```python
# modules/analysis.py

import geopandas as gpd
import numpy as np
import pandas as pd
import pymannkendall as mk
import rasterio
import requests
import streamlit as st
from rasterio.mask import mask
from rasterio.warp import Resampling, reproject
from scipy import stats
from scipy.stats import loglaplace, norm

from modules.config import Config


@st.cache_data
def calculate_spi(df, col=Config.PRECIPITATION_COL, window=12):
    """
    Calcula el Índice Estandarizado de Precipitación (SPI) usando transformación logarítmica.
    """
    if df is None or df.empty:
        return df

    # Asegurar que sea un DataFrame con índice fecha si viene como Serie
    if isinstance(df, pd.Series):
        df = df.to_frame(name=col)

    df_spi = df.copy().sort_index()  # Asumimos índice fecha

    # Si el índice no es datetime, intentamos usar la columna de configuración
    if (
        not isinstance(df_spi.index, pd.DatetimeIndex)
        and Config.DATE_COL in df_spi.columns
    ):
        df_spi = df_spi.set_index(Config.DATE_COL).sort_index()

    # 1. Calcular acumulado móvil (Rolling Sum)
    # min_periods=window asegura que no calcule si falta un mes en la ventana
    df_spi["rolling_precip"] = (
        df_spi[col].rolling(window=window, min_periods=window).sum()
    )

    # 2. Calcular SPI (Aproximación Z-Score sobre logaritmos)
    try:
```

```
        # Filtrar ceros y nulos para el logaritmo
        valid_mask = (df_spi["rolling_precip"] > 0) & (df_spi["rolling_precip"].notna())

        # Transformación Logarítmica para normalizar
        log_precip = np.log(df_spi.loc[valid_mask, "rolling_precip"])

        mean = log_precip.mean()
        std = log_precip.std()

        df_spi["spi"] = np.nan  # Inicializar

        if std == 0:
            df_spi["spi"] = 0
        else:
            # Calcular Z-score
            df_spi.loc[valid_mask, "spi"] = (log_precip - mean) / std

            # Manejo de Ceros (Sequía Extrema)
            # Si llovió 0 en X meses, el SPI debe ser muy bajo
            min_spi = df_spi["spi"].min()
            if pd.isna(min_spi):
                min_spi = -3.0
            # Asignar mínimo a los valores que eran 0 originalmente
            df_spi.loc[
                ~valid_mask
                & df_spi["rolling_precip"].notna()
                & (df_spi["rolling_precip"] == 0),
                "spi",
            ] = min_spi

    except Exception:
        # st.warning(f"Error cálculo SPI: {e}") # Debug silencioso
        df_spi["spi"] = np.nan

    # Retornar solo la serie SPI
    return df_spi["spi"]


@st.cache_data
def calculate_spei(precip_series, et_series, window=12):
    """
    Calcula el SPEI usando la distribución Log-Laplace.
    CORREGIDO: Ahora acepta el argumento 'window' para ser compatible con el visualizador.
    """
    scale = int(window)  # Mapeo de argumento window -> scale
```

```
# Validación
if (
    precip_series is None
    or et_series is None
    or precip_series.empty
    or et_series.empty
):
    return pd.Series(dtype=float)

# Alineación
df = pd.DataFrame({"precip": precip_series, "et": et_series}).sort_index()
df = df.asfreq("MS")

# Limpieza
df.dropna(subset=["precip"], inplace=True)
df["et"] = df["et"].fillna(method="ffill").fillna(method="bfill")
df.dropna(subset=["et"], inplace=True)

# --- AJUSTE INTELIGENTE DE UNIDADES ---
# Si 'et' parece ser Temperatura (< 40 promedio), lo convertimos a ET aprox.
if df["et"].mean() < 40:
    df["et"] = df["et"] * 4.5  # Aprox Thornthwaite simple para trópico

if len(df) < scale * 2:
    return pd.Series(dtype=float)

# Balance (D)
water_balance = df["precip"] - df["et"]

# Acumulación
rolling_balance = water_balance.rolling(window=scale, min_periods=scale).sum()

# Ajuste Log-Laplace
data_for_fit = rolling_balance.dropna()
data_for_fit = data_for_fit[np.isfinite(data_for_fit)]

spei = pd.Series(np.nan, index=rolling_balance.index)

if not data_for_fit.empty and len(data_for_fit.unique()) > 1:
    try:
        # Ajuste de parámetros
        params = loglaplace.fit(
            data_for_fit,
            floc=data_for_fit.min() - 1e-5 if data_for_fit.min() <= 0 else 0,
        )
        # CDF
```

```
            cdf = loglaplace.cdf(rolling_balance.dropna(), *params)
            cdf_series = pd.Series(cdf, index=rolling_balance.dropna().index)
            # Clipping y Z-Score
            cdf_clipped = np.clip(cdf_series.values, 1e-7, 1 - 1e-7)
            spei.loc[cdf_series.index] = norm.ppf(cdf_clipped)
        except Exception:
            pass  # Retorna NaN si falla el ajuste matemático

    spei.replace([np.inf, -np.inf], np.nan, inplace=True)
    return spei


def calculate_monthly_anomalies(df_monthly, df_long_full):
    """Calcula anomalías mensuales."""
    if df_monthly.empty:
        return pd.DataFrame()
    climatology = (
        df_long_full.groupby(Config.MONTH_COL)[Config.PRECIPITATION_COL]
        .mean()
        .reset_index()
    )
    climatology = climatology.rename(columns={Config.PRECIPITATION_COL: "mean_ppt"})
    merged = pd.merge(df_monthly, climatology, on=Config.MONTH_COL, how="left")
    merged["anomalia"] = merged[Config.PRECIPITATION_COL] - merged["climatology_ppt"]
    return merged


def estimate_temperature(altitude_m):
    """Estimación de temperatura basada en gradiente térmico vertical (Andes)."""
    if pd.isna(altitude_m):
        return 25.0
    # Gradiente típico: 28°C a nivel del mar, disminuye 0.6°C por cada 100m
    temp = 28.0 - (0.006 * altitude_m)
    return max(temp, 1.0)  # Evitar temperaturas bajo cero extremas en modelo simple


def calculate_water_balance_turc(precip_mm, temp_c):
    """
    Estima Evapotranspiración Real (ETR) y Escorrentía (Q) usando la fórmula de Turc.
    """
    if pd.isna(precip_mm) or pd.isna(temp_c):
        return 0, 0

    # Fórmula de Turc para ETR
    L = 300 + 25 * temp_c + 0.05 * (temp_c**3)
    etr = precip_mm / np.sqrt(0.9 + (precip_mm / L) ** 2)
```

```
    # Limitar ETR a la precipitación (no puede evaporarse más de lo que llueve en este modelo simple)
    etr = min(etr, precip_mm)

    # Escorrentía = Precipitación - ETR
    q = precip_mm - etr
    return etr, q


def classify_holdridge_point(precip_mm, altitude_m):
    """Clasificación simple de Holdridge."""
    if pd.isna(precip_mm) or pd.isna(altitude_m):
        return "N/A"
    if altitude_m < 1000:
        piso = "Tropical"
    elif altitude_m < 2000:
        piso = "Premontano"
    elif altitude_m < 3000:
        piso = "Montano Bajo"
    elif altitude_m < 3500:
        piso = "Montano-Paramo"
    else:
        piso = "Montano"

    if precip_mm < 1000:
        prov = "Bosque Seco"
    elif precip_mm < 2000:
        prov = "Bosque Húmedo"
    elif precip_mm < 4000:
        prov = "Bosque Muy Húmedo"
    else:
        prov = "Bosque Pluvial"

    return f"{prov} {piso}"


def calculate_percentiles_and_extremes(df_long, station_name, p_lower=10, p_upper=90):
    """
    Calcula umbrales de percentiles y clasifica eventos extremos para una estación.
    """
    df_station_full = df_long[df_long[Config.STATION_NAME_COL] == station_name].copy()
    df_thresholds = (
        df_station_full.groupby(Config.MONTH_COL)[Config.PRECIPITATION_COL]
        .agg(
            p_lower=lambda x: np.nanpercentile(x.dropna(), p_lower),
            p_upper=lambda x: np.nanpercentile(x.dropna(), p_upper),
```

```
            mean_monthly="mean",
        )
        .reset_index()
    )
    df_station_extremes = pd.merge(
        df_station_full, df_thresholds, on=Config.MONTH_COL, how="left"
    )
    df_station_extremes["event_type"] = "Normal"
    is_dry = (
        df_station_extremes[Config.PRECIPITATION_COL] < df_station_extremes["p_lower"]
    )
    df_station_extremes.loc[is_dry, "event_type"] = f"Sequía Extrema (< P{p_lower}%)"
    is_wet = (
        df_station_extremes[Config.PRECIPITATION_COL] > df_station_extremes["p_upper"]
    )
    df_station_extremes.loc[is_wet, "event_type"] = f"Húmedo Extremo (> P{p_upper}%)"
    return df_station_extremes.dropna(subset=[Config.PRECIPITATION_COL]), df_thresholds


@st.cache_data
def calculate_climatological_anomalies(
    _df_monthly_filtered, _df_long, baseline_start, baseline_end
):
    """
    Calcula las anomalías mensuales con respecto a un período base climatológico fijo.
    """
    df_monthly_filtered = _df_monthly_filtered.copy()
    df_long = _df_long.copy()

    baseline_df = df_long[
        (df_long[Config.YEAR_COL] >= baseline_start)
        & (df_long[Config.YEAR_COL] <= baseline_end)
    ]

    df_climatology = (
        baseline_df.groupby([Config.STATION_NAME_COL, Config.MONTH_COL])[
            Config.PRECIPITATION_COL
        ]
        .mean()
        .reset_index()
        .rename(columns={Config.PRECIPITATION_COL: "precip_promedio_climatologico"})
    )

    df_anomalias = pd.merge(
        df_monthly_filtered,
        df_climatology,
```

```
        on=[Config.STATION_NAME_COL, Config.MONTH_COL],
        how="left",
    )

    df_anomalias["anomalia"] = (
        df_anomalias[Config.PRECIPITATION_COL]
        - df_anomalias["precip_promedio_climatologico"]
    )
    return df_anomalias


@st.cache_data
def analyze_events(index_series, threshold, event_type="drought"):
    """
    Identifica y caracteriza eventos de sequía o humedad en una serie de tiempo de índices.
    """
    if event_type == "drought":
        is_event = index_series < threshold
    else:  # 'wet'
        is_event = index_series > threshold

    event_blocks = (is_event.diff() != 0).cumsum()
    active_events = is_event[is_event]
    if active_events.empty:
        return pd.DataFrame()

    events = []
    for event_id, group in active_events.groupby(event_blocks):
        start_date = group.index.min()
        end_date = group.index.max()
        duration = len(group)

        event_values = index_series.loc[start_date:end_date]

        magnitude = event_values.sum()
        intensity = event_values.mean()
        peak = event_values.min() if event_type == "drought" else event_values.max()

        events.append(
            {
                "Fecha Inicio": start_date,
                "Fecha Fin": end_date,
                "Duración (meses)": duration,
                "Magnitud": magnitude,
                "Intensidad": intensity,
                "Pico": peak,
```

```
            }
        )

    if not events:
        return pd.DataFrame()

    events_df = pd.DataFrame(events)
    return events_df.sort_values(by="Fecha Inicio").reset_index(drop=True)


@st.cache_data
def calculate_basin_stats(
    _gdf_stations, _gdf_basins, _df_monthly, basin_name, basin_col_name
):
    """
    Calcula estadísticas de precipitación para todas las estaciones dentro de una cuenca específica.
    """
    if _gdf_basins is None or basin_col_name not in _gdf_basins.columns:
        return (
            pd.DataFrame(),
            [],
            "El GeoDataFrame de cuencas o la columna de nombres no es válida.",
        )

    target_basin = _gdf_basins[_gdf_basins[basin_col_name] == basin_name]
    if target_basin.empty:
        return pd.DataFrame(), [], f"No se encontró la cuenca llamada '{basin_name}'."

    stations_in_basin = gpd.sjoin(
        _gdf_stations, target_basin, how="inner", predicate="within"
    )
    station_names_in_basin = (
        stations_in_basin[Config.STATION_NAME_COL].unique().tolist()
    )

    if not station_names_in_basin:
        return pd.DataFrame(), [], None

    df_basin_precip = _df_monthly[
        _df_monthly[Config.STATION_NAME_COL].isin(station_names_in_basin)
    ]
    if df_basin_precip.empty:
        return (
            pd.DataFrame(),
            station_names_in_basin,
            "No hay datos de precipitación para las estaciones en esta cuenca.",
```

```
        )

    stats = df_basin_precip[Config.PRECIPITATION_COL].describe().reset_index()
    stats.columns = ["Métrica", "Valor"]
    stats["Valor"] = stats["Valor"].round(2)

    return stats, station_names_in_basin, None


@st.cache_data
def get_mean_altitude_for_basin(_basin_geometry):
    """
    Calcula la altitud media de una cuenca consultando la API de Open-Elevation.
    """
    try:
        # Simplificamos la geometría para reducir el tamaño de la consulta
        simplified_geom = _basin_geometry.simplify(tolerance=0.01)

        # Obtenemos los puntos del contorno exterior del polígono
        exterior_coords = list(simplified_geom.exterior.coords)

        # Creamos la estructura de datos para la API
        locations = [
            {"latitude": lat, "longitude": lon} for lon, lat in exterior_coords
        ]

        # Hacemos la llamada a la API de Open-Elevation
        response = requests.post(
            "https://api.open-elevation.com/api/v1/lookup",
            json={"locations": locations},
        )
        response.raise_for_status()

        results = response.json()["results"]
        elevations = [res["elevation"] for res in results]

        # Calculamos la media de las elevaciones obtenidas
        mean_altitude = np.mean(elevations)

        return mean_altitude, None
    except Exception as e:
        error_message = f"No se pudo obtener la altitud de la cuenca: {e}"
        st.warning(error_message)
        return None, error_message
```

```
def calculate_hydrological_balance(precip_mm, alt_m, gdf_basin, delta_temp_c=0):
    """
    Balance Hídrico (Turc). Corregido variable 'q'.
    """
    if precip_mm is None or np.isnan(precip_mm) or precip_mm <= 0:
        return {
            "P": 0,
            "ET": 0,
            "Q": 0,
            "Q_mm": 0,
            "Vol": 0,
            "Q_m3_año": 0,
            "Alt": 0,
            "Area": 0,
        }

    t = estimate_temperature(alt_m) + delta_temp_c

    # Fórmula Turc
    L = 300 + 25 * t + 0.05 * t**3
    denom = np.sqrt(0.9 + (precip_mm / L) ** 2)
    etr = precip_mm / denom if denom != 0 else precip_mm
    etr = min(etr, precip_mm)

    # CORRECCIÓN: Definir 'q' explícitamente
    q = precip_mm - etr

    morph = calculate_morphometry(gdf_basin)
    area = morph["area_km2"]

    # Volumen
    vol = (q * area) / 1000.0  # Mm3

    return {
        "P": precip_mm,
        "P_media_anual_mm": precip_mm,
        "ET": etr,
        "ET_media_anual_mm": etr,
        "Q": q,
        "Q_mm": q,  # Ahora 'q' está definido
        "Vol": vol,
        "Q_m3_año": vol,
        "Alt": alt_m,
        "Area": area,
    }
```

```
def calculate_morphometry(gdf_basin):
    """Calcula métricas morfométricas básicas."""
    if gdf_basin is None or gdf_basin.empty:
        return {
            k: 0
            for k in [
                "area_km2",
                "perimetro_km",
                "indice_forma",
                "alt_max",
                "alt_min",
                "alt_med",
                "pendiente_prom",
            ]
        }

    # Proyectar a metros (EPSG:3116) para cálculos de área/distancia
    try:
        gdf_proj = gdf_basin.to_crs(epsg=3116)
    except:
        gdf_proj = gdf_basin  # Fallback si falla la proyección

    area_km2 = gdf_proj.area.sum() / 1e6
    perimetro_km = gdf_proj.length.sum() / 1000

    # Índice de compacidad (Gravelius) -> Kc = 0.28 * P / raiz(A)
    indice_forma = (0.28 * perimetro_km) / np.sqrt(area_km2) if area_km2 > 0 else 0

    # --- ESTIMACIÓN DE ALTITUDES ---
    # Como no estamos procesando el DEM pixel por pixel aquí (sería muy lento),
    # usamos valores típicos o los calculados previamente si existen.
    # Si tuviéramos el DEM cargado en memoria, haríamos zonal statistics.
    # Valores simulados coherentes para la región (a falta de Zonal Stats):
    alt_max = 2800
    alt_min = 800
    alt_med = (alt_max + alt_min) / 2

    # Estimación de Pendiente Global (%) (Relief Ratio simplificado)
    # Pendiente ~ (Desnivel / Longitud Característica)
    longitud_aprox_km = np.sqrt(area_km2)  # Asumiendo forma cuadrada
    if longitud_aprox_km > 0:
        pendiente_prom = ((alt_max - alt_min) / (longitud_aprox_km * 1000)) * 100
    else:
        pendiente_prom = 0
```

```
    return {
        "area_km2": area_km2,
        "perimetro_km": perimetro_km,
        "indice_forma": indice_forma,
        "alt_max_m": alt_max,
        "alt_min_m": alt_min,
        "alt_prom_m": alt_med,
        "pendiente_prom": pendiente_prom,
    }


def calculate_hypsometric_curve(gdf_basin, dem_path=None):
    """
    Genera la curva hipsométrica REAL leyendo el DEM y recortando por la cuenca.
    """
    if gdf_basin is None or gdf_basin.empty:
        return None

    elevations = []

    # 1. INTENTAR LEER DEL DEM SI EXISTE
    if dem_path:
        try:
            import rasterio
            from rasterio.mask import mask

            with rasterio.open(dem_path) as src:
                # Asegurar que la geometría de la cuenca esté en el mismo CRS que el DEM
                if gdf_basin.crs != src.crs:
                    gdf_basin_proj = gdf_basin.to_crs(src.crs)
                else:
                    gdf_basin_proj = gdf_basin

                # Recortar el DEM usando la geometría de la cuenca
                out_image, out_transform = mask(
                    src,
                    gdf_basin_proj.geometry,
                    crop=True,
                    nodata=src.nodata
                )

                # Aplanar el array y eliminar valores NoData
                data = out_image[0] # Banda 1
                elevations = data[data != src.nodata]

                # Filtrar valores absurdos (ej. errores de medición < -100 o > 6000 en Colombia)
```

```
            elevations = elevations[(elevations > -100) & (elevations < 6000)]

    except Exception as e:
        print(f"Error leyendo DEM: {e}")
        elevations = []

# 2. SI FALLA EL DEM O NO HAY RUTA, USAR MODELO TEÓRICO (FALLBACK)
if len(elevations) == 0:
    # Tu lógica original de simulación S-Curve como respaldo
    morph = calculate_morphometry(gdf_basin)
    min_z, max_z = morph["alt_min_m"], morph["alt_max_m"]
    n_points = 1000
    # Simulación simple lineal si no hay DEM
    elevations = np.linspace(min_z, max_z, n_points)

# 3. CALCULAR CURVA (Histograma acumulado)
# Ordenar de mayor a menor (para hipsometría: área sobre cota X)
elevations_sorted = np.sort(elevations)[::-1]
n_pixels = len(elevations_sorted)

# Eje X: Porcentaje del área (0 a 100)
area_percent = np.arange(1, n_pixels + 1) / n_pixels * 100

# Reducir resolución para graficar (no necesitamos 1 millón de puntos)
if n_pixels > 200:
    indices = np.linspace(0, n_pixels - 1, 200, dtype=int)
    elevations_plot = elevations_sorted[indices]
    area_plot = area_percent[indices]
else:
    elevations_plot = elevations_sorted
    area_plot = area_percent

# 4. Ajuste de Ecuación (Polinomio Grado 3)
try:
    coeffs = np.polyfit(area_plot, elevations_plot, 3)
    poly_model = np.poly1d(coeffs)

    eq_str = (
        f"H = {coeffs[0]:.4f}A³ "
        f"{'+' if coeffs[1]>=0 else '-'} {abs(coeffs[1]):.3f}A² "
        f"{'+' if coeffs[2]>=0 else '-'} {abs(coeffs[2]):.2f}A "
        f"{'+' if coeffs[3]>=0 else '-'} {abs(coeffs[3]):.0f}"
    )
except:
    eq_str = "N/A"
    poly_model = lambda x: x
```

```
    return {
        "elevations": elevations_plot,  # Eje Y
        "area_percent": area_plot,      # Eje X
        "equation": eq_str,
        "poly_model": poly_model,
        "source": "DEM Real" if dem_path and len(elevations) > 0 else "Simulado"
    }

def calculate_all_station_trends(df_anual, gdf_stations):
    """
    Calcula la tendencia de Mann-Kendall y la Pendiente de Sen para todas las
    estaciones con datos suficientes.
    """
    trend_results = []
    stations_with_data = df_anual[Config.STATION_NAME_COL].unique()

    for station in stations_with_data:
        station_data = df_anual[df_anual[Config.STATION_NAME_COL] == station].dropna(
            subset=[Config.PRECIPITATION_COL]
        )
        # Asegurar un mínimo de 10 años para una tendencia más robusta
        if len(station_data) >= 10:
            try:
                mk_result = mk.original_test(station_data[Config.PRECIPITATION_COL])
                trend_results.append(
                    {
                        Config.STATION_NAME_COL: station,
                        "slope_sen": mk_result.slope,
                        "p_value": mk_result.p,
                    }
                )
            except Exception:
                continue  # Ignora estaciones donde la prueba pueda fallar

    if not trend_results:
        return gpd.GeoDataFrame()

    df_trends = pd.DataFrame(trend_results)

    # Unir los resultados con la información geoespacial de las estaciones
    gdf_trends = pd.merge(gdf_stations, df_trends, on=Config.STATION_NAME_COL)

    return gpd.GeoDataFrame(gdf_trends)
```

```
def generate_life_zone_raster(dem_path, ppt_path, mask_geom=None, downscale_factor=1):
    """
    Genera una matriz de Zonas de Vida cruzando Raster de Altura (DEM) y Raster de Precipitación.
    """
    try:
        # 1. Abrir DEM (Maestro)
        with rasterio.open(dem_path) as src_dem:
            # Calcular nueva resolución (Downscale para rendimiento)
            dst_height = src_dem.height // downscale_factor
            dst_width = src_dem.width // downscale_factor
            dst_transform = src_dem.transform * src_dem.transform.scale(
                (src_dem.width / dst_width), (src_dem.height / dst_height)
            )

            # Leer y redimensionar DEM
            dem_arr = np.empty((dst_height, dst_width), dtype=np.float32)
            reproject(
                source=rasterio.band(src_dem, 1),
                destination=dem_arr,
                src_transform=src_dem.transform,
                src_crs=src_dem.crs,
                dst_transform=dst_transform,
                dst_crs=src_dem.crs,
                resampling=Resampling.bilinear,
            )

            # Guardar perfil para aplicar máscara después
            profile = src_dem.profile.copy()
            profile.update(
                {
                    "height": dst_height,
                    "width": dst_width,
                    "transform": dst_transform,
                    "dtype": "int16",
                    "nodata": 0,
                }
            )

        # 2. Abrir PPT y alinearlo al DEM
        with rasterio.open(ppt_path) as src_ppt:
            ppt_arr = np.empty((dst_height, dst_width), dtype=np.float32)
            reproject(
                source=rasterio.band(src_ppt, 1),
                destination=ppt_arr,
                src_transform=src_ppt.transform,
                src_crs=src_ppt.crs,
```

```
        dst_transform=dst_transform,  # Usamos la rejilla del DEM
        dst_crs=src_dem.crs,  # Usamos el CRS del DEM
        resampling=Resampling.bilinear,
    )

# 3. Clasificación Vectorizada (Holdridge Simplificado)
# Inicializar con 0 (Sin Datos)
lz_arr = np.zeros_like(dem_arr, dtype=np.int16)

# Máscara de datos válidos
valid = (dem_arr > -100) & (ppt_arr > 0)

if np.any(valid):
    alt = dem_arr[valid]
    ppt = ppt_arr[valid]
    zones = np.zeros_like(alt, dtype=np.int16)

    # Lógica Holdridge (IDs arbitrarios para visualización)
    # 1: Bosque Seco Tropical, 2: Bosque Húmedo Premontano, etc.
    # Esta lógica debe coincidir con tu leyenda en visualizer

    # Tropical (<1000m)
    mask_T = alt < 1000
    zones[mask_T & (ppt < 1000)] = 1  # b-s-T
    zones[mask_T & (ppt >= 1000) & (ppt < 2000)] = 2  # b-h-T
    zones[mask_T & (ppt >= 2000) & (ppt < 4000)] = 3  # b-mh-T
    zones[mask_T & (ppt >= 4000)] = 4  # b-pl-T

    # Premontano (1000-2000m)
    mask_P = (alt >= 1000) & (alt < 2000)
    zones[mask_P & (ppt < 1000)] = 5
    zones[mask_P & (ppt >= 1000) & (ppt < 2000)] = 6
    zones[mask_P & (ppt >= 2000) & (ppt < 4000)] = 7
    zones[mask_P & (ppt >= 4000)] = 8

    # Montano Bajo (2000-3000m)
    mask_MB = (alt >= 2000) & (alt < 3000)
    zones[mask_MB & (ppt < 1000)] = 9
    zones[mask_MB & (ppt >= 1000) & (ppt < 2000)] = 10
    zones[mask_MB & (ppt >= 2000) & (ppt < 4000)] = 11
    zones[mask_MB & (ppt >= 4000)] = 12

    # Montano (>3000m)
    mask_M = alt >= 3000
    zones[mask_M & (ppt < 1000)] = 13
    zones[mask_M & (ppt >= 1000)] = 14
```

```
            lz_arr[valid] = zones

        # 4. Aplicar Máscara de Cuenca (Si existe)
        if mask_geom is not None:
            # Crear archivo en memoria para enmascarar usando rasterio.mask
            from rasterio.io import MemoryFile

            with MemoryFile() as memfile:
                with memfile.open(**profile) as dataset:
                    dataset.write(lz_arr, 1)

                    # Reproyectar geometría si es necesario
                    if hasattr(mask_geom, "crs") and mask_geom.crs != src_dem.crs:
                        mask_geom = mask_geom.to_crs(src_dem.crs)

                    try:
                        out_img, _ = mask(
                            dataset, mask_geom.geometry, crop=True, nodata=0
                        )
                        lz_arr = out_img[0]
                        # Actualizar transform para el recorte
                        # (Simplificado: Devolvemos el array recortado, la visualización se ajustará)
                    except Exception:
                        pass  # Si falla la máscara, devolvemos el original

        return lz_arr, dst_transform, src_dem.crs

    except Exception as e:
        return None, None, str(e)


# --- NUEVA FUNCIÓN: ÍNDICES CLIMÁTICOS ---
def calculate_climatic_indices(series_mensual, alt_m):
    """Calcula Índices de Aridez (Martonne) y Erosividad (Fournier)."""
    if series_mensual.empty:
        return {}

    # Datos base
    P_anual = (
        series_mensual.sum()
    )  # Total anual promedio (si la serie es un año promedio)
    # Si la serie es histórica completa, tomamos el promedio de las sumas anuales
    # Asumimos que 'series_mensual' aquí es el ciclo anual promedio (12 meses) o una serie larga
    # Para robustez, recalculamos P_anual como (promedio_mensual * 12)
    P_total = series_mensual.mean() * 12
```

```
T_media = estimate_temperature(alt_m)

# 1. Índice de Martonne (Aridez)
# I_M = P / (T + 10)
im = P_total / (T_media + 10)

if im < 5:
   im_cat = "Desierto Absoluto"
elif im < 10:
   im_cat = "Árido"
elif im < 20:
   im_cat = "Semiárido"
elif im < 30:
   im_cat = "Mediterráneo / Semihúmedo"
elif im < 60:
   im_cat = "Húmedo"
else:
   im_cat = "Perhúmedo"

# 2. Índice de Fournier Modificado (Erosividad - MFI)
# MFI = Sum(pi^2) / P_total
# pi = precipitación del mes i
# Si la serie es larga, agrupamos por mes primero para tener el ciclo promedio
try:
   # Intentar agrupar si tiene índice fecha, sino usar directo
   if isinstance(series_mensual.index, pd.DatetimeIndex):
      monthly_means = series_mensual.groupby(series_mensual.index.month).mean()
   else:
      monthly_means = series_mensual  # Asumimos que ya son 12 valores

   sum_p2 = (monthly_means**2).sum()
   mfi = sum_p2 / P_total if P_total > 0 else 0

   if mfi < 60:
      mfi_cat = "Muy Baja"
   elif mfi < 90:
      mfi_cat = "Baja"
   elif mfi < 120:
      mfi_cat = "Moderada"
   elif mfi < 160:
      mfi_cat = "Alta"
   else:
      mfi_cat = "Muy Alta"
except:
   mfi = 0
   mfi_cat = "N/A"
```

```
    return {
        "martonne_val": im,
        "martonne_class": im_cat,
        "fournier_val": mfi,
        "fournier_class": mfi_cat,
        "temp_media": T_media,
    }


def calculate_return_periods(df_long, station_name):
    """
    Calcula períodos de retorno usando la distribución de Gumbel sobre máximos anuales.
    """
    # 1. Filtrar datos de la estación
    df_station = df_long[df_long[Config.STATION_NAME_COL] == station_name].copy()

    if df_station.empty:
        return None, None

    # 2. Obtener Máximos Anuales
    # Agrupamos por año y tomamos el valor máximo de precipitación de ese año
    annual_max = (
        df_station.groupby(Config.YEAR_COL)[Config.PRECIPITATION_COL].max().dropna()
    )

    if len(annual_max) < 10:  # Se requieren min 10 años para estadística de extremos
        return None, "Insuficientes datos anuales (<10 años) para ajuste de Gumbel."

    # 3. Ajuste Distribución Gumbel
    # params = (loc, scale)
    params = stats.gumbel_r.fit(annual_max)

    # 4. Calcular Valores para Tr específicos
    tr_list = [2, 5, 10, 25, 50, 100]  # Años de retorno
    probs = [1 - (1 / tr) for tr in tr_list]

    # ppf = Percent point function (inversa de cdf)
    precip_values = stats.gumbel_r.ppf(probs, *params)

    df_results = pd.DataFrame(
        {
            "Período de Retorno (Tr)": tr_list,
            "Probabilidad Excedencia": [f"{1/tr:.1%}" for tr in tr_list],
            "Ppt Máxima Esperada (mm)": precip_values,
        }
```

```
    )

    return df_results, {"params": params, "data": annual_max}


def calculate_percentiles_extremes(df_long, station_name, p_low=10, p_high=90):
    """Identifica eventos por encima/debajo de percentiles."""
    df_station = df_long[df_long[Config.STATION_NAME_COL] == station_name].copy()
    if df_station.empty:
        return None

    val = df_station[Config.PRECIPITATION_COL].dropna()
    if val.empty:
        return None

    # Calcular umbrales
    thresh_low = np.percentile(val, p_low)
    thresh_high = np.percentile(val, p_high)

    # Filtrar eventos
    df_station["Tipo Evento"] = "Normal"
    df_station.loc[
        df_station[Config.PRECIPITATION_COL] <= thresh_low, "Tipo Evento"
    ] = f"Bajo (<P{p_low})"
    df_station.loc[
        df_station[Config.PRECIPITATION_COL] >= thresh_high, "Tipo Evento"
    ] = f"Alto (>P{p_high})"

    return df_station, thresh_low, thresh_high


def calculate_duration_curve(series_mensual, runoff_coeff, area_km2):
    """
    Calcula la Curva de Duración de Caudales (FDC) con ajuste polinómico y R².
    """
    if series_mensual is None or series_mensual.empty:
        return None

    # Q (m3/s) = P(mm/mes) * C * Area(km2) * 1000 / (30.44 * 86400)
    # Factor de conversión de mm/mes a m3/s:
    # (Area * 1000) / (30.44 días/mes * 24h * 3600s)
    # Usamos 30.4375 días como promedio exacto del mes (365.25/12)
    factor = (area_km2 * 1000) / (30.4375 * 86400)
    q_m3s = series_mensual * runoff_coeff * factor

    # Ordenar descendente (Duración)
```

```
sorted_q = q_m3s.sort_values(ascending=False)
n = len(sorted_q)

if n < 5:
    return None

# Probabilidad de Excedencia (Weibull: i / (n+1))
probs = np.arange(1, n + 1) / (n + 1) * 100

try:
    # Ajuste Polinómico Grado 3
    coeffs = np.polyfit(probs, sorted_q.values, 3)
    poly_model = np.poly1d(coeffs)

    # Calcular R² (Coeficiente de Determinación)
    y_pred = poly_model(probs)
    y_true = sorted_q.values
    ss_res = np.sum((y_true - y_pred) ** 2)
    ss_tot = np.sum((y_true - np.mean(y_true)) ** 2)
    r_squared = 1 - (ss_res / ss_tot)

    # Formato DECIMAL LEGIBLE (evitando notación científica)
    # Q = aP³ + bP² + cP + d
    # Usamos 6 decimales para el cúbico porque suele ser pequeño, 2 para la constante
    sign_b = "+" if coeffs[1] >= 0 else "-"
    sign_c = "+" if coeffs[2] >= 0 else "-"
    sign_d = "+" if coeffs[3] >= 0 else "-"

    eq_str = (
        f"Q = {coeffs[0]:.6f}P^3 "
        f"{sign_b} {abs(coeffs[1]):.4f}P^2 "
        f"{sign_c} {abs(coeffs[2]):.2f}P "
        f"{sign_d} {abs(coeffs[3]):.2f}"
    )

    # Generar línea de tendencia para graficar
    trend_x = np.linspace(0, 100, 100)
    trend_y = poly_model(trend_x)
    trend_y = np.maximum(trend_y, 0)  # Caudal no negativo

except:
    eq_str = "N/A"
    r_squared = 0
    trend_x, trend_y = [], []

return {
```

```
        "data": pd.DataFrame(
            {"Probabilidad Excedencia (%)": probs, "Caudal (m³/s)": sorted_q.values}
        ),
        "equation": eq_str,
        "r_squared": r_squared,
        "trend_x": trend_x,
        "trend_y": trend_y,
    }


# --- 4. CORRECCIÓN DE SESGO (BIAS CORRECTION) ---
def calculate_bias_correction_metrics(df_stations, df_satellite):
    """
    Calcula el sesgo (Bias) entre estaciones y satélite.
    Retorna DataFrame con métricas por estación.
    """
    try:
        # Unir por estación
        df_merge = pd.merge(
            df_stations[
                [Config.STATION_NAME_COL, Config.PRECIPITATION_COL, "geometry"]
            ],
            df_satellite[[Config.STATION_NAME_COL, "ppt_sat"]],
            on=Config.STATION_NAME_COL,
        )

        if df_merge.empty:
            return None

        # Calcular métricas
        # Bias = P_estacion / P_satelite (Factor de corrección multiplicativo)
        # Evitar división por cero
        df_merge["bias_factor"] = df_merge[Config.PRECIPITATION_COL] / df_merge[
            "ppt_sat"
        ].replace(0, 0.01)

        # Diff = P_estacion - P_satelite (Sesgo aditivo)
        df_merge["bias_diff"] = df_merge[Config.PRECIPITATION_COL] - df_merge["ppt_sat"]

        # Limpieza de valores extremos (outliers)
        df_merge = df_merge[
            df_merge["bias_factor"].between(0.1, 10)
        ]  # Filtro de seguridad

        return df_merge
    except Exception as e:
```

```
print(f"Error cálculo sesgo: {e}")
return None
```