: iniciando ===")

    # Una sola copia del monitor MUNDIAL por computadora (el de México usa otros puertos)
    import socket
    port = cfg["panel"].get("puerto", 8767)
    lock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    if os.name == "nt":
        lock.setsockopt(socket.SOL_SOCKET, socket.SO_EXCLUSIVEADDRUSE, 1)
    # tras un reinicio, la copia anterior tarda unos segundos en soltar los puertos: se reintenta
    tries = 40 if "--reinicio" in sys.argv else 1
    for i in range(tries):
        try:
            lock.bind(("127.0.0.1", port + 1))
            lock.listen(1)
            break
        except OSError:
            if i == tries - 1:
                lock = None
            else:
                time.sleep(0.5)
    if lock is None:
        log("Ya hay un Monitor Sísmico MUNDIAL ejecutándose en esta computadora (o el puerto "
            f"{port + 1} está ocupado). Esta copia se cierra; abre el panel en http://localhost:{port}/")
        if "--sin-navegador" not in sys.argv:
            open_panel(f"http://localhost:{port}/", cfg.get("panel"))
        return
    globals()["_instance_lock"] = lock

    log("Cargando tablas de tiempos de viaje (Tierra esférica)…")
    tt = TravelTimes(cfg["deteccion"].get("modelo_velocidades", "ak135"), data_dir, log)
    log(f"Modelo de tiempos de viaje: {tt.name}")
    storage = Storage(data_dir)
    alerts = Alerts(cfg, os.path.join(data_dir, "reportes"), log)
    engine = Engine(cfg, tt, storage, alerts, log)
    try:
        ns = engine.calib.seed_mb(engine.events)       # v2.0.0
        if ns:
            log(f"Calibración v2.0.0: sesgo del mb iniciado con {ns} sismos confirmados por el USGS "
                f"(corrección {engine.calib.mb_offset():+.2f})")
    except Exception as e:
        log(f"Calibración v2.0.0: {e}")
    try:
        nb = backfill_prelim(engine, os.path.join(data_dir, "registros"))
        if nb:
            log(f"Magnitud preliminar recuperada del registro para {nb} sismos anteriores")
        split_sessions_at_restarts(engine, os.path.join(data_dir, "registros"))
        nf = backfill_final_time(engine, os.path.join(data_dir, "registros"))
        if nf:
            log(f"Hora de la solución final recuperada del registro para {nf} sismos anteriores")
    except Exception as e:
        log(f"Magnitud preliminar: no se pudo recuperar del registro ({e})")
    health = Health()
    engine.health = alerts.health = health

    if cfg.get("ia", {}).get("activa"):
        try:
            from sismo.ai import AIPicker
            engine.ai = AIPicker(cfg["ia"], log)
        except Exception as e:
            log(f"IA no disponible ({e}). Ejecuta instalar_ia.bat para activarla. Se continúa con el picador AIC.")

    # Estaciones sin datos (v0.4.2): se conservan suscritas hasta DIAS_SIN_DATOS días seguidos sin transmitir
    # (pueden estar fuera de servicio temporalmente); después se retiran y se reintentan pasado el mismo plazo.
    # Archivo: {estación: {"desde": inicio del silencio, "retirada": hora de retiro o null}}
    dead_path = "estaciones_sin_datos.json"
    DIAS = float(cfg.get("estaciones", {}).get("dias_sin_datos", 15))
    raw = storage.load_json(dead_path, {})
    silent = {}
    for k, v in raw.items():
        if isinstance(v, dict):
            silent[k] = {"desde": float(v.get("desde", time.time())), "retirada": v.get("retirada")}
        # formato anterior (v0.3/0.4.1: retiradas a los 30 min): se olvidan y se vuelven a probar
```

```
  for k in [k for k, v in silent.items() if v["retirada"] and (DIAS <= 0 or time.time() - v["retirada"] > DIAS * 86400)]:
      silent[k]["retirada"] = None     # retiro vencido (o retiros desactivados): se vuelve a incluir
  engine.silent = silent               # para el reporte de estaciones
  retired = {k for k, v in silent.items() if v["retirada"]}
  storage.save_json(dead_path, silent)
  cfg.setdefault("estaciones", {})["excluir"] = list(set(cfg["estaciones"].get("excluir", [])) | retired)

  # límites de placas tectónicas para el mapa (se descargan una sola vez, en segundo plano)
  def _placas():
      try:
          from sismo import placas
          placas.load(data_dir, log)
      except Exception as e:
          log(f"Placas tectónicas: {e}")
  threading.Thread(target=_placas, daemon=True, name="placas").start()

  log("Obteniendo estaciones de la red mundial…")
  metas = load_stations(cfg, os.path.join(data_dir, "estaciones.json"), log)
  if getattr(load_stations, "from_cache", False):
      health.fail("metadatos", "Los servicios FDSN no respondieron; se usó la lista guardada de estaciones")
  elif getattr(load_stations, "failed", None):
      health.fail("metadatos", "Sin respuesta de metadatos de: " + ", ".join(load_stations.failed)
                  + " (se usó su lista guardada si existía). Revisa internet/antivirus y reinicia el monitor")
  else:
      health.beat("metadatos", "lista actualizada desde FDSN")
  for m in metas:
      engine.add_station(m)
  for x in cfg.get("estaciones_propias", []):
      if x.get("activa"):
          engine.add_station({"key": f"{x['red']}_{x['estacion']}", "net": x["red"], "sta": x["estacion"], "lat": x["lat"],
                              "lon": x["lon"], "fuente": "propia",
                              "chans": {f"{x.get('ubicacion', '00')}.{c}": x.get("cuentas_por_m_s", 0) for c in x.get("canales", ["EHZ"])}})

  # Cargar y registrar catálogo de estaciones Android SIDESIS (Red SS)
  ss_file = "estaciones_ss.json"
  ss_stas = storage.load_json(ss_file, {}) or {}
  for sta_code, meta in ss_stas.items():
      engine.register_ss_station(meta, persist=False)
  # Purga automática de estaciones SS inactivas (> 7 días)
  engine.purge_inactive_ss_stations(max_days=7)
  ss_active_count = sum(1 for s in engine.stations.values() if s.net == "SS")
  log(f"SIDESIS: {ss_active_count} estación(es) de la Red SS registradas en el motor")

  # Cargar y registrar estaciones personales Raspberry Shake (DataCast)
  rs_cfg = cfg.get("raspberryshake_datacast", {}) or cfg.get("raspberryshake", {})
  rs_stas = rs_cfg.get("estaciones", [])
  for est in rs_stas:
      p = int(est.get("puerto") or 8888)
      code = str(est.get("codigo") or f"RS{p}").strip()
      net = str(est.get("red") or "AM").strip()
      lat = float(est.get("lat", 19.4326))
      lon = float(est.get("lon", -99.1332))
      nombre = est.get("nombre") or f"Raspberry Shake {code}"
      scale = float(est.get("cuentas_por_m_s") or est.get("scale") or 3.99e8)
      engine.add_station({
          "key": f"{net}_{code}",
          "net": net,
          "sta": code,
          "lat": lat,
          "lon": lon,
          "where": nombre,
          "fuente": "Raspberry Shake (DataCast)",
          "pref": "00.EH",
          "chans": {"00.EHZ": scale, "00.SHZ": scale, "EHZ": scale, "SHZ": scale}
      })
  if rs_stas:
      log(f"Raspberry Shake: {len(rs_stas)} estación(es) personales registradas en el motor")

  def _ss_cleanup_worker():
      while True:
          time.sleep(21600)  # Cada 6 horas
          try:
              engine.purge_inactive_ss_stations(max_days=7)
          except Exception as ex:
              log(f"SIDESIS: error en purga periódica ({ex})")
  threading.Thread(target=_ss_cleanup_worker, daemon=True, name="ss_cleanup").start()

  log(f"{len(engine.stations)} estaciones configuradas")
  engine.assoc.build(engine.stations)

  # Buffer de absorción de ráfagas (20,000 paquetes) para evitar caídas y entrecortes de señal
  q = queue.Queue(maxsize=20000)
  archive = Archive(os.path.join(data_dir, "archivo"), cfg["archivo"]["dias"], log) if cfg["archivo"].get("activo") else None
  sources = make_sources(cfg, metas, q, archive, log)

```

```
  def start_staggered_sources():
      """Conexión escalonada de fuentes por zonas (1 a 8):
      - Inicia Zona 1 (Norteamérica)
      - Monitorea hasta que la cola esté en nivel óptimo (q.qsize() < 200 durante ~6 s)
      - Si tarda, tiene un tope de seguridad de máx 60 s por zona (carga total < 8-10 min)
      - Avanza sucesivamente a la siguiente zona hasta completar las 8 zonas.
      """
      zones = {}
      for s in sources:
          s.stations = engine.stations
          zid = getattr(s, "zone_id", 8)
          zones.setdefault(zid, []).append(s)

      zone_names = {
          1: "Norteamérica",
          2: "Centroamérica y el Caribe",
          3: "Sudamérica",
          4: "África",
          5: "Europa",
          6: "Asia",
          7: "Oceanía",
          8: "El resto"
      }

      t_global_start = time.time()
      for zid in sorted(zone_names.keys()):
          zsources = zones.get(zid, [])
          if not zsources:
              continue
          zname = zone_names[zid]
          n_sta = sum(len(s.streams) for s in zsources)
          log(f"Inicio escalonado [Zona {zid}/8 · {zname}]: conectando {len(zsources)} fuente
(s) con {n_sta} estaciones…")
          for s in zsources:
              if not s.is_alive():
                  try:
                      s.start()
                  except RuntimeError:
                      pass

          # Monitoreo de nivel óptimo antes de liberar la siguiente zona (máx 60 s por zona)
          t_zone_start = time.time()
          stable_ok = 0
          while time.time() - t_zone_start < 60:
              time.sleep(2)
              qs = q.qsize()
              if qs < 200:
                  stable_ok += 1
                  if stable_ok >= 3:
                      log(f"Inicio escalonado [Zona {zid} · {zname}]: cola en nivel óptimo ({
qs} paquetes). Listo para siguiente zona.")
                      break
              else:
                  stable_ok = 0

          if stable_ok < 3 and zid < 8:
              log(f"Inicio escalonado [Zona {zid} · {zname}]: tiempo límite de zona alcanzado
 (cola: {q.qsize()}); avanzando a siguiente zona por seguridad.")

      elapsed = time.time() - t_global_start
      log(f"=== Inicio escalonado COMPLETADO en {elapsed / 60:.1f} min: las 8 zonas mundiales
 están conectadas ===")

  threading.Thread(target=start_staggered_sources, daemon=True, name="staggered-connect").sta
rt()

  # DataCast UDP para estaciones Raspberry Shake propias
  try:
      from sismo.datacast import DataCastServer
      datacast_srv = DataCastServer(engine, q, cfg, log)
      datacast_srv.start()
  except Exception as e_dc:
      log(f"DataCast UDP: {e_dc}")

  usgs_cache = {"list": []}
  vcfg = cfg.get("validacion", {})

  def usgs_ref():
      return usgs_cache["list"]

  def refresh_usgs():
      try:
          world = usgs_query(2, vcfg.get("magnitud_minima_usgs", 4.5))
          usgs_cache["list"] = world
          engine.set_usgs(world)
          health.beat("usgs", f"{len(world)} sismos de 2 días")
      except Exception as e:
          log(f"USGS: sin respuesta ({e})")
          health.fail("usgs", f"sin respuesta ({e})")

  def replace_dead_stations():
      """Cada 30 min: anota desde cuándo cada estación no transmite y borra de la lista las q
ue volvieron.
      Solo las que llevan DIAS días seguidos sin datos se retiran y se sustituyen por otras d
e la zona."""
```

```
        nonlocal metas, sources
        now = time.time()
        with engine.lock:
            mute = {k for s in sources for k in s.rejected}
            mute |= {k for k, st in engine.stations.items()
                     if getattr(st, "fuente", "") != "propia" and (not st.last_data or now - st.last_data > 1800)}
            alive = set(engine.stations) - mute
        changed = False
        for k in mute:
            if k not in silent:
                silent[k] = {"desde": now, "retirada": None}
                changed = True
        for k in list(silent):
            if k in alive:
                silent.pop(k)
                changed = True
        # dias_sin_datos = 0: nunca se retira ninguna estación (v0.4.6); solo se anota desde cuándo no transmite
        due = {k for k, v in silent.items() if DIAS > 0 and not v["retirada"] and now - v["desde"] > DIAS * 86400}
        for k in due:
            silent[k]["retirada"] = now
        if changed or due:
            storage.save_json(dead_path, silent)
        waiting = sum(1 for v in silent.values() if not v["retirada"])
        if not due:
            if changed:
                log(f"Estaciones sin datos: {waiting} en espera" + (f" (se conservan hasta {DIAS:.0f} días por si vuelven a transmitir)"
                                                                    if DIAS > 0 else " (se conservan siempre; no se retira ninguna)"))
            return
        cfg["estaciones"]["excluir"] = list(set(cfg["estaciones"].get("excluir", [])) | due)
        try:
            new = load_stations(cfg, os.path.join(data_dir, "estaciones.json"), log)
        except Exception as e:
            log(f"Reemplazo de estaciones: no se pudo consultar la red ({e})")
            return
        added = [m for m in new if m["key"] not in engine.stations]
        with engine.lock:
            for k in due:
                engine.stations.pop(k, None)
            for m in added:
                engine.add_station(m)
        log(f"Estaciones sin datos durante {DIAS:.0f} días: {len(due)} retiradas; {len(added)} nuevas en su lugar. Reconectando…")
        for s in sources:
            s.stop()
        metas = new
        sources = make_sources(cfg, metas, q, archive, log)
        api.sources = sources
        for s in sources:
            s.start()

    web = cfg["panel"]
    api = Api(engine, alerts, sources, os.path.join(BASE, "web"), usgs_ref)
    api.ctx = {"engine": engine, "alerts": alerts, "queue": q, "storage": storage, "cfg": cfg, "data_dir": data_dir,
               "started": time.time(), "health": health, "log_path": os.path.join(data_dir, "registros", "monitor.log")}
    restart_req = threading.Event()
    api.on_restart = restart_req.set
    srv = None
    for i in range(40 if "--reinicio" in sys.argv else 1):
        try:
            srv = start_server(api, web.get("host", "127.0.0.1"), web.get("puerto", 8767))
            break
        except OSError:
            if i == 39 or "--reinicio" not in sys.argv:
                raise
            time.sleep(0.5)
    url = f"http://localhost:{web.get('puerto', 8767)}/"
    log(f"Panel disponible en {url}")
    if "--sin-navegador" not in sys.argv and "--reinicio" not in sys.argv and web.get("abrir_navegador", True):
        threading.Timer(2, lambda: open_panel(url, web)).start()

    def start_raw_mseed_bridge(raw_port=16001, dl_port=16000):
        """Puente TCP opcional para Modo A (Raw 512B TCP stream):
        Recibe bloques continuos de 512 bytes de miniSEED sin encabezado ASCII,
        extrae el timestamp y lo inyecta como DataLink WRITE a ringserver."""
        def client_worker(conn):
            dl_sock = None
            try:
                dl_sock = socket.create_connection(("127.0.0.1", dl_port), timeout=10)
                buf = bytearray()
                while True:
                    chunk = conn.recv(4096)
                    if not chunk:
                        break
                    buf.extend(chunk)
                    while len(buf) >= 512:
                        rec = bytes(buf[:512])
                        del buf[:512]
```

```
                        from sismo.acquisition import _header
                        h = _header(rec)
                        if h:
                            net, sta, loc, cha, t0 = h
                            hpstart = int(t0 * 1_000_000)
                            hpend = hpstart + 1_980_000
                            cmd = f"WRITE {net}_{sta}_{loc}_{cha}/MSEED {hpstart} {hpend} N 512
    ".encode("ascii")
                            dl_pkt = b"DL" + bytes([len(cmd)]) + cmd + rec
                            dl_sock.sendall(dl_pkt)
            except Exception:
                pass
            finally:
                try:
                    conn.close()
                except Exception:
                    pass
                if dl_sock:
                    try:
                        dl_sock.close()
                    except Exception:
                        pass

        def server_loop():
            try:
                s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
                s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
                s.bind(("0.0.0.0", raw_port))
                s.listen(20)
                while True:
                    conn, _ = s.accept()
                    threading.Thread(target=client_worker, args=(conn,), daemon=True).start()
            except Exception as e:
                log(f"Puente Raw miniSEED (puerto {raw_port}): {e}")

        threading.Thread(target=server_loop, daemon=True, name="raw-mseed-bridge").start()

    start_raw_mseed_bridge(16001, 16000)

    t_loop = t_usgs = t_save = t_sess = t_dog = t_tk = 0.0
    t_start = time.time()
    t_dead = 0.0
    tickets_dir = os.path.join(data_dir, "tickets")

    def save_tickets(now):
        """Ticket técnico automático: al tener solución final y otra vez al cerrar el evento (6
    0 min).
        Solo lee los eventos; cualquier error se registra y no afecta la operación."""
        mmin = engine.cfg.get("alertas", {}).get("magnitud_minima", 4.0)
        with engine.lock:
            todo = []
            for ev in engine.events[:50]:
                if not ev.get("live"):
                    continue
                if (ev.get("mag") is not None and ev["mag"] < mmin) or ev.get("noMag"):
                    continue
                age = now - ev.get("created", now)
                if ev.get("final") and not ev.get("_tk1"):
                    ev["_tk1"] = True
                    todo.append(ev)
                elif ev.get("match") and not ev.get("_tk_match"):
                    ev["_tk_match"] = True
                    todo.append(ev)
                elif ev.get("_tk_dirty"):
                    ev["_tk_dirty"] = False
                    todo.append(ev)
                elif age > 59 * 60 and not ev.get("_tk2"):
                    ev["_tk2"] = True
                    todo.append(ev)
            for ev in todo:
                try:
                    path = ticket_save(engine, ev, tickets_dir)
                    log(f"Ticket técnico guardado: {os.path.relpath(path, BASE)}")
                    health.beat("tickets")
                except Exception as e:
                    log(f"Ticket técnico: no se pudo guardar ({e})")
                    health.fail("tickets", e)
    def do_restart():
        """Reinicio ordenado: guarda lo pendiente, suelta los puertos y arranca una copia nueva
    ."""
        log("=== Reinicio solicitado: guardando y reiniciando el monitor ===")
        try:
            with engine.lock:
                ids = set(engine.dirty)
                engine.dirty.clear()
                if ids:
                    storage.save_events([e for e in engine.events if e["id"] in ids])
                engine.mark_session()
                # v0.6.3: guarda las formas de onda de los sismos recientes antes de reiniciar
    (se pierden de la memoria)
                for ev in engine.events[:20]:
                    if ev.get("live") and not ev.get("wfSaved"):
                        try:
                            storage.save_waveforms(ev["id"], engine.section_traces(ev, 0.5, 4,
    decimate=True))
```

```
                            ev["wfSaved"] = True
                        except Exception:
                            pass
        except Exception as e:
            log(f"Reinicio: aviso al guardar ({e})")
        for s in sources:
            try:
                s.stop()
            except Exception:
                pass
        try:
            if archive is not None:
                archive.flush()
        except Exception:
            pass
        try:
            srv.shutdown()
            srv.server_close()
        except Exception:
            pass
        try:
            globals()["_instance_lock"].close()
        except Exception:
            pass
        args = [a for a in sys.argv[1:] if a not in ("--reinicio", "--sin-navegador")]
        if "--bucle" in sys.argv:
            # iniciar.bat vuelve a arrancar el monitor en la misma ventana al ver el código 3
            logging.shutdown()
            os._exit(3)
        import subprocess
        flags = 0
        if os.name == "nt":
            hidden = os.path.basename(sys.executable).lower().startswith("pythonw")
            flags = (0x00000008 | 0x00000200) if hidden else 0x00000010   # DETACHED+NEW_GROUP
    | NEW_CONSOLE
        subprocess.Popen([sys.executable, os.path.abspath(__file__), *args, "--reinicio", "--si
    n-navegador"],
                         cwd=BASE, creationflags=flags, close_fds=True)
        logging.shutdown()
        os._exit(0)

    while True:
        try:
            if restart_req.is_set():
                do_restart()
            for _ in range(10000):
                try:
                    item = q.get(timeout=0.05)
                except queue.Empty:
                    break
                engine.on_trace(*item)
            now = time.time()
            health.beat("ciclo")
            if now - t_loop > 3:
                t_loop = now
                try:
                    engine.phase_loop()
                    health.beat("fases")
                except Exception as e:
                    health.fail("fases", e)
                    raise
                try:
                    alerts.poll(engine)   # aviso PRELIMINAR en cuanto hay magnitud
                except Exception as e:
                    log(f"Alertas: {e}")
            if now - t_usgs > 180:
                t_usgs = now
                threading.Thread(target=refresh_usgs, daemon=True).start()
            if now - t_save > 5 and engine.dirty:
                t_save = now
                with engine.lock:
                    ids = set(engine.dirty)
                    engine.dirty.clear()
                    try:
                        storage.save_events([e for e in engine.events if e["id"] in ids])
                        health.beat("guardado")
                    except Exception as e:
                        health.fail("guardado", e)
                        log(f"Guardado de sismos: {e}")
            if now - t_sess > 60:
                t_sess = now
                engine.mark_session()
                if q.qsize() > 20000:
                    log(f"Aviso: la PC va retrasada procesando datos ({q.qsize()} paquetes en e
    spera). "
                        "Reduce estaciones.max_estaciones en config.json.")
            if now - t_tk > 20:
                t_tk = now
                try:
                    save_tickets(now)
                except Exception as e:
                    log(f"Ticket técnico: {e}")
            if now - t_dog > 30:
                t_dog = now
                for s in sources:
```

```
                    s.watchdog(180)
            if now - t_start > 30 * 60 and now - t_dead > 30 * 60:
                t_dead = now
                try:
                    replace_dead_stations()
                except Exception as e:
                    log(f"Reemplazo de estaciones: {e}")
        except KeyboardInterrupt:
            log("Detenido por el usuario.")
            break
        except Exception as e:
            log(f"Error en el ciclo principal: {e}")
            time.sleep(1)


if __name__ == "__main__":
    main()
```

sismo/__init__.py — 1 líneas

```
1  """Monitor Sísmico México: motor de detección, localización y magnitud."""
```

sismo/acquisition.py — 324 líneas

```
"""Adquisición en tiempo real por SeedLink.

Cliente SeedLink v3 propio (socket TCP), compatible con el servidor de EarthScope (ringserver con
SeedLink 4). El cliente de ObsPy falla con ese servidor al interpretar su saludo, por eso aquí solo
se usa ObsPy para decodificar los registros miniSEED.
"""
import io
import socket
import threading
import time
import struct
import calendar
import queue as _queue

PACKET = 520          # "SL" + 6 dígitos de secuencia + registro miniSEED de 512 bytes


def _decoder():
    """Decodificador miniSEED: ObsPy (_read_mseed directo en C, 6.4x más rápido que el envoltorio obspy.read)
    si está instalado; si no, el lector propio (sismo/mseed.py)."""
    try:
        from obspy.io.mseed.core import _read_mseed

        def dec(rec):
            out = []
            for tr in _read_mseed(io.BytesIO(rec)):
                s = tr.stats
                out.append((s.network, s.station, s.location, s.channel, float(s.sampling_rate),
                            float(s.starttime.timestamp), tr.data))
            return out
        return dec
    except ImportError:
        from .mseed import read_records
        return lambda rec: list(read_records(rec))


def _header(rec):
    """v0.5.2: lee red/estación/ubicación/canal/hora del encabezado miniSEED SIN decodificar las muestras."""
    try:
        sta = rec[8:13].decode("ascii").strip(); loc = rec[13:15].decode("ascii").strip()
        cha = rec[15:18].decode("ascii").strip(); net = rec[18:20].decode("ascii").strip()
        y, d = struct.unpack(">HH", rec[20:24])
        if not 1900 < y < 2100:
            y, d = struct.unpack("<HH", rec[20:24])
        t0 = calendar.timegm((y, 1, 1, rec[24], rec[25], rec[26])) + (d - 1) * 86400
        return net, sta, loc, cha, t0
    except Exception:
        return None


class SeedLinkSource(threading.Thread):
    """streams: lista de (red, estación) o (red, estación, selectores). Los selectores por estación
    permiten pedir solo la ubicación elegida (p. ej. "00BH?"), la mitad de datos en redes con dos sensores."""

    def __init__(self, server, streams, queue, archive=None, log=print, selectors=("?H?",), name=None):
        super().__init__(daemon=True, name=f"seedlink-{server}")
        self.server, self.streams, self.queue, self.archive, self.log = server, streams, queue, archive, log
        self.selectors = selectors
        self.label = name or server
        self.last_data = 0.0
        self.packets = 0
        self.queued = 0
        self.sock = None
        self.stop_flag = False
        self.rejected = set()
        self.accepted = set()

    def stop(self):
        self.stop_flag = True
        try:
            self.sock.sendall(b"BYE\r\n")
        except Exception:
            pass
        try:
            self.sock.shutdown(socket.SHUT_RDWR)
        except Exception:
            pass

    # ------------------------------------------------------------ protocolo
    def _cmd(self, f, text, expect=True):
        self.sock.sendall((text + "\r\n").encode("ascii"))
        if not expect:
            return ""
        line = f.readline()
        if not line:
            raise ConnectionError("el servidor cerró la conexión")
```

```
        return line.decode("ascii", "replace").strip()

    def _session(self):
        host, port = self.server.rsplit(":", 1)
        self.sock = socket.create_connection((host, int(port)), timeout=30)
        self.sock.settimeout(60)
        f = self.sock.makefile("rb")
        hello = self._cmd(f, "HELLO")
        org = f.readline().decode("ascii", "replace").strip()
        self.log(f"SeedLink: conectado a {self.server} — {hello} / {org}")
        # comandos en lotes de 20 estaciones (cada comando responde una línea OK/ERROR). Alguno
s servidores
        # SeisComP (p. ej. IPGP) no contestan comandos enviados en lote: si no responden, la si
guiente conexión
        # manda las estaciones de una en una (v0.4.5).
        ok = 0
        self._configuring = True
        # v0.5.6: los servidores SeisComP clásicos (SeedLink v3, sin RingServer) leen UN comand
o por paquete y
        # descartan los demás que llegan juntos: con ellos se manda comando por comando esperan
do cada "OK".
        if "RingServer" not in hello and "HMB" not in hello and not getattr(self, "batch_ok", F
alse):
            self.sequential = True
        if getattr(self, "sequential", False):
            ok = self._subscribe_one_by_one(f)
            self._configuring = False
            self.sock.settimeout(60)
            if ok == 0:
                raise ConnectionError("el servidor no aceptó ninguna estación")
            self._cmd(f, "END", expect=False)
            self.log(f"SeedLink {self.label} ({self.server}): {ok} de {len(self.streams)} estac
iones aceptadas; recibiendo datos")
            self.last_data = time.time()
            return self._read_loop(f)
        step = 20
        self.sock.settimeout(30)
        for i in range(0, len(self.streams), step):
            batch = self.streams[i:i + step]
            cmds, per = [], []
            for item in batch:
                net, sta = item[0], item[1]
                sels = item[2] if len(item) > 2 and item[2] else self.selectors
                cmds.append(f"STATION {sta} {net}")
                cmds += [f"SELECT {sel}" for sel in sels]
                cmds.append(self._data_cmd())
                per.append(2 + len(sels))
            self.sock.sendall(("".join(c + "\r\n" for c in cmds)).encode("ascii"))
            for j, item in enumerate(batch):
                resp = []
                for _ in range(per[j]):
                    line = f.readline()
                    if not line:
                        raise ConnectionError("el servidor cerró la conexión al configurar esta
ciones")
                    resp.append(line.decode("ascii", "replace").strip())
                key = f"{item[0]}_{item[1]}"
                if resp[0].startswith("OK") and resp[-1].startswith("OK"):
                    ok += 1
                    self.accepted.add(key)
                else:
                    self.rejected.add(key)
        self._configuring = False
        self.sock.settimeout(60)
        if ok == 0:
            raise ConnectionError("el servidor no aceptó ninguna estación")
        self._cmd(f, "END", expect=False)
        self.log(f"SeedLink {self.label} ({self.server}): {ok} de {len(self.streams)} estacione
s aceptadas; recibiendo datos")
        self.last_data = time.time()
        return self._read_loop(f)

    def _data_cmd(self):
        """v0.5.8: en la PRIMERA conexión tras arrancar se piden los últimos minutos guardados
en el servidor
        (comando TIME), para no perder un sismo que ocurrió justo antes de un reinicio. Después
, solo datos nuevos."""
        back = getattr(self, "backfill_s", 0)
        if back and not getattr(self, "_backfilled", False):
            return "TIME " + time.strftime("%Y,%m,%d,%H,%M,%S", time.gmtime(time.time() - back)
)
        return "DATA"

    def _subscribe_one_by_one(self, f):
        ok = 0
        self.sock.settimeout(20)
        for item in self.streams:
            net, sta = item[0], item[1]
            sels = item[2] if len(item) > 2 and item[2] else self.selectors
            key = f"{net}_{sta}"
            r = self._cmd(f, f"STATION {sta} {net}")
            if not r.startswith("OK"):
                self.rejected.add(key)
                continue
            good = 0
```

```
            for sel in sels:
                if self._cmd(f, f"SELECT {sel}").startswith("OK"):
                    good += 1
            dcmd = self._data_cmd()
            r = self._cmd(f, dcmd)
            if not r.startswith("OK") and dcmd != "DATA":
                self._backfilled = True
                r = self._cmd(f, "DATA")
            if good and r.startswith("OK"):
                ok += 1
                self.accepted.add(key)
            else:
                self.rejected.add(key)
        return ok

    def _read_loop(self, f):
        self._backfilled = True
        decode = _decoder()
        while not self.stop_flag:
            pkt = f.read(PACKET)
            if not pkt or len(pkt) < PACKET:
                raise ConnectionError("conexión interrumpida")
            if pkt[:6] == b"SLINFO":
                continue
            if pkt[:2] != b"SL":
                raise ConnectionError("paquete desconocido; se reinicia la conexión")
            rec = pkt[8:]
            h = _header(rec)
            # Opción A: Descarte de ráfagas acumuladas al reconectar (paquetes viejos > max_pac
    ket_age_s)
            if h is not None:
                max_age = getattr(self, "max_packet_age_s", 180.0)
                if max_age and (time.time() - h[4] > max_age):
                    self.last_data = time.time()
                    self.stale_dropped = getattr(self, "stale_dropped", 0) + 1
                    if time.time() - getattr(self, "_last_stale_warn", 0) > 60:
                        self._last_stale_warn = time.time()
                        self.log(f"SeedLink {self.label}: descartando ráfaga acumulada ({time.t
    ime() - h[4]:.0f} s de antigüedad)")
                    continue

            # v0.5.2: los canales que el motor no usa (otras bandas de «todos los canales») sol
    o se archivan,
            # sin decodificarlos: decodificar todo saturaba la PC durante sismos grandes.
            want = getattr(self, "want", None)
            eng_st = getattr(self, "stations", None)
            if (want or eng_st) and h is not None:
                k_st = f"{h[0]}_{h[1]}"
                cur_b = f"{h[2]}.{h[3][:2]}"
                pref = want.get(k_st) if want else None
                if pref and cur_b != pref:
                    self.last_data = time.time()
                    self.packets += 1
                    if self.archive is not None:
                        try:
                            self.archive.write_record((h[0], h[1], h[2], h[3], 0.0, h[4]), rec)
                        except Exception:
                            pass
                    continue
                if eng_st is not None:
                    sobj = eng_st.get(k_st)
                    if sobj is not None and sobj.band and sobj.band != cur_b and sobj.last_data
     and time.time() - sobj.last_data < 120:
                        if cur_b != sobj.pref:
                            self.last_data = time.time()
                            self.packets += 1
                            continue
            try:
                st = decode(rec)
            except Exception:
                continue
            self.last_data = time.time()
            self.packets += 1
            for item in st:
                self.queued += 1
                self._put(item)
            if self.archive is not None and len(st):
                try:
                    self.archive.write_record(st[0], rec)
                except Exception as e:
                    if time.time() - getattr(self, "_arch_err", 0) > 600:   # a lo más un aviso
     cada 10 min
                        self._arch_err = time.time()
                        self.log(f"Archivo: no se pudo escribir ({e})")

    def _put(self, item):
        """v0.5.2: nunca bloquea la lectura del socket (si se bloquea, el servidor corta la con
    exión).
        Si la cola está llena se descarta el paquete más viejo."""
        try:
            self.queue.put_nowait(item)
        except _queue.Full:
            try:
                self.queue.get_nowait()
            except _queue.Empty:
```

```
                pass
            try:
                self.queue.put_nowait(item)
            except _queue.Full:
                pass
            self.dropped = getattr(self, "dropped", 0) + 1
            self.drop_t = time.time()
            if time.time() - getattr(self, "_drop_t", 0) > 600:
                self._drop_t = time.time()
                self.log(f"SeedLink {self.server}: la PC va retrasada; se descartaron {self.dro
pped} paquetes viejos "
                         "para no perder la conexión")

  def run(self):
      wait = 5
      while not self.stop_flag:
          started = time.time()
          p0 = self.packets
          try:
              self.log(f"SeedLink: conectando a {self.server} ({len(self.streams)} estaciones
)…")
              self._session()
          except Exception as e:
              if getattr(self, "_configuring", False) and isinstance(e, (socket.timeout, Time
outError)):
                  self._configuring = False
                  if not getattr(self, "sequential", False):
                      self.sequential = True
                      self.log(f"SeedLink {self.server}: no respondió a la suscripción en lot
e; "
                               "se reintenta enviando las estaciones de una en una")
                      continue
                  e = "el servidor acepta la conexión pero no responde a la suscripción de es
taciones " \
                      "(puede no ofrecerlas en tiempo real o restringir el acceso)"
                  self.config_fail = time.time()
              if not self.stop_flag:          # al cerrar a propósito (reemplazo de estacione
s) no es un error
                  self.log(f"SeedLink {self.server}: {e}")
          finally:
              try:
                  self.sock.close()
              except Exception:
                  pass
          if self.stop_flag:
              break
          # v2.1.0: historial de cortes para el indicador de calidad de internet del panel (s
olo si la sesión ya transmitía datos)
          if self.packets > p0:
              self.cuts = [t for t in getattr(self, "cuts", []) if time.time() - t < 3600] +
[time.time()]
          if time.time() - started > 300:
              wait = 5
              self._fails = 0
          else:
              self._fails = getattr(self, "_fails", 0) + 1
          alt = getattr(self, "alt_server", None)
          if alt and alt != self.server and self._fails >= 3:      # v0.5.6: servidor de resp
aldo (p. ej. INETER .59 ⇔ .42)
              self.server, self.alt_server, self._fails = alt, self.server, 0
              self.log(f"SeedLink {self.label}: se cambia al servidor de respaldo {self.serve
r}")
          self.log(f"SeedLink {self.server}: reconexión en {wait} s")
          time.sleep(wait)
          wait = min(330, wait * 2)

  def watchdog(self, limit=180):
      """Si no llegan datos en 'limit' segundos, fuerza la reconexión."""
      if self.sock is not None and self.last_data and time.time() - self.last_data > limit:
          self.log(f"SeedLink {self.server}: sin datos en {limit} s, reconectando")
          self.last_data = time.time()
          try:
              self.sock.shutdown(socket.SHUT_RDWR)
          except Exception:
              pass
```

sismo/ai.py — 86 líneas

```
"""Picado de fases con inteligencia artificial (PhaseNet vía SeisBench). Opcional.

Se usa para refinar cada sismo detectado: en cada estación se analiza la ventana alrededor de
los arribos P y S esperados; si la red neuronal encuentra la fase con probabilidad suficiente,
su pick sustituye al automático (AIC). Los picks manuales nunca se reemplazan.
"""
import numpy as np

from .locator import dist_km


class AIPicker:
    def __init__(self, cfg, log=print):
        import seisbench.models as sbm  # requiere instalar_ia.bat
        self.log = log
        self.thr_p = cfg.get("umbral_p", 0.3)
        self.thr_s = cfg.get("umbral_s", 0.3)
        name = cfg.get("modelo_entrenado", "stead")
        self.model = sbm.PhaseNet.from_pretrained(name)
        self.model.eval()
        log(f"IA: PhaseNet cargado (pesos '{name}')")

    def _stream(self, s, t0, t1):
        from obspy import Stream, Trace, UTCDateTime
        st = Stream()
        for comp, c in s.comps.items():
            got = c.ring.get(t0, t1)
            if got is None:
                return None
            ts, data = got
            tr = Trace(data=data.astype(np.float32))
            tr.stats.network, tr.stats.station = s.net, s.sta
            tr.stats.location = s.band.split(".")[0]
            tr.stats.channel = s.band.split(".")[1] + comp
            tr.stats.sampling_rate = c.fs
            tr.stats.starttime = UTCDateTime(ts)
            st += tr
        return st

    def refine(self, engine, ev):
        """Devuelve True si cambió algún pick del evento."""
        changed = False
        done = ev.setdefault("aiDone", set())
        for key in {p["key"] for p in ev["picks"] if p["phase"] == "P"}:
            if key in done:
                continue
            s = engine.stations.get(key)
            if not s or "Z" not in s.comps:
                done.add(key)
                continue
            d = float(dist_km(ev["lat"], ev["lon"], s.lat, s.lon))
            if d > 2500:          # en esta fase la IA solo se usa a distancias regionales
                done.add(key)
                continue
            tp = ev["org"] + engine.ttf("P", d, ev["z"])
            ts = ev["org"] + engine.ttf("S", d, ev["z"])
            t0, t1 = tp - 30, ts + 30
            if min((c.next_t or 0) for c in s.comps.values()) < t1:
                continue  # aún no hay datos
            done.add(key)
            st = self._stream(s, t0, t1)
            if st is None:
                continue
            try:
                out = self.model.classify(st, P_threshold=self.thr_p, S_threshold=self.thr_s)
                picks = getattr(out, "picks", out)
            except Exception as e:
                self.log(f"IA: error en {s.code}: {e}")
                continue
            for phase, tref, tol in (("P", tp, max(4, 0.04 * (tp - ev["org"]))), ("S", ts, max(
    6, 0.08 * (ts - ev["org"])))):
                cand = [pk for pk in picks if getattr(pk, "phase", "") == phase and abs(pk.peak
    _time.timestamp - tref) < tol]
                if not cand:
                    continue
                best = max(cand, key=lambda pk: pk.peak_value)
                prob = float(best.peak_value)
                old = next((p for p in ev["picks"] if p["key"] == key and p["phase"] == phase),
     None)
                if old is not None and old.get("manual"):
                    continue
                sigma = (0.08 if phase == "P" else 0.2) + (1 - prob) * (0.3 if phase == "P" els
    e 0.6)
                newp = {"key": key, "code": s.code, "lat": s.lat, "lon": s.lon, "t": float(best
    .peak_time.timestamp),
                        "phase": phase, "sigma": sigma, "ai": True, "prob": prob}
                if old is not None:
                    ev["picks"].remove(old)
                ev["picks"].append(newp)
                changed = True
        return changed
```

sismo/alerts.py — 565 líneas

```
"""Alertas: Telegram (texto y PDF), correo electrónico y sonido en la computadora."""
import html
import os
import queue
import re
import smtplib
import ssl
import threading
import time
import urllib.parse
import urllib.request
import uuid
from email.message import EmailMessage

from .places import where_is
from .report import make_pdf
from . import ticket as tk
from .zones import zone_of
from .sonido import Sonidos


def _fmt(t):
    return time.strftime("%d-%m %H:%M:%S", time.gmtime(t - 6 * 3600))


class Alerts:
    def __init__(self, cfg, report_dir, log=print):
        self.cfg, self.report_dir, self.log = cfg, report_dir, log
        os.makedirs(report_dir, exist_ok=True)
        # un solo hilo de envío: los mensajes salen en el orden en que se generan
        # (el preliminar siempre antes que la confirmación)
        self.q = queue.Queue()
        self._usgs_primed = False   # v2.1.0: al arrancar no se reenvían confrontas de sismos y a cerrados
        threading.Thread(target=self._worker, daemon=True, name="alertas").start()
        self.sonidos = Sonidos(log)

    def _worker(self):
        while True:
            text, pdf = self.q.get()
            try:
                if callable(text):          # v2.1.0: el texto se arma en este hilo (consulta al USGS sin bloquear el motor)
                    text = text()
                # 1. Enviar el texto inmediatamente: el preliminar sale al instante sin esperar ningún PDF ni archivo
                if text:
                    self._send_text(text)
                # 2. Si hay PDF (solo en solución final), se genera y envía en un hilo desacoplado
                #    para que NUNCA detenga la salida de un nuevo Quórum Preliminar en cola
                if pdf:
                    def _async_pdf(p_func=pdf):
                        try:
                            pdf_file = p_func() if callable(p_func) else p_func
                            if pdf_file and os.path.exists(pdf_file):
                                self._send_pdf(pdf_file)
                        except Exception as ex:
                            self.log(f"Reporte PDF en segundo plano: {ex}")
                    threading.Thread(target=_async_pdf, daemon=True, name="alert-pdf").start()
            except Exception as e:
                self.log(f"Alertas: {e}")

    # ------------------------------------------------------------ criterios
    def zone(self, ev):
        """Primera zona de alerta (config.json → alertas.zonas, en orden) que contiene el epicentro."""
        return zone_of(self.cfg["alertas"].get("zonas") or [], ev["lat"], ev["lon"])

    def _should(self, engine, ev):
        a = self.cfg["alertas"]
        M = engine.mag_of(ev) or 0
        z = self.zone(ev)
        mmin = z.get("magnitud_minima", a.get("magnitud_minima", 6.0)) if z else a.get("magnitud_minima", 6.0)
        mmin_c = z.get("magnitud_calidad_c", mmin + 0.5) if z else a.get("magnitud_aviso_calidad_c", mmin + 0.5)
        ts = ev.get("tsunami") or {}
        # posible tsunami (nivel 2 o más): se avisa aunque no se alcance el umbral de la zona
        if a.get("alertar_tsunami", True) and ts.get("level", 0) >= 2:
            return True
        # una ML de una sola estación no basta para alertar (v0.4.0) a menos que sea preliminar temprano
        if ev.get("magWeak") and not ev.get("early"):
            return False
        # soluciones débiles (calidad C) solo se avisan si la magnitud es grande
        if engine.quality(ev) not in ("A", "B") and M < mmin_c:
            return False
        Imax = max([p["I"] for p in engine.poi_impact(ev)] or [0])
        return M >= mmin or Imax >= a["intensidad_minima"]

    def text(self, engine, ev, final):
        """Ticket técnico en formato de mensaje de Telegram (HTML). Título distinto para preliminar y confirmación."""
        t = tk.build(engine, ev)
```

```
        o, m = t["origin"], t["magnitude"]
        e = html.escape
        f = tk._f
        M = m["preferred"]
        if ev.get("early") and M is not None:
            mag = f"M {M:.1f}? {m['preferred_type']}"
        else:
            mag = f"M {M:.1f} {m['preferred_type']}" if M is not None else "M —"
        tsl = (ev.get("tsunami") or {}).get("level", 0)
        pre_info = ""
        if final:
            head = ["    <b>SISMO MUNDIAL — SOLUCIÓN FINAL (IIGEA-SIMMSA)</b>    ",
                    "<b>Solución consolidada del Monitor MUNDIAL (ondas P, S y superficiales)</b>"]
            p_str = ev.get("magPreStr") or (f"{ev['magPre']:.1f}" if ev.get("magPre") is not None else None)
            if p_str:
                org = ev.get("org", 0)
                tpre_sec = (ev.get("tEarly") or ev.get("tPrelim") or ev.get("created", 0)) - org
                tpre_min = tpre_sec / 60 if org and tpre_sec > 0 else None
                pre_info = f"\n<i>Preliminar temprano emitido:</i> <b>M {e(p_str)} {e(ev.get('magPreType') or '')}</b>" + (f" ({tpre_min:.1f} min tras el origen)" if tpre_min else "")
        elif ev.get("early"):
            head = ["    <b>SISMO MUNDIAL — PRELIMINAR TEMPRANO</b>    ",
                    "<i>Primera estimación automática ultra-temprana (quórum inicial de estaciones).</i>",
                    "   <i>En proceso de cálculo dinámico: magnitud y epicentro en refinamiento.</i>"]
        else:
            head = ["    <b>SISMO MUNDIAL — PRELIMINAR</b>    ",
                    "<i>Primera detección automática del Monitor MUNDIAL (ondas P). Puede actualizarse al llegar más fases.</i>"]
        if tsl >= 2:
            head.insert(1, f"  <b>{e(ev['tsunami']['title'])}</b> (informativo, no oficial)")
        status_label = ' (Final)' if final else (' (Preliminar Temprano)' if ev.get('early') else ' (Preliminar)')
        L = head + ["",
                    f"<b>{e(mag)}</b>{status_label} · {e(o['region'])}{pre_info}",
                    *( [f"Zona de alerta: {e(t['zone'])}"] if t.get("zone") else [] ),
                    *( [f"Región: {e(o['fe_region'])}"] if o.get("fe_region") and o["fe_region"] not in o["region"] else [] ),
                    f"<code>{e(t['ticket'])}</code>",
                    "",
                    "<b>ORIGEN</b>",
                    f"Hora CDMX: <b>{e(o['time_cdmx'])}</b>",
                    f"Hora UTC: {e(o['time_utc'])}",
                    f"Epicentro: {o['lat']:.4f}, {o['lon']:.4f}",
                    f"Profundidad: <b>{o['depth_km']:.0f} ± {f(o['err_depth_km'], 0)} km</b> · {e(o['depth_class'])}"
                    + (f" · {e(o['depth_constraint'])}" if o.get("depth_constraint") else ""),
                    ("Elipse 68 %: no acotada (&gt; 200 km)" if o["err_open"] else
                     f"Elipse 68 %: {f(o['err_h_major_km'])} × {f(o['err_h_minor_km'])} km, az {f(o['err_azimuth_deg'], 0)}°"),
                    f"Calidad <b>{e(str(o['quality']))}</b> · RMS {f(o['rms_s'], 2)} s · gap {f(o['gap_deg'], 0)}°",
                    f"Fases {o['n_P']} P / {o['n_S']} S" + (f" en {o['n_stations']} est. · {f((o['min_dist_km'] or 0) / 111.195, 1)}°–{f((o['max_dist_km'] or 0) / 111.195, 0)}°" if o["n_stations"] else ""),
                    f"Modelo: {e(o['velocity_model'])}",
                    "",
                    "<b>MAGNITUD</b>",
                    f"ML {f(m['ml'], 2)}" + (f" ± {f(m['ml_sd'], 2)}" if m["ml_sd"] is not None else "")
                    + (f" ({m['ml_n']} est., comp. {e(str(m['ml_components']))})" if m["ml_n"] else ""),
                    f"Mwp {f(m['mwp'], 2)}" + (f" ± {f(m.get('mwp_sd'), 2)}" if m.get("mwp_sd") is not None else "") + (f" ({m['mwp_n']} est.)" if m["mwp"] is not None else "Mwp: no calculada",
                    " · ".join(f"{lab} {f(m[k], 2)} ({m[k + '_n']} est.)" for lab, k in (("mb", "mb"), ("mB", "mB"), ("Ms", "ms")) if m.get(k) is not None) or "mb/mB/Ms: aún no",
                    (f"<b>Mw estimada {f(m['mw'], 1)}</b>" + (f" ± {f(m['mw_sd'], 1)}" if m.get("mw_sd") is not None else "")
                     + " (" + ", ".join(f"{k}→{v:.1f}" for k, v in m["mw_parts"].items()) + ")"
                     + (" · descartada: " + ", ".join(f"{k}→{v:.1f}" for k, v in m["mw_dropped"].items()) if m.get("mw_dropped") else "")) if m.get("mw") is not None else "Mw estimada: aún no"]
        if t["intensity_poi"]:
            L += ["", "<b>INTENSIDAD ESTIMADA</b>"]
            for p in t["intensity_poi"]:
                L.append(f"{e(p['name'])} ({e(p['soil'])}, {p['dist_km']:.0f} km): <b>{p['roman']}</b> ({p['mmi']:.1f}) · onda S a {p['s_travel_s']:.0f} s del origen")
        if t["nearest_cities"]:
            L += ["", "<b>CIUDADES CERCANAS</b>"]
            for c in t["nearest_cities"][:4]:
                L.append(f"{e(c['name'])}: {c['dist_km']:.0f} km · MMI {c['roman']} (terreno firme)")
        P = [p for p in t["picks"] if p["phase"] == "P"]
        if P:
            rows = [f"{'Estación':<10}{'Dist°':>6}{'Res s':>7}{'Mwp':>5}"]
            for p in sorted(P, key=lambda p: p["dist_km"])[:15]:
                rows.append(f"{p['station'][:10]:<10}{p['dist_deg']:>6.1f}{f(p['residual_s'], 2):>7}{f(p.get('mwp') if p.get('mwp') is not None else p['ml'], 1):>5}")
            if len(P) > 15:
                rows.append(f"… y {len(P) - 15} estaciones más")
            L += ["", "<b>ESTACIONES (onda P)</b>", "<pre>" + e("\n".join(rows)) + "</pre>"]
        d = t["derived"]
        if d:
```

```
            L += ["<b>PARÁMETROS DERIVADOS</b> (aprox.)",
                  f"M0 {d['moment_Nm']:.1e} N·m · energía {d['energy_J']:.1e} J · ruptura ≈ {d['rupture_length_km']:.0f} km"]
        ts = t.get("tsunami")
        if ts:
            icon = "    " if ts["level"] >= 2 else ""
            L += ["", f"{icon}<b>TSUNAMI (informativo): {e(ts['title'])}</b>", e(ts["text"]),
                  f"Criterio: M {ts['M_used']} · prof. {ts['depth_km']} km · tierra/mar: {e(ts['method'])}"]
            if ts.get("cities"):
                L.append("Costas dentro del radio: " + "; ".join(f"{e(c['name'])} {c['dist_km']} km (~{c['eta_min']} min)" for c in ts["cities"][:8]))
            L.append("<i>" + e(tk.TSUNAMI_NOTE) + "</i>")
        u = t["usgs"]
        L += ["", "<b>USGS</b>"]
        if u:
            L.append(f"M{f(u['mag'])} {e(u['magType'] or '')} · {e(u['place'] or '')}")
            L.append(f"Dif.: epicentro {f(u['diff_epicenter_km'], 0)} km · prof. {f(u['diff_depth_km'], 0)} km · mag {f(u['diff_mag'], 2)}")
        else:
            L.append("Aún sin evento USGS correspondiente.")
        L += ["", "<i>" + ("Solución final = cálculo consolidado interno del Monitor MUNDIAL (ondas P, S y superficiales). " if final else "")
                + ("La Ms llega con la onda Rayleigh, hasta 1 h después; el ticket final la incluye. " if final else "")
                + "Detección automática experimental con la red sismológica global; información oficial: USGS y agencias nacionales.</i>"]
        return "\n".join(L)

    @staticmethod
    def plain(text):
        """El mismo texto sin formato HTML (para correo)."""
        return html.unescape(re.sub(r"<[^>]+>", "", text))

    def event(self, engine, ev, final):
        a = self.cfg["alertas"]
        if not final and (ev.get("final") or ev.get("sentFinal")):
            return  # ya hay solución final: no se manda un preliminar después de la confirmación
        self.sound_check(engine, ev, final)
        if not self._should(engine, ev):
            return
        flag = "sentFinal" if final else "sentFirst"
        if ev.get(flag):
            return
        ev[flag] = True
        snap = dict(ev, picks=[dict(p) for p in ev.get("picks", [])])
        build_text = lambda snap=snap, final=final: self.text(engine, snap, final)
        build_pdf = None
        if final and a.get("adjuntar_pdf", True):
            def build_pdf(snap=snap):
                try:
                    return make_pdf(engine, snap, os.path.join(self.report_dir, f"reporte_{time.strftime('%Y%m%d_%H%M%S', time.gmtime(snap['org']))}.pdf"))
                except Exception as e:
                    self.log(f"Reporte PDF: {e}")
                    return None
        self.log(f"Alerta {'CONFIRMACIÓN' if final else ('PRELIMINAR TEMPRANO' if ev.get('early') else 'PRELIMINAR')} en cola: {engine.mag_str(ev)} {where_is(ev['lat'], ev['lon'])}")
        self.q.put((build_text, build_pdf))

    def sound_check(self, engine, ev, final):
        """Sonido en la PC (v0.3.9). Una sola vez por nivel y por sismo:
        - PRELIMINAR (alarma EAS): magnitud ≥ umbral de su zona y zona con sonido.
        - SOLUCIÓN FINAL (desfibrilador): al cerrar un sismo que sonó o que llega al umbral de su zona.
        - M ≥ 7.5 (alarma severa) y M ≥ 8.5 (alarma de seguridad): en cualquier zona, también si la
          magnitud sube después del primer aviso."""
        a = self.cfg["alertas"]
        if not a.get("sonido", True):
            return
        M = engine.mag_of(ev)
        if M is None or ev.get("magWeak"):
            return
        zn = self.zone(ev) or {}
        zmin = zn.get("magnitud_minima", a.get("magnitud_minima", 6.0))
        zsnd = zn.get("sonido", True)
        big = 3 if M >= a.get("magnitud_alarma_maxima", 8.5) else 2 if M >= a.get("magnitud_alarma", 7.5) else 0
        done = ev.get("_sndLvl", 0)
        if big > done:
            ev["_sndLvl"] = big
            if final:
                ev["_sndFinal"] = True
            self.sonidos.play("alarma_8_5" if big == 3 else "alarma_7_5")
            self.log(f"Sonido: alarma M ≥ {'8.5' if big == 3 else '7.5'} ({engine.mag_str(ev)} {where_is(ev['lat'], ev['lon'])})")
            return
        if not final:
            if done == 0 and zsnd and M >= zmin:
                ev["_sndLvl"] = 1
                self.sonidos.play("preliminar")
        elif not ev.get("_sndFinal") and (done >= 1 or (zsnd and M >= zmin)):
            ev["_sndFinal"] = True
```

```
            self.sonidos.play("final")

    # ------------------------------------------------------------ v2.1.0: confronta Monitor vs USGS
    def usgs_text(self, snap):
        """3er mensaje: SISMO CONFIRMADO POR USGS — confronta del monitor contra el USGS y quién confirmó primero."""
        from .usgsreport import _usgs_info
        e, u = html.escape, snap["match"]
        org = snap["org"]
        info = {}
        try:
            info = _usgs_info([(u["id"], org)], os.path.dirname(self.report_dir), self.log).get(u["id"], {}) or {}
        except Exception as ex:
            self.log(f"Confronta USGS: {ex}")
        mu = info.get("mag", u.get("mag"))
        tu = info.get("magType", u.get("magType")) or ""
        pub = info.get("pub")
        n = lambda v, d=1: "—" if v is None else f"{v:.{d}f}"
        sg = lambda v, d=1: "—" if v is None else f"{v:+.{d}f}"
        mm, mt = snap.get("mag"), snap.get("magType") or ""
        L = ["    <b>SISMO CONFIRMADO POR USGS</b>    ",
             f"<b>{e(snap['region'])}</b>",
             f"Hora CDMX: {e(_fmt(org))}", "",
             "<b>CONFRONTA MONITOR vs USGS</b>",
             f"Magnitud: monitor <b>{n(mm)} {e(mt)}</b> · USGS <b>{n(mu)} {e(tu)}</b> → dif. {sg(mm - mu if mm is not None and mu is not None else None)}"]
        L += [f"Epicentro: diferencia <b>{n(u.get('dEpi'), 0)} km</b>",
              f"Profundidad: monitor {n(snap['z'], 0)} km · USGS {n(u.get('z'), 0)} km → dif. {sg(u.get('dZ'), 0)} km",
              f"Hora de origen: dif. {sg(u.get('dT'), 1)} s", ""]
        t_pre_sec = snap.get("tEarly") or snap.get("tPrelim") or snap.get("created")
        tpre = (t_pre_sec - org) / 60 if t_pre_sec and org else None
        tfin = (snap["tFinal"] - org) / 60 if snap.get("tFinal") and org else None
        tusg = (pub - org) / 60 if pub and org else None
        primera_tipo = "Preliminar temprano" if snap.get("early") else "Preliminar"
        L += ["<b>TIEMPOS (minutos después del sismo)</b>",
              f"Primera alerta monitor ({primera_tipo}): <b>{n(tpre, 1)} min</b>",
              f"Primera publicación USGS: <b>{n(tusg, 1)} min</b>",
              f"Solución final monitor (consolidada): {n(tfin, 1)} min", ""]
        if tpre is not None and tusg is not None:
            dif = tusg - tpre
            if abs(dif) < 0.5:
                L.append("  <b>Alertaron prácticamente al mismo tiempo</b> (< 30 s de diferencia)")
            elif dif > 0:
                L.append(f"  <b>Alertó primero: MONITOR MUNDIAL</b> ({dif:.1f} min antes que el USGS)")
            else:
                L.append(f"  <b>Publicó primero: USGS</b> ({-dif:.1f} min antes que el monitor)")
            if tfin is not None:
                if tfin < tusg:
                    L.append(f"   El monitor completó su solución final antes de que el USGS publicara ({tusg - tfin:.1f} min antes).")
                else:
                    L.append(f"   La solución final del monitor consolidó todas las fases a los {tfin:.1f} min.")
        elif tfin is not None and tusg is not None:
            dif = tusg - tfin
            if dif > 0:
                L.append(f"  <b>Solución final lista antes que el USGS</b> ({dif:.1f} min antes)")
            else:
                L.append(f"   El USGS publicó a los {tusg:.1f} min; el monitor finalizó a los {tfin:.1f} min")
        else:
            L.append("  No se pudo comparar la rapidez de alerta (falta la hora de publicación del USGS)")
        L += ["", f"USGS {e(u.get('id', ''))}: {e(u.get('place') or '')}"]
        if u.get("url"):
            L.append(e(u["url"]))
        return "\n".join(L)

    def _usgs_due(self, engine):
        """Sismos ya avisados por Telegram, con solución final y emparejados con el USGS, aún sin confronta."""
        a = self.cfg["alertas"]
        if not a.get("alertar_confirmacion_usgs", True):
            return
        now = time.time()
        first = not self._usgs_primed
        self._usgs_primed = True
        for ev in engine.events[:30]:
            if ev.get("sentUsgs") or not (ev.get("sentFirst") or ev.get("sentFinal")):
                continue
            if not (ev.get("final") and (ev.get("match") or {}).get("mag") is not None):
                continue
            ev["sentUsgs"] = True
            engine.dirty.add(ev["id"])
            if first or now - ev["org"] > 12 * 3600:
                continue          # sismos que ya estaban cerrados y confirmados al arrancar: no se reenvían
            ds, lab = None, None
```

```
            try:
                ds = engine._dm_same(ev)
                lab = ev["match"].get("dMtype")
            except Exception:
                pass
            snap = {"org": ev["org"], "created": ev.get("created"), "tPrelim": ev.get("tPrelim"), "tFinal": ev.get("tFinal"), "z": ev.get("z"),
                    "mag": engine.mag_of(ev), "magType": ev.get("magType"), "region": where_is(ev["lat"], ev["lon"]),
                    "match": dict(ev["match"]), "dSame": ds, "dSameLab": lab}
            self.log(f"Alerta CONFIRMADO POR USGS en cola: {engine.mag_str(ev)} {snap['region']} ({ev['match'].get('id')})")
            self.q.put((lambda snap=snap: self.usgs_text(snap), None))

  def poll(self, engine):
      """Revisión periódica: manda el PRELIMINAR en cuanto el sismo tiene magnitud
      (la magnitud puede calcularse unos segundos después de la detección) y revisa el sonido
      de los sismos en curso por si la magnitud sube a 7.5 u 8.5."""
      with engine.lock:
          for ev in engine.events[:30]:
              if ev.get("live") and not ev.get("final") and not ev.get("sentFinal"):
                  self.sound_check(engine, ev, False)
              if ev.get("live") and not ev.get("final") and not ev.get("sentFirst") and engine.mag_of(ev) is not None:
                  self.event(engine, ev, final=False)
          try:
              self._usgs_due(engine)
          except Exception as e:
              self.log(f"Confronta USGS: {e}")

  # ------------------------------------------------------------ v2.1.0: varias cuentas de Telegram
  def tg_accounts(self):
      """Cuenta principal (alertas.telegram) + alertas.telegram.cuentas_adicionales (cada una con su bot y chat)."""
      tg = self.cfg["alertas"]["telegram"]
      if not tg.get("activo"):
          return []
      out = []
      if tg.get("token") and tg.get("chat_id"):
          out.append({"nombre": tg.get("nombre") or "principal", "token": tg["token"], "chat_id": tg["chat_id"]})
      for i, c in enumerate(tg.get("cuentas_adicionales") or []):
          if c.get("activo", True) and c.get("token"):
              out.append({"nombre": c.get("nombre") or f"cuenta {i + 2}", "token": c["token"], "chat_id": c.get("chat_id")})
      return out

  def _chat_of(self, acct):
      """chat_id de una cuenta. Si falta, es «auto» o es el número del propio bot (un bot no puede escribirse a sí
      mismo), se busca el chat privado de quien abrió el bot y presionó «Iniciar» (getUpdates) y se guarda en
      datos/telegram_chats.json."""
      import json
      cid = str(acct.get("chat_id") or "").strip()
      bot_id = acct["token"].split(":")[0]
      if cid and cid.lower() != "auto" and cid != bot_id:
          return cid
      path = os.path.join(os.path.dirname(self.report_dir), "telegram_chats.json")
      try:
          with open(path, encoding="utf-8") as f:
              saved = json.load(f)
      except Exception:
          saved = {}
      if saved.get(bot_id):
          return saved[bot_id]
      r = json.loads(urllib.request.urlopen(f"https://api.telegram.org/bot{acct['token']}/getUpdates", timeout=20).read())
      chats = []
      for u in r.get("result") or []:
          c = ((u.get("message") or u.get("edited_message") or {}).get("chat") or {})
          if c.get("type") == "private" and c.get("id") and c["id"] not in chats:
              chats.append(c["id"])
      if not chats:
          raise RuntimeError(f"la cuenta «{acct['nombre']}» aún no tiene chat: abre su bot en Telegram y presiona «Iniciar»")
      saved[bot_id] = str(chats[0])
      try:
          with open(path, "w", encoding="utf-8") as f:
              json.dump(saved, f)
      except OSError:
          pass
      self.log(f"Telegram «{acct['nombre']}»: chat encontrado y guardado ({len(chats)} chat(s) privado(s); se usa el primero)")
      return saved[bot_id]

  def _tg_each(self, fn):
      """Ejecuta fn(cuenta) en cada cuenta; una cuenta que falla no impide el envío a las demás."""
      hl = getattr(self, "health", None)
      oks, errs = [], []
      for acct in self.tg_accounts():
          try:
              acct = dict(acct, chat_id=self._chat_of(acct))
```

```
                fn(acct)
                oks.append(acct["nombre"])
            except Exception as e:
                errs.append(f"{acct['nombre']}: {e}")
                self.log(f"Telegram «{acct['nombre']}»: {e}")
        if hl:
            if errs:
                hl.fail("telegram", "; ".join(errs) + (f" (sí llegó a: {', '.join(oks)})" if ok
s else ""))
            elif oks:
                hl.beat("telegram")
        return oks, errs

    def _send_text(self, text):
        def send(acct):
            self.telegram(text, html_mode=True, acct=acct)
        self._tg_each(send)
        em = self.cfg["alertas"]["correo"]
        if em.get("activo") and em.get("servidor") and em.get("para"):
            try:
                self.email(self.plain(text), None)
                hl = getattr(self, "health", None)
                hl and hl.beat("correo")
            except Exception as e:
                self.log(f"Correo: {e}")
                hl = getattr(self, "health", None)
                hl and hl.fail("correo", e)

    def _send_pdf(self, pdf):
        def send(acct):
            self.telegram_file(pdf, acct=acct)
        self._tg_each(send)

    def _send_all(self, text, pdf):
        self._send_text(text)
        if pdf:
            self._send_pdf(pdf)

    # ------------------------------------------------------------ canales
    def telegram(self, text, html_mode=False, acct=None):
        tg = acct or self.cfg["alertas"]["telegram"]
        url = f"https://api.telegram.org/bot{tg['token']}/sendMessage"
        parts = self._split(text, 3900)
        for part in parts:
            fields = {"chat_id": tg["chat_id"], "text": part, "disable_web_page_preview": "true
"}
            if html_mode:
                fields["parse_mode"] = "HTML"
            try:
                urllib.request.urlopen(url, data=urllib.parse.urlencode(fields).encode(), timeo
ut=20).read()
            except Exception:
                if not html_mode:
                    raise
                # si Telegram rechaza el formato, se reenvía el mismo contenido sin formato
                fields.pop("parse_mode", None)
                fields["text"] = self.plain(part)
                urllib.request.urlopen(url, data=urllib.parse.urlencode(fields).encode(), timeo
ut=20).read()

    @staticmethod
    def _split(text, n):
        """Telegram acepta hasta 4096 caracteres por mensaje: se corta por bloques sin partir e
tiquetas."""
        if len(text) <= n:
            return [text]
        out, cur = [], ""
        for block in text.split("\n\n"):
            if cur and len(cur) + len(block) + 2 > n:
                out.append(cur)
                cur = block
            else:
                cur = f"{cur}\n\n{block}" if cur else block
        if cur:
            out.append(cur)
        return out

    def telegram_file(self, path, acct=None):
        tg = acct or self.cfg["alertas"]["telegram"]
        boundary = uuid.uuid4().hex
        with open(path, "rb") as f:
            content = f.read()
        body = (f"--{boundary}\r\nContent-Disposition: form-data; name=\"chat_id\"\r\n\r\n{tg['
chat_id']}\r\n"
                f"--{boundary}\r\nContent-Disposition: form-data; name=\"document\"; filename=\
"{os.path.basename(path)}\"\r\n"
                f"Content-Type: application/pdf\r\n\r\n").encode() + content + f"\r\n--{boundar
y}--\r\n".encode()
        req = urllib.request.Request(f"https://api.telegram.org/bot{tg['token']}/sendDocument",
 data=body,
                                     headers={"Content-Type": f"multipart/form-data; boundary={
boundary}"})
        urllib.request.urlopen(req, timeout=60).read()

    def email(self, text, pdf=None):
```

```
        em = self.cfg["alertas"]["correo"]
        msg = EmailMessage()
        lines = [l.strip() for l in text.splitlines() if l.strip()]
        msg["Subject"] = " · ".join(lines[:3:2] if len(lines) > 2 else lines[:2]).replace(" ", "").replace(" ", "").replace(" ", "").strip()
        msg["From"] = em.get("de") or em["usuario"]
        msg["To"] = ", ".join(em["para"]) if isinstance(em["para"], list) else em["para"]
        msg.set_content(text)
        if pdf:
            with open(pdf, "rb") as f:
                msg.add_attachment(f.read(), maintype="application", subtype="pdf", filename=os.path.basename(pdf))
        with smtplib.SMTP(em["servidor"], int(em.get("puerto", 587)), timeout=30) as s:
            s.starttls(context=ssl.create_default_context())
            if em.get("usuario"):
                s.login(em["usuario"], em["contrasena"])
            s.send_message(msg)

    @staticmethod
    def _beep():
        try:
            import winsound
            for _ in range(3):
                winsound.Beep(880, 400)
                time.sleep(0.15)
        except Exception:
            pass

    def test(self, engine=None):
        """Envía un mensaje de prueba por los canales activos. Si ya hay sismos en el historial,
        manda además el formato PRELIMINAR y el de CONFIRMACIÓN del más reciente, marcados como prueba."""
        res = []
        tg, em = self.cfg["alertas"]["telegram"], self.cfg["alertas"]["correo"]
        samples, usnap = [], None
        if engine is not None:
            with engine.lock:
                ev = engine.events[0] if engine.events else None
                if ev is not None:
                    note = "  <b>PRUEBA DE FORMATO</b> — sismo del historial, no es un sismo nuevo\n\n"
                else:
                    # historial vacío: sismo de demostración con datos ficticios, solo para ver el formato
                    now = time.time()
                    ev = {"id": "demostracion", "created": now, "org": now - 600, "lat": -33.0, "lon": -72.0, "z": 30.0,
                          "errH": 14.0, "errMin": 7.0, "errAz": 20.0, "errZ": 14.0, "errOpen": False, "rms": 1.0, "gap": 25.0,
                          "nP": 60, "nS": 4, "mwp": 7.0, "mwpN": 45, "mag": 7.0, "magType": "Mwp",
                          "picks": [], "live": False, "final": True}
                    note = "  <b>PRUEBA DE FORMATO</b> — DATOS FICTICIOS, no es un sismo real\n\n"
                try:
                    samples = [note + self.text(engine, ev, False), note + self.text(engine, ev, True)]
                    if (ev.get("match") or {}).get("mag") is not None:
                        usnap = {"org": ev["org"], "created": ev.get("created"), "tFinal": ev.get("tFinal"),
                                 "z": ev.get("z"), "mag": engine.mag_of(ev), "magType": ev.get("magType"),
                                 "region": where_is(ev["lat"], ev["lon"]), "match": dict(ev["match"]),
                                 "dSame": engine._dm_same(ev), "dSameLab": ev["match"].get("dMtype"), "note": note}
                except Exception as e:
                    self.log(f"Prueba de formato: {e}")
        if usnap:      # fuera del candado del motor: consulta la hora de publicación del USGS
            try:
                samples.append(usnap["note"] + self.usgs_text(usnap))
            except Exception as e:
                self.log(f"Prueba de formato (USGS): {e}")
        if tg.get("activo"):
            def send(acct):
                self.telegram(f"Prueba del Monitor Sísmico MUNDIAL ({acct['nombre']}): las alertas por Telegram funcionan.", acct=acct)
                for t in samples:
                    self.telegram(t, html_mode=True, acct=acct)
            oks, errs = self._tg_each(send)
            extra = (" (con ejemplo preliminar, confirmación" + (" y confirmado por USGS)" if len(samples) > 2 else ")")) if samples else ""
            for n in oks:
                res.append(f"Telegram «{n}»: enviado{extra}")
            for e in errs:
                res.append(f"Telegram {e} (error)")
        if em.get("activo"):
            try:
                self.email("Prueba del Monitor Sísmico MUNDIAL\nLas alertas por correo funcionan.")
                res.append("Correo: enviado")
                getattr(self, "health", None) and self.health.beat("correo")
            except Exception as e:
                res.append(f"Correo: error ({e})")
```

```
                getattr(self, "health", None) and self.health.fail("correo", e)
        return res or ["No hay canales activos en config.json"]
```

sismo/associator.py — 187 líneas

```
"""Asociador GLOBAL de arribos P (nucleación en dos etapas sobre la esfera).

Etapa A — malla global de Fibonacci (~2°) a 4 profundidades (20, 120, 350, 600 km) con tiempos de
viaje P precalculados nodo-estación. Se toma como ancla el arribo más reciente; para cada nodo se
cuentan las estaciones cuyo origen implícito coincide con el del ancla dentro de una tolerancia que
crece con la lentitud (el nodo puede estar hasta ~1.2° del epicentro real) y con el tiempo de viaje.

Etapa B — malla fina (0.25°, 8 profundidades) alrededor de los mejores nodos, con tolerancia
estrecha. Devuelve el mayor grupo de estaciones distintas con un origen común y la posición
aproximada, que el localizador usa como punto de partida.
"""
import numpy as np

from .locator import dist_km, fibonacci_nodes, wrap_lon, RAD

NODE_DEG = 2.0
DEPTHS_A = (20.0, 120.0, 350.0, 600.0)
DEPTHS_B = (10.0, 35.0, 70.0, 120.0, 200.0, 300.0, 450.0, 600.0)
DELTA_A = 0.62 * NODE_DEG * 111.2          # distancia máxima (km) entre el evento y el nodo más cercano
DELTA_B = 0.18 * 111.2


class Associator:
    def __init__(self, tt, min_sta=8):
        self.tt = tt
        self.min_sta = min_sta
        self.max_groups = 5
        self.keys = None
        self.nlat, self.nlon = fibonacci_nodes(NODE_DEG)

    def build(self, stations):
        """stations: dict key -> objeto con lat, lon. Se recalcula solo si cambia la lista."""
        keys = sorted(stations)
        if keys == self.keys:
            return
        slat = np.array([stations[k].lat for k in keys])
        slon = np.array([stations[k].lon for k in keys])
        D = dist_km(self.nlat[:, None], self.nlon[:, None], slat[None, :], slon[None, :])
        T, U = [], []
        for z in DEPTHS_A:
            t0 = self.tt.tt("P", D, z)
            t1 = self.tt.tt("P", D + 25.0, z)
            T.append(t0.astype(np.float32))
            U.append(np.abs(t1 - t0).astype(np.float16) / np.float16(25.0))
        self.T = np.vstack(T)                      # (4·N, S)
        self.U = np.vstack(U)
        self.H = np.ascontiguousarray(0.006 * self.T + DELTA_A * self.U.astype(np.float32), dtype=np.float32)
        self.S0 = np.ascontiguousarray(0.003 * self.T + 0.4 * DELTA_A * self.U.astype(np.float32), dtype=np.float32)
        self.node_lat = np.tile(self.nlat, len(DEPTHS_A))
        self.node_lon = np.tile(self.nlon, len(DEPTHS_A))
        self.node_z = np.repeat(np.array(DEPTHS_A), len(self.nlat))
        self.keys = keys
        self.idx = {k: i for i, k in enumerate(keys)}

    # ------------------------------------------------------------------
    def nucleate(self, cand, anchor, min_seed=None):
        """cand: picks P sin asignar (incluye al ancla). Devuelve una lista de hasta max_groups
        candidatos [(grupo, (lat, lon, z)), ...], del mejor calificado al peor."""
        cand = [q for q in cand if q["key"] in self.idx]
        if min_seed is None:
            min_seed = max(5, self.min_sta - 3)
        if anchor not in cand or len({q["key"] for q in cand}) < min_seed:
            return []
        if getattr(self, "H", None) is None or self.H.shape != self.T.shape:
            self.H = np.ascontiguousarray(0.006 * self.T + DELTA_A * self.U.astype(np.float32), dtype=np.float32)
            self.S0 = np.ascontiguousarray(0.003 * self.T + 0.4 * DELTA_A * self.U.astype(np.float32), dtype=np.float32)
        a = cand.index(anchor)
        si = np.array([self.idx[q["key"]] for q in cand])
        sa = si[a]
        t = np.array([q["t"] for q in cand])
        dt_t = (t - t[a]).astype(np.float32)
        keys = np.array([q["key"] for q in cand])
        uniq, inv = np.unique(keys, return_inverse=True)
        dup = len(uniq) < len(keys)
        M = self.T.shape[0]
        Ta = self.T[:, sa:sa + 1]
        Ua = self.U[:, sa:sa + 1].astype(np.float32)
        tol_a = (2.5 + DELTA_A * Ua).astype(np.float32)
        sig_a = (1.0 + 0.4 * DELTA_A * Ua).astype(np.float32)
        score = np.full(M, -1.0, dtype=np.float32)
        chunk = 2048
        for i0 in range(0, M, chunk):
            i1 = min(M, i0 + chunk)
            Tn_c = self.T[i0:i1, si]
            dO_c = dt_t[None, :] - (Tn_c - Ta[i0:i1])
            inw_c = np.abs(dO_c) <= tol_a[i0:i1] + self.H[i0:i1, si]
```

```
            n_c = inw_c.sum(axis=1)
            near_c = (inw_c & (Tn_c <= 250.0)).sum(axis=1)
            rows_c = np.nonzero((n_c >= self.min_sta) | ((near_c >= min_seed) & (Ta[i0:i1, 0] <
= 350.0)) | ((n_c >= min_seed) & (Ta[i0:i1, 0] <= 450.0)))[0]
            if not len(rows_c):
                continue
            r_idx, c_idx = np.nonzero(inw_c[rows_c])
            row_glob = i0 + rows_c[r_idx]
            col_sta = si[c_idx]
            sig_v = sig_a[row_glob, 0] + self.S0[row_glob, col_sta]
            dO_v = dO_c[rows_c[r_idx], c_idx]
            w_v = np.exp(-0.5 * (dO_v / sig_v) ** 2) / sig_v
            if dup:
                ws = np.zeros((len(rows_c), len(uniq)), np.float32)
                np.maximum.at(ws, (r_idx, inv[c_idx]), w_v)
                nr = (ws > 0).sum(axis=1)
                sc = ws.sum(axis=1)
            else:
                nr = n_c[rows_c]
                sc = np.bincount(r_idx, weights=w_v, minlength=len(rows_c))
            ok_row = (nr >= self.min_sta) | ((nr >= min_seed) & ((near_c[rows_c] >= min_seed) |
 (Ta[row_glob_uniq := i0 + rows_c, 0] <= 450.0)) & (sc >= 2.0))
            score[i0 + rows_c] = np.where(ok_row, sc, -1.0)
        if score.max() < 0:
            return []
        n = score
        order = np.argsort(-n)[:400]
        found = []
        tried = []
        for r in order:
            if n[r] < 0 or len(tried) >= 24:
                break
            lat0, lon0 = self.node_lat[r], self.node_lon[r]
            if tried and float(np.min(dist_km(lat0, lon0, np.array([x[0] for x in tried]), np.a
rray([x[1] for x in tried])))) < 250:
                continue
            tried.append((lat0, lon0))
            dO_r = dt_t - (self.T[r, si] - Ta[r, 0])
            inw_r = np.abs(dO_r) <= tol_a[r, 0] + self.H[r, si]
            sub = [cand[j] for j in np.nonzero(inw_r)[0]]
            if anchor not in sub:
                sub.append(anchor)
            g, pos, score = self._fine(sub, anchor, lat0, lon0)
            if g is not None:
                found.append((score, len(g), g, pos))
        # varios candidatos (distintos entre sí); el motor los localiza y se queda con el que e
s congruente
        found.sort(key=lambda x: (x[0], x[1]), reverse=True)
        out = []
        for _, _, g, pos in found:
            if all(float(dist_km(pos[0], pos[1], p2[0], p2[1])) > 300 for _, p2 in out):
                out.append((g, pos))
            if len(out) >= self.max_groups:
                break
        return out

    def _fine(self, sub, anchor, lat0, lon0):
        """Malla fina alrededor de un nodo. Cada nodo se califica como una verosimilitud: cada
estación
        suma exp(-½(ΔO/σ)²)/σ, donde ΔO es la diferencia entre su origen implícito y el del anc
la y σ
        el error esperado (pick + modelo + distancia al nodo). Un sismo real, en el nodo correc
to,
        suma casi 1/σ por estación; las coincidencias casuales con tolerancias amplias suman po
co."""
        span, step = 1.5, 0.25
        lats = np.arange(lat0 - span, lat0 + span + 1e-9, step)
        lats = lats[np.abs(lats) < 89.9]
        c = max(0.15, np.cos(lat0 * RAD))
        lons = np.arange(-span / c, span / c + 1e-9, step / c)
        LA, LO = np.meshgrid(lats, lons, indexing="ij")
        la, lo = LA.ravel(), wrap_lon(lon0 + LO.ravel())
        plat = np.array([q["lat"] for q in sub])
        plon = np.array([q["lon"] for q in sub])
        t = np.array([q["t"] for q in sub])
        a = sub.index(anchor)
        D = dist_km(la[:, None], lo[:, None], plat[None, :], plon[None, :])
        keys = [q["key"] for q in sub]
        uniq, inv = np.unique(keys, return_inverse=True)
        best = (None, None, -1.0)
        for z in DEPTHS_B:
            Tn = self.tt.tt("P", D, z)
            Un = np.abs(self.tt.tt("P", D + 25.0, z) - Tn) / 25.0
            O = t[None, :] - Tn
            dO = O - O[:, a:a + 1]
            sig = 0.8 + 0.002 * Tn + 0.5 * DELTA_B * (Un + Un[:, a:a + 1])
            w = np.exp(-0.5 * (dO / sig) ** 2) / sig
            if len(uniq) < len(keys):
                ws = np.zeros((w.shape[0], len(uniq)))
                np.maximum.at(ws.T, inv, w.T)
                score = ws.sum(axis=1)
            else:
                score = w.sum(axis=1)
            r = int(np.argmax(score))
            if score[r] <= best[2]:
```

```
                continue
            ok = np.abs(dO[r]) <= 2.5 * sig[r]
            pick_of = {}
            for j in np.nonzero(ok)[0]:
                k = keys[j]
                if k not in pick_of or abs(dO[r, j]) < abs(dO[r, pick_of[k]]):
                    pick_of[k] = j
            js = list(pick_of.values())
            min_seed = max(5, self.min_sta - 3)
            if len(js) < self.min_sta:
                if len(js) < min_seed or score[r] < 2.2 or int((Tn[r, js] <= 250.0).sum()) < mi
n_seed:
                    continue
            best = ([sub[j] for j in js], (float(la[r]), float(lo[r]), float(z)), float(score[r
]))
        return best[0], best[1], best[2]
```

sismo/calibracion.py — 194 líneas

```
"""Calibración automática de las magnitudes contra el catálogo del USGS.

1) Q(Δ,h) de mb y mB: cuando un sismo confirmado tiene mb en el USGS, cada estación aporta el residuo
   (mb_USGS − mb_estación) a su intervalo de distancia (10°) y profundidad (< 70, 70–300, > 300 km).
   La corrección del intervalo es la media de los residuos, "encogida" hacia 0 con pocos datos:
   corr = Σr / (n + 5). Así nunca salta con uno o dos sismos.
2) Conversión a Mw: con cada sismo que el USGS reporta en Mw (Mww, Mwc, Mwb, Mwr, Mi…) se guarda la
   pareja (magnitud del monitor, Mw del USGS) por tipo. Con 15 parejas o más se ajusta una recta
   robusta (mínimos cuadrados con exclusión iterativa de residuos > 0.5) que sustituye a la relación
   publicada por omisión.
3) v2.0.0 — ajuste temprano: con 6 a 14 parejas (antes no se corregía nada) se aplica a la relación publicada
   un desplazamiento robusto = mediana(Mw_USGS − Mw_publicada) · n/(n+4); desde 15 parejas, la recta ajustada.
4) v2.0.0 — sesgo del mb del sismo: con cada sismo que el USGS reporta en mb (y que el monitor midió con 5 o más
   estaciones) se guarda mb_USGS − mb_monitor (sin este ajuste); el mb del sismo se corrige con
   mediana · n/(n+5) (desde 5 sismos; últimos 60).
Todo se guarda en datos/calibracion.json y se puede borrar para volver a los valores de la literatura.
"""
import time

import numpy as np

MW_TYPES = ("mww", "mwc", "mwb", "mwr", "mw", "mi", "mwp_usgs")


def _zcls(z):
    return "s" if z < 70 else "i" if z < 300 else "p"


def _bin(delta_deg, z):
    return f"{int(delta_deg // 10) * 10}_{_zcls(z)}"


class Calibration:
    def __init__(self, storage=None):
        self.storage = storage
        d = storage.load_json("calibracion.json", {}) if storage else {}
        self.q = {"mb": d.get("q_mb", {}), "mB": d.get("q_mB", {}), "mbR": d.get("q_mbR", {})}
        self.pairs = d.get("pares", {})          # tipo -> [[m_monitor, mw_usgs, t], ...]
        self.events = set(d.get("eventos", []))
        self.mb_res = d.get("mb_evento", [])     # [[mb_USGS − mb_monitor, t], ...]
        self.fit = {}
        self.off = {}
        self._refit()

    # ------------------------------------------------------------------ uso
    def qcorr(self, kind, delta_deg, z):
        c = self.q.get(kind, {}).get(_bin(delta_deg, z))
        return (c[0] / (c[1] + 5)) if c else 0.0

    def mw_calib(self):
        """{tipo: (a, b, n)} para magglobal.to_mw, más "_off": {tipo: desplazamiento} (ajuste temprano v2.0.0)."""
        return dict(self.fit, _off=dict(self.off))

    def mb_offset(self):
        """v2.0.0: corrección del sesgo del mb del sismo frente al mb del USGS."""
        k = len(self.mb_res)
        if k < 1:
            return 0.0
        return float(np.median([r[0] for r in self.mb_res])) * k / (k + 5)

    def mb_raw_bias(self):
        """Mediana directa de (mb_USGS − mb_monitor) en los sismos confirmados."""
        if not self.mb_res:
            return 0.0
        return float(np.median([r[0] for r in self.mb_res]))

    # ------------------------------------------------------------ aprendizaje
    def learn(self, ev):
        """Aprende de un sismo confirmado por el USGS (una sola vez por sismo). Devuelve True si cambió."""
        m = ev.get("match")
        if not m or m.get("mag") is None or ev["id"] in self.events:
            return False
        if time.time() - m["t"] < 3600:
            return False                              # la magnitud del USGS se revisa en la primera hora
        self.events.add(ev["id"])
        typ = (m.get("magType") or "").lower()
        changed = False
        if typ == "mb":
            for p in ev.get("picks", []):
                if p.get("phase") != "P":
                    continue
                for kind, key in (("mb", "mb_sta"), ("mB", "mB_sta"), ("mbR", "mbR_sta")):
                    v = p.get(key)
```

```
                    if v is None or p.get("dist_deg") is None:
                        continue
                    r = float(m["mag"]) - v
                    if abs(r) > 1.0:
                        continue
                    b = _bin(p["dist_deg"], ev["z"])
                    c = self.q[kind].setdefault(b, [0.0, 0])
                    c[0] += r
                    c[1] += 1
                    if c[1] > 200:                      # memoria finita: los datos viejos pesa
n menos
                        c[0] *= 200 / c[1]
                        c[1] = 200
                    changed = True
        if typ == "mb":
            mbr = ev.get("mbRaw", ev.get("mb"))
            if mbr is not None and (ev.get("mbN") or 0) >= 5 and abs(float(m["mag"]) - mbr) < 1
.0:
                self.mb_res.append([float(m["mag"]) - float(mbr), time.time()])
                self.mb_res = self.mb_res[-60:]
                changed = True
        if typ in MW_TYPES:
            for kind, key in (("Mwp", "mwp"), ("mB", "mB"), ("mb", "mbRaw"), ("Ms", "ms")):
                if key == "mbRaw" and ev.get(key) is None:
                    key = "mb"
                if ev.get(key) is not None:
                    self.pairs.setdefault(kind, []).append([float(ev[key]), float(m["mag"]), ti
me.time()])
                    self.pairs[kind] = self.pairs[kind][-500:]
                    changed = True
            self._refit()
        if changed:
            self.save()
        return changed

    def seed_mb(self, events):
        """v2.0.0: la primera vez, el sesgo del mb se inicia con los sismos ya confirmados que
tienen mb del USGS
        (los mismos que habría aprendido de haber existido el ajuste). Devuelve cuántos se usar
on."""
        if self.mb_res:
            return 0
        n = 0
        for ev in sorted(events, key=lambda e: e.get("org", 0)):
            m = ev.get("match") or {}
            mbr = ev.get("mbRaw", ev.get("mb"))
            if (m.get("magType") or "").lower() == "mb" and m.get("mag") is not None and mbr is
 not None \
                    and (ev.get("mbN") or 0) >= 5 and abs(float(m["mag"]) - mbr) < 1.0:
                self.mb_res.append([float(m["mag"]) - float(mbr), ev.get("org", time.time())])
                n += 1
        self.mb_res = self.mb_res[-60:]
        if n:
            self.save()
        return n

    def _refit(self):
        from .magglobal import DEFAULT_CONV
        self.fit = {}
        self.off = {}
        for kind, pr in self.pairs.items():
            if 1 <= len(pr) < 15 and kind in DEFAULT_CONV:
                a0, b0 = DEFAULT_CONV[kind][:2]
                n = len(pr)
                r = np.array([p[1] - (a0 * p[0] + b0) for p in pr])
                r = r[np.abs(r - np.median(r)) < 0.5]
                if len(r) >= 1:
                    self.off[kind] = float(np.median(r)) * n / (n + 5)
            if len(pr) < 15:
                continue
            x = np.array([p[0] for p in pr])
            y = np.array([p[1] for p in pr])
            keep = np.ones(len(x), bool)
            ab = None
            for _ in range(5):
                if keep.sum() < 10 or np.ptp(x[keep]) < 0.5:
                    ab = None
                    break
                a = float(np.clip(np.polyfit(x[keep], y[keep], 1)[0], 0.6, 1.5))
                b = float(np.median(y[keep] - a * x[keep]))
                ab = (a, b)
                new = np.abs(y - (a * x + b)) < 0.5
                if (new == keep).all():
                    break
                keep = new
            if ab is not None and keep.sum() >= 10:
                self.fit[kind] = (ab[0], ab[1], int(keep.sum()))

    def save(self):
        if self.storage:
            self.storage.save_json("calibracion.json", {"q_mb": self.q["mb"], "q_mB": self.q["m
B"], "q_mbR": self.q["mbR"], "pares": self.pairs,
                                                        "mb_evento": self.mb_res, "eventos": so
rted(self.events)[-3000:]})

```

```
    def summary(self):
        from .magglobal import DEFAULT_CONV
        rows = []
        for kind in ("Mwp", "mB", "mb", "Ms"):
            n = len(self.pairs.get(kind, []))
            if kind in self.fit:
                a, b, k = self.fit[kind]
                rows.append({"tipo": kind, "pares": n, "relacion": f"Mw = {a:.3f}·{kind} {'+' i
f b >= 0 else '−'} {abs(b):.3f}",
                             "origen": f"ajustada con {k} sismos del USGS"})
            else:
                keys = ["Ms_lo", "Ms_hi"] if kind == "Ms" else [kind]
                rel = " / ".join(f"Mw = {DEFAULT_CONV[k][0]:g}·{kind} {'+' if DEFAULT_CONV[k][1
] >= 0 else '−'} {abs(DEFAULT_CONV[k][1]):g}"
                                 f" ({DEFAULT_CONV[k][2]:g}–{DEFAULT_CONV[k][3]:g})" for k in k
eys)
                o = self.off.get(kind)
                if o is not None:
                    rel += f" {'+' if o >= 0 else '−'} {abs(o):.2f}"
                    org = f"{DEFAULT_CONV[keys[0]][4]} con ajuste temprano de {o:+.2f} ({n} sis
mos del USGS; recta propia desde 15)"
                else:
                    org = f"{DEFAULT_CONV[keys[0]][4]} (faltan {max(0, 15 - n)} sismos para aju
star)"
                rows.append({"tipo": kind, "pares": n, "relacion": rel, "origen": org})
        qn = {k: sum(c[1] for c in v.values()) for k, v in self.q.items()}
        qb = {k: {b: round(c[0] / (c[1] + 5), 2) for b, c in sorted(v.items())} for k, v in sel
f.q.items()}
        return {"conversiones": rows, "q_lecturas": qn, "q_correcciones": qb,
                "mb_sesgo": {"correccion": round(self.mb_offset(), 3), "medido": round(self.mb_
raw_bias(), 3), "sismos": len(self.mb_res)}}
```

sismo/datacast.py — 172 líneas

```
"""Receptor UDP DataCast para estaciones Raspberry Shake personales (SIMMSA).

Permite recibir directamente el flujo de datos en tiempo real de estaciones Raspberry Shake
personales mediante su función nativa DATACAST (UDP):
  {'CHANNEL', TIMESTAMP, DATA_1, DATA_2, DATA_3, ...}
Ejemplo:
  {'EHZ', 1730248000.123, 843, 850, 849, 835, 820, ...}
"""
import socket
import threading
import time
import re
import numpy as np
from .places import where_is

_DC_RE = re.compile(r"^\{['\"]?([A-Za-z0-9_]{3})['\"]?\s*,\s*([0-9]+\.?[0-9]*)\s*,\s*(.+)\}$")

# Sensibilidad estándar de geófono Racotech 4.5 Hz de Raspberry Shake: ~3.99e8 cuentas / (m/s)
RS_DEFAULT_SCALE = 399000000.0


class DataCastServer:
    def __init__(self, engine, queue, cfg=None, log=print):
        self.engine = engine
        self.queue = queue
        self.log = log
        self.cfg = cfg or {}
        self.running = True
        self.sockets = []
        self.threads = []
        self.packets_total = 0
        self.last_data = 0.0

        # Estaciones configuradas por puerto
        self.station_configs = {}

        rs_cfg = self.cfg.get("raspberryshake_datacast", {}) or self.cfg.get("raspberryshake",
    {})
        estaciones = rs_cfg.get("estaciones", [])

        # Si no se especificaron, crear por defecto los dos puertos 8888 y 8889 para las 2 esta
    ciones
        if not estaciones:
            estaciones = [
                {"codigo": "RS01", "net": "AM", "puerto": 8888, "lat": 19.4326, "lon": -99.1332
    , "where": "Raspberry Shake 1"},
                {"codigo": "RS02", "net": "AM", "puerto": 8889, "lat": 16.8531, "lon": -99.8237
    , "where": "Raspberry Shake 2"}
            ]

        for est in estaciones:
            p = int(est.get("puerto") or est.get("puerto_udp") or 8888)
            code = str(est.get("codigo") or est.get("sta") or f"RS{p}").strip()
            net = str(est.get("red") or est.get("net") or "AM").strip()
            lat = float(est.get("lat", 19.4326))
            lon = float(est.get("lon", -99.1332))
            nombre = est.get("nombre") or est.get("where") or f"Raspberry Shake {code}"
            scale = float(est.get("scale") or est.get("cuentas_por_m_s") or RS_DEFAULT_SCALE)

            self.station_configs[p] = {
                "sta": code,
                "net": net,
                "lat": lat,
                "lon": lon,
                "where": nombre,
                "scale": scale,
                "puerto": p,
                "packets": 0,
                "last_data": 0.0
            }

    def start(self):
        ports = list(self.station_configs.keys())
        for port in ports:
            try:
                sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
                sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
                sock.bind(("0.0.0.0", port))
                self.sockets.append(sock)
                t = threading.Thread(target=self._listen_loop, args=(sock, port), daemon=True,
    name=f"datacast-udp-{port}")
                t.start()
                self.threads.append(t)
                st = self.station_configs[port]
                self.log(f"DataCast UDP: escuchando en puerto {port} para estación {st['net']}.
    {st['sta']} ({st['where']})")
            except Exception as e:
                self.log(f"DataCast UDP: no se pudo enlazar puerto {port} ({e})")

    def _listen_loop(self, sock, port):
        st_cfg = self.station_configs[port]
        net = st_cfg["net"]
        sta = st_cfg["sta"]
        key = f"{net}_{sta}"

```

```
        # Registrar la estación inicialmente en el inventario del motor para que aparezca en el mapa
        meta = {
            "key": key,
            "net": net,
            "sta": sta,
            "lat": st_cfg["lat"],
            "lon": st_cfg["lon"],
            "where": st_cfg["where"],
            "fuente": "Raspberry Shake (DataCast)",
            "pref": "00.EH",
            "chans": {"00.EHZ": st_cfg["scale"], "00.SHZ": st_cfg["scale"]}
        }
        from .engine import Station
        with self.engine.lock:
            if key not in self.engine.stations:
                self.engine.stations[key] = Station(meta)
                try:
                    self.engine.assoc.build(self.engine.stations)
                except Exception:
                    pass

        while self.running:
            try:
                raw, addr = sock.recvfrom(65536)
                if not raw:
                    continue
                text = raw.decode("ascii", errors="ignore").strip()
                if not (text.startswith("{") and text.endswith("}")):
                    continue

                m = _DC_RE.match(text)
                if not m:
                    parts = [x.strip().strip("'\"") for x in text[1:-1].split(",")]
                    if len(parts) < 3:
                        continue
                    cha = parts[0].upper()
                    t0 = float(parts[1])
                    samples = np.array([float(x) for x in parts[2:]], dtype=np.float64)
                else:
                    cha = m.group(1).upper()
                    t0 = float(m.group(2))
                    samples = np.fromstring(m.group(3), sep=",", dtype=np.float64)

                n = len(samples)
                if n == 0:
                    continue

                # Frecuencia de muestreo típica
                if n == 25 or cha.startswith("E") or cha.startswith("H"):
                    fs = 100.0
                elif n == 50 or cha.startswith("S"):
                    fs = 50.0
                else:
                    fs = 100.0

                now = time.time()
                self.last_data = now
                self.packets_total += 1
                st_cfg["last_data"] = now
                st_cfg["packets"] += 1

                # Asegurar registro de banda y ganancia
                with self.engine.lock:
                    s = self.engine.stations.get(key)
                    if s:
                        if not s.chans.get(f"00.{cha}"):
                            s.chans[f"00.{cha}"] = st_cfg["scale"]

                # Entregar el bloque de trazas a la cola central
                self.queue.put((net, sta, "00", cha, fs, t0, samples))

            except Exception:
                if not self.running:
                    break
                time.sleep(0.01)

    def stop(self):
        self.running = False
        for s in self.sockets:
            try:
                s.close()
            except Exception:
                pass
```

sismo/depthphases.py — 162 líneas

```
"""Fases de profundidad pP y sP: profundidad confiable con estaciones lejanas (Fase 3).

Con solo ondas P telesísmicas, la profundidad y la hora de origen se compensan entre sí y la profundidad queda mal resuelta. Las fases reflejadas en la superficie sobre el foco (pP, sP) llegan después de la P con un retraso que depende casi solo de la profundidad (pP − P ≈ h/4 s a 60°).

Método (apilamiento por profundidad, robusto aunque cada estación sea ruidosa):
1. En cada estación a 25°–95° con P leída se calcula una función característica STA/LTA
   (0.5–2.5 Hz, 1 s / 8 s) en los 230 s siguientes a la P.
2. Para cada profundidad de prueba h (10–700 km cada 5 km) se toma, en cada estación, el máximo de
   la función alrededor de P + [pP − P](Δ, h) y P + [sP − P](Δ, h), y se promedian las estaciones.
3. La profundidad con el mayor apilamiento es la candidata. Se acepta si destaca claramente
   (pico ≥ 3 × promedio de la curva lejos del pico), al menos 4 estaciones muestran pP y 6 entre pP y sP,
   y no contradice la profundidad que da la P sola (diferencia ≤ máx(60 km, 3 σ)).
4. En esas estaciones se lee el arribo pP / sP con AIC y se agrega al sismo como fase; el
   localizador vuelve a calcular el hipocentro y la profundidad queda amarrada por las diferencias
   pP − P y sP − P.
"""
import numpy as np
from scipy import signal
from scipy.ndimage import maximum_filter1d, median_filter

from .dsp import aic_window
from .locator import dist_km, KM_DEG

DMIN, DMAX = 25.0, 95.0
WIN = 230.0
HS = np.arange(25.0, 701.0, 5.0)     # por arriba de ~25 km la pP se confunde con la propia P


def _cf(ring, scale, tp):
    got = ring.get(tp - 40, tp + WIN)
    if got is None:
        return None
    t0, data = got
    fs = ring.fs
    if t0 > tp - 20 or t0 + len(data) / fs < tp + WIN - 1:
        return None
    sos = signal.butter(2, [0.5, min(2.5, 0.45 * fs)], btype="bandpass", fs=fs, output="sos")
    x = signal.sosfiltfilt(sos, data - data.mean())
    e = x * x
    ns, nl = max(1, int(1.0 * fs)), max(2, int(8.0 * fs))
    c = np.concatenate([[0.0], np.cumsum(e)])
    i = np.arange(len(e))
    sta = (c[i + 1] - c[np.maximum(0, i + 1 - ns)]) / ns
    j = np.maximum(0, i + 1 - ns)
    lta = (c[j] - c[np.maximum(0, j - nl)]) / nl
    r = sta / np.maximum(lta, 1e-30)
    cf = np.clip(r - 1.0, 0, 50)
    tt = t0 + i / fs
    k = (tt >= tp + 6.0)                    # se excluye la P misma (sus primeros 6 s)
    # v2.1.0: segunda función característica, independiente de la primera: logaritmo de la envolvente menos su
    # tendencia (mediana móvil de 40 s). Resalta los pulsos que sobresalen de la coda de la P.
    env = np.abs(signal.hilbert(x))
    nw = max(1, int(fs))
    env = np.convolve(env, np.ones(nw) / nw, "same")
    noise = np.median(env[:max(1, int(15 * fs))]) + 1e-30
    le = np.log(env + 0.1 * noise)
    step = max(1, int(fs // 5))                              # mediana sobre la serie submuestreada (rápida)
    base = np.interp(i, i[::step], median_filter(le[::step], size=max(3, int(40 * fs / step)), mode="nearest"))
    cf2 = np.clip(le - base, 0, None)
    return tt[k], cf[k], cf2[k]


def ready_count(engine, ev):
    n = 0
    for p in ev["picks"]:
        if p["phase"] != "P":
            continue
        s = engine.stations.get(p["key"])
        if not s or "Z" not in s.comps:
            continue
        d = float(dist_km(ev["lat"], ev["lon"], s.lat, s.lon))
        if DMIN * KM_DEG <= d <= DMAX * KM_DEG and (s.comps["Z"].next_t or 0) >= p["t"] + WIN:
            n += 1
    return n


def scan(engine, ev):
    """Busca fases de profundidad. Devuelve (resultado, picks_nuevos) o (None, []) si aún no hay datos."""
    cands = []
    for p in ev["picks"]:
        if p["phase"] != "P":
            continue
        s = engine.stations.get(p["key"])
        if not s or "Z" not in s.comps:
```

```
            continue
        d = float(dist_km(ev["lat"], ev["lon"], s.lat, s.lon))
        if not (DMIN * KM_DEG <= d <= DMAX * KM_DEG):
            continue
        cands.append((p, s, d))
    ready = [(p, s, d) for p, s, d in cands if (s.comps["Z"].next_t or 0) >= p["t"] + WIN]
    if len(ready) < 6:
        return None, []
    tt = engine.tt
    rows_pp, rows_sp, used, rows2 = [], [], [], []
    for p, s, d in ready:
        got = _cf(s.comps["Z"].ring, s.comps["Z"].scale, p["t"])
        if got is None:
            continue
        t, cf, cf2 = got
        dd = np.full(len(HS), d)
        tP = tt.tt_zgrid("P", dd, HS)
        dpp = tt.tt_zgrid("pP", dd, HS) - tP
        dsp = tt.tt_zgrid("sP", dd, HS) - tP
        fs = s.comps["Z"].ring.fs
        mx = maximum_filter1d(cf, size=int(2 * 1.2 * fs) + 1)          # máximo en ±1.2 s
        ipp = np.clip(np.round((p["t"] + dpp - t[0]) * fs).astype(int), 0, len(mx) - 1)
        isp = np.clip(np.round((p["t"] + dsp - t[0]) * fs).astype(int), 0, len(mx) - 1)
        vpp = np.where(p["t"] + dpp >= t[0], mx[ipp], 0.0)
        vsp = np.where(p["t"] + dsp >= t[0], mx[isp], 0.0)
        mx2 = maximum_filter1d(cf2, size=int(2 * 1.5 * fs) + 1)
        v2 = np.where(p["t"] + dpp >= t[0], mx2[ipp], 0.0) + 0.7 * np.where(p["t"] + dsp >= t[0
], mx2[isp], 0.0)
        rows2.append(v2 / (np.median(cf2) + 1e-6))
        # normalización por estación: una estación muy ruidosa no domina el apilamiento
        ref = np.median(cf) + 1e-6
        rows_pp.append(np.minimum(vpp / (ref * 10), 3.0))
        rows_sp.append(np.minimum(vsp / (ref * 10), 3.0))
        used.append((p, s, d, dpp, dsp, vpp, vsp, ref))
    if len(used) < 6:
        return None, []
    Spp, Ssp = np.mean(rows_pp, axis=0), np.mean(rows_sp, axis=0)
    S = Spp + 0.7 * Ssp
    k = int(np.argmax(S))
    h = float(HS[k])
    far = np.abs(HS - HS[k]) > 40                       # fondo: profundidades lejos del pico
    contrast = float(S[k] / (np.mean(S[far]) + 0.02 * S[k] + 1e-6)) if far.any() else 0.0
    # estaciones donde la fase se ve individualmente a la profundidad elegida
    npp = sum(1 for u in used if u[5][k] >= 3.0 and u[5][k] >= 6 * u[7])
    nsp = sum(1 for u in used if u[6][k] >= 3.0 and u[6][k] >= 6 * u[7])
    res = {"h": h, "contrast": round(contrast, 2), "nStations": len(used), "npP": npp, "nsP": n
sp,
           "curve": [round(float(x), 3) for x in S[::2]], "curveDepths": [float(x) for x in HS[
::2]]}
    # congruencia con la profundidad que da la P sola (cuando la P sí la resuelve, la pP no pue
de contradecirla)
    zP, ezP = float(ev["z"]), float(ev.get("errZ") or 999)
    res["consistentP"] = bool(abs(h - zP) <= max(60.0, 3.0 * ezP))
    ok = contrast >= 3.0 and npp >= 4 and npp + nsp >= 6 and res["consistentP"]
    # v2.1.0: segunda vía de aceptación. Con datos reales la vía anterior casi nunca se cumplía
 (contraste ≈ 1.2).
    # Dos funciones características independientes que dan la misma profundidad (± 30 km) con 8
 o más estaciones
    # es una señal confiable (en la revisión con sismos reales: 4 de 4 correctas, sin falsas).
    S2 = np.mean(rows2, axis=0)
    h2 = float(HS[int(np.argmax(S2))])
    res["h2"] = h2
    agree = abs(h - h2) <= 30.0 and len(used) >= 8 and h >= 25.0 and abs(h - zP) <= max(100.0,
3.0 * ezP)
    res["via"] = "contraste" if ok else ("acuerdo" if agree else None)
    res["sigma"] = 10.0 if ok else 15.0
    if not ok and agree:
        res["h"] = h = round((h + h2) / 2 / 5) * 5
        ok = True
    res["accepted"] = bool(ok)
    if not ok:
        return res, []
    picks = []
    for p, s, d, dpp, dsp, vpp, vsp, ref in used:
        for ph, delays, vals in (("pP", dpp, vpp), ("sP", dsp, vsp)):
            if vals[k] < 3.0 or vals[k] < 6 * ref:
                continue
            tpred = p["t"] + delays[k]
            r = aic_window(s.comps["Z"].ring, 0.5, 2.5, tpred - 2.0, tpred + 2.0, snr_win=3.0)
            if not r or r["snr"] < 1.5:
                continue
            picks.append(engine._mkpick(s, r["t"], ph, max(0.3, r["sigma"]), auto=True, depthPh
ase=True))
    return res, picks
```

sismo/depthprior.py — 139 líneas

```
"""Profundidad previa por región (v2.1.0).

Con solo ondas P lejanas, la profundidad y la hora de origen se compensan y el localizador puede poner a 100 km
un sismo superficial (o al revés). Para acotarlo se usa lo que ya se sabe de cada región: la profundidad de los
sismos que el USGS ha registrado alrededor del epicentro (M ≥ 4.5 desde 2000, unos 185 000 sismos).

Con esas profundidades se arma una densidad de probabilidad p(h) (núcleo gaussiano de 10 km o 12 % de h) y el
localizador suma al ajuste la penalización  −2·ln[(1−ε)·p(h)/p_max + ε]  (ε = 5 %, máximo ≈ 6 unidades de χ²).
Cuando los tiempos de llegada definen bien la profundidad mandan los datos; cuando no la definen (estaciones
lejanas, pocas fases), gana la profundidad típica de la zona: superficial en dorsales y fallas, intermedia o
profunda en las placas que se hunden (Tonga-Fiyi, Hindu Kush, Bucaramanga, Marianas…).

El catálogo se descarga una vez en segundo plano a datos/profundidad_usgs.npz y se renueva cada 30 días.
"""
import os
import threading
import time

import numpy as np

URL = ("https://earthquake.usgs.gov/fdsnws/event/1/query?format=text&minmagnitude={m}"
       "&starttime={a}-01-01&endtime={b}-01-01&limit=20000")
FILE = "profundidad_usgs.npz"
ZG = np.arange(0, 701, 5.0)          # malla de profundidad de la densidad (km)
R_KM = 150.0                         # radio alrededor del epicentro
N_MIN = 15                           # sismos históricos mínimos para usar la información previa
EPS = 0.05


def pen_from_depths(depths, weight=1.0):
    """Función pen(h) en unidades de χ² a partir de las profundidades históricas cercanas."""
    z = np.asarray(depths, float)
    z = z[np.isfinite(z)]
    if len(z) < N_MIN:
        # En regiones con poca o nula sismicidad (cratones, interiores continentales),
        # la sismicidad en el manto profundo es físicamente imposible; se acota a corteza (≤ 35 km)
        pen = np.where(ZG <= 35, 0.0, np.minimum(120.0, ((ZG - 35) / 10.0) ** 2)) * (weight * 3.0)
        def f_crust(h):
            return float(np.interp(h, ZG, pen))
        f_crust.n = len(z)
        f_crust.zbest = 15.0
        return f_crust
    z = np.clip(z, 0, 700)
    zmax = float(np.max(z))
    sig = np.maximum(10.0, 0.12 * z)
    dens = (np.exp(-0.5 * ((ZG[:, None] - z[None, :]) / sig[None, :]) ** 2) / sig[None, :]).sum(axis=1)
    dens /= dens.max()
    pen = -2.0 * np.log((1 - EPS) * dens + EPS) * weight
    # Si la profundidad buscada supera el máximo histórico regional + 30 km, se penaliza cuadráticamente
    # para evitar inversiones falsas en el manto profundo (p. ej. 590 km en Alaska o Irán donde no hay placa a esa prof.)
    pen += np.maximum(0.0, (ZG - (zmax + 30.0)) / 10.0) ** 2 * (weight * 4.0)

    def f(h):
        return float(np.interp(h, ZG, pen))
    f.n = len(z)
    f.zbest = float(ZG[int(np.argmin(pen))])
    return f


class DepthPrior:
    def __init__(self, data_dir, log=print, weight=1.0, active=True, minmag=4.5, since=2000):
        self.data_dir, self.log, self.weight, self.active = data_dir, log, float(weight), active
        self.minmag, self.since = minmag, since
        self.lat = self.lon = self.z = None
        self.cache = {}
        self.lock = threading.Lock()
        if not active:
            return
        path = os.path.join(data_dir, FILE)
        if os.path.exists(path):
            try:
                d = np.load(path)
                self.lat, self.lon, self.z = d["lat"], d["lon"], d["z"]
                self.log(f"Profundidad previa: {len(self.z)} sismos históricos del USGS")
            except Exception as e:
                self.log(f"Profundidad previa: no se pudo leer {FILE} ({e})")
        if self.z is None or time.time() - os.path.getmtime(path) > 30 * 86400:
            threading.Thread(target=self._download, daemon=True, name="profundidad").start()

    def _download(self):
```

```
        from .netio import http_get
        lat, lon, z, ok = [], [], [], True
        year = time.gmtime().tm_year
        for a in range(self.since, year + 1, 2):
            try:
                txt = http_get(URL.format(m=self.minmag, a=a, b=a + 2), 120).decode("utf-8", "r
eplace")
            except Exception as e:
                self.log(f"Profundidad previa: el USGS no respondió para {a}-{a + 1} ({e}); se
reintentará en el próximo arranque")
                ok = False
                break
            for line in txt.splitlines():
                if not line or line[0] == "#":
                    continue
                f = line.split("|")
                try:
                    la, lo, h = float(f[2]), float(f[3]), float(f[4])
                except (IndexError, ValueError):
                    continue
                lat.append(la)
                lon.append(lo)
                z.append(h)
        if not ok or len(z) < 1000:
            return
        lat, lon, z = (np.array(v, np.float32) for v in (lat, lon, z))
        try:
            np.savez_compressed(os.path.join(self.data_dir, FILE), lat=lat, lon=lon, z=z)
        except OSError as e:
            self.log(f"Profundidad previa: no se pudo guardar ({e})")
        with self.lock:
            self.lat, self.lon, self.z = lat, lon, z
            self.cache = {}
        self.log(f"Profundidad previa: catálogo del USGS descargado ({len(z)} sismos M ≥ {self.
minmag} desde {self.since})")

    def depths_near(self, lat, lon):
        if self.z is None:
            return None
        with self.lock:
            la, lo, z = self.lat, self.lon, self.z
        sel = np.abs(la - lat) <= R_KM / 111.0 + 0.01
        if not sel.any():
            return np.array([])
        la, lo, z = la[sel], lo[sel], z[sel]
        r = np.radians
        c = np.sin(r(lat)) * np.sin(r(la)) + np.cos(r(lat)) * np.cos(r(la)) * np.cos(r(lo - lon
))
        d = 6371.0 * np.arccos(np.clip(c, -1, 1))
        return z[d <= R_KM]

    def __call__(self, lat, lon):
        """pen(h) para un epicentro, o None si no hay suficientes sismos históricos cerca."""
        if not self.active or self.z is None:
            return None
        key = (round(lat * 2) / 2, round(lon * 2) / 2)
        if key in self.cache:
            return self.cache[key]
        f = pen_from_depths(self.depths_near(lat, lon), self.weight)
        if len(self.cache) > 5000:
            self.cache = {}
        self.cache[key] = f
        return f
```

sismo/dsp.py — 238 líneas

```
"""Procesamiento de señales: búfer circular, filtros, detector STA/LTA y picador AIC."""
import numpy as np
from scipy import signal


class Ring:
    """Búfer circular de muestras con tiempo absoluto (segundos epoch)."""

    def __init__(self, fs, seconds):
        self.fs = float(fs)
        self.n = int(np.ceil(fs * seconds))
        self.a = np.zeros(self.n, np.float32)
        self.w = 0
        self.count = 0
        self.end = None  # tiempo de la siguiente muestra

    def push(self, t_start, vals):
        dt = 1.0 / self.fs
        vals = np.asarray(vals, np.float32)
        if self.end is not None:
            gap = t_start - self.end
            if gap > dt / 2:
                gap_samples = int(round(gap * self.fs))
                if gap_samples >= self.n:
                    self.count = 0
                    self.w = 0
                else:
                    curr_zero = min(self.n, gap_samples)
                    i1 = min(self.n - self.w, curr_zero)
                    self.a[self.w:self.w + i1] = 0
                    if curr_zero > i1:
                        self.a[:curr_zero - i1] = 0
                    self.w = (self.w + curr_zero) % self.n
                    self.count = min(self.n, self.count + curr_zero)
            elif gap < -dt / 2:
                overlap = int(round(-gap * self.fs))
                if overlap >= len(vals):
                    return
                vals = vals[overlap:]
                t_start = t_start + overlap * dt

        if len(vals) >= self.n:
            vals = vals[-self.n:]
            t_start = t_start + (len(vals) - self.n) * dt
        k = len(vals)
        i1 = min(self.n - self.w, k)
        self.a[self.w:self.w + i1] = vals[:i1]
        if k > i1:
            self.a[:k - i1] = vals[i1:]
        self.w = (self.w + k) % self.n
        self.count = min(self.n, self.count + k)
        self.end = t_start + k * dt

    def get(self, t1, t2):
        """Devuelve (t0, datos) del intervalo [t1, t2] recortado a lo disponible, o None."""
        if self.end is None or self.count == 0:
            return None
        dt = 1.0 / self.fs
        start = self.end - self.count * dt
        a, b = max(t1, start), min(t2, self.end)
        if b - a < dt:
            return None
        i0 = int(round((a - start) * self.fs))
        i1 = int(round((b - start) * self.fs))
        base = (self.w - self.count) % self.n
        idx = (base + np.arange(i0, i1)) % self.n
        return start + i0 * dt, self.a[idx].astype(np.float64)

    def span(self):
        if self.end is None or self.count == 0:
            return None
        return self.end - self.count / self.fs, self.end


def bandpass_sos(fs, f1, f2, order=2):
    f2 = min(f2, 0.45 * fs)
    return signal.butter(order, [f1, f2], btype="bandpass", fs=fs, output="sos")


def bandpass(x, fs, f1, f2):
    return signal.sosfilt(bandpass_sos(fs, f1, f2), x)


def wood_anderson_ba(fs):
    """Respuesta Wood-Anderson (ganancia unitaria): velocidad (m/s) -> desplazamiento (m)."""
    h, w0 = 0.7, 2 * np.pi / 0.8
    return signal.bilinear([1.0, 0.0], [1.0, 2 * h * w0, w0 * w0], fs=fs)


class StaLta:
    """Detector STA/LTA recursivo en flujo continuo. Local: 1.5–8 Hz, 1/30 s. Lejano: 0.5–2 Hz,
    3/60 s."""

    def __init__(self, fs, sta=1.0, lta=30.0, f1=1.5, f2=8.0):
        self.fs = fs
```

```
        self.band = (f1, f2)
        self.sos = bandpass_sos(fs, f1, f2)
        self.zi = np.zeros((self.sos.shape[0], 2))
        self.cs, self.cl = 1.0 / (sta * fs), 1.0 / (lta * fs)
        self.sta = 0.0
        self.lta = 0.0
        self.n = 0
        self.warm = int(2 * lta * fs)
        self.trig = False
        self.last_trig = -1e18

    def process(self, t_start, x, thr, dead=30.0):
        """Procesa un bloque; devuelve lista de (tiempo_disparo, razón)."""
        f, self.zi = signal.sosfilt(self.sos, x, zi=self.zi)
        e = f * f
        sta, _ = signal.lfilter([self.cs], [1, -(1 - self.cs)], e, zi=[self.sta * (1 - self.cs)])
        lta, _ = signal.lfilter([self.cl], [1, -(1 - self.cl)], e, zi=[self.lta * (1 - self.cl)])
        self.sta, self.lta = float(sta[-1]), float(lta[-1])
        ratio = sta / np.maximum(lta, 1e-30)
        out = []
        n0 = self.n
        self.n += len(x)
        if self.n <= self.warm:
            return out
        start = max(0, self.warm - n0)
        dt = 1.0 / self.fs
        i = start
        while i < len(x):
            if not self.trig:
                idx = np.nonzero(ratio[i:] > thr)[0]
                if not len(idx):
                    break
                i += idx[0]
                t = t_start + i * dt
                if t - self.last_trig > dead:
                    self.trig = True
                    self.last_trig = t
                    out.append((t, float(ratio[i])))
                else:
                    i += 1
                    continue
            else:
                idx = np.nonzero(ratio[i:] < 1.5)[0]
                if not len(idx):
                    break
                i += idx[0]
                self.trig = False
            i += 1
        return out


def aic_pick(x, lo, hi, fs):
    """Picador AIC de Maeda (1985): índice del mínimo entre lo y hi, e incertidumbre (s)."""
    n = len(x)
    if n < 40:
        return None
    a, b = max(10, lo), min(n - 10, hi)
    if b <= a:
        return None
    s1 = np.concatenate([[0.0], np.cumsum(x)])
    s2 = np.concatenate([[0.0], np.cumsum(x * x)])
    k = np.arange(a, b)
    n1, n2 = k.astype(float), (n - k).astype(float)
    v1 = (s2[k] - s1[k] ** 2 / n1) / n1
    v2 = ((s2[n] - s2[k]) - (s1[n] - s1[k]) ** 2 / n2) / n2
    aic = n1 * np.log(np.maximum(v1, 0) + 1e-20) + n2 * np.log(np.maximum(v2, 0) + 1e-20)
    j = int(np.argmin(aic))
    vmin = aic[j]
    l = j
    while l > 0 and aic[l - 1] - vmin < 4:
        l -= 1
    r = j
    while r < len(aic) - 1 and aic[r + 1] - vmin < 4:
        r += 1
    return a + j, float(min(1.5, max(0.05, (r - l) / (2 * fs))))


def rms(x, i0, i1):
    seg = x[max(0, i0):min(len(x), i1)]
    return float(np.sqrt(np.mean(seg * seg))) if len(seg) else 0.0


def aic_window(ring, f1, f2, tA, tB, snr_win=2.0):
    """AIC en la ventana [tA, tB] (s). Devuelve dict(t, sigma, snr) o None.
    snr_win: duración (s) de las ventanas antes/después del arribo para la razón señal/ruido.""
    "
    got = ring.get(tA - 40, tB + 3)
    if got is None:
        return None
    t0, data = got
    fs = ring.fs
    if t0 > tA - 6 or t0 + len(data) / fs < tB + snr_win + 0.2:
        return None
```

```
    f = bandpass(data, fs, f1, f2)
    off = max(0, int(round((tA - 5 - t0) * fs)))
    x = f[off:]
    lo = int(round((tA - (t0 + off / fs)) * fs))
    hi = int(round((tB - (t0 + off / fs)) * fs))
    r = aic_pick(x, lo, hi, fs)
    if r is None:
        return None
    k, sigma = r
    n2 = int(round(snr_win * fs))
    snr = rms(x, k, k + n2) / (rms(x, k - n2, k) or 1e-30)
    return {"t": t0 + (off + k) / fs, "sigma": sigma, "snr": snr}


def aic_window_rotated(ringN, ringE, baz_deg, f1, f2, tA, tB, snr_win=2.0):
    """Calcula AIC sobre la componente Transversal (T) rotada a partir de N y E
    para aislar limpiamente la onda SH suprimiendo la interferencia de la coda P."""
    gotN = ringN.get(tA - 40, tB + 3)
    gotE = ringE.get(tA - 40, tB + 3)
    if gotN is None or gotE is None:
        return None
    t0_N, dN = gotN
    t0_E, dE = gotE
    fs = ringN.fs
    if abs(fs - ringE.fs) > 0.1 or fs < 10:
        return None
    t0 = max(t0_N, t0_E)
    t1 = min(t0_N + len(dN) / fs, t0_E + len(dE) / fs)
    if t0 > tA - 6 or t1 < tB + snr_win + 0.2:
        return None
    iN0 = int(round((t0 - t0_N) * fs))
    iE0 = int(round((t0 - t0_E) * fs))
    npts = int(round((t1 - t0) * fs))
    if npts < 40 or iN0 + npts > len(dN) or iE0 + npts > len(dE):
        return None
    dataN = dN[iN0:iN0 + npts]
    dataE = dE[iE0:iE0 + npts]
    rad = np.radians(float(baz_deg))
    # Rotación a Transversal (SH): perpendicular a la dirección del rayo
    dataT = -dataN * np.sin(rad) + dataE * np.cos(rad)
    fT = bandpass(dataT, fs, f1, f2)
    off = max(0, int(round((tA - 5 - t0) * fs)))
    x = fT[off:]
    lo = int(round((tA - (t0 + off / fs)) * fs))
    hi = int(round((tB - (t0 + off / fs)) * fs))
    r = aic_pick(x, lo, hi, fs)
    if r is None:
        return None
    k, sigma = r
    n2 = int(round(snr_win * fs))
    snr = rms(x, k, k + n2) / (rms(x, k - n2, k) or 1e-30)
    return {"t": t0 + (off + k) / fs, "sigma": sigma, "snr": snr, "component": "T"}
```

sismo/engine.py — 1966 líneas

```
"""Motor MUNDIAL en tiempo real: ingesta, detección (local + lejana), asociación global, localización
sobre la esfera, fases S regionales y P dirigidas, magnitudes y validación contra el USGS."""
import threading
import time
from collections import deque

import numpy as np

from .associator import Associator
from .dsp import Ring, StaLta, aic_window, aic_window_rotated, bandpass
from .locator import Locator, dist_km, KM_DEG, bearing
from .magnitude import ml_iaspei, wa_peak, mwp_station, mwp_window, mmi, roman, SOIL, MWP_DMIN, MWP_DMAX
from .places import where_is, region_name
from .magglobal import mb_station, mB_station, ms_station, rayleigh_window, combine_mw, MB_DMIN, MB_DMAX, MS_DMIN, MS_DMAX, MS_ZMAX, MBR_DMIN, MBR_DMAX
from .calibracion import Calibration
from . import depthphases
from . import tsunami as tsu

BANDS = {"BH": 0, "HH": 1, "EH": 2, "SH": 2, "EN": 3, "HN": 4}
SS_DEFAULT_SCALE = 1e6     # SIDESIS Android: 1,000,000 counts = 1.0 m/s² (1 count = 1e-6 m/s²)
RS_DEFAULT_SCALE = 3.99e8  # Raspberry Shake geófono 4.5 Hz: ~3.99e8 counts / (m/s)
MIN_PICK_SNR = 2.5
ML_DMAX = 600.0            # la ML (Wood-Anderson) solo vale hasta ~600 km
S_DMAX = 2500.0            # onda S automática solo a distancias regionales en esta fase
TELE_KM = 20 * KM_DEG      # distancia a partir de la cual una estación es "telesísmica"
LIVE_S = 75 * 60           # un sismo se sigue actualizando durante 75 min (la Rayleigh tarda hasta 1 h a 100°)
PHASE_S = 30 * 60          # fases dirigidas: los primeros 30 min


def band_rank(band):
    loc, bb = band.split(".")
    return BANDS.get(bb, 9) + (0 if loc == "00" else 0.2 if loc == "" else 0.5)


def comp_of(cha):
    c = cha[-1:]
    # v0.5.5: X/Y = horizontales ortogonales (p. ej. red PS en Japón y el Pacífico), como 1/2
    return "Z" if c == "Z" else "N" if c in ("N", "1", "X") else "E" if c in ("E", "2", "Y") else None


class Comp:
    def __init__(self, fs, cha, scale, seconds):
        self.fs, self.cha, self.scale = fs, cha, scale
        self.ring = Ring(fs, seconds)
        self.next_t = None
        self.x0 = None
        self.last = 0.0


class Station:
    def __init__(self, meta):
        self.key = meta["key"]
        self.net, self.sta = meta["net"], meta["sta"]
        self.code = f"{self.net}.{self.sta}"
        self.lat, self.lon = meta["lat"], meta["lon"]
        self.chans = meta.get("chans", {})
        self.fuente = meta.get("fuente", "")
        self.where_custom = meta.get("where", "")
        # banda elegida con los metadatos (ganancias conocidas); con «todos los canales» llegan también otras
        # bandas y acelerómetros: se archivan, pero el motor usa solo ésta (v0.4.6)
        self.pref = f"{meta.get('loc', '')}.{meta['band']}" if meta.get("band") else None
        self.band = self.pref if self.net == "SS" and self.pref else None
        self.comps = {}
        self.det = None          # detector local 1.5–8 Hz
        self.det2 = None         # detector de ondas lejanas 0.5–2 Hz
        self.pending = []
        self.last_data = 0.0
        self.data_end = 0.0
        self.packets = 0
        self.trig_times = deque(maxlen=400)
        self.last_pick = -1e18
        self.thr_eff = 0.0
        self.armed_s = None


class Engine:
    def __init__(self, cfg, tt, storage, alerts=None, log=print):
        self.cfg, self.tt, self.storage, self.alerts, self.log = cfg, tt, storage, alerts, log
        self.lock = threading.RLock()
        det = cfg["deteccion"]
        self.thr = det["umbral_sta_lta"]
        self.thr2 = det.get("umbral_sta_lta_lejano", 3.5)
        self.min_sta = det["estaciones_minimas"]
        self.use_corr = det["correcciones_por_estacion"]
        mem = cfg.get("memoria", {})
        self.ring_z = mem.get("segundos_vertical", 1500)
        self.ring_h = mem.get("segundos_horizontales", 900)
        self.stations = {}
        self.picks = []
```

```
        self.events = storage.load_events() if storage else []
        for e in self.events:
            e["live"] = False
            if not e.get("final"):          # sismos que quedaron abiertos (v0.3.x): se cierran al cargar
                e["final"] = True
                e["noMag"] = e.get("mag") is None
        self.corr = storage.load_json("correcciones.json", {}) if storage else {}
        self.sessions_from = float(cfg.get("validacion", {}).get("desde_ts", 0.0))
        self.sessions = [s for s in (storage.load_json("sesiones.json", []) if storage else []) if s[0] >= self.sessions_from]
        self.usgs = []
        self.pois = cfg.get("puntos_de_interes", [])
        # v2.1.0: profundidad previa por región (sismicidad histórica del USGS)
        dcfg = cfg.get("deteccion", {}).get("profundidad_previa", {})
        self.dprior = None
        if storage and dcfg.get("activa", True):
            try:
                from .depthprior import DepthPrior
                self.dprior = DepthPrior(storage.folder, log, weight=dcfg.get("peso", 3.0))
            except Exception as e:
                log(f"Profundidad previa: {e}")
        self.locator = Locator(tt, self.corr_of, depth_prior=self.dprior)
        self.calib = Calibration(storage)
        folder = cfg.get("_carpeta_tierra") or (storage.folder if storage else None)
        self.landmask = tsu.LandMask(folder, log) if folder else None
        self.assoc = Associator(tt, self.min_sta)
        self.ai = None
        self.dirty = set()
        self.started = time.time()
        self.usgs_minmag = cfg.get("validacion", {}).get("magnitud_minima_usgs", 4.5)
        self.missed_minmag = cfg.get("validacion", {}).get("magnitud_no_detectados", 5.5)

    # ------------------------------------------------------------------ utilidades
    def corr_of(self, key, phase, d=None):
        """Corrección de tiempo aprendida para la estación; distinta para trayectorias regionales
        (< 20°) y telesísmicas, porque muestrean partes diferentes del manto."""
        if not self.use_corr:
            return 0.0
        tab = self.corr.get(self.tt.model, {})
        c = tab.get(key + ("|R" if (d is not None and d < TELE_KM) else "|T")) if d is not None else None
        if not c or c[1] < 2:
            return 0.0
        return (c[0] / c[1]) * (1.7 if phase == "S" else 1.0)

    def ttf(self, phase, d, z):
        return float(self.tt.tt(phase, d, z))

    def add_station(self, meta):
        with self.lock:
            self.stations[meta["key"]] = Station(meta)

    def register_ss_station(self, raw, persist=True):
        """Registra o actualiza en caliente una estación acelerográfica Android SIDESIS (red SS,
        canales ENZ/ENN/ENE, 50 Hz, 1e6 counts = 1.0 m/s²) y sincroniza datos/estaciones_ss.json."""
        sta_raw = str(raw.get("sta") or raw.get("station_id") or raw.get("id") or raw.get("codigo") or raw.get("code") or "").strip().upper()
        if "." in sta_raw:
            sta_raw = sta_raw.split(".")[-1].strip()
        if "_" in sta_raw:
            sta_raw = sta_raw.split("_")[-1].strip()
        if not sta_raw:
            return None
        sta = sta_raw[:5]
        key = f"SS_{sta}"
        with self.lock:
            existing = self.stations.get(key)
            lat = raw.get("lat") if raw.get("lat") is not None else raw.get("latitude", raw.get("latitud"))
            lon = raw.get("lon") if raw.get("lon") is not None else raw.get("lng", raw.get("longitude", raw.get("longitud")))
            lat = float(lat) if lat is not None else (existing.lat if existing else 19.4326)
            lon = float(lon) if lon is not None else (existing.lon if existing else -99.1332)
            wname = str(raw.get("where") or raw.get("station_name") or raw.get("name") or raw.get("nombre") or "").strip()
            if not wname and existing and getattr(existing, "where_custom", ""):
                wname = existing.where_custom
            scale = float(raw.get("scale") or SS_DEFAULT_SCALE)
            chans = {
                "00.ENZ": scale, "00.ENN": scale, "00.ENE": scale,
                ".ENZ": scale, ".ENN": scale, ".ENE": scale,
                "ENZ": scale, "ENN": scale, "ENE": scale,
            }
            meta = {
                "key": key,
                "net": "SS",
                "sta": sta,
                "lat": lat,
                "lon": lon,
                "loc": "00",
                "band": "EN",
```

```
                "chans": chans,
                "fuente": "SIDESIS",
                "where": wname,
                "updated": time.time(),
            }
            if existing is not None:
                existing.lat = lat
                existing.lon = lon
                existing.chans.update(chans)
                existing.fuente = "SIDESIS"
                if wname:
                    existing.where_custom = wname
            else:
                self.stations[key] = Station(meta)
                try:
                    self.assoc.build(self.stations)
                except Exception:
                    pass
            if persist and self.storage:
                try:
                    saved = self.storage.load_json("estaciones_ss.json", {})
                    if not isinstance(saved, dict):
                        saved = {}
                    saved[sta] = {
                        "key": key, "net": "SS", "sta": sta,
                        "lat": lat, "lon": lon, "loc": "00", "band": "EN",
                        "scale": scale, "where": wname, "updated": time.time()
                    }
                    self.storage.save_json("estaciones_ss.json", saved)
                except Exception as e:
                    self.log(f"SIDESIS: error guardando estaciones_ss.json ({e})")
            return self.stations[key]

    def purge_inactive_ss_stations(self, max_days=7):
        """Elimina definitivamente del catálogo y de memoria las estaciones de la Red SS
        que no hayan recibido datos ni heartbeat en los últimos `max_days` días."""
        cutoff = time.time() - (float(max_days) * 86400.0)
        purged = []
        with self.lock:
            saved = {}
            if self.storage:
                try:
                    saved = self.storage.load_json("estaciones_ss.json", {}) or {}
                except Exception:
                    saved = {}

            to_delete = []
            for sta_code, meta in saved.items():
                upd = float(meta.get("updated") or 0.0)
                s = self.stations.get(meta.get("key", f"SS_{sta_code}"))
                last_active = max(upd, getattr(s, "last_data", 0.0) if s else 0.0)
                if last_active < cutoff:
                    to_delete.append(sta_code)

            for sta_code in to_delete:
                purged.append(sta_code)
                saved.pop(sta_code, None)
                key = f"SS_{sta_code}"
                if key in self.stations:
                    del self.stations[key]

            for key, s in list(self.stations.items()):
                if s.net == "SS":
                    last_active = getattr(s, "last_data", 0.0)
                    if last_active and last_active < cutoff:
                        if s.sta not in purged:
                            purged.append(s.sta)
                        del self.stations[key]

            if self.storage and to_delete:
                try:
                    self.storage.save_json("estaciones_ss.json", saved)
                except Exception as e:
                    self.log(f"SIDESIS: error guardando estaciones_ss.json tras purga ({e})")

            if purged:
                try:
                    self.assoc.build(self.stations)
                except Exception:
                    pass
                self.log(f"SIDESIS: Purgadas {len(purged)} estación(es) inactivas (> {max_days}
días): {', '.join(purged)}")
        return purged

    # ------------------------------------------------------------------ ingesta
    def on_trace(self, net, sta, loc, cha, fs, t0, data):
        s = self.stations.get(f"{net}_{sta}")
        if s is None and net == "SS":
            s = self.register_ss_station({"sta": sta}, persist=True)
        comp = comp_of(cha)
        if s is None or comp is None or cha[:2] not in BANDS or not fs or not len(data):
            return
        with self.lock:
            band = f"{loc}.{cha[:2]}"
            if s.pref and band != s.pref:
```

```
                if s.band == s.pref and s.last_data and time.time() - s.last_data < 120:
                    return
                if s.last_data and time.time() - s.last_data < 120 and cha[:2] != s.pref.split(
".")[-1]:
                    return
            if s.band != band:
                if comp != "Z":
                    return
                if s.band and not (band_rank(band) < band_rank(s.band) and (s.det.n if s.det el
se 0) < 2 * 30 * fs):
                    return
                s.band, s.comps, s.det, s.det2, s.pending = band, {}, None, None, []
            c = s.comps.get(comp)
            if c is None or c.fs != fs:
                scale = s.chans.get(f"{loc}.{cha}") or (s.chans.get(f"{s.pref.split('.')[0]}.{c
ha}", 0.0) if s.pref else 0.0)
                if not scale and (s.net in ("SS", "AM") or cha[:2] in ("EN", "HN", "SH", "EH"))
:
                    scale = RS_DEFAULT_SCALE if (s.net == "AM" or cha[:2] in ("SH", "EH")) else
 SS_DEFAULT_SCALE
                c = s.comps[comp] = Comp(fs, cha, scale, self.ring_z if comp == "Z" else self.r
ing_h)
            dt = 1.0 / fs
            data = np.asarray(data, np.float64)
            k0, gap = 0, False
            if c.next_t is not None:
                g = t0 - c.next_t
                if g < -dt / 2:
                    k0 = int(round(-g * fs))
                    if k0 >= len(data):
                        return
                elif g > max(0.20, 2.0 * dt):
                    gap = True
            if c.x0 is None or gap:
                c.x0 = float(data[k0])
            vals = data[k0:] - c.x0
            t_start = t0 + k0 * dt
            c.ring.push(t_start, vals)
            c.next_t = t0 + len(data) * dt
            c.last = time.time()
            if comp == "Z":
                self._detect(s, c, t_start, vals, gap)
                s.last_data = time.time()
                s.data_end = c.next_t
                s.packets += 1
            if getattr(s, "armed_s", None):
                self._check_autonomous_s(s)

  def _detect(self, s, c, t_start, vals, gap):
        if gap:
            s.pending.clear()
        if s.det is None or s.det.fs != c.fs or gap:
            s.det = StaLta(c.fs, 1.0, 30.0, 1.5, 8.0)
        if s.det2 is None or s.det2.fs != c.fs or gap:
            s.det2 = StaLta(c.fs, 3.0, 60.0, 0.5, 2.0)
        if gap:
            return  # Inmunidad a saltos de paquete: no evaluar disparo en el bloque de reinici
o tras un corte
        # umbral adaptativo: una estación que dispara mucho por ruido se vuelve más exigente
        rate = sum(1 for t in s.trig_times if t_start - t < 3600)
        f = min(1.5, 1.0 + 0.05 * max(0, rate - 8))
        s.thr_eff = self.thr * f
        for t, r in s.det.process(t_start, vals, s.thr_eff):
            s.pending.append((t, r, "L"))
        for t, r in s.det2.process(t_start, vals, self.thr2 * f):
            s.pending.append((t, r, "T"))
        for p in list(s.pending):
            if c.next_t >= p[0] + 3.8:
                s.pending.remove(p)
                if p[2] == "L":
                    a = aic_window(c.ring, 1.5, 8, p[0] - 6, p[0] + 1.5)
                else:
                    a = aic_window(c.ring, 0.5, 2.0, p[0] - 10, p[0] + 1.5, snr_win=5.0)
                # control de calidad: un arribo real tiene un cambio claro de energía (SNR)
                min_snr_req = 4.5 if (getattr(s, "net", "") == "SS" or s.key.startswith("SS_"))
 else MIN_PICK_SNR
                if a is None or a["snr"] < min_snr_req:
                    det_obj = s.det if p[2] == "L" else s.det2
                    if det_obj is not None:
                        det_obj.trig = False
                        det_obj.last_trig = p[0] - 25.0
                    s.trig_times.append(p[0])
                    s.rejected = getattr(s, "rejected", 0) + 1
                    continue
                if getattr(s, "armed_s", None) and a["t"] >= s.armed_s["tp"] + 2.5:
                    self._check_autonomous_s(s)
                    continue
                if abs(a["t"] - s.last_pick) < 15:      # mismo arribo visto por los dos detect
ores
                    continue
                s.last_pick = a["t"]
                net_code = getattr(s, "net", "") or (s.key.split("_")[0] if "_" in s.key else "
")
                snr_val = float(a.get("snr", 3.0))
                raw_sig = float(a.get("sigma", 0.15))
```

```
                if net_code == "SS" or s.key.startswith("SS_"):
                    # Red casera/móvil SIDESIS: mayor incertidumbre temporal para evitar que di
storsione redes permanentes
                    sig_eff = max(0.40, raw_sig)
                elif net_code in ("MX", "GS", "IU", "II", "CU", "GI", "SV", "GE", "G", "AU", "N
Z", "CL"):
                    # Redes sismológicas oficiales de alta fidelidad: ponderación fuerte por SN
R
                    if snr_val >= 12.0:
                        sig_eff = min(0.08, max(0.04, raw_sig))
                    elif snr_val >= 6.0:
                        sig_eff = min(0.14, max(0.07, raw_sig))
                    else:
                        sig_eff = min(0.25, max(0.12, raw_sig))
                else:
                    if snr_val >= 10.0:
                        sig_eff = min(0.12, max(0.06, raw_sig))
                    else:
                        sig_eff = min(0.25, max(0.12, raw_sig))
                self._on_pick(s, a["t"], snr_val, sig_eff, p[2])

  # ------------------------------------------------------------------ asociación
  def _mkpick(self, s, t, phase, sigma, **kw):
      d = {"key": s.key, "code": s.code, "lat": s.lat, "lon": s.lon, "t": float(t), "phase":
phase, "sigma": float(sigma)}
      d.update(kw)
      return d

  def _coda_end(self, ev, d):
      """Hasta cuándo una estación sigue registrando el sismo (fases tardías, ondas superfici
ales,
      coda). A distancias lejanas (> 2500 km), solo sismos moderados o fuertes (M ≥ 5.5) prod
ucen
      coda con amplitud suficiente para contaminar otras regiones del planeta."""
      M = self.mag_of(ev) or 4.0
      if d > 2500.0 and M < 5.5:
          # Sismo regional o lejano menor a 5.5: no genera coda en otras placas continentales
          return ev["org"]
      return ev["org"] + d / 2.8 + 120 + 300 * max(0.0, M - 5.0)

  def _late_phase(self, ev, d, t):
      """¿El tiempo t coincide con una fase tardía del sismo ev en una estación a d km? Devue
lve su nombre o None.
      Solo se consideran las fases que en la banda de detección (0.5–8 Hz, vertical) tienen a
mplitud suficiente
      para disparar según la magnitud y la profundidad: un sismo moderado o profundo casi no
genera PP, SS
      ni ondas superficiales de periodo corto, y descartar esos lapsos ocultaba sismos trasla
pados."""
      org, z = ev["org"], ev["z"]
      M = self.mag_of(ev) or 4.5
      tt = lambda ph, dd, zz: float(self.tt.tt(ph, dd, zz))
      cand = []
      if d < 3000 or M >= 5.8:
          cand.append(("S", org + tt("S", d, z)))
      if d > 2000 and z >= 15:
          cand += [("pP", org + tt("pP", d, z)), ("sP", org + tt("sP", d, z))]
      if M >= 5.8 and d > 2000:
          cand.append(("PP", org + tt("P", d / 2, z) + tt("P", d / 2, 0.0)))
      if M >= 6.5 and d > 2000:
          cand.append(("SS", org + tt("S", d / 2, z) + tt("S", d / 2, 0.0)))
      for ph, tph in cand:
          if abs(t - tph) < 6.0 + 0.015 * (tph - org):
              return ph
      # ondas superficiales: solo sismos superficiales y grandes las tienen en la banda de de
tección
      if M >= 6.0 and z < 70 and org + d / 4.5 - 20 < t < org + d / 3.0 + 60:
          return "superficial"
      return None

  def _max_p_dist(self, ev):
      """Distancia máxima físicamente admisible para registrar ondas P según la magnitud o ma
durez del sismo."""
      m = ev.get("mag") or ev.get("magPre") or ev.get("magMb") or ev.get("magMwp") or ev.get(
"magMl")
      if m is None:
          return 2200.0
      if m < 4.0:
          return 900.0
      elif m < 4.5:
          return 1800.0
      elif m < 5.0:
          return 3800.0
      elif m < 5.5:
          return 7500.0
      elif m < 6.0:
          return 12500.0
      else:
          return 20000.0

  def _on_pick(self, s, t, snr, sigma, det="L"):
      s.trig_times.append(t)
      p = self._mkpick(s, t, "P", sigma, snr=snr, det=det)
      p["_ev"] = None
      # Armar detector autónomo de onda S para esta estación en ventana regional (2.5 a 75 s)
```

```
        s.armed_s = {
            "tp": float(t),
            "deadline": float(t) + 75.0,
            "last_eval": float(t) + 2.0,
            "snr_p": float(snr),
            "p_pick": p
        }
        for ev in self.events:
            if not ev["live"] or t - ev["org"] > 2 * LIVE_S:
                continue
            d = float(dist_km(ev["lat"], ev["lon"], s.lat, s.lon))
            if d > self._max_p_dist(ev):
                continue
            tp = ev["org"] + self.ttf("P", d, ev["z"]) + self.corr_of(s.key, "P", d)
            u = float(self.tt.slowness("P", d, ev["z"]))
            tol = min(3.8, max(2.5, 2.0 * sigma) + min(1.5, u * ev.get("errH", 10)))
            if abs(t - tp) < tol:
                if any(q["key"] == s.key and q["phase"] == "P" for q in ev["picks"]):
                    return
                ev["picks"].append(p)
                p["_ev"] = ev["id"]
                self.relocate(ev)
                return
            if tp - 5 < t < self._coda_end(ev, d):
                # Dentro de la coda de un sismo ya detectado. Si el arribo coincide con una fas
e tardía
                # de ese sismo (S, pP, sP, PP, SS, PcP u ondas superficiales) se descarta. Si n
o coincide
                # con ninguna, puede ser la P de OTRO sismo que ocurre minutos después (sismos
traslapados,
                # p. ej. Nueva Bretaña 11:09 y Islas Volcano 11:12 del 4-oct-2026): se conserva
 marcado.
                if self._late_phase(ev, d, t):
                    return
                p["coda"] = ev["id"]
        self.picks.append(p)
        self.picks = [q for q in self.picks if t - q["t"] < 1500]
        self._try_new_event(p)

    def _check_autonomous_s(self, s):
        """Detector autónomo de onda S en tiempo real para estaciones armadas tras el arribo P.
        Evalúa componentes horizontales (N/E) o vertical (Z) en ventana regional (2.5 a 75 s)."
""
        armed = getattr(s, "armed_s", None)
        if not armed:
            return
        tp = armed["tp"]
        c_z = s.comps.get("Z")
        if not c_z or c_z.next_t is None:
            return
        t_curr = c_z.next_t
        if t_curr > armed["deadline"]:
            s.armed_s = None
            return
        if t_curr < tp + 2.5:
            return
        if t_curr - armed.get("last_eval", 0) < 1.5:
            return
        armed["last_eval"] = t_curr

        tA = tp + 2.5
        tB = min(tp + 70.0, t_curr - 2.5)
        if tB - tA < 1.5:
            return

        r_best = None
        comp_used = "Z"
        N, E = s.comps.get("N"), s.comps.get("E")

        # 1. Componentes horizontales (N y E) si existen y tienen datos frescos
        if N and E and N.next_t is not None and E.next_t is not None:
            min_t = min(N.next_t, E.next_t)
            if min_t >= tB + 2.2:
                ev_id = armed.get("p_pick", {}).get("_ev")
                ev = next((x for x in self.events if x["id"] == ev_id), None) if ev_id else Non
e
                if ev:
                    try:
                        baz = float(bearing(s.lat, s.lon, ev["lat"], ev["lon"]))
                        r_T = aic_window_rotated(N.ring, E.ring, baz, 0.7, 5.0, tA, tB, 2.0)
                        if r_T and r_T.get("snr", 0) >= 2.5:
                            r_best = r_T
                            comp_used = "T"
                    except Exception:
                        pass
                if not r_best:
                    picks_h = []
                    for h_comp, c_h in (("N", N), ("E", E)):
                        r_h = aic_window(c_h.ring, 0.7, 5.0, tA, tB, 2.0)
                        if r_h and r_h.get("snr", 0) >= 2.5:
                            picks_h.append((r_h, h_comp))
                    if picks_h:
                        best_h = max(picks_h, key=lambda x: x[0]["snr"])
                        r_best = best_h[0]
                        comp_used = best_h[1]
```

```
525
526         # 2. Componente vertical Z (estaciones uniaxiales o cuando la onda S genera salto promi
    nente en Z)
527         if not r_best and c_z.next_t >= tB + 2.2:
528             r_z = aic_window(c_z.ring, 0.7, 5.0, tA, tB, 2.0)
529             if r_z and r_z.get("snr", 0) >= 2.8:
530                 r_best = r_z
531                 comp_used = "Z"
532 
533         if not r_best:
534             return
535 
536         ts = r_best["t"]
537         if ts <= tp + 2.0:
538             return
539 
540         snr_s = float(r_best.get("snr", 3.0))
541         sig_eff = min(1.8, max(0.15, float(r_best.get("sigma", 0.3))))
542         if snr_s >= 6.0:
543             sig_eff = min(0.20, sig_eff)
544 
545         p_s = self._mkpick(s, ts, "S", sig_eff, snr=snr_s, comp=comp_used, auto=True)
546         s.armed_s = None
547         s.last_pick = ts
548         p_p = armed["p_pick"]
549         dt_sp = ts - tp
550         self.log(f"Onda S autonoma detectada en {s.code} ({comp_used}): S-P = {dt_sp:.2f} s (~{
    dt_sp*8.4:.0f} km), SNR={snr_s:.1f}")
551         self._on_autonomous_s(s, p_s, p_p)
552 
553     def _on_autonomous_s(self, s, p_s, p_p):
554         ev_id = p_p.get("_ev")
555         ev = next((x for x in self.events if x["id"] == ev_id), None) if ev_id else None
556         if ev:
557             p_s["_ev"] = ev["id"]
558             if not any(q["key"] == s.key and q["phase"] == "S" for q in ev["picks"]):
559                 ev["picks"].append(p_s)
560                 ev["triedS"].add(s.key)
561                 self.relocate(ev)
562                 self.dirty.add(ev["id"])
563                 self.log(f"Onda S de {s.code} incorporada al sismo {ev['id']} -> relocalizado (
    {len(ev['picks'])} picks)")
564         else:
565             p_s["_ev"] = None
566             self.picks.append(p_s)
567             self.picks = [q for q in self.picks if p_s["t"] - q["t"] < 1500]
568             self._try_new_event(p_p)
569 
570     def _try_new_event(self, anchor):
571         tnow = anchor["t"]
572         # Sensores acelerográficos móviles / caseros (Red SS) no pueden ser el ancla solitaria
573         # de un sismo global; requieren al menos otra estación SS confirmando en ventana local.
574         if anchor["key"].startswith("SS_") or getattr(self.stations.get(anchor["key"]), "net",
    "") == "SS":
575             coinc_ss = [q for q in self.picks if q.get("_ev") is None and (q["key"].startswith(
    "SS_") or getattr(self.stations.get(q["key"]), "net", "") == "SS") and q["key"] != anchor["key"] and abs(q["t"] - tnow) < 20.0]
576             if len(coinc_ss) < 1:
577                 return
578 
579         cand = [q for q in self.picks if q.get("_ev") is None and 0 <= tnow - q["t"] < 1300 and
     q["key"] in self.stations]
580         p_keys = {q["key"] for q in cand if q.get("phase") == "P"}
581         s_keys = {q["key"] for q in cand if q.get("phase") == "S"}
582         sp_keys = p_keys.intersection(s_keys)
583 
584         # Quórum adaptativo según información multifase S-P:
585         if len(sp_keys) >= 2:
586             min_seed = max(3, self.min_sta - 5)
587         elif len(sp_keys) >= 1:
588             min_seed = max(3, self.min_sta - 4)
589         else:
590             min_seed = max(5, self.min_sta - 3)
591 
592         if len(p_keys) < min_seed:
593             return
594 
595         cand_p = [q for q in cand if q.get("phase") == "P"]
596         if anchor not in cand_p and anchor.get("phase") == "P":
597             cand_p.append(anchor)
598 
599         self.assoc.build(self.stations)
600         for group, pos in self.assoc.nucleate(cand_p, anchor, min_seed=min_seed):
601             got = self._solve_group(group, pos, min_seed=min_seed)
602             if got is not None:
603                 L1, inl = got
604                 break
605         else:
606             return
607         # Un sismo formado con arribos que caen en la coda de otro debe ser claramente distinto
    :
608         # más estaciones, origen separado y que sus arribos no sean, en su mayoría, fases tardí
    as del primero.
609         coda_ids = {q.get("coda") for q in inl if q.get("coda")}
610         if coda_ids:
611             for cid in list(coda_ids):
```

```
                ev0 = next((e for e in self.events if e["id"] == cid), None)
                if ev0 is None:
                    continue
                d_ev = float(dist_km(L1["lat"], L1["lon"], ev0["lat"], ev0["lon"]))
                m0 = self.mag_of(ev0) or 4.0
                # Si el sismo anterior ocurrió a más de 2500 km y fue < 5.5, no es una réplica ni coda real
                if d_ev > 2500 and m0 < 5.5:
                    coda_ids.remove(cid)
                    continue
                if abs(L1["org"] - ev0["org"]) < 20 and d_ev < 1500:
                    return
            if coda_ids:
                # No secuestrar quórum temprano si el origen está claramente separado (> 60 s) del sismo previo
                separate_origin = all(abs(L1["org"] - (next((e["org"] for e in self.events if e["id"] == c), 0))) > 60 for c in coda_ids)
                if not (L1.get("early") and separate_origin):
                    if len(inl) < self.min_sta + 2:
                        return
        # ¿es el mismo sismo que uno ya detectado (localizado distinto con menos estaciones)?
        twin = self._twin(L1["lat"], L1["lon"], L1["org"]) or self._dup_of(L1["lat"], L1["lon"], L1["org"], inl)
        if twin is not None:
            have = {p["key"] for p in twin["picks"] if p["phase"] == "P"}
            for q in inl:
                if q["key"] not in have:
                    q["_ev"] = twin["id"]
                    twin["picks"].append(q)
            self.relocate(twin)
            return
        is_mex = (12.0 <= L1["lat"] <= 33.0 and -118.0 <= L1["lon"] <= -86.0)
        early_init = bool(L1.get("early")) or (is_mex and len(inl) < 10)
        ev = {"id": "ev" + np.base_repr(int(time.time() * 1000), 36).lower(), "created": time.time(), "live": True,
              "model": self.tt.model, "triedS": set(), "triedP": set(), "aiDone": set(), "picks": inl,
              "early": early_init, "prelim_consolidated": not early_init}
        if ev["early"]:
            ev["tEarly"] = time.time()
            ev["tPrelim"] = ev["tEarly"]
        for q in inl:
            q["_ev"] = ev["id"]
        # Incorporar picks S autónomos ya existentes para las estaciones del grupo
        have_s_keys = {p["key"] for p in inl if p["phase"] == "S"}
        for q in self.picks:
            if q.get("phase") == "S" and q.get("_ev") is None and q["key"] in {p["key"] for p in inl} and q["key"] not in have_s_keys:
                q["_ev"] = ev["id"]
                inl.append(q)
                ev["triedS"].add(q["key"])
                have_s_keys.add(q["key"])
        self._apply(ev, L1)
        if len(have_s_keys) > 0:
            self.relocate(ev)
        self.events.insert(0, ev)
        self.events = self.events[:1000]
        self.update_mag(ev)
        mmin = self.cfg.get("alertas", {}).get("magnitud_minima", 4.0)
        if ev.get("mag") is not None and ev["mag"] < mmin - 0.2 and not ev.get("early"):
            if ev in self.events:
                self.events.remove(ev)
            for q in inl:
                q["_ev"] = None
            return
        self.match_event(ev)
        self.dirty.add(ev["id"])
        self.log(f"Sismo detectado ({'TEMPRANO' if ev.get('early') else 'OFICIAL'}): {self.mag_str(ev)} {where_is(ev['lat'], ev['lon'])} prof {ev['z']:.0f} km ({len(inl)} est.)")
        if self.alerts:
            self.alerts.event(self, ev, final=False)

    def _solve_group(self, group, pos, min_seed=None):
        """Localización iterativa sobre un grupo asociado: se quitan los picks incongruentes.
        Devuelve (solución, picks) si el grupo es un sismo plausible, o None."""
        if min_seed is None:
            min_seed = max(5, self.min_sta - 3)
        if len(group) < min_seed:
            return None

        # Veto de Coherencia Espacial de Frente de Onda:
        # Descartar picks espurios que violen la propagación física (ej. una estación a > 400 km
        # que reporte arribo más de 10 s antes que estaciones situadas 250 km más cerca del foco).
        clean_group = []
        for q in group:
            dq = float(dist_km(pos[0], pos[1], q["lat"], q["lon"]))
            closer = [c for c in group if float(dist_km(pos[0], pos[1], c["lat"], c["lon"])) < dq - 250.0]
            if len(closer) >= 2 and q["t"] < min(c["t"] for c in closer) - 10.0:
                continue  # Descartar pick por contradicción física de frente de onda
            clean_group.append(q)
        if len(clean_group) >= min_seed:
            inl = clean_group
```

```
        else:
            inl = group

        for _ in range(5):
            L1 = self.locator.locate(inl, guess=pos, full=False)
            pos = (L1["lat"], L1["lon"], L1["z"])
            keep = [q for i, q in enumerate(inl) if abs(L1["res"][i]) < min(8.0, max(2.5, 2.5 *
 L1["sig"][i]))]
            # P difractada / PKP (> 100°) solo se aceptan si el sismo es grande, es decir, si t
ambién
            # lo registran muchas estaciones a distancias normales (< 98°)
            dk = [float(dist_km(L1["lat"], L1["lon"], q["lat"], q["lon"])) for q in keep]
            if sum(1 for x in dk if x < 98 * KM_DEG) < 10:
                keep = [q for q, x in zip(keep, dk) if x < 98 * KM_DEG]
            if len(keep) < min_seed:
                return None
            if len(keep) == len(inl):
                break
            inl = keep
        else:
            return None
        if L1["rms"] <= 2.2:
            have_k = {q["key"] for q in inl}
            dmax_r = min(2600.0, max(1200.0, float(np.median(dk)) * 1.3))
            rescued = []
            for s in self.stations.values():
                if s.key in have_k or not s.last_data:
                    continue
                ds = float(dist_km(L1["lat"], L1["lon"], s.lat, s.lon))
                if ds <= dmax_r:
                    rp = self._pick_p_targeted(s, L1)
                    if isinstance(rp, dict):
                        tp = L1["org"] + self.ttf("P", ds, L1["z"]) + self.corr_of(s.key, "P",
ds)
                        if abs(rp["t"] - tp) < 5.0:
                            rescued.append(rp)
            if rescued:
                inl = inl + rescued
                L1 = self.locator.locate(inl, guess=pos, full=False)
                pos = (L1["lat"], L1["lon"], L1["z"])
                inl = [q for i, q in enumerate(inl) if abs(L1["res"][i]) < min(8.0, max(2.5, 2.
5 * L1["sig"][i]))]
        is_plausible = self._plausible(L1, inl, early=False)
        if len(inl) >= self.min_sta and is_plausible:
            L1 = self.locator.locate(inl, guess=pos)          # solución completa con elipse de
 error
            L1["early"] = False
            return L1, inl
        # Quórum Preliminar Temprano: al menos 6 estaciones (con min_sta=8) y convergencia bási
ca
        if len(inl) >= max(6, self.min_sta - 2) and L1.get("rms", 99.0) <= 2.2 and L1.get("gap"
, 360) <= 340:
            if self._plausible(L1, inl, early=True):
                L1 = self.locator.locate(inl, guess=pos)
                L1["early"] = True
                return L1, inl
        return None

    def _twin(self, lat, lon, org, exclude=None):
        for ev in self.events:
            if ev is exclude or not ev["live"]:
                continue
            if abs(ev["org"] - org) < 30 and float(dist_km(ev["lat"], ev["lon"], lat, lon)) < 6
00:
                return ev
        return None

    def _explained(self, ev, picks):
        """Fracción de los arribos P de 'picks' que la solución de 'ev' explica (residuo pequeñ
o)."""
        n = ok = 0
        for p in picks:
            if p["phase"] != "P":
                continue
            d = float(dist_km(ev["lat"], ev["lon"], p["lat"], p["lon"]))
            tp = ev["org"] + self.ttf("P", d, ev["z"]) + self.corr_of(p["key"], "P", d)
            n += 1
            ok += abs(p["t"] - tp) < max(8.0, 0.02 * (tp - ev["org"]))
        return (ok / n if n else 0.0), n

    def _dup_of(self, lat, lon, org, picks, exclude=None):
        """Sismo ya detectado del que estos arribos son, en realidad, parte (v0.4.0): mismo min
uto, a menos
        de 2500 km, y su solución explica al menos 60 % de los arribos. Típico de sismos profun
dos mal
        localizados al principio con pocas estaciones (p. ej. Kermadec/Norfolk del 6-oct-2026).
"""
        for ev in self.events:
            if ev is exclude or not ev["live"]:
                continue
            if abs(ev["org"] - org) > 180 or float(dist_km(ev["lat"], ev["lon"], lat, lon)) > 2
500:
                continue
            frac, n = self._explained(ev, picks)
            if n >= 5 and frac >= 0.6:
```

```
                return ev
        return None

    def _merge_twins(self):
        live = [e for e in self.events if e["live"]]
        for ev in live:
            other = self._twin(ev["lat"], ev["lon"], ev["org"], exclude=ev)
            if other is None and ev in self.events:
                other = self._dup_of(ev["lat"], ev["lon"], ev["org"], ev["picks"], exclude=ev)
            if other is None or other not in self.events or ev not in self.events:
                continue
            keep, drop = (ev, other) if ev["created"] <= other["created"] else (other, ev)
            have = {(p["key"], p["phase"]) for p in keep["picks"]}
            for q in drop["picks"]:
                if (q["key"], q["phase"]) not in have:
                    q["_ev"] = keep["id"]
                    keep["picks"].append(q)
            self.events.remove(drop)
            if self.storage:
                self.storage.delete_event(drop["id"])
            self.log(f"Sismo {drop['id']} unido a {keep['id']} (era el mismo evento)")
            self.relocate(keep)

    def _plausible(self, L1, picks, early=False):
        """Filtros contra asociaciones falsas (disparos de ruido en estaciones lejanas entre sí
    )."""
        rms_max = (2.2 if early else 2.0) if (L1["dmin"] < 1000 or len(picks) < 12) else 3.0
        gap_max = 350 if (len(picks) >= 10 and L1["rms"] <= 0.9 and L1["dmin"] < 800) else 330
        if L1["rms"] > rms_max or L1["gap"] > gap_max:
            return False
        # Filtro de distancia mínima y número de estaciones (ruido lejano y fantasmas oceánicos
     antipodales)
        n_p = len(picks)
        d_min = L1["dmin"]
        # Candado antipodal: ningún sismo oceánico real puede declararse con pocas estaciones s
    i la estación
        # más cercana está a miles de kilómetros sin estaciones intermedias que lo respalden
        if d_min > 1700 and n_p < 14:
            return False
        if d_min > 2200 and n_p < 18:
            return False
        if d_min > 2600:
            return False

        if not early:
            if d_min > 650 and n_p < 25:
                return False
            if d_min > 850 and n_p < 40:
                return False
        # Filtro de consistencia geofísica de profundidad frente a la sismicidad histórica
        if getattr(self, "dprior", None) and L1["z"] > 180:
            hist_z = self.dprior.depths_near(L1["lat"], L1["lon"])
            if hist_z is not None and len(hist_z) >= 15:
                max_hist_z = float(np.max(hist_z))
                if L1["z"] > max_hist_z + 40.0:
                    return False
        d = np.array([float(dist_km(L1["lat"], L1["lon"], p["lat"], p["lon"])) for p in picks])
        # Congruencia de la red: estaciones con datos, más cerca del epicentro que la mediana d
    e las que
        # dispararon, también debieron registrar la P. Un grupo de disparos de ruido deja "huec
    os".
        t_first = min(p["t"] for p in picks)
        keys = {p["key"] for p in picks}
        dref = float(np.median(d))
        near_in_picks = int(np.sum(d <= dref))
        silent_first = expected = hit = 0
        now = time.time()
        for s in self.stations.values():
            if s.key in keys or not s.det or s.det.n < s.det.warm or now - s.last_data > 120:
                continue
            ds = float(dist_km(L1["lat"], L1["lon"], s.lat, s.lon))
            if ds > dref or ds > 100 * KM_DEG:
                continue
            tp = L1["org"] + self.ttf("P", ds, L1["z"]) + self.corr_of(s.key, "P", ds)
            if s.data_end < tp + 5:
                continue                    # sus datos aún no llegan: no se puede juzgar
            ok = any(abs(t - tp) < max(6.0, 0.015 * (tp - L1["org"])) for t in s.trig_times)
            expected += 1
            hit += ok
            if tp < t_first - 10 and not ok:
                silent_first += 1
        ratio = (len(keys) + hit) / (len(keys) + expected)
        # si hay estaciones más cercanas que la mitad de las que dispararon, al menos un tercio
     de
        # ellas (incluyendo las cercanas que sí picaron la P) debe haber registrado la P
        near_ok = expected < 3 or (near_in_picks + hit) >= (near_in_picks + expected) / 3

        # Veto Global de Proximidad por Evidencia Negativa (Distance Quorum Universal):
        # Principio sismológico fundamental: la amplitud decae con la distancia (1/r). Si una o
    nda
        # tuvo energía para disparar estaciones lejanas a d_min_pick (ej. 300-2500 km), cualqui
    er
        # conjunto de estaciones activas más cercanas (d <= 0.55 * d_min_pick) debió registrar
        # amplitudes mucho mayores. Si las estaciones más cercanas están activas y en silencio
        # absoluto (0 confirmaciones), la solución es físicamente un artefacto (cruce de fases)
```

```
.
      d_min_pick = float(np.min(d)) if len(d) else 9999.0
      if d_min_pick > 200.0:
          d_veto_max = min(800.0, 0.55 * d_min_pick)
          near_expected = near_hit = 0
          for s in self.stations.values():
              if s.key in keys or now - getattr(s, "last_data", 0.0) > 120:
                  continue
              ds = float(dist_km(L1["lat"], L1["lon"], s.lat, s.lon))
              if ds <= d_veto_max:
                  tp = L1["org"] + self.ttf("P", ds, L1["z"]) + self.corr_of(s.key, "P", ds)
                  if getattr(s, "data_end", 0.0) >= tp + 6.0:
                      near_expected += 1
                      if any(abs(t - tp) < 8.0 for t in getattr(s, "trig_times", [])):
                          near_hit += 1
          # Si existen 2 o más estaciones activas más cercanas y NINGUNA registró la P: VETO
          if near_expected >= 2 and near_hit == 0:
              return False
          # Si hay 4 o más estaciones cercanas activas y menos del 25% registraron la P: VETO
          if near_expected >= 4 and near_hit < near_expected * 0.25:
              return False

      if d_min_pick > 300.0 and silent_first >= 3:
          return False

      first_ok = silent_first < (6 if early else 3)
      return first_ok and (ratio >= (0.35 if early else 0.4) or (len(keys) >= 7 and L1["rms"]
 <= 1.2 and d_min_pick <= 300)) and near_ok

  def _apply(self, ev, L1):
      for i, p in enumerate(ev["picks"]):
          p["res"] = L1["res"][i]
      for k in ("lat", "lon", "z", "org", "rms", "gap", "edge", "errH", "errMin", "errAz", "e
rrZ", "errOpen", "dmin"):
          ev[k] = L1[k]
      ev["nP"] = sum(1 for p in ev["picks"] if p["phase"] == "P")
      ev["nS"] = sum(1 for p in ev["picks"] if p["phase"] == "S")
      ev["npP"] = sum(1 for p in ev["picks"] if p["phase"] == "pP")
      ev["nsP"] = sum(1 for p in ev["picks"] if p["phase"] == "sP")
      ev["model"] = self.tt.model
      ev["region"] = region_name(round(ev["lat"], 2), round(ev["lon"], 2))

  def relocate(self, ev):
      old_lat, old_lon, old_z = ev.get("lat"), ev.get("lon"), ev.get("z")
      guess = (ev["lat"], ev["lon"], ev["z"])
      zfix = (ev["dpH"], ev.get("dpSig", 15.0)) if ev.get("dpH") is not None else None

      # Descartar picks automáticos que queden fuera del alcance físico permitido (distancia
vs magnitud)
      max_d = self._max_p_dist(ev)
      d_picks = [float(dist_km(ev["lat"], ev["lon"], p["lat"], p["lon"])) for p in ev["picks"
]]
      keep_dist = [p for p, d_k in zip(ev["picks"], d_picks) if p.get("manual") or d_k <= max
_d]
      if len(keep_dist) < len(ev["picks"]) and sum(1 for p in keep_dist if p["phase"] == "P")
 >= 4:
          for p in ev["picks"]:
              if p not in keep_dist:
                  p["_ev"] = None
          ev["picks"] = keep_dist

      L1 = self.locator.locate(ev["picks"], guess=guess, zfix=zfix)
      # Residuo máximo permitido para picks automáticos: 3.2 s o 2.0*sigma
      keep = [p for i, p in enumerate(ev["picks"]) if p.get("manual") or abs(L1["res"][i]) <=
 min(3.2, max(2.0, 2.0 * L1["sig"][i]))]
      if len(keep) < len(ev["picks"]) and sum(1 for p in keep if p["phase"] == "P") >= 4:
          for p in ev["picks"]:
              if p not in keep:
                  p["_ev"] = None
                  if p["phase"] == "S":
                      ev.setdefault("triedS", set()).add(p["key"])
          ev["picks"] = keep
          L1 = self.locator.locate(keep, guess=(L1["lat"], L1["lon"], L1["z"]), zfix=zfix)
      ev["zPrior"] = bool(L1.get("prior"))
      self._apply(ev, L1)

      # Si el hipocentro se desplazó sustancialmente (> 20 km en superficie o > 15 km en prof
undidad),
      # las magnitudes previas calculadas por estación (mb, mbR, Mwp) quedan invalidadas
      # porque la distancia y la atenuación geométrica han cambiado
      if old_lat is not None and old_lon is not None:
          shift_km = float(dist_km(old_lat, old_lon, ev["lat"], ev["lon"]))
          shift_z = abs((old_z or 0.0) - (ev.get("z") or 0.0))
          if shift_km > 20.0 or shift_z > 15.0:
              ml_prev = ev.get("ml")
              nml_prev = ev.get("nml")
              mlComp_prev = ev.get("mlComp")
              for p in ev.get("picks", []):
                  for k in ("_bodyRT", "_bodyT", "_mwpT", "mb_sta", "mbR_sta", "mwp", "mbA",
"mbT", "mB_sta"):
                      p.pop(k, None)
              for k in ("mb", "mbN", "mbSd", "mbRaw", "mbR", "mbRN", "mbRSd", "mwp", "mwpN",
"mwpSd", "mwpTried",
                        "mw", "mwSd", "mwParts", "mwDropped", "mB", "mBN", "ms", "msN", "mag"
, "magType", "magStr"):
```

```
                    ev.pop(k, None)
                if ev.get("ml") is None and ml_prev is not None:
                    ev["ml"] = ml_prev
                    ev["nml"] = nml_prev
                    ev["mlComp"] = mlComp_prev

        if ev.get("early") and len(ev["picks"]) >= self.min_sta:
            if self._plausible(L1, ev["picks"], early=False):
                # Ventana de maduración para redes con latencia de transporte (como red MX vía EarthScope):
                # no consolidar inmediatamente si el sismo es en México/Centroamérica y tiene menos de 60s
                now_t = time.time()
                is_mex = (12.0 <= ev["lat"] <= 33.0 and -118.0 <= ev["lon"] <= -86.0)
                wait_time = 60.0 if is_mex else 30.0
                if now_t - ev["org"] >= wait_time or len(ev["picks"]) >= 12:
                    ev["early"] = False
                    ev["prelim_consolidated"] = True
                    if ev.get("mag") is not None:
                        ev["magPre"] = ev["mag"]
                        ev["magPreType"] = ev.get("magType")
                        ev["magPreStr"] = f"{ev['mag']:.1f}"
                    self.log(f"Preliminar CONSOLIDADO: {self.mag_str(ev)} {where_is(ev['lat'], ev['lon'])} prof {ev['z']:.0f} km ({len(ev['picks'])} est.)")
        self.update_mag(ev)
        self.match_event(ev)
        self.dirty.add(ev["id"])

    # ------------------------------------------------------------------ fases S y P dirigidas
    def _pick_s(self, s, ev):
        N, E = s.comps.get("N"), s.comps.get("E")
        if not N or not E:
            return None
        d = float(dist_km(ev["lat"], ev["lon"], s.lat, s.lon))
        if d > S_DMAX:
            return None
        tp = ev["org"] + self.ttf("P", d, ev["z"])
        ts = ev["org"] + self.ttf("S", d, ev["z"]) + self.corr_of(s.key, "S", d)
        half = max(3, 0.1 * (ts - ev["org"]))
        a, b = max(tp + 0.6 * (ts - tp), ts - half), ts + half
        if min(N.next_t or 0, E.next_t or 0) < b + 3:
            return "wait"
        f1, f2 = (0.7, 5) if d < 800 else (0.3, 2)

        # 1. Picado sobre componente Transversal rotada (aislamiento de onda SH pura y rechazo de coda P)
        try:
            baz = float(bearing(s.lat, s.lon, ev["lat"], ev["lon"]))
            r_T = aic_window_rotated(N.ring, E.ring, baz, f1, f2, a, b, 2.0 if d < 800 else 4.0)
            if r_T and r_T.get("snr", 0) >= 2.5:
                sig_eff = min(1.8, max(0.15, r_T["sigma"]))
                if r_T["snr"] >= 6.0:
                    sig_eff = min(0.20, sig_eff)
                return self._mkpick(s, r_T["t"], "S", sig_eff, snr=r_T["snr"], comp="T", auto=True)
        except Exception:
            pass

        # 2. Respaldo sobre canales N y E estándar si no se alcanzó el umbral en la rotación T
        rs = [r for r in (aic_window(c.ring, f1, f2, a, b, 2.0 if d < 800 else 4.0) for c in (N, E)) if r and r["snr"] >= 2]
        if not rs:
            return None
        if len(rs) == 2 and abs(rs[0]["t"] - rs[1]["t"]) < 1.5:
            t = (rs[0]["t"] + rs[1]["t"]) / 2
            sigma = max(rs[0]["sigma"], rs[1]["sigma"], abs(rs[0]["t"] - rs[1]["t"]) / 2)
        else:
            r = max(rs, key=lambda r: r["snr"])
            t, sigma = r["t"], r["sigma"]
        return self._mkpick(s, t, "S", min(2.0, max(0.2, sigma)), auto=True)

    def _pick_p_targeted(self, s, ev):
        Z = s.comps.get("Z")
        if not Z:
            return None
        d = float(dist_km(ev["lat"], ev["lon"], s.lat, s.lon))
        if d > self._max_p_dist(ev):
            return None
        if 98 * KM_DEG < d < 140 * KM_DEG:       # zona de sombra del núcleo: P difractada débil
            return None
        tp = ev["org"] + self.ttf("P", d, ev["z"]) + self.corr_of(s.key, "P", d)
        win = min(3.5, max(2.0, 0.005 * (tp - ev["org"]) + 0.1 * ev.get("errH", 10) / 8.0))
        if (Z.next_t or 0) < tp + win + 8:
            return "wait"
        if Z.ring.span() is None or Z.ring.span()[0] > tp - win - 50:
            return None                                  # ya salió de la memoria
        if d < 15 * KM_DEG:
            r = aic_window(Z.ring, 1.5, 8, tp - win, tp + win)
        else:
            r = aic_window(Z.ring, 0.5, 2.0, tp - win, tp + win, snr_win=5.0)
        min_snr = 4.2 if d > 15 * KM_DEG else 3.5
        if not r or r["snr"] < min_snr:
            return None
```

```
        return self._mkpick(s, r["t"], "P", max(0.15, r["sigma"]), targeted=True)

    def phase_loop(self):
        with self.lock:
            self._merge_twins()
            # Evaluar estaciones armadas para detección autónoma de onda S
            for s in self.stations.values():
                if getattr(s, "armed_s", None):
                    self._check_autonomous_s(s)
            now = time.time()
            for ev in self.events:
                if not ev["live"]:
                    continue
                if now - ev["created"] > LIVE_S:
                    try:
                        if self.calib.learn(ev):
                            self.log(f"Calibración de magnitudes actualizada con el sismo {ev['id']} (USGS M{ev['match']['mag']} {ev['match'].get('magType')})")
                    except Exception as e:
                        self.log(f"Calibración: {e}")
                    ev["live"] = False
                    self.dirty.add(ev["id"])
                    continue
                if ev.get("final") and self._surface(ev):
                    self._combine(ev)
                    self._preferred(ev)
                    self._tsunami(ev)
                    self.match_event(ev)
                    self.dirty.add(ev["id"])
                changed = False
                if now - ev["created"] < PHASE_S:
                    for s in self.stations.values():
                        if not s.last_data:
                            continue
                        hasP = any(p["key"] == s.key and p["phase"] == "P" for p in ev["picks"])
                        if hasP and s.key not in ev["triedS"] and not any(p["key"] == s.key and p["phase"] == "S" for p in ev["picks"]):
                            r = self._pick_s(s, ev)
                            if r != "wait":
                                ev["triedS"].add(s.key)
                                if r:
                                    s.last_pick = r["t"]
                                    ev["picks"].append(r)
                                    changed = True
                        if not hasP and s.key not in ev["triedP"]:
                            r = self._pick_p_targeted(s, ev)
                            if r != "wait":
                                ev["triedP"].add(s.key)
                                if r:
                                    s.last_pick = r["t"]
                                    ev["picks"].append(r)
                                    changed = True
                    if self.ai is not None:
                        changed = self.ai.refine(self, ev) or changed
                    changed = self._depth_phases(ev, now) or changed
                if changed:
                    self.relocate(ev)
                elif not ev.get("final"):
                    self.update_mag(ev)
                    self.dirty.add(ev["id"])

                # Refinamiento continuo de mecanismo focal en tiempo real (mínimo 4 picks P)
                p_count = sum(1 for p in ev.get("picks", []) if p.get("phase") == "P")
                focal_due = (now - ev.get("last_focal_calc", 0.0) >= 15.0) or (changed and now - ev.get("last_focal_calc", 0.0) >= 5.0)
                if p_count >= 4 and not ev.get("focal_manual") and focal_due:
                    ev["last_focal_calc"] = now
                    try:
                        from . import focal
                        if focal.evaluate_event_focal(self, ev):
                            self.dirty.add(ev["id"])
                    except Exception as ex_foc:
                        pass

                if not ev.get("wfSaved") and now - ev["created"] > 20 * 60:
                    ev["wfSaved"] = True
                    if self.storage:
                        wf = self.section_traces(ev, 0.5, 4, decimate=True)
                        if not wf or any(tr.get("t0", 0) - ev["org"] > 10.0 for tr in wf[:5]):
                            try:
                                from .storage import archive_traces
                                evc = {"org": ev["org"], "picks": [dict(p) for p in ev.get("picks", [])]}
                                prefs = {k: type("P", (), {"pref": getattr(st, "pref", None)})() for k, st in self.stations.items()}
                                wf = archive_traces(os.path.join(self.storage.folder, "archivo"), evc, prefs, 0.5, 4.0)
                            except Exception:
                                pass
                        if wf:
                            self.storage.save_waveforms(ev["id"], wf)
                    try:
                        from . import focal
                        if focal.evaluate_event_focal(self, ev):
```

```
                        self.dirty.add(ev["id"])
                except Exception as ex_foc:
                    self.log(f"Mecanismo focal {ev['id']}: {ex_foc}")

    # ------------------------------------------------------------------ magnitudes
    def update_mag(self, ev, force=False):
        if not ev.get("live") and not force:
            return
        out, pending = [], 0
        for key in {p["key"] for p in ev["picks"] if p["phase"] == "P"}:
            s = self.stations.get(key)
            if not s:
                continue
            d = float(dist_km(ev["lat"], ev["lon"], s.lat, s.lon))
            R = float(np.hypot(d, ev["z"]))
            if d > ML_DMAX:
                continue
            tp = ev["org"] + self.ttf("P", d, ev["z"])
            ts = ev["org"] + self.ttf("S", d, ev["z"])
            t0, t1 = tp - 2, min(tp - 2 + 300, ts + max(30, 0.5 * (ts - tp)))
            H = [k for k in ("N", "E") if k in s.comps and s.comps[k].scale]
            comps = H if len(H) == 2 else (["Z"] if "Z" in s.comps and s.comps["Z"].scale else
[])
            if not comps:
                continue
            mls = []
            for k in comps:
                is_acc = s.comps[k].cha[:2] in ("EN", "HN")
                A = wa_peak(s.comps[k].ring, s.comps[k].scale, t0, t1, is_accel=is_acc)
                if A > 0:
                    mls.append(float(ml_iaspei(A * 1e9, R)))
            if any((s.comps[k].next_t or 0) < t1 for k in comps):
                pending += 1
            if mls:
                out.append((key, float(np.mean(mls)), "H" if len(comps) == 2 else "Z"))
        for p in ev["picks"]:
            if p["phase"] == "P":
                m = [o[1] for o in out if o[0] == p["key"]]
                p["ml"] = m[0] if m else None
        if out:
            vals = np.array([o[1] for o in out])
            ev["ml"] = float(np.median(vals))
            ev["nml"] = len(out)
            ev["mlSd"] = float(np.sqrt(np.mean((vals - ev["ml"]) ** 2))) if len(out) > 1 else N
one
            ev["mlComp"] = "H" if all(o[2] == "H" for o in out) else "H+Z" if any(o[2] == "H" f
or o in out) else "Z"
        pending += self._mwp(ev)
        pending += self._body_mags(ev)
        self._surface(ev)
        self._combine(ev)
        self._preferred(ev)
        self._tsunami(ev)
        was_final = ev.get("final")
        far = any(float(dist_km(ev["lat"], ev["lon"], p["lat"], p["lon"])) > TELE_KM for p in e
v["picks"])
        now_t = time.time()
        age_ok = (now_t - ev["org"] > (16 * 60 if far else 6 * 60)) and (now_t - ev.get("create
d", now_t) >= 60)
        # sismos pequeños en los que ninguna magnitud pudo medirse: se cierran a los 60 min "si
n magnitud"
        no_mag = pending == 0 and ev.get("mag") is None and now_t - ev["org"] > 3600
        ev["noMag"] = bool(no_mag and not was_final) or ev.get("noMag", False)
        ev["final"] = bool(was_final or (pending == 0 and ev.get("mag") is not None and age_ok)
 or no_mag)
        mmin = self.cfg.get("alertas", {}).get("magnitud_minima", 4.0)
        if not force and ((ev.get("mag") is not None and ev["mag"] < mmin) or ev.get("noMag")):
            if ev in self.events:
                self.events.remove(ev)
            if self.storage:
                self.storage.delete_event(ev["id"])
            for p in ev["picks"]:
                p["_ev"] = None
            m_desc = f"M {ev['mag']:.1f}" if ev.get("mag") is not None else "sin magnitud"
            self.log(f"Sismo {ev['id']} descartado por magnitud menor al umbral ({m_desc} < {mm
in})")
            return
        if ev["final"] and not was_final and not ev.get("finalNotified"):
            ev["finalNotified"] = True
            ev.setdefault("tFinal", time.time())      # v0.5.4: hora de la solución final (repo
rte Monitor vs USGS)
            diag = ""
            if ev.get("noMag"):
                ds = [float(dist_km(ev["lat"], ev["lon"], p["lat"], p["lon"])) / KM_DEG for p i
n ev["picks"] if p["phase"] == "P"]
                n_ml = sum(1 for d in ds if d * KM_DEG <= ML_DMAX)
                n_r = sum(1 for d in ds if MBR_DMIN <= d < MBR_DMAX)
                n_t = sum(1 for d in ds if MB_DMIN <= d <= MB_DMAX)
                ev["magDiag"] = (f"estaciones con P: {n_ml} a < 600 km (ML), {n_r} a 5°–20° (mb
 regional, necesita 2), "
                                 f"{n_t} a 20°–100° (mb, necesita 3); ninguna con amplitud medi
ble suficiente")
                diag = f" (sin magnitud: {ev['magDiag']})"
            self.log(f"Solución final: {self.mag_str(ev)} {ev.get('magType') or ''} {where_is(e
v['lat'], ev['lon'])}" + diag)
```

```
            if self.alerts:
                self.alerts.event(self, ev, final=True)

    def _mwp(self, ev):
        """Mwp por estación (3°–95°). Devuelve cuántas estaciones aún esperan datos."""
        vals, pending = [], 0
        for p in ev["picks"]:
            if p["phase"] != "P":
                continue
            s = self.stations.get(p["key"])
            if not s or "Z" not in s.comps or s.band.split(".")[1] not in ("BH", "HH"):
                continue
            d = float(dist_km(ev["lat"], ev["lon"], s.lat, s.lon))
            if not (MWP_DMIN <= d <= MWP_DMAX):
                continue
            c = s.comps["Z"]
            tp = p["t"]
            ts = ev["org"] + self.ttf("S", d, ev["z"])
            if "mwp" in p and p.get("_mwpT") == tp:
                if p["mwp"] is not None:
                    vals.append(p["mwp"])
                continue                        # ya medida (los datos pueden haber salido de la memoria)
            if (c.next_t or 0) < mwp_window(tp, ts, d) + 2:
                pending += 1
                continue
            m = mwp_station(c.ring, c.scale, tp, ts, d)
            p["mwp"] = float(m) if m is not None and np.isfinite(m) else None
            p["_mwpT"] = tp
            if p["mwp"] is not None:
                vals.append(p["mwp"])
        measured = sum(1 for p in ev["picks"] if p["phase"] == "P" and "_mwpT" in p)
        ev["mwpTried"] = measured
        # Mwp solo si la señal domina en buena parte de las estaciones medidas: si apenas unas cuantas pasan
        # el control de ruido (sismo moderado), las que pasan son las de ruido más alto y la Mwp sale inflada
        # v2.0.0: en sismos grandes (Mw de referencia ≥ 5.8 por mB/mb) la señal domina y basta con 5 estaciones
        # y el 20 % de las medidas; antes la regla del 40 % descartaba la Mwp en todos los sismos reales
        ref = max([v for v in (ev.get("mw"), ev.get("mag")) if v is not None] or [0])
        big = ref >= 5.8 and len(vals) >= 5 and len(vals) >= 0.2 * measured
        if (len(vals) >= 3 and len(vals) >= 0.4 * measured) or big:
            ev["mwp"], ev["mwpN"] = float(np.median(vals)), len(vals)
            ev["mwpSd"] = float(np.sqrt(np.mean((np.array(vals) - ev["mwp"]) ** 2)))
        else:
            ev["mwp"] = ev["mwpN"] = ev["mwpSd"] = None
        return pending

    def _depth_phases(self, ev, now):
        """Fases de profundidad pP/sP (ver sismo/depthphases.py). Hasta 3 intentos conforme llegan estaciones."""
        dp = ev.get("depthPhases") or {}
        if dp.get("accepted") or ev.get("_dpTries", 0) >= 3 or now - ev.get("_dpLast", 0) < 20:
            return False
        n = depthphases.ready_count(self, ev)
        if n < 6 or n < 1.5 * ev.get("_dpN", 0):
            return False
        ev["_dpLast"], ev["_dpN"] = now, n
        ev["_dpTries"] = ev.get("_dpTries", 0) + 1
        try:
            res, picks = depthphases.scan(self, ev)
        except Exception as e:
            self.log(f"Fases de profundidad: {e}")
            return False
        if res is None:
            return False
        if not res["accepted"]:
            ev["depthPhases"] = res
            self.dirty.add(ev["id"])
            return False
        ev["zPonly"], ev["errZPonly"] = ev["z"], ev.get("errZ")
        have = {(p["key"], p["phase"]) for p in ev["picks"]}
        added = [q for q in picks if (q["key"], q["phase"]) not in have]
        ev["picks"] += added
        ev["depthPhases"] = res
        ev["dpH"], ev["dpSig"] = float(res["h"]), float(res.get("sigma", 15.0))   # v2.1.0: amarra la profundidad
        self.log(f"Fases de profundidad: h ≈ {res['h']:.0f} km por apilamiento ({res['npP']} pP, {res['nsP']} sP, "
                 f"contraste {res['contrast']}, vía {res.get('via')}); profundidad solo con P: {ev['z']:.0f} ± {ev.get('errZ', 0):.0f} km")
        return True

    def _tsunami(self, ev):
        try:
            ev["tsunami"] = tsu.assess(self, ev, self.landmask)
        except Exception as e:
            self.log(f"Tsunami: {e}")

    def _body_mags(self, ev):
        """mb y mB_BB por estación (20°–100°). Devuelve cuántas estaciones aún esperan datos.""
        mbs, mBs, mbRs, pending = [], [], [], 0
```

```
        for p in ev["picks"]:
            if p["phase"] != "P":
                continue
            s = self.stations.get(p["key"])
            if not s or "Z" not in s.comps:
                continue
            d = float(dist_km(ev["lat"], ev["lon"], s.lat, s.lon))
            deg = d / KM_DEG
            p["dist_deg"] = deg
            c = s.comps["Z"]
            if MBR_DMIN <= deg < MBR_DMAX:
                # mb regional (v0.5.0)
                if p.get("_bodyRT") == p["t"]:
                    if p.get("mbR_sta") is not None:
                        mbRs.append(p["mbR_sta"])
                    continue
                avail = (c.next_t or 0) - p["t"]
                if avail < 32:
                    pending += 1
                    if avail >= 9:
                        r = mb_station(c.ring, c.scale, p["t"], deg, ev["z"], self.calib.qcorr("mbR", deg, ev["z"]), regional=True, win_s=avail - 1.5)
                        if r:
                            p["mbR_sta"] = r["val"]
                            mbRs.append(r["val"])
                    continue
                p["_bodyRT"] = p["t"]
                r = mb_station(c.ring, c.scale, p["t"], deg, ev["z"], self.calib.qcorr("mbR", deg, ev["z"]), regional=True)
                p["mbR_sta"] = r["val"] if r else None
                if r:
                    mbRs.append(r["val"])
                continue
            if not (MB_DMIN <= deg <= MB_DMAX):
                continue
            ts = ev["org"] + self.ttf("S", d, ev["z"])
            if p.get("_bodyT") == p["t"]:                # ya medidas
                if p.get("mb_sta") is not None:
                    mbs.append(p["mb_sta"])
                if p.get("mB_sta") is not None:
                    mBs.append(p["mB_sta"])
                continue
            avail = (c.next_t or 0) - p["t"]
            if (c.next_t or 0) < min(p["t"] + 62, ts - 3):
                pending += 1
                if avail >= 10:
                    r = mb_station(c.ring, c.scale, p["t"], deg, ev["z"], self.calib.qcorr("mb", deg, ev["z"]), win_s=min(30.0, avail - 1.5))
                    if r:
                        p["mb_sta"] = r["val"]
                        p["mbA"], p["mbT"] = r["A"], r["T"]
                        mbs.append(r["val"])
                continue
            p["_bodyT"] = p["t"]
            r = mb_station(c.ring, c.scale, p["t"], deg, ev["z"], self.calib.qcorr("mb", deg, ev["z"]))
            p["mb_sta"] = r["val"] if r else None
            if r:
                p["mbA"], p["mbT"] = r["A"], r["T"]
                if r["val"] <= 7.8:
                    mbs.append(r["val"])
            if s.band and s.band.split(".")[1] in ("BH", "HH"):
                r = mB_station(c.ring, c.scale, p["t"], ts, deg, ev["z"], self.calib.qcorr("mB", deg, ev["z"]))
                p["mB_sta"] = r["val"] if r else None
                if r and r["val"] <= 8.5:
                    mBs.append(r["val"])
        if len(mbRs) >= 2:
            v = np.array([x for x in mbRs if x <= 7.5])
            if len(v) >= 2:
                med = float(np.median(v))
                ev["mbR"], ev["mbRN"], ev["mbRSd"] = med, len(v), float(np.sqrt(np.mean((v - med) ** 2)))
        for key, vals in (("mb", mbs), ("mB", mBs)):
            if len(vals) >= 3:
                v = np.array(vals)
                med = float(np.median(v))
                sd = float(np.sqrt(np.mean((v - med) ** 2)))
                # Control de dispersión y quórum para magnitudes mayores
                if len(vals) <= 4 and sd > 0.85:
                    continue
                if key == "mB" and med >= 6.5 and len(vals) < 4:
                    continue
                ev[key], ev[key + "N"] = med, len(v)
                ev[key + "Sd"] = sd
                if key == "mb":     # v2.0.0: sesgo del mb del sismo aprendido contra el mb del USGS
                    ev["mbRaw"] = med
                    ev["mb"] = med + self.calib.mb_offset()
        return pending

    def _surface(self, ev):
        """Ms_BB por estación con la onda Rayleigh (solo sismos de menos de 60 km de profundidad)."""
        if ev["z"] >= MS_ZMAX:
```

```
            return False
        tried = ev.setdefault("msTried", set())
        changed = False
        for p in ev["picks"]:
            if p["phase"] != "P" or p["key"] in tried:
                continue
            s = self.stations.get(p["key"])
            if not s or "Z" not in s.comps or not s.band or s.band.split(".")[1] not in ("BH",
"HH"):
                tried.add(p["key"])
                continue
            d = float(dist_km(ev["lat"], ev["lon"], s.lat, s.lon))
            deg = d / KM_DEG
            if not (MS_DMIN <= deg <= MS_DMAX):
                tried.add(p["key"])
                continue
            c = s.comps["Z"]
            a, b = rayleigh_window(ev["org"], d)
            if (c.next_t or 0) < b:
                continue                                  # la Rayleigh aún no termina de pasa
r
            tried.add(p["key"])
            r = ms_station(c.ring, c.scale, ev["org"], d, deg, ev["z"])
            p["ms_sta"] = r["val"] if r else None
            changed = True
        vals = [p["ms_sta"] for p in ev["picks"] if p.get("ms_sta") is not None and 2.0 <= p["m
s_sta"] <= 9.0]
        if len(vals) >= 2:
            # Control de dispersión en Ms: si 2 estaciones difieren en > 1.2 grados, no hay con
senso
            if len(vals) == 2 and abs(vals[0] - vals[1]) > 1.2:
                pass
            else:
                v = np.array(vals)
                med = float(np.median(v))
                sd = float(np.sqrt(np.mean((v - med) ** 2)))
                if len(vals) <= 4 and sd > 0.85:
                    pass
                # Para declarar un terremoto mayor por Ms (Ms >= 6.8), se requieren al menos 4
estaciones
                elif med >= 6.8 and len(vals) < 4:
                    pass
                else:
                    ev["ms"], ev["msN"], ev["msSd"] = med, len(v), sd
        return changed

    def _check_coda_contamination(self, ev):
        """Protección contra dobletes: verifica si este sismo ocurrió durante la coda u ondas
        superficiales de un sismo mayor previo (M >= 6.5 a < 3500 km o M >= 7.3 global)."""
        if ev.get("coda_contaminated"):
            return True
        for other in self.events:
            if other is ev or not other.get("live"):
                continue
            dt = ev["org"] - other["org"]
            if 0 < dt < 1500:  # hasta 25 min posteriores al sismo anterior
                m_other = other.get("mag") or other.get("magPre") or 0.0
                d_km = float(dist_km(ev["lat"], ev["lon"], other["lat"], other["lon"]))
                if (m_other >= 6.5 and d_km < 3500) or (m_other >= 7.3):
                    ev["coda_contaminated"] = True
                    return True
        return False

    def _combine(self, ev):
        self._check_coda_contamination(ev)
        r = combine_mw(ev, self.calib.mw_calib())
        if r:
            ev["mw"], ev["mwSd"], ev["mwParts"], ev["mwDropped"] = r
        return r

    @staticmethod
    def _preferred(ev):
        ml, mwp, mw, mb, mbR = ev.get("ml"), ev.get("mwp"), ev.get("mw"), ev.get("mb"), ev.get(
"mbR")
        ev["magWeak"] = False
        ev["magFew"] = False
        # En sismos moderados (mb < 5.0 sin mB ni Mwp grande), la recta de Scordilis (0.85·mb +
 1.03) infla
        # artificialmente la Mw (+0.3 a +0.4); igual que el USGS, se prefiere directamente el m
b medido.
        mod_mb = (mb is not None and mb < 5.0 and ev.get("mB") is None and (mwp is None or mwp
< 5.2))
        # Si la Mw proviene únicamente de Ms moderada (Ms < 5.0, sin mb ni mB), no se usa Scord
ilis (0.67·Ms + 2.07)
        # porque el ruido microsísmico oceánico infla artificialmente los sismos regionales de
M 4.0–4.5.
        only_ms = (mw is not None and set(ev.get("mwParts") or {}) == {"Ms"} and (ev.get("ms")
or 0) < 5.0)
        # v2.0.0: una Mw que sale solo de un mb de 3–4 estaciones es poco confiable (las pocas
que pasan el control de
        # ruido son las de mayor amplitud y el mb sale alto): si hay ML de 2 o más estaciones s
e prefiere la ML;
        # si no, se muestra con «?» (sin bloquear la alerta)
        few_mb = (mw is not None and set(ev.get("mwParts") or {}) == {"mb"} and (ev.get("mbN")
or 0) < 5)
```

```
        if mod_mb and (ev.get("mbN") or 0) >= 5:
            ev["mag"], ev["magType"] = mb, "mb"
        elif mod_mb and ml is not None and (ev.get("nml") or 0) >= 2:
            ev["mag"], ev["magType"] = float(0.5 * (mb + ml)), "mb"
        elif mod_mb:
            ev["mag"], ev["magType"] = mb, "mb"
            ev["magFew"] = (ev.get("mbN") or 0) < 5
        elif only_ms and ml is not None and (ev.get("nml") or 0) >= 2:
            ev["mag"], ev["magType"] = ml, "ML"
        elif only_ms and mbR is not None:
            ev["mag"], ev["magType"] = mbR, "mb(reg)"
        mwp_ok = mwp is not None and (
            (ev.get("mwpN") or 0) >= 3 and (ml is None or ml >= 5.5 or mwp >= 5.5 or (ev.get("nml") or 0) < 3)
            or ((ev.get("mwpN") or 0) >= 2 and mwp >= 6.5)
        )
        if not mod_mb and mw is not None and len(ev.get("mwParts") or {}) >= 2:
            ev["mag"], ev["magType"] = mw, "Mw"
        elif not mod_mb and mwp_ok:
            ev["mag"], ev["magType"] = mwp, "Mwp"
        elif not mod_mb and few_mb and ml is not None and (ev.get("nml") or 0) >= 2:
            ev["mag"], ev["magType"] = ml, "ML"
        elif not mod_mb and mw is not None and not only_ms:
            ev["mag"], ev["magType"] = mw, "Mw"
            ev["magFew"] = few_mb
        elif ml is not None and (ev.get("nml") or 0) >= 2:
            ev["mag"], ev["magType"] = ml, "ML"
        elif mbR is not None:
            # mb regional (v0.5.0): sismos pequeños vistos solo por estaciones a 5°–20°
            if (ev.get("mbRN") or 0) <= 2 and ml is not None and abs(mbR - ml) < 0.8:
                ev["mag"], ev["magType"] = float(0.65 * mbR + 0.35 * ml), "mb(reg)"
            else:
                ev["mag"], ev["magType"] = mbR, "mb(reg)"
        elif ml is not None:
            # una ML de una sola estación no es confiable (v0.4.0): se muestra con "?" y no genera alertas
            ev["mag"], ev["magType"] = ml, "ML"
            ev["magWeak"] = True
        elif mwp is not None and (ev.get("mwpN") or 0) >= 2 and mwp >= 6.0:
            ev["mag"], ev["magType"] = mwp, "Mwp"
            ev["magFew"] = True
        # v0.5.3: se guarda la primera magnitud publicada oficial (la preliminar consolidada)
        if ev.get("mag") is not None and ev.get("magPre") is None and not ev.get("final"):
            if not ev.get("early"):
                ev["magPre"], ev["magPreType"] = ev["mag"], ev.get("magType")
                ev["magPreStr"] = f"{ev['mag']:.1f}" + ("?" if ev.get("magWeak") or ev.get("magFew") else "")
                ev.setdefault("tPrelim", time.time())
        elif ev.get("final") and ev.get("magPre") is None and ev.get("mag") is not None:
            ev["magPre"], ev["magPreType"] = ev["mag"], ev.get("magType")
            ev["magPreStr"] = f"{ev['mag']:.1f}"
            ev.setdefault("tPrelim", ev.get("tEarly", ev.get("created", time.time())))

    @staticmethod
    def mag_of(ev):
        return ev.get("mag")

    def mag_str(self, ev):
        m = self.mag_of(ev)
        if m is None:
            return "s/m" if ev.get("noMag") else "…"
        is_tentative = ev.get("early") or ev.get("magWeak") or ev.get("magFew") or not ev.get("prelim_consolidated", True)
        return f"{m:.1f}" + ("?" if is_tentative else "")

    # ------------------------------------------------------------------ calidad e intensidad
    @staticmethod
    def quality(ev):
        if ev.get("errOpen"):
            return "C"
        nP = ev.get("nP", 0)
        gap = ev.get("gap", 360)
        errH = ev.get("errH", 999)
        errZ = ev.get("errZ", 999)
        rms = ev.get("rms", 999)

        # Calidad A: Cobertura óptima, baja incertidumbre y buena distribución
        if nP >= 12 and gap < 150 and errH < 20 and errZ < 25:
            return "A"
        if nP >= 16 and rms <= 0.8 and gap < 180 and errH < 25:
            return "A"

        # Calidad B: Estándar
        if nP >= 8 and gap < 250 and errH < 40 and errZ < 60:
            return "B"
        if nP >= 8 and rms <= 0.9 and gap < 295 and errH < 55:
            return "B"
        if nP >= 12 and rms <= 1.1 and gap < 310 and errH < 60:
            return "B"

        # Calidad B: Zonas Costeras y de Subducción Marina
        # (Reconoce que en costas y fosas marinas el gap > 295° es geométricamente inevitable)
        if nP >= 8 and rms <= 1.0 and gap < 315 and errH <= 28:
            return "B"
```

```
        # Calidad B: Eventos Oceánicos / Telesísmicos con Consistencia Global
        # (Alta estadística de fases nP >= 18 y residuo temporal excelente RMS <= 0.70 s)
        if nP >= 18 and rms <= 0.70 and errZ <= 50 and errH <= 80:
            return "B"

        return "C"

    def poi_impact(self, ev):
        M = self.mag_of(ev)
        if M is None:
            return []
        out = []
        for p in self.pois:
            d = float(dist_km(ev["lat"], ev["lon"], p["lat"], p["lon"]))
            ts = ev["org"] + self.ttf("S", d, ev["z"])
            I = float(mmi(M, d, ev["z"], p.get("suelo", "firme")))
            out.append({"name": p["nombre"], "lat": p["lat"], "lon": p["lon"], "soil": p.get("suelo", "firme"),
                        "soilName": SOIL.get(p.get("suelo", "firme"), ("?",))[0], "d": d, "I": I, "roman": roman(I),
                        "ts": ts, "remain": ts - time.time()})
        return out

    # ------------------------------------------------------------------ validación USGS
    def set_usgs(self, world_list):
        with self.lock:
            self.usgs = world_list
            for ev in self.events[:300]:
                self.match_event(ev)
                if ev.get("match"):
                    try:
                        self.calib.learn(ev)
                    except Exception:
                        pass

    def calibrate_now(self, usgs_list):
        """Calibración manual («Calibrar ahora»): empareja todos los sismos guardados con el catálogo del USGS
        (lista amplia, no solo 2 días), aprende de los confirmados que aún no se usaron y vuelve a combinar la
        Mw de todos con las conversiones resultantes. Cada sismo se aprende una sola vez."""
        now = time.time()
        c = self.calib
        q0 = {k: sum(v[1] for v in d.values()) for k, d in c.q.items()}
        p0 = {k: len(v) for k, v in c.pairs.items()}
        r = {"revisados": 0, "confirmados": 0, "aprendidos": 0, "ya_usados": 0, "recientes": 0, "sin_usgs": 0,
             "sin_dato_util": 0, "cambios": []}
        saved = self.usgs
        self.usgs = usgs_list
        try:
            for ev in self.events:
                r["revisados"] += 1
                self.match_event(ev)
                m = ev.get("match")
                if not m:
                    r["sin_usgs"] += 1
                    continue
                r["confirmados"] += 1
                if ev["id"] in c.events:
                    r["ya_usados"] += 1
                    continue
                if now - m["t"] < 3600:
                    r["recientes"] += 1
                    continue
                if c.learn(ev):
                    r["aprendidos"] += 1
                else:
                    r["sin_dato_util"] += 1
            for ev in self.events:
                old = self.mag_of(ev)
                if self._combine(ev):
                    self._preferred(ev)
                new = self.mag_of(ev)
                if old is not None and new is not None and abs(new - old) >= 0.05:
                    r["cambios"].append({"id": ev["id"], "lugar": where_is(ev["lat"], ev["lon"]), "antes": round(old, 2),
                                         "ahora": round(new, 2), "usgs": (ev.get("match") or {}).get("mag")})
                    self.dirty.add(ev["id"])
        finally:
            self.usgs = saved
        c.save()
        q1 = {k: sum(v[1] for v in d.values()) for k, d in c.q.items()}
        r["lecturas_q"] = {k: [q0.get(k, 0), q1[k]] for k in q1}
        r["pares"] = {k: [p0.get(k, 0), len(v)] for k, v in c.pairs.items()}
        r["resumen"] = c.summary()
        self.log(f"Calibración manual: {r['confirmados']} sismos confirmados por el USGS, {r['aprendidos']} nuevos aprendidos, "
                 f"{len(r['cambios'])} magnitudes ajustadas")
        return r

    @staticmethod
    def _dm_same(ev):
        """Diferencia directa entre la magnitud final preferente de SIMMSA y la magnitud public
```

```
        ada del USGS."""
1635            u = ev.get("match") or {}
1636            if u.get("mag") is None or ev.get("mag") is None:
1637                return None
1638            return ev["mag"] - u["mag"]
1639
1640        def match_event(self, ev):
1641            best = None
1642            for u in self.usgs:
1643                dt = abs(u["t"] - ev["org"])
1644                if dt > 60:
1645                    continue
1646                d = float(dist_km(u["lat"], u["lon"], ev["lat"], ev["lon"]))
1647                if d < 300:
1648                    sc = dt / 8 + d / 60
1649                    if best is None or sc < best[0]:
1650                        best = (sc, u, d)
1651            old = ev.get("match", {}) or {}
1652            if best:
1653                u, d = best[1], best[2]
1654                m = self.mag_of(ev)
1655                # Comparación directa entre la magnitud final preferente de SIMMSA y la magnitud of
        icial del USGS
1656                diff_mag = (m - u["mag"]) if (m is not None and u.get("mag") is not None) else None
1657                lab = f"{ev.get('magType') or 'M'} SIMMSA vs {u.get('magType') or 'M'} USGS"
1658                ev["match"] = {"id": u["id"], "mag": u["mag"], "magType": u.get("magType"), "lat":
        u["lat"], "lon": u["lon"], "z": u["z"],
1659                               "t": u["t"], "place": u.get("place"), "url": u.get("url"), "updated"
        : u.get("updated"),
1660                               "dEpi": d, "dZ": ev["z"] - u["z"],
1661                               "dM": diff_mag, "dMtype": lab,
1662                               "dMpref": diff_mag, "dT": ev["org"] - u["t"]}
1663                self._learn(ev, u)
1664            else:
1665                # v0.4.0: la lista del USGS cubre solo 2 días; un sismo más viejo (o sin lista, si
        el USGS no
1666                # respondió) conserva la confirmación que ya tenía en lugar de perderla
1667                t0 = min((u["t"] for u in self.usgs), default=None)
1668                if not (old and (t0 is None or ev["org"] < t0 + 3600)):
1669                    ev["match"] = None
1670            ev["tele"] = None
1671            if (ev.get("match") or {}).get("id") != old.get("id"):
1672                ev["_tk_dirty"] = True
1673                self.dirty.add(ev["id"])
1674
1675        def _learn(self, ev, u):
1676            if ev.get("corrLearned") or u["mag"] is None or u["mag"] < 4.5 or not ev["live"]:
1677                return
1678            if time.time() - u["t"] < 1800:
1679                return          # se espera a que la solución del USGS sea más estable (30 min)
1680            ev["corrLearned"] = True
1681            m = self.corr.setdefault(self.tt.model, {})
1682            for p in ev["picks"]:
1683                if p["phase"] != "P" or p.get("manual"):
1684                    continue
1685                d = float(dist_km(u["lat"], u["lon"], p["lat"], p["lon"]))
1686                if 98 * KM_DEG < d:
1687                    continue
1688                r = p["t"] - (u["t"] + self.ttf("P", d, u["z"]))
1689                if abs(r) < 6:
1690                    k = p["key"] + ("|R" if d < TELE_KM else "|T")
1691                    c = m.get(k, [0.0, 0])
1692                    c[0] += r
1693                    c[1] += 1
1694                    if c[1] > 50:
1695                        c[0] *= 50 / c[1]
1696                        c[1] = 50
1697                    m[k] = c
1698            if self.storage:
1699                self.storage.save_json("correcciones.json", self.corr)
1700
1701        def mark_session(self):
1702            now = time.time()
1703            # v0.5.8: cada arranque abre una sesión nueva (un reinicio corta el procesamiento aunqu
        e dure segundos)
1704            if self.sessions and now - self.sessions[-1][1] < 180 and getattr(self, "_session_open"
        , False):
1705                self.sessions[-1][1] = now
1706            else:
1707                self.sessions.append([now, now])
1708                self._session_open = True
1709            self.sessions = [s for s in self.sessions if now - s[1] < 14 * 86400 and s[0] >= self.s
        essions_from][-500:]
1710            if self.storage:
1711                self.storage.save_json("sesiones.json", self.sessions)
1712
1713        def stats(self):
1714            now = time.time()
1715            done = [e for e in self.events if now - e["org"] > 3600]
1716            matched = [e for e in self.events if e.get("match")]
1717            # v0.5.8: cuenta como «encendido» solo si hubo 20 min de datos después del sismo (no ju
        sto antes de apagar)
1718            covered = lambda t: any(a + 600 <= t <= b - 1200 for a, b in self.sessions)
1719            ids = {e["match"]["id"] for e in matched}
1720            missed = [u for u in self.usgs if (u["mag"] or 0) >= self.missed_minmag
```

```
                and now - u["t"] > 1800 and covered(u["t"]) and u["id"] not in ids]
        avg = lambda f: (float(np.mean(v)) if (v := [f(e) for e in matched if f(e) is not None]
) else None)
        # Confiabilidad frente al USGS (Tríada Sismológica Internacional):
        #  · evalúa únicamente los sismos confirmados por el USGS
        #  · 65% Calidad de Magnitud (<=0.1: 100%, 0.1-0.3: 100% a 85%, 0.3-0.5: 85% a 60%, 0.5
-0.8: 60% a 25%)
        #  · 25% Precisión Epicentral (<=20km: 100%, 20-50km: 100% a 85%, 50-100km: 85% a 65%,
100-200km: 65% a 30%)
        #  · 10% Precisión de Tiempo T0 (<=2s: 100%, 2-5s: 100% a 90%, 5-10s: 90% a 70%, 10-25s
: 70% a 30%)
        dms = [abs(self._dm_same(e)) for e in matched if self._dm_same(e) is not None]
        q_list, qm_list, qe_list, qt_list, adv_list = [], [], [], [], []

        def _calc_qm(dm):
            adm = abs(dm)
            if adm <= 0.10:
                return 1.0
            elif adm <= 0.30:
                return 1.0 - (adm - 0.10) / 0.20 * 0.15
            elif adm <= 0.50:
                return 0.85 - (adm - 0.30) / 0.20 * 0.25
            elif adm <= 0.80:
                return 0.60 - (adm - 0.50) / 0.30 * 0.35
            else:
                return max(0.0, 0.25 - (adm - 0.80) / 0.40 * 0.25)

        def _calc_qepi(depi):
            if depi is None:
                return None
            d = float(depi)
            if d <= 20.0:
                return 1.0
            elif d <= 50.0:
                return 1.0 - (d - 20.0) / 30.0 * 0.15
            elif d <= 100.0:
                return 0.85 - (d - 50.0) / 50.0 * 0.20
            elif d <= 200.0:
                return 0.65 - (d - 100.0) / 100.0 * 0.35
            else:
                return max(0.0, 0.30 - (d - 200.0) / 200.0 * 0.30)

        def _calc_qt(dt):
            if dt is None:
                return None
            d = abs(float(dt))
            if d <= 2.0:
                return 1.0
            elif d <= 5.0:
                return 1.0 - (d - 2.0) / 3.0 * 0.10
            elif d <= 10.0:
                return 0.90 - (d - 5.0) / 5.0 * 0.20
            elif d <= 25.0:
                return 0.70 - (d - 10.0) / 15.0 * 0.40
            else:
                return max(0.0, 0.30 - (d - 25.0) / 35.0 * 0.30)

        for e in matched:
            dm = self._dm_same(e)
            if dm is None:
                continue
            qm = _calc_qm(dm)
            depi = (e.get("match") or {}).get("dEpi")
            qe = _calc_qepi(depi) if depi is not None else qm
            dt = (e.get("match") or {}).get("dT")
            qt = _calc_qt(dt) if dt is not None else qm
            qm_list.append(qm)
            qe_list.append(qe)
            qt_list.append(qt)
            q_list.append(0.65 * qm + 0.25 * qe + 0.10 * qt)

            # Ventaja temporal de emisión temprana frente a la publicación automatizada del USG
S (~14 min)
            t_simmsa = e.get("tEarly") or e.get("tPrelim") or e.get("created") or (e["org"] + 6
0)
            lat_s = max(0.5, (t_simmsa - e["org"]) / 60.0)
            u_up = (e.get("match") or {}).get("updated")
            if u_up and u_up > (e.get("match") or {}).get("t", 0):
                lat_u = (u_up - (e.get("match") or {}).get("t", 0)) / 60.0
                if lat_u > 30.0 or lat_u < 4.0:
                    lat_u = 14.0
            else:
                lat_u = 14.0
            adv_min = max(1.0, lat_u - lat_s)
            adv_list.append(adv_min)

        magq = float(np.mean(q_list)) if q_list else None
        qm_avg = float(np.mean(qm_list)) if qm_list else None
        qe_avg = float(np.mean(qe_list)) if qe_list else None
        qt_avg = float(np.mean(qt_list)) if qt_list else None
        adv_avg = float(np.mean(adv_list)) if adv_list else None
        conf = {"total": round(100 * magq) if magq is not None else None,
                "det": None if qe_avg is None else round(100 * qe_avg), "nDet": len(matched), "
detOk": len(matched),
                "mag": None if qm_avg is None else round(100 * qm_avg), "nMag": len(dms),
```

```
                "t0": None if qt_avg is None else round(100 * qt_avg),
                "advMin": round(adv_avg, 1) if adv_avg is not None else None,
                "dMabs": float(np.mean(dms)) if dms else None,
                # v2.1.0: los sismos del monitor que el USGS no confirma no entran en el porcen
taje
                "nSinConf": len([e for e in self.events if not e.get("match")])}
      return {"conf": conf, "detected": len(self.events), "confirmed": len(matched), "unconfi
rmed": len([e for e in done if not e.get("match")]),
                "missed": 0, "missedMag": self.missed_minmag,
                "missedList": [],
                "eEpi": avg(lambda e: e["match"]["dEpi"]), "eZ": avg(lambda e: abs(e["match"]["
dZ"])), "bZ": avg(lambda e: e["match"]["dZ"]),
                "eM": avg(lambda e: e["match"]["dM"]), "eT": avg(lambda e: abs(e["match"]["dT"]
))}

  # ------------------------------------------------------------------ revisión manual
  def manual_pick(self, ev_id, key, phase, t, delete=False):
      with self.lock:
          ev = next((e for e in self.events if e["id"] == ev_id), None)
          s = self.stations.get(key)
          if ev is None:
              return False
          if delete:
              cand = sorted([p for p in ev["picks"] if p["key"] == key], key=lambda p: abs(p[
"t"] - t))
              if not cand:
                  return False
              ev["picks"].remove(cand[0])
              (ev.setdefault("triedS", set()) if cand[0]["phase"] == "S" else ev.setdefault("
triedP", set())).add(key)
          else:
              ref = s or next((p for p in ev["picks"] if p["key"] == key), None)
              if ref is None:
                  return False
              ev["picks"] = [p for p in ev["picks"] if not (p["key"] == key and p["phase"] ==
 phase)]
              lat, lon = (ref.lat, ref.lon) if s else (ref["lat"], ref["lon"])
              ev["picks"].append({"key": key, "code": key.replace("_", "."), "lat": lat, "lon
": lon, "t": float(t),
                                  "phase": phase, "sigma": 0.25, "manual": True})
          ev["manual"] = True
          if sum(1 for p in ev["picks"] if p["phase"] == "P") >= 4:
              self.relocate(ev)
          self.dirty.add(ev["id"])
          return True

  def recalculate_event(self, ev_id, z_fix=None, disabled_keys=None, model=None):
      """Recálculo manual integral de evento:
      - Soporta toggle de exclusión de estaciones (disabled_keys) sin borrarlas
      - Permite fijar profundidad (z_fix) o dejarla libre
      - Re-ejecuta inversión Geiger completa
      - Actualiza distancias reales y residuos para todos los arribos
      - Recalcula todas las magnitudes (ML, mb, mbR, mB, Mwp, Ms, Mw)
      - Recalcula polaridades y mecanismo focal ObsPy
      - Re-evalúa calidad (A/B/C) y persiste en almacenamiento.
      """
      with self.lock:
          ev = next((e for e in self.events if e["id"] == ev_id), None)
          if ev is None and self.storage:
              ev = self.storage.load_event(ev_id)
          if ev is None:
              return None

          disabled_set = set(disabled_keys or [])
          for p in ev.get("picks", []):
              p["disabled"] = (p.get("key") in disabled_set) or (p.get("code") in disabled_se
t)

          active_picks = [p for p in ev.get("picks", []) if not p.get("disabled")]
          if sum(1 for p in active_picks if p.get("phase") == "P") < 4:
              return None

          zfix = None
          if z_fix is not None and str(z_fix).lower() not in ("auto", "none", "", "null"):
              try:
                  zf = float(z_fix)
                  zfix = (zf, 2.0)
                  ev["dpH"] = zf
                  ev["zPrior"] = True
              except Exception:
                  zfix = None
          else:
              ev["dpH"] = None
              ev["zPrior"] = False

          guess = (ev["lat"], ev["lon"], ev["dpH"] if ev.get("dpH") is not None else ev["z"])
          L1 = self.locator.locate(active_picks, guess=guess, zfix=zfix)
          self._apply(ev, L1)

          # Actualizar distancias y residuos de todos los arribos
          for p in ev.get("picks", []):
              d = float(dist_km(ev["lat"], ev["lon"], p["lat"], p["lon"]))
              p["dist"] = d
              p["dist_deg"] = d / KM_DEG
              if not p.get("disabled"):
```

```
                    try:
                        teo = float(self.tt.tt(p["phase"], d, ev["z"]))
                        p["res"] = p["t"] - (ev["org"] + teo + self.corr_of(p["key"], p["phase"], d))
                    except Exception:
                        pass

            # Limpiar mediciones de magnitud previas cacheadas bajo el epicentro anterior
            ml_prev = ev.get("ml")
            nml_prev = ev.get("nml")
            mlComp_prev = ev.get("mlComp")
            for p in ev.get("picks", []):
                for k in ("_bodyRT", "_bodyT", "_mwpT", "mb_sta", "mbR_sta", "mwp", "mbA", "mbT", "mB_sta"):
                    p.pop(k, None)
            for k in ("mb", "mbN", "mbSd", "mbRaw", "mbR", "mbRN", "mbRSd", "mwp", "mwpN", "mwpSd", "mwpTried",
                      "mw", "mwSd", "mwParts", "mwDropped", "mB", "mBN", "ms", "msN", "mag", "magType", "magStr",
                      "magPre", "magPreType", "magPreStr"):
                ev.pop(k, None)

            try:
                self.update_mag(ev, force=True)
            except Exception as e:
                self.log(f"update_mag error in recalculate: {e}")

            # Si el evento no está en el anillo de RAM para ML en vivo pero tenía ML medido en estación regional
            if ev.get("ml") is None and ml_prev is not None:
                ev["ml"] = ml_prev
                ev["nml"] = nml_prev
                ev["mlComp"] = mlComp_prev
                self._preferred(ev)

            ev["where"] = where_is(ev["lat"], ev["lon"])
            ev["magStr"] = self.mag_str(ev)

            # Recalcular mecanismo focal ObsPy
            try:
                from . import focal
                focal.evaluate_event_focal(self, ev)
            except Exception as e:
                self.log(f"Focal recalculate: {e}")

            ev["calidad"] = self.quality(ev)
            ev["manual"] = True

            if self.storage:
                self.storage.save_events([ev])
            self.dirty.add(ev["id"])
            return ev

    # ------------------------------------------------------------------ datos para el panel
    def section_traces(self, ev, f1=0.5, f2=4.0, decimate=False):
        keys = list(dict.fromkeys(p["key"] for p in ev["picks"]))[:60]
        t_end = min(1300, max([90] + [p["t"] - ev["org"] + 90 for p in ev["picks"]]))
        out = []
        for k in keys:
            s = self.stations.get(k)
            c = s.comps.get("Z") if s else None
            if not c:
                continue
            got = c.ring.get(ev["org"] - 60, ev["org"] + t_end)
            if got is None or len(got[1]) < c.fs * 40:
                continue
            t0, data = got
            f = bandpass(data, c.fs, f1, f2)
            skip = max(0, int(round((ev["org"] - 20 - t0) * c.fs)))
            f = f[skip:]
            fs = c.fs
            if decimate and fs > 10:
                q = int(round(fs / 10))
                f = f[::q]
                fs = fs / q
            out.append({"key": k, "t0": t0 + skip / c.fs, "fs": fs, "data": f.astype(np.float32)})
        return out
```

sismo/estreport.py — 206 líneas

```
"""Reporte de estado de las estaciones (v0.4.4): PDF y CSV con estaciones conectadas y sin datos."""
import csv
import io
import json
import os
import time

from .places import where_is


def _fmt(t):
    return time.strftime("%d-%m-%Y %H:%M", time.gmtime(t - 6 * 3600)) if t else "—"


def _dur(s):
    if s is None:
        return "—"
    if s < 120:
        return f"{s:.0f} s"
    if s < 7200:
        return f"{s / 60:.0f} min"
    if s < 172800:
        return f"{s / 3600:.1f} h"
    return f"{s / 86400:.1f} días"


def collect(engine, sources):
    """Filas por estación + resumen. Estados: conectada (datos en los últimos 2 min), intermitente (datos en las
    últimas 24 h), sin datos (suscrita pero sin transmitir), retirada (15 días sin datos, excluida)."""
    now = time.time()
    silent = getattr(engine, "silent", {}) or {}
    rejected = {k for s in sources or [] for k in getattr(s, "rejected", set())}
    rows = []
    with engine.lock:
        for s in engine.stations.values():
            ld = s.last_data or 0
            if ld and now - ld < 120:
                st = "Conectada"
            elif ld and now - ld < 86400:
                st = "Intermitente"
            else:
                st = "Sin datos"
            sil = silent.get(s.key) or {}
            rows.append({"estacion": s.code, "red": s.net, "fuente": getattr(s, "fuente", ""), "lat": s.lat, "lon": s.lon,
                         "ubicacion": where_is(s.lat, s.lon), "canal": s.band or "—",
                         "componentes": "".join(k for k in ("Z", "N", "E") if k in s.comps and now - s.comps[k].last < 120),
                         "estado": st, "ultimo_dato": ld or None,
                         "latencia_s": (now - s.data_end) if (s.data_end and st == "Conectada") else None,
                         "paquetes": s.packets, "sin_datos_desde": sil.get("desde"),
                         "rechazada": s.key in rejected})
        for k, v in silent.items():
            if v.get("retirada") and k not in engine.stations:
                net, sta = (k.split("_", 1) + [""])[:2]
                rows.append({"estacion": f"{net}.{sta}", "red": net, "fuente": "", "lat": None, "lon": None, "ubicacion": "—",
                             "canal": "—", "componentes": "", "estado": "Retirada", "ultimo_dato": None, "latencia_s": None,
                             "paquetes": 0, "sin_datos_desde": v.get("desde"), "rechazada": False})
    order = {"Conectada": 0, "Intermitente": 1, "Sin datos": 2, "Retirada": 3}
    rows.sort(key=lambda r: (order[r["estado"]], r["red"], r["estacion"]))
    nets = {}
    for r in rows:
        d = nets.setdefault(r["red"], {"Conectada": 0, "Intermitente": 0, "Sin datos": 0, "Retirada": 0})
        d[r["estado"]] += 1
    srcs = {}
    for r in rows:
        if r["fuente"]:
            d = srcs.setdefault(r["fuente"], {"Conectada": 0, "total": 0})
            d["total"] += 1
            d["Conectada"] += r["estado"] == "Conectada"
    tot = {k: sum(1 for r in rows if r["estado"] == k) for k in order}
    return {"generado": now, "filas": rows, "redes": nets, "fuentes": srcs, "totales": tot}


def to_csv(data):
    buf = io.StringIO()
    w = csv.writer(buf)
    w.writerow(["estacion", "red", "fuente", "estado", "ubicacion", "latitud", "longitud", "canal", "componentes",
                "ultimo_dato_cdmx", "latencia_s", "paquetes", "sin_datos_desde_cdmx", "rechazada_por_servidor"])
    for r in data["filas"]:
        w.writerow([r["estacion"], r["red"], r["fuente"], r["estado"], r["ubicacion"],
                    "" if r["lat"] is None else f"{r['lat']:.4f}", "" if r["lon"] is None else f"{r['lon']:.4f}",
                    r["canal"], r["componentes"], _fmt(r["ultimo_dato"]) if r["ultimo_dato"] else "",
```

```
                    "" if r["latencia_s"] is None else f"{r['latencia_s']:.0f}", r["paquetes"],
                    _fmt(r["sin_datos_desde"]) if r["sin_datos_desde"] else "", "sí" if r["rechazada"] else ""])
    return "" + buf.getvalue()


def _land(ax, folder):
    path = os.path.join(folder or "", "tierra.geojson")
    try:
        with open(path, encoding="utf-8") as f:
            gj = json.load(f)
    except Exception:
        return
    from matplotlib.patches import Polygon
    for ft in gj.get("features", []):
        g = ft.get("geometry") or {}
        polys = [g["coordinates"]] if g.get("type") == "Polygon" else g.get("coordinates", []) if g.get("type") == "MultiPolygon" else []
        for poly in polys:
            if poly and len(poly[0]) > 2:
                ax.add_patch(Polygon(poly[0], closed=True, fc="#e6e6e6", ec="#b5b5b5", lw=0.3, zorder=0))


def make_pdf(data, path, data_dir=None):
    import matplotlib
    matplotlib.use("Agg")
    import matplotlib.pyplot as plt
    from matplotlib.backends.backend_pdf import PdfPages

    rows, tot = data["filas"], data["totales"]
    colors = {"Conectada": "#1a9a44", "Intermitente": "#e0a000", "Sin datos": "#d62728", "Retirada": "#7f7f7f"}
    with PdfPages(path) as pdf:
        # ---- página 1: resumen y mapa
        fig = plt.figure(figsize=(11, 8.5))
        fig.text(0.05, 0.95, "Monitor Sísmico MUNDIAL — reporte de estado de las estaciones", fontsize=15, weight="bold")
        fig.text(0.05, 0.925, f"Generado: {_fmt(data['generado'])} (hora del centro de México)", fontsize=8, color="#555")
        n = len(rows) or 1
        fig.text(0.05, 0.89, "   ".join(f"{k}: {v} ({100 * v / n:.0f} %)" for k, v in tot.items()) + f"   ·   Total: {len(rows)}",
                 fontsize=10)
        fig.text(0.05, 0.865, "Conectada = datos en los últimos 2 min · Intermitente = datos en las últimas 24 h · Sin datos = suscrita "
                 "sin transmitir (se conserva hasta 15 días) · Retirada = 15 días sin datos", fontsize=7, color="#555")
        ax = fig.add_axes([0.05, 0.33, 0.9, 0.51])
        _land(ax, data_dir)
        for st in ("Retirada", "Sin datos", "Intermitente", "Conectada"):
            pts = [(r["lon"], r["lat"]) for r in rows if r["estado"] == st and r["lat"] is not None]
            if pts:
                ax.scatter([p[0] for p in pts], [p[1] for p in pts], s=10, marker="^", c=colors[st], label=f"{st} ({len(pts)})",
                           edgecolors="none", zorder=3)
        ax.set_xlim(-180, 180)
        ax.set_ylim(-80, 85)
        ax.set_aspect("equal")
        ax.grid(color="#ddd", lw=0.4)
        ax.tick_params(labelsize=7)
        ax.legend(loc="lower left", fontsize=7, framealpha=0.9)
        # tablas por red y por fuente
        ax2 = fig.add_axes([0.05, 0.03, 0.55, 0.27])
        ax2.axis("off")
        nets = sorted(data["redes"].items(), key=lambda x: -sum(x[1].values()))
        cell = [[k, v["Conectada"], v["Intermitente"], v["Sin datos"], v["Retirada"], sum(v.values()),
                 f"{100 * v['Conectada'] / max(1, sum(v.values())):.0f} %"] for k, v in nets[:22]]
        if cell:
            t = ax2.table(cellText=cell, colLabels=["Red", "Conect.", "Interm.", "Sin datos", "Retir.", "Total", "% conect."],
                          loc="upper left", cellLoc="center")
            t.auto_set_font_size(False)
            t.set_fontsize(6.5)
            t.scale(1, 0.95)
        ax2.set_title("Por red" + (" (las 22 con más estaciones; todas en las páginas siguientes)" if len(nets) > 22 else ""),
                      fontsize=8, loc="left")
        ax3 = fig.add_axes([0.64, 0.03, 0.31, 0.27])
        ax3.axis("off")
        cell = [[k, v["Conectada"], v["total"], f"{100 * v['Conectada'] / max(1, v['total']):.0f} %"] for k, v in data["fuentes"].items()]
        if cell:
            t = ax3.table(cellText=cell, colLabels=["Fuente (SeedLink)", "Conectadas", "Total", "%"], loc="upper left", cellLoc="center")
            t.auto_set_font_size(False)
            t.set_fontsize(7)
        ax3.set_title("Por servidor", fontsize=8, loc="left")
        pdf.savefig(fig)
        plt.close(fig)

        # ---- páginas de listas
```

```
        def table_pages(title, sel, cols, fmt_row, widths=None):
            items = [r for r in rows if sel(r)]
            per = 45
            for i in range(0, max(1, len(items)), per):
                fig = plt.figure(figsize=(8.5, 11))
                fig.text(0.05, 0.96, title + (f" (continuación)" if i else f" — {len(items)} estaciones"), fontsize=12, weight="bold")
                ax = fig.add_axes([0.04, 0.03, 0.92, 0.9])
                ax.axis("off")
                chunk = items[i:i + per]
                if chunk:
                    t = ax.table(cellText=[fmt_row(r) for r in chunk], colLabels=cols, loc="upper center", cellLoc="left",
                                 colWidths=widths)
                    t.auto_set_font_size(False)
                    t.set_fontsize(6.5)
                    t.scale(1, 1.12)
                else:
                    ax.text(0.5, 0.9, "Ninguna.", ha="center", fontsize=10)
                pdf.savefig(fig)
                plt.close(fig)

        table_pages("Estaciones conectadas", lambda r: r["estado"] == "Conectada",
                    ["Estación", "Fuente", "Canal", "Comp.", "Latencia", "Paquetes", "Ubicación"],
                    lambda r: [r["estacion"], r["fuente"], r["canal"], r["componentes"] or "—", _dur(r["latencia_s"]),
                               r["paquetes"], r["ubicacion"][:48]],
                    [0.11, 0.11, 0.08, 0.07, 0.09, 0.09, 0.45])
        now = data["generado"]
        table_pages("Estaciones no conectadas (intermitentes, sin datos y retiradas)", lambda r: r["estado"] != "Conectada",
                    ["Estación", "Fuente", "Estado", "Último dato", "Sin datos desde", "Tiempo", "Ubicación"],
                    lambda r: [r["estacion"], r["fuente"] or "—", r["estado"] + (" (rechazada)" if r["rechazada"] else ""),
                               _fmt(r["ultimo_dato"]) if r["ultimo_dato"] else "nunca",
                               _fmt(r["sin_datos_desde"]) if r["sin_datos_desde"] else "—",
                               _dur(now - r["sin_datos_desde"]) if r["sin_datos_desde"] else "—", r["ubicacion"][:36]],
                    [0.10, 0.10, 0.13, 0.13, 0.13, 0.08, 0.33])
        if len(nets) > 22:
            fig = plt.figure(figsize=(8.5, 11))
            fig.text(0.05, 0.96, f"Resumen por red (todas: {len(nets)})", fontsize=12, weight="bold")
            ax = fig.add_axes([0.04, 0.03, 0.92, 0.9])
            ax.axis("off")
            t = ax.table(cellText=[[k, v["Conectada"], v["Intermitente"], v["Sin datos"], v["Retirada"], sum(v.values()),
                                    f"{100 * v['Conectada'] / max(1, sum(v.values())):.0f} %"] for k, v in nets[:60]],
                         colLabels=["Red", "Conectadas", "Intermitentes", "Sin datos", "Retiradas", "Total", "% conectadas"],
                         loc="upper center", cellLoc="center")
            t.auto_set_font_size(False)
            t.set_fontsize(7)
            pdf.savefig(fig)
            plt.close(fig)
    return path
```

sismo/focal.py — 355 líneas

```
"""Módulo de Inversión y Cálculo de Mecanismo Focal (SIMMSA).

Calcula de forma 100% independiente los mecanismos focales (pelotita de playa)
a partir de los registros de sismogramas de las estaciones de la red:
  1. Detección automática de polaridad del primer arribo de onda P (+ Compresión / - Dilatación).
  2. Cálculo de azimut y ángulo de emergencia (take-off angle) según modelo cortical/manto iasp91/ak135.
  3. Inversión por búsqueda de grilla completa (Double-Couple Grid Search, HASH method)
     optimizando Strike (0-360°), Dip (10-90°) y Rake (-180 a +180°).
  4. Cálculo del plano nodal auxiliar, ejes principales de esfuerzo (P y T) y régimen tectónico.
"""

import math
import numpy as np
from .traveltime import R_E, KM_DEG, iasp91
from .locator import dist_km

# Grilla precalculada para optimización rápida (paso de 5° = 88,128 soluciones)
_GRID_STRIKES = np.arange(0, 360, 5)
_GRID_DIPS = np.arange(10, 91, 5)
_GRID_RAKES = np.arange(-180, 180, 5)

_S, _D, _R = np.meshgrid(
    np.radians(_GRID_STRIKES),
    np.radians(_GRID_DIPS),
    np.radians(_GRID_RAKES),
    indexing="ij"
)
_S_RAVEL = _S.ravel()
_D_RAVEL = _D.ravel()
_R_RAVEL = _R.ravel()

# Vectores normales n y de deslizamiento u precalculados
_N_VECS = np.column_stack([
    -np.sin(_D_RAVEL) * np.sin(_S_RAVEL),
    np.sin(_D_RAVEL) * np.cos(_S_RAVEL),
    -np.cos(_D_RAVEL)
])
_U_VECS = np.column_stack([
    np.cos(_R_RAVEL) * np.cos(_S_RAVEL) + np.sin(_R_RAVEL) * np.cos(_D_RAVEL) * np.sin(_S_RAVEL),
    np.cos(_R_RAVEL) * np.sin(_S_RAVEL) - np.sin(_R_RAVEL) * np.cos(_D_RAVEL) * np.cos(_S_RAVEL),
    -np.sin(_R_RAVEL) * np.sin(_D_RAVEL)
])


def calc_azimuth(lat1, lon1, lat2, lon2):
    """Azimut en grados (0-360°) desde (lat1, lon1) hacia (lat2, lon2)."""
    p1, l1 = math.radians(lat1), math.radians(lon1)
    p2, l2 = math.radians(lat2), math.radians(lon2)
    dl = l2 - l1
    y = math.sin(dl) * math.cos(p2)
    x = math.cos(p1) * math.sin(p2) - math.sin(p1) * math.cos(p2) * math.cos(dl)
    return float(math.degrees(math.atan2(y, x)) % 360)


def calc_takeoff(tt, delta_km, z_km):
    """Ángulo de emergencia (take-off angle, grados de 0 a 180 desde el nadir)."""
    p_slowness = float(tt.slowness("P", delta_km, z_km))  # s/km
    r_hypo = max(10.0, R_E - z_km)
    vp_hypo = iasp91(r_hypo)[0]  # km/s
    sin_ih = np.clip(p_slowness * vp_hypo * (R_E / r_hypo), 0.0, 1.0)
    ih_deg = float(np.degrees(np.arcsin(sin_ih)))
    # Si la estación está muy cerca del epicentro (rayo directo que viaja hacia arriba):
    if delta_km < 1.5 * z_km:
        ih_deg = 180.0 - ih_deg
    return ih_deg


def extract_polarity(data, fs, t0, tp):
    """Extrae la polaridad del primer quiebre de onda P sobre la componente Z.
    Retorna (pol, conf, snr):
      pol: '+' (compresión), '-' (dilatación) o None si ambiguo
      conf: float [0.0, 1.0]
      snr: relación señal-ruido
    """
    idx = int(round((tp - t0) * fs))
    if idx < 10 or idx >= len(data) - 10:
        return None, 0.0, 0.0

    # Ruido previo al arribo: 1.5 s
    n_pre = max(5, int(round(1.5 * fs)))
    pre = data[max(0, idx - n_pre):idx]
    noise_std = float(np.std(pre)) if len(pre) > 5 else 1e-6
    if noise_std < 1e-12:
        noise_std = 1e-6

    # Ventana inicial de primer movimiento: 0.35 s
    n_post = max(4, int(round(0.35 * fs)))
    post = data[idx:min(len(data), idx + n_post)]
    if len(post) < 3:
        return None, 0.0, 0.0
```

```
    imax = int(np.argmax(post))
    imin = int(np.argmin(post))
    vmax = float(post[imax])
    vmin = float(post[imin])

    snr = max(vmax, abs(vmin)) / noise_std
    if snr < 1.3:
        return None, 0.0, snr

    pol = None
    if imax < imin and vmax > 1.2 * abs(vmin):
        pol = "+"
    elif imin < imax and abs(vmin) > 1.2 * vmax:
        pol = "-"
    elif vmax > 1.5 * abs(vmin):
        pol = "+"
    elif abs(vmin) > 1.5 * vmax:
        pol = "-"
    else:
        # Pendiente inicial en las primeras 2-3 muestras
        k_eval = min(3, len(post) - 1)
        slope = post[k_eval] - post[0]
        if slope > 1.2 * noise_std:
            pol = "+"
        elif slope < -1.2 * noise_std:
            pol = "-"

    if pol is None:
        return None, 0.0, snr

    conf = min(1.0, snr / 3.5)
    return pol, conf, snr


def auxiliary_plane(strike, dip, rake):
    """Calcula el segundo plano nodal ortogonal (strike2, dip2, rake2)."""
    s = math.radians(strike)
    d = math.radians(dip)
    r = math.radians(rake)

    n = np.array([-math.sin(d) * math.sin(s), math.sin(d) * math.cos(s), -math.cos(d)])
    u = np.array([
        math.cos(r) * math.cos(s) + math.sin(r) * math.cos(d) * math.sin(s),
        math.cos(r) * math.sin(s) - math.sin(r) * math.cos(d) * math.cos(s),
        -math.sin(r) * math.sin(d)
    ])

    if u[2] > 0:
        n2, u2 = -u, -n
    else:
        n2, u2 = u, n

    dip2 = float(math.degrees(math.acos(max(-1.0, min(1.0, -n2[2])))))
    strike2 = float(math.degrees(math.atan2(-n2[0], n2[1])) % 360)

    s2 = math.radians(strike2)
    d2 = math.radians(dip2)
    v_s2 = np.array([math.cos(s2), math.sin(s2), 0.0])
    v_d2 = np.array([-math.cos(d2) * math.sin(s2), math.cos(d2) * math.cos(s2), math.sin(d2)])

    rake2 = float(math.degrees(math.atan2(-np.dot(u2, v_d2), np.dot(u2, v_s2))))
    return round(strike2, 1), round(dip2, 1), round(rake2, 1)


def stress_axes(strike, dip, rake):
    """Calcula los ejes principales de esfuerzo P (Presión) y T (Tensión)."""
    s = math.radians(strike)
    d = math.radians(dip)
    r = math.radians(rake)

    n = np.array([-math.sin(d) * math.sin(s), math.sin(d) * math.cos(s), -math.cos(d)])
    u = np.array([
        math.cos(r) * math.cos(s) + math.sin(r) * math.cos(d) * math.sin(s),
        math.cos(r) * math.sin(s) - math.sin(r) * math.cos(d) * math.cos(s),
        -math.sin(r) * math.sin(d)
    ])

    # T = (u + n)/sqrt(2), P = (u - n)/sqrt(2)
    T = (u + n) / math.sqrt(2.0)
    P = (u - n) / math.sqrt(2.0)

    def _vec_to_trend_plunge(v):
        if v[2] < 0:
            v = -v
        plunge = math.degrees(math.asin(max(0.0, min(1.0, v[2]))))
        trend = math.degrees(math.atan2(v[1], v[0])) % 360
        return round(trend, 1), round(plunge, 1)

    t_az, t_pl = _vec_to_trend_plunge(T)
    p_az, p_pl = _vec_to_trend_plunge(P)
    return {"t_axis": {"az": t_az, "plunge": t_pl}, "p_axis": {"az": p_az, "plunge": p_pl}}


def classify_regime(rake):
```

```
    """Clasifica el régimen sismotectónico según el ángulo de deslizamiento (Rake)."""
    if -135 <= rake <= -45:
        return "Normal"
    elif 45 <= rake <= 135:
        return "Inversa / Cabalgamiento"
    elif abs(rake) <= 30 or abs(rake) >= 150:
        return "Rumbo / Desgarre"
    elif -45 < rake < 45:
        return "Oblicuo Rumbo-Inversa" if rake > 0 else "Oblicuo Rumbo-Normal"
    else:
        return "Oblicuo"


def solve_focal_mechanism(stations_pol):
    """Inversión de mecanismo focal por búsqueda de grilla completa.
    stations_pol: lista de dicts con:
      az: azimut (grados)
      takeoff: take-off angle (grados)
      pol: '+' o '-'
      weight: peso float [0.1, 1.0]
    """
    if not stations_pol:
        return None

    sta_az = np.radians([s["az"] for s in stations_pol])
    sta_th = np.radians([s["takeoff"] for s in stations_pol])
    obs_pol = np.array([1.0 if s["pol"] == "+" else -1.0 for s in stations_pol])
    weights = np.array([max(0.1, s.get("weight", 1.0)) for s in stations_pol])

    # Vector de rayo gamma (N, E, D)
    gamma = np.column_stack([
        np.sin(sta_th) * np.cos(sta_az),
        np.sin(sta_th) * np.sin(sta_az),
        np.cos(sta_th)
    ])

    gn = np.dot(gamma, _N_VECS.T)
    gu = np.dot(gamma, _U_VECS.T)
    Ap = 2.0 * gn * gu

    pred_sign = np.where(Ap >= 0, 1.0, -1.0)
    misfit = np.dot(weights, (pred_sign != obs_pol[:, None]).astype(float))

    best_idx = int(np.argmin(misfit))
    strike = float(np.degrees(_S_RAVEL[best_idx]))
    dip = float(np.degrees(_D_RAVEL[best_idx]))
    rake = float(np.degrees(_R_RAVEL[best_idx]))

    strike2, dip2, rake2 = auxiliary_plane(strike, dip, rake)
    axes = stress_axes(strike, dip, rake)
    regime = classify_regime(rake)

    w_sum = float(weights.sum()) or 1.0
    misfit_score = float(misfit[best_idx])
    misfit_pct = float((misfit_score / w_sum) * 100.0)

    # Calidad según discordancia y número de estaciones
    n_sta = len(stations_pol)
    if misfit_pct <= 12.0 and n_sta >= 10:
        quality = "A"
    elif misfit_pct <= 20.0 and n_sta >= 6:
        quality = "B"
    elif misfit_pct <= 30.0:
        quality = "C"
    else:
        quality = "D"

    # Marcar polaridades predichas vs observadas en cada estación
    best_pred = pred_sign[:, best_idx]
    for i, s in enumerate(stations_pol):
        s["pol_pred"] = "+" if best_pred[i] > 0 else "-"
        s["fit"] = bool(s["pol"] == s["pol_pred"])

    return {
        "strike": round(strike, 1),
        "dip": round(dip, 1),
        "rake": round(rake, 1),
        "strike2": strike2,
        "dip2": dip2,
        "rake2": rake2,
        "regime": regime,
        "quality": quality,
        "misfit_pct": round(misfit_pct, 1),
        "n_stations": n_sta,
        "p_axis": axes["p_axis"],
        "t_axis": axes["t_axis"],
        "stations": stations_pol
    }


def evaluate_event_focal(engine, ev):
    """Evalúa un sismo completo extrayendo polaridades y calculando el mecanismo focal SIMMSA."
  ""
    picks = [p for p in ev.get("picks", []) if p.get("phase") == "P"]
    if len(picks) < 4:
```

```
        return None

    # Intentar obtener trazas desde el almacenamiento o en vivo
    traces_by_key = {}
    if engine.storage:
        loaded = engine.storage.load_waveforms(ev["id"])
        for tr in loaded:
            traces_by_key[tr["key"]] = tr

    # Si faltan trazas y el evento está en vivo, consultar los rings de las estaciones
    for p in picks:
        k = p["key"]
        if k in traces_by_key:
            continue
        s = getattr(engine, "stations", {}).get(k) if hasattr(engine, "stations") and engine.st
ations else None
        if s and "Z" in s.comps:
            c = s.comps["Z"]
            tp = p["t"]
            got = c.ring.get(tp - 5.0, tp + 5.0)
            if got is not None and len(got[1]) > c.fs * 3:
                from .dsp import bandpass
                filt = bandpass(got[1], c.fs, 1.0, 8.0)
                traces_by_key[k] = {"key": k, "t0": got[0], "fs": c.fs, "data": filt}

    stations_pol = []
    for p in picks:
        k = p["key"]
        tr = traces_by_key.get(k)
        if not tr:
            continue
        pol, conf, snr = extract_polarity(tr["data"], tr["fs"], tr["t0"], p["t"])
        if not pol:
            continue

        p["pol"] = pol
        p["pol_conf"] = round(conf, 2)
        p["snr"] = round(snr, 1)

        sta_lat, sta_lon = p["lat"], p["lon"]
        delta_km = float(dist_km(ev["lat"], ev["lon"], sta_lat, sta_lon))
        azimuth = calc_azimuth(ev["lat"], ev["lon"], sta_lat, sta_lon)
        takeoff = calc_takeoff(engine.tt, delta_km, ev["z"])

        stations_pol.append({
            "key": k,
            "net": p.get("net", k.split("_")[0] if "_" in k else ""),
            "sta": p.get("sta", k.split("_")[1] if "_" in k else k),
            "pol": pol,
            "weight": max(0.2, conf),
            "az": round(azimuth, 1),
            "takeoff": round(takeoff, 1),
            "delta_km": round(delta_km, 1),
            "snr": round(snr, 1)
        })

    if len(stations_pol) < 4:
        return None

    solution = solve_focal_mechanism(stations_pol)
    if solution:
        ev["mecanismo_focal"] = solution
        ev["strike"] = solution["strike"]
        ev["dip"] = solution["dip"]
        ev["rake"] = solution["rake"]
        ev["strike2"] = solution["strike2"]
        ev["dip2"] = solution["dip2"]
        ev["rake2"] = solution["rake2"]
        ev["focal_regime"] = solution["regime"]
        ev["focal_quality"] = solution["quality"]
        ev["focal_misfit"] = solution["misfit_pct"]
        ev["focal_source"] = "SIMMSA_P_POL"
        ev["p_axis"] = solution["p_axis"]
        ev["t_axis"] = solution["t_axis"]
        ev["focal_stations"] = solution["stations"]

    return solution
```

sismo/health.py — 396 líneas

```
"""Monitoreo del estado del sistema: revisa cada capa y cada parte del monitor.

Cada revisión devuelve un estado:
  "ok"       punto verde: funciona bien
  "falla"    punto rojo: se detectó un problema (el mensaje dice cuál y qué hacer)
  "inactivo" punto gris: la parte está desactivada en config.json o todavía no aplica

Las partes que corren en segundo plano (ciclo principal, fases, USGS, guardado, tickets, alertas)
reportan aquí su último latido o su último error (Health.beat / Health.fail).
"""
import os
import shutil
import sqlite3
import statistics
import threading
import time


class Health:
    def __init__(self):
        self.lock = threading.Lock()
        self.items = {}          # nombre -> {"t_ok", "t_fail", "msg", "detail"}

    def beat(self, name, detail=""):
        with self.lock:
            it = self.items.setdefault(name, {})
            it["t_ok"] = time.time()
            if detail:
                it["detail"] = detail

    def fail(self, name, msg):
        with self.lock:
            it = self.items.setdefault(name, {})
            it["t_fail"] = time.time()
            it["msg"] = str(msg)[:300]

    def get(self, name):
        with self.lock:
            return dict(self.items.get(name, {}))


def _ago(t, now):
    if not t:
        return "nunca"
    s = now - t
    return f"hace {s:.0f} s" if s < 120 else f"hace {s / 60:.0f} min" if s < 7200 else f"hace {s / 3600:.1f} h"


def _rss_mb():
    """Memoria del proceso (MB) sin dependencias extra."""
    try:
        import psutil
        return psutil.Process().memory_info().rss / 1e6
    except Exception:
        pass
    if os.name == "nt":
        try:
            import ctypes
            from ctypes import wintypes

            class PMC(ctypes.Structure):
                _fields_ = [("cb", wintypes.DWORD), ("PageFaultCount", wintypes.DWORD), ("PeakWorkingSetSize", ctypes.c_size_t),
                            ("WorkingSetSize", ctypes.c_size_t), ("QuotaPeakPagedPoolUsage", ctypes.c_size_t),
                            ("QuotaPagedPoolUsage", ctypes.c_size_t), ("QuotaPeakNonPagedPoolUsage", ctypes.c_size_t),
                            ("QuotaNonPagedPoolUsage", ctypes.c_size_t), ("PagefileUsage", ctypes.c_size_t),
                            ("PeakPagefileUsage", ctypes.c_size_t)]
            pmc = PMC()
            pmc.cb = ctypes.sizeof(PMC)
            h = ctypes.windll.kernel32.GetCurrentProcess()
            if ctypes.windll.psapi.GetProcessMemoryInfo(h, ctypes.byref(pmc), pmc.cb):
                return pmc.WorkingSetSize / 1e6
        except Exception:
            return None
    try:
        import resource
        return resource.getrusage(resource.RUSAGE_SELF).ru_maxrss / 1e3
    except Exception:
        return None


def _hb(h, name, now, max_age, label_ok, never_msg):
    """Revisión por latido: verde si hubo latido reciente y no hay un error posterior."""
    it = h.get(name)
    t_ok, t_fail = it.get("t_ok"), it.get("t_fail")
    if t_fail and (not t_ok or t_fail > t_ok):
        return "falla", f"Error {_ago(t_fail, now)}: {it.get('msg', '')}"
    if not t_ok:
        return "falla", never_msg
    if now - t_ok > max_age:
```

```
        return "falla", f"Sin actividad desde {_ago(t_ok, now)} (se espera cada {max_age:.0f} s o menos)"
    return "ok", label_ok + (f" · {it['detail']}" if it.get("detail") else "") + f" · {_ago(t_ok, now)}"


def report(ctx):
    """ctx: dict con engine, alerts, sources, queue, storage, cfg, data_dir, started, health, log_path."""
    e, h, cfg = ctx["engine"], ctx["health"], ctx["cfg"]
    now = time.time()
    up = now - ctx["started"]
    warm = up < 300                       # primeros 5 min: las estaciones apenas se están conectando
    layers = []

    def add(layer, name, state, msg):
        for L in layers:
            if L["capa"] == layer:
                L["items"].append({"nombre": name, "estado": state, "detalle": msg})
                return
        layers.append({"capa": layer, "items": [{"nombre": name, "estado": state, "detalle": msg}]})

    with e.lock:
        st = list(e.stations.values())
        online = [s for s in st if s.last_data and now - s.last_data < 120]
        lat = [now - s.data_end for s in online if s.data_end]
        warmed = [s for s in online if s.det and s.det.n >= s.det.warm]
        trig1h = [sum(1 for t in s.trig_times if now - t < 3600) for s in online]
        events = list(e.events[:20])
        assoc_ok = e.assoc.keys is not None and len(e.assoc.keys) == len(e.stations)

    # ---------------- Capa 1: adquisición
    L1 = "1 · Adquisición de datos"
    for src in ctx["sources"]:
        name = f"Conexión SeedLink {getattr(src, 'label', src.server)}"
        acc, rej = len(getattr(src, "accepted", ())), len(getattr(src, "rejected", ()))
        if src.last_data and now - src.last_data < 120:
            add(L1, name, "ok", f"Recibiendo datos de {src.server} · {src.packets} paquetes · {acc} estaciones aceptadas"
                + (f", {rej} rechazadas" if rej else ""))
        else:
            add(L1, name, "falla" if not (warm or getattr(src, "config_fail", 0)) else "inactivo",
                (f"Conectando con {src.server}…" if warm else
                 (f"{src.server} acepta la conexión pero no responde a la suscripción de sus {len(src.streams)} estaciones "
                  "(no las ofrece en tiempo real o restringe el acceso). Las demás fuentes no se afectan; se puede "
                  "desactivar en config.json → fuentes.")
                 if getattr(src, "config_fail", 0) else
                 f"Sin datos de {src.server} {_ago(src.last_data, now) if src.last_data else 'desde el arranque'}; "
                 "revisa internet o firewall (puerto 18000).").")
    if not ctx["sources"]:
        add(L1, "Conexiones SeedLink", "falla", "No hay ninguna fuente activa en config.json → fuentes.")
    meta = h.get("metadatos")
    if st:
        add(L1, "Metadatos de estaciones (FDSN)", "ok" if meta.get("t_ok") else "falla",
            f"{len(st)} estaciones configuradas" + (f" · {meta.get('detail')}" if meta.get("detail") else "")
            + ("" if meta.get("t_ok") else f" · {meta.get('msg', 'se usó la lista guardada')}"))
    else:
        add(L1, "Metadatos de estaciones (FDSN)", "falla", "No hay estaciones configuradas.")
    if st:
        frac = len(online) / len(st)
        state = "ok" if frac >= 0.5 else ("inactivo" if warm else "falla")
        add(L1, "Estaciones en línea", state,
            f"{len(online)} de {len(st)} ({frac * 100:.0f} %)" + ("" if frac >= 0.5 else
                                                                  " · menos de la mitad envía datos" + (" (arrancando)" if warm else "")))
    if lat:
        med = statistics.median(lat)
        add(L1, "Latencia de los datos", "ok" if med < 90 else "falla",
            f"Mediana {med:.0f} s" + ("" if med < 90 else " · los datos llegan tarde; revisa el ancho de banda"))
    q = ctx.get("queue")
    if q is not None:
        n = q.qsize()
        add(L1, "Cola de procesamiento", "ok" if n < 5000 else "falla",
            f"{n} paquetes en espera" + ("" if n < 5000 else " · la PC no alcanza a procesar; baja estaciones.max_estaciones"))
    if cfg.get("archivo", {}).get("activo"):
        errs = [getattr(s, "_arch_err", 0) for s in ctx["sources"]]
        arch = next((getattr(s, "archive", None) for s in ctx["sources"] if getattr(s, "archive", None) is not None), None)
        if arch is not None:
            errs.append(getattr(arch, "error_t", 0))
        recent = any(t and now - t < 1800 for t in errs)
        stale = arch is not None and arch.last_flush and now - arch.last_flush > 120
        gb = (getattr(arch, "bytes_written", 0) or 0) / 1e9
        dias = cfg.get("archivo", {}).get("dias", 2)
```

```
        add(L1, "Archivo miniSEED continuo", "falla" if (recent or stale) else "ok",
            "Error al escribir en datos\\archivo en los últimos 30 min (ver monitor.log)" if re
cent else
            "No se ha escrito en más de 2 min" if stale else
            f"Escribiendo en datos\\archivo · {gb:.2f} GB desde el arranque · se conservan {dia
s} días")
      else:
        add(L1, "Archivo miniSEED continuo", "inactivo", "Desactivado en config.json → archivo"
)

      # ---------------- Capa 2: procesamiento de señal
      L2 = "2 · Procesamiento de señal"
      if online:
        f = len(warmed) / len(online)
        add(L2, "Detectores STA/LTA (local y lejano)", "ok" if f >= 0.5 or warm else "falla",
            f"{len(warmed)} de {len(online)} estaciones con detector listo" + (" (calentando, 2
 min)" if warm and f < 0.5 else ""))
        med = statistics.median(trig1h) if trig1h else 0
        noisy = sum(1 for x in trig1h if x > 40)
        add(L2, "Tasa de disparos", "ok" if med <= 20 else "falla",
            f"Mediana {med:.0f} disparos/h por estación" + (f" · {noisy} estaciones muy ruidosa
s" if noisy else "")
            + ("" if med <= 20 else " · demasiado ruido: sube deteccion.umbral_sta_lta"))
      else:
        add(L2, "Detectores STA/LTA (local y lejano)", "inactivo" if warm else "falla", "Sin es
taciones en línea")
      if cfg.get("ia", {}).get("activa"):
        add(L2, "Picado con IA (PhaseNet)", "ok" if e.ai is not None else "falla",
            "Modelo cargado" if e.ai is not None else "No se pudo cargar; ejecuta instalar_ia.b
at")
      else:
        add(L2, "Picado con IA (PhaseNet)", "inactivo", "Desactivado en config.json → ia")

      # ---------------- Capa 3: detección y asociación
      L3 = "3 · Detección y asociación"
      add(L3, "Ciclo principal del motor", *_hb(h, "ciclo", now, 15, "Funcionando", "El ciclo pri
ncipal no ha reportado"))
      add(L3, "Fases dirigidas y magnitudes (cada 3 s)", *_hb(h, "fases", now, 20, "Funcionando",
 "El bucle de fases no ha corrido"))
      add(L3, "Asociador global", "ok" if assoc_ok else "falla",
        f"Tablas listas para {len(e.assoc.keys or [])} estaciones" if assoc_ok else "Tablas sin
 construir o desactualizadas")

      # ---------------- Capa 4: localización
      L4 = "4 · Localización"
      tabs = getattr(e.tt, "tab", {}) or {}
      need = {"P", "S", "pP", "sP"}
      add(L4, "Tablas de tiempos de viaje", "ok" if need <= set(tabs) else "falla",
        f"{e.tt.name} · fases {', '.join(sorted(tabs))}" if need <= set(tabs) else f"Faltan fas
es: {', '.join(sorted(need - set(tabs)))}")
      if "TauP" not in e.tt.name and cfg.get("deteccion", {}).get("modelo_velocidades", "ak135")
!= "iasp91-propio":
        add(L4, "Modelo ak135 (TauP)", "falla", "Se está usando el respaldo iasp91: ObsPy/TauP
no disponible. Repite instalar.bat.")
      else:
        add(L4, "Modelo ak135 (TauP)", "ok", "En uso")
      loc = [ev for ev in events if ev.get("rms") is not None]
      if loc:
        # Un solo sismo con residuo algo alto no es una falla del sistema (en telesismos con po
cas
        # estaciones un RMS de 3-4 s es habitual). Se marca rojo solo si el localizador falla d
e forma
        # sistemática: mediana del RMS de los últimos 5 sismos > 4 s, o el último quedó sin cer
rar.
        ev = loc[0]
        med = float(statistics.median([x["rms"] for x in loc[:5]]))
        bad = bool(ev.get("errOpen")) or (len(loc) >= 3 and med > 4.0)
        txt = f"{ev['id']}: RMS {ev['rms']:.2f} s, calidad {e.quality(ev)}"
        if ev.get("errOpen"):
            txt += " · solución sin cerrar (error abierto)"
        elif bad:
            txt += f" · RMS mediano de los últimos sismos {med:.1f} s: revisar estaciones/corre
cciones"
        elif ev["rms"] > 3.0:
            txt += " · residuo algo alto, aceptable para un telesismo"
        add(L4, "Última localización", "falla" if bad else "ok", txt)
      else:
        add(L4, "Última localización", "inactivo", "Todavía no hay sismos")
      dp = next((ev.get("depthPhases") for ev in events if ev.get("depthPhases")), None)
      add(L4, "Fases de profundidad pP/sP", "ok" if dp is not None else "inactivo",
        ("Última búsqueda: " + ("aceptada" if dp.get("accepted") else "no concluyente (normal e
n sismos superficiales o pequeños)"))
        if dp is not None else "Se activa con sismos registrados a 25°–95°")

      # ---------------- Capa 5: magnitudes
      L5 = "5 · Magnitudes"
      mag = [ev for ev in events if ev.get("live") or ev.get("final")]
      if events:
        ev = events[0]
        has = [k for k, lab in (("mwp", "Mwp"), ("mb", "mb"), ("mB", "mB"), ("ms", "Ms"), ("ml"
, "ML")) if ev.get(k) is not None]
        sd = ev.get("mwSd")
        # v0.5.1: un sismo pequeño sin magnitud (s/m) es normal; solo es falla si la mayoría de
 los recientes no la tienen
        recent = [x for x in events[:10] if x.get("final") or not x.get("live")]
```

```
        nomag = [x for x in recent if x.get("mag") is None]
        systemic = len(recent) >= 5 and len(nomag) / len(recent) > 0.6
        bad = systemic or (sd is not None and sd > 0.6)
        if ev.get("mag") is not None:
            txt = f"M {ev['mag']:.1f} {ev.get('magType', '')} con " + ", ".join(has)
        else:
            txt = ("Último sismo sin magnitud (s/m): señal débil o estaciones con ganancia dudosa; normal en sismos pequeños"
                   + (f" · últimos {len(recent)}: {len(recent) - len(nomag)} con magnitud" if recent else ""))
        if systemic:
            txt += f" · {len(nomag)} de {len(recent)} sismos recientes sin magnitud: revisa ganancias/metadatos"
        add(L5, "Cálculo de magnitudes (último sismo)", "falla" if bad else ("ok" if ev.get("mag") is not None else "inactivo"),
            txt
            + (f" · dispersión alta ± {sd:.1f}: revisa el ticket" if sd is not None and sd > 0.6 else "")
            + (f" · descartada: {', '.join(ev['mwDropped'])}" if ev.get("mwDropped") else ""))
    else:
        add(L5, "Cálculo de magnitudes (último sismo)", "inactivo", "Todavía no hay sismos")
    try:
        cs = e.calib.summary()
        fitted = [r["tipo"] for r in cs["conversiones"] if r["origen"].startswith("ajustada")]
        add(L5, "Calibración contra el USGS", "ok",
            f"Lecturas de Q: mb {cs['q_lecturas'].get('mb', 0)}, mB {cs['q_lecturas'].get('mB', 0)} · conversiones ajustadas: "
            + (", ".join(fitted) if fitted else "ninguna todavía (se necesitan 15 sismos con Mw del USGS)"))
    except Exception as ex:
        add(L5, "Calibración contra el USGS", "falla", f"No se pudo leer: {ex}")

    # ---------------- Capa 6: impacto
    L6 = "6 · Impacto (lugares, intensidad, tsunami, zonas)"
    try:
        from .places import region_name, where_is
        where_is(19.43, -99.13)
        reg = region_name(19.43, -99.13)
        add(L6, "Ciudades de referencia", "ok", "465 ciudades cargadas")
        add(L6, "Regiones Flinn-Engdahl (ObsPy)", "ok" if reg else "falla",
            f"Funciona (ej. {reg})" if reg else "No disponible: los nombres de región salen vacíos. Revisa la instalación de ObsPy.")
    except Exception as ex:
        add(L6, "Ciudades y regiones", "falla", str(ex))
    lm = getattr(e, "landmask", None)
    if (lm is None or lm.rings is None) and warm:
        add(L6, "Contorno tierra/mar (tsunami)", "inactivo", "Descargando el contorno de Natural Earth (solo la primera vez)…")
    else:
       add(L6, "Contorno tierra/mar (tsunami)", "ok" if lm is not None and lm.rings is not None else "falla",
           "Natural Earth cargado" if lm is not None and lm.rings is not None else
           "No descargado: el aviso de tsunami usa la región Flinn-Engdahl. Revisa internet; se reintenta al reiniciar.")
    zones = cfg.get("alertas", {}).get("zonas") or []
    add(L6, "Zonas de alerta", "ok" if zones else "falla",
        f"{len(zones)} zonas configuradas" if zones else "Sin zonas: se usa la magnitud mínima general")
    pf = os.path.join(ctx.get("data_dir") or "", "placas_pb2002.json")
    if os.path.exists(pf):
        add(L6, "Placas tectónicas (mapa)", "ok", "Límites PB2002 cargados: primarios, secundarios y microplacas")
    else:
        add(L6, "Placas tectónicas (mapa)", "inactivo" if warm else "falla",
            "Descargando los límites PB2002…" if warm else "No se pudieron descargar los límites de placas (sin acceso a GitHub)")

    # ---------------- Capa 7: validación
    L7 = "7 · Validación y calibración (USGS)"
    add(L7, "Catálogo del USGS", *_hb(h, "usgs", now, 600, "Actualizado", "Aún no se ha consultado (cada 3 min)"))
    try:
        nc = sum(len(v) for v in e.corr.values())
        add(L7, "Correcciones por estación", "ok", f"{nc} correcciones aprendidas")
    except Exception as ex:
        add(L7, "Correcciones por estación", "falla", str(ex))
    st_ = e.stats()
    if st_["confirmed"]:
        add(L7, "Coincidencia con el USGS", "ok" if (st_["eEpi"] or 0) < 60 else "falla",
            f"{st_['confirmed']} sismos confirmados · error epicentral medio {st_['eEpi'] or 0:.0f} km"
            + (f" · sesgo de magnitud {st_['eM']:+.2f}" if st_.get("eM") is not None else ""))
    else:
        add(L7, "Coincidencia con el USGS", "inactivo", "Todavía no hay sismos confirmados")

    # ---------------- Capa 8: persistencia
    L8 = "8 · Almacenamiento"
    try:
        with sqlite3.connect(ctx["storage"].db_path, timeout=5) as c:
            n = c.execute("SELECT COUNT(*) FROM sismos").fetchone()[0]
        add(L8, "Base de datos de sismos (SQLite)", "ok", f"{n} sismos guardados")
    except Exception as ex:
        add(L8, "Base de datos de sismos (SQLite)", "falla", f"No se puede leer: {ex}")
    it = h.get("guardado")
    if it.get("t_fail") and (not it.get("t_ok") or it["t_fail"] > it["t_ok"]):
```

```
        add(L8, "Guardado de sismos", "falla", f"Error {_ago(it['t_fail'], now)}: {it.get('msg', '')}")
    else:
        add(L8, "Guardado de sismos", "ok", f"Último guardado {_ago(it['t_ok'], now)}" if it.get("t_ok") else "Sin cambios pendientes")
    it = h.get("tickets")
    add(L8, "Tickets técnicos", "falla" if it.get("t_fail") and (not it.get("t_ok") or it["t_fail"] > it["t_ok"]) else "ok",
        it.get("msg") if it.get("t_fail") and (not it.get("t_ok") or it["t_fail"] > it["t_ok"]) else
        (f"Último guardado {_ago(it['t_ok'], now)}" if it.get("t_ok") else "Sin tickets todavía"))
    try:
        du = shutil.disk_usage(ctx["data_dir"])
        free = du.free / 1e9
        add(L8, "Espacio en disco", "ok" if free > 2 else "falla",
            f"{free:.1f} GB libres" + ("" if free > 2 else " · libera espacio o reduce archivo.dias"))
    except Exception as ex:
        add(L8, "Espacio en disco", "falla", str(ex))

    # ---------------- Capa 9: presentación y alertas
    L9 = "9 · Panel y alertas"
    add(L9, "Servidor web y API", "ok", f"Respondiendo en el puerto {cfg.get('panel', {}).get('puerto', 8767)}")
    for key, name, label in (("telegram", "Telegram", "Telegram"), ("correo", "Correo electrónico", "Correo")):
        c = cfg.get("alertas", {}).get(key, {})
        if not c.get("activo"):
            add(L9, name, "inactivo", f"Desactivado en config.json → alertas.{key}")
            continue
        incomplete = (key == "telegram" and not (c.get("token") and c.get("chat_id"))) or \
                     (key == "correo" and not (c.get("servidor") and c.get("para")))
        if incomplete:
            add(L9, name, "falla", "Activo pero incompleto en config.json (faltan datos)")
            continue
        it = h.get(key)
        if it.get("t_fail") and (not it.get("t_ok") or it["t_fail"] > it["t_ok"]):
            add(L9, name, "falla", f"Último envío falló {_ago(it['t_fail'], now)}: {it.get('msg', '')}")
        elif it.get("t_ok"):
            add(L9, name, "ok", f"Último envío correcto {_ago(it['t_ok'], now)}")
        else:
            add(L9, name, "ok", "Configurado; sin envíos todavía (usa «Probar alertas» para verificar)")
    al = ctx.get("alerts")
    if al is not None:
        n = al.q.qsize()
        add(L9, "Cola de alertas", "ok" if n < 5 else "falla", f"{n} mensajes en espera" + ("" if n < 5 else " · los envíos se están atorando"))
        sn = getattr(al, "sonidos", None)
        if sn is not None:
            from .sonido import BASE as SBASE, TIPOS as STIPOS
            falta = [f for f in STIPOS.values() if not os.path.exists(os.path.join(SBASE, f))]
            if not cfg.get("alertas", {}).get("sonido", True):
                add(L9, "Sonido en la PC", "inactivo", "Desactivado en config.json → alertas.sonido")
            elif not sn.ok:
                add(L9, "Sonido en la PC", "inactivo", "No disponible en este sistema (solo Windows)")
            elif falta:
                add(L9, "Sonido en la PC", "falla", f"Faltan archivos en sonidos\\: {', '.join(falta)} (se usarán pitidos)")
            elif sn.error:
                add(L9, "Sonido en la PC", "falla", f"No se pudo reproducir: {sn.error}")
            else:
                add(L9, "Sonido en la PC", "ok", (f"Último sonido: {sn.last[1]} {_ago(sn.last[0], now)}" if sn.last
                                                  else "4 sonidos listos (usa «Probar sonido» para escucharlos)"))

    # ---------------- Capa 10: operación
    L10 = "10 · Operación"
    add(L10, "Tiempo en funcionamiento", "ok", f"{up / 3600:.1f} h desde el arranque")
    rss = _rss_mb()
    if rss is not None:
        add(L10, "Memoria del proceso", "ok" if rss < 2500 else "falla",
            f"{rss:.0f} MB" + ("" if rss < 2500 else " · uso alto: reduce estaciones.max_estaciones o memoria.*"))
    lp = ctx.get("log_path")
    try:
        ok = bool(lp) and os.path.exists(lp) and os.access(lp, os.W_OK)
        add(L10, "Registro de actividad (monitor.log)", "ok" if ok else "falla",
            os.path.relpath(lp, os.path.dirname(ctx["data_dir"])) if ok else "No se puede escribir el registro")
    except Exception as ex:
        add(L10, "Registro de actividad (monitor.log)", "falla", str(ex))
    add(L10, "Una sola copia del monitor", "ok", f"Candado activo en el puerto {cfg.get('panel', {}).get('puerto', 8767) + 1}")

    bad = sum(1 for L in layers for i in L["items"] if i["estado"] == "falla")
    good = sum(1 for L in layers for i in L["items"] if i["estado"] == "ok")
    return {"now": now, "ok": good, "fallas": bad, "inactivos": sum(len(L["items"]) for L in layers) - bad - good,
            "general": "ok" if bad == 0 else "falla", "capas": layers}
```

sismo/locator.py — 226 líneas

```
"""Localización hipocentral GLOBAL sobre la esfera (0–700 km).

1) Búsqueda robusta (norma L1) en malla: si no hay posición inicial, malla global de Fibonacci (~2°) y después mallas locales cada vez más finas en latitud, longitud y profundidad.
2) Malla probabilística (L2) alrededor de la mejor solución: hipocentro de máxima verosimilitud, elipse de error al 68 % y error de profundidad.
El error de modelo crece con el tiempo de viaje (heterogeneidad 3D de la Tierra no incluida en el
modelo 1D): ~0.6 s a distancias regionales, ~2.5 s para P telesísmica.
"""
import numpy as np

R_E = 6371.0
KM_DEG = 111.195
RAD = np.pi / 180


def dist_km(la1, lo1, la2, lo2):
    la1, lo1, la2, lo2 = (np.asarray(v, float) * RAD for v in (la1, lo1, la2, lo2))
    s = np.sin((la2 - la1) / 2) ** 2 + np.cos(la1) * np.cos(la2) * np.sin((lo2 - lo1) / 2) ** 2
    return 2 * R_E * np.arcsin(np.minimum(1, np.sqrt(s)))


def bearing(la1, lo1, la2, lo2):
    la1, lo1, la2, lo2 = (np.asarray(v, float) * RAD for v in (la1, lo1, la2, lo2))
    y = np.sin(lo2 - lo1) * np.cos(la2)
    x = np.cos(la1) * np.sin(la2) - np.sin(la1) * np.cos(la2) * np.cos(lo2 - lo1)
    return (np.degrees(np.arctan2(y, x)) + 360) % 360


def wrap_lon(lon):
    return (np.asarray(lon, float) + 180.0) % 360.0 - 180.0


def fibonacci_nodes(spacing_deg):
    """Puntos casi equidistantes sobre la esfera (espaciado aproximado en grados)."""
    n = int(round(4 * np.pi / (spacing_deg * RAD) ** 2))
    i = np.arange(n) + 0.5
    lat = np.degrees(np.arcsin(1 - 2 * i / n))
    lon = wrap_lon(np.degrees(np.pi * (1 + 5 ** 0.5) * i))
    return lat, lon


def model_sigma(tt, is_p):
    """Error del modelo de velocidades (s) según el tiempo de viaje."""
    tt = np.asarray(tt, float)
    if is_p is True:
        return 0.35 + 0.012 * np.minimum(tt, 100) + 0.0015 * np.maximum(0, tt - 100)
    if is_p is False:
        return 0.6 + 0.02 * np.minimum(tt, 150) + 0.003 * np.maximum(0, tt - 150)
    return np.where(is_p, 0.35 + 0.012 * np.minimum(tt, 100) + 0.0015 * np.maximum(0, tt - 100),
                    0.6 + 0.02 * np.minimum(tt, 150) + 0.003 * np.maximum(0, tt - 150))


ZMAX = 700.0


class Locator:
    def __init__(self, tt, corr_fn=None, depth_prior=None):
        self.tt = tt
        self.corr_fn = corr_fn or (lambda key, phase, d=None: 0.0)
        # v2.1.0: información previa de profundidad (sismicidad histórica del USGS alrededor del epicentro).
        # depth_prior(lat, lon) -> función pen(z) en unidades de chi² (0 en la profundidad más probable), o None
        self.depth_prior = depth_prior
        self.glat, self.glon = fibonacci_nodes(2.0)

    # ---- evaluación vectorizada ----
    def _eval(self, picks, lat, lon, z, robust, pen=None):
        """lat, lon: arreglos de nodos (M,). Devuelve origen y costo por nodo."""
        t = np.array([p["t"] for p in picks])
        sig_pick = np.array([p["sigma"] for p in picks])
        ph = np.array([p["phase"] for p in picks])
        isP = ph != "S"                              # P, pP y sP llevan error de modelo tipo P
        corr = np.array([p["_corr"] for p in picks])
        plat = np.array([p["lat"] for p in picks])
        plon = np.array([p["lon"] for p in picks])
        d = dist_km(lat[:, None], lon[:, None], plat[None, :], plon[None, :])  # (M, n)
        tt = self.tt.tt("P", d, z)
        for other in set(ph) - {"P"}:
            tt = np.where((ph == other)[None, :], self.tt.tt(other, d, z), tt)
        tt = tt + corr[None, :]
        o = t[None, :] - tt
        w = 1.0 / (sig_pick[None, :] ** 2 + model_sigma(tt, isP[None, :]) ** 2)
        extra = float(pen(z)) if pen is not None else 0.0
        if robust:
            org = np.median(o, axis=1)
            cost = np.minimum(np.abs(o - org[:, None]) * np.sqrt(w), 4).sum(axis=1) + 0.5 * extra
            return org, cost
        org = (w * o).sum(axis=1) / w.sum(axis=1)
```

```
        chi2 = (w * (o - org[:, None]) ** 2).sum(axis=1) + extra
        return org, chi2

    @staticmethod
    def _box(lat0, lon0, span, step):
        """Malla lat/lon alrededor de un punto; el paso en longitud se ajusta por cos(lat)."""
        lats = np.arange(lat0 - span, lat0 + span + 1e-9, step)
        lats = lats[np.abs(lats) <= 89.9]
        c = max(0.15, np.cos(lat0 * RAD))
        lons = np.arange(-span / c, span / c + 1e-9, step / c)
        LA, LO = np.meshgrid(lats, lons, indexing="ij")
        return LA.ravel(), wrap_lon(lon0 + LO.ravel())

    def _grid(self, picks, la, lo, zs, robust, pen=None):
        best = None
        allres = []
        for z in zs:
            org, cost = self._eval(picks, la, lo, z, robust, pen)
            allres.append((z, org, cost))
            j = int(np.argmin(cost))
            if best is None or cost[j] < best["cost"]:
                best = {"lat": float(la[j]), "lon": float(lo[j]), "z": float(z), "org": float(o
rg[j]), "cost": float(cost[j])}
        return best, allres

    def _robust(self, picks, guess=None, pen=None):
        if guess is None:
            best, _ = self._grid(picks, self.glat, self.glon, [15, 60, 150, 300, 500, 650], Tru
e)
            span0 = 2.5
        else:
            best = {"lat": guess[0], "lon": guess[1], "z": guess[2], "cost": np.inf}
            span0 = 3.0
        la, lo = self._box(best["lat"], best["lon"], span0, 0.25)
        b, _ = self._grid(picks, la, lo, [5, 15, 33, 60, 100, 150, 220, 300, 400, 500, 600, 690
], True, pen)
        if b["cost"] <= best["cost"]:
            best = b
        for span, step, dz in ((0.6, 0.06, 15), (0.12, 0.02, 5)):
            c = best
            zs = np.arange(max(0, c["z"] - 4 * dz), min(ZMAX, c["z"] + 4 * dz) + 1e-9, dz)
            la, lo = self._box(c["lat"], c["lon"], span, step)
            b2, _ = self._grid(picks, la, lo, zs, True, pen)
            if b2["cost"] <= best["cost"]:
                best = b2
        return best

    def _pdf(self, picks, c, span, pen=None):
        step = span / 20
        zs = np.arange(max(0, c["z"] - 150), min(ZMAX, c["z"] + 150) + 1e-9, 10 if span > 1 els
e 5)
        la, lo = self._box(c["lat"], c["lon"], span, step)
        _, allres = self._grid(picks, la, lo, zs, False, pen)
        Z = np.concatenate([np.full(len(la), z) for z, _, _ in allres])
        ORG = np.concatenate([o for _, o, _ in allres])
        CHI = np.concatenate([ch for _, _, ch in allres])
        LA, LO = np.tile(la, len(allres)), np.tile(lo, len(allres))
        j = int(np.argmin(CHI))
        chimin = CHI[j]
        dof = max(1, len(picks) - 4)
        f2 = max(1.0, chimin / dof)
        w = np.exp(-(CHI - chimin) / (2 * f2))
        W = w.sum()
        cosl = np.cos(LA[j] * RAD)
        x = wrap_lon(LO - LO[j]) * KM_DEG * cosl
        y = (LA - LA[j]) * KM_DEG
        mx, my, mz = (w * x).sum() / W, (w * y).sum() / W, (w * Z).sum() / W
        sxx = (w * (x - mx) ** 2).sum() / W
        syy = (w * (y - my) ** 2).sum() / W
        sxy = (w * (x - mx) * (y - my)).sum() / W
        szz = (w * (Z - mz) ** 2).sum() / W
        tr, dd = (sxx + syy) / 2, np.sqrt(((sxx - syy) / 2) ** 2 + sxy ** 2)
        k68 = 1.52
        cl = max(0.15, np.cos(c["lat"] * RAD))
        edge = (np.abs(LA - c["lat"]) > span - step / 2) | (np.abs(wrap_lon(LO - c["lon"])) * c
l > span - step / 2)
        theta = 0.5 * np.arctan2(2 * sxy, sxx - syy)
        return {"lat": float(LA[j]), "lon": float(LO[j]), "z": float(Z[j]), "org": float(ORG[j]
),
                "errH": float(k68 * np.sqrt(tr + dd)), "errMin": float(k68 * np.sqrt(max(0, tr
- dd))),
                "errAz": float(((90 - np.degrees(theta)) % 180 + 180) % 180), "errZ": float(np.
sqrt(szz)),
                "open": bool(w[edge].sum() / W > 0.1)}

    def locate(self, picks_in, guess=None, full=True, zfix=None):
        """picks_in: lista de dict(key, lat, lon, t, phase, sigma). guess: (lat, lon, z) opcion
al.
        full=False: solo la búsqueda robusta (rápida, para depurar grupos); sin elipse de error
."""
        # corrección por estación según la distancia aproximada (regional o telesísmica)
        if guess is not None:
            ref = guess
        else:
            ref = None
```

```
        picks = []
        for p in picks_in:
            d = float(dist_km(ref[0], ref[1], p["lat"], p["lon"])) if ref else None
            picks.append(dict(p, _corr=self.corr_fn(p["key"], p["phase"], d)))
        pen = None
        if zfix is not None:
            # v2.1.0: profundidad medida con fases pP/sP (h, σ): restricción gaussiana; sustitu
    ye a la previa regional
            h0, s0 = float(zfix[0]), max(5.0, float(zfix[1]))
            pen = lambda z, h0=h0, s0=s0: ((z - h0) / s0) ** 2
        elif self.depth_prior is not None and guess is not None:
            try:
                pen = self.depth_prior(guess[0], guess[1])
            except Exception:
                pen = None
        r = self._robust(picks, guess, pen)
        if guess is None:      # con la primera posición se recalculan las correcciones por dis
    tancia
            for p in picks:
                p["_corr"] = self.corr_fn(p["key"], p["phase"], float(dist_km(r["lat"], r["lon"
    ], p["lat"], p["lon"])))
            if self.depth_prior is not None and zfix is None:
                try:
                    pen = self.depth_prior(r["lat"], r["lon"])
                except Exception:
                    pen = None
                if pen is not None:     # se refina la primera solución con la profundidad prev
    ia
                    r = self._robust(picks, (r["lat"], r["lon"], r["z"]), pen)
        if full:
            span = 0.8
            g = self._pdf(picks, r, span, pen)
            while g["open"] and span < 4:
                span *= 2.2
                g = self._pdf(picks, g, span, pen)
        else:
            g = {"lat": r["lat"], "lon": r["lon"], "z": r["z"], "org": r["org"], "errH": 999.0,
     "errMin": 999.0,
                 "errAz": 0.0, "errZ": 999.0, "open": False}
        lat, lon, z, org = g["lat"], g["lon"], g["z"], g["org"]
        plat = np.array([p["lat"] for p in picks])
        plon = np.array([p["lon"] for p in picks])
        d = dist_km(lat, lon, plat, plon)
        tts = np.array([float(self.tt.tt(p["phase"], d[i], z)) for i, p in enumerate(picks)])
        res = np.array([p["t"] - (org + tts[i] + p["_corr"]) for i, p in enumerate(picks)])
        sig = np.array([np.hypot(p["sigma"], float(model_sigma(tts[i], p["phase"] != "S"))) for
     i, p in enumerate(picks)])
        keys = {}
        for p in picks:
            keys.setdefault(p["key"], (p["lat"], p["lon"]))
        az = np.sort([float(bearing(lat, lon, la, lo)) for la, lo in keys.values()])
        gap = 360.0
        if len(az) > 1:
            gaps = np.diff(np.concatenate([az, [az[0] + 360]]))
            gap = float(gaps.max())
        return {"lat": lat, "lon": lon, "z": z, "org": org, "res": res.tolist(), "sig": sig.tol
    ist(),
                "rms": float(np.sqrt(np.mean(res ** 2))), "gap": gap, "edge": False, "dmin": fl
    oat(d.min()),
                "errH": g["errH"], "errMin": g["errMin"], "errAz": g["errAz"], "errZ": g["errZ"
    ],
                "errOpen": bool(g["open"]), "prior": pen is not None}
```

sismo/magglobal.py — 249 líneas

```
"""Magnitudes globales (Fase 2): mb, mB_BB, Ms_BB y Mw estimada.

Fórmulas del estándar de magnitudes de la IASPEI (2013):
  mb    = log10(A/T) + Q(Δ,h) − 3.0
          A: amplitud de desplazamiento del suelo (nm) de la onda P en registro de periodo cort
o
          (simulación WWSSN-SP), T ≤ 3 s, 20° ≤ Δ ≤ 100°, ventana de la P de 30 s.
  mB_BB = log10(Vmax/2π) + Q(Δ,h) − 3.0
          Vmax: velocidad vertical máxima (nm/s) en banda ancha, en el tren de la P (hasta 60 s
 o la PP).
  Ms_BB = log10(Vmax/2π) + 1.66·log10(Δ) + 0.3
          Vmax: velocidad vertical máxima (nm/s) de la onda Rayleigh, periodos de 3 a 60 s,
          2° ≤ Δ ≤ 160°, profundidad < 60 km.
Q(Δ,h) es la función de calibración de Gutenberg y Richter (1956) para PZ. Aquí se usa una
aproximación suavizada de esa curva (Q_BASE, poco más de ±0.1 respecto de la tabla original) má
s
una corrección por profundidad; el monitor la recalibra por intervalos de distancia y profundid
ad
con los mb del catálogo del USGS (sismo/calibracion.py).

Mw estimada: combinación ponderada de las magnitudes convertidas a Mw con relaciones publicadas
(Scordilis 2006 para mb y Ms; Bormann y Saul 2008 para mB; Mwp ≈ Mw, Tsuboi et al. 1995), cada
 una
dentro de su intervalo de validez. Con suficientes sismos confirmados, las relaciones se sustit
uyen
por las ajustadas contra el Mww del USGS.
"""
import numpy as np
from scipy import signal

# ---- Función de calibración Q(Δ, h) — aproximación de Gutenberg y Richter (1956), PZ, foco su
perficial
Q_DIST = np.array([20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80, 85, 90, 95, 100, 105], f
loat)
Q_BASE = np.array([6.0, 6.1, 6.4, 6.5, 6.6, 6.6, 6.6, 6.7, 6.8, 6.8, 6.9, 6.8, 6.9, 7.0, 7.0, 7
.1, 7.4, 7.5])
MB_DMIN, MB_DMAX = 20.0, 100.0
# mb regional (v0.5.0): misma medición de la P de periodo corto, entre 5° y 20° (≈ 550–2200 km)
, donde no
# aplican ni la ML (< 600 km) ni el mb telesísmico (≥ 20°). Calibración provisional plana (Q ≈
5.95, el valor de
# Gutenberg-Richter en el extremo de 20°) que el monitor corrige por intervalos con los mb del
USGS.
MBR_DMIN, MBR_DMAX = 5.0, 20.0
QR_BASE = 5.95
MS_DMIN, MS_DMAX = 2.0, 160.0
MS_ZMAX = 60.0


def q_gr(delta_deg, depth_km, corr=0.0):
    """Q(Δ,h) aproximada; corr: corrección aprendida del USGS para el intervalo."""
    q = float(np.interp(delta_deg, Q_DIST, Q_BASE))
    return q - 0.0007 * min(max(depth_km, 0.0), 700.0) + corr


# ---- respuestas
def _wwssn_sp_sos(fs):
    """Forma de respuesta de periodo corto tipo WWSSN-SP (máx. ~1 Hz), sobre desplazamiento."""
    return signal.butter(4, [0.7, min(2.5, 0.45 * fs)], btype="bandpass", fs=fs, output="sos")


def _gain(sos, fs, f):
    w, h = signal.sosfreqz(sos, worN=[f], fs=fs)
    return float(abs(h[0]))


def _half_period(x, i, fs):
    """Periodo dominante alrededor de la muestra i: doble del intervalo entre cruces por cero v
ecinos."""
    s = np.sign(x)
    a = i
    while a > 0 and s[a - 1] == s[i]:
        a -= 1
    b = i
    while b < len(x) - 1 and s[b + 1] == s[i]:
        b += 1
    return max(2.0 * (b - a + 1) / fs, 2.0 / fs)


def mb_station(ring, scale, tp, delta_deg, depth_km, qcorr=0.0, regional=False, win_s=30.0):
    """mb de una estación (dict con valor, A nm, T s) o None si no aplica. regional=True: mb re
gional 5°–20°."""
    lo_d, hi_d = (MBR_DMIN, MBR_DMAX) if regional else (MB_DMIN, MB_DMAX)
    if not scale or not (lo_d <= delta_deg < hi_d if regional else lo_d <= delta_deg <= hi_d):
        return None
    w_s = max(6.0, min(30.0, float(win_s)))
    got = ring.get(tp - 60, tp + w_s + 1.0)
    if got is None:
        return None
    t0, data = got
    fs = ring.fs
    if fs < 8 or t0 > tp - 30:
        return None
    v = (data - data[: int(10 * fs)].mean()) / scale                    # m/s
    u = np.cumsum(v) / fs * 1e9                                         # nm
```

```
    sos = _wwssn_sp_sos(fs)
    y = signal.sosfiltfilt(sos, u)
    ip = int(round((tp - t0) * fs))
    i0, i1 = ip - int(1 * fs), min(len(y), ip + int(w_s * fs))
    noise = float(np.abs(y[max(0, ip - int(40 * fs)):ip - int(5 * fs)]).max()) if ip > 45 * fs else 0.0
    seg = y[i0:i1]
    if len(seg) < 10:
        return None
    k = int(np.argmax(np.abs(seg)))
    # amplitud: mitad del pico a pico con el semiciclo adyacente
    j = i0 + k
    T = _half_period(y, j, fs)
    w = int(T * fs / 2) + 1
    lo, hi = max(0, j - w), min(len(y), j + w + 1)
    A_rec = (y[lo:hi].max() - y[lo:hi].min()) / 2
    if A_rec <= 0 or A_rec < 2.0 * noise or T > 3.0 or T < 0.2:
        return None
    A = A_rec / max(_gain(sos, fs, 1.0 / T), 1e-3)                      # desplazamiento del suelo (nm)
    z_clamped = min(max(depth_km, 0.0), 700.0)
    q_depth = (0.0007 * min(z_clamped, 100.0) + 0.0022 * max(0.0, z_clamped - 100.0)) if regional else 0.0007 * z_clamped
    q = (QR_BASE - q_depth + qcorr) if regional else q_gr(delta_deg, depth_km, qcorr)
    val = np.log10(A / T) + q - 3.0
    return {"val": float(val), "A": float(A), "T": float(T)}


def mB_station(ring, scale, tp, ts, delta_deg, depth_km, qcorr=0.0):
    """mB_BB de una estación: velocidad máxima en banda ancha (0.033–3 Hz) en el tren de la P."""
    if not scale or not (MB_DMIN <= delta_deg <= MB_DMAX):
        return None
    t_end = min(tp + 60.0, ts - 5.0)
    got = ring.get(tp - 60, t_end + 1)
    if got is None or t_end - tp < 10:
        return None
    t0, data = got
    fs = ring.fs
    if t0 > tp - 30:
        return None
    sos = signal.butter(2, [0.033, min(3.0, 0.45 * fs)], btype="bandpass", fs=fs, output="sos")
    v = signal.sosfiltfilt(sos, (data - data.mean()) / scale) * 1e9      # nm/s
    ip = int(round((tp - t0) * fs))
    ie = int(round((t_end - t0) * fs))
    noise = float(np.abs(v[max(0, ip - int(40 * fs)):ip - int(3 * fs)]).max()) if ip > 45 * fs else 0.0
    Vmax = float(np.abs(v[ip - int(1 * fs):ie]).max())
    if Vmax <= 0 or Vmax < 2.0 * noise:
        return None
    return {"val": float(np.log10(Vmax / (2 * np.pi)) + q_gr(delta_deg, depth_km, qcorr) - 3.0), "V": Vmax}


def rayleigh_window(org, dist_km):
    """Ventana de la onda Rayleigh (velocidad de grupo de 4.2 a 2.8 km/s)."""
    return org + dist_km / 4.2 - 30, org + dist_km / 2.8 + 60


def ms_station(ring, scale, org, dist_km, delta_deg, depth_km):
    """Ms_BB de una estación (componente vertical de banda ancha)."""
    if not scale or depth_km >= MS_ZMAX or not (MS_DMIN <= delta_deg <= MS_DMAX):
        return None
    a, b = rayleigh_window(org, dist_km)
    got = ring.get(a - 300, b)
    if got is None:
        return None
    t0, data = got
    fs = ring.fs
    if t0 > a - 200:
        return None
    sos = signal.butter(2, [1 / 60.0, 1 / 3.0], btype="bandpass", fs=fs, output="sos")
    v = signal.sosfiltfilt(sos, (data - data.mean()) / scale) * 1e9       # nm/s
    ia = int(round((a - t0) * fs))
    noise = float(np.abs(v[int(50 * fs):max(int(51 * fs), ia - int(60 * fs))]).max())
    seg = v[ia:]
    if len(seg) < fs * 30:
        return None
    k = int(np.argmax(np.abs(seg)))
    Vmax = float(abs(seg[k]))
    T = _half_period(seg, k, fs)
    if Vmax < 3.0 * noise or not (3.0 <= T <= 60.0):
        return None
    return {"val": float(np.log10(Vmax / (2 * np.pi)) + 1.66 * np.log10(delta_deg) + 0.3), "V": Vmax, "T": T}


# ---- conversión a Mw y combinación
DEFAULT_CONV = {
    # tipo: (pendiente, ordenada, mín., máx. de validez, referencia)
    "mb": (0.85, 1.03, 3.5, 6.2, "Scordilis (2006)"),
    "mB": (1.22, -1.58, 5.0, 8.5, "Bormann y Saul (2008)"),
    "Ms_lo": (0.67, 2.07, 3.0, 6.1, "Scordilis (2006)"),
    "Ms_hi": (0.99, 0.08, 6.2, 8.2, "Scordilis (2006)"),
    "Mwp": (1.0, 0.0, 5.0, 7.8, "Tsuboi et al. (1995)"),
```

```
}


def to_mw(kind, m, calib=None, ref=None):
    """Convierte una magnitud a Mw. calib: {tipo: (a, b, n)} ajustado contra el USGS (opcional).
      ref: tamaño de referencia del sismo (mediana de todas las estimaciones) para elegir el peso; así una
      magnitud equivocada no se da a sí misma más peso. Devuelve (Mw, peso) o None."""
    if m is None:
        return None
    if calib and kind in calib and calib[kind][2] >= 15:
        a, b, _ = calib[kind]
        mw = a * m + b
    else:
        key = kind if kind != "Ms" else ("Ms_lo" if m < 6.15 else "Ms_hi")
        a, b, lo, hi, _ = DEFAULT_CONV[key]
        mw = a * m + b + ((calib or {}).get("_off") or {}).get(kind, 0.0)   # v2.0.0: ajuste temprano
    r = mw if ref is None else ref
    # pesos según el intervalo donde cada magnitud es confiable (saturación y ruido)
    if kind == "mb":
        w = 1.0 if m <= 5.8 else 0.4 if m <= 6.2 else 0.0
    elif kind == "mB":
        w = 0.5 if r < 5.5 else 1.0
    elif kind == "Ms":
        w = 0.5 if r < 5.5 else 1.2
    else:  # Mwp: con sismos moderados la doble integración acumula ruido; con grandes, la ventana se queda corta
        w = 0.3 if r < 5.8 else 0.6 if r < 6.3 else 1.0 if r <= 7.5 else 0.5
    return (float(mw), w) if w > 0 else None


MAX_SPREAD = 0.7


def combine_mw(ev, calib=None):
    """Mw estimada a partir de las magnitudes disponibles del evento. Devuelve (Mw, ±, componentes, descartadas).

      Robustez (v0.3.1): los pesos dependen del tamaño de referencia (mediana de todas las estimaciones), no
      de la propia magnitud; y una estimación que se aparta más de 0.7 de la mediana de las demás se descarta
      (con dos estimaciones en desacuerdo se descarta la Mwp, la más sensible al ruido)."""
    raw = {}
    # v2.0.0: la conversión a Mw usa el mb sin la corrección de sesgo frente al mb del USGS (mbRaw)
    mbkey = "mbRaw" if ev.get("mbRaw") is not None else "mb"
    for kind, key, nkey, nmin in (("Mwp", "mwp", "mwpN", 3), ("mB", "mB", "mBN", 3), ("mb", mbkey, "mbN", 3), ("Ms", "ms", "msN", 2)):
        if ev.get(key) is not None and (ev.get(nkey) or 0) >= nmin:
            # Protección contra dobletes sísmicos: si el sismo ocurrió en la coda de un evento mayor previo,
            # las ondas superficiales Rayleigh (Ms) están severamente contaminadas y no deben usarse
            if kind == "Ms" and ev.get("coda_contaminated"):
                continue
            r = to_mw(kind, ev[key], calib)
            if r:
                raw[kind] = (r[0], (ev.get(nkey) or 0))
    if not raw:
        return None
    dropped = {}
    if len(raw) >= 3:
        for k in list(raw):
            others = [v[0] for kk, v in raw.items() if kk != k and kk not in dropped]
            if len(others) >= 2 and abs(raw[k][0] - float(np.median(others))) > MAX_SPREAD:
                # Regla de robustez: si mb tiene un consenso masivo (>= 15 estaciones y supera a las demás combinadas),
                # no puede ser descartada por discrepancia con pocas estaciones de Ms o mB
                if k == "mb" and raw["mb"][1] >= 15 and raw["mb"][1] >= 2 * sum(v[1] for kk, v in raw.items() if kk != "mb"):
                    continue
                dropped[k] = round(raw[k][0], 2)
    elif len(raw) == 2 and "Mwp" in raw:
        other = [v[0] for k, v in raw.items() if k != "Mwp"][0]
        if abs(raw["Mwp"][0] - other) > MAX_SPREAD and other < 6.5:
            dropped["Mwp"] = round(raw["Mwp"][0], 2)
    use = {k: v for k, v in raw.items() if k not in dropped}
    ref = float(np.median([v[0] for v in use.values()]))
    parts = {}
    for kind, (mw, n) in use.items():
        key = {"Mwp": "mwp", "mB": "mB", "mb": mbkey, "Ms": "ms"}[kind]
        r = to_mw(kind, ev[key], calib, ref)
        if r:
            parts[kind] = (r[0], r[1] * min(1.0, n / 6.0))
    if not parts:
        return None
    W = sum(w for _, w in parts.values())
    mw = sum(m * w for m, w in parts.values()) / W
    sd = float(np.sqrt(sum(w * (m - mw) ** 2 for m, w in parts.values()) / W)) if len(parts) > 1 else None
    return float(mw), sd, {k: round(v[0], 2) for k, v in parts.items()}, dropped
```

sismo/magnitude.py — 112 líneas

```
"""Magnitudes (ML IASPEI, Mwp) e intensidad estimada (Mercalli Modificada)."""
import numpy as np
from scipy import signal

from .dsp import wood_anderson_ba


def ml_iaspei(a_nm, r_km):
    """ML IASPEI. a_nm: amplitud de desplazamiento Wood-Anderson (nm, ganancia unitaria); r_km: distancia hipocentral."""
    return np.log10(a_nm) + 1.11 * np.log10(r_km) + 0.00189 * r_km - 2.09


def wa_peak(ring, scale, t0, t1, is_accel=False):
    """Amplitud pico Wood-Anderson (m) en [t0, t1] a partir de velocidad (o aceleración si is_accel=True) en cuentas."""
    if not scale:
        return 0.0
    got = ring.get(t0 - 30, t1)
    if got is None:
        return 0.0
    ts, data = got
    fs = ring.fs
    v = signal.sosfilt(signal.butter(2, 0.3, "highpass", fs=fs, output="sos"), data) / scale
    if is_accel:
        # Integración causal de aceleración (m/s²) a velocidad (m/s)
        v = np.cumsum(v) / fs
        v = signal.sosfilt(signal.butter(2, 0.5, "highpass", fs=fs, output="sos"), v)
    b, a = wood_anderson_ba(fs)
    y = signal.lfilter(b, a, v)
    i0 = max(0, int(round((t0 - ts) * fs)))
    return float(np.abs(y[i0:]).max()) if i0 < len(y) else 0.0


# ---- Mwp (Tsuboi et al., 1995) ----
# Válida con la componente vertical de banda ancha a distancias de 5° a 95° (también se usa desde
# 3° con menor peso). M0 = max|∫∫v dt| · 4πρα³r / Fp ; Mw = (log10 M0 − 9.1) / 1.5
MWP_RHO, MWP_ALPHA, MWP_FP = 3400.0, 7900.0, 0.52
MWP_DMIN, MWP_DMAX = 330.0, 10560.0      # 3° a 95°
MWP_SNR = 4.0


def mwp_window(tp, ts, dist_km):
    """Fin de la ventana de integración: antes de la S (o de PP/PcP en distancias grandes), máx. 120 s."""
    return min(ts - 2, tp + 120.0)


def mwp_station(ring, scale, tp, ts, dist_km):
    """Mwp de una estación con la componente vertical en banda ancha. None si no aplica.

    Procesamiento (v0.3.1):
    - Línea base: recta ajustada a los 100 s previos a la P y restada a todo el registro (quita el
      desplazamiento y la deriva lenta; antes solo se restaba el promedio de 10 s, y su error, integrado dos
      veces durante 120 s, inflaba la Mwp de los sismos moderados en más de 2 unidades).
    - Filtro causal 300 s – 12.5 s (0.0033–0.08 Hz): conserva la meseta de ∫u (bajas frecuencias) y quita
      el microsismo oceánico (0.1–0.3 Hz), que en un sismo M5 es tan grande como la propia señal.
    - Control de calidad: el mismo cálculo sobre una ventana de ruido de igual duración antes de la P;
      la señal debe superarlo MWP_SNR veces."""
    if not scale or dist_km < MWP_DMIN or dist_km > MWP_DMAX:
        return None
    t_end = mwp_window(tp, ts, dist_km)
    L = t_end - tp
    if L < 10:
        return None
    pre = max(L + 30, 100.0) + 120.0
    got = ring.get(tp - pre, t_end)
    if got is None:
        return None
    t0, data = got
    if t0 > tp - pre + 10:
        return None
    fs = ring.fs
    x = (np.asarray(data, float) - float(np.mean(data))) / scale
    tt = (np.arange(len(x)) / fs) + t0 - tp
    base = (tt >= -100) & (tt <= -2)
    if base.sum() < 20 * fs:
        return None
    c = np.polyfit(tt[base], x[base], 1)
    x = x - np.polyval(c, tt)
    sos = signal.butter(2, [1 / 300.0, 0.08], btype="bandpass", fs=fs, output="sos")
    x = signal.sosfilt(sos, x)              # causal: un filtro de fase cero adelanta energía a antes de la P e infla ∫u
    ip = int(round((tp - t0) * fs))
    n = int(round(L * fs))

    def peak(i0):
        disp = np.cumsum(x[i0:i0 + n]) / fs
        integ = np.cumsum(disp) / fs
```

```
        return float(np.abs(integ).max()) if len(integ) else 0.0

    sig = peak(ip)
    i_noise = ip - n - int(20 * fs)
    if i_noise < int(30 * fs):
        return None
    noise = peak(i_noise)
    if sig <= 0 or sig < MWP_SNR * noise:
        return None
    m0 = sig * 4 * np.pi * MWP_RHO * MWP_ALPHA ** 3 * (dist_km * 1000) / MWP_FP
    return (np.log10(m0) - 9.1) / 1.5


# ---- Intensidad (Allen, Wald y Worden, 2012; distancia a la ruptura aproximada) ----
SOIL = {"roca": ("Roca", 0.0), "firme": ("Terreno firme", 0.3), "transicion": ("Transición", 1.0), "lago": ("Zona de lago", 2.0)}
ROMAN = ["", "I", "II", "III", "IV", "V", "VI", "VII", "VIII", "IX", "X"]


def mmi(M, d_epi, z, soil="roca"):
    L = 10 ** (-2.44 + 0.59 * M)
    R = np.hypot(np.maximum(0, np.asarray(d_epi, float) - L / 2), max(z, 3))
    I = 3.950 + 0.913 * M - 1.107 * np.log(np.sqrt(R * R + (1 + 0.813 * np.exp(M - 5)) ** 2)) + SOIL.get(soil, ("", 0))[1]
    return np.clip(I, 1, 10)


def roman(I):
    return ROMAN[int(max(1, min(10, round(float(I)))))]
```

sismo/mseed.py — 119 líneas

```
"""Lector mínimo de miniSEED 2 (registros de 512 bytes o más; codificaciones Steim1, Steim2, INT16, INT32, FLOAT32).

Sirve para revisar y reprocesar el archivo continuo sin depender de ObsPy.
"""
import struct

import numpy as np


def _steim(frames, nframes, bo, steim2):
    out = []
    x0 = xn = None
    for f in range(nframes):
        fr = frames[f * 64:(f + 1) * 64]
        if len(fr) < 64:
            break
        w = struct.unpack(bo + "16I", fr)
        ctrl = w[0]
        for i in range(1, 16):
            nib = (ctrl >> (30 - 2 * i)) & 3
            word = w[i]
            if f == 0 and i == 1:
                x0 = struct.unpack(">i", struct.pack(">I", word))[0]
                continue
            if f == 0 and i == 2:
                xn = struct.unpack(">i", struct.pack(">I", word))[0]
                continue
            if nib == 0:
                continue
            if nib == 1:
                b = struct.pack(">I", word)
                out.extend(struct.unpack(">4b", b))
            elif not steim2:
                if nib == 2:
                    out.extend(struct.unpack(">2h", struct.pack(">I", word)))
                else:
                    out.append(struct.unpack(">i", struct.pack(">I", word))[0])
            else:
                dnib = word >> 30
                if nib == 2:
                    spec = {1: (1, 30), 2: (2, 15), 3: (3, 10)}.get(dnib)
                else:
                    spec = {0: (5, 6), 1: (6, 5), 2: (7, 4)}.get(dnib)
                if spec is None:
                    continue
                n, bits = spec
                mask = (1 << bits) - 1
                for k in range(n):
                    v = (word >> (bits * (n - 1 - k))) & mask
                    if v & (1 << (bits - 1)):
                        v -= 1 << bits
                    out.append(v)
    return x0, xn, out


def read_records(buf, t_a=None, t_b=None):
    """Genera (red, estación, ubicación, canal, fs, t0_epoch, datos) por registro.
    Si t_a y t_b se especifican, omite la descompresión Steim de registros fuera del rango."""
    import calendar
    pos = 0
    n = len(buf)
    while pos + 64 <= n:
        h = buf[pos:pos + 48]
        # orden de bytes: el año debe ser razonable
        bo = ">"
        year = struct.unpack(">H", h[20:22])[0]
        if not 1900 < year < 2100:
            bo = "<"
            year = struct.unpack("<H", h[20:22])[0]
        doy, hh, mm, ss, _, frac = struct.unpack(bo + "HBBBBH", h[22:30])
        nsamp, fac, mult = struct.unpack(bo + "Hhh", h[30:36])
        nblk = h[39]
        data_off, blk_off = struct.unpack(bo + "HH", h[44:48])
        sta, loc, cha, net = h[8:13].decode(errors="replace").strip(), h[13:15].decode(errors="replace").strip(), \
            h[15:18].decode(errors="replace").strip(), h[18:20].decode(errors="replace").strip()
        enc, reclen = 11, 9
        b = blk_off
        for _ in range(nblk):
            if b <= 0 or b + 4 > 512:
                break
            btype, bnext = struct.unpack(bo + "HH", buf[pos + b:pos + b + 4])
            if btype == 1000:
                enc = buf[pos + b + 4]
                reclen = buf[pos + b + 6]
            b = bnext
        rl = 1 << reclen
        if fac > 0 and mult > 0:
            fs = fac * mult
        elif fac > 0 and mult < 0:
            fs = -fac / mult
        elif fac < 0 and mult > 0:
            fs = -mult / fac
```

```
        else:
            fs = 1.0 / (fac * mult) if fac and mult else 0.0
        t0 = calendar.timegm((year, 1, 1, hh, mm, ss)) + (doy - 1) * 86400 + frac / 1e4
        dur = (nsamp / fs) if fs > 0 else 0.0
        if t_a is not None and t0 + dur < t_a:
            pos += rl
            continue
        if t_b is not None and t0 > t_b:
            pos += rl
            continue
        body = buf[pos + data_off:pos + rl]
        data = None
        if enc in (10, 11):
            x0, xn, d = _steim(body, len(body) // 64, bo, enc == 11)
            if x0 is not None and d:
                arr = np.cumsum(np.array(d[:nsamp], dtype=np.int64))
                arr = arr - arr[0] + x0
                data = arr
        elif enc == 3:
            data = np.frombuffer(body[:4 * nsamp], dtype=bo + "i4").astype(np.int64)
        elif enc == 1:
            data = np.frombuffer(body[:2 * nsamp], dtype=bo + "i2").astype(np.int64)
        elif enc == 4:
            data = np.frombuffer(body[:4 * nsamp], dtype=bo + "f4").astype(np.float64)
        if data is not None and fs > 0:
            yield net, sta, loc, cha, float(fs), float(t0), data
        pos += rl
```

sismo/netio.py — 301 líneas

```
"""Acceso a servicios web: metadatos de estaciones globales (FDSN) y catálogo mundial del USGS.
"""
import json
import os
import time
import urllib.parse
import urllib.request

import numpy as np

from .locator import dist_km

UA = {"User-Agent": "MonitorSismicoMundial/0.1"}


FEDCATALOG = "https://service.iris.edu/irisws/fedcatalog/1/query"


def _ssl_ctx():
    """v2.0.0: certificados de certifi (los instala ObsPy) para que funcione aunque Windows no tenga al día los
    certificados raíz (en una PC recién instalada el servicio de EarthScope puede fallar por eso)."""
    import ssl
    try:
        import certifi
        return ssl.create_default_context(cafile=certifi.where())
    except Exception:
        return None


def http_get(url, timeout=30):
    req = urllib.request.Request(url, headers=UA)
    try:
        with urllib.request.urlopen(req, timeout=timeout) as r:
            return r.read()
    except Exception as e:
        ctx = _ssl_ctx()
        if ctx is None or "CERTIFICATE" not in str(e).upper() and "SSL" not in str(e).upper():
            raise
        with urllib.request.urlopen(req, timeout=timeout, context=ctx) as r:
            return r.read()


def _bands(src):
    """Bandas de canal de una fuente, en orden de preferencia: "BH" o "BH,HH" (v0.4.1)."""
    return [b.strip()[:2].upper() for b in str(src.get("canales", "BH")).split(",") if b.strip()] or ["BH"]


def _parse_fdsn_text(txt, bands):
    """Estaciones con su mejor ubicación y banda (00 preferido, banda en el orden dado) y ganancias por canal."""
    if isinstance(bands, str):
        bands = [bands]
    st = {}
    for line in txt.splitlines():
        if not line or line.startswith("#"):
            continue
        f = line.split("|")
        if len(f) < 15:
            continue
        net, sta, loc, cha = f[0], f[1], f[2].strip(), f[3]
        if cha[:2] not in bands or cha[-1:] not in ("Z", "N", "E", "1", "2", "X", "Y"):
            continue
        end_t = f[16].strip() if len(f) > 16 else ""
        if end_t and end_t[:4] < "2025":
            continue
        try:
            lat, lon = float(f[4]), float(f[5])
        except ValueError:
            continue
        units = f[13].upper()
        try:
            scale = float(f[11]) if f[11] and units.startswith("M/S") and "**2" not in units else 0.0
        except ValueError:
            scale = 0.0
        key = f"{net}_{sta}"
        s = st.setdefault(key, {"key": key, "net": net, "sta": sta, "lat": lat, "lon": lon, "locs": {}, "open_locs": set()})
        s["lat"], s["lon"] = lat, lon
        is_open = not end_t or end_t[:4] >= "2026"
        if is_open:
            s["open_locs"].add((loc, cha[:2]))
        d_loc = s["locs"].setdefault((loc, cha[:2]), {})
        if scale or f"{loc}.{cha}" not in d_loc:
            if is_open or f"{loc}.{cha}" not in d_loc:
                d_loc[f"{loc}.{cha}"] = scale
    out = []
    for s in st.values():
        if not s["open_locs"]:
            continue
        def rank(lb):
            ch = s["locs"][lb]
```

```
            has_z = any(k.endswith("Z") and v for k, v in ch.items())
            three = len({k[-1] for k, v in ch.items() if v}) >= 3
            open_rk = 0 if lb in s["open_locs"] else 1
            return (open_rk, 0 if has_z else 1, 0 if three else 1, bands.index(lb[1]), {"00": 0, "": 1, "10": 2}.get(lb[0], 3), lb[0])
        ordered = [lb for lb in sorted(s["locs"], key=rank) if lb in s["open_locs"]] or sorted(s["locs"], key=rank)
        loc, band = ordered[0]
        chans = {}
        for lb in reversed(ordered):
            chans.update(s["locs"][lb])
        sels = [f"{l}{b}?" if l else f"{b}?" for l, b in ordered[:2]]
        out.append({"key": s["key"], "net": s["net"], "sta": s["sta"], "lat": s["lat"], "lon": s["lon"],
                    "loc": loc, "band": band, "sels": sels, "chans": chans})
    return out


def _query_source(src, log):
    bands = _bands(src)
    q = urllib.parse.urlencode({"net": ",".join(src["redes"]), "channel": ",".join(f"{b}?" for b in bands),
                                "level": "channel", "format": "text", "endafter": "2025-09-01T00:00:00"})
    # v2.0.0: si los servidores propios de la fuente fallan, se intenta el fedcatalog de IRIS (metadatos de todos
    # los centros de datos); el primero se repite una vez. Una respuesta vacía cuenta como falla
    hosts = list(dict.fromkeys(h for h in (src.get("fdsn"), src.get("fdsn_respaldo"), FEDCATALOG) if h))
    for host in hosts + hosts[:1]:
        if not host:
            continue
        try:
            # v0.5.7: «fedcatalog» de IRIS = metadatos de todos los centros de datos del mundo (EIDA, AusPass, SED…)
            url = f"{host.rstrip('/')}?{q}" if "fedcatalog" in host else f"{host.rstrip('/')}/fdsnws/station/1/query?{q}"
            txt = http_get(url, 90).decode("utf-8", "replace")
            got = _parse_fdsn_text(txt, bands)
            if got:
                return got
            log(f"Metadatos {src['nombre']}: {host} respondió sin estaciones")
        except Exception as e:
            log(f"Metadatos {src['nombre']}: {host} no respondió ({e})")
    return None


def select_spread(stations, max_n, min_sep_km, order):
    """Selección con buena cobertura mundial (en tres pasadas):
    1) cobertura base: una estación cada ~400 km en todo el mundo (de la red preferida que haya en cada zona);
    2) cobertura regional equilibrada: una estación cada ~220 km en todo el mundo antes de saturar zonas densas;
    3) relleno fino: hasta max_n estaciones con la separación mínima normal, en el orden de preferencia."""
    pref = {n: i for i, n in enumerate(order)}
    stations = sorted(stations, key=lambda s: (pref.get(s["net"], len(pref)), s["key"]))
    chosen, la, lo, used = [], [], [], set()
    for sep in (max(3 * min_sep_km, 400.0), max(2 * min_sep_km, 220.0), min_sep_km):
        for s in stations:
            if len(chosen) >= max_n:
                break
            if s["key"] in used:
                continue
            if la and float(np.min(dist_km(np.array(la), np.array(lo), s["lat"], s["lon"]))) < sep:
                continue
            chosen.append(s)
            used.add(s["key"])
            la.append(s["lat"])
            lo.append(s["lon"])
    return chosen


def _offered(cfg, log, cache_path=None):
    """v0.5.6: lista de estaciones que ofrece cada servidor SeedLink (INFO STATIONS), consultada en paralelo.
    Sirve para pedir cada estación al servidor que de verdad la transmite (p. ej. NU a INETER y no a EarthScope)."""
    from concurrent.futures import ThreadPoolExecutor
    from .slprobe import probe
    srcs = [x for x in cfg["fuentes"] if x.get("activa")]

    def one(src):
        if src["nombre"] == "INETER" and saved.get("INETER"):
            return "INETER", saved["INETER"]
        host, _, port = src["seedlink"].partition(":")
        try:
            r = probe(host, int(port or 18000), timeout=10)
        except Exception:
            return src["nombre"], None
        if not r.get("completa"):      # v2.0.0: una lista cortada no sirve para filtrar (se piden todas)
            if r.get("claves"):
```

```
                log(f"SeedLink {src['nombre']}: la lista de estaciones llegó incompleta ({len(r
['claves'])}); "
                    "se piden todas las de sus metadatos")
            return src["nombre"], None
        return src["nombre"], set(r.get("claves") or []) or None

    # v2.0.0: las listas completas se guardan (datos/seedlink_ofrecidas.json) y se usan cuando
un servidor no
    # entrega la suya o llega cortada, en lugar de pedir todas las estaciones de los metadatos
    saved = {}
    if cache_path:
        try:
            with open(cache_path, encoding="utf-8") as f:
                saved = {k: set(v) for k, v in json.load(f).items()}
        except Exception:
            saved = {}
    out = {}
    with ThreadPoolExecutor(max_workers=max(1, len(srcs))) as ex:
        for name, keys in ex.map(one, srcs):
            if keys is None and saved.get(name):
                keys = saved[name]
                log(f"SeedLink {name}: sin lista completa; se usa la guardada ({len(keys)} esta
ciones)")
            elif keys is None:
                log(f"SeedLink {name}: no entregó su lista de estaciones; se piden todas las de
 sus metadatos")
            out[name] = keys
    if cache_path:
        try:
            keep = dict(saved)
            keep.update({k: v for k, v in out.items() if v})
            with open(cache_path, "w", encoding="utf-8") as f:
                json.dump({k: sorted(v) for k, v in keep.items()}, f)
        except OSError:
            pass
    return out


def load_stations(cfg, cache_path, log=print):
    """Lista de estaciones por fuente: {nombre_fuente: [estaciones]}. Usa la caché si no hay re
d."""
    ecfg = cfg.get("estaciones", {})
    excl = set(ecfg.get("excluir", []))
    allst, failed = [], False
    load_stations.failed = []
    offered = (_offered(cfg, log, os.path.join(os.path.dirname(cache_path), "seedlink_ofrecidas
.json"))
               if ecfg.get("solo_ofrecidas", True) else {})
    # v2.0.0: lista guardada (o la que trae el instalador) para reponer una fuente que no respo
nda
    cached, seen_c = [], set()
    for path in (cache_path, os.path.join(os.path.dirname(cache_path), "estaciones_instalador.j
son")):
        try:
            with open(path, encoding="utf-8") as f:
                for s in json.load(f):
                    if s.get("key") and s["key"] not in seen_c:
                        seen_c.add(s["key"])
                        cached.append(s)
        except Exception:
            pass
    for src in cfg["fuentes"]:
        if not src.get("activa"):
            continue
        got = _query_source(src, log)
        if got is None:
            failed = True
            load_stations.failed.append(src["nombre"])
            prev = [s for s in cached if s.get("fuente") == src["nombre"] and s["key"] not in e
xcl]
            if prev:
                allst += prev
                log(f"Metadatos {src['nombre']}: no respondió; se usan sus {len(prev)} estacion
es de la lista guardada")
            else:
                log(f"Metadatos {src['nombre']}: no respondió y no hay lista guardada; sus esta
ciones no se cargan")
            continue
        for s in got:
            s["fuente"] = src["nombre"]
        off = offered.get(src["nombre"])
        n0 = len(got)
        if off:     # v0.5.6: solo las estaciones que el servidor SeedLink realmente transmite
            got = [s for s in got if s["key"] in off]
        allst += [s for s in got if s["key"] not in excl]
        log(f"Metadatos {src['nombre']}: {n0} estaciones con canales {src.get('canales', 'BH')}
"
            + (f"; {len(got)} las transmite {src['seedlink']}" if off else ""))
    if not allst:
        if cached:
            log("Metadatos: usando la lista guardada de estaciones.")
            load_stations.from_cache = True
            return cached
        raise RuntimeError("No se pudieron obtener los metadatos de las estaciones")
    # una estación puede aparecer en dos fuentes: se queda la de la primera fuente activa
```

```
    seen, uniq = set(), []
    for s in allst:
        if s["key"] not in seen:
            seen.add(s["key"])
            uniq.append(s)
    load_stations.from_cache = False
    # Selección de estaciones: si separacion_minima_km <= 0 se integran todas sin podar densida
d
    full = set(ecfg.get("redes_completas", []))
    min_sep = float(ecfg.get("separacion_minima_km", 0))
    max_sta = int(ecfg.get("max_estaciones", 5000))
    if min_sep <= 0:
        out = list(uniq)[:max_sta]
        log(f"Estaciones seleccionadas: {len(out)} de {len(uniq)} (sin filtro de separación mín
ima; densidad global máxima)")
    else:
        rest = [s for s in uniq if s["net"] not in full]
        out = select_spread(rest, max_sta, min_sep, ecfg.get("orden_de_preferencia", []))
        allnet = [s for s in uniq if s["net"] in full]
        out += allnet
        log(f"Estaciones seleccionadas: {len(out)} de {len(uniq)} (separación mínima {min_sep:.
0f} km"
            + (f"; redes completas {', '.join(sorted(full))}: {len(allnet)}" if full else "") +
 ")")
    if failed and os.path.exists(cache_path):
        log("Aviso: alguna fuente no respondió; se usan las estaciones de las demás.")
    if not (failed and len(out) < len(cached)):      # v2.0.0: no se reemplaza la lista guardad
a por una incompleta
        try:
            with open(cache_path, "w", encoding="utf-8") as f:
                json.dump(out, f)
        except OSError:
            pass
    return out


def usgs_query(days, minmag, box=None):
    """Catálogo USGS: búsqueda FDSN y, si falla, los resúmenes precalculados (feeds)."""
    start = time.time() - days * 86400
    q = {"format": "geojson", "starttime": time.strftime("%Y-%m-%dT%H:%M:%S", time.gmtime(start
)),
         "minmagnitude": minmag, "orderby": "time"}
    if box:
        q.update(minlatitude=box["lat0"], maxlatitude=box["lat1"], minlongitude=box["lon0"], ma
xlongitude=box["lon1"])
    try:
        feats = json.loads(http_get("https://earthquake.usgs.gov/fdsnws/event/1/query?" + urlli
b.parse.urlencode(q), 30))["features"]
    except Exception:
        lvl = "4.5" if minmag >= 4.5 else "2.5"
        per = "day" if days <= 1 else "week" if days <= 7 else "month"
        feats = json.loads(http_get(f"https://earthquake.usgs.gov/earthquakes/feed/v1.0/summary
/{lvl}_{per}.geojson", 30))["features"]
        feats = [f for f in feats if f["properties"]["time"] / 1000 >= start and (f["properties
"]["mag"] or 0) >= minmag]
    out = []
    for f in feats:
        c, p = f["geometry"]["coordinates"], f["properties"]
        out.append({"id": f["id"], "t": p["time"] / 1000, "lat": c[1], "lon": c[0], "z": c[2] i
f c[2] is not None else 0.0,
                    "mag": p.get("mag"), "magType": p.get("magType"), "place": p.get("place"),
"url": p.get("url"),
                    "updated": (p.get("updated") or 0) / 1000})
    return out
```

sismo/placas.py — 99 líneas

```
"""Límites de placas tectónicas (modelo PB2002 de Bird, 2003) para el mapa del panel (v0.4.6).

Se descarga una sola vez (GeoJSON público del modelo PB2002) y se guarda en datos/placas_pb2002.json ya
procesado. Cada límite se clasifica por las placas que separa:
  1 = primario   (entre placas mayores: Pacífica, Norteamericana, Sudamericana, Euroasiática, Africana,
                  Antártica, Australiana, India)
  2 = secundario (involucra una placa menor: Nazca, Cocos, Caribe, Filipinas, Arábiga, Juan de Fuca,
                  Escocia, Somalí, Sonda, Amur, Ojotsk, Yangtsé)
  3 = microplacas (las demás placas del modelo PB2002)
"""
import json
import os
import re
import threading

from .netio import http_get

URLS = [
    "https://raw.githubusercontent.com/fraxen/tectonicplates/master/GeoJSON/PB2002_boundaries.json",
    "https://cdn.jsdelivr.net/gh/fraxen/tectonicplates@master/GeoJSON/PB2002_boundaries.json",
]
MAYORES = {"PA", "NA", "SA", "EU", "AF", "AN", "AU", "IN"}
MENORES = {"NZ", "CO", "CA", "PS", "AR", "JF", "SC", "SO", "SU", "AM", "OK", "YA"}
NOMBRES = {
    "PA": "Pacífica", "NA": "Norteamericana", "SA": "Sudamericana", "EU": "Euroasiática", "AF": "Africana",
    "AN": "Antártica", "AU": "Australiana", "IN": "India", "NZ": "Nazca", "CO": "Cocos", "CA": "Caribe",
    "PS": "Mar de Filipinas", "AR": "Arábiga", "JF": "Juan de Fuca", "SC": "Escocia", "SO": "Somalí",
    "SU": "Sonda", "AM": "Amur", "OK": "Ojotsk", "YA": "Yangtsé", "RI": "Rivera", "SW": "Sándwich",
    "BH": "Cabeza de Pájaro", "MS": "Molucas", "TI": "Timor", "BS": "Mar de Banda", "MA": "Marianas",
    "PM": "Panamá", "ND": "Norandina", "AP": "Altiplano", "NH": "Nuevas Hébridas", "KE": "Kermadec",
    "TO": "Tonga", "FT": "Futuna", "NI": "Niuafo'ou", "CR": "Arrecife Conway", "BR": "Balmoral",
    "NB": "Bismarck Norte", "SB": "Bismarck Sur", "MN": "Manus", "SS": "Salomón", "WL": "Woodlark",
    "CL": "Carolinas", "ON": "Okinawa", "AT": "Anatolia", "AS": "Egeo", "EA": "Isla de Pascua",
    "JZ": "Juan Fernández", "GP": "Galápagos", "MO": "Maoke", "SH": "Shetland", "BU": "Birmania",
}
_lock = threading.Lock()


def _tier(a, b):
    s = {a, b}
    if s <= MAYORES:
        return 1
    if s <= (MAYORES | MENORES):
        return 2
    return 3


def _split_dateline(coords):
    """Corta las líneas que cruzan el antimeridiano (evita trazos de lado a lado del mapa)."""
    parts, cur = [], []
    for lon, lat in coords:
        if cur and abs(lon - cur[-1][1]) > 180:
            if len(cur) > 1:
                parts.append(cur)
            cur = []
        cur.append([round(lat, 3), round(lon, 3)])
    if len(cur) > 1:
        parts.append(cur)
    return parts


def _process(gj):
    out = []
    for ft in gj.get("features", []):
        pr = ft.get("properties") or {}
        g = ft.get("geometry") or {}
        name = str(pr.get("Name") or pr.get("name") or "")
        codes = re.findall(r"[A-Z]{2}", name)
        a = pr.get("PlateA") or (codes[0] if codes else "")
        b = pr.get("PlateB") or (codes[1] if len(codes) > 1 else "")
        lines = [g["coordinates"]] if g.get("type") == "LineString" else g.get("coordinates", []) if g.get("type") == "MultiLineString" else []
        sub = "/" in name or "\\" in name          # en PB2002 «/» y «\» marcan zonas de subducción
        for ln in lines:
            for part in _split_dateline(ln):
                out.append({"a": a, "b": b, "t": _tier(a, b), "sub": sub,
                            "n": f"{NOMBRES.get(a, a)} – {NOMBRES.get(b, b)}", "c": part})
    return {"modelo": "PB2002 (Bird, 2003)", "limites": out}


def load(folder, log=print):
```

```
    """Límites procesados (desde la caché o descargándolos una vez). None si no hay forma de ob
tenerlos."""
    path = os.path.join(folder, "placas_pb2002.json")
    with _lock:
        try:
            with open(path, encoding="utf-8") as f:
                return json.load(f)
        except Exception:
            pass
        for url in URLS:
            try:
                data = _process(json.loads(http_get(url, 60).decode("utf-8")))
                if data["limites"]:
                    with open(path, "w", encoding="utf-8") as f:
                        json.dump(data, f, separators=(",", ":"))
                    log(f"Placas tectónicas: {len(data['limites'])} segmentos de límite descarg
ados (PB2002)")
                    return data
            except Exception as e:
                log(f"Placas tectónicas: {url} no respondió ({e})")
        return None
```

sismo/places.py — 224 líneas

```
"""Referencias geográficas mundiales para describir epicentros ("120 km al O de Valparaíso, Chile").

- Ciudades de referencia de todo el mundo, con énfasis en las zonas sísmicas (cinturón de fuego del
  Pacífico, cinturón Alpino-Himalayo, dorsales con islas habitadas).
- Región sísmica Flinn-Engdahl (estándar del USGS/EMSC, en inglés) mediante ObsPy si está instalado.
"""
from functools import lru_cache

import numpy as np

from .locator import dist_km, bearing

CITIES = [
    # México y Centroamérica
    ("Ciudad de México", 19.43, -99.13), ("Acapulco, Gro., México", 16.86, -99.88), ("Oaxaca, México", 17.07, -96.73),
    ("Salina Cruz, Oax., México", 16.17, -95.2), ("Pinotepa Nacional, Oax., México", 16.34, -98.05),
    ("Tapachula, Chis., México", 14.9, -92.26), ("Tuxtla Gutiérrez, México", 16.75, -93.12),
    ("Lázaro Cárdenas, Mich., México", 17.96, -102.2), ("Manzanillo, Col., México", 19.05, -104.32),
    ("Puerto Vallarta, México", 20.65, -105.23), ("Guadalajara, México", 20.67, -103.35), ("Veracruz, México", 19.17, -96.13),
    ("Mérida, México", 20.97, -89.62), ("Mazatlán, México", 23.25, -106.41), ("La Paz, BCS, México", 24.14, -110.31),
    ("Cabo San Lucas, México", 22.89, -109.91), ("Guaymas, Son., México", 27.92, -110.9), ("Mexicali, México", 32.62, -115.45),
    ("Ensenada, México", 31.87, -116.6), ("Monterrey, México", 25.67, -100.31), ("Chihuahua, México", 28.63, -106.07),
    ("Ciudad de Guatemala", 14.63, -90.51), ("San Salvador, El Salvador", 13.69, -89.22), ("Tegucigalpa, Honduras", 14.08, -87.21),
    ("Managua, Nicaragua", 12.13, -86.25), ("San José, Costa Rica", 9.93, -84.08), ("Ciudad de Panamá", 8.98, -79.52),
    ("Puerto Armuelles, Panamá", 8.28, -82.86), ("Belmopán, Belice", 17.25, -88.77),
    # Caribe
    ("La Habana, Cuba", 23.11, -82.37), ("Santiago de Cuba", 20.02, -75.82), ("Kingston, Jamaica", 17.97, -76.79),
    ("Puerto Príncipe, Haití", 18.54, -72.34), ("Santo Domingo, R. Dominicana", 18.47, -69.89),
    ("San Juan, Puerto Rico", 18.47, -66.11),
    ("Basse-Terre, Guadalupe", 16.0, -61.73), ("Fort-de-France, Martinica", 14.6, -61.07), ("Bridgetown, Barbados", 13.1, -59.61),
    ("Puerto España, Trinidad", 10.65, -61.52), ("Willemstad, Curazao", 12.11, -68.93), ("Nassau, Bahamas", 25.05, -77.35),
    # Sudamérica
    ("Caracas, Venezuela", 10.49, -66.88), ("Cumaná, Venezuela", 10.46, -64.17), ("Bogotá, Colombia", 4.71, -74.07),
    ("Cali, Colombia", 3.45, -76.53), ("Medellín, Colombia", 6.24, -75.58), ("Tumaco, Colombia", 1.8, -78.76),
    ("Quito, Ecuador", -0.18, -78.47), ("Guayaquil, Ecuador", -2.17, -79.92), ("Esmeraldas, Ecuador", 0.97, -79.65),
    ("Puerto Ayora, Galápagos", -0.74, -90.31), ("Lima, Perú", -12.05, -77.04), ("Arequipa, Perú", -16.41, -71.54),
    ("Piura, Perú", -5.19, -80.63), ("Trujillo, Perú", -8.11, -79.03), ("Cusco, Perú", -13.53, -71.97), ("Ica, Perú", -14.07, -75.73),
    ("Tacna, Perú", -18.01, -70.25), ("Iquitos, Perú", -3.75, -73.25), ("La Paz, Bolivia", -16.5, -68.15),
    ("Santa Cruz, Bolivia", -17.78, -63.18), ("Arica, Chile", -18.48, -70.31), ("Iquique, Chile", -20.21, -70.15),
    ("Antofagasta, Chile", -23.65, -70.4), ("Copiapó, Chile", -27.37, -70.33), ("La Serena, Chile", -29.9, -71.25),
    ("Valparaíso, Chile", -33.05, -71.62), ("Santiago, Chile", -33.45, -70.67), ("Concepción, Chile", -36.83, -73.05),
    ("Valdivia, Chile", -39.81, -73.25), ("Puerto Montt, Chile", -41.47, -72.94), ("Coyhaique, Chile", -45.57, -72.07),
    ("Punta Arenas, Chile", -53.16, -70.91), ("Ushuaia, Argentina", -54.8, -68.3), ("Mendoza, Argentina", -32.89, -68.83),
    ("San Juan, Argentina", -31.54, -68.54), ("Salta, Argentina", -24.79, -65.41), ("Santiago del Estero, Argentina", -27.8, -64.26),
    ("Buenos Aires, Argentina", -34.6, -58.38), ("Asunción, Paraguay", -25.26, -57.58), ("Montevideo, Uruguay", -34.9, -56.16),
    ("São Paulo, Brasil", -23.55, -46.63), ("Río de Janeiro, Brasil", -22.91, -43.17), ("Brasilia, Brasil", -15.79, -47.88),
    ("Manaos, Brasil", -3.12, -60.02), ("Recife, Brasil", -8.05, -34.88), ("Rio Branco, Brasil", -9.97, -67.81),
    ("Stanley, Islas Malvinas", -51.7, -57.85), ("Grytviken, Georgia del Sur", -54.28, -36.51),
    # Norteamérica
    ("Los Ángeles, EUA", 34.05, -118.24), ("San Diego, EUA", 32.72, -117.16), ("San Francisco, EUA", 37.77, -122.42),
    ("Eureka, California, EUA", 40.8, -124.16), ("Las Vegas, EUA", 36.17, -115.14), ("Reno, EUA", 39.53, -119.81),
    ("Portland, EUA", 45.52, -122.68), ("Seattle, EUA", 47.61, -122.33), ("Salt Lake City, EUA", 40.76, -111.89),
    ("Phoenix, EUA", 33.45, -112.07), ("Denver, EUA", 39.74, -104.99), ("Albuquerque, EUA", 35.08, -106.65),
    ("Dallas, EUA", 32.78, -96.8), ("Houston, EUA", 29.76, -95.37), ("Oklahoma City, EUA", 35.47, -97.52),
    ("Memphis, EUA", 35.15, -90.05), ("St. Louis, EUA", 38.63, -90.2), ("Chicago, EUA", 41.88, -87.63), ("Miami, EUA", 25.76, -80.19),
    ("Atlanta, EUA", 33.75, -84.39), ("Washington, EUA", 38.9, -77.04), ("Nueva York, EUA", 40.71, -74.01), ("Boston, EUA", 42.36, -71.06),
    ("Yellowstone, EUA", 44.43, -110.59), ("Helena, Montana, EUA", 46.59, -112.04), ("Boise, EUA", 43.62, -116.2),
```

```
    ("Vancouver, Canadá", 49.28, -123.12), ("Victoria, Canadá", 48.43, -123.37), ("Port Hardy, Canadá", 50.72, -127.5),
    ("Haida Gwaii, Canadá", 53.25, -132.08), ("Prince Rupert, Canadá", 54.32, -130.32), ("White horse, Canadá", 60.72, -135.06),
    ("Calgary, Canadá", 51.05, -114.07), ("Montreal, Canadá", 45.5, -73.57), ("Ottawa, Canadá", 45.42, -75.69),
    ("Yellowknife, Canadá", 62.45, -114.37), ("Iqaluit, Canadá", 63.75, -68.52), ("Resolute, Canadá", 74.7, -94.83),
    ("Anchorage, Alaska", 61.22, -149.9), ("Fairbanks, Alaska", 64.84, -147.72), ("Juneau, Alaska", 58.3, -134.42),
    ("Yakutat, Alaska", 59.55, -139.73), ("Kodiak, Alaska", 57.79, -152.41), ("Sand Point, Alaska", 55.34, -160.5),
    ("Unalaska, Alaska", 53.87, -166.54), ("Adak, Alaska", 51.88, -176.66), ("Attu, Alaska", 52.84, 173.18), ("Nome, Alaska", 64.5, -165.41),
    ("Nuuk, Groenlandia", 64.18, -51.72), ("Reikiavik, Islandia", 64.15, -21.94), ("Akureyri, Islandia", 65.68, -18.09),
    # Pacífico
    ("Honolulu, Hawái", 21.31, -157.86), ("Hilo, Hawái", 19.72, -155.08), ("Papeete, Tahití", -17.54, -149.57),
    ("Apia, Samoa", -13.83, -171.76), ("Pago Pago, Samoa Americana", -14.28, -170.7), ("Nukualofa, Tonga", -21.14, -175.2),
    ("Neiafu, Tonga", -18.65, -173.98), ("Suva, Fiyi", -18.14, 178.44), ("Lambasa, Fiyi", -16.42, 179.38),
    ("Port Vila, Vanuatu", -17.73, 168.32), ("Luganville, Vanuatu", -15.51, 167.18), ("Numea, Nueva Caledonia", -22.27, 166.46),
    ("Honiara, Islas Salomón", -9.43, 159.95), ("Kirakira, Islas Salomón", -10.45, 161.92), ("Port Moresby, Papúa Nueva Guinea", -9.44, 147.18),
    ("Lae, Papúa Nueva Guinea", -6.72, 146.99), ("Rabaul, Papúa Nueva Guinea", -4.2, 152.17), ("Kimbe, Papúa Nueva Guinea", -5.55, 150.14),
    ("Arawa, Bougainville", -6.23, 155.57), ("Kavieng, Papúa Nueva Guinea", -2.57, 150.8), ("Wewak, Papúa Nueva Guinea", -3.55, 143.63),
    ("Hagåtña, Guam", 13.48, 144.75), ("Saipán, Islas Marianas", 15.18, 145.75), ("Majuro, Islas Marshall", 7.09, 171.38),
    ("Tarawa, Kiribati", 1.45, 173.0), ("Hanga Roa, Isla de Pascua", -27.15, -109.43), ("Adamstown, Islas Pitcairn", -25.07, -130.1),
    ("Avarua, Islas Cook", -21.21, -159.78), ("Isla Macquarie, Australia", -54.5, 158.94),
    # Nueva Zelanda y Australia
    ("Wellington, Nueva Zelanda", -41.29, 174.78), ("Auckland, Nueva Zelanda", -36.85, 174.76), ("Gisborne, Nueva Zelanda", -38.66, 178.02),
    ("Napier, Nueva Zelanda", -39.49, 176.91), ("Christchurch, Nueva Zelanda", -43.53, 172.64), ("Dunedin, Nueva Zelanda", -45.87, 170.5),
    ("Invercargill, Nueva Zelanda", -46.41, 168.35), ("Islas Kermadec (Raoul)", -29.27, -177.92),
    ("Sídney, Australia", -33.87, 151.21), ("Melbourne, Australia", -37.81, 144.96), ("Brisbane, Australia", -27.47, 153.03),
    ("Perth, Australia", -31.95, 115.86), ("Adelaida, Australia", -34.93, 138.6), ("Darwin, Australia", -12.46, 130.84),
    ("Alice Springs, Australia", -23.7, 133.88), ("Hobart, Australia", -42.88, 147.33), ("Port Hedland, Australia", -20.31, 118.6),
    # Japón, Kuriles, Kamchatka, Rusia
    ("Tokio, Japón", 35.68, 139.69), ("Osaka, Japón", 34.69, 135.5), ("Nagoya, Japón", 35.18, 136.91), ("Sendai, Japón", 38.27, 140.87),
    ("Miyako, Japón", 39.64, 141.95), ("Hachinohe, Japón", 40.51, 141.49), ("Sapporo, Japón", 43.06, 141.35), ("Kushiro, Japón", 42.98, 144.38),
    ("Nemuro, Japón", 43.33, 145.58), ("Niigata, Japón", 37.92, 139.04), ("Kanazawa, Japón", 36.56, 136.66), ("Hiroshima, Japón", 34.39, 132.46),
    ("Kochi, Japón", 33.56, 133.53), ("Fukuoka, Japón", 33.59, 130.4), ("Kumamoto, Japón", 32.8, 130.71), ("Kagoshima, Japón", 31.6, 130.56),
    ("Naha, Okinawa, Japón", 26.21, 127.68), ("Ishigaki, Japón", 24.34, 124.16), ("Hachijō-jima, Japón", 33.11, 139.79),
    ("Chichi-jima, Japón", 27.09, 142.19), ("Severo-Kurilsk, Rusia", 50.68, 156.12), ("Kurilsk, Rusia", 45.23, 147.88),
    ("Petropavlovsk-Kamchatski, Rusia", 53.02, 158.65), ("Ust-Kamchatsk, Rusia", 56.22, 162.48), ("Magadán, Rusia", 59.56, 150.8),
    ("Yuzhno-Sajalinsk, Rusia", 46.96, 142.74), ("Vladivostok, Rusia", 43.12, 131.89), ("Irkutsk, Rusia", 52.29, 104.3),
    ("Yakutsk, Rusia", 62.03, 129.73), ("Anádyr, Rusia", 64.73, 177.51), ("Novosibirsk, Rusia", 55.01, 82.93), ("Moscú, Rusia", 55.76, 37.62),
    ("Sochi, Rusia", 43.6, 39.73), ("Majachkalá, Rusia", 42.98, 47.5), ("Norilsk, Rusia", 69.35, 88.2), ("Tiksi, Rusia", 71.64, 128.87),
    # Asia oriental y sudoriental
    ("Seúl, Corea del Sur", 37.57, 126.98), ("Busan, Corea del Sur", 35.18, 129.08), ("Pionyang, Corea del Norte", 39.02, 125.75),
    ("Pekín, China", 39.9, 116.4), ("Tangshan, China", 39.63, 118.18), ("Shanghái, China", 31.23, 121.47), ("Cantón, China", 23.13, 113.26),
    ("Hong Kong, China", 22.32, 114.17), ("Chengdú, China", 30.57, 104.07), ("Kunming, China", 25.04, 102.71), ("Lanzhou, China", 36.06, 103.83),
    ("Xining, China", 36.62, 101.78), ("Lhasa, China", 29.65, 91.13), ("Urumchi, China", 43.83, 87.62), ("Kashgar, China", 39.47, 75.99),
    ("Hotan, China", 37.11, 79.92), ("Golmud, China", 36.4, 94.9), ("Taipéi, Taiwán", 25.03, 121.57), ("Hualien, Taiwán", 23.99, 121.6),
    ("Kaohsiung, Taiwán", 22.63, 120.3), ("Manila, Filipinas", 14.6, 120.98), ("Baguio, Filipinas", 16.4, 120.6), ("Legazpi, Filipinas", 13.14, 123.74),
    ("Cebú, Filipinas", 10.32, 123.89), ("Tacloban, Filipinas", 11.24, 125.0), ("Davao, Filipinas", 7.07, 125.61),
    ("General Santos, Filipinas", 6.11, 125.17), ("Surigao, Filipinas", 9.79, 125.49), ("Zamboanga, Filipinas", 6.91, 122.08),
    ("Hanói, Vietnam", 21.03, 105.85), ("Ho Chi Minh, Vietnam", 10.82, 106.63), ("Bangkok, Tailandia", 13.76, 100.5),
    ("Chiang Mai, Tailandia", 18.79, 98.98), ("Rangún, Myanmar", 16.87, 96.2), ("Mandalay, Myanmar", 21.97, 96.08),
    ("Kuala Lumpur, Malasia", 3.14, 101.69), ("Kota Kinabalu, Malasia", 5.98, 116.07), ("Singapur", 1.35, 103.82),
    ("Yakarta, Indonesia", -6.21, 106.85), ("Bandung, Indonesia", -6.92, 107.61), ("Yogyakarta, Indonesia", -7.8, 110.36),
```

```
    ("Surabaya, Indonesia", -7.25, 112.75), ("Denpasar, Bali, Indonesia", -8.65, 115.22), ("Mataram, Lombok, Indonesia", -8.58, 116.12),
    ("Kupang, Indonesia", -10.17, 123.61), ("Maumere, Flores, Indonesia", -8.62, 122.21), ("Banda Aceh, Indonesia", 5.55, 95.32),
    ("Medan, Indonesia", 3.59, 98.67), ("Padang, Indonesia", -0.95, 100.35), ("Bengkulu, Indonesia", -3.8, 102.27),
    ("Gunungsitoli, Nias, Indonesia", 1.29, 97.61), ("Palu, Indonesia", -0.9, 119.87), ("Makassar, Indonesia", -5.15, 119.43),
    ("Manado, Indonesia", 1.47, 124.84), ("Gorontalo, Indonesia", 0.54, 123.06), ("Ternate, Indonesia", 0.79, 127.38),
    ("Ambon, Indonesia", -3.7, 128.18), ("Tual, Indonesia", -5.64, 132.75), ("Sorong, Indonesia", -0.88, 131.26),
    ("Manokwari, Indonesia", -0.86, 134.06), ("Jayapura, Indonesia", -2.53, 140.72), ("Nabire, Indonesia", -3.37, 135.5),
    ("Dili, Timor Oriental", -8.56, 125.57), ("Saumlaki, Indonesia", -7.98, 131.3), ("Bandar Seri Begawan, Brunéi", 4.9, 114.94),
    # Asia meridional y central
    ("Katmandú, Nepal", 27.72, 85.32), ("Pokhara, Nepal", 28.21, 83.99), ("Timbu, Bután", 27.47, 89.64), ("Daca, Bangladés", 23.81, 90.41),
    ("Chittagong, Bangladés", 22.36, 91.78), ("Nueva Delhi, India", 28.61, 77.21), ("Bombay, India", 19.08, 72.88), ("Calcuta, India", 22.57, 88.36),
    ("Chennai, India", 13.08, 80.27), ("Bangalore, India", 12.97, 77.59), ("Guwahati, India", 26.14, 91.74), ("Imphal, India", 24.82, 93.94),
    ("Srinagar, India", 34.08, 74.8), ("Bhuj, India", 23.24, 69.67), ("Port Blair, Islas Andamán", 11.62, 92.73), ("Colombo, Sri Lanka", 6.93, 79.86),
    ("Malé, Maldivas", 4.18, 73.51), ("Islamabad, Pakistán", 33.68, 73.05), ("Karachi, Pakistán", 24.86, 67.0), ("Quetta, Pakistán", 30.18, 66.98),
    ("Peshawar, Pakistán", 34.01, 71.58), ("Kabul, Afganistán", 34.56, 69.21), ("Feyzabad, Afganistán", 37.12, 70.58),
    ("Herat, Afganistán", 34.35, 62.2), ("Dusambé, Tayikistán", 38.56, 68.79), ("Biskek, Kirguistán", 42.87, 74.59),
    ("Almaty, Kazajistán", 43.24, 76.89), ("Taskent, Uzbekistán", 41.3, 69.24), ("Asjabad, Turkmenistán", 37.95, 58.38),
    ("Ulán Bator, Mongolia", 47.89, 106.91),
    # Medio Oriente, Cáucaso, Turquía
    ("Teherán, Irán", 35.69, 51.39), ("Tabriz, Irán", 38.08, 46.29), ("Mashhad, Irán", 36.3, 59.6), ("Kermanshah, Irán", 34.31, 47.07),
    ("Shiraz, Irán", 29.59, 52.58), ("Bandar Abbas, Irán", 27.18, 56.27), ("Kerman, Irán", 30.28, 57.08), ("Bam, Irán", 29.1, 58.36),
    ("Zahedán, Irán", 29.5, 60.86), ("Bagdad, Irak", 33.31, 44.37), ("Erbil, Irak", 36.19, 44.01), ("Kuwait", 29.38, 47.99),
    ("Riad, Arabia Saudita", 24.71, 46.68), ("Yeda, Arabia Saudita", 21.49, 39.19), ("Saná, Yemen", 15.37, 44.19), ("Adén, Yemen", 12.79, 45.03),
    ("Mascate, Omán", 23.59, 58.41), ("Dubái, EAU", 25.2, 55.27), ("Damasco, Siria", 33.51, 36.28), ("Alepo, Siria", 36.2, 37.13),
    ("Beirut, Líbano", 33.89, 35.5), ("Jerusalén", 31.77, 35.21), ("Amán, Jordania", 31.95, 35.93), ("Ereván, Armenia", 40.18, 44.51),
    ("Tiflis, Georgia", 41.72, 44.79), ("Bakú, Azerbaiyán", 40.41, 49.87), ("Ankara, Turquía", 39.93, 32.86), ("Estambul, Turquía", 41.01, 28.98),
    ("Esmirna, Turquía", 38.42, 27.14), ("Antalya, Turquía", 36.9, 30.7), ("Adana, Turquía", 37.0, 35.32), ("Kahramanmaraş, Turquía", 37.58, 36.94),
    ("Malatya, Turquía", 38.35, 38.31), ("Erzurum, Turquía", 39.9, 41.27), ("Van, Turquía", 38.49, 43.38), ("Nicosia, Chipre", 35.17, 33.36),
    # Europa
    ("Atenas, Grecia", 37.98, 23.73), ("Salónica, Grecia", 40.64, 22.94), ("Heraclión, Creta", 35.34, 25.13), ("Patras, Grecia", 38.25, 21.73),
    ("Rodas, Grecia", 36.43, 28.22), ("Tirana, Albania", 41.33, 19.82), ("Skopie, Macedonia del Norte", 42.0, 21.43), ("Sofía, Bulgaria", 42.7, 23.32),
    ("Bucarest, Rumania", 44.43, 26.1), ("Belgrado, Serbia", 44.79, 20.45), ("Sarajevo, Bosnia", 43.86, 18.41), ("Zagreb, Croacia", 45.81, 15.98),
    ("Liubliana, Eslovenia", 46.06, 14.51), ("Roma, Italia", 41.9, 12.5), ("Nápoles, Italia", 40.85, 14.27), ("L'Aquila, Italia", 42.35, 13.4),
    ("Perugia, Italia", 43.11, 12.39), ("Bolonia, Italia", 44.49, 11.34), ("Milán, Italia", 45.46, 9.19), ("Catania, Sicilia", 37.5, 15.09),
    ("Palermo, Sicilia", 38.12, 13.36), ("Cosenza, Italia", 39.3, 16.25), ("La Valeta, Malta", 35.9, 14.51), ("Viena, Austria", 48.21, 16.37),
    ("Zúrich, Suiza", 47.38, 8.54), ("Berlín, Alemania", 52.52, 13.4), ("Colonia, Alemania", 50.94, 6.96), ("París, Francia", 48.86, 2.35),
    ("Niza, Francia", 43.7, 7.27), ("Madrid, España", 40.42, -3.7), ("Granada, España", 37.18, -3.6), ("Murcia, España", 37.99, -1.13),
    ("Barcelona, España", 41.39, 2.17), ("Lisboa, Portugal", 38.72, -9.14), ("Faro, Portugal", 37.02, -7.93), ("Ponta Delgada, Azores", 37.74, -25.67),
    ("Horta, Azores", 38.53, -28.63), ("Funchal, Madeira", 32.65, -16.91), ("Las Palmas, Canarias", 28.12, -15.43),
    ("Santa Cruz de Tenerife, Canarias", 28.46, -16.25), ("Londres, Reino Unido", 51.51, -0.13), ("Dublín, Irlanda", 53.35, -6.26),
    ("Oslo, Noruega", 59.91, 10.75), ("Bergen, Noruega", 60.39, 5.32), ("Tromsø, Noruega", 69.65, 18.96), ("Longyearbyen, Svalbard", 78.22, 15.65),
    ("Jan Mayen", 70.94, -8.67), ("Estocolmo, Suecia", 59.33, 18.07), ("Helsinki, Finlandia", 60.17, 24.94), ("Varsovia, Polonia", 52.23, 21.01),
    ("Kiev, Ucrania", 50.45, 30.52), ("Chisináu, Moldavia", 47.01, 28.86),
    # África
    ("Argel, Argelia", 36.75, 3.06), ("Orán, Argelia", 35.7, -0.63), ("Constantina, Argelia", 36.37, 6.61), ("Rabat, Marruecos", 34.02, -6.84),
    ("Marrakech, Marruecos", 31.63, -7.98), ("Agadir, Marruecos", 30.43, -9.6), ("Alhucemas, Marruecos", 35.25, -3.93), ("Túnez", 36.81, 10.18),
    ("Trípoli, Libia", 32.89, 13.19), ("El Cairo, Egipto", 30.04, 31.24), ("Asuán, Egipto", 24.09, 32.9), ("Jartum, Sudán", 15.5, 32.56),
    ("Adís Abeba, Etiopía", 9.03, 38.74), ("Asmara, Eritrea", 15.32, 38.93), ("Yibuti", 11.59, 43.15), ("Mogadiscio, Somalia", 2.05, 45.32),
    ("Nairobi, Kenia", -1.29, 36.82), ("Kampala, Uganda", 0.35, 32.58), ("Kigali, Ruanda", -1.94, 30.06), ("Goma, R. D. del Congo", -1.68, 29.22),
    ("Bukavu, R. D. del Congo", -2.51, 28.86), ("Kinsasa, R. D. del Congo", -4.44, 15.27), ("Dar es-Salam, Tanzania", -6.79, 39.21),
    ("Dodoma, Tanzania", -6.16, 35.75), ("Lusaka, Zambia", -15.39, 28.32), ("Lilongüe, Malaui",
```

```
     -13.96, 33.79), ("Beira, Mozambique", -19.84, 34.84),
    ("Maputo, Mozambique", -25.97, 32.57), ("Harare, Zimbabue", -17.83, 31.05), ("Johannesburgo, Sudáfrica", -26.2, 28.05),
    ("Ciudad del Cabo, Sudáfrica", -33.92, 18.42), ("Windhoek, Namibia", -22.56, 17.08), ("Gaborone, Botsuana", -24.65, 25.91),
    ("Luanda, Angola", -8.84, 13.23), ("Lagos, Nigeria", 6.52, 3.38), ("Acra, Ghana", 5.6, -0.19), ("Abiyán, Costa de Marfil", 5.36, -4.01),
    ("Dakar, Senegal", 14.72, -17.47), ("Duala, Camerún", 4.05, 9.7), ("Malabo, Guinea Ecuatorial", 3.75, 8.78),
    ("Antananarivo, Madagascar", -18.88, 47.51), ("Port Louis, Mauricio", -20.16, 57.5), ("Saint-Denis, Reunión", -20.88, 55.45),
    ("Moroni, Comoras", -11.7, 43.26), ("Mamoudzou, Mayotte", -12.78, 45.23), ("Victoria, Seychelles", -4.62, 55.45),
    ("Praia, Cabo Verde", 14.93, -23.51), ("Jamestown, Santa Elena", -15.93, -5.72), ("Georgetown, Isla Ascensión", -7.93, -14.42),
    ("Edimburgo de los Siete Mares, Tristán da Cunha", -37.07, -12.31), ("Diego García", -7.31, 72.41),
    ("Port-aux-Français, Kerguelen", -49.35, 70.22), ("Isla Amsterdam", -37.8, 77.57),
    # Antártida
    ("Base McMurdo, Antártida", -77.85, 166.67), ("Polo Sur", -89.99, 0.0), ("Base Esperanza, Antártida", -63.4, -56.99),
    ("Base Casey, Antártida", -66.28, 110.53), ("Base Mawson, Antártida", -67.6, 62.87), ("Dumont d'Urville, Antártida", -66.66, 140.0),
    ("Base Syowa, Antártida", -69.0, 39.58),
]
COMPASS = ["N", "NE", "E", "SE", "S", "SO", "O", "NO"]


def _fe():
    try:
        from obspy.geodetics import FlinnEngdahl
        return FlinnEngdahl()
    except Exception:
        return None


_FE = [None, False]


@lru_cache(maxsize=4096)
def region_name(lat, lon):
    """Región Flinn-Engdahl (p. ej. 'NEAR COAST OF CENTRAL CHILE') o '' si ObsPy no está disponible."""
    if not _FE[1]:
        _FE[0], _FE[1] = _fe(), True
    if _FE[0] is None:
        return ""
    try:
        return _FE[0].get_region(float(lon), float(lat)).title()
    except Exception:
        return ""


_CLAT = np.array([c[1] for c in CITIES])
_CLON = np.array([c[2] for c in CITIES])


def nearest_city(lat, lon):
    d = dist_km(_CLAT, _CLON, lat, lon)
    j = int(np.argmin(d))
    return CITIES[j], float(d[j])


def where_is(lat, lon):
    return _where(round(float(lat), 3), round(float(lon), 3))


@lru_cache(maxsize=8192)
def _where(lat, lon):
    best, d = nearest_city(lat, lon)
    if d < 8:
        return f"Cerca de {best[0]}"
    b = float(bearing(best[1], best[2], lat, lon))
    txt = f"{round(d)} km al {COMPASS[int(round(b / 45)) % 8]} de {best[0]}"
    if d > 400:
        reg = region_name(round(lat, 2), round(lon, 2))
        if reg:
            return f"{reg} ({txt})"
    return txt
```

sismo/report.py — 546 líneas

```
"""Reporte PDF de Evento Sísmico de Alta Calidad para IIGEA-SIMMSA.
Incluye:
  Página 1: Boletín Ejecutivo (Ficha Técnica Hipocentral + Mecanismo Focal + Gráfica TravelTime vs Distancia).
  Página 2: Mapa Mundial de Cobertura de Estaciones con Continentes Reales (tierra.geojson), Rayos Radiales y Residuales P.
  Página 3: Sección de Registro Sísmico (Record Section ordenada por distancia).
  Página 4: Tabla Detallada de Fases y Estaciones.
"""
import os
import time
import json
import math
import numpy as np

import matplotlib
matplotlib.use("Agg")
import matplotlib.pyplot as plt
from matplotlib.backends.backend_pdf import PdfPages
from matplotlib.patches import Polygon, Circle
import matplotlib.patheffects as pe

from obspy.imaging.beachball import beach

try:
    from .locator import dist_km, KM_DEG
    from .places import where_is, CITIES
    from . import focal
except ImportError:
    from sismo.locator import dist_km, KM_DEG
    from sismo.places import where_is, CITIES
    from sismo import focal

R_E = 6371.0


def _fmt_cdmx(t):
    # UTC - 6 horas (hora estándar centro de México)
    return time.strftime("%d-%m-%Y %H:%M:%S", time.gmtime(t - 6 * 3600))


def _fmt_utc(t):
    return time.strftime("%Y-%m-%d %H:%M:%S", time.gmtime(t))


def _draw_continents(ax, data_dir, lon0):
    """Dibuja continentes reales desde tierra.geojson centrados en el epicentro (lon0)."""
    geojson_path = os.path.join(data_dir or "/opt/monitor_mundial/datos", "tierra.geojson")
    if not os.path.exists(geojson_path):
        return
    try:
        with open(geojson_path, "r", encoding="utf-8") as f:
            gj = json.load(f)
    except Exception:
        return

    wl = lambda x: ((np.asarray(x, float) - lon0 + 180) % 360) - 180 + lon0

    for ft in gj.get("features", []):
        g = ft.get("geometry") or {}
        g_type = g.get("type")
        coords = g.get("coordinates", [])
        polys = [coords] if g_type == "Polygon" else coords if g_type == "MultiPolygon" else []

        for poly in polys:
            if not poly or len(poly[0]) < 3:
                continue
            ring = poly[0]
            lons = np.array([pt[0] for pt in ring], dtype=float)
            lats = np.array([pt[1] for pt in ring], dtype=float)

            # Proyección envolvente centrada en lon0
            x = wl(lons)
            y = lats

            # Detección de salto del antimeridiano (|dx| > 180)
            diffs = np.abs(np.diff(x))
            jump_idx = np.where(diffs > 180)[0]

            if len(jump_idx) == 0:
                # Polígono continuo sin cruce de borde: rellenar y contornear
                pts = np.column_stack((x, y))
                ax.add_patch(Polygon(pts, closed=True, facecolor="#f8fafc", edgecolor="#94a3b8", lw=0.45, zorder=1))
            else:
                # Polígono que cruza el antimeridiano: trazar segmentos de línea sin artefactos horizontales
                segs = np.split(np.arange(len(x)), jump_idx + 1)
                for seg in segs:
                    if len(seg) >= 2:
                        ax.plot(x[seg], y[seg], color="#94a3b8", lw=0.5, zorder=1)


def make_pdf(engine, ev, path):
    data_dir = getattr(engine, "data_dir", "/opt/monitor_mundial/datos")
```

```
    M = engine.mag_of(ev)
    q = engine.quality(ev)
    mag_str = engine.mag_str(ev)
    mag_type = ev.get("magType") or "M"
    reg_name = where_is(ev["lat"], ev["lon"])
    fe_region = ev.get("region") or ""

    # Evaluar mecanismo focal si aún no existe
    if not ev.get("focal") and len([p for p in ev.get("picks", []) if p.get("phase") == "P"]) >= 4:
        try:
            focal.evaluate_event_focal(engine, ev)
        except Exception:
            pass
    foc = ev.get("focal")

    with PdfPages(path) as pdf:
        # ========================================================================
        # PÁGINA 1: BOLETÍN EJECUTIVO (FICHA HIPOCENTRAL + MECANISMO FOCAL + DROMOCRONA)
        # ========================================================================
        fig = plt.figure(figsize=(8.5, 11), facecolor="#ffffff")

        # 1. ENCABEZADO INSTITUCIONAL
        fig.text(0.06, 0.955, "IIGEA-SIMMSA", fontsize=16, weight="bold", color="#0f172a", fontfamily="sans-serif")
        fig.text(0.24, 0.955, "·  MONITOR SÍSMICO MUNDIAL", fontsize=13, weight="semibold", color="#2563eb", fontfamily="sans-serif")
        fig.text(0.06, 0.938, "IIGEA-SIMMSA  ·  Red Sismológica Global de Monitoreo en Tiempo Real", fontsize=8, color="#64748b")
        fig.text(0.94, 0.938, f"Emisión: {_fmt_utc(time.time())} UTC", fontsize=8, color="#64748b", ha="right")

        # Línea separadora superior
        fig.add_artist(plt.Line2D([0.06, 0.94], [0.930, 0.930], color="#2563eb", lw=1.8))

        # 2. BANNER DEL EVENTO PRINCIPAL
        # Tarjeta de Magnitud y Título
        ev_title = f"M {mag_str} {mag_type}  —  {reg_name}"
        if fe_region and fe_region != reg_name:
            ev_title += f" ({fe_region})"
        fig.text(0.06, 0.902, ev_title, fontsize=12.5, weight="bold", color="#0f172a")

        t_utc_str = _fmt_utc(ev["org"]) + " UTC"
        t_cdmx_str = _fmt_cdmx(ev["org"]) + " (Hora Centro de México)"
        fig.text(0.06, 0.884, f"Tiempo de Origen:  {t_utc_str}   /   {t_cdmx_str}", fontsize=9, color="#334155")

        # 3. BLOQUE TÉCNICO (COLUMNA IZQUIERDA: PARÁMETROS, COLUMNA DERECHA: MECANISMO FOCAL)
        # Bounding box visual para Columna Izquierda (Parámetros Técnicos)
        ax_tech_bg = fig.add_axes([0.06, 0.535, 0.48, 0.335])
        ax_tech_bg.set_facecolor("#f8fafc")
        for spine in ax_tech_bg.spines.values():
            spine.set_color("#cbd5e1")
            spine.set_linewidth(0.8)
        ax_tech_bg.set_xticks([])
        ax_tech_bg.set_yticks([])

        # Contenido Columna Izquierda
        tech_lines = [
            ("Epicentro:", f"{ev['lat']:.3f}° N, {ev['lon']:.3f}° E" if ev['lat'] >= 0 else f"{abs(ev['lat']):.3f}° S, {ev['lon']:.3f}° E"),
            ("Profundidad:", f"{ev['z']:.1f} km ± {ev.get('errZ', 0):.1f} km" + (" [Fija]" if ev.get("zPrior") else "")),
            ("Calidad de Solución:", f"Grado {q}  (RMS: {ev.get('rms', 0):.2f} s)"),
            ("Gap Azimutal:", f"{ev.get('gap', 0):.0f}°  ·  Dist. Mínima: {ev.get('dmin', 0):.1f} km"),
            ("Fases Utilizadas:", f"{ev.get('nP', 0)} P, {ev.get('nS', 0)} S  (Total: {len(ev.get('picks', []))} arribos)"),
            ("Modelo de Velocidad:", f"{getattr(engine.tt, 'name', 'ak135')} (Tierra esférica)"),
        ]
        if ev.get("errH") and not ev.get("errOpen"):
            tech_lines.append(("Elipse de Error (68%):", f"{ev['errH']:.1f} × {ev.get('errMin', ev['errH']):.1f} km, Az: {ev.get('errAz', 0):.0f}°"))

        # Magnitudes detalladas
        m_list = []
        if ev.get("mw") is not None: m_list.append(f"Mw {ev['mw']:.1f}")
        if ev.get("mwp") is not None: m_list.append(f"Mwp {ev['mwp']:.1f}")
        if ev.get("mb") is not None: m_list.append(f"mb {ev['mb']:.1f}")
        if ev.get("mbR") is not None: m_list.append(f"mb(reg) {ev['mbR']:.1f}")
        if ev.get("mB") is not None: m_list.append(f"mB {ev['mB']:.1f}")
        if ev.get("ms") is not None: m_list.append(f"Ms {ev['ms']:.1f}")
        if ev.get("ml") is not None: m_list.append(f"ML {ev['ml']:.1f}")
        if m_list:
            tech_lines.append(("Magnitudes:", "  ·  ".join(m_list[:4])))
            if len(m_list) > 4:
                tech_lines.append(("", "  ·  ".join(m_list[4:])))

        # Fases de profundidad
        if ev.get("depthPhases") and (ev["depthPhases"].get("accepted") or ev.get("npP")):
            tech_lines.append(("Fases de Profundidad:", f"{ev.get('npP', 0)} pP, {ev.get('nsP', 0)} sP"))

        # Alerta de Tsunami
```

```
        if ev.get("tsunami"):
            tech_lines.append(("Evaluación Tsunami:", f"{ev['tsunami'].get('title', 'Evaluado')
}"))

        # Comparación USGS
        if ev.get("match"):
            m_usgs = ev["match"]
            tech_lines.append(("Catálogo USGS:", f"M{m_usgs.get('mag', '')} (ΔEpi: {m_usgs.get(
'dEpi', 0):.0f} km, ΔM: {m_usgs.get('dM', 0):+.1f})"))

        tech_lines.append(("Agencia Responsable:", "IIGEA-SIMMSA"))

        y_t = 0.852
        fig.text(0.08, 0.854, "PARÁMETROS HIPOCENTRALES", fontsize=9.5, weight="bold", color="#
1e293b")
        for lbl, val in tech_lines:
            if lbl:
                fig.text(0.08, y_t, lbl, fontsize=8.2, weight="bold", color="#475569")
                fig.text(0.25, y_t, val, fontsize=8.2, color="#0f172a")
            else:
                fig.text(0.25, y_t, val, fontsize=8.2, color="#0f172a")
            y_t -= 0.0245

        # Bounding box visual para Columna Derecha (Mecanismo Focal)
        ax_foc_bg = fig.add_axes([0.56, 0.535, 0.38, 0.335])
        ax_foc_bg.set_facecolor("#f8fafc")
        for spine in ax_foc_bg.spines.values():
            spine.set_color("#cbd5e1")
            spine.set_linewidth(0.8)
        ax_foc_bg.set_xticks([])
        ax_foc_bg.set_yticks([])

        fig.text(0.58, 0.854, "MECANISMO FOCAL (IIGEA-SIMMSA)", fontsize=9.5, weight="bold", co
lor="#1e293b")

        if foc:
            strike = foc.get("strike", 0)
            dip = foc.get("dip", 0)
            rake = foc.get("rake", 0)
            regime = foc.get("regime") or "Doble Pareja"
            quality = foc.get("quality", "C")
            misfit = foc.get("misfit", 0.0)
            n_pol = foc.get("n_pol", 0)
            s2 = foc.get("strike2", 0)
            d2 = foc.get("dip2", 0)
            r2 = foc.get("rake2", 0)

            # Eje dedicado para la Beachball
            ax_bb = fig.add_axes([0.575, 0.640, 0.17, 0.19])
            ax_bb.set_aspect("equal")
            ax_bb.axis("off")

            try:
                # Pelota de playa con ObsPy
                bb = beach([strike, dip, rake], xy=(0.5, 0.5), width=0.85, facecolor="#dc2626",
 edgecolor="#0f172a", linewidth=1.2)
                ax_bb.add_collection(bb)

                # Graficar polaridades en el hemisferio inferior si existen estaciones
                st_pols = foc.get("stations", [])
                for st in st_pols[:45]:
                    az_r = math.radians(st.get("az", 0))
                    tk_r = math.radians(st.get("takeoff", 90))
                    # Proyección estereográfica / Schmidt de igual área
                    # r = sqrt(2) * sin(takeoff/2) normalizado a radio de pelota (0.425)
                    r_norm = math.sqrt(2) * math.sin(min(tk_r, math.pi / 2) / 2) * 0.425
                    sx = 0.5 + r_norm * math.sin(az_r)
                    sy = 0.5 + r_norm * math.cos(az_r)
                    pol_char = st.get("pol")
                    if pol_char == "+":
                        ax_bb.plot(sx, sy, "o", ms=4, mfc="#ffffff", mec="#000000", mew=0.9, zo
rder=5)
                    elif pol_char == "-":
                        ax_bb.plot(sx, sy, "o", ms=3.5, mfc="#0f172a", mec="#ffffff", mew=0.7,
zorder=5)

                # Letras de ejes P y T
                axes_pt = foc.get("axes", {})
                if axes_pt.get("T"):
                    t_tr = math.radians(axes_pt["T"][0])
                    t_pl = math.radians(axes_pt["T"][1])
                    r_t = math.sqrt(2) * math.sin((math.pi / 2 - t_pl) / 2) * 0.38
                    ax_bb.text(0.5 + r_t * math.sin(t_tr), 0.5 + r_t * math.cos(t_tr), "T", col
or="#ffffff",
                               fontsize=8.5, weight="bold", ha="center", va="center", zorder=6,
                               path_effects=[pe.withStroke(linewidth=2, foreground="#0f172a")])
                if axes_pt.get("P"):
                    p_tr = math.radians(axes_pt["P"][0])
                    p_pl = math.radians(axes_pt["P"][1])
                    r_p = math.sqrt(2) * math.sin((math.pi / 2 - p_pl) / 2) * 0.38
                    ax_bb.text(0.5 + r_p * math.sin(p_tr), 0.5 + r_p * math.cos(p_tr), "P", col
or="#0f172a",
                               fontsize=8.5, weight="bold", ha="center", va="center", zorder=6,
                               path_effects=[pe.withStroke(linewidth=2, foreground="#ffffff")])
            except Exception as e:
```

```
                ax_bb.text(0.5, 0.5, f"Beachball: {e}", fontsize=7, ha="center")

            ax_bb.set_xlim(0, 1)
            ax_bb.set_ylim(0, 1)

            # Texto al lado y debajo de la Beachball
            fig.text(0.76, 0.775, regime, fontsize=10, weight="bold", color="#0f172a")
            fig.text(0.76, 0.755, f"Calidad: {quality}  ·  Misfit: {misfit:.1f}%", fontsize=8,
color="#475569")
            fig.text(0.76, 0.735, f"Polaridades: {n_pol} fases P", fontsize=8, color="#475569")

            fig.text(0.58, 0.620, f"Plano Nodal 1 (NP1):", fontsize=8.2, weight="bold", color="
#475569")
            fig.text(0.76, 0.620, f"Rumbo {strike:.0f}° / Buz {dip:.0f}° / Rake {rake:.0f}°", f
ontsize=8.2, color="#0f172a")

            fig.text(0.58, 0.598, f"Plano Nodal 2 (NP2):", fontsize=8.2, weight="bold", color="
#475569")
            fig.text(0.76, 0.598, f"Rumbo {s2:.0f}° / Buz {d2:.0f}° / Rake {r2:.0f}°", fontsize
=8.2, color="#0f172a")

            if axes_pt.get("P") and axes_pt.get("T"):
                fig.text(0.58, 0.576, f"Eje P (Presión):", fontsize=8.2, weight="bold", color="
#475569")
                fig.text(0.76, 0.576, f"Azimut {axes_pt['P'][0]:.0f}° / Inclinación {axes_pt['P
'][1]:.0f}°", fontsize=8.2, color="#0f172a")
                fig.text(0.58, 0.554, f"Eje T (Tensión):", fontsize=8.2, weight="bold", color="
#475569")
                fig.text(0.76, 0.554, f"Azimut {axes_pt['T'][0]:.0f}° / Inclinación {axes_pt['T
'][1]:.0f}°", fontsize=8.2, color="#0f172a")
        else:
            fig.text(0.75, 0.70, "Polaridades P insuficientes\npara inversión focal (< 4 fases)
", fontsize=9, color="#94a3b8", ha="center")

        # 4. MITAD INFERIOR: GRÁFICA TRAVELTIME VS DISTANCIA (DROMOCRONA COMPLETA)
        ax_drom = fig.add_axes([0.10, 0.08, 0.84, 0.39])
        ax_drom.set_facecolor("#ffffff")

        # Picks válidos con distancia
        picks = ev.get("picks", [])
        v_picks = []
        for p in picks:
            if p.get("t") is not None:
                d = dist_km(ev["lat"], ev["lon"], p["lat"], p["lon"])
                v_picks.append((d, p["t"] - ev["org"], p))

        max_d = max([x[0] for x in v_picks] + [1200.0]) * 1.05
        max_t = max([x[1] for x in v_picks] + [200.0]) + 30.0

        # Curvas teóricas con el modelo IASP91/ak135 del motor
        d_grid = np.linspace(0, max_d, 250)
        t_P_grid = []
        t_S_grid = []
        for dg in d_grid:
            try:
                tp_val = engine.ttf("P", dg, ev["z"])
            except Exception:
                deg = dg / KM_DEG
                tp_val = 6.0 + dg / 8.05 if deg <= 15 else (-0.045 * deg ** 2 + 11.2 * deg + 65
)
            t_P_grid.append(tp_val)

            # S solo si d < 103° (zona de sombra)
            if dg <= 103 * KM_DEG:
                try:
                    ts_val = engine.ttf("S", dg, ev["z"])
                except Exception:
                    ts_val = tp_val * 1.76
                t_S_grid.append(ts_val)
            else:
                t_S_grid.append(np.nan)

        ax_drom.plot(d_grid, t_P_grid, color="#dc2626", lw=1.8, label="Onda P teórica (ak135)",
 zorder=2)
        ax_drom.plot(d_grid, t_S_grid, color="#10b981", lw=1.8, label="Onda S teórica (ak135)",
 zorder=2)

        # Graficar arribos observados
        p_picks = [x for x in v_picks if x[2]["phase"] == "P"]
        s_picks = [x for x in v_picks if x[2]["phase"] == "S"]

        if p_picks:
            ax_drom.scatter([x[0] for x in p_picks], [x[1] for x in p_picks], color="#ef4444",
edgecolors="#7f1d1d",
                            s=36, marker="o", label=f"Arribos P observados ({len(p_picks)})", z
order=4)
        if s_picks:
            ax_drom.scatter([x[0] for x in s_picks], [x[1] for x in s_picks], color="#10b981",
edgecolors="#064e3b",
                            s=36, marker="s", label=f"Arribos S observados ({len(s_picks)})", z
order=4)

        ax_drom.set_xlim(0, max_d)
        ax_drom.set_ylim(0, max_t)
        ax_drom.set_title("CURVAS TEÓRICAS Y TIEMPOS DE VIAJE OBSERVADOS (TRAVELTIME VS DISTANC
```

```
IA)", fontsize=9.5, weight="bold", color="#0f172a", pad=8)
      ax_drom.set_xlabel("Distancia Epicentral (km)", fontsize=8.5, weight="bold", color="#334155")
      ax_drom.set_ylabel("Tiempo de Viaje desde el Origen (s)", fontsize=8.5, weight="bold", color="#334155")
      ax_drom.grid(True, linestyle="--", alpha=0.35, color="#94a3b8")
      ax_drom.legend(loc="upper left", fontsize=8, framealpha=0.9, edgecolor="#cbd5e1")
      ax_drom.tick_params(labelsize=8)

      # Eje secundario superior en grados epicentrales
      ax_deg = ax_drom.twiny()
      ax_deg.set_xlim(0, max_d / KM_DEG)
      ax_deg.set_xlabel("Distancia Epicentral (Grados °)", fontsize=8.5, color="#64748b")
      ax_deg.tick_params(labelsize=8, colors="#64748b")

      # Pie de página institucional
      fig.text(0.06, 0.022, "IIGEA-SIMMSA · Sistema de Detección y Monitoreo Sísmico Global en Tiempo Real", fontsize=7.5, color="#64748b")
      fig.text(0.94, 0.022, f"Página 1 de 4  ·  ID: {ev['id']}", fontsize=7.5, color="#64748b", ha="right")

      pdf.savefig(fig)
      plt.close(fig)

      # ═══════════════════════════════════════════════════════════════════
      # PÁGINA 2: MAPA MUNDIAL DE COBERTURA DE ESTACIONES CON CONTINENTES REALES
      # ═══════════════════════════════════════════════════════════════════
      fig = plt.figure(figsize=(11, 8.5), facecolor="#ffffff")

      # Encabezado
      fig.text(0.06, 0.950, "IIGEA-SIMMSA", fontsize=15, weight="bold", color="#0f172a")
      fig.text(0.20, 0.950, "·  COBERTURA RADIAL DE ESTACIONES SÍSMICAS", fontsize=12, weight="semibold", color="#2563eb")
      fig.text(0.06, 0.930, f"Estaciones que registraron la onda P del evento {ev['id']}  ·  Epicentro: {ev['lat']:.3f}°, {ev['lon']:.3f}°", fontsize=8.5, color="#64748b")
      fig.text(0.94, 0.930, f"Calidad: {q}  ·  Fases P: {len([p for p in ev['picks'] if p['phase'] == 'P'])}", fontsize=8.5, color="#64748b", ha="right")

      ax_map = fig.add_axes([0.06, 0.12, 0.88, 0.77])
      ax_map.set_facecolor("#e0f2fe")  # Azul océano limpio

      lon0 = ev["lon"]
      wl = lambda x: ((np.asarray(x, float) - lon0 + 180) % 360) - 180 + lon0

      # DIBUJAR CONTINENTES REALES CON TIERRA.GEOJSON
      _draw_continents(ax_map, data_dir, lon0)

      # Círculos geodésicos de distancia concéntrica
      for d_deg, ls, lbl in ((30, ":", "30° (3330 km)"), (60, ":", "60° (6660 km)"), (90, "--", "90° (10000 km)"), (120, ":", "120°")):
          az = np.linspace(0, 2 * np.pi, 361)
          la1 = np.radians(ev["lat"])
          dd = np.radians(d_deg)
          lat2 = np.arcsin(np.sin(la1) * np.cos(dd) + np.cos(la1) * np.sin(dd) * np.cos(az))
          lon2 = np.radians(ev["lon"]) + np.arctan2(np.sin(az) * np.sin(dd) * np.cos(la1), np.cos(dd) - np.sin(la1) * np.sin(lat2))
          x2, y2 = wl(np.degrees(lon2)), np.degrees(lat2)
          br = np.nonzero(np.abs(np.diff(x2)) > 180)[0]
          for seg in np.split(np.arange(len(x2)), br + 1):
              ax_map.plot(x2[seg], y2[seg], ls, color="#64748b", lw=0.75, zorder=2)
          # Etiqueta sobre el círculo
          lbl_lon = wl(ev["lon"] + d_deg)
          if -90 <= ev["lat"] <= 90:
              ax_map.text(lbl_lon, ev["lat"] + 1.2, lbl, fontsize=6.5, color="#475569", zorder=3)

      P = [p for p in ev["picks"] if p["phase"] == "P"]

      # Trazar rayos geodésicos radiales desde el epicentro hacia cada estación
      for p in P:
          px = wl(p["lon"])
          py = p["lat"]
          # Si no cruza el corte del antimeridiano
          if abs(px - lon0) < 180:
              ax_map.plot([lon0, px], [ev["lat"], py], color="#38bdf8", alpha=0.35, lw=0.6, zorder=3)

      # Graficar estaciones con color según residuo P
      if P:
          p_lons = wl([p["lon"] for p in P])
          p_lats = [p["lat"] for p in P]
          p_res = [p.get("res", 0.0) or 0.0 for p in P]

          sc = ax_map.scatter(p_lons, p_lats, c=p_res, cmap="coolwarm", vmin=-2.5, vmax=2.5,
                              marker="^", s=55, edgecolors="#0f172a", linewidths=0.7, zorder=4)

          # Barra de color inferior
          cb = fig.colorbar(sc, ax=ax_map, orientation="horizontal", fraction=0.035, pad=0.06, aspect=45)
          cb.set_label("Residuo de Tiempo de Viaje Onda P (s)  ·  Azul: Anticipada (-)  ·  Rojo: Tardía (+)", fontsize=8, color="#334155")
          cb.ax.tick_params(labelsize=7.5)

          # Etiquetas de texto para las 10 estaciones más cercanas
```

```
            P_sorted = sorted(P, key=lambda p: dist_km(ev["lat"], ev["lon"], p["lat"], p["lon"]
))
            for p in P_sorted[:12]:
                sta_code = (p.get("code") or p.get("key") or "").split(".")[1] if "." in (p.get
("code") or "") else p.get("code") or p.get("key")
                ax_map.text(wl(p["lon"]) + 1.5, p["lat"] + 0.8, sta_code, fontsize=6.5, weight=
"bold", color="#0f172a",
                            zorder=5, path_effects=[pe.withStroke(linewidth=1.5, foreground="#f
fffff")])

        # Epicentro (Estrella destacada)
        ax_map.plot(lon0, ev["lat"], "*", ms=18, color="#e11d48", markeredgecolor="#ffffff", ma
rkeredgewidth=1.2, zorder=6)
        ax_map.plot(lon0, ev["lat"], "o", ms=4, color="#ffffff", zorder=7)

        ax_map.set_xlim(lon0 - 180, lon0 + 180)
        ax_map.set_ylim(-85, 85)
        ax_map.set_xlabel("Longitud (grados, centrado en el epicentro)", fontsize=8, color="#47
5569")
        ax_map.set_ylabel("Latitud (grados)", fontsize=8, color="#475569")
        ax_map.tick_params(labelsize=7.5)

        fig.text(0.06, 0.020, "IIGEA-SIMMSA · Red Sismológica Mundial de Cobertura Continua", f
ontsize=7.5, color="#64748b")
        fig.text(0.94, 0.020, f"Página 2 de 4  ·  ID: {ev['id']}", fontsize=7.5, color="#64748b
", ha="right")

        pdf.savefig(fig)
        plt.close(fig)

        # ========================================================================
        # PÁGINA 3: SECCIÓN DE REGISTRO SÍSMICO (RECORD SECTION)
        # ========================================================================
        traces = engine.section_traces(ev) if ev.get("live") else (engine.storage.load_waveform
s(ev["id"]) if getattr(engine, "storage", None) else [])
        if traces:
            fig, ax = plt.subplots(figsize=(8.5, 11), facecolor="#ffffff")
            fig.subplots_adjust(top=0.92, bottom=0.06, left=0.10, right=0.94)

            fig.text(0.06, 0.955, "IIGEA-SIMMSA", fontsize=15, weight="bold", color="#0f172a")
            fig.text(0.24, 0.955, "·  SECCIÓN DE REGISTRO SÍSMICO", fontsize=12, weight="semibo
ld", color="#2563eb")
            fig.text(0.06, 0.935, f"Componente Z (0.5–4.0 Hz), normalizada y ordenada por dista
ncia epicentral creciente", fontsize=8, color="#64748b")

            items = []
            for tr in traces:
                s = engine.stations.get(tr["key"])
                ref = s or next((p for p in ev["picks"] if p["key"] == tr["key"]), None)
                if ref is None:
                    continue
                lat, lon = (s.lat, s.lon) if s else (ref["lat"], ref["lon"])
                items.append((float(dist_km(ev["lat"], ev["lon"], lat, lon)), tr))
            items.sort(key=lambda x: x[0])

            max_traces = min(len(items), 28)
            for i, (d, tr) in enumerate(items[:max_traces]):
                x_tr = tr["t0"] - ev["org"] + np.arange(len(tr["data"])) / tr["fs"]
                max_amp = np.abs(tr["data"]).max() or 1.0
                y_tr = tr["data"] / max_amp
                ax.plot(x_tr, i + 0.42 * y_tr, lw=0.6, color="#0284c7")

                sta_lbl = f"{tr['key'].replace('_', '.')} ({d:.0f} km · {d / KM_DEG:.1f}°)"
                ax.text(x_tr[0] if len(x_tr) else 0, i + 0.12, sta_lbl, fontsize=6.8, weight="b
old", color="#0f172a")

                for p in ev["picks"]:
                    if p["key"] == tr["key"]:
                        is_p = p["phase"] == "P"
                        ax.plot([p["t"] - ev["org"]] * 2, [i - 0.42, i + 0.42],
                                color="#dc2626" if is_p else "#10b981", lw=1.6, zorder=4)

            ax.set_xlabel("Segundos desde el Tiempo de Origen (s)", fontsize=8.5, weight="bold"
, color="#334155")
            ax.set_yticks([])
            ax.set_title("Marcas de Arribo: Rojo = Onda P, Verde = Onda S", fontsize=8.5, color
="#475569", pad=6)
            ax.grid(True, axis="x", linestyle="--", alpha=0.3, color="#94a3b8")

            fig.text(0.06, 0.020, "IIGEA-SIMMSA · Trazas sismológicas en tiempo real", fontsize
=7.5, color="#64748b")
            fig.text(0.94, 0.020, f"Página 3 de 4  ·  ID: {ev['id']}", fontsize=7.5, color="#64
748b", ha="right")

            pdf.savefig(fig)
            plt.close(fig)

        # ========================================================================
        # PÁGINA 4: TABLA DETALLADA DE FASES Y ESTACIONES
        # ========================================================================
        fig = plt.figure(figsize=(8.5, 11), facecolor="#ffffff")

        fig.text(0.06, 0.955, "IIGEA-SIMMSA", fontsize=15, weight="bold", color="#0f172a")
        fig.text(0.24, 0.955, "·  LISTADO DETALLADO DE FASES SÍSMICAS", fontsize=12, weight="se
mibold", color="#2563eb")
```

```
        fig.text(0.06, 0.935, f"Estaciones asociadas y residuales de la solución hipocentral  ·  Total: {len(ev['picks'])} fases", fontsize=8, color="#64748b")

        # Encabezado de tabla
        header_y = 0.905
        fig.text(0.06, header_y, "ESTACIÓN", fontsize=7.8, weight="bold", color="#0f172a")
        fig.text(0.20, header_y, "FASE", fontsize=7.8, weight="bold", color="#0f172a")
        fig.text(0.28, header_y, "DISTANCIA", fontsize=7.8, weight="bold", color="#0f172a")
        fig.text(0.42, header_y, "AZIMUT", fontsize=7.8, weight="bold", color="#0f172a")
        fig.text(0.52, header_y, "ARRIBO (UTC)", fontsize=7.8, weight="bold", color="#0f172a")
        fig.text(0.68, header_y, "TIEMPO REL.", fontsize=7.8, weight="bold", color="#0f172a")
        fig.text(0.80, header_y, "RESIDUO", fontsize=7.8, weight="bold", color="#0f172a")
        fig.text(0.89, header_y, "POL.", fontsize=7.8, weight="bold", color="#0f172a")

        fig.add_artist(plt.Line2D([0.06, 0.94], [header_y - 0.008, header_y - 0.008], color="#0f172a", lw=1.0))

        sorted_picks = sorted(ev["picks"], key=lambda p: p.get("t", 0))
        y_row = header_y - 0.024
        for i, p in enumerate(sorted_picks[:52]):
            d = dist_km(ev["lat"], ev["lon"], p["lat"], p["lon"])
            az = math.degrees(math.atan2(p["lon"] - ev["lon"], p["lat"] - ev["lat"])) % 360
            t_rel = p["t"] - ev["org"]
            res = p.get("res", 0.0) or 0.0
            res_str = f"{res:+.2f} s"
            pol_str = p.get("pol") or "·"

            # Fondo alternado
            if i % 2 == 1:
                ax_row_bg = fig.add_axes([0.06, y_row - 0.004, 0.88, 0.015])
                ax_row_bg.set_facecolor("#f1f5f9")
                ax_row_bg.axis("off")

            res_color = "#047857" if abs(res) <= 0.4 else "#b91c1c" if abs(res) > 1.0 else "#b45309"

            fig.text(0.06, y_row, p.get("code") or p.get("key"), fontsize=7.2, fontfamily="monospace", weight="bold")
            fig.text(0.20, y_row, p.get("phase"), fontsize=7.2, color="#dc2626" if p.get("phase") == "P" else "#10b981", weight="bold")
            fig.text(0.28, y_row, f"{d:.0f} km ({d / KM_DEG:.1f}°)", fontsize=7.2, fontfamily="monospace")
            fig.text(0.42, y_row, f"{az:.0f}°", fontsize=7.2, fontfamily="monospace")
            fig.text(0.52, y_row, f"{time.strftime('%H:%M:%S', time.gmtime(p['t']))}.{int((p['t'] % 1) * 100):02d}", fontsize=7.2, fontfamily="monospace")
            fig.text(0.68, y_row, f"+{t_rel:.2f} s", fontsize=7.2, fontfamily="monospace")
            fig.text(0.80, y_row, res_str, fontsize=7.2, fontfamily="monospace", weight="bold", color=res_color)
            fig.text(0.90, y_row, pol_str, fontsize=7.2, fontfamily="monospace", weight="bold")

            y_row -= 0.0158

        fig.text(0.06, 0.020, "IIGEA-SIMMSA · Red Sismológica Global", fontsize=7.5, color="#64748b")
        fig.text(0.94, 0.020, f"Página 4 de 4  ·  ID: {ev['id']}", fontsize=7.5, color="#64748b", ha="right")

        pdf.savefig(fig)
        plt.close(fig)

    return path
```

sismo/server.py — 942 líneas

```
"""Servidor web local: sirve el panel y una API JSON con el estado del motor (Monitor Sísmico MUNDIAL v2.3.1)."""
import base64
import json
import os
import re
import statistics
import tempfile
import threading
import time
import urllib.parse
from http.server import BaseHTTPRequestHandler, ThreadingHTTPServer

import numpy as np

from scipy import signal as _sig

from .dsp import bandpass
from .locator import dist_km
from .places import where_is
from .zones import zone_of
from .report import make_pdf
from . import ticket as tk


def _json_default(o):
    if isinstance(o, (set, frozenset)):
        return list(o)
    if isinstance(o, np.generic):
        return o.item()
    if isinstance(o, np.ndarray):
        return o.tolist()
    return str(o)


def _envelope(t_rel0, fs, data, nbins=1400):
    n = len(data)
    if n == 0:
        return None
    per = max(1, int(np.ceil(n / nbins)))
    m = n // per
    d = data[:m * per].reshape(m, per)
    return {"x0": float(t_rel0), "dx": per / fs, "lo": np.round(d.min(axis=1), 3).tolist(), "hi": np.round(d.max(axis=1), 3).tolist()}


PROVIDERS = [
    ("earthscope", "EarthScope", "EarthScope Consortium (EE. UU. / Global)", r"earthscope\.org"),
    ("geofon", "GEOFON", "GFZ Potsdam (Alemania / Global)", r"geofon\.gfz\.de"),
    ("orfeus", "ORFEUS / EIDA", "Observatories & Research Facilities for European Seismology", r"orfeus-eu\.org"),
    ("bgr", "BGR Alemania", "Federal Inst. Geosciences & Natural Resources (Alemania)", r"bgr\.de"),
    ("csn", "CSN Chile", "Centro Sismológico Nacional (Univ. de Chile)", r"csn\.uchile\.cl"),
    ("sed", "SED Suiza", "Servicio Sismológico Suizo (ETH Zürich)", r"ethz\.ch"),
    ("igf", "IGF Polonia", "Instytut Geofizyki PAN (Polonia)", r"igf\.edu\.pl"),
    ("auspass", "AusPass", "Australian Passive Seismic Server (Australia)", r"auspass\.edu\.au"),
    ("geonet", "GeoNet", "GNS Science / GeoNet (Nueva Zelanda)", r"geonet\.org\.nz"),
    ("usp", "USP Brasil", "Universidade de São Paulo (Centro de Sismologia)", r"usp\.br"),
    ("ineter", "INETER", "Instituto Nicaragüense de Estudios Territoriales", r"165\.98\.224\.42"),
    ("beg", "TexNet Texas", "Bureau of Economic Geology (Univ. of Texas)", r"beg\.utexas\.edu"),
    ("ou", "OGS Oklahoma", "Oklahoma Geological Survey (Univ. of Oklahoma)", r"ou\.edu"),
    ("uprm", "RSPR Puerto Rico", "Red Sísmica de Puerto Rico (Univ. de Puerto Rico)", r"uprm\.edu"),
    ("ens", "ENS Francia", "École Normale Supérieure (París, Francia)", r"ens\.fr"),
    ("sismocal", "Sismocal Italia", "Sismocal (Italia)", r"sismocal\.org"),
]


class NetworkTracker:
    def __init__(self, api):
        self.api = api
        self.lock = threading.Lock()
        self._history = {}  # pid -> {m_epoch: {"status": ..., "lat": ...}}
        self._running = True

    def start(self):
        t = threading.Thread(target=self._run_sampler, daemon=True, name="net-tracker")
        t.start()

    def _sample_once(self):
        try:
            now = time.time()
            now_m = int(now // 60) * 60
            e = self.api.e
            with e.lock:
                stas = list(e.stations.values())

            with self.lock:
                for pid, name, inst, pattern in PROVIDERS:
                    matching_srcs = [s for s in self.api.sources if re.search(pattern, getattr(
```

```
      s, "server", "")) or re.search(pattern, getattr(s, "label", ""))]
                    if not matching_srcs:
                        continue
                    last_data = max((s.last_data for s in matching_srcs), default=0.0)

                    st_matches = [s for s in stas if any(re.search(pattern, getattr(s, "fuente"
      , "")) or re.search(pattern, getattr(src, "server", "")) for src in matching_srcs)]
                    lats = [now - s.data_end for s in st_matches if s.data_end and 0 <= now - s
      .data_end < 3600]
                    lat = round(statistics.median(lats), 1) if lats else None

                    if last_data and now - last_data < 120:
                        st = "retraso" if (lat is not None and lat > 35) else "ok"
                    else:
                        st = "inactivo" if now - e.started < 90 else "caida"

                    if pid not in self._history:
                        self._history[pid] = {}
                    self._history[pid][now_m] = {"status": st, "lat": lat}
                    cutoff = now_m - 4200
                    self._history[pid] = {k: v for k, v in self._history[pid].items() if k >= c
      utoff}
          except Exception:
              pass

      def _run_sampler(self):
          self._sample_once()
          while self._running:
              time.sleep(15)
              self._sample_once()

      def get_networks(self):
          now = time.time()
          now_m = int(now // 60) * 60
          e = self.api.e
          started = getattr(e, "started", now - 3600)

          with e.lock:
              stas = list(e.stations.values())

          result = []
          with self.lock:
              for pid, name, inst, pattern in PROVIDERS:
                  matching_srcs = [s for s in self.api.sources if re.search(pattern, getattr(s, "
      server", "")) or re.search(pattern, getattr(s, "label", ""))]
                  if not matching_srcs:
                      continue
                  last_data = max((s.last_data for s in matching_srcs), default=0.0)
                  packets = sum(s.packets for s in matching_srcs)
                  cuts = tuple(t for s in matching_srcs for t in getattr(s, "cuts", []) if now -
      t < 3600)
                  server_label = matching_srcs[0].server

                  st_matches = [s for s in stas if any(re.search(pattern, getattr(s, "fuente", ""
      )) or re.search(pattern, getattr(src, "server", "")) for src in matching_srcs)]
                  n_tot = len(st_matches)
                  n_on = sum(1 for s in st_matches if s.last_data and now - s.last_data < 120)
                  lats = [now - s.data_end for s in st_matches if s.data_end and 0 <= now - s.dat
      a_end < 3600]
                  lat = round(statistics.median(lats), 1) if lats else None

                  if last_data and now - last_data < 120:
                      cur_st = "retraso" if (lat is not None and lat > 35) else "ok"
                  else:
                      cur_st = "inactivo" if now - started < 90 else "caida"

                  pid_hist = self._history.get(pid, {})
                  bars = []
                  active_min = 0
                  ok_min = 0

                  for i in range(60):
                      m_epoch = now_m - (59 - i) * 60
                      mins_ago = 59 - i

                      if m_epoch in pid_hist:
                          st = pid_hist[m_epoch]["status"]
                          bar_lat = pid_hist[m_epoch]["lat"]
                      else:
                          if m_epoch < started - 30:
                              st = "inactivo"
                              bar_lat = None
                          else:
                              had_cut = any(abs(t - m_epoch) < 60 for t in cuts)
                              if had_cut:
                                  st = "caida"
                                  bar_lat = None
                              else:
                                  st = cur_st
                                  bar_lat = lat

                      if st != "inactivo":
                          active_min += 1
                          if st == "ok":
                              ok_min += 1
```

```
                        elif st == "retraso":
                            ok_min += 0.8

                    time_lbl = "Ahora (este minuto)" if mins_ago == 0 else f"Hace {mins_ago} mi
n"
                    st_lbl = {
                        "ok": f"Conectado · latencia {bar_lat} s" if bar_lat is not None else "
Conectado · operativo",
                        "retraso": f"Retraso en datos · latencia {bar_lat} s" if bar_lat is not
 None else "Retraso en datos",
                        "caida": "Interrupción / sin datos",
                        "inactivo": "Sin datos (previo al arranque)"
                    }.get(st, st)

                    bars.append({
                        "m": mins_ago,
                        "status": st,
                        "lat": bar_lat,
                        "tip": f"{time_lbl}: {st_lbl}"
                    })

                uptime_pct = round((ok_min / active_min * 100), 1) if active_min > 0 else 100.0

                result.append({
                    "id": pid,
                    "name": name,
                    "institucion": inst,
                    "server": server_label,
                    "status": cur_st,
                    "uptime_pct": uptime_pct,
                    "latency": lat,
                    "packets": packets,
                    "stations_total": n_tot,
                    "stations_online": n_on,
                    "history": bars
                })

        return result


class Api:
    def __init__(self, engine, alerts, sources, web_dir, usgs_ref):
        self.e, self.alerts, self.sources, self.web_dir, self.usgs_ref = engine, alerts, source
s, web_dir, usgs_ref
        self.ctx = None          # contexto para el monitoreo del estado del sistema (lo llena
run.py)
        self.on_restart = None   # función de reinicio (la pone run.py)
        self.net_tracker = NetworkTracker(self)
        self.net_tracker.start()

    def get_networks_status(self):
        return self.net_tracker.get_networks()

    def event_summary(self, ev):
        e = self.e
        return {k: ev.get(k) for k in ("id", "org", "lat", "lon", "z", "errH", "errMin", "errAz
", "errZ", "errOpen", "rms", "gap",
                                       "nP", "nS", "ml", "mlSd", "nml", "mlComp", "mwp", "mwpN
", "mwpSd", "mb", "mbN", "mbSd", "mbR", "mbRN", "mbRSd", "mB", "mBN", "mBSd", "ms", "msN", "msSd", "mw", "mwSd", "mwParts", "tsunami", "depthPhases", "zPonly", "errZPonly
                                       "live", "manual", "match", "model", "region", "dmin",
                                       "strike", "dip", "rake", "strike2", "dip2", "rake2",
                                       "focal_regime", "focal_quality", "focal_misfit", "focal
_source",
                                       "p_axis", "t_axis", "focal_stations")} | {
            "where": where_is(ev["lat"], ev["lon"]), "quality": e.quality(ev), "magStr": e.mag_
str(ev),
            "zone": (zone_of(e.cfg.get("alertas", {}).get("zonas") or [], ev["lat"], ev["lon"])
 or {}).get("nombre")}

    def status(self):
        e = self.e
        now = time.time()
        with e.lock:
            on = [s for s in e.stations.values() if s.last_data and now - s.last_data < 120]
            srcs = [{"server": s.server, "ok": now - s.last_data < 120, "packets": s.packets} f
or s in self.sources]

            return {"now": now, "started": e.started, "model": e.tt.name,
                    "stations": len(e.stations),
                    "online": len(on),
                    "threeComp": sum(1 for s in on if len([c for c in s.comps.values() if now -
 c.last < 120]) == 3),
                    "triggered": sum(1 for s in on if s.last_pick and now - s.last_pick <= 5.0)
,
                    "trig5": sum(1 for s in e.stations.values() for t in s.trig_times if now -
t < 300),
                    "sources": srcs,
                    "ai": e.ai is not None, "stats": e.stats(),
                    "net": self.net_quality(now, on)}

    def net_quality(self, now, on):
        """v2.1.0: calidad de la conexión a internet para el monitor (punto verde / naranja / r
ojo en el panel).
        Se evalúa con lo que el monitor recibe: retraso de los datos (latencia mediana de las e
staciones), estaciones
```

```
253         en línea frente al máximo de la sesión, servidores caídos, cortes de conexión en la últ
    ima hora y paquetes
254         descartados (PC saturada). Llamar con el candado del motor tomado."""
255         e = self.e
256         lat = sorted(now - s.data_end for s in on if s.data_end and 0 <= now - s.data_end < 360
    0)
257         med = lat[len(lat) // 2] if lat else None
258         hist = getattr(self, "_net_hist", [])
259         hist = [h for h in hist if now - h[0] < 6 * 3600] + [(now, len(on))]
260         self._net_hist = hist
261         peak = max(h[1] for h in hist) or 1
262         ratio = len(on) / peak
263         tot = sum(s.packets for s in self.sources)
264         qtot = sum(getattr(s, "queued", s.packets) for s in self.sources)
265         prev = getattr(self, "_net_pk", None)
266         if prev is None or now - prev[0] >= 30:
267             if prev is not None and now > prev[0]:
268                 self._net_rate = (tot - prev[1]) / (now - prev[0])
269                 self._net_qrate = (qtot - prev[2]) / (now - prev[0]) if len(prev) > 2 else self
    ._net_rate
270             self._net_pk = (now, tot, qtot)
271         rate = getattr(self, "_net_rate", None)
272         qrate = getattr(self, "_net_qrate", rate)
273         kbps = rate * 520 * 8 / 1000 if rate is not None else None
274         down = [s.server for s in self.sources if s.packets > 0 and now - s.last_data > 120]
275         cuts = sum(len([t for t in getattr(s, "cuts", []) if now - t < 3600]) for s in self.sou
    rces if s.packets > 0)
276         drops = any(now - getattr(s, "drop_t", 0) < 60 for s in self.sources)
277         # v2.1.0 ajuste: el retraso que se mide incluye la espera en la cola de procesamiento d
    e la PC; se separa
278         # para no culpar al internet cuando la que va retrasada es la PC
279         qn = 0
280         try:
281             qn = (self.ctx or {}).get("queue").qsize()
282         except Exception:
283             qn = 0
284         proc = qn / qrate if qrate and qrate > 0 else 0.0
285         med_raw = med
286         if med is not None:
287             med = max(0.0, med - proc)
288         warm = now - e.started < 90          # v2.1.0 ajuste: gris solo el primer minuto y medi
    o
289         early = now - e.started < 300        # el % de estaciones se evalúa después de 5 min (a
    ún se están conectando)
290         red, orange = [], []
291         if med is not None and med > 60:
292             red.append(f"datos con {med:.0f} s de retraso")
293         elif med is not None and med > 25:
294             orange.append(f"datos con {med:.0f} s de retraso")
295         if not early:
296             if ratio < 0.7:
297                 red.append(f"solo {len(on)} de {peak} estaciones en línea ({100 * ratio:.0f} %)
    ")
298             elif ratio < 0.9:
299                 orange.append(f"{len(on)} de {peak} estaciones en línea ({100 * ratio:.0f} %)")
300         main = self.sources[0].server if self.sources else None
301         if main in down or len(down) >= 3:
302             red.append("sin datos de " + ", ".join(down))
303         elif down:
304             orange.append("sin datos de " + ", ".join(down))
305         if cuts >= 10:
306             red.append(f"{cuts} cortes de conexión en la última hora")
307         elif cuts >= 3:
308             orange.append(f"{cuts} cortes de conexión en la última hora")
309         pc = "ok"
310         if qn >= 2700 or proc > 5.0:
311             pc = "rojo"
312         elif qn > 1500 or proc > 2.5 or (drops and qn > 300):
313             pc = "naranja"
314         level = "rojo" if red else "naranja" if orange else ("gris" if warm else "verde")
315         txt = {"verde": "Internet óptimo para el monitor", "naranja": "Internet regular: el mon
    itor funciona con retrasos o pérdidas",
316                "rojo": "Internet deficiente: el monitor puede perder sismos", "gris": "Midiendo
     la conexión (primer minuto y medio tras arrancar)"}[level]
317         pc_txt = {"ok": "la PC procesa al día",
318                   "naranja": f"la PC va retrasada: {qn} paquetes en espera (≈ {proc:.0f} s)",
319                   "rojo": f"la PC no alcanza a procesar: {qn} paquetes en espera (≈ {proc:.0f}
    s); alertas con retraso"}[pc]
320         return {"nivel": level, "texto": txt, "problemas": red + orange, "latencia": med, "late
    nciaTotal": med_raw,
321                 "kbps": kbps, "enLinea": len(on), "maximo": peak, "cortes": cuts, "caidos": dow
    n,
322                 "pc": pc, "pcTexto": pc_txt, "cola": qn, "colaSeg": proc}
323
324     def stations(self):
325         e = self.e
326         now = time.time()
327         with e.lock:
328             rows = []
329             for s in e.stations.values():
330                 on = bool(s.last_data and now - s.last_data < 120)
331                 tab = e.corr.get(e.tt.model, {})
332                 c = tab.get(s.key + "|T") or tab.get(s.key + "|R")
333                 w_custom = getattr(s, "where_custom", "")
334                 w_place = where_is(s.lat, s.lon)
```

```
                where_txt = f"{w_custom} · {w_place}" if (w_custom and w_custom not in w_place) else (w_custom or w_place)
                is_red = bool(s.last_pick and (now - s.last_pick <= 5.0))
                rows.append({"key": s.key, "code": s.code, "net": s.net, "lat": s.lat, "lon": s.lon, "where": where_txt,
                             "band": s.band, "comps": "".join(k for k in ("Z", "N", "E") if k in s.comps and now - s.comps[k].last < 120),
                             "online": on, "trig": is_red,
                             "last_trig": s.last_pick if (s.last_pick and now - s.last_pick < 65) else None,
                             "latency": (now - s.data_end) if s.data_end else None,
                             "packets": s.packets, "trig1h": sum(1 for t in s.trig_times if now - t < 3600),
                             "fuente": getattr(s, "fuente", ""),
                             "corr": (c[0] / c[1]) if c and c[1] >= 2 else None, "ncorr": c[1] if c else 0})
            rows.sort(key=lambda r: (not r["online"], r["code"]))
            return rows

    def live_triggers(self):
        """Micro-endpoint de alto rendimiento para propagación térmica de ondas en el mapa.
        Solo reporta estaciones con picks confirmados por Maeda AIC (evita ruidos STA/LTA rechazados)."""
        now = time.time()
        out = []
        with self.e.lock:
            # 1. Estaciones con pick automático confirmado por AIC
            for s in self.e.stations.values():
                t_pick = s.last_pick
                if t_pick and (now - t_pick < 65):
                    out.append({"key": s.key, "t": t_pick, "trig": True})

            # 2. Picks de eventos activos recientes (por si hubo picado dirigido de P o S)
            seen_keys = {item["key"] for item in out}
            for ev in getattr(self.e, "events", [])[:50]:
                if abs(now - ev.get("org", 0)) > 3600:
                    continue
                for p in ev.get("picks", []):
                    pk = p.get("key")
                    pt = p.get("t")
                    if pk and pt and pk not in seen_keys and (now - pt < 65):
                        seen_keys.add(pk)
                        out.append({"key": pk, "t": pt, "trig": True})

        return {"now": now, "triggers": out}

    def events(self, n=100):
        with self.e.lock:
            mmin = self.e.cfg.get("alertas", {}).get("magnitud_minima", 4.0)
            valid = [ev for ev in self.e.events if ev.get("mag") is None or ev["mag"] >= mmin]
            return [self.event_summary(ev) for ev in valid[:n]]

    def event(self, ev_id):
        e = self.e
        with e.lock:
            ev = next((x for x in e.events if x["id"] == ev_id), None)
            if ev is None:
                return None
            out = self.event_summary(ev)
            out["picks"] = sorted([{k: v for k, v in p.items() if not k.startswith("_")} | {
                "dist": float(dist_km(ev["lat"], ev["lon"], p["lat"], p["lon"]))} for p in ev["picks"]], key=lambda p: p["t"])
            out["impact"] = e.poi_impact(ev)
            return out

    def section(self, ev_id, f1, f2):
        e = self.e
        with e.lock:
            ev = next((x for x in e.events if x["id"] == ev_id), None)
            if ev is None:
                return None
            old = time.time() - ev.get("created", 0) > 20 * 60 or not ev.get("live")
            traces = (e.storage.load_waveforms(ev_id) if (old and e.storage) else []) or (e.section_traces(ev, f1, f2) if ev.get("live") else [])
            # Verificación de integridad: si las trazas guardadas están truncadas (t0 relativo > 10 s),
            # o incompletas respecto al origen del sismo, reconstruir desde el archivo continuo
            if traces and any((tr.get("t0", 0) - ev["org"]) > 10.0 for tr in traces[:5]):
                traces = []
        if not traces and e.storage:
            # v0.6.3: si no se guardaron, se reconstruyen del archivo miniSEED continuo (2 días) y se guardan.
            # Se lee FUERA del candado del motor para no detener el procesamiento en vivo.
            try:
                from .storage import archive_traces
                evc = {"org": ev["org"], "picks": [dict(p) for p in ev.get("picks", [])]}
                prefs = {k: type("P", (), {"pref": getattr(st, "pref", None)})() for k, st in e.stations.items()}
                traces = archive_traces(os.path.join(e.storage.folder, "archivo"), evc, prefs, f1, f2)
                if not traces:
                    e.log(f"Sección de registro {ev_id}: sin datos en el archivo continuo para sus estaciones")
                elif (f1, f2) == (0.5, 4.0):
                    e.storage.save_waveforms(ev_id, traces)
```

```
            except Exception as ex:
                e.log(f"Sección de registro {ev_id}: no se pudo leer el archivo continuo ({ex})
")
                traces = []
        with e.lock:
            out = []
            for tr in traces:
                s = e.stations.get(tr["key"])
                ref = s or next((p for p in ev["picks"] if p["key"] == tr["key"]), None)
                if ref is None:
                    continue
                lat, lon = (s.lat, s.lon) if s else (ref["lat"], ref["lon"])
                d = float(dist_km(ev["lat"], ev["lon"], lat, lon))
                data = np.asarray(tr["data"], float)
                data = data / (np.abs(data).max() or 1)
                out.append({"key": tr["key"], "code": tr["key"].replace("_", "."), "dist": d,
                            "tp": float(e.tt.tt("P", d, ev["z"])) + e.corr_of(tr["key"], "P", d
),
                            "ts": (float(e.tt.tt("S", d, ev["z"])) + e.corr_of(tr["key"], "S",
d)) if d < 2500 else None,
                            "env": _envelope(tr["t0"] - ev["org"], tr["fs"], data, 2400)})
            out.sort(key=lambda t: t["dist"])
            picks = [{"key": p["key"], "phase": p["phase"], "t": p["t"] - ev["org"], "manual":
bool(p.get("manual")), "ai": bool(p.get("ai"))} for p in ev["picks"]]
            return {"org": ev["org"], "traces": out, "picks": picks}

    def live(self, keys, comps, seconds, f1=0.5, f2=8.0):
        """Trazas en vivo para el visor continuo: tiempo absoluto de inicio, paso y envolvente
mín./máx."""
        e = self.e
        out = []
        now = time.time()
        to_process = []
        picks_out = []
        with e.lock:
            for key in keys:
                s = e.stations.get(key)
                if not s:
                    continue
                for comp in comps:
                    c = s.comps.get(comp)
                    if not c or c.next_t is None or now - c.last > 300:
                        continue
                    got = c.ring.get(c.next_t - seconds - 30, c.next_t)
                    if got is None:
                        continue
                    to_process.append((key, s.code, comp, c.cha, c.next_t, c.fs, got[0], got[1]
))

            # Picks automáticos y asociados en la ventana temporal para las estaciones consulta
das (estilo SeisComP)
            keys_set = set(keys) | {k.replace("_", ".") for k in keys if k} | {k.replace(".", "
_") for k in keys if k}
            codes_set = {k.replace("_", ".").split(".")[-1] for k in keys if k}
            t_min = now - seconds - 60
            t_max = now + 10
            seen_picks = set()

            # 1. Picks activos en el búfer de detección del motor (e.picks)
            for p in getattr(e, "picks", []):
                pk = p.get("key", "")
                pc = p.get("code") or (pk.replace("_", ".").split(".")[-1] if pk else "")
                pk_alt = pk.replace("_", ".") if pk else ""
                pt = p.get("t", 0)
                phase = p.get("phase", "P")
                if (pk in keys_set or pk_alt in keys_set or pc in codes_set) and t_min <= pt <=
 t_max:
                    dedup = (pk or pc, round(pt, 1), phase)
                    if dedup not in seen_picks:
                        seen_picks.add(dedup)
                        picks_out.append({
                            "key": pk,
                            "code": pc,
                            "phase": phase,
                            "t": pt,
                            "manual": bool(p.get("manual", False)),
                            "snr": round(float(p.get("snr", 0)), 1) if p.get("snr") is not None
 else None,
                            "ev": p.get("_ev")
                        })

            # 2. Picks asociados en los eventos recientes
            for ev in getattr(e, "events", [])[:80]:
                ev_org = ev.get("org", 0)
                if abs(now - ev_org) > 7200:
                    continue
                for p in ev.get("picks", []):
                    pk = p.get("key", "")
                    pc = p.get("code") or (pk.replace("_", ".").split(".")[-1] if pk else "")
                    pk_alt = pk.replace("_", ".") if pk else ""
                    pt = p.get("t", 0)
                    phase = p.get("phase", "P")
                    if (pk in keys_set or pk_alt in keys_set or pc in codes_set) and t_min <= p
t <= t_max:
                        dedup = (pk or pc, round(pt, 1), phase)
```

```
                        if dedup not in seen_picks:
                            seen_picks.add(dedup)
                            picks_out.append({
                                "key": pk,
                                "code": pc,
                                "phase": phase,
                                "t": pt,
                                "manual": bool(p.get("manual", False)),
                                "snr": round(float(p.get("snr", 0)), 1) if p.get("snr") is not None else None,
                                "ev": ev.get("id"),
                                "mag": ev.get("mag")
                            })

        for key, code, comp, cha, next_t, fs, t0, data in to_process:
            f = bandpass(data, fs, f1, f2)
            skip = max(0, int(len(f) - seconds * fs))
            f = f[skip:]
            # resolución alta (estilo SWARM): hasta 3200 puntos mín./máx.; con ventanas cortas, muestra por muestra
            env = _envelope(0.0, fs, f, 3200)
            if env is None:
                continue
            env["t0"] = t0 + skip / fs
            out.append({"key": key, "code": code, "comp": comp, "cha": cha, "end": next_t,
                        "latency": now - next_t, "env": env})
        return {"now": now, "traces": out, "picks": picks_out}

    def spectrogram(self, key, comp, seconds, fmax=None):
        """Espectrograma de una traza (estilo SWARM): potencia en dB cuantizada a 0-255 por columna de tiempo."""
        e = self.e
        with e.lock:
            s = e.stations.get(key)
            c = s.comps.get(comp) if s else None
            if not c or c.next_t is None:
                return {"ok": False}
            got = c.ring.get(c.next_t - seconds - 10, c.next_t)
            if got is None:
                return {"ok": False}
            t0, data = got
            fs, code, cha = c.fs, s.code, c.cha
        x = np.asarray(data, float)
        if len(x) < fs * 20:
            return {"ok": False}
        x = _sig.detrend(x)
        nper = int(2 ** np.round(np.log2(max(64, fs * 5.12))))          # ~5 s por ventana (buena resolución en frecuencia)
        hop = max(1, nper // 8)                                          # 87.5 % de traslape (columnas finas)
        f, tt, sxx = _sig.spectrogram(x, fs=fs, window="hann", nperseg=nper, noverlap=nper - hop,
                                      detrend="constant", scaling="density", mode="psd")
        fmax = min(fs / 2, fmax or fs / 2)
        keep = (f > 0) & (f <= fmax + 1e-9)
        f, sxx = f[keep], sxx[keep]
        db = 10 * np.log10(np.maximum(sxx, 1e-30))
        lo, hi = np.percentile(db, 2), np.percentile(db, 99.9) + 3
        hi = max(hi, lo + 20)                                            # al menos 20 dB de rango
        q = np.clip((db - lo) / (hi - lo) * 255, 0, 255).astype(np.uint8)
        q = q[::-1]                                                      # fila 0 = frecuencia más alta
        return {"ok": True, "code": code, "cha": cha, "t0": float(t0 + tt[0] - hop / fs / 2), "dt": hop / fs,
                "ncol": int(q.shape[1]), "nrow": int(q.shape[0]), "fmax": float(f[-1]), "fmin": float(f[0]),
                "dbmin": float(lo), "dbmax": float(hi), "img": base64.b64encode(q.T.tobytes()).decode("ascii")}


def make_handler(api):
    class H(BaseHTTPRequestHandler):
        def log_message(self, *a):
            pass

        def _send(self, code, body, ctype="application/json; charset=utf-8"):
            try:
                if not isinstance(body, (bytes, bytearray)):
                    body = json.dumps(body, default=_json_default, ensure_ascii=False).encode("utf-8")
                self.send_response(code)
                self.send_header("Content-Type", ctype)
                self.send_header("Cache-Control", "no-store")
                self.send_header("Content-Length", str(len(body)))
                self.end_headers()
                self.wfile.write(body)
            except (BrokenPipeError, ConnectionResetError):
                pass

        def do_GET(self):
            u = urllib.parse.urlparse(self.path)
            q = dict(urllib.parse.parse_qsl(u.query))
            p = u.path
            try:
```

```
                if p in ("/", "/index.html"):
                    with open(os.path.join(api.web_dir, "panel.html"), "rb") as f:
                        return self._send(200, f.read(), "text/html; charset=utf-8")
                if p in ("/traceview", "/traceview.html"):
                    with open(os.path.join(api.web_dir, "traceview.html"), "rb") as f:
                        return self._send(200, f.read(), "text/html; charset=utf-8")
                if p == "/api/probar_seedlink":
                    from .slprobe import probe
                    hp = q.get("h", "")
                    host, _, port = hp.partition(":")
                    if not host or not (port or "18000").isdigit():
                        return self._send(400, {"error": "use ?h=servidor:puerto"})
                    if q.get("cmd"):
                        from .slprobe import dialog
                        return self._send(200, {"dialogo": dialog(host, int(port or 18000), q["
cmd"].split("|"))})
                    r = probe(host, int(port or 18000))
                    r.pop("claves", None)
                    return self._send(200, r)
                if p == "/api/estado":
                    return self._send(200, api.status())
                if p in ("/api/disparos_vivos", "/api/triggers"):
                    return self._send(200, api.live_triggers())
                if p == "/api/estaciones":
                    return self._send(200, api.stations())
                if p == "/api/estaciones_ss":
                    saved = api.e.storage.load_json("estaciones_ss.json", {}) if api.e.storage
else {}
                    return self._send(200, {"ok": True, "estaciones": saved, "datalink_port": 1
6000, "seedlink_port": 18000})
                if p in ("/api/admin/purgar_estaciones_ss", "/api/purgar_estaciones_ss"):
                    days = float(q.get("dias") or q.get("days") or 7.0)
                    purged = api.e.purge_inactive_ss_stations(max_days=days)
                    return self._send(200, {"ok": True, "purgadas": purged, "total": len(purged
), "dias_corte": days})
                if p in ("/api/heartbeat", "/api/recibir_estacion"):
                    if any(k in q for k in ("station_id", "sta", "id", "codigo", "code")):
                        st = api.e.register_ss_station(q, persist=True)
                        return self._send(200, {"ok": True, "status": "ok", "key": st.key if st
 else None, "code": st.code if st else None,
                                                "lat": st.lat if st else None, "lon": st.lon if
 st else None,
                                                "datalink_port": 16000, "seedlink_port": 18000}
)
                    return self._send(200, {"ok": True, "status": "ok", "datalink_port": 16000,
 "seedlink_port": 18000})
                if p == "/api/sismos":
                    return self._send(200, api.events(int(q.get("n", 100))))
                if p == "/api/salud":
                    if api.ctx is None:
                        return self._send(503, {"error": "monitoreo no disponible"})
                    from .health import report
                    rep = report(dict(api.ctx, sources=api.sources))
                    try:
                        rep["networks"] = api.get_networks_status()
                    except Exception:
                        rep["networks"] = []
                    return self._send(200, rep)
                if p == "/api/redes":
                    try:
                        return self._send(200, api.get_networks_status())
                    except Exception as ex:
                        return self._send(500, {"error": str(ex)})
                if p == "/api/zonas":
                    a = api.e.cfg.get("alertas", {})
                    return self._send(200, {"zonas": a.get("zonas") or [], "alertar_tsunami": a
.get("alertar_tsunami", True),
                                            "magnitud_minima": a.get("magnitud_minima")})
                if p == "/api/calibracion":
                    with api.e.lock:
                        return self._send(200, api.e.calib.summary())
                if p == "/api/usgs":
                    return self._send(200, api.usgs_ref())
                if p.startswith("/api/sismo/") and p.endswith("/mecanismo_focal"):
                    ev_id = p.split("/")[3]
                    with api.e.lock:
                        ev = next((x for x in api.e.events if x["id"] == ev_id), None)
                    if not ev:
                        return self._send(404, {"error": "sismo no encontrado"})
                    from . import focal
                    sol = focal.evaluate_event_focal(api.e, ev)
                    if sol:
                        with api.e.lock:
                            api.e.dirty.add(ev["id"])
                            if api.e.storage:
                                api.e.storage.save_events([ev])
                        return self._send(200, {"ok": True, "mecanismo": sol, "evento": api.eve
nt_summary(ev)})
                    return self._send(200, {"ok": False, "mensaje": "Insuficientes polaridades
para calcular mecanismo", "mecanismo": None, "evento": api.event_summary(ev)})
                if p.startswith("/api/sismo/"):
                    r = api.event(p.split("/")[-1])
                    return self._send(200 if r else 404, r or {"error": "no encontrado"})
                if p.startswith("/api/seccion/"):
                    r = api.section(p.split("/")[-1], float(q.get("f1", 0.5)), float(q.get("f2"
```

```
, 4)))
                    return self._send(200 if r else 404, r or {"error": "no encontrado"})
                if p == "/api/vivo":
                    keys = [k for k in q.get("keys", q.get("key", "")).split(",") if k][:32]
                    comps = [c for c in q.get("comp", "Z") if c in "ZNE"] or ["Z"]
                    r = api.live(keys, comps, min(3600, float(q.get("s", 300))), float(q.get("f
1", 0.5)), float(q.get("f2", 8)))
                    return self._send(200, r)
                if p == "/api/placas":
                    from . import placas
                    folder = api.e.storage.folder if api.e.storage else None
                    data = placas.load(folder, api.e.log) if folder else None
                    return self._send(200 if data else 503, data or {"error": "límites de placa
s no disponibles"})
                if p == "/api/espectrograma":
                    r = api.spectrogram(q.get("key", ""), (q.get("comp", "Z") or "Z")[:1], min(
3600, float(q.get("s", 300))),
                                        float(q["fmax"]) if q.get("fmax") else None)
                    return self._send(200, r)
                if p.startswith("/api/ticket/"):
                    # ticket técnico: /api/ticket/<id>.txt | .json | .xml (QuakeML)
                    name = p.split("/")[-1]
                    ev_id, _, ext = name.partition(".")
                    ext = ext or "txt"
                    if ext not in ("txt", "json", "xml"):
                        return self._send(400, {"error": "formato no válido (txt, json o xml)"}
)
                    with api.e.lock:
                        ev = next((x for x in api.e.events if x["id"] == ev_id), None)
                        if ev is None:
                            return self._send(404, {"error": "no encontrado"})
                        t = tk.build(api.e, ev)
                        stem = tk.file_stem(ev)
                    text = tk.to_text(t) if ext == "txt" else tk.to_json(t) if ext == "json" el
se tk.to_quakeml(t)
                    data = text.encode("utf-8")
                    ctype = {"txt": "text/plain", "json": "application/json", "xml": "applicati
on/xml"}[ext] + "; charset=utf-8"
                    self.send_response(200)
                    self.send_header("Content-Type", ctype)
                    self.send_header("Cache-Control", "no-store")
                    if q.get("ver") != "1":
                        self.send_header("Content-Disposition", f'attachment; filename="{stem}.
{"quakeml.xml" if ext == "xml" else ext}"')
                    self.send_header("Content-Length", str(len(data)))
                    self.end_headers()
                    self.wfile.write(data)
                    return
                if p in ("/api/reporte_usgs.pdf", "/api/reporte_usgs.csv"):
                    from . import usgsreport
                    folder = api.e.storage.folder if api.e.storage else None
                    data = usgsreport.collect(api.e, folder, getattr(api.e, "log", None))
                    stamp = time.strftime("%Y%m%d_%H%M", time.gmtime(time.time() - 6 * 3600))
                    if p.endswith(".csv"):
                        body = usgsreport.to_csv(data).encode("utf-8")
                        ctype, name = "text/csv; charset=utf-8", f"monitor_vs_usgs_{stamp}.csv"
                    else:
                        path = os.path.join(tempfile.gettempdir(), f"monitor_vs_usgs_{stamp}.pd
f")
                        usgsreport.make_pdf(data, path)
                        with open(path, "rb") as f:
                            body = f.read()
                        ctype, name = "application/pdf", f"monitor_vs_usgs_{stamp}.pdf"
                    if folder:
                        try:            # copia en datos\reportes
                            os.makedirs(os.path.join(folder, "reportes"), exist_ok=True)
                            with open(os.path.join(folder, "reportes", name), "wb") as f:
                                f.write(body)
                        except OSError:
                            pass
                    self.send_response(200)
                    self.send_header("Content-Type", ctype)
                    self.send_header("Content-Disposition", f'attachment; filename="{name}"')
                    self.send_header("Content-Length", str(len(body)))
                    self.end_headers()
                    self.wfile.write(body)
                    return
                if p in ("/api/reporte_estaciones.pdf", "/api/reporte_estaciones.csv"):
                    from . import estreport
                    data = estreport.collect(api.e, api.sources)
                    stamp = time.strftime("%Y%m%d_%H%M", time.gmtime(time.time() - 6 * 3600))
                    if p.endswith(".csv"):
                        body = estreport.to_csv(data).encode("utf-8")
                        ctype, name = "text/csv; charset=utf-8", f"estaciones_{stamp}.csv"
                    else:
                        folder = api.e.storage.folder if api.e.storage else None
                        path = os.path.join(tempfile.gettempdir(), f"estaciones_{stamp}.pdf")
                        estreport.make_pdf(data, path, folder)
                        with open(path, "rb") as f:
                            body = f.read()
                        ctype, name = "application/pdf", f"reporte_estaciones_{stamp}.pdf"
                        if folder:
                            try:            # copia en datos\reportes
                                os.makedirs(os.path.join(folder, "reportes"), exist_ok=True)
                                with open(os.path.join(folder, "reportes", name), "wb") as f:
```

```
                                f.write(body)
                        except OSError:
                            pass
                self.send_response(200)
                self.send_header("Content-Type", ctype)
                self.send_header("Content-Disposition", f'attachment; filename="{name}"')
                self.send_header("Content-Length", str(len(body)))
                self.end_headers()
                self.wfile.write(body)
                return
            if p.startswith("/api/reporte/"):
                ev_id = p.split("/")[-1].replace(".pdf", "")
                with api.e.lock:
                    ev = next((x for x in api.e.events if x["id"] == ev_id), None)
                    if ev is None:
                        return self._send(404, {"error": "no encontrado"})
                    path = os.path.join(tempfile.gettempdir(), f"reporte_{ev_id}.pdf")
                    make_pdf(api.e, ev, path)
                with open(path, "rb") as f:
                    data = f.read()
                self.send_response(200)
                self.send_header("Content-Type", "application/pdf")
                self.send_header("Content-Disposition", f'attachment; filename="reporte_{ev_id}.pdf"')
                self.send_header("Content-Length", str(len(data)))
                self.end_headers()
                self.wfile.write(data)
                return
            if p.startswith("/api/descargar"):
                doc = q.get("archivo", "").lower().strip()
                allowed = {
                    "manual_usuario": ("Manual_Monitor_Mundial_v2.3.1.pdf", "application/pdf"),
                    "manual_tecnico": ("Manual_Tecnico_Ficha_Tecnica_Monitor_Mundial_v2.3.1.pdf", "application/pdf"),
                    "manual_sidesis": ("Manual_Red_SIDESIS_v2.3.1.pdf", "application/pdf"),
                    "manual_usuario_v230": ("Manual_Monitor_Mundial_v2.3.0.pdf", "application/pdf"),
                    "manual_tecnico_v230": ("Manual_Tecnico_Ficha_Tecnica_Monitor_Mundial_v2.3.0.pdf", "application/pdf"),
                    "manual_sidesis_v230": ("Manual_Red_SIDESIS_v2.3.0.pdf", "application/pdf"),
                    "cambios": ("CAMBIOS.txt", "text/plain; charset=utf-8"),
                    "mejoras": ("MEJORAS_v2.3.1.txt", "text/plain; charset=utf-8"),
                    "mejoras_v230": ("MEJORAS_v2.3.0.txt", "text/plain; charset=utf-8"),
                    "leeme_aws": ("LEEME_AWS.txt", "text/plain; charset=utf-8"),
                    "leeme": ("LEEME.txt", "text/plain; charset=utf-8"),
                }
                if doc not in allowed:
                    return self._send(400, {"error": f"Archivo '{doc}' no permitido para descarga."})
                fname, ctype = allowed[doc]
                base_prog = os.path.dirname(api.web_dir)
                parent_dir = os.path.dirname(base_prog)
                candidates = [
                    os.path.join(parent_dir, fname),
                    os.path.join(base_prog, fname),
                    os.path.join(api.web_dir, fname),
                    os.path.join("/opt/monitor_mundial", fname),
                    os.path.join(r"c:\Users\Carlos\OneDrive\Documentos\Agencia SIMMSA\Monitor_Mundial_Linux_v2.1.0", fname),
                    os.path.join(r"c:\Users\Carlos\OneDrive\Documentos\Agencia SIMMSA\Monitor_Mundial_Linux_v2.1.0\programa", fname),
                ]
                found = next((cp for cp in candidates if os.path.isfile(cp)), None)
                if not found:
                    return self._send(404, {"error": f"El archivo {fname} no se encuentra en el servidor."})
                with open(found, "rb") as f:
                    content = f.read()
                self.send_response(200)
                self.send_header("Content-Type", ctype)
                self.send_header("Content-Disposition", f'attachment; filename="{fname}"')
                self.send_header("Content-Length", str(len(content)))
                self.end_headers()
                self.wfile.write(content)
                return
            self._send(404, {"error": "ruta desconocida"})
        except (BrokenPipeError, ConnectionResetError):
            pass
        except Exception as ex:
            try:
                self._send(500, {"error": str(ex)})
            except Exception:
                pass

    def do_POST(self):
        p = urllib.parse.urlparse(self.path).path
        try:
            n = int(self.headers.get("Content-Length", 0))
            body = json.loads(self.rfile.read(n) or b"{}")
            if p in ("/api/heartbeat", "/api/recibir_estacion", "/api/estaciones_ss"):
                st = api.e.register_ss_station(body, persist=True)
                return self._send(200, {"ok": True, "status": "ok", "key": st.key if st else None, "code": st.code if st else None,
```

```
                                        "lat": st.lat if st else None, "lon": st.lon if st else None,
                                        "datalink_port": 16000, "seedlink_port": 18000})
                if p.startswith("/api/sismo/") and p.endswith("/mecanismo_focal"):
                    ev_id = p.split("/")[3]
                    with api.e.lock:
                        ev = next((x for x in api.e.events if x["id"] == ev_id), None)
                    if not ev:
                        return self._send(404, {"error": "sismo no encontrado"})
                    from . import focal
                    s1 = float(body.get("strike", ev.get("strike", 0)))
                    d1 = float(body.get("dip", ev.get("dip", 45)))
                    r1 = float(body.get("rake", ev.get("rake", 90)))
                    s2, d2, r2 = focal.auxiliary_plane(s1, d1, r1)
                    axes = focal.stress_axes(s1, d1, r1)
                    regime = focal.classify_regime(r1)
                    ev["strike"] = s1
                    ev["dip"] = d1
                    ev["rake"] = r1
                    ev["strike2"] = s2
                    ev["dip2"] = d2
                    ev["rake2"] = r2
                    ev["focal_regime"] = regime
                    ev["focal_source"] = "SIMMSA_MANUAL"
                    ev["p_axis"] = axes["p_axis"]
                    ev["t_axis"] = axes["t_axis"]
                    with api.e.lock:
                        api.e.dirty.add(ev["id"])
                        if api.e.storage:
                            api.e.storage.save_events([ev])
                    return self._send(200, {"ok": True, "evento": api.event_summary(ev)})
                if p == "/api/pick":
                    ok = api.e.manual_pick(body["id"], body["key"], body.get("phase", "P"), float(body["t"]), bool(body.get("delete")))
                    return self._send(200 if ok else 400, {"ok": ok})
                if p == "/api/recalc":
                    ev_id = body.get("id")
                    z_fix = body.get("z_fix")
                    disabled = body.get("disabled_keys", [])
                    ev = api.e.recalculate_event(ev_id, z_fix=z_fix, disabled_keys=disabled)
                    if ev is not None:
                        picks_out = [{k: v for k, v in q.items() if not k.startswith("_")} for q in ev.get("picks", [])]
                        return self._send(200, {"ok": True, "evento": api.event_summary(ev), "picks": picks_out})
                    else:
                        return self._send(400, {"ok": False, "err": "Se requieren al menos 4 fases P activas para relocalizar"})
                if p == "/api/prueba_alertas":
                    res = api.alerts.test(api.e) if api.alerts else ["Alertas no configuradas"]
                    api.e.log("Prueba de alertas: " + " · ".join(res))
                    return self._send(200, {"result": res})
                if p == "/api/borrar_historial":
                    with api.e.lock:
                        api.e.events = [ev for ev in api.e.events if ev.get("live")]
                        if api.e.storage:
                            api.e.storage.delete_all()
                            api.e.storage.save_events(api.e.events)
                    return self._send(200, {"ok": True})
                if p == "/api/calibrar":
                    if self.headers.get("X-Monitor") != "calibrar":
                        return self._send(403, {"error": "no permitido"})
                    from .netio import usgs_query
                    with api.e.lock:
                        orgs = [ev["org"] for ev in api.e.events]
                        vmin = api.e.cfg.get("validacion", {}).get("magnitud_minima_usgs", 4.5)
                    days = int(min(30, max(2, (time.time() - min(orgs)) / 86400 + 1))) if orgs else 2
                    try:
                        world = usgs_query(days, vmin)
                    except Exception as ex:
                        return self._send(502, {"error": f"El catálogo del USGS no respondió ({ex}). Intenta más tarde."})
                    with api.e.lock:
                        r = api.e.calibrate_now(world)
                    r["dias"], r["usgs"] = days, len(world)
                    return self._send(200, r)
                if p == "/api/prueba_sonido":
                    if self.headers.get("X-Monitor") != "sonido" or not api.alerts:
                        return self._send(403, {"error": "no permitido"})
                    sn = api.alerts.sonidos
                    tipo = body.get("tipo", "preliminar")
                    sn.play(tipo)
                    return self._send(200, {"ok": sn.ok, "tipo": tipo})
                if p == "/api/reiniciar":
                    # el encabezado propio obliga al navegador a una verificación previa que este servidor no acepta
                    # desde otros sitios: solo el panel o reiniciar.bat pueden pedir el reinicio
                    if self.headers.get("X-Monitor") != "reiniciar" or not api.on_restart:
                        return self._send(403, {"error": "no permitido"})
                    self._send(200, {"ok": True, "mensaje": "Reiniciando el monitor…"})
                    threading.Timer(0.5, api.on_restart).start()
                    return
                self._send(404, {"error": "ruta desconocida"})
```

```
            except (BrokenPipeError, ConnectionResetError):
                pass
            except Exception as ex:
                try:
                    self._send(500, {"error": str(ex)})
                except Exception:
                    pass
    return H


class _Server(ThreadingHTTPServer):
    # En Windows SO_REUSEADDR permite que dos programas usen el mismo puerto; se desactiva
    allow_reuse_address = os.name != "nt"
    daemon_threads = True


def start_server(api, host, port):
    srv = _Server((host, port), make_handler(api))
    threading.Thread(target=srv.serve_forever, daemon=True, name="http").start()
    return srv
```

sismo/slprobe.py — 86 líneas

```
"""Prueba de un servidor SeedLink (v0.5.6): ¿responde?, ¿qué software es? y ¿qué redes ofrece?
Se usa desde el panel (/api/probar_seedlink) antes de agregar una fuente nueva en config.json."""
import re
import socket
import time


def probe(host, port=18000, timeout=6.0):
    t0 = time.time()
    out = {"servidor": f"{host}:{port}", "ok": False}
    try:
        s = socket.create_connection((host, int(port)), timeout=timeout)
    except Exception as e:
        out["error"] = f"no conecta: {e}"
        return out
    s.settimeout(timeout)
    f = s.makefile("rb")
    try:
        s.sendall(b"HELLO\r\n")
        l1 = f.readline(200).decode("ascii", "replace").strip()
        l2 = f.readline(200).decode("ascii", "replace").strip()
        out.update(ok=bool(l1), hello=l1, organizacion=l2, ms=round((time.time() - t0) * 1000))
        if not l1:
            out["error"] = "conecta pero no responde como SeedLink"
            return out
        # lista de estaciones: INFO STREAMS (RingServer y SeisComP responden con XML en paquetes SLINFO)
        s.sendall(b"INFO STATIONS\r\n")
        # v2.0.0: la lista de EarthScope es muy larga; con internet lento no alcanzaba el tiempo y se usaba una lista
        # incompleta (el monitor descartaba la mayoría de sus estaciones). Ahora se marca si la lista llegó completa.
        buf, end, complete = b"", time.time() + timeout * 8, False
        while time.time() < end:
            pkt = f.read(520)
            if not pkt or len(pkt) < 520:
                break
            if pkt[:6] == b"SLINFO":
                rec = pkt[8:]
                n = int.from_bytes(rec[30:32], "big")
                off = int.from_bytes(rec[44:46], "big")
                if not (0 < off < 512 and 0 < n <= 512 - off):        # orden de bytes inverso
                    n, off = int.from_bytes(rec[30:32], "little"), int.from_bytes(rec[44:46], "little")
                buf += rec[off:off + n] if 0 < off < 512 else rec[48:]
            if pkt[:6] == b"SLINFO" and pkt[7:8] != b"*":      # último paquete de la respuesta
                complete = True
                break
        txt = buf.decode("latin-1", "replace")
        sta = re.findall(r'<station name="([^"]+)" network="([^"]+)"', txt)
        nets = {}
        for st, net in sta:
            nets[net] = nets.get(net, 0) + 1
        out["estaciones"] = len(sta)
        out["completa"] = complete
        out["claves"] = sorted({f"{n}_{st}" for st, n in sta})
        out["redes"] = dict(sorted(nets.items(), key=lambda x: -x[1]))
        if not sta:
            out["nota"] = "respondió HELLO pero no entregó la lista de estaciones (puede restringir INFO)"
    except Exception as e:
        out["error"] = out.get("error") or f"{e}"
    finally:
        try:
            s.sendall(b"BYE\r\n")
            s.close()
        except Exception:
            pass
    return out


def dialog(host, port, cmds, timeout=6.0):
    """Diagnóstico: manda comandos SeedLink y devuelve lo que contesta el servidor (texto crudo)."""
    out = []
    s = socket.create_connection((host, int(port)), timeout=timeout)
    s.settimeout(timeout)
    try:
        for c in cmds:
            s.sendall((c + "\r\n").encode("ascii"))
            time.sleep(0.4)
            try:
                data = s.recv(4096)
            except socket.timeout:
                data = b"<sin respuesta>"
            out.append([c, data[:200].decode("latin-1", "replace")])
    finally:
        try:
            s.close()
        except Exception:
            pass
    return out
```

sismo/sonido.py — 56 líneas

```
"""Avisos sonoros en las bocinas de la PC (v0.3.9).

Sonidos (archivos WAV en la carpeta sonidos/, convertidos de los MP3 elegidos por el usuario):
- preliminar.wav ... aviso PRELIMINAR de un sismo que llega al umbral de su zona (alarma japonesa EAS)
- final.wav ........ SOLUCIÓN FINAL de ese sismo (desfibrilador)
- alarma_7_5.wav ... magnitud ≥ 7.5 (severe warning alarm) — suena en cualquier zona
- alarma_8_5.wav ... magnitud ≥ 8.5 (security alarm, 17 s) — suena en cualquier zona

Los sonidos se reproducen en orden en un hilo propio (no detienen al monitor). Fuera de Windows
no se reproduce nada.
"""
import os
import queue
import threading
import time

BASE = os.path.join(os.path.dirname(os.path.dirname(os.path.abspath(__file__))), "sonidos")
TIPOS = {"preliminar": "preliminar.wav", "final": "final.wav", "alarma_7_5": "alarma_7_5.wav",
  "alarma_8_5": "alarma_8_5.wav"}


class Sonidos:
    def __init__(self, log=print):
        self.log = log
        self.q = queue.Queue()
        self.last = None             # (hora, tipo)
        self.error = None
        try:
            import winsound          # noqa: F401
            self.ok = True
        except Exception:
            self.ok = False
        threading.Thread(target=self._worker, daemon=True, name="sonido").start()

    def play(self, tipo):
        if tipo in TIPOS:
            self.q.put(tipo)

    def _worker(self):
        while True:
            tipo = self.q.get()
            path = os.path.join(BASE, TIPOS[tipo])
            if not self.ok:
                continue
            try:
                import winsound
                if os.path.exists(path):
                    winsound.PlaySound(path, winsound.SND_FILENAME | winsound.SND_NODEFAULT)
                else:                # sin el archivo: pitidos de respaldo
                    for _ in range(3):
                        winsound.Beep(880, 400)
                        time.sleep(0.15)
                self.last = (time.time(), tipo)
                self.error = None
            except Exception as e:
                self.error = str(e)[:200]
                self.log(f"Sonido: no se pudo reproducir {tipo} ({e})")
```

sismo/storage.py — 247 líneas

```
"""Almacenamiento: base SQLite de sismos, archivos JSON de estado, formas de onda y archivo miniSEED."""
import json
import os
import sqlite3
import threading
import time

import numpy as np

SKIP = {"live", "triedS", "triedP", "aiDone", "msTried"}


def _clean(ev):
    out = {k: v for k, v in ev.items() if k not in SKIP and not k.startswith("_")}
    out["picks"] = [{k: v for k, v in p.items() if not k.startswith("_")} for p in ev["picks"]]
    return out


class Storage:
    def __init__(self, folder):
        self.folder = folder
        os.makedirs(folder, exist_ok=True)
        os.makedirs(os.path.join(folder, "formas_onda"), exist_ok=True)
        self.db_path = os.path.join(folder, "sismos.sqlite")
        self.lock = threading.Lock()
        with self._conn() as c:
            c.execute("CREATE TABLE IF NOT EXISTS sismos (id TEXT PRIMARY KEY, origen REAL, datos TEXT)")

    def _conn(self):
        return sqlite3.connect(self.db_path, timeout=10)

    def save_events(self, events):
        rows = [(e["id"], e["org"], json.dumps(_clean(e), default=float)) for e in events]
        if not rows:
            return
        with self.lock, self._conn() as c:
            c.executemany("INSERT OR REPLACE INTO sismos VALUES (?,?,?)", rows)

    def load_events(self, limit=500):
        with self.lock, self._conn() as c:
            rows = c.execute("SELECT datos FROM sismos ORDER BY origen DESC LIMIT ?", (limit,)).fetchall()
        out = []
        for (d,) in rows:
            e = json.loads(d)
            e.update(live=False, triedS=set(), triedP=set(), aiDone=set(), msTried=set())
            out.append(e)
        return out

    def delete_event(self, ev_id):
        with self.lock, self._conn() as c:
            c.execute("DELETE FROM sismos WHERE id = ?", (ev_id,))

    def delete_all(self):
        with self.lock, self._conn() as c:
            c.execute("DELETE FROM sismos")

    def load_json(self, name, default):
        try:
            with open(os.path.join(self.folder, name), encoding="utf-8") as f:
                return json.load(f)
        except Exception:
            return default

    def save_json(self, name, obj):
        """Guarda de forma atómica. En Windows un antivirus o un respaldo puede bloquear el archivo
        unos segundos: se reintenta y, si sigue bloqueado, se deja para el siguiente guardado sin detener nada."""
        path = os.path.join(self.folder, name)
        text = json.dumps(obj, ensure_ascii=False)
        for intento in range(5):
            tmp = f"{path}.{os.getpid()}.{threading.get_ident()}.{intento}.tmp"
            try:
                with open(tmp, "w", encoding="utf-8") as f:
                    f.write(text)
                os.replace(tmp, path)
                return True
            except OSError:
                try:
                    os.remove(tmp)
                except OSError:
                    pass
                time.sleep(0.3 * (intento + 1))
        return False

    def save_waveforms(self, ev_id, traces):
        if not traces:
            return
        np.savez_compressed(os.path.join(self.folder, "formas_onda", ev_id + ".npz"),
                            meta=json.dumps([{k: t[k] for k in ("key", "t0", "fs")} for t in traces]),
                            **{f"d{i}": t["data"] for i, t in enumerate(traces)})
```

```

    def load_waveforms(self, ev_id):
        path = os.path.join(self.folder, "formas_onda", ev_id + ".npz")
        if not os.path.exists(path):
            return []
        d = np.load(path)
        meta = json.loads(str(d["meta"]))
        return [dict(m, data=d[f"d{i}"]) for i, m in enumerate(meta)]


def archive_traces(folder, ev, stations, f1=0.5, f2=4.0, max_n=60):
    """v0.6.3: sección de registro reconstruida desde el archivo miniSEED continuo (datos/archivo) para sismos
    cuyas formas de onda no se guardaron (p. ej. el monitor se reinició antes de los 20 min). Componente vertical."""
    import glob
    from .mseed import read_records
    from .dsp import bandpass
    keys = list(dict.fromkeys(p["key"] for p in ev.get("picks", [])))[:max_n]
    t_end = min(1300, max([90] + [p["t"] - ev["org"] + 90 for p in ev.get("picks", [])]))
    t_a, t_b = ev["org"] - 60, ev["org"] + t_end
    days = sorted({time.strftime("%Y/%m/%d", time.gmtime(t)) for t in (t_a, t_b)})
    out = []
    for k in keys:
        net, sta = k.split("_", 1)
        s = stations.get(k)
        pref = getattr(s, "pref", None) or ""
        loc, band = (pref.split(".") + [""])[:2] if pref else ("*", "?H")
        recs = []
        for day in days:
            pat = os.path.join(folder, day, f"{net}.{sta}.{loc if pref else '*'}.{band if pref else '?H'}Z.mseed")
            for path in glob.glob(pat)[:1]:
                try:
                    with open(path, "rb") as fh:
                        buf = fh.read()
                except OSError:
                    continue
                try:
                    for r in read_records(buf, t_a=t_a, t_b=t_b):
                        fs, t0, data = r[4], r[5], r[6]
                        if fs and len(data) and t0 + len(data) / fs >= t_a and t0 <= t_b:
                            recs.append((t0, fs, np.asarray(data, float)))
                except Exception:
                    pass        # registro dañado o de otro tipo: se usan los que se leyeron
        if not recs:
            continue
        try:
            tr = _assemble(recs, ev, f1, f2, k)
        except Exception:
            tr = None
        if tr:
            out.append(tr)
    return out


def _assemble(recs, ev, f1, f2, k):
    from .dsp import bandpass
    if True:
        recs.sort(key=lambda r: r[0])
        fs = recs[0][1]
        t0 = recs[0][0]
        n = int((recs[-1][0] + len(recs[-1][2]) / fs - t0) * fs) + 1
        if n < fs * 40 or n > fs * 4000:
            return None
        y = np.full(n, np.nan)
        for rt, _, d in recs:
            i = int(round((rt - t0) * fs))
            y[i:i + len(d)] = d[:max(0, n - i)]
        good = ~np.isnan(y)
        if good.sum() < fs * 40:
            return None
        y[~good] = np.nanmean(y)
        f = bandpass(y - y.mean(), fs, f1, f2)
        skip = max(0, int(round((ev["org"] - 20 - t0) * fs)))
        f = f[skip:]
        q = int(round(fs / 10)) if fs > 10 else 1
        return {"key": k, "t0": t0 + skip / fs, "fs": fs / q, "data": f[::q].astype(np.float32)
    }


class Archive:
    """Archivo continuo en miniSEED: datos/archivo/AAAA/MM/DD/RED.EST.LOC.CANAL.mseed

    v0.4.5: los registros se juntan en memoria y se escriben cada 20 s, un archivo por canal (antes se abría y
    cerraba el archivo con cada registro de 512 bytes: cientos de aperturas por segundo con 600 estaciones,
    pesado para el disco y el antivirus de Windows)."""

    def __init__(self, folder, days, log=print):
        import threading
        self.folder, self.days, self.log = folder, days, log
        self.last_clean = 0
        self.buf = {}
```

```
        self.lock = threading.Lock()
        self.error_t = 0.0
        self.bytes_written = 0
        self.last_flush = 0.0
        threading.Thread(target=self._loop, daemon=True, name="archivo").start()
        import atexit
        atexit.register(self.flush)

    def write(self, trace):
        import io
        st = trace.stats
        buf = io.BytesIO()
        trace.write(buf, format="MSEED")
        self.write_record((st.network, st.station, st.location, st.channel, 0, st.starttime.tim
estamp), buf.getvalue())

    def write_record(self, item, rec):
        """Agrega el registro miniSEED original (512 bytes) al archivo diario del canal (en el
próximo vaciado).
        item: (red, estación, ubicación, canal, fs, t0, datos)."""
        net, sta, loc, cha, _, t0 = item[:6]
        key = (time.strftime("%Y/%m/%d", time.gmtime(t0)), f"{net}.{sta}.{loc}.{cha}.mseed")
        with self.lock:
            self.buf.setdefault(key, []).append(rec)

    def flush(self):
        with self.lock:
            buf, self.buf = self.buf, {}
        n = 0
        for (day, name), recs in buf.items():
            try:
                path = os.path.join(self.folder, day)
                os.makedirs(path, exist_ok=True)
                data = b"".join(recs)
                with open(os.path.join(path, name), "ab") as f:
                    f.write(data)
                n += len(data)
            except OSError as e:
                if time.time() - self.error_t > 600:
                    self.log(f"Archivo: no se pudo escribir ({e})")
                self.error_t = time.time()
        self.bytes_written += n
        self.last_flush = time.time()

    def _loop(self):
        while True:
            time.sleep(20)
            try:
                self.flush()
                if time.time() - self.last_clean > 6 * 3600:
                    self.last_clean = time.time()
                    self.clean()
            except Exception as e:
                self.error_t = time.time()
                self.log(f"Archivo: {e}")

    def clean(self):
        limit = time.time() - self.days * 86400
        for root, dirs, files in os.walk(self.folder, topdown=False):
            for fn in files:
                p = os.path.join(root, fn)
                try:
                    if os.path.getmtime(p) < limit:
                        os.remove(p)
                except OSError:
                    pass
            if root != self.folder and not os.listdir(root):
                try:
                    os.rmdir(root)
                except OSError:
                    pass
```

sismo/ticket.py — 531 líneas

```
"""Ticket técnico de un sismo: toda la información sismológica de la solución en tres formatos.

- TXT  : ficha legible para imprimir o archivar.
- JSON : los mismos datos, estructurados, para análisis posterior.
- XML  : QuakeML 1.2 (formato estándar de catálogos sísmicos: SeisComP, ObsPy, FDSN).

Solo lee el evento; no modifica nada del motor ni de la solución.
"""
import json
import math
import os
import time
from datetime import datetime, timezone
from xml.sax.saxutils import escape

from .locator import dist_km, bearing
from .magnitude import mmi, roman, SOIL
from .places import CITIES, where_is
from .zones import zone_of
from .tsunami import DISCLAIMER as TSUNAMI_NOTE

VERSION = "2.3.1"
COMPASS = ["N", "NE", "E", "SE", "S", "SO", "O", "NO"]


# ---------------------------------------------------------------- utilidades
def _utc(t, ms=True):
    if t is None:
        return "—"
    d = datetime.fromtimestamp(float(t), tz=timezone.utc)
    return d.strftime("%Y-%m-%d %H:%M:%S") + (f".{int(round(d.microsecond / 1000)) % 1000:03d}"
   if ms else "")


def _cdmx(t):
    return time.strftime("%d-%m-%Y %H:%M:%S", time.gmtime(float(t) - 6 * 3600))


def _iso(t):
    d = datetime.fromtimestamp(float(t), tz=timezone.utc)
    return d.strftime("%Y-%m-%dT%H:%M:%S.") + f"{d.microsecond // 1000:03d}Z"


def _f(x, nd=1, suf=""):
    if x is None:
        return "—"
    try:
        if not math.isfinite(float(x)):
            return "—"
    except (TypeError, ValueError):
        return str(x)
    return f"{float(x):.{nd}f}{suf}"


def _depth_class(z):
    if z < 70:
        return "Superficial (< 70 km)"
    if z < 300:
        return "Intermedio (70–300 km)"
    return "Profundo (> 300 km)"


def _t_ref(ev):
    # hora de detección: no cambia al relocalizar, así el número de ticket y el archivo son est
  ables
    return ev.get("created") or ev["org"]


def ticket_id(ev):
    return f"SMW-{time.strftime('%Y%m%d-%H%M%S', time.gmtime(_t_ref(ev)))}-{ev['id']}"


def file_stem(ev):
    return f"ticket_{time.strftime('%Y%m%d_%H%M%S', time.gmtime(_t_ref(ev)))}_{ev['id']}"


def _dp_summary(ev):
    dp = ev.get("depthPhases")
    if not dp:
        return None
    return {"accepted": dp.get("accepted"), "stack_depth_km": dp.get("h"), "contrast": dp.get("
  contrast"),
            "stations_scanned": dp.get("nStations"), "n_pP": ev.get("npP", 0), "n_sP": ev.get("
  nsP", 0),
            "depth_P_only_km": ev.get("zPonly"), "err_depth_P_only_km": ev.get("errZPonly")}


# ---------------------------------------------------------------- construcción del ticket
def build(engine, ev):
    """Reúne todos los datos técnicos del evento en un diccionario (sin tocar el evento)."""
    e = engine
    lat, lon, z, org = float(ev["lat"]), float(ev["lon"]), float(ev["z"]), float(ev["org"])
    M = e.mag_of(ev)
    q = e.quality(ev)

```

```
  # --- fases
  picks = []
  for p in sorted(ev.get("picks", []), key=lambda p: p["t"]):
      d = float(dist_km(lat, lon, p["lat"], p["lon"]))
      az = float(bearing(lat, lon, p["lat"], p["lon"]))
      R = math.hypot(d, z)
      res = p.get("res")
      t_obs = float(p["t"]) - org
      corr = e.corr_of(p["key"], p["phase"], d) if hasattr(e, "corr_of") else 0.0
      t_theo = float(e.tt.tt(p["phase"], d, z))
      method = "manual" if p.get("manual") else "IA" if p.get("ai") else "AIC" if p["phase"] == "P" and p.get("snr") is not None else "AIC dirigido"
      s = e.stations.get(p["key"])
      picks.append({
          "station": p.get("code", p["key"].replace("_", ".")), "key": p["key"], "phase": p["phase"],
          "lat": p["lat"], "lon": p["lon"], "dist_km": d, "dist_deg": d / 111.195, "azimuth": az, "hypo_km": R,
          "arrival_utc": _utc(p["t"]), "arrival_epoch": float(p["t"]), "travel_obs_s": t_obs, "travel_theo_s": t_theo,
          "station_corr_s": corr, "residual_s": res, "sigma_s": p.get("sigma"),
          "trigger_ratio": p.get("snr") if p["phase"] == "P" else None, "ml": p.get("ml"), "mwp": p.get("mwp"),
          "mb": p.get("mb_sta"), "mb_A_nm": p.get("mbA"), "mb_T_s": p.get("mbT"), "mB": p.get("mB_sta"), "ms": p.get("ms_sta"),
          "method": method,
          "detector": {"L": "local", "T": "lejano"}.get(p.get("det")) if p["phase"] == "P" else None,
          "band": getattr(s, "band", None) if s else None,
          "comps": "".join(k for k in ("Z", "N", "E") if s and k in s.comps) or None,
      })
  P = [p for p in picks if p["phase"] == "P"]
  stations = sorted({p["key"] for p in picks})
  dists = [p["dist_km"] for p in picks]
  res = [p["residual_s"] for p in picks if p["residual_s"] is not None]

  # --- intensidad en puntos de interés y ciudades cercanas
  poi = [{"name": x["name"], "soil": x["soilName"], "dist_km": x["d"], "mmi": x["I"], "roman": x["roman"],
          "s_arrival_utc": _utc(x["ts"]), "s_travel_s": x["ts"] - org} for x in e.poi_impact(ev)]
  cities = []
  if M is not None:
      dc = dist_km([c[1] for c in CITIES], [c[2] for c in CITIES], lat, lon)
      near = [CITIES[i] for i in sorted(range(len(CITIES)), key=lambda i: dc[i])[:6]]
      for name, cla, clo in near:
          d = float(dist_km(lat, lon, cla, clo))
          I = float(mmi(M, d, z, "firme"))
          b = float(bearing(cla, clo, lat, lon))
          cities.append({"name": name, "dist_km": d, "epicenter_dir": COMPASS[int(round(b / 45)) % 8],
                         "mmi_firm": I, "roman": roman(I),
                         "p_arrival_s": float(e.tt.tt("P", d, z)), "s_arrival_s": float(e.tt.tt("S", d, z))})

  # --- parámetros físicos derivados (relaciones empíricas; valores aproximados)
  derived = None
  if M is not None:
      mw_like = ev.get("mw") if ev.get("mw") is not None else ev.get("mwp") if ev.get("mwp") is not None else M
      derived = {
          "basis": "Mw estimada" if ev.get("mw") is not None else "Mwp" if ev.get("mwp") is not None else f"{ev.get('magType', 'ML')} usada como aproximación de Mw",
          "moment_Nm": 10 ** (1.5 * mw_like + 9.1),
          "energy_J": 10 ** (1.5 * M + 4.8),
          "tnt_equiv_t": 10 ** (1.5 * M + 4.8) / 4.184e9,
          "rupture_length_km": 10 ** (-2.44 + 0.59 * M),
          "rupture_area_km2": 10 ** (-3.49 + 0.91 * M),
          "refs": "M0: Hanks y Kanamori (1979); energía: Gutenberg-Richter (1956); longitud y área de ruptura: "
                  "Wells y Coppersmith (1994), todos los tipos de falla.",
      }

  t_pre = ev.get("tEarly") or ev.get("tPrelim") or ev.get("created")
  t_fin = ev.get("tFinal")
  m = ev.get("match")
  usgs = None
  if m:
      pub = None
      if hasattr(e, "storage") and e.storage:
          cache = e.storage.load_json("usgs_publicacion.json", {})
          pub = (cache.get(m["id"]) or {}).get("pub")
      t_usgs = (pub - org) / 60 if pub else None
      t_pre_el = (t_pre - org) / 60 if t_pre else None
      diff_alert = (t_usgs - t_pre_el) if (t_usgs is not None and t_pre_el is not None) else None
      winner = None
      if diff_alert is not None:
          if abs(diff_alert) < 0.5:
              winner = "Alerta simultánea (< 30 s de diferencia)"
          elif diff_alert > 0:
              winner = f"MONITOR MUNDIAL alertó {diff_alert:.1f} min antes que el USGS"
          else:
              winner = f"USGS publicó {-diff_alert:.1f} min antes que el monitor"

```

```
      usgs = {"id": m.get("id"), "mag": m.get("mag"), "magType": m.get("magType"), "lat": m.get("lat"), "lon": m.get("lon"),
              "depth_km": m.get("z"), "time_utc": _utc(m.get("t")), "place": m.get("place"), "url": m.get("url"),
              "diff_epicenter_km": m.get("dEpi"), "diff_depth_km": m.get("dZ"), "diff_mag": m.get("dM"), "diff_time_s": m.get("dT"),
              "diff_mag_type": m.get("dMtype"), "diff_mag_preferred": m.get("dMpref"),
              "time_pub_utc": _utc(pub) if pub else None, "elapsed_pub_min": t_usgs,
              "alert_diff_min": diff_alert, "alert_winner": winner}

  det = e.cfg.get("deteccion", {})
  return {
      "ticket": ticket_id(ev), "event_id": ev["id"], "generated_utc": _utc(time.time(), ms=False),
      "software": f"Monitor Sísmico MUNDIAL v{VERSION}",
      "status": ("final" if ev.get("final") else "preliminar") + (" · revisado manualmente" if ev.get("manual") else "")
                + ("" if ev.get("live") else " · archivado"),
      "origin": {
          "time_utc": _utc(org), "time_cdmx": _cdmx(org), "epoch": org,
          "lat": lat, "lon": lon, "depth_km": z, "depth_class": _depth_class(z),
          # v2.1.0: cómo se amarró la profundidad
          "depth_constraint": ("fases pP/sP" if ev.get("dpH") is not None else
                               "sismicidad histórica de la zona + ondas P" if ev.get("zPrior") else "solo ondas P"),
          "region": where_is(lat, lon),
          "fe_region": ev.get("region") or "",
          "err_h_major_km": None if ev.get("errOpen") else ev.get("errH"),
          "err_h_minor_km": None if ev.get("errOpen") else ev.get("errMin"),
          "err_azimuth_deg": None if ev.get("errOpen") else ev.get("errAz"),
          "err_open": bool(ev.get("errOpen")), "err_depth_km": ev.get("errZ"),
          "rms_s": ev.get("rms"), "gap_deg": ev.get("gap"),
          "n_P": ev.get("nP"), "n_S": ev.get("nS"), "n_stations": len(stations),
          "min_dist_km": min(dists) if dists else None, "max_dist_km": max(dists) if dists else None,
          "max_abs_residual_s": max(abs(r) for r in res) if res else None,
          "quality": q, "velocity_model": e.tt.name, "station_corrections": bool(det.get("correcciones_por_estacion")),
          "edge": bool(ev.get("edge")), "teleseism": ev.get("tele"),
      },
      "magnitude": {
          "preferred": M, "preferred_type": ev.get("magType", "ML"),
          "preliminary": ev.get("magPre"), "preliminary_type": ev.get("magPreType"), "preliminary_str": ev.get("magPreStr"),
          "time_prelim_utc": _utc(t_pre) if t_pre else None,
          "elapsed_prelim_min": (t_pre - org) / 60 if (t_pre and org) else None,
          "time_final_utc": _utc(t_fin) if t_fin else None,
          "elapsed_final_min": (t_fin - org) / 60 if (t_fin and org) else None,
          "ml": ev.get("ml"), "ml_sd": ev.get("mlSd"), "ml_n": ev.get("nml"), "ml_components": ev.get("mlComp"),
          "mwp": ev.get("mwp"), "mwp_n": ev.get("mwpN"), "mwp_sd": ev.get("mwpSd"),
          "station_mwp": [{"station": p["station"], "mwp": p["mwp"], "dist_deg": p["dist_deg"]} for p in P if p.get("mwp") is not None],
          "mb": ev.get("mb"), "mb_n": ev.get("mbN"), "mb_sd": ev.get("mbSd"),
          "mbR": ev.get("mbR"), "mbR_n": ev.get("mbRN"), "mbR_sd": ev.get("mbRSd"),
          "mB": ev.get("mB"), "mB_n": ev.get("mBN"), "mB_sd": ev.get("mBSd"),
          "ms": ev.get("ms"), "ms_n": ev.get("msN"), "ms_sd": ev.get("msSd"),
          "mw": ev.get("mw"), "mw_sd": ev.get("mwSd"), "mw_parts": ev.get("mwParts") or {}, "mw_dropped": ev.get("mwDropped") or {},
          "station_global": [{"station": p["station"], "dist_deg": p["dist_deg"], "mwp": p.get("mwp"), "mb": p.get("mb"),
                              "mb_A_nm": p.get("mb_A_nm"), "mb_T_s": p.get("mb_T_s"), "mB": p.get("mB"), "ms": p.get("ms")}
                             for p in P if any(p.get(k) is not None for k in ("mwp", "mb", "mB", "ms"))],
          "conversions": e.calib.summary()["conversiones"] if hasattr(e, "calib") else [],
          "station_ml": [{"station": p["station"], "ml": p["ml"], "hypo_km": p["hypo_km"]} for p in P if p["ml"] is not None],
      },
      "picks": picks, "intensity_poi": poi, "nearest_cities": cities, "derived": derived, "usgs": usgs,
      "method": {
          "detector": f"dos STA/LTA recursivos por estación: local 1 s / 30 s, 1.5–8 Hz, umbral {det.get('umbral_sta_lta')}; "
                      f"ondas lejanas 3 s / 60 s, 0.5–2 Hz, umbral {det.get('umbral_sta_lta_lejano')} (adaptativos), tiempo muerto 30 s",
          "picker": "AIC de Maeda (1985) en la banda del detector; control de calidad SNR ≥ 2.5",
          "associator": f"nucleación global en dos etapas: malla de Fibonacci 2° × 20/120/350/600 km y malla fina 0.25° × 8 "
                        f"profundidades, calificación por verosimilitud; mínimo {det.get('estaciones_minimas')} estaciones",
          "locator": "Tierra esférica, 0–700 km: búsqueda robusta en malla (L1) + malla probabilística (L2), elipse de error al 68 %",
          "ml": "IASPEI: ML = log10(A) + 1.11·log10(R) + 0.00189·R − 2.09 (A en nm Wood-Anderson, R hipocentral km); solo estaciones < 600 km",
          "mwp": "Tsuboi et al. (1995), ρ 3400 kg/m³, α 7.9 km/s, Fp 0.52, estaciones de banda ancha a 3°–95°, SNR ≥ 3; mediana",
          "mb": "IASPEI: mb = log10(A/T) + Q(Δ,h) − 3.0; A desplazamiento (nm) en simulación de periodo corto (0.7–2.5 Hz), "
                "T ≤ 3 s, ventana de 30 s tras la P, 20°–100°; Q de Gutenberg-Richter (1956) aproximada y recalibrada con el USGS",
          "mB": "IASPEI: mB_BB = log10(Vmax/2π) + Q(Δ,h) − 3.0; Vmax velocidad vertical (nm/s) en 0.033–3 Hz, hasta 60 s tras la P, 20°–100°",
          "ms": "IASPEI: Ms_BB = log10(Vmax/2π) + 1.66·log10(Δ) + 0.3; Vmax velocidad vertica
```

```
l (nm/s) de la Rayleigh, 3–60 s, 2°–160°, h < 60 km",
            "mw": "Mw estimada: promedio ponderado de Mwp, mB, mb y Ms convertidas a Mw (Scordi
lis 2006; Bormann y Saul 2008), "
                  "o con las relaciones ajustadas contra el Mw del USGS cuando hay 15 sismos o
más",
            "intensity": "Allen, Wald y Worden (2012) con término de suelo (roca 0, firme 0.3,
transición 1.0, lago 2.0)",
            "depth_phases": "pP y sP: apilamiento por profundidad (25–700 km) de STA/LTA 0.5–2.
5 Hz en estaciones a 25°–95°, "
                            "lectura AIC y relocalización con las diferencias pP − P y sP − P (
tiempos: iasp91 sobre el foco)",
            "tsunami": "informativo, umbrales tipo PTWC: 6.5–7.0 sin amenaza destructiva; 7.1–7
.5 < 300 km; 7.6–7.8 < 1000 km; "
                       "≥ 7.9 cuenca oceánica; solo profundidad ≤ 100 km y epicentro en el mar
o a ≤ 50 km de la costa",
        },
        "zone": (zone_of(e.cfg.get("alertas", {}).get("zonas") or [], lat, lon) or {}).get("nom
bre"),
        "tsunami": ev.get("tsunami"),
        "depth_phases": _dp_summary(ev),
        "disclaimer": "Solución automática experimental. No es información oficial; consulte al
 USGS y a la agencia sismológica de cada país.",
    }


# ---------------------------------------------------------------- formato texto
def to_text(t):
    o, m = t["origin"], t["magnitude"]
    L = []
    w = 100
    L.append("=" * w)
    L.append("TICKET TÉCNICO DE SISMO".center(w))
    L.append(f"{t['software']} — detección y solución automática".center(w))
    L.append("=" * w)
    L.append(f"Ticket: {t['ticket']}      Evento: {t['event_id']}      Estado: {t['status']}")
    L.append(f"Generado: {t['generated_utc']} UTC")
    L.append("")
    mag = f"{_f(m['preferred'])} {m['preferred_type']}" if m["preferred"] is not None else "—"
    L.append(f"  MAGNITUD {mag}   ·   {o['region']}")
    if o.get("fe_region") and o["fe_region"] not in o["region"]:
        L.append(f"  Región sísmica (Flinn-Engdahl): {o['fe_region']}")
    L.append("")
    L.append("1. ORIGEN (HIPOCENTRO)")
    L.append("-" * w)
    rows = [
        ("Hora de origen (UTC)", o["time_utc"]), ("Hora de origen (CDMX, UTC−6)", o["time_cdmx"
]),
        ("Latitud", f"{o['lat']:.4f}°"), ("Longitud", f"{o['lon']:.4f}°"),
        ("Profundidad", f"{o['depth_km']:.1f} km ± {_f(o['err_depth_km'], 1)} km   [{o['depth_c
lass']}]"),
        ("Profundidad amarrada con", o.get("depth_constraint") or "—"),
        ("Región", o["region"]),
        ("Elipse de error 68 %", "no acotada (> 200 km)" if o["err_open"] else
         f"semieje mayor {_f(o['err_h_major_km'])} km, menor {_f(o['err_h_minor_km'])} km, azim
ut {_f(o['err_azimuth_deg'], 0)}°"),
        ("RMS de residuos", f"{_f(o['rms_s'], 2)} s   (máx. |residuo| {_f(o['max_abs_residual_s
'], 2)} s)"),
        ("Hueco azimutal (gap)", f"{_f(o['gap_deg'], 0)}°"),
        ("Fases usadas", f"{o['n_P']} P, {o['n_S']} S en {o['n_stations']} estaciones"),
        ("Distancia estación más cercana / lejana", f"{_f(o['min_dist_km'], 0)} km / {_f(o['max
_dist_km'], 0)} km"),
        ("Calidad de la solución", o["quality"]),
        ("Modelo de velocidades", o["velocity_model"] + (" + correcciones por estación" if o["s
tation_corrections"] else "")),
    ]
    if o["teleseism"]:
        rows.append(("Observación", o["teleseism"]))
    dp = t.get("depth_phases")
    if dp:
        rows.append(("Fases de profundidad (pP, sP)",
                     (f"usadas: {dp['n_pP']} pP y {dp['n_sP']} sP; apilamiento h ≈ {_f(dp['stac
k_depth_km'], 0)} km "
                      f"(contraste {_f(dp['contrast'], 1)}); profundidad solo con P: {_f(dp['de
pth_P_only_km'], 0)} ± "
                      f"{_f(dp['err_depth_P_only_km'], 0)} km") if dp["accepted"] else
                     f"no concluyentes ({dp['stations_scanned']} estaciones revisadas); la prof
undidad sale solo de la P"))
    if t.get("zone"):
        rows.append(("Zona de alerta", t["zone"]))
    for k, v in rows:
        L.append(f"  {k:<40} {v}")
    L.append("")
    L.append("2. MAGNITUD")
    L.append("-" * w)
    fin_time = f"   (consolidada a los {m['elapsed_final_min']:.1f} min del sismo)" if m.get("e
lapsed_final_min") is not None else ""
    L.append(f"  {'Magnitud preferida (final)':<40} {mag}{fin_time}")
    if m.get("preliminary_str") or m.get("preliminary") is not None:
        p_str = f"M {m.get('preliminary_str') or m.get('preliminary')} {m.get('preliminary_type
') or ''}"
        p_time = f"   (alertada a los {m['elapsed_prelim_min']:.1f} min del sismo)" if m.get("e
lapsed_prelim_min") is not None else ""
        L.append(f"  {'Magnitud preliminar emitida':<40} {p_str}{p_time}")
    L.append(f"  {'ML (IASPEI)':<40} {_f(m['ml'], 2)}" + (f" ± {_f(m['ml_sd'], 2)}" if m["ml_sd
"] is not None else "")
```

```
            + (f"  ({m['ml_n']} estaciones, componentes {m['ml_components']})" if m["ml_n"] el
se ""))
  L.append(f"  {'Mwp (Tsuboi)':<40} " + (f"{_f(m['mwp'], 2)}" + (f" ± {_f(m.get('mwp_sd'), 2)
}" if m.get("mwp_sd") is not None else ""))
            + f"  ({m['mwp_n']} estaciones)" if m["mwp"] is not None else "no calculada (menos
 de 3 estaciones de banda ancha a 3°–95° con señal suficiente)"))
  for lab, k in (("mb (IASPEI)", "mb"), ("mb regional (5°–20°, provisional)", "mbR"), ("mB_BB
 (IASPEI)", "mB"), ("Ms_BB (IASPEI)", "ms")):
      if k == "mbR" and m.get("mbR") is None:
          continue
      L.append(f"  {lab:<40} " + (f"{_f(m[k], 2)}" + (f" ± {_f(m[k + '_sd'], 2)}" if m.get(k
+ "_sd") is not None else ""))
              + f"  ({m[k + '_n']} estaciones)" if m.get(k) is not None else
              ("no aplica (sismo de 60 km de profundidad o más)" if k == "ms" and o["depth_k
m"] >= 60 else "no calculada todavía o sin estaciones suficientes")))
  L.append(f"  {'Mw estimada':<40} " + (f"{_f(m['mw'], 2)}" + (f" ± {_f(m['mw_sd'], 2)}" if m
.get("mw_sd") is not None else ""))
            + "   a partir de " + ", ".join(f"{k} → {v:.2f}" for k, v in m["mw_parts"].items()
)
            + ("   (descartada por discrepar: " + ", ".join(f"{k} → {v:.2f}" for k, v in m["mw
_dropped"].items()) + ")" if m.get("mw_dropped") else "")
            if m.get("mw") is not None else "—"))
  if m.get("conversions"):
      L.append("")
      L.append("  Conversión a Mw en uso:")
      for c in m["conversions"]:
          L.append(f"    {c['tipo']:<5} {c['relacion']}   [{c['origen']}]")
  if m.get("station_global"):
      L.append("")
      L.append(f"  {'Estación':<14}{'Dist °':>8}{'Mwp':>7}{'mb':>7}{'A nm':>10}{'T s':>6}{'mB
':>7}{'Ms':>7}")
      for s_ in sorted(m["station_global"], key=lambda s_: s_["dist_deg"]):
          L.append(f"  {s_['station']:<14}{s_['dist_deg']:>8.1f}{_f(s_['mwp'], 2):>7}{_f(s_['
mb'], 2):>7}{_f(s_['mb_A_nm'], 0):>10}"
                   f"{_f(s_['mb_T_s'], 2):>6}{_f(s_['mB'], 2):>7}{_f(s_['ms'], 2):>7}")
  if m["station_ml"]:
      L.append("")
      L.append(f"  {'Estación':<14}{'ML':>6}{'Dist. hipocentral':>20}")
      for s in sorted(m["station_ml"], key=lambda s: s["hypo_km"]):
          L.append(f"  {s['station']:<14}{_f(s['ml'], 2):>6}{_f(s['hypo_km'], 0, ' km'):>20}"
)
  L.append("")
  L.append("3. FASES (ARRIBOS)")
  L.append("-" * w)
  L.append(f"  {'Estación':<12}{'Fase':<5}{'Dist km':>8}{'Dist °':>7}{'Az °':>6}{'Llegada UTC
':>25}{'T.obs s':>9}"
           f"{'T.teo s':>9}{'Corr s':>8}{'Res s':>7}{'σ s':>6}{'ML':>6}  Método")
  for p in t["picks"]:
      L.append(f"  {p['station']:<12}{p['phase']:<5}{p['dist_km']:>8.1f}{p['dist_deg']:>7.2f}
{p['azimuth']:>6.0f}"
               f"{p['arrival_utc']:>25}{p['travel_obs_s']:>9.2f}{p['travel_theo_s']:>9.2f}{p[
'station_corr_s']:>8.2f}"
               f"{_f(p['residual_s'], 2):>7}{_f(p['sigma_s'], 2):>6}{_f(p['ml'], 1):>6}  {p['
method']}")
  L.append("")
  L.append("4. INTENSIDAD ESTIMADA (Mercalli Modificada)")
  L.append("-" * w)
  if t["intensity_poi"]:
      L.append(f"  {'Punto de interés':<30}{'Suelo':<16}{'Dist km':>8}{'MMI':>6}{'':>4}{'Lleg
ada S (UTC)':>26}{'S tras origen':>15}")
      for p in t["intensity_poi"]:
          L.append(f"  {p['name']:<30}{p['soil']:<16}{p['dist_km']:>8.0f}{p['mmi']:>6.1f}{p['
roman']:>4}{p['s_arrival_utc']:>26}{p['s_travel_s']:>13.1f} s")
  if t["nearest_cities"]:
      L.append("")
      L.append(f"  {'Ciudad cercana':<30}{'Dist km':>8}{'Epicentro al':>14}{'MMI (firme)':>13
}{'':>5}{'P s':>7}{'S s':>7}")
      for c in t["nearest_cities"]:
          L.append(f"  {c['name']:<30}{c['dist_km']:>8.0f}{c['epicenter_dir']:>14}{c['mmi_fir
m']:>13.1f}{c['roman']:>5}"
                   f"{c['p_arrival_s']:>7.1f}{c['s_arrival_s']:>7.1f}")
  L.append("")
  if t["derived"]:
      d = t["derived"]
      L.append("5. PARÁMETROS FÍSICOS DERIVADOS (aproximados, relaciones empíricas)")
      L.append("-" * w)
      L.append(f"  {'Momento sísmico M0':<40} {d['moment_Nm']:.2e} N·m   (base: {d['basis']})
")
      L.append(f"  {'Energía liberada':<40} {d['energy_J']:.2e} J   (≈ {d['tnt_equiv_t']:,.0f
} t de TNT)")
      L.append(f"  {'Longitud de ruptura':<40} {d['rupture_length_km']:.1f} km")
      L.append(f"  {'Área de ruptura':<40} {d['rupture_area_km2']:.0f} km²")
      L.append(f"  Referencias: {d['refs']}")
      L.append("")
  ts = t.get("tsunami")
  if ts:
      L.append("5b. TSUNAMI (MENSAJE INFORMATIVO, NO OFICIAL)")
      L.append("-" * w)
      L.append(f"  {ts['title']}")
      L.append(f"  {ts['text']}")
      L.append(f"  Criterio: magnitud {ts['M_used']} · profundidad {ts['depth_km']} km · tier
ra/mar por {ts['method']}"
               + (f" · costa a {ts['coast_km']} km" if ts.get("coast_km") is not None else ""
))
      for c in ts.get("cities", []):
```

```
            L.append(f"    {c['name']:<36}{c['dist_km']:>7} km   llegada orientativa ~{c['eta_min']} min")
        L.append(f"  {TSUNAMI_NOTE}")
        L.append("")
    L.append("6. COMPARACIÓN CON EL CATÁLOGO USGS")
    L.append("-" * w)
    u = t["usgs"]
    if u:
        L.append(f"  {'Evento USGS':<40} {u['id']} — M{_f(u['mag'])} {u['magType'] or ''} — {u['place'] or ''}")
        L.append(f"  {'Origen USGS':<40} {u['time_utc']} UTC   {_f(u['lat'], 3)}, {_f(u['lon'], 3)}   prof. {_f(u['depth_km'])} km")
        L.append(f"  {'Diferencias (monitor – USGS)':<40} epicentro {_f(u['diff_epicenter_km'], 1)} km · profundidad "
                 f"{_f(u['diff_depth_km'], 1)} km · magnitud {_f(u['diff_mag'], 2)} · hora {_f(u['diff_time_s'], 2)} s")
        if u.get("diff_mag_type"):
            L.append(f"  {'Comparación de magnitud':<40} {u['diff_mag_type']}")
        if u.get("elapsed_pub_min") is not None:
            L.append(f"  {'Primera publicación USGS':<40} {u['time_pub_utc']} UTC  ({u['elapsed_pub_min']:.1f} min tras el origen)")
        if u.get("alert_winner"):
            L.append(f"  {'Velocidad de primera alerta':<40} {u['alert_winner']}")
        if u["url"]:
            L.append(f"  {'Enlace':<40} {u['url']}")
    else:
        L.append("  Sin evento USGS correspondiente todavía (el catálogo se consulta cada 3 minutos).")
    L.append("")
    L.append("7. MÉTODO Y PARÁMETROS DE PROCESAMIENTO")
    L.append("-" * w)
    names = {"detector": "Detector", "picker": "Picador", "associator": "Asociador", "locator": "Localizador",
             "ml": "Magnitud local", "mwp": "Mwp", "mb": "mb", "mB": "mB_BB", "ms": "Ms_BB", "mw": "Mw", "intensity": "Intensidad",
             "depth_phases": "Fases profundidad", "tsunami": "Tsunami"}
    for k, v in t["method"].items():
        L.append(f"  {names.get(k, k):<18} {v}")
    L.append("")
    L.append("Glosario: T.obs = tiempo de viaje observado; T.teo = teórico del modelo; Corr = corrección aprendida de la estación;")
    L.append("Res = residuo tras la localización; σ = incertidumbre del arribo; Az = azimut del epicentro a la estación.")
    L.append("")
    L.append(t["disclaimer"])
    L.append("=" * w)
    return "\n".join(L) + "\n"


# ---------------------------------------------------------------- QuakeML 1.2
def to_quakeml(t):
    o, m = t["origin"], t["magnitude"]
    rid = f"smi:local/monitormundial/{t['event_id']}"
    x = ['<?xml version="1.0" encoding="UTF-8"?>',
         '<q:quakeml xmlns:q="http://quakeml.org/xmlns/quakeml/1.2" xmlns="http://quakeml.org/xmlns/bed/1.2">',
         f'  <eventParameters publicID="{rid}/parametros">',
         f'    <event publicID="{rid}">',
         f'      <preferredOriginID>{rid}/origen</preferredOriginID>']
    if m["preferred"] is not None:
        x.append(f'      <preferredMagnitudeID>{rid}/mag/{escape(m["preferred_type"])}</preferredMagnitudeID>')
    x += ['      <type>earthquake</type>',
          f'      <description><text>{escape(o["region"])}</text><type>region name</type></description>']
    for p in t["picks"]:
        net, _, sta = p["station"].partition(".")
        x += [f'      <pick publicID="{rid}/pick/{escape(p["key"])}/{p["phase"]}">',
              f'        <time><value>{_iso(p["arrival_epoch"])}</value>'
              + (f'<uncertainty>{p["sigma_s"]:.3f}</uncertainty>' if p["sigma_s"] is not None else "") + '</time>',
              f'        <waveformID networkCode="{escape(net)}" stationCode="{escape(sta)}"/>',
              f'        <phaseHint>{p["phase"]}</phaseHint>',
              f'        <evaluationMode>{"manual" if p["method"] == "manual" else "automatic"}</evaluationMode>',
              '      </pick>']
    x += [f'      <origin publicID="{rid}/origen">',
          f'        <time><value>{_iso(o["epoch"])}</value></time>',
          f'        <latitude><value>{o["lat"]:.5f}</value></latitude>',
          f'        <longitude><value>{o["lon"]:.5f}</value></longitude>',
          f'        <depth><value>{o["depth_km"] * 1000:.0f}</value>'
          + (f'<uncertainty>{o["err_depth_km"] * 1000:.0f}</uncertainty>' if o["err_depth_km"] is not None else "") + '</depth>',
          f'        <depthType>{"constrained by depth phases" if (t.get("depth_phases") or {}).get("accepted") else "from location"}</depthType>',
          f'        <methodID>smi:local/monitorsismico/localizador-malla-L1-L2</methodID>',
          f'        <earthModelID>smi:local/monitorsismico/modelo/{escape(o["velocity_model"])}</earthModelID>',
          '        <quality>',
          f'          <associatedPhaseCount>{len(t["picks"])}</associatedPhaseCount>',
          f'          <usedPhaseCount>{len(t["picks"])}</usedPhaseCount>',
          f'          <usedStationCount>{o["n_stations"]}</usedStationCount>',
          (f'          <standardError>{o["rms_s"]:.3f}</standardError>' if o["rms_s"] is not None else ""),
          (f'          <azimuthalGap>{o["gap_deg"]:.1f}</azimuthalGap>' if o["gap_deg"] is not
```

```
    None else ""),
          (f'          <minimumDistance>{o["min_dist_km"] / 111.195:.3f}</minimumDistance>' if o["min_dist_km"] else ""),
          (f'          <maximumDistance>{o["max_dist_km"] / 111.195:.3f}</maximumDistance>' if o["max_dist_km"] else ""),
          '        </quality>']
    if not o["err_open"] and o["err_h_major_km"] is not None:
        x += ['        <originUncertainty>',
              f'          <maxHorizontalUncertainty>{o["err_h_major_km"] * 1000:.0f}</maxHorizontalUncertainty>',
              f'          <minHorizontalUncertainty>{(o["err_h_minor_km"] or 0) * 1000:.0f}</minHorizontalUncertainty>',
              f'          <azimuthMaxHorizontalUncertainty>{(o["err_azimuth_deg"] or 0):.1f}</azimuthMaxHorizontalUncertainty>',
              '          <confidenceLevel>68</confidenceLevel>',
              '          <preferredDescription>uncertainty ellipse</preferredDescription>',
              '        </originUncertainty>']
    for p in t["picks"]:
        x += [f'        <arrival publicID="{rid}/arr/{escape(p["key"])}/{p["phase"]}">',
              f'          <pickID>{rid}/pick/{escape(p["key"])}/{p["phase"]}</pickID>',
              f'          <phase>{p["phase"]}</phase>',
              f'          <azimuth>{p["azimuth"]:.1f}</azimuth>',
              f'          <distance>{p["dist_deg"]:.4f}</distance>',
              (f'          <timeResidual>{p["residual_s"]:.3f}</timeResidual>' if p["residual_s"] is not None else ""),
              '        </arrival>']
    x += [f'        <evaluationMode>{"manual" if "manual" in t["status"] else "automatic"}</evaluationMode>',
          f'        <evaluationStatus>{"final" if t["status"].startswith("final") else "preliminary"}</evaluationStatus>',
          '        <creationInfo><agencyID>MonitorSismicoMundial</agencyID>'
          f'<creationTime>{_iso(time.time())}</creationTime><version>{VERSION}</version></creationInfo>',
          '      </origin>']
    mags = []
    if m["ml"] is not None:
        mags.append(("ML", m["ml"], m["ml_sd"], m["ml_n"]))
    if m["mwp"] is not None:
        mags.append(("Mwp", m["mwp"], m.get("mwp_sd"), m["mwp_n"]))
    for typ, k in (("mb", "mb"), ("mB", "mB"), ("Ms_BB", "ms")):
        if m.get(k) is not None:
            mags.append((typ, m[k], m.get(k + "_sd"), m.get(k + "_n")))
    if m.get("mw") is not None:
        mags.append(("Mw", m["mw"], m.get("mw_sd"), None))
    for typ, val, sd, n in mags:
        x += [f'      <magnitude publicID="{rid}/mag/{typ}">',
              f'        <mag><value>{val:.2f}</value>' + (f'<uncertainty>{sd:.2f}</uncertainty>' if sd is not None else "") + '</mag>',
              f'        <type>{typ}</type>',
              f'        <originID>{rid}/origen</originID>',
              (f'        <stationCount>{n}</stationCount>' if n else ""),
              '      </magnitude>']
    for p in t["picks"]:
        if p["phase"] != "P":
            continue
        net, _, sta = p["station"].partition(".")
        for typ, k in (("ML", "ml"), ("Mwp", "mwp"), ("mb", "mb"), ("mB", "mB"), ("Ms_BB", "ms")):
            if p.get(k) is None:
                continue
            x += [f'      <stationMagnitude publicID="{rid}/stamag/{escape(p["key"])}/{typ}">',
                  f'        <originID>{rid}/origen</originID>',
                  f'        <mag><value>{p[k]:.2f}</value></mag>',
                  f'        <type>{typ}</type>',
                  f'        <waveformID networkCode="{escape(net)}" stationCode="{escape(sta)}" />',
                  '      </stationMagnitude>']
    x += ['    </event>', '  </eventParameters>', '</q:quakeml>']
    return "\n".join(l for l in x if l) + "\n"


def to_json(t):
    return json.dumps(t, ensure_ascii=False, indent=2, default=lambda o: o.item() if hasattr(o, "item") else str(o))


# ---------------------------------------------------------------- guardado en disco
def save(engine, ev, folder):
    """Escribe TXT, JSON y QuakeML en datos/tickets/AAAA-MM/. Devuelve la ruta del TXT."""
    t = build(engine, ev)
    sub = os.path.join(folder, time.strftime("%Y-%m", time.gmtime(_t_ref(ev))))
    os.makedirs(sub, exist_ok=True)
    stem = os.path.join(sub, file_stem(ev))
    for ext, text in ((".txt", to_text(t)), (".json", to_json(t)), (".xml", to_quakeml(t))):
        tmp = stem + ext + ".tmp"
        with open(tmp, "w", encoding="utf-8") as f:
            f.write(text)
        os.replace(tmp, stem + ext)
    return stem + ".txt"
```

sismo/traveltime.py — 345 líneas

```
"""Tiempos de viaje en Tierra ESFÉRICA para todo el planeta (0–180°, profundidad 0–700 km).

Dos fuentes de tablas, con la misma interfaz tt(fase, distancia_km, profundidad_km):

  * "ak135" (o "iasp91") con TauP de ObsPy — es la referencia sismológica estándar. La tabla se
    calcula una sola vez (1–5 min) y queda en datos/tt_global_<modelo>_taup.npz.
  * Trazador de rayos propio, modelo iasp91 completo (corteza, manto, núcleo externo e interno),
    en capas esféricas finas con transformación de Tierra plana exacta capa por capa. No requiere
    ObsPy; sirve de respaldo y para la prueba sintética. Coincide con TauP dentro de ~1 s.

Fases:
  "P": primera llegada de tipo P: Pg/Pn/P directa, Pdiff (difractada en el núcleo) y PKP/PKIKP.
  "S": primera llegada de tipo S del manto: Sg/Sn/S directa y Sdiff (no incluye SKS).
"""
import os
import time

import numpy as np

R_E = 6371.0
KM_DEG = 111.195
DSTEP = 5.0                                  # km
DMAX = np.pi * R_E                           # 20015 km = 180°
ND = int(np.ceil(DMAX / DSTEP)) + 2
DEPTHS = np.array([0, 5, 10, 15, 20, 25, 30, 35, 40, 50, 60, 70, 80, 100, 120, 150, 200, 250, 300, 350,
                   400, 450, 500, 550, 600, 650, 700], float)
TABLE_VERSION = 3
DIFF_MAX_DEG = 25.0


# ---------------------------------------------------------------------------- modelo iasp91
def iasp91(r):
    """Velocidades (vp, vs) km/s del modelo iasp91 (Kennett y Engdahl, 1991) en el radio r (km)."""
    x = r / R_E
    if r < 1217.1:
        return 11.24094 - 4.09689 * x * x, 3.56454 - 3.45241 * x * x
    if r < 3482.0:
        return 10.03904 + 3.75665 * x - 13.67046 * x * x, 0.0
    if r < 3631.0:
        return 14.49470 - 1.47089 * x, 8.16616 - 1.58206 * x
    if r < 5611.0:
        return (25.1486 - 41.1538 * x + 51.9932 * x * x - 26.6083 * x ** 3,
                12.9303 - 21.2590 * x + 27.8988 * x * x - 14.1080 * x ** 3)
    if r < 5711.0:
        return 25.969838 - 16.934118 * x, 20.768902 - 16.531471 * x
    if r < 5961.0:
        return 29.38896 - 21.40656 * x, 17.70732 - 13.50652 * x
    if r < 6161.0:
        return 30.78765 - 23.25415 * x, 15.24213 - 11.08552 * x
    if r < 6251.0:
        return 25.41389 - 17.69722 * x, 5.75020 - 1.27420 * x
    if r < 6336.0:
        return 8.78541 - 0.74953 * x, 6.706231 - 2.248585 * x
    if r < 6351.0:
        return 6.50, 3.75
    return 5.80, 3.36


_DISC = [20.0, 35.0, 120.0, 210.0, 410.0, 660.0, 760.0, 2740.0, 2889.0, 5153.9]
CMB_DEPTH = 2889.0


def _layers():
    """Bordes de capa (profundidad, km): 1 km hasta 800 km; más abajo capas de 5 km o más delgadas,
    de modo que el espesor aplanado no pase de ~15 km (precisión cerca del centro). Hasta r = 60 km."""
    edges = list(np.arange(0, 801, 1.0))
    r = R_E - 800.0
    while r > 60.0:
        r -= min(5.0, 0.0025 * r)
        edges.append(R_E - r)
    e = np.array(sorted(set(np.round(edges, 4)) | set(_DISC)))
    # quitar bordes casi duplicados junto a las discontinuidades
    keep = np.concatenate([[True], np.diff(e) > 0.2])
    e = e[keep]
    for d in _DISC:
        if not np.any(np.abs(e - d) < 1e-6):
            e = np.sort(np.append(e, d))
    return e


# ---------------------------------------------------------------------------- trazador propio
def _trace_tables(phase):
    """Tabla (profundidad x distancia) de la primera llegada por trazado de rayos esférico.

    Transformación de Tierra plana (exacta) y capas con gradiente lineal de velocidad: dentro de cada
    capa el rayo es un arco de círculo y se integra de forma analítica, incluso el punto de retorno.
    Así no hay los picos falsos de los modelos de capas de velocidad constante."""
```

```
    edges = _layers()
    rt, rb = R_E - edges[:-1], R_E - edges[1:]
    k = 0 if phase == "P" else 1
    vt = np.array([iasp91(r - 1e-6)[k] for r in rt]) * R_E / rt      # velocidad aplanada en el techo
    vb = np.array([iasp91(r + 1e-6)[k] for r in rb]) * R_E / rb      # y en la base de cada capa
    h = R_E * np.log(rt / rb)
    if phase == "S":
        core = edges[:-1] >= CMB_DEPTH - 1e-6
        vt, vb = np.where(core, 1e6, vt), np.where(core, 1e6, vb)   # la S no entra al núcleo líquido
    nl = len(h)
    pmax = 1.0 / vt[0]
    plist = list(np.linspace(0, pmax, 12000, endpoint=False)[1:])
    vv = np.concatenate([vt[vt < 1e5], vb[vb < 1e5]])
    plist += list(1.0 / vv * (1 - 1e-7))
    # muestreo denso (geométrico) cerca de los rayos casi horizontales: en la superficie, en cada
    # discontinuidad y en cada profundidad de la fuente (definen las distancias cortas y Pn)
    eps = 10 ** np.linspace(-9, -1.5, 400)
    crit = {vt[0]}
    for d in list(_DISC) + list(DEPTHS):
        j = int(np.searchsorted(edges, d))
        if j < len(vt) and vt[j] < 1e5:
            crit.add(vt[j])
        if 0 < j <= len(vb) and vb[j - 1] < 1e5:
            crit.add(vb[j - 1])
    for vc in crit:
        plist += list((1.0 / vc) * (1 - eps))
    p_all = np.unique(np.array([p for p in plist if 0 < p < pmax]))
    zidx = [int(np.searchsorted(edges, z)) for z in DEPTHS]
    for z, kk in zip(DEPTHS, zidx):
        assert abs(edges[kk] - z) < 1e-6
    T = np.full((len(DEPTHS), ND), np.inf)
    kc = int(np.searchsorted(edges, CMB_DEPTH)) - 1   # última capa del manto
    p_graze = 1.0 / vb[kc]
    graze = [None] * len(DEPTHS)
    dv = vb - vt
    flat = np.abs(dv) < 1e-9
    g = np.where(flat, 1.0, dv / h)
    for c0 in range(0, len(p_all), 1500):
        p = p_all[c0:c0 + 1500][:, None]
        st, sb = p * vt, p * vb
        through = sb < 1                              # la capa se cruza completa
        enter = st < 1                                # el rayo entra a la capa
        ct_ = np.sqrt(np.clip(1 - st * st, 0, 1))
        cb_ = np.sqrt(np.clip(1 - sb * sb, 0, 1))
        # capa completa
        dx = np.where(through, p * h * (vt + vb) / np.maximum(ct_ + cb_, 1e-12), 0.0)
        with np.errstate(divide="ignore", invalid="ignore"):
            dt_grad = np.log((vb / vt) * (1 + ct_) / np.maximum(1 + cb_, 1e-12)) / g
        dt_flat = h / (vt * np.maximum(ct_, 1e-12))
        dt = np.where(through, np.where(flat, dt_flat, dt_grad), 0.0)
        # contribución de la capa donde el rayo regresa (gradiente positivo, p·vt < 1 <= p·vb)
        turn_here = enter & ~through & ~flat & (dv > 0)
        with np.errstate(divide="ignore", invalid="ignore"):
            vturn = 1.0 / p
            hx = np.where(turn_here, h * (vturn - vt) / np.where(flat, 1.0, dv), 0.0)
            dx_turn = np.where(turn_here, p * hx * (vt + vturn) / np.maximum(ct_, 1e-12), 0.0)
            dt_turn = np.where(turn_here, np.log((1 + ct_) / np.maximum(st, 1e-300)) / g, 0.0)
        cx = np.concatenate([np.zeros((len(p), 1)), np.cumsum(dx, axis=1)], axis=1)
        ctm = np.concatenate([np.zeros((len(p), 1)), np.cumsum(dt, axis=1)], axis=1)
        rows = np.arange(len(p))
        pp = p[:, 0]
        for di, zs in enumerate(zidx):
            up_ok = through[:, :zs].all(axis=1) if zs > 0 else np.ones(len(pp), bool)
            xu, tu = cx[:, zs], ctm[:, zs]
            stop = ~through[:, zs:]
            turns = stop.any(axis=1)
            kt = zs + np.argmax(stop, axis=1)
            ktc = np.minimum(kt, nl - 1)
            X = xu + 2 * (cx[rows, kt] - cx[:, zs] + dx_turn[rows, ktc])
            Tt = tu + 2 * (ctm[rows, kt] - ctm[:, zs] + dt_turn[rows, ktc])
            ok = up_ok & turns
            Xa = np.concatenate([xu[up_ok], X[ok]])
            Ta = np.concatenate([tu[up_ok], Tt[ok]])
            Pa = np.concatenate([pp[up_ok], pp[ok]])
            Xa = np.mod(Xa, 2 * DMAX)
            back = Xa > DMAX                          # más allá de 180°: se mide por el otro lado
            Xa = np.where(back, 2 * DMAX - Xa, Xa)
            Pa = np.where(back, -Pa, Pa)
            idx = np.rint(Xa / DSTEP).astype(int)
            m = (idx >= 0) & (idx < ND)
            Tc = Ta + Pa * (idx * DSTEP - Xa)         # corrección exacta al centro del intervalo (dT/dX = p)
            np.minimum.at(T[di], idx[m], Tc[m])
            j = np.nonzero(ok & (pp > p_graze))[0]    # rayo más cercano al rasante del núcleo
            if len(j):
                jj = j[np.argmin(pp[j])]
                if graze[di] is None or pp[jj] < graze[di][2]:
                    graze[di] = (float(X[jj]), float(Tt[jj]), float(pp[jj]))
    ii = np.arange(ND)
    for di in range(len(DEPTHS)):
```

```
        row = T[di]
        if graze[di] is not None:                    # onda difractada en el núcleo
            xg, tg, pg = graze[di]
            dif = tg + pg * (ii * DSTEP - xg)
            # Pdiff/Sdiff solo hasta 25° más allá del rayo rasante: más lejos se atenúan tanto
    que
            # la primera llegada observable es PKIKP (para P)
            zone = (ii * DSTEP > xg) & (ii * DSTEP < xg + DIFF_MAX_DEG * KM_DEG)
            row = np.where(zone, np.minimum(row, dif), row)
        T[di] = _fill(row)
    return T


def _fill(row):
    """Rellena huecos por interpolación y hace la primera llegada monótona con la distancia."""
    row = row.copy()
    fin = np.isfinite(row)
    ii = np.arange(len(row))
    if fin.sum() >= 2:
        row = np.interp(ii, ii[fin], row[fin])
    for i in range(len(row) - 2, -1, -1):
        if row[i] > row[i + 1]:
            row[i] = row[i + 1]
    return row


# ---------------------------------------------------------------------- TauP (ObsPy)
TAUP_PHASES = {"P": ["p", "P", "Pn", "Pg", "Pdiff", "PKP", "PKIKP", "PKiKP"],
               "S": ["s", "S", "Sn", "Sg", "Sdiff"]}


def _taup_tables(model, log=print):
    from obspy.taup import TauPyModel
    m = TauPyModel(model=model)
    deg = np.unique(np.concatenate([np.arange(0, 10, 0.1), np.arange(10, 180.01, 0.5)]))
    deg[0] = 0.01
    out = {}
    t0 = time.time()
    for ph in ("P", "S"):
        tab = np.full((len(DEPTHS), ND), np.inf)
        for di, z in enumerate(DEPTHS):
            vals = np.full(len(deg), np.inf)
            for k, d in enumerate(deg):
                try:
                    arr = m.get_travel_times(source_depth_in_km=float(z), distance_in_degree=fl
    oat(d),
                                             phase_list=TAUP_PHASES[ph])
                except Exception:
                    continue
                # Pdiff/Sdiff solo hasta 125° (más lejos la primera llegada observable es PKIKP
    )
                arr = [a for a in arr if not (a.name.endswith("diff") and d > 125)]
                if arr:
                    vals[k] = min(a.time for a in arr)
            fin = np.isfinite(vals)
            km = deg[fin] * KM_DEG
            row = np.full(ND, np.inf)
            grid = np.arange(ND) * DSTEP
            inside = grid <= km.max()
            row[inside] = np.interp(grid[inside], km, vals[fin])
            tab[di] = _fill(row)
            log(f"  TauP {model} {ph}: profundidad {z:.0f} km lista ({time.time() - t0:.0f} s)"
    )
        out[ph] = tab
    return out


# ---------------------------------------------------------------------- interfaz
def _depth_phase_tables(Ptab):
    """Fases de profundidad pP y sP (reflejadas en la superficie sobre el foco).

    tiempo(pP) = tiempo(P) + 2·∫₀ʰ √(1/v_P² − p²) dz
    tiempo(sP) = tiempo(P) + ∫₀ʰ [√(1/v_S² − p²) + √(1/v_P² − p²)] dz
    con p el parámetro de rayo de la P a esa distancia y profundidad (pendiente de la tabla) y
    la
    integral en coordenadas de Tierra plana (exacta para la Tierra esférica). Modelo iasp91 sob
    re el foco.
    Solo tienen sentido a distancias telesísmicas (≈ 20°–100°), donde se usan."""
    zs = np.arange(0.0, DEPTHS[-1] + 0.5, 0.5)
    rt, rb = R_E - zs[:-1], R_E - zs[1:]
    rm = (rt + rb) / 2
    hf = R_E * np.log(rt / rb)
    vp = np.array([iasp91(r)[0] for r in rm]) * R_E / rm
    vs = np.array([iasp91(r)[1] for r in rm]) * R_E / rm
    out = {"pP": np.empty_like(Ptab), "sP": np.empty_like(Ptab)}
    for di, h in enumerate(DEPTHS):
        row = Ptab[di]
        p = np.gradient(row, DSTEP)                      # s/km
        p = np.clip(p, 0, None)
        k = int(round(h / 0.5))
        if k == 0:
            out["pP"][di] = row
            out["sP"][di] = row
            continue
        qp = np.sqrt(np.clip(1.0 / vp[None, :k] ** 2 - p[:, None] ** 2, 0, None)) * hf[None, :k
```

```
    ]
        qs = np.sqrt(np.clip(1.0 / vs[None, :k] ** 2 - p[:, None] ** 2, 0, None)) * hf[None, :k
    ]
        out["pP"][di] = row + 2 * qp.sum(axis=1)
        out["sP"][di] = row + qp.sum(axis=1) + qs.sum(axis=1)
    return out


class TravelTimes:
    """model: 'ak135' o 'iasp91' (TauP si ObsPy está disponible) o 'iasp91-propio' (trazador in
    terno)."""

    def __init__(self, model="ak135", cache_dir="datos", log=print, allow_taup=True):
        os.makedirs(cache_dir, exist_ok=True)
        self.tab = None
        if model != "iasp91-propio" and allow_taup:
            path = os.path.join(cache_dir, f"tt_global_{model}_taup.npz")
            self.tab = self._load(path)
            if self.tab is None:
                try:
                    import obspy.taup  # noqa: F401
                    log(f"Calculando tablas de tiempos de viaje con TauP ({model}); solo la pri
    mera vez, puede tardar varios minutos…")
                    self.tab = _taup_tables(model, log)
                    np.savez_compressed(path, version=TABLE_VERSION, **self.tab)
                except ImportError:
                    log("TauP (ObsPy) no disponible: se usa el trazador propio con el modelo ia
    sp91.")
                except Exception as e:
                    log(f"TauP falló ({e}): se usa el trazador propio con el modelo iasp91.")
            if self.tab is not None:
                self.model, self.name = model, f"{model} (TauP, Tierra esférica)"
        if self.tab is None:
            path = os.path.join(cache_dir, "tt_global_iasp91_propio.npz")
            self.tab = self._load(path)
            if self.tab is None:
                log("Calculando tablas iasp91 con el trazador propio (unos segundos)…")
                self.tab = {"P": _trace_tables("P"), "S": _trace_tables("S")}
                np.savez_compressed(path, version=TABLE_VERSION, **self.tab)
            self.model, self.name = "iasp91", "iasp91 (trazador propio, Tierra esférica)"
        self.tab.update(_depth_phase_tables(self.tab["P"]))

    @staticmethod
    def _load(path):
        if not os.path.exists(path):
            return None
        try:
            d = np.load(path)
            if int(d["version"]) != TABLE_VERSION:
                return None
            return {"P": d["P"], "S": d["S"]}
        except Exception:
            return None

    def tt(self, phase, dist_km, z_km):
        """Tiempo de viaje (s). dist_km escalar o arreglo (km sobre la superficie); z_km escala
    r."""
        tab = self.tab[phase]
        z = min(DEPTHS[-1], max(0.0, float(z_km)))
        di = int(np.searchsorted(DEPTHS, z, side="right") - 1)
        di = min(max(di, 0), len(DEPTHS) - 2)
        fz = (z - DEPTHS[di]) / (DEPTHS[di + 1] - DEPTHS[di])
        x = np.clip(np.asarray(dist_km, float), 0, (ND - 2) * DSTEP) / DSTEP
        i = np.floor(x).astype(int)
        fx = x - i
        a = tab[di][i] * (1 - fx) + tab[di][i + 1] * fx
        b = tab[di + 1][i] * (1 - fx) + tab[di + 1][i + 1] * fx
        return a * (1 - fz) + b * fz

    def tt_zgrid(self, phase, dist_km, z_km):
        """Igual que tt() pero con z como arreglo (misma forma que dist_km)."""
        tab = self.tab[phase]
        z = np.clip(np.asarray(z_km, float), 0, DEPTHS[-1])
        di = np.clip(np.searchsorted(DEPTHS, z, side="right") - 1, 0, len(DEPTHS) - 2)
        fz = (z - DEPTHS[di]) / (DEPTHS[di + 1] - DEPTHS[di])
        x = np.clip(np.asarray(dist_km, float), 0, (ND - 2) * DSTEP) / DSTEP
        i = np.floor(x).astype(int)
        fx = x - i
        a = tab[di, i] * (1 - fx) + tab[di, i + 1] * fx
        b = tab[di + 1, i] * (1 - fx) + tab[di + 1, i + 1] * fx
        return a * (1 - fz) + b * fz

    def slowness(self, phase, dist_km, z_km):
        """Lentitud horizontal dT/dΔ (s/km) por diferencias finitas."""
        d = np.asarray(dist_km, float)
        return (self.tt(phase, d + 10, z_km) - self.tt(phase, np.maximum(0, d - 10), z_km)) / (
    d + 10 - np.maximum(0, d - 10))
```

sismo/tsunami.py — 188 líneas

```
"""Aviso INFORMATIVO de posible tsunami (Fase 3). No es un aviso oficial.

Criterios (basados en los umbrales de los mensajes del Centro de Alerta de Tsunamis del Pacífico, PTWC):
  - Solo sismos con epicentro en el mar o cerca de la costa (≤ 50 km tierra adentro) y profundidad ≤ 100 km.
  - Magnitud preferida redondeada a una décima (si la Mwp es ≥ 7.5 se toma la mayor de las dos, porque los
    sismos muy grandes tienden a subestimarse):
      6.5–7.0  sin amenaza de tsunami destructivo; posibles variaciones menores del nivel del mar cerca del epicentro
      7.1–7.5  posible tsunami peligroso en costas a menos de 300 km
      7.6–7.8  posible tsunami peligroso en costas a menos de 1000 km
      ≥ 7.9    posible tsunami peligroso en costas de toda la cuenca oceánica
¿Mar o tierra? Con el contorno de tierra de Natural Earth (1:50 millones; se descarga una vez y queda en
datos/tierra.geojson). Si no está disponible, con el nombre de la región Flinn-Engdahl (OFF COAST, SEA,
OCEAN, RIDGE, ISLANDS…). Si tampoco, se trata como "posiblemente en el mar" y el mensaje lo dice.
"""
import json
import os
import threading
import urllib.request

import numpy as np

from .locator import dist_km
from .places import CITIES, region_name

URLS = ["https://raw.githubusercontent.com/nvkelso/natural-earth-vector/master/geojson/ne_50m_land.geojson",
        "https://raw.githubusercontent.com/nvkelso/natural-earth-vector/master/geojson/ne_110m_land.geojson"]
MARINE_WORDS = ("OFF ", "COAST", "SEA", "OCEAN", "RIDGE", "RISE", "TRENCH", "ISLANDS", "ISLAND", "GULF", "STRAIT",
                "BASIN", "BAY", "CHANNEL", "PASSAGE", "FRACTURE", "ARCHIPELAGO")
LAND_PREFIX = ("CENTRAL ", "NORTHERN ", "SOUTHERN ", "EASTERN ", "WESTERN ")
SPEED_KMH = 700.0          # onda larga en mar abierto (≈ 4 km de profundidad); solo orientativo


class LandMask:
    def __init__(self, folder, log=print):
        self.path = os.path.join(folder, "tierra.geojson")
        self.log = log
        self.rings = None
        self._load()
        if self.rings is None:
            threading.Thread(target=self._download, daemon=True, name="tierra").start()

    def _load(self):
        try:
            with open(self.path, encoding="utf-8") as f:
                gj = json.load(f)
        except Exception:
            return
        rings = []
        for feat in gj.get("features", []):
            g = feat.get("geometry") or {}
            polys = [g["coordinates"]] if g.get("type") == "Polygon" else g.get("coordinates", []) if g.get("type") == "MultiPolygon" else []
            for poly in polys:
                if poly:
                    a = np.asarray(poly[0], float)              # anillo exterior
                    if len(a) >= 4:
                        rings.append((a[:, 0], a[:, 1], a[:, 0].min(), a[:, 0].max(), a[:, 1].min(), a[:, 1].max()))
        if rings:
            self.rings = rings
            self.vlon = np.concatenate([r[0] for r in rings])
            self.vlat = np.concatenate([r[1] for r in rings])

    def _download(self):
        for url in URLS:
            try:
                req = urllib.request.Request(url, headers={"User-Agent": "MonitorSismicoMundial"})
                data = urllib.request.urlopen(req, timeout=60).read()
                json.loads(data)
                tmp = self.path + ".tmp"
                with open(tmp, "wb") as f:
                    f.write(data)
                os.replace(tmp, self.path)
                self._load()
                self.log(f"Contorno de tierra (Natural Earth) descargado: {os.path.basename(url)}")
                return
            except Exception as e:
                last = e
        self.log(f"Contorno de tierra no disponible ({last}); el aviso de tsunami usará la región Flinn-Engdahl.")

    def on_land(self, lat, lon):
```

```
        if self.rings is None:
            return None
        inside = False
        for x, y, x0, x1, y0, y1 in self.rings:
            if not (x0 <= lon <= x1 and y0 <= lat <= y1):
                continue
            # rayo hacia el este: número de cruces impar = dentro
            xa, ya, xb, yb = x[:-1], y[:-1], x[1:], y[1:]
            c = ((ya > lat) != (yb > lat))
            xi = xa[c] + (lat - ya[c]) * (xb[c] - xa[c]) / (yb[c] - ya[c])
            if (np.count_nonzero(xi > lon) % 2) == 1:
                return True
        return inside

    def coast_km(self, lat, lon):
        if self.rings is None:
            return None
        return float(np.min(dist_km(self.vlat, self.vlon, lat, lon)))


def _marine_by_region(lat, lon):
    reg = (region_name(round(lat, 2), round(lon, 2)) or "").upper()
    if not reg:
        return None, ""
    if any(w in reg for w in MARINE_WORDS):
        return True, reg
    return False, reg


def assess(engine, ev, mask=None):
    """Evaluación informativa. Devuelve dict o None si la magnitud es menor de 6.5."""
    Mp = engine.mag_of(ev)
    M = Mp if Mp is not None else ev.get("mwp")
    if ev.get("mwp") is not None and ev["mwp"] >= 7.5:
        M = max(M or 0, ev["mwp"])                      # sismos muy grandes: las magnitudes ti
enden a quedarse cortas
    M = float(np.floor((M or 0) * 10 + 0.5) / 10)       # a una décima, como los boletines
    if M < 6.5:
        return None
    z = float(ev["z"])
    land = mask.on_land(ev["lat"], ev["lon"]) if mask else None
    coast = mask.coast_km(ev["lat"], ev["lon"]) if mask else None
    method = "contorno de tierra Natural Earth"
    if land is None:
        marine, reg = _marine_by_region(ev["lat"], ev["lon"])
        method = f"región Flinn-Engdahl ({reg.title()})" if reg else "sin datos de tierra/mar"
        land = None if marine is None else (not marine)
    out = {"M_used": round(M, 1), "depth_km": round(z), "on_land": land, "coast_km": None if co
ast is None else round(coast),
           "method": method, "radius_km": 0, "cities": []}
    inland_far = land is True and (coast is None or coast > 50)
    if z > 100:
        out.update(level=0, title="Sin amenaza de tsunami por la profundidad",
                   text=f"Sismo de {z:.0f} km de profundidad (> 100 km): no se espera tsunami."
)
        return out
    if inland_far:
        out.update(level=0, title="Sin amenaza de tsunami: epicentro en tierra",
                   text="El epicentro está en tierra" + (f", a unos {coast:.0f} km de la costa"
 if coast is not None else "")
                   + ": no se espera tsunami.")
        return out
    # Blindaje contra falsas alarmas de tsunami por artefactos de pocas estaciones:
    # Para alertar posible tsunami destructivo (M >= 7.1), se exige quórum de certeza física:
    n_mag_sta = (ev.get("msN") or 0) + (ev.get("mwpN") or 0) + (ev.get("mbN") or 0) + (ev.get("
mBN") or 0)
    has_quorum = (len(ev.get("picks", [])) >= 15 and n_mag_sta >= 4) or ((ev.get("mwpN") or 0)
>= 3 and (ev.get("mwp") or 0) >= 7.0)
    is_uncertain = ev.get("magWeak") or ev.get("magFew") or ev.get("early")

    if M >= 7.1 and (not has_quorum or is_uncertain):
        level, radius = 1, 0
        title = "Magnitud alta en mar — En verificación por la red"
        text = "Sismo registrado en zona marina con magnitud preliminar en proceso de consolida
ción. Sin confirmación de amenaza destructiva."
    elif M <= 7.0:
        level, radius = 1, 0
        title = "Sin amenaza de tsunami destructivo"
        text = "Posibles variaciones menores del nivel del mar cerca del epicentro."
    elif M <= 7.5:
        level, radius = 2, 300
        title = "POSIBLE TSUNAMI LOCAL"
        text = "Posibles olas peligrosas en costas a menos de 300 km del epicentro."
    elif M <= 7.8:
        level, radius = 3, 1000
        title = "POSIBLE TSUNAMI REGIONAL"
        text = "Posibles olas peligrosas en costas a menos de 1000 km del epicentro."
    else:
        level, radius = 4, 20000
        title = "POSIBLE TSUNAMI EN TODA LA CUENCA OCEÁNICA"
        text = ("Posibles olas peligrosas en costas de toda la cuenca oceánica. Con magnitudes
de este tamaño la "
                "magnitud automática puede estar subestimada.")
    if land is None:
        text += " No se pudo confirmar si el epicentro está en el mar; se trata como si lo estu
```

```
    viera."
    elif land:
        text += f" El epicentro está en tierra pero cerca de la costa ({coast:.0f} km)."
    out.update(level=level, title=title, text=text, radius_km=radius)
    if radius:
        lim = min(radius, 3000)
        cl = np.array([c[1] for c in CITIES])
        co = np.array([c[2] for c in CITIES])
        d = dist_km(cl, co, ev["lat"], ev["lon"])
        for i in np.argsort(d):
            if d[i] > lim or len(out["cities"]) >= 12:
                break
            name, la, lo = CITIES[int(i)]
            if mask is not None and mask.rings is not None and (mask.coast_km(la, lo) or 0) > 60:
                continue                                  # ciudad del interior
            out["cities"].append({"name": name, "dist_km": round(float(d[i])),
                                  "eta_min": round(float(d[i]) / SPEED_KMH * 60), "eta_note": "orientativo"})
    return out


DISCLAIMER = ("Mensaje INFORMATIVO de un monitor experimental, no es un aviso oficial. Para tsunamis consulte "
              "tsunami.gov (PTWC/NTWC) y la autoridad de su país (en México, el Centro de Alerta de Tsunamis de SEMAR). "
              "Los tiempos de llegada son orientativos (mar abierto, línea recta).")
```

sismo/usgsreport.py — 194 líneas

```
"""Reporte «Monitor vs USGS» (v0.5.4): cada sismo confirmado por el USGS con la magnitud preliminar y final del
monitor, la magnitud del USGS y los tiempos (preliminar, final y publicación del USGS, en minutos tras el sismo)."""
import csv
import io
import json
import os
import time

from .places import where_is

CACHE = "usgs_publicacion.json"


def _fmt(t):
    return time.strftime("%d-%m-%Y %H:%M", time.gmtime(t - 6 * 3600)) if t else "—"


def _usgs_info(ids, folder, log=None, max_fetch=80):
    """Hora de la primera publicación del USGS (producto más antiguo del evento) y magnitud vigente.
      Se guarda en datos/usgs_publicacion.json; los sismos de menos de 3 días se vuelven a consultar (el USGS
      suele actualizar la magnitud a Mww)."""
    from .netio import http_get
    path = os.path.join(folder, CACHE) if folder else None
    cache = {}
    if path and os.path.exists(path):
        try:
            with open(path, encoding="utf-8") as f:
                cache = json.load(f)
        except (OSError, ValueError):
            cache = {}
    now, n, changed = time.time(), 0, False
    for uid, t_org in ids:
        c = cache.get(uid)
        fresh = c and (now - c.get("consultado", 0) < 6 * 3600 or now - t_org > 3 * 86400)
        if fresh or n >= max_fetch:
            continue
        n += 1
        try:
            d = json.loads(http_get("https://earthquake.usgs.gov/fdsnws/event/1/query?format=geojson&eventid=" + uid, 15))
            P = d["properties"]
            times = [o.get("updateTime") for v in (P.get("products") or {}).values() for o in v if o.get("updateTime")]
            cache[uid] = {"pub": min(times) / 1000 if times else None, "mag": P.get("mag"), "magType": P.get("magType"),
                          "lugar": P.get("place"), "consultado": now}
            changed = True
        except Exception as e:
            if log:
                log(f"Reporte USGS: no se pudo consultar {uid} ({e})")
    if changed and path:
        try:
            with open(path, "w", encoding="utf-8") as f:
                json.dump(cache, f)
        except OSError:
            pass
    return cache


def collect(engine, folder=None, log=None):
    with engine.lock:
        evs = [dict(e) for e in engine.events if e.get("match")]
    evs.sort(key=lambda e: e["org"])
    info = _usgs_info([(e["match"]["id"], e["org"]) for e in evs], folder, log)
    rows = []
    for e in evs:
        m, u = e["match"], info.get(e["match"]["id"], {})
        org, cr, tf = e["org"], (e.get("tPrelim") or e.get("created")), e.get("tFinal")
        pub = u.get("pub")
        mu = u.get("mag", m.get("mag"))
        rows.append({
            "fecha": _fmt(org), "lugar": where_is(e["lat"], e["lon"]),
            "pre": e.get("magPre"), "preStr": e.get("magPreStr") or "—",
            "fin": e.get("mag"), "finType": e.get("magType") or "",
            "usgs": mu, "usgsType": u.get("magType", m.get("magType")) or "",
            "dM": (e["mag"] - mu) if (e.get("mag") is not None and mu is not None) else None,
            "tPre": (cr - org) / 60 if cr else None,
            "tFin": (tf - org) / 60 if tf else None,
            "tUsgs": (pub - org) / 60 if pub else None,
            "id": m["id"], "evento": e["id"],
        })
    for r in rows:
        r["preFin"] = (r["tFin"] - r["tPre"]) if (r["tFin"] is not None and r["tPre"] is not None) else None
        r["finUsgs"] = (r["tUsgs"] - r["tFin"]) if (r["tFin"] is not None and r["tUsgs"] is not None) else None
        r["preUsgs"] = (r["tUsgs"] - r["tPre"]) if (r["tPre"] is not None and r["tUsgs"] is not None) else None
    return {"generado": time.time(), "filas": rows, "resumen": summary(rows)}


```

```
def _med(v):
    v = sorted(x for x in v if x is not None)
    if not v:
        return None
    k = len(v) // 2
    return v[k] if len(v) % 2 else (v[k - 1] + v[k]) / 2


def summary(rows):
    d = [abs(r["dM"]) for r in rows if r["dM"] is not None]
    dp = [abs(r["pre"] - r["usgs"]) for r in rows if r["pre"] is not None and r["usgs"] is not None]
    pu = [r for r in rows if r["preUsgs"] is not None]
    fu = [r for r in rows if r["finUsgs"] is not None]
    return {
        "n": len(rows),
        "dM": sum(d) / len(d) if d else None, "dM03": sum(1 for x in d if x <= 0.3), "nM": len(d),
        "dMpre": sum(dp) / len(dp) if dp else None,
        "medPre": _med(r["tPre"] for r in rows), "medFin": _med(r["tFin"] for r in rows), "medUsgs": _med(r["tUsgs"] for r in rows),
        "preAntes": sum(1 for r in pu if r["preUsgs"] > 0), "nPre": len(pu),
        "finAntes": sum(1 for r in fu if r["finUsgs"] > 0), "nFin": len(fu),
    }


def _n(v, d=1):
    if v is None:
        return "—"
    t = f"{v:.{d}f}"
    return t[1:] if t.startswith("-") and float(t) == 0 else t


COLS = ["Fecha (CDMX)", "Epicentro", "Mag. preliminar monitor", "Mag. final monitor", "Tipo", "Mag. USGS", "Tipo USGS",
        "Final – USGS", "Preliminar (min)", "Final (min)", "Preliminar → final (min)", "USGS publicó (min)",
        "Final monitor → USGS (min)", "ID USGS"]


def to_csv(data):
    out = io.StringIO()
    out.write("")                       # para que Excel abra bien los acentos
    w = csv.writer(out)
    w.writerow(COLS)
    for r in data["filas"]:
        w.writerow([r["fecha"], r["lugar"], r["preStr"], _n(r["fin"]), r["finType"], _n(r["usgs"]), r["usgsType"], _n(r["dM"], 2),
                    _n(r["tPre"], 0), _n(r["tFin"], 0), _n(r["preFin"], 0), _n(r["tUsgs"], 0), _n(r["finUsgs"], 0), r["id"]])
    return out.getvalue()


def make_pdf(data, path):
    import matplotlib
    matplotlib.use("Agg")
    import matplotlib.pyplot as plt
    from matplotlib.backends.backend_pdf import PdfPages

    rows, s = data["filas"], data["resumen"]
    with PdfPages(path) as pdf:
        per = 26
        pages = max(1, (len(rows) + per - 1) // per)
        for pg in range(pages):
            fig = plt.figure(figsize=(11, 8.5))
            fig.text(0.04, 0.955, "Monitor Sísmico MUNDIAL — comparación Monitor vs USGS", fontsize=15, weight="bold")
            fig.text(0.04, 0.932, f"Generado: {_fmt(data['generado'])} (hora del centro de México) · página {pg + 1} de {pages} · "
                     "tiempos en minutos desde la hora de origen del sismo", fontsize=8, color="#555")
            if pg == 0:
                lines = [
                    f"Sismos confirmados por el USGS: {s['n']}",
                    f"Magnitud: |final − USGS| promedio {_n(s['dM'], 2)} ({s['dM03']} de {s['nM']} con diferencia ≤ 0.3) · "
                    f"|preliminar − USGS| promedio {_n(s['dMpre'], 2)}",
                    f"Tiempos (mediana): alerta preliminar del monitor {_n(s['medPre'], 0)} min · primera publicación USGS {_n(s['medUsgs'], 0)} min · "
                    f"solución final del monitor {_n(s['medFin'], 0)} min",
                    f"¿Quién alertó primero?  Monitor antes que el USGS: {s['preAntes']} de {s['nPre']} sismos ({100 * s['preAntes'] / max(1, s['nPre']):.0f} %)",
                ]
                for i, t in enumerate(lines):
                    fig.text(0.04, 0.90 - i * 0.022, t, fontsize=9, weight="bold" if i == 3 else None)
                top = 0.80
            else:
                top = 0.90
            ax = fig.add_axes([0.03, 0.05, 0.94, top - 0.05])
            ax.axis("off")
            chunk = rows[pg * per:(pg + 1) * per]
            cell = [[r["fecha"], r["lugar"][:38], r["preStr"], _n(r["fin"]) + " " + r["finType"], _n(r["usgs"]) + " " + r["usgsType"],
                     _n(r["dM"], 2), _n(r["tPre"], 0), _n(r["tUsgs"], 0), _n(r["preUsgs"], 0),
```

```
      _n(r["tFin"], 0), _n(r["finUsgs"], 0)]
166                   for r in chunk]
167           hdr = ["Fecha (CDMX)", "Epicentro", "Prelim.", "Final monitor", "USGS", "Final−USGS
      ", "Prelim.\n(min)", "USGS\npublicó",
168                  "Prelim.→\nUSGS", "Final\n(min)", "Final→\nUSGS"]
169           if cell:
170               t = ax.table(cellText=cell, colLabels=hdr, loc="upper center", cellLoc="center"
      ,
171                            colWidths=[0.1, 0.25, 0.055, 0.085, 0.08, 0.07, 0.06, 0.06, 0.065,
       0.065, 0.065])
172               t.auto_set_font_size(False)
173               t.set_fontsize(6.8)
174               t.scale(1, 1.2)
175               for (i, j), c in t.get_celld().items():
176                   if i == 0:
177                       c.set_facecolor("#1f3a5f")
178                       c.get_text().set_color("white")
179                       c.get_text().set_weight("bold")
180                   elif j == 2:
181                       c.get_text().set_color("#c00000")
182                       c.get_text().set_weight("bold")
183                   elif j == 3:
184                       c.get_text().set_color("#1f5fbf")
185                       c.get_text().set_weight("bold")
186                   elif j == 8 and cell[i - 1][8] != "—":
187                       c.set_facecolor("#e6f4ea" if float(cell[i - 1][8]) > 0 else "#fdecea")
188           else:
189               ax.text(0.5, 0.5, "Todavía no hay sismos confirmados por el USGS.", ha="center"
      , fontsize=12)
190           fig.text(0.04, 0.025, "Verde en «Prelim.→USGS»: el monitor emitió la alerta prelimi
      nar antes que el USGS · rojo: el USGS publicó primero. "
191                    "La hora del USGS es la de la primera versión de su catálogo; la magnitud
      USGS es la vigente.",
192                    fontsize=6.5, color="#555")
193           pdf.savefig(fig)
194           plt.close(fig)
```

sismo/zones.py — 25 líneas

```
"""Zonas de alerta: cada zona tiene sus propios umbrales (config.json → alertas.zonas, en orden
  ).

Tipos: "caja" (lat0, lat1, lon0, lon1; si lon0 > lon1 la caja cruza el meridiano 180),
"circulo" (lat, lon, radio_km) y "todo" (cualquier lugar; conviene ponerla al final).
"""
from .locator import dist_km


def contains(z, lat, lon):
    t = z.get("tipo", "todo")
    if t == "caja":
        if not (z["lat0"] <= lat <= z["lat1"]):
            return False
        lon0, lon1 = z["lon0"], z["lon1"]
        return lon0 <= lon <= lon1 if lon0 <= lon1 else (lon >= lon0 or lon <= lon1)
    if t == "circulo":
        return float(dist_km(z["lat"], z["lon"], lat, lon)) <= z["radio_km"]
    return True


def zone_of(zones, lat, lon):
    for z in zones:
        if z.get("activa", True) and contains(z, lat, lon):
            return z
    return None
```