```python
# ──────────────────────────────────────────────────────────────────────
# hydro_pipeline/__init__.py
# ──────────────────────────────────────────────────────────────────────
"""Package principal du pipeline hydrographique ALOS → GRASS.

Fournit une interface unifiée pour l'analyse hydrographique complète :
1. Téléchargement des MNT ALOS AW3D30
2. Prétraitement des données raster
3. Analyse hydrologique sous GRASS GIS
4. Export des résultats pour SWAT+ et SIG

API publique:
    Config: Schéma de configuration principal
    from_yaml_pair() : Charge les configurations YAML
    merge_config() : Fusionne les configurations
    setup_logging() : Configure le système de logs
    validate_environment() : Vérifie les dépendances
    calculate_bbox_wgs84() : Calcule l'emprise de travail
    pretraiter_mnt() : Prétraitement du MNT
    analyse_hydro_grass() : Workflow hydrologique complet
    export_grass_results() : Export des résultats finaux.

Version : 3.2.0 (Reproductible - Configuration 100% YAML)
"""

from __future__ import annotations

# Metadata
__version__ = "3.2.0"
__author__ = "Zo RASOANAIVO"
__contact__ = "<razorivo85@gmail.com>"
__license__ = "MIT"
__copyright__ = "2025"

# Imports principaux
from .config import Config
from .config_io import (
    from_yaml_pair,
    merge_config,
    save_editable_yaml,
)
from .logging_setup import setup_logging
from .env_utils import validate_environment, safe_subprocess
from .download_dem import calculate_bbox_wgs84, telecharger_mnt
from .preprocess_dem import pretraiter_mnt
from .grass_session import init_grass_modules, initialiser_grass
from .hydro_analysis import analyse_hydro_grass
from .export_results import export_grass_results

# Structure du package
__all__ = [
    # Modules
    "config",
    "config_io",
    "logging_setup",
    "env_utils",
    "download_dem",
    "preprocess_dem",
    "grass_session",
    "hydro_analysis",
    "export_results",
    "gui",

    # Fonctions principales
    "Config",
    "from_yaml_pair",
    "merge_config",
    "save_editable_yaml",
    "setup_logging",
    "validate_environment",
    "safe_subprocess",
    "calculate_bbox_wgs84",
    "telecharger_mnt",
    "pretraiter_mnt",
    "init_grass_modules",
    "initialiser_grass",
    "analyse_hydro_grass",
    "export_grass_results",

    # Metadata
    "__version__"
]

# Documentation supplémentaire
__doc__ += """
Workflow typique:
    1. Charger la configuration (from_yaml_pair)
    2. Valider l'environnement (validate_environment)
    3. Télécharger le MNT (telecharger_mnt)
    4. Prétraiter les données (pretraiter_mnt)
    5. Lancer l'analyse (analyse_hydro_grass)
    6. Exporter les résultats (export_grass_results)
```

```
Référence:
    Rasoanaivo, Z. (2025) - ORCID: 0009-0003-0725-3764
    Pipeline modulaire et reproductible pour l'hydrographie
    avec ALOS AW3D30 et GRASS GIS.
"""

# ============================================================================
# hydro_pipeline/config.py
# ============================================================================
"""Schéma de configuration typé pour le pipeline hydrographique.

Ce module définit la structure de configuration sans valeurs par défaut intégrées,
garantissant que tous les paramètres proviennent des fichiers de configuration YAML.

Attributes:
    Config: Dataclass contenant tous les paramètres configurables du pipeline.
"""

from __future__ import annotations
from dataclasses import dataclass
from typing import Optional


@dataclass
class Config:
    """Conteneur de configuration pour le pipeline hydrographique.

    Tous les champs sont optionnels et doivent être renseignés via les fichiers
    de configuration YAML. Aucune valeur par défaut n'est codée en dur.

    Champs:
        LAT: Latitude du point central (degrés décimaux)
        LON: Longitude du point central (degrés décimaux)
        BBOX_SIZE_KM: Demi-largeur de la zone d'étude (km)
        SITE_NAME: Nom du site d'étude
        EPSG_CIBLE: Code EPSG pour la projection cible
        NODATA_VALUE: Valeur NoData pour les rasters
        STREAM_THRESHOLD_KM2: Seuil d'accumulation pour les cours d'eau (km²)
        GRASS_GISBASE: Chemin d'installation de GRASS GIS
        QGIS_PATH: Chemin d'installation de QGIS (optionnel)
        GDALWARP_CMD: Commande gdalwarp complète
        GDAL_DATA_EXT: Chemin des données GDAL
        PROJ_LIB_EXT: Chemin de la bibliothèque PROJ
        GDAL_BIN_EXT: Chemin des binaires GDAL
        OUTPUT_DIR: Répertoire de sortie
        TEMP_DIR: Répertoire temporaire
        GRASS_DB_DIR: Répertoire de la base GRASS
        OPENTOPOGRAPHY_API_KEY: Clé API OpenTopography
        DEV_MODE: Active le mode développeur (booléen)
    """

    # Géographie et zone d'étude
    LAT: Optional[float] = None
    LON: Optional[float] = None
    BBOX_SIZE_KM: Optional[float] = None
    SITE_NAME: Optional[str] = None

    # Système de référence et données raster
    EPSG_CIBLE: Optional[int] = None
    NODATA_VALUE: Optional[float] = None

    # Paramètres hydrologiques
    STREAM_THRESHOLD_KM2: Optional[float] = None

    # Environnement logiciel
    GRASS_GISBASE: Optional[str] = None
    QGIS_PATH: Optional[str] = None

    # Configuration GDAL/PROJ
    GDALWARP_CMD: Optional[str] = None
    GDAL_DATA_EXT: Optional[str] = None
    PROJ_LIB_EXT: Optional[str] = None
    GDAL_BIN_EXT: Optional[str] = None

    # Gestion des fichiers
    OUTPUT_DIR: Optional[str] = None
    TEMP_DIR: Optional[str] = None
    GRASS_DB_DIR: Optional[str] = None

    # Accès aux données
    OPENTOPOGRAPHY_API_KEY: Optional[str] = None

    # Développement et débogage
    DEV_MODE: Optional[bool] = None

# ============================================================================
# hydro_pipeline/config_io.py
# ============================================================================
"""Gestion robuste des configurations YAML pour le pipeline hydrographique.

Ce module implémente la lecture/écriture sécurisée des fichiers de configuration
```

```
avec résolution des variables d'environnement et fusion intelligente des valeurs.
"""

from __future__ import annotations
from dataclasses import asdict
from typing import Any, Mapping, Dict
import os
import re
import logging
from pathlib import Path

try:
    import yaml
except ImportError as e:
    raise RuntimeError(
        "Le package PyYAML est requis pour ce module. "
        "Installez-le avec: pip install pyyaml"
    ) from e

from .config import Config

# Configuration du logger
logger = logging.getLogger(__name__)

# Pattern pour la détection des variables d'environnement
_ENV_VAR_PATTERN = re.compile(r"^\$\{env:([A-Za-z_][A-Za-z0-9_]*)}$")


class ConfigIOError(RuntimeError):
    """Exception personnalisée pour les erreurs de gestion de configuration."""
    pass


def _resolve_env_vars(value: Any) -> Any:
    """Résout les variables d'environnement dans les valeurs de configuration.

    Args:
        value: Valeur à analyser (peut être de n'importe quel type)

    Returns:
        La valeur originale ou la valeur résolue depuis l'environnement

    Note:
        Seules les chaînes au format exact ${env:VAR} sont résolues.
        Les autres valeurs sont retournées inchangées.
    """
    if isinstance(value, str):
        match = _ENV_VAR_PATTERN.match(value.strip())
        if match:
            env_var = match.group(1)
            resolved_value = os.getenv(env_var, "")
            logger.debug(f"Résolution variable d'environnement: {env_var} -> {resolved_value}")
            return resolved_value
    return value


def _load_single_config(file_path: str) -> Dict[str, Any]:
    """Charge un fichier YAML et résout ses variables d'environnement.

    Args:
        file_path: Chemin vers le fichier YAML

    Returns:
        Dictionnaire contenant la configuration lue

    Raises:
        ConfigIOError: Si le fichier est inaccessible ou invalide
    """
    try:
        path = Path(file_path).absolute()
        logger.info(f"Chargement configuration depuis: {path}")

        with open(path, 'r', encoding='utf-8') as f:
            raw_config = yaml.safe_load(f) or {}

        return {
            key: _resolve_env_vars(value)
            for key, value in raw_config.items()
        }

    except yaml.YAMLError as e:
        error_msg = f"Erreur de syntaxe YAML dans {file_path}: {str(e)}"
        logger.error(error_msg)
        raise ConfigIOError(error_msg) from e
    except OSError as e:
        error_msg = f"Erreur d'accès au fichier {file_path}: {str(e)}"
        logger.error(error_msg)
        raise ConfigIOError(error_msg) from e


def _is_valid_config_value(value: Any) -> bool:
    """Détermine si une valeur de configuration est considérée comme valide.
```

```
    Args:
        value: Valeur à évaluer

    Returns:
        True si la valeur est non-nulle et non-vide, False sinon

    Note:
        Une valeur est considérée valide si :.
        * Elle n'est pas None
        * Pour les strings: non vide après strip()
        * Pour les collections (list, dict, set) : non vides
    """
    if value is None:
        return False
    if isinstance(value, str) and not value.strip():
        return False
    if isinstance(value, (list, dict, set)) and not value:
        return False
    return True


def merge_configs(
        user_config: Mapping[str, Any],
        default_config: Mapping[str, Any]
) -> Dict[str, Any]:
    """Fusionne deux configurations selon la politique 'empty-is-missing'.

    Args:
        user_config: Configuration utilisateur (prioritaire)
        default_config: Configuration par défaut (fallback)

    Returns:
        Dictionnaire contenant la configuration fusionnée

    Raises:
        ConfigIOError: Si la fusion échoue

    Note:
        Pour chaque clé, la valeur d'user_config est utilisée si elle est valide
        (selon _is_valid_config_value), sinon la valeur de default_config est utilisée.
    """
    try:
        merged_config = {}
        all_keys = set(default_config.keys()) | set(user_config.keys())

        for key in all_keys:
            user_value = user_config.get(key)
            merged_config[key] = (
                user_value
                if _is_valid_config_value(user_value)
                else default_config.get(key)
            )

        logger.debug(f"Fusion config terminée. Clés fusionnées: {len(merged_config)}")
        return merged_config

    except Exception as e:
        error_msg = f"Erreur lors de la fusion des configurations: {str(e)}"
        logger.error(error_msg)
        raise ConfigIOError(error_msg) from e


def load_config_pair(
        user_config_path: str,
        default_config_path: str
) -> Config:
    """Charge et fusionne les configurations utilisateur et par défaut.

    Args:
        user_config_path: Chemin vers config.yaml (prioritaire)
        default_config_path: Chemin vers default_config.yaml (fallback)

    Returns:
        Instance de Config initialisée avec les valeurs fusionnées

    Raises:
        ConfigIOError: Si le chargement ou la fusion échoue
    """
    try:
        logger.info(
            f"Chargement paire de configurations:\n"
            f"- User: {user_config_path}\n"
            f"- Default: {default_config_path}"
        )

        default_config = _load_single_config(default_config_path)
        user_config = _load_single_config(user_config_path)

        merged = merge_configs(user_config, default_config)
        return Config(**merged)
```

```
    except Exception as e:
        error_msg = (
            f"Impossible de charger la configuration:\n"
            f"User: {user_config_path}\n"
            f"Default: {default_config_path}\n"
            f"Erreur: {str(e)}"
        )
        logger.error(error_msg)
        raise ConfigIOError(error_msg) from e


def save_config(
        config: Config,
        output_path: str,
        *,
        minimal_output: bool = True
) -> None:
    """Sauvegarde une configuration dans un fichier YAML.

    Args:
        config: Configuration à sauvegarder
        output_path: Chemin de destination
        minimal_output: Si True, n'inclut que les valeurs non-nulles

    Raises:
        ConfigIOError: Si l'écriture échoue
    """
    try:
        path = Path(output_path).absolute()
        logger.info(f"Sauvegarde configuration vers: {path}")

        config_dict = asdict(config)

        if minimal_output:
            config_dict = {
                k: v for k, v in config_dict.items()
                if _is_valid_config_value(v)
            }

        with open(path, 'w', encoding='utf-8') as f:
            yaml.safe_dump(
                config_dict,
                f,
                sort_keys=False,
                allow_unicode=True,
                default_flow_style=False
            )

        logger.debug(f"Configuration sauvegardée avec succès. Taille: {path.stat().st_size} octets")

    except Exception as e:
        error_msg = f"Erreur lors de la sauvegarde vers {output_path}: {str(e)}"
        logger.error(error_msg)
        raise ConfigIOError(error_msg) from e


# Alias pour compatibilité ascendante
from_yaml_pair = load_config_pair
merge_config = merge_configs
save_editable_yaml = save_config

# ---------------------------------------------------------------------------
# hydro_pipeline/logging_setup.py
# ---------------------------------------------------------------------------
"""Politique générale de journalisation.

Ce module centralise la configuration des logs pour l'ensemble du pipeline.
Il garantit une journalisation structurée, horodatée et adaptée aux besoins
scientifiques (traçabilité, reproductibilité, intégration dans les rapports).

Fonction publique
-----------------
setup_logging(level=logging.INFO)
    Configure et retourne le logger principal du projet.
"""

from __future__ import annotations
import logging
import sys


def setup_logging(level: int = logging.INFO) -> logging.Logger:
    """Initialise et retourne le logger principal du pipeline hydro.

    La configuration applique :
    * Un format standard avec horodatage et niveau de log
    * Une sortie vers stdout (compatible avec les environnements conteneurisés)
    * Un niveau de log paramétrable.

    Notes
    -----
    L'appel à basicConfig est idempotent : une seconde invocation ne modifie pas
```

```
    la configuration existante sauf si force=True est spécifié.

    Paramètres
    ----------
    level : int, optionnel
        Niveau minimal de journalisation (par défaut : logging.INFO).

    Retour
    ------
    logging.Logger
        Instance configurée du logger nommé 'hydro_pipeline'.
    """
    logging.basicConfig(
        level=level,
        format="%(asctime)s [%(levelname)s] %(message)s",
        handlers=[logging.StreamHandler(sys.stdout)],
        force=True  # S'assure de surcharger toute config existante
    )

    logger = logging.getLogger("hydro_pipeline")
    logger.debug("Logger hydro_pipeline configuré avec niveau %s", logging.getLevelName(level))

    return logger

# ============================================================================
# hydro_pipeline/env_utils.py
# ============================================================================
"""Gestion robuste de l'environnement d'exécution et des sous-processus.

Ce module assure :
* La validation des dépendances externes (GRASS, GDAL, PROJ)
* La configuration sécurisée des environnements d'exécution
* L'exécution tracée de commandes système avec gestion d'erreurs complète.

Éléments publics
----------------
EnvError
    Exception signalant une anomalie de configuration environnementale.
safe_subprocess(cmd_args, env=None, timeout=600, logger=None)
    Lanceur sécurisé de processus externes avec journalisation détaillée.
validate_environment(cfg, logger) → dict
    Vérifie l'intégrité de l'environnement et construit les contextes d'exécution.
"""

from __future__ import annotations
import os
import subprocess
from typing import Dict, Optional

from .config import Config


class EnvError(RuntimeError):
    """Erreur critique liée à la configuration de l'environnement d'exécution."""
    pass


def safe_subprocess(
        cmd_args: list[str],
        env: Optional[Dict[str, str]] = None,
        timeout: int = 600,
        logger=None,
) -> None:
    """Exécute une commande système avec gestion complète des erreurs."""
    cmd_display = " ".join(cmd_args)[:200] + ("…" if len(" ".join(cmd_args)) > 200 else "")
    if logger:
        logger.info("Lancement sous-processus: %s", cmd_display)

    try:
        result = subprocess.run(
            cmd_args,
            check=True,
            capture_output=True,
            text=True,
            env=env,
            timeout=timeout,
        )

        if logger:
            if result.stdout:
                logger.debug("Sortie standard:\n%s", result.stdout.strip())
            if result.stderr:
                logger.debug("Sortie erreur:\n%s", result.stderr.strip())

    except subprocess.CalledProcessError as e:
        if logger:
            logger.error(
                "Échec commande (code %d): %s\nSortie erreur:\n%s",
                e.returncode,
                cmd_display,
                (e.stderr or "").strip(),
            )
```

```
        raise
    except subprocess.TimeoutExpired:
        if logger:
            logger.error("Timeout dépassé (%ds) sur commande: %s", timeout, cmd_display)
        raise
    except Exception as e:
        if logger:
            logger.error("Erreur système lors de l'exécution: %s", str(e))
        raise


def validate_environment(cfg: Config, logger) -> dict:
    """Vérifie l'intégrité de l'environnement et prépare les contextes d'exécution."""
    # 1) Clé API
    if not cfg.OPENTOPOGRAPHY_API_KEY:
        raise EnvError("Configuration manquante: OPENTOPOGRAPHY_API_KEY requis")

    # 2) GRASS
    if not cfg.GRASS_GISBASE or not os.path.isdir(cfg.GRASS_GISBASE):
        raise EnvError(f"Chemin GRASS invalide: {cfg.GRASS_GISBASE}")

    # Racine OSGeo4W (heuristique Windows)
    osgeo4w_root = os.path.dirname(os.path.dirname(os.path.dirname(cfg.GRASS_GISBASE)))
    grass_candidates = [
        os.path.join(osgeo4w_root, "bin", "grass84.bat"),
        os.path.join(cfg.GRASS_GISBASE, "bin", "grass84.bat"),
    ]
    grass_cmd = next((c for c in grass_candidates if os.path.isfile(c)), None)
    if not grass_cmd:
        raise EnvError(f"Aucun binaire GRASS trouvé parmi: {grass_candidates}")

    # Chemin Python de GRASS
    grass_python_path = os.path.join(cfg.GRASS_GISBASE, "etc", "python")
    if not os.path.isdir(grass_python_path):
        raise EnvError(f"Librairies Python GRASS introuvables: {grass_python_path}")

    # 3) GDAL/PROJ
    required_paths = {
        "GDALWARP_CMD": ("gdalwarp", os.path.isfile),
        "GDAL_DATA_EXT": ("GDAL_DATA", os.path.isdir),
        "PROJ_LIB_EXT": ("PROJ_LIB", os.path.isdir),
        "GDAL_BIN_EXT": ("GDAL BIN", os.path.isdir),
    }
    for attr, (label, check) in required_paths.items():
        path = getattr(cfg, attr)
        if not path or not check(path):
            raise EnvError(f"{label} invalide: {path}")

    logger.info("Environnement validé avec succès (GRASS/GDAL/PROJ)")

    # 4) Construction env GDAL + Python OSGeo4W
    gdal_env = os.environ.copy()
    gdal_env.update({
        "GDAL_DATA": cfg.GDAL_DATA_EXT,
        "PROJ_LIB": cfg.PROJ_LIB_EXT,
        "PATH": f"{cfg.GDAL_BIN_EXT}{os.pathsep}{gdal_env.get('PATH', '')}",
    })

    # Définir un PYTHONHOME cohérent pour calmer "<prefix>"
    python_home_candidates = [
        os.path.join(osgeo4w_root, "apps", "Python312"),
        os.path.join(osgeo4w_root, "apps", "Python311"),
        os.path.join(osgeo4w_root, "apps", "Python310"),
    ]
    python_home = next((p for p in python_home_candidates if os.path.isdir(p)), None)
    if python_home:
        gdal_env["PYTHONHOME"] = python_home

    # Assurer PYTHONPATH incluant les libs GRASS
    grass_py = grass_python_path
    if gdal_env.get("PYTHONPATH"):
        gdal_env["PYTHONPATH"] = os.pathsep.join([grass_py, gdal_env["PYTHONPATH"]])
    else:
        gdal_env["PYTHONPATH"] = grass_py

    # Propager aussi à l'environnement du processus courant (pour GRASS Python)
    os.environ.update(gdal_env)

    return {
        "GRASS_CMD": grass_cmd,
        "GRASS_PYTHON_PATH": grass_python_path,
        "GDAL_ENV": gdal_env,
    }


# ──────────────────────────────────────────────────────────────────────
# hydro_pipeline/download_dem.py
# ──────────────────────────────────────────────────────────────────────
"""Gestion du téléchargement de MNT ALOS AW3D30 depuis OpenTopography.

Ce module fournit :
- Un calcul précis de bounding box WGS84 géodésiquement correct
```

```
- Un téléchargement robuste de données raster avec :
  * Gestion des gros fichiers par flux
  * Journalisation détaillée
  * Gestion complète des erreurs réseau/IO

Fonctions publiques
-------------------
calculate_bbox_wgs84(lat, lon, size_km) -> (west, south, east, north)
    Calcule une emprise géographique centrée sur un point.
telecharger_mnt(bbox, temp_dir, api_key, logger) -> str
    Télécharge un MNT AW3D30 et retourne le chemin du fichier local.
"""

from __future__ import annotations
import os
from typing import Tuple

import numpy as np
import requests


def calculate_bbox_wgs84(lat: float, lon: float, size_km: float) -> Tuple[float, float, float, float]:
    """Calcule une emprise géographique centrée sur (lat, lon).

    La bounding box est calculée en tenant compte de :
    * La courbure terrestre (approximation sphérique)
    * La réduction des distances longitudinales aux hautes latitudes.

    Paramètres
    ----------
    lat : float
        Latitude du centre en degrés décimaux (WGS84)
    lon : float
        Longitude du centre en degrés décimaux (WGS84)
    size_km : float
        Demi-côté de la zone carrée en kilomètres

    Retour
    ------
    Tuple[float, float, float, float]
        Coordonnées (ouest, sud, est, nord) en degrés décimaux

    Notes
    -----
    Utilise une approximation sphérique de la Terre (rayon = 6371 km)
    avec 1 degré ≈ 111 km. La précision est suffisante pour des zones
    de quelques centaines de kilomètres.
    """
    EARTH_DEGREE_KM = 111.0  # Approximation km par degré

    # Conversion latitude
    dlat = size_km / EARTH_DEGREE_KM

    # Conversion longitude avec correction cos(latitude)
    coslat = np.cos(np.radians(lat))
    dlon = size_km / (EARTH_DEGREE_KM * max(coslat, 0.01))  # Seuil à 0.01 pour éviter les valeurs extrêmes

    return (
        lon - dlon,  # ouest
        lat - dlat,  # sud
        lon + dlon,  # est
        lat + dlat  # nord
    )


def telecharger_mnt(bbox: Tuple[float, float, float, float], temp_dir: str, api_key: str, logger) -> str:
    """Télécharge un MNT ALOS AW3D30 depuis l'API OpenTopography.

    Features clés :
    * Téléchargement par blocs pour économiser la mémoire
    * Gestion des timeouts et erreurs réseau
    * Vérification du dossier de destination
    * Journalisation complète du processus.

    Paramètres
    ----------
    bbox : Tuple[float, float, float, float]
        Emprise géographique (ouest, sud, est, nord) en degrés WGS84
    temp_dir : str
        Chemin absolu du dossier de destination
    api_key : str
        Clé d'API OpenTopography valide
    logger : logging.Logger
        Logger pour le suivi d'exécution

    Retour
    ------
    str
        Chemin absolu du fichier GeoTIFF téléchargé

    Exceptions
    ----------
```

```
    requests.RequestException
        Pour les erreurs HTTP/timeout
    OSError
        Pour les problèmes d'accès au filesystem
    RuntimeError
        Si le dossier de destination ne peut être créé
    """
    west, south, east, north = bbox
    OPENTOPO_URL = "https://portal.opentopography.org/API/globaldem"
    OUTPUT_FILENAME = "alos_raw.tif"

    # Préparation du dossier de destination
    try:
        os.makedirs(temp_dir, exist_ok=True)
    except OSError as e:
        logger.error("Erreur création dossier %s: %s", temp_dir, str(e))
        raise RuntimeError(f"Impossible de créer le dossier {temp_dir}") from e

    out_path = os.path.join(temp_dir, OUTPUT_FILENAME)
    logger.info(
        "Début téléchargement AW3D30 - Emprise: W=%.5f S=%.5f E=%.5f N=%.5f",
        west, south, east, north
    )

    # Configuration requête
    params = {
        "demtype": "AW3D30",
        "south": south,
        "north": north,
        "west": west,
        "east": east,
        "outputFormat": "GTiff",
        "API_Key": api_key,
    }

    try:
        with requests.get(
                OPENTOPO_URL,
                params=params,
                stream=True,
                timeout=300  # 5 minutes timeout
        ) as response:
            response.raise_for_status()

            # Écriture progressive par blocs de 1MB
            with open(out_path, "wb") as f:
                for chunk in response.iter_content(chunk_size=1 << 20):  # 1MB
                    if chunk:  # Filtrer les keep-alive chunks vides
                        f.write(chunk)

        logger.info("Téléchargement terminé: %s (%.1f MB)",
                    out_path,
                    os.path.getsize(out_path) / (1024 * 1024))
        return out_path

    except requests.RequestException as e:
        logger.error("Erreur API OpenTopography: %s", str(e))
        raise
    except OSError as e:
        logger.error("Erreur écriture fichier %s: %s", out_path, str(e))
        raise

# ------------------------------------------------------------------------------
# hydro_pipeline/preprocess_dem.py
# ------------------------------------------------------------------------------
"""Prétraitement du MNT : reprojection et assainissement des valeurs.

Ce module délègue la reprojection à "gdalwarp" (GDAL) et effectue un
nettoyage simple des valeurs aberrantes avec Rasterio :
* reprojection vers l'EPSG cible (interpolation bilinéaire) ;
* remplacement de valeurs évidemment invalides par une valeur NoData cohérente.
"""

from __future__ import annotations
import os
from typing import Any

import numpy as np
import rasterio
from rasterio.io import DatasetWriter
from logging import Logger

from .env_utils import safe_subprocess


def _reprojection_gdalwarp(
        raw_dem_path: str,
        out_path: str,
        epsg_cible: int,
        gdal_env: dict[str, Any],
        gdalwarp_cmd: str,
        logger: Logger
```

```
) -> None:
    """Exécute la reprojection du MNT avec gdalwarp.

    Args:
        raw_dem_path: Chemin vers le MNT source
        out_path: Chemin de sortie pour le MNT reprojeté
        epsg_cible: Code EPSG de destination
        gdal_env: Environnement d'exécution GDAL
        gdalwarp_cmd: Chemin vers l'exécutable gdalwarp
        logger: Logger pour le suivi

    Raises:
        RuntimeError: Si l'exécution de gdalwarp échoue
    """
    cmd = [
        gdalwarp_cmd,
        "-overwrite",
        "-t_srs", f"EPSG:{epsg_cible}",
        "-r", "bilinear",
        raw_dem_path,
        out_path,
    ]
    safe_subprocess(cmd, env=gdal_env, timeout=900, logger=logger)


def _nettoyage_valeurs_rasterio(
        out_path: str,
        nodata_value: float,
        logger: Logger
) -> None:
    """Nettoie les valeurs aberrantes du raster et met à jour les métadonnées.

    Args:
        out_path: Chemin vers le fichier raster à nettoyer
        nodata_value: Valeur NoData à utiliser
        logger: Logger pour le suivi

    Raises:
        RuntimeError: En cas d'erreur de traitement du raster
    """
    with rasterio.open(out_path, "r+") as ds:
        arr = ds.read(1, masked=True)
        mask = (arr < -100) | (arr > 10000)
        arr = np.ma.masked_where(mask, arr)

        filled = arr.filled(nodata_value)
        ds.write(filled, 1)

        _mettre_a_jour_metadonnees_nodata(ds, nodata_value)

    logger.info("MNT reprojeté et nettoyé : %s", out_path)


def _mettre_a_jour_metadonnees_nodata(
        ds: DatasetWriter,
        nodata_value: float
) -> None:
    """Met à jour les métadonnées NoData du raster.

    Args:
        ds: Dataset Rasterio en mode écriture
        nodata_value: Valeur NoData à enregistrer
    """
    ds.update_tags(nodata=nodata_value)
    try:
        ds.nodata = nodata_value
    except Exception:
        pass


def pretraiter_mnt(
        raw_dem_path: str,
        temp_dir: str,
        epsg_cible: int,
        nodata_value: float,
        gdal_env: dict[str, Any],
        gdalwarp_cmd: str,
        logger: Logger,
) -> str:
    """Reprojeter et assainir le MNT.

    Étapes:
        1) Reprojection vers l'EPSG cible via gdalwarp (bilinéaire)
        2) Assainissement : détection de valeurs aberrantes et remplacement par NoData

    Args:
        raw_dem_path: Chemin du GeoTIFF brut
        temp_dir: Dossier de travail temporaire
        epsg_cible: Code EPSG de destination
        nodata_value: Valeur NoData cohérente
        gdal_env: Environnement d'exécution GDAL
        gdalwarp_cmd: Chemin vers l'exécutable gdalwarp
```

```
        logger: Logger pour le suivi

    Returns:
        Chemin du GeoTIFF reprojeté et nettoyé

    Raises:
        RuntimeError: En cas d'échec de reprojection ou de traitement raster
    """
    out_path = os.path.join(temp_dir, "cleaned_dem_temp.tif")

    try:
        os.makedirs(os.path.dirname(out_path), exist_ok=True)
        _reprojection_gdalwarp(raw_dem_path, out_path, epsg_cible, gdal_env, gdalwarp_cmd, logger)
        _nettoyage_valeurs_rasterio(out_path, nodata_value, logger)
        return out_path

    except Exception as e:
        raise RuntimeError(f"Échec du prétraitement du MNT : {e}") from e

# ─────────────────────────────────────────────────────────────────────
# hydro_pipeline/grass_session.py
# ─────────────────────────────────────────────────────────────────────
"""Import dynamique des modules GRASS et initialisation d'une session.

Ce module fournit :
* un import sécurisé des modules Python de GRASS depuis un chemin spécifique
* une fonction d'initialisation de LOCATION/MAPSET avec création si nécessaire
* une configuration correcte des variables d'environnement GRASS.

Fonctions publiques
-------------------
init_grass_modules(grass_python_path, logger) → GrassImports
    Charge les modules GRASS depuis le chemin spécifié.
initialiser_grass(gi, grass_cmd, gisdb_path, location_name, mapset_name, epsg_code, logger)
    Initialise une session GRASS avec les paramètres donnés.
"""

from __future__ import annotations
import os
import sys
import subprocess
import importlib
from typing import Optional, Any


class GrassImports:
    """Conteneur pour les imports dynamiques des modules GRASS."""

    def __init__(self):
        self.grass: Any = None   # Module grass.script
        self.gsetup: Any = None  # Module grass.script.setup
        self.GrassError: Optional[type] = None  # Classe d'exception GRASS


def init_grass_modules(grass_python_path: str, logger) -> GrassImports:
    """Importe les modules Python de GRASS depuis le chemin spécifié."""
    normalized_path = os.path.normpath(grass_python_path)
    sys.path = [p for p in sys.path if os.path.normpath(p) != normalized_path]

    if normalized_path not in [os.path.normpath(p) for p in sys.path]:
        sys.path.insert(0, normalized_path)
        logger.info("Ajout à sys.path : %s", normalized_path)

    try:
        grass = importlib.import_module("grass.script")
        gsetup = importlib.import_module("grass.script.setup")
        try:
            GrassError = importlib.import_module("grass.exceptions").GrassError  # type: ignore[attr-defined]
        except Exception:
            GrassError = Exception
            logger.warning("Fallback Exception pour GrassError (grass.exceptions indisponible).")

        gi = GrassImports()
        gi.grass = grass
        gi.gsetup = gsetup
        gi.GrassError = GrassError
        logger.info("Modules GRASS importés avec succès.")
        return gi

    except Exception as e:
        logger.critical("Échec de l'import des modules GRASS : %s", e)
        raise


def _deduire_gisbase_depuis_module_grass(gi: GrassImports) -> str:
    """Déduit le chemin GISBASE à partir du module grass.script."""
    script_file = getattr(gi.grass, "__file__", None)
    if not script_file:
        raise RuntimeError("Impossible de localiser le module 'grass.script'.")

    path = os.path.dirname(script_file)
    for _ in range(15):
```

```
        head, tail = os.path.split(path)
        if tail.lower() == "etc":
            gisbase = head
            if os.path.isdir(gisbase):
                return gisbase
            break
        if not head or head == path:
            break
        path = head

    raise RuntimeError("Structure des modules GRASS inattendue - impossible de déduire GISBASE.")


def initialiser_grass(
    gi: GrassImports,
    grass_cmd: str,
    gisdb_path: str,
    location_name: str,
    mapset_name: str,
    epsg_code: int,
    logger,
) -> None:
    """Initialise une session GRASS avec création si nécessaire."""
    os.makedirs(gisdb_path, exist_ok=True)
    location_path = os.path.join(gisdb_path, location_name)

    if not os.path.isdir(location_path):
        logger.info("Création de la LOCATION GRASS : %s", location_path)
        cmd = [grass_cmd, "--text", "-c", f"EPSG:{epsg_code}", location_path]
        subprocess.run(
            cmd, input="exit\n", check=True, capture_output=True, text=True, timeout=300
        )
    else:
        logger.info("LOCATION existante détectée : %s", location_name)

    gisbase = _deduire_gisbase_depuis_module_grass(gi)
    os.environ["GISBASE"] = gisbase
    grass_bin = os.path.join(gisbase, "bin")
    os.environ["PATH"] = f"{grass_bin}{os.pathsep}{os.environ.get('PATH', '')}"

    # Réduction maximale de la verbosité (supprime les 'ATTENTION')
    os.environ["GRASS_VERBOSE"] = "0"  # 0 = erreurs seules

    gi.gsetup.init(gisdb_path, location_name, mapset_name)
    try:
        gi.grass.run_command("g.gisenv", set="VERBOSE=0", quiet=True)
    except Exception:
        pass

    logger.info("Session GRASS initialisée : %s/%s", location_name, mapset_name)


# ============================================================================
# hydro_pipeline/hydro_analysis.py
# ============================================================================
"""Chaîne d'analyse hydrologique sous GRASS GIS.

Ce module exécute un workflow complet d'analyse hydrologique incluant :
* Import et traitement du MNT
* Extraction du réseau hydrographique
* Délimitation de bassin versant
* Préparation des données pour SWAT+
"""

from __future__ import annotations
import os
from typing import Tuple
from pyproj import Transformer


def analyse_hydro_grass(
        gi,
        cleaned_dem_path: str,
        lat: float,
        lon: float,
        threshold_km2: float,
        epsg_wgs84: int,
        epsg_target: int,
        logger,
) -> Tuple[float, float]:
    g = gi.grass

    try:
        _import_and_setup_dem(g, cleaned_dem_path)
        _run_hydrological_analysis(g, logger)
        seuil = _calculate_threshold(g, threshold_km2, logger)
        _extract_stream_network(g, seuil)

        out_x, out_y = _process_input_point(
            g, gi, lat, lon, epsg_wgs84, epsg_target, cleaned_dem_path, logger
        )
```

```
        _delineate_watershed(g, out_x, out_y)
        _prepare_swat_outlet(g, out_x, out_y, cleaned_dem_path)
        _post_processing(g, logger)

        return out_x, out_y

    except gi.GrassError as e:
        logger.error("Erreur GRASS dans l'analyse hydrologique: %s", str(e))
        raise
    except Exception as e:
        logger.critical("Erreur inattendue dans l'analyse hydrologique: %s", str(e))
        raise RuntimeError(f"Échec de l'analyse hydrologique: {e}") from e


def _import_and_setup_dem(g, dem_path: str) -> None:
    g.run_command("r.in.gdal", input=dem_path, output="dem", flags="o", overwrite=True, quiet=True)
    g.run_command("g.region", raster="dem", flags="p", quiet=True)


def _run_hydrological_analysis(g, logger) -> None:
    g.run_command(
        "r.fill.dir",
        input="dem",
        output="dem_filled",
        direction="drain_map_for_outlet",
        overwrite=True,
        quiet=True,
    )
    g.run_command(
        "r.watershed",
        elevation="dem_filled",
        accumulation="flow_acc",
        drainage="drain_map_for_outlet",
        threshold=1,
        overwrite=True,
        quiet=True,
    )
    logger.debug("Traitements hydrologiques de base terminés")


def _calculate_threshold(g, threshold_km2: float, logger) -> int:
    region = g.parse_command("g.region", flags="g", quiet=True)
    cell_area = float(region["ewres"]) * float(region["nsres"])
    seuil = max(1, int(threshold_km2 * 1_000_000.0 / cell_area))
    logger.info("Seuil d'accumulation: %d cellules (%.2f km²)", seuil, threshold_km2)
    return seuil


def _extract_stream_network(g, threshold: int) -> None:
    g.run_command(
        "r.stream.extract",
        elevation="dem_filled",
        accumulation="flow_acc",
        threshold=threshold,
        stream_rast="streams",
        stream_vect="rivieres",
        overwrite=True,
        quiet=True,
    )


def _process_input_point(
        g, gi, lat: float, lon: float, epsg_wgs84: int, epsg_target: int, base_path: str, logger
) -> Tuple[float, float]:
    transformer = Transformer.from_crs(
        f"EPSG:{epsg_wgs84}", f"EPSG:{epsg_target}", always_xy=True
    )
    proj_x, proj_y = transformer.transform(lon, lat)

    csv_path = os.path.join(os.path.dirname(base_path), "input_point_proj.csv")
    with open(csv_path, "w", encoding="utf-8") as f:
        f.write("x,y,name\n")
        f.write(f"{proj_x},{proj_y},input_location\n")

    g.run_command(
        "v.in.ascii",
        input=csv_path,
        output="input_point_proj",
        x=1,
        y=2,
        separator="comma",
        skip=1,
        overwrite=True,
        quiet=True,
    )

    out = g.read_command(
        "v.distance",
        from_="input_point_proj",
        to="rivieres",
        upload="dist,to_x,to_y",
        to_type="line",
```

```
        flags="p",
        quiet=True,
    )
    lines = (out or "").strip().splitlines()
    if len(lines) < 2:
        raise gi.GrassError("Aucun cours d'eau trouvé à proximité du point d'entrée")

    parts = lines[1].split("|")
    out_x, out_y = float(parts[2]), float(parts[3])
    logger.info("Exutoire ajusté: (%.2f, %.2f)", out_x, out_y)
    return out_x, out_y


def _delineate_watershed(g, x: float, y: float) -> None:
    g.run_command(
        "r.water.outlet",
        input="drain_map_for_outlet",
        output="main_basin",
        coordinates=[x, y],
        overwrite=True,
        quiet=True,
    )
    g.run_command(
        "r.to.vect",
        input="main_basin",
        output="main_basin_vect",
        type="area",
        flags="s",
        overwrite=True,
        quiet=True,
    )


def _prepare_swat_outlet(g, x: float, y: float, base_path: str) -> None:
    csv_path = os.path.join(os.path.dirname(base_path), "outlet_point_swat.csv")
    with open(csv_path, "w", encoding="utf-8") as f:
        f.write("x,y,PointId,RES,INLET,ID,PTSOURCE\n")
        f.write(f"{x},{y},1,0,0,1,0\n")

    g.run_command(
        "v.in.ascii",
        input=csv_path,
        output="outlet_point",
        x=1,
        y=2,
        separator="comma",
        skip=1,
        overwrite=True,
        quiet=True,
    )

    columns = [
        ("PointId", "1"), ("RES", "0"),
        ("INLET", "0"), ("ID", "1"), ("PTSOURCE", "0")
    ]
    g.run_command(
        "v.db.addcolumn",
        map="outlet_point",
        columns=",".join(f"{col} INTEGER" for col, _ in columns),
        quiet=True,
    )
    for col, val in columns:
        g.run_command("v.db.update", map="outlet_point", column=col, value=val, quiet=True)


def _post_processing(g, logger) -> None:
    """Post-traitement silencieux (sans avertissements)."""
    # 1) Réseau lignes uniquement
    g.run_command(
        "v.extract",
        input="rivieres",
        type="line",
        output="rivieres_only_lines",
        overwrite=True,
        quiet=True,
    )
    g.run_command("v.build", map="rivieres_only_lines", quiet=True)

    # 2) Dissoudre le bassin par catégorie pour éviter l'avertissement
    g.run_command(
        "v.dissolve",
        input="main_basin_vect",
        output="main_basin_diss",
        column="cat",          # évite "No 'column' option specified..."
        overwrite=True,
        quiet=True,
    )

    # 3) Clip du réseau par le bassin dissous
    g.run_command(
        "v.clip",
        input="rivieres_only_lines",
```

```
        clip="main_basin_diss",
        output="rivieres_clipped",
        overwrite=True,
        quiet=True,
    )

    # 4) Clip du point d'exutoire
    g.run_command(
        "v.select",
        ainput="outlet_point",
        binput="main_basin_diss",
        output="outlet_point_clipped",
        operator="within",
        overwrite=True,
        quiet=True,
    )

    # 5) Masque raster final
    g.run_command("r.mask", vector="main_basin_diss", overwrite=True, quiet=True)
    g.run_command("r.mapcalc", expression="dem_filled_masked=dem_filled", overwrite=True, quiet=True)
    g.run_command("r.mask", flags="r", overwrite=True, quiet=True)

    logger.debug("Post-traitement terminé (silencieux).")


# ─────────────────────────────────────────────────────────────────────
# hydro_pipeline/export_results.py
# ─────────────────────────────────────────────────────────────────────
"""Export des résultats GRASS : couches vecteur et MNT masqué.

Ce module gère l'export des produits finaux de l'analyse hydrologique vers :
* Un GeoPackage contenant le réseau hydrographique, l'exutoire et le bassin versant
* Un GeoTIFF du MNT masqué par le bassin versant.
"""

from __future__ import annotations
import os
from pathlib import Path


def export_grass_results(gi, output_dir: str, logger) -> None:
    """Exporte les produits finaux (GPKG + GeoTIFF) depuis GRASS."""
    g = gi.grass

    try:
        gpkg_path, dem_path = _prepare_output_paths(output_dir, logger)

        # Export vecteur — réduire la verbosité (quiet=True)
        _export_vector_layers(g, gpkg_path, logger)

        # Export raster — idem
        _export_dem_raster(g, dem_path, logger)

        logger.info("Exports finalisés avec succès : %s | %s", gpkg_path, dem_path)

    except gi.GrassError as e:
        logger.error("Erreur GRASS lors de l'export : %s", str(e))
        raise
    except Exception as e:
        logger.critical("Erreur lors de l'export des résultats : %s", str(e))
        raise RuntimeError(f"Échec de l'export des résultats : {e}") from e


def _prepare_output_paths(output_dir: str, logger) -> tuple[str, str]:
    """Prépare les chemins de sortie et nettoie les fichiers existants."""
    try:
        Path(output_dir).mkdir(parents=True, exist_ok=True)
    except Exception as e:
        raise RuntimeError(f"Impossible de créer le dossier de sortie {output_dir} : {e}") from e

    gpkg_path = os.path.join(output_dir, "hydro_results.gpkg")
    dem_path = os.path.join(output_dir, "MNT_decoupe_bassin_versant.tif")

    # Nettoyage des exports précédents
    for file_path in (gpkg_path, dem_path):
        try:
            if os.path.exists(file_path):
                os.remove(file_path)
                logger.debug("Fichier existant supprimé : %s", file_path)
        except OSError as e:
            logger.warning("Impossible de supprimer le fichier %s : %s", file_path, str(e))

    return gpkg_path, dem_path


def _export_vector_layers(g, gpkg_path: str, logger) -> None:
    """Exporte les couches vectorielles vers un GeoPackage (silencieux)."""
    layers = [
            ("rivieres_clipped", "rivieres"),
            ("outlet_point_clipped", "exutoire"),
            ("main_basin_diss", "main_basin"),
    ]
```

```
    for i, (input_layer, output_layer) in enumerate(layers):
        try:
            kwargs = dict(
                input=input_layer,
                output=gpkg_path,
                format="GPKG",
                output_layer=output_layer,
                overwrite=True,
                quiet=True,
            )
            if i > 0:
                kwargs["flags"] = "a"  # append
            g.run_command("v.out.ogr", **kwargs)

            logger.debug("Couche exportée : %s → %s", input_layer, output_layer)
        except Exception as e:
            logger.error("Échec export couche %s : %s", input_layer, str(e))
            raise


def _export_dem_raster(g, dem_path: str, logger) -> None:
    """Exporte le raster MNT vers un GeoTIFF compressé (silencieux)."""
    try:
        g.run_command(
            "r.out.gdal",
            input="dem_filled_masked",
            output=dem_path,
            format="GTiff",
            createopt="COMPRESS=DEFLATE,PREDICTOR=2",
            flags="c",
            overwrite=True,
            quiet=True,  # supprime les messages verbeux de GDAL
        )
        logger.debug("MNT exporté : %s", dem_path)
    except Exception as e:
        logger.error("Échec export MNT : %s", str(e))
        raise


# ─────────────────────────────────────────────────────────────────────────────
# hydro_pipeline/gui.py
# ─────────────────────────────────────────────────────────────────────────────
"""Interface Tkinter pour configurer et exécuter le pipeline hydrologique.

Fonctions principales :
* Chargement des fichiers de configuration YAML
* Édition des paramètres
* Sauvegarde de la configuration
* Exécution du pipeline avec affichage des logs
* Gestion du mode développeur
"""

from __future__ import annotations
import os
import shutil
import threading
import tkinter as tk
from tkinter import ttk, filedialog, messagebox
from datetime import datetime

from .config import Config
from .config_io import from_yaml_pair, save_editable_yaml
from .logging_setup import setup_logging
from .env_utils import validate_environment
from .download_dem import calculate_bbox_wgs84, telecharger_mnt
from .preprocess_dem import pretraiter_mnt
from .grass_session import init_grass_modules, initialiser_grass
from .hydro_analysis import analyse_hydro_grass
from .export_results import export_grass_results


class PipelineConfigGUI(tk.Tk):
    """Interface graphique pour le pipeline hydrologique."""

    def __init__(self, cfg: Config | None = None):
        """Initialise l'interface avec une configuration optionnelle."""
        super().__init__()
        self.title("Configuration du Pipeline Hydrologique")
        self.geometry("900x750")
        self.logger = setup_logging()
        self.cfg = cfg
        self._initialiser_interface()

    # ─────────────────────────────────────────────────────────────────────
    # Méthodes internes pour la construction de l'interface
    # ─────────────────────────────────────────────────────────────────────

    def _initialiser_interface(self):
        """Configure les widgets principaux de l'interface."""
        self._creer_widgets_principaux()
        self._creer_variables_tk()
```

```python
        self._organiser_widgets()

    def _creer_widgets_principaux(self):
        """Crée les conteneurs principaux avec scrollbar."""
        self.main_frame = ttk.Frame(self)
        self.main_frame.pack(fill="both", expand=True, padx=10, pady=10)

        self.canvas = tk.Canvas(self.main_frame)
        self.scrollbar = ttk.Scrollbar(
            self.main_frame,
            orient="vertical",
            command=self.canvas.yview
        )

        self.content_frame = ttk.Frame(self.canvas)
        self.content_frame.bind(
            "<Configure>",
            lambda e: self.canvas.configure(scrollregion=self.canvas.bbox("all"))
        )

        self.canvas.create_window((0, 0), window=self.content_frame, anchor="nw")
        self.canvas.configure(yscrollcommand=self.scrollbar.set)

        self.canvas.pack(side="left", fill="both", expand=True)
        self.scrollbar.pack(side="right", fill="y")
        self.content_frame.columnconfigure(1, weight=1)

    def _creer_variables_tk(self):
        """Initialise les variables Tkinter pour stocker les valeurs."""
        cfg = self.cfg

        # Variables géographiques
        self.lat = tk.DoubleVar(value=cfg.LAT if cfg else 0.0)
        self.lon = tk.DoubleVar(value=cfg.LON if cfg else 0.0)
        self.boxkm = tk.DoubleVar(value=cfg.BBOX_SIZE_KM if cfg else 0.0)
        self.site = tk.StringVar(value=cfg.SITE_NAME if cfg else "")
        self.epsg = tk.IntVar(value=cfg.EPSG_CIBLE if cfg else 0)

        # Variables MNT et hydrologie
        self.nodata = tk.DoubleVar(value=cfg.NODATA_VALUE if cfg else 0.0)
        self.thrkm2 = tk.DoubleVar(value=cfg.STREAM_THRESHOLD_KM2 if cfg else 0.0)

        # Variables d'environnement
        self.api = tk.StringVar(value=cfg.OPENTOPOGRAPHY_API_KEY if cfg else "")
        self.grass_base = tk.StringVar(value=cfg.GRASS_GISBASE if cfg else "")
        self.qgis = tk.StringVar(value=cfg.QGIS_PATH if cfg else "")

        # Variables GDAL/PROJ
        self.gdalwarp = tk.StringVar(value=cfg.GDALWARP_CMD if cfg else "")
        self.gdal_data = tk.StringVar(value=cfg.GDAL_DATA_EXT if cfg else "")
        self.proj = tk.StringVar(value=cfg.PROJ_LIB_EXT if cfg else "")
        self.gdal_bin = tk.StringVar(value=cfg.GDAL_BIN_EXT if cfg else "")

        # Variables de chemins
        self.outdir = tk.StringVar(value=cfg.OUTPUT_DIR if cfg else "")
        self.tmpdir = tk.StringVar(value=cfg.TEMP_DIR if cfg else "")
        self.grassdb = tk.StringVar(value=cfg.GRASS_DB_DIR if cfg else "")

        # Mode développeur
        self.dev = tk.BooleanVar(value=bool(cfg.DEV_MODE) if cfg else False)

    def _organiser_widgets(self):
        """Positionne les widgets dans l'interface."""
        row = 0

        # Section Paramètres géographiques
        self._ajouter_section("Paramètres géographiques", row)
        row = self._ajouter_champs_geographiques(row + 1)

        # Section Paramètres MNT & hydrologie
        self._ajouter_section("Paramètres MNT & hydrologie", row)
        row = self._ajouter_champs_mnt(row + 1)

        # Section Clés API
        self._ajouter_section("Clés et accès API", row)
        row = self._ajouter_champs_api(row + 1)

        # Section Chemins d'installation
        self._ajouter_section("Chemins d'installation", row)
        row = self._ajouter_champs_installation(row + 1)

        # Section Composants GDAL/PROJ
        self._ajouter_section("Composants GDAL/PROJ", row)
        row = self._ajouter_champs_gdal(row + 1)

        # Section Dossiers de travail
        self._ajouter_section("Dossiers de travail et sortie", row)
        row = self._ajouter_champs_dossiers(row + 1)

        # Checkbox mode développeur
        self._ajouter_checkbox_dev(row)
        row += 1
```

```
        # Boutons d'action
        self._ajouter_boutons_actions(row)

    def _ajouter_section(self, titre: str, row: int):
        """Ajoute un titre de section."""
        ttk.Label(
            self.content_frame,
            text=titre,
            font=("Arial", 12, "bold")
        ).grid(row=row, columnspan=3, pady=(10, 5), sticky="w")

    def _ajouter_champs_geographiques(self, row: int) -> int:
        """Ajoute les champs pour les paramètres géographiques."""
        self._creer_champ_entree("Latitude (°)", self.lat, row)
        self._creer_champ_entree("Longitude (°)", self.lon, row + 1)
        self._creer_champ_entree("Demi-côté de la bbox (km)", self.boxkm, row + 2)
        self._creer_champ_entree("Nom du site", self.site, row + 3)
        self._creer_champ_entree("EPSG cible", self.epsg, row + 4)
        return row + 5

    def _ajouter_champs_mnt(self, row: int) -> int:
        """Ajoute les champs pour les paramètres MNT."""
        self._creer_champ_entree("Valeur NoData", self.nodata, row)
        self._creer_champ_entree("Seuil cours d'eau (km²)", self.thrkm2, row + 1)
        return row + 2

    def _ajouter_champs_api(self, row: int) -> int:
        """Ajoute le champ pour la clé API."""
        self._creer_champ_entree("OpenTopography API", self.api, row)
        return row + 1

    def _ajouter_champs_installation(self, row: int) -> int:
        """Ajoute les champs pour les chemins d'installation."""
        self._creer_champ_chemin("GRASS GISBASE", self.grass_base, row)
        self._creer_champ_chemin("QGIS (optionnel)", self.qgis, row + 1)
        return row + 2

    def _ajouter_champs_gdal(self, row: int) -> int:
        """Ajoute les champs pour les composants GDAL."""
        self._creer_champ_chemin("gdalwarp.exe", self.gdalwarp, row, False)
        self._creer_champ_chemin("GDAL_DATA", self.gdal_data, row + 1)
        self._creer_champ_chemin("PROJ_LIB", self.proj, row + 2)
        self._creer_champ_chemin("GDAL bin", self.gdal_bin, row + 3)
        return row + 4

    def _ajouter_champs_dossiers(self, row: int) -> int:
        """Ajoute les champs pour les dossiers de travail."""
        self._creer_champ_chemin("Dossier de sortie", self.outdir, row)
        self._creer_champ_chemin("Dossier temporaire", self.tmpdir, row + 1)
        self._creer_champ_chemin("Répertoire GRASSDATA", self.grassdb, row + 2)
        return row + 3

    def _ajouter_checkbox_dev(self, row: int):
        """Ajoute la checkbox pour le mode développeur."""
        ttk.Checkbutton(
            self.content_frame,
            text="Mode développeur (conserver les temporaires)",
            variable=self.dev
        ).grid(row=row, columnspan=3, pady=10, sticky="w")

    def _ajouter_boutons_actions(self, row: int):
        """Ajoute les boutons d'action principaux."""
        frame_boutons = ttk.Frame(self.content_frame)
        frame_boutons.grid(row=row, columnspan=3, sticky="w", pady=10)

        ttk.Button(
            frame_boutons,
            text="Charger 2 YAML…",
            command=self._charger_yaml
        ).pack(side="left", padx=(0, 6))

        ttk.Button(
            frame_boutons,
            text="Enregistrer config.yaml…",
            command=self._sauvegarder_yaml
        ).pack(side="left", padx=(0, 6))

        ttk.Button(
            frame_boutons,
            text="Lancer le pipeline",
            command=self._executer_pipeline
        ).pack(side="left")

    # ──────────────────────────────────────────────────────────────
    # Méthodes utilitaires pour les widgets
    # ──────────────────────────────────────────────────────────────

    def _creer_champ_entree(self, label: str, variable: tk.Variable, row: int):
        """Crée un champ d'entrée avec étiquette."""
        ttk.Label(self.content_frame, text=label).grid(
            row=row, column=0, sticky="w", pady=2, padx=5
```

```
        )
        ttk.Entry(self.content_frame, textvariable=variable, width=50).grid(
            row=row, column=1, sticky="ew", pady=2, padx=5
        )

    def _creer_champ_chemin(self, label: str, variable: tk.Variable, row: int, est_repertoire: bool = True):
        """Crée un champ de sélection de chemin."""
        self._creer_champ_entree(label, variable, row)
        ttk.Button(
            self.content_frame,
            text="Parcourir",
            command=lambda: self._selectionner_chemin(variable, est_repertoire)
        ).grid(row=row, column=2, sticky="e", pady=2, padx=5)

    @staticmethod
    def _selectionner_chemin(variable: tk.Variable, est_repertoire: bool):
        """Ouvre un dialogue de sélection de fichier/dossier."""
        chemin = filedialog.askdirectory() if est_repertoire else filedialog.askopenfilename()
        if chemin:
            variable.set(chemin)

    # ------------------------------------------------------------------
    # Gestion de la configuration
    # ------------------------------------------------------------------

    def _collecter_config(self) -> Config:
        """Crée un objet Config à partir des valeurs de l'interface."""
        return Config(
            LAT=self.lat.get(),
            LON=self.lon.get(),
            BBOX_SIZE_KM=self.boxkm.get(),
            SITE_NAME=self.site.get(),
            EPSG_CIBLE=self.epsg.get(),
            NODATA_VALUE=self.nodata.get(),
            STREAM_THRESHOLD_KM2=self.thrkm2.get(),
            GRASS_GISBASE=self.grass_base.get(),
            QGIS_PATH=self.qgis.get(),
            GDALWARP_CMD=self.gdalwarp.get(),
            GDAL_DATA_EXT=self.gdal_data.get(),
            PROJ_LIB_EXT=self.proj.get(),
            GDAL_BIN_EXT=self.gdal_bin.get(),
            OUTPUT_DIR=self.outdir.get(),
            TEMP_DIR=self.tmpdir.get(),
            GRASS_DB_DIR=self.grassdb.get(),
            OPENTOPOGRAPHY_API_KEY=self.api.get(),
            DEV_MODE=self.dev.get(),
        )

    def _appliquer_config(self, cfg: Config):
        """Applique une configuration à l'interface."""
        self.lat.set(cfg.LAT)
        self.lon.set(cfg.LON)
        self.boxkm.set(cfg.BBOX_SIZE_KM)
        self.site.set(cfg.SITE_NAME)
        self.epsg.set(cfg.EPSG_CIBLE)
        self.nodata.set(cfg.NODATA_VALUE)
        self.thrkm2.set(cfg.STREAM_THRESHOLD_KM2)
        self.api.set(cfg.OPENTOPOGRAPHY_API_KEY)
        self.grass_base.set(cfg.GRASS_GISBASE)
        self.qgis.set(cfg.QGIS_PATH)
        self.gdalwarp.set(cfg.GDALWARP_CMD)
        self.gdal_data.set(cfg.GDAL_DATA_EXT)
        self.proj.set(cfg.PROJ_LIB_EXT)
        self.gdal_bin.set(cfg.GDAL_BIN_EXT)
        self.outdir.set(cfg.OUTPUT_DIR)
        self.tmpdir.set(cfg.TEMP_DIR)
        self.grassdb.set(cfg.GRASS_DB_DIR)
        self.dev.set(cfg.DEV_MODE)

    def _charger_yaml(self):
        """Charge les fichiers de configuration YAML."""
        chemin_config = filedialog.askopenfilename(
            title="Choisir config.yaml",
            filetypes=[("YAML", "*.yml *.yaml")],
        )
        if not chemin_config:
            return

        chemin_defaut = filedialog.askopenfilename(
            title="Choisir default_config.yaml",
            filetypes=[("YAML", "*.yml *.yaml")],
        )
        if not chemin_defaut:
            return

        try:
            cfg = from_yaml_pair(chemin_config, chemin_defaut)
            self._appliquer_config(cfg)
            self.cfg = cfg
            messagebox.showinfo(
                "Configuration",
                f"Configurations chargées avec succès :\n- {chemin_config}\n- {chemin_defaut}"
```

```
            )
        except Exception as e:
            messagebox.showerror("Erreur", f"Impossible de charger les YAML : {e}")

    def _sauvegarder_yaml(self):
        """Sauvegarde la configuration actuelle dans un fichier YAML."""
        if self.cfg is None:
            messagebox.showerror("Erreur", "Aucune configuration chargée.")
            return

        chemin = filedialog.asksaveasfilename(
            defaultextension=".yaml",
            filetypes=[("YAML", "*.yaml"), ("YML", "*.yml")],
        )
        if not chemin:
            return

        try:
            save_editable_yaml(self._collecter_config(), chemin)
            messagebox.showinfo("Configuration", f"Fichier enregistré : {chemin}")
        except Exception as e:
            messagebox.showerror("Erreur", f"Impossible d'enregistrer : {e}")

    # ──────────────────────────────────────────────────────────────
    # Exécution du pipeline
    # ──────────────────────────────────────────────────────────────

    def _executer_pipeline(self):
        """Lance l'exécution du pipeline dans un thread séparé."""
        cfg = self._collecter_config()
        logger = self.logger

        # Création de la fenêtre de logs
        self.withdraw()
        fenetre_logs = tk.Toplevel(self)
        fenetre_logs.title("Exécution du Pipeline")
        zone_texte = tk.Text(fenetre_logs, wrap="word", width=80, height=24)
        zone_texte.pack(fill="both", expand=True, padx=10, pady=10)

        # Configuration du handler de logs
        import logging
        class HandlerLogsTexte(logging.Handler):
            def __init__(self, widget):
                super().__init__()
                self.widget = widget
                self.setFormatter(logging.Formatter("%(asctime)s [%(levelname)s] %(message)s"))

            def emit(self, record):
                try:
                    message = self.format(record) + "\n"
                    self.widget.insert(tk.END, message)
                    self.widget.see(tk.END)
                    self.widget.update_idletasks()
                except Exception:
                    pass

        handler = HandlerLogsTexte(zone_texte)
        logger.addHandler(handler)

        def _travail():
            """Fonction exécutée dans le thread pour le pipeline."""
            dossier_temp = None
            base_grass = None

            try:
                # Validation de l'environnement
                infos_env = validate_environment(cfg, logger)

                # Préparation des dossiers
                bbox = calculate_bbox_wgs84(cfg.LAT, cfg.LON, cfg.BBOX_SIZE_KM)
                session = f"{cfg.SITE_NAME}_{datetime.now().strftime('%Y%m%d_%H%M%S')}"
                dossier_sortie = os.path.join(cfg.OUTPUT_DIR, session)
                dossier_temp = os.path.join(cfg.TEMP_DIR, session)
                base_grass = os.path.join(cfg.GRASS_DB_DIR, f"hydro_grass_db_{session}")

                os.makedirs(dossier_sortie, exist_ok=True)
                os.makedirs(dossier_temp, exist_ok=True)
                os.makedirs(cfg.GRASS_DB_DIR, exist_ok=True)
                logger.info("Dossiers préparés : temp=%s | db=%s | out=%s",
                            dossier_temp, base_grass, dossier_sortie)

                # Étapes du pipeline
                mnt_brut = telecharger_mnt(bbox, dossier_temp, cfg.OPENTOPOGRAPHY_API_KEY, logger)
                mnt_propre = pretraiter_mnt(
                    mnt_brut, dossier_temp, cfg.EPSG_CIBLE, cfg.NODATA_VALUE,
                    infos_env["GDAL_ENV"], cfg.GDALWARP_CMD, logger
                )

                gi = init_grass_modules(infos_env["GRASS_PYTHON_PATH"], logger)
                initialiser_grass(
                    gi, infos_env["GRASS_CMD"], cfg.GRASS_DB_DIR,
                    f"hydro_loc_{session}", "PERMANENT", cfg.EPSG_CIBLE, logger
```

```
                )

                x_out, y_out = analyse_hydro_grass(
                    gi, mnt_propre, cfg.LAT, cfg.LON,
                    cfg.STREAM_THRESHOLD_KM2, 4326, cfg.EPSG_CIBLE, logger
                )

                # Utilisation explicite pour éviter l'avertissement "non utilisé"
                logger.info("Exutoire (SCR cible) : (%.2f, %.2f)", x_out, y_out)

                export_grass_results(gi, dossier_sortie, logger)
                self.after(
                    0,
                    messagebox.showinfo,
                    "Succès",
                    f"Pipeline terminé.\nRésultats : {dossier_sortie}"
                )


            except Exception as e:
                self.after(
                    0,
                    messagebox.showerror,
                    "Échec",
                    f"Le pipeline a échoué : {e}"
                )


            finally:
                # Nettoyage
                try:
                    logger.removeHandler(handler)
                except Exception:
                    pass
                fenetre_logs.destroy()
                self.deiconify()

                if not cfg.DEV_MODE:
                    self._nettoyer_temporaires(dossier_temp, base_grass, logger=logger)

        threading.Thread(target=_travail, daemon=True).start()

    @staticmethod
    def _nettoyer_temporaires(*chemins, logger=None):
        """Nettoie les fichiers temporaires si nécessaire."""
        for chemin in chemins:
            try:
                if chemin and os.path.isdir(chemin):
                    shutil.rmtree(chemin)
                    if logger:
                        logger.info("Temporaire supprimé : %s", chemin)
            except Exception as e:
                if logger:
                    logger.warning("Échec suppression du temporaire %s : %s", chemin, e)
                continue


# ----------------------------------------------------------------------------
# hydro_pipeline/main.py
# ----------------------------------------------------------------------------
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Point d'entrée principal du pipeline hydrologique.

Fournit deux modes d'exécution :
1. En tant que module : `python -m hydro_pipeline.main`
2. En tant que script : `python hydro_pipeline/main.py`

Gère automatiquement les chemins d'import pour les deux modes.
"""

from __future__ import annotations
import argparse
import sys
import os
import logging
from typing import Optional

# ----------------------------------------------------------------------------
# Configuration des imports selon le mode d'exécution
# ----------------------------------------------------------------------------

def _ajuster_chemins_import():
    """Ajuste sys.path pour permettre l'exécution directe du script."""
    if __package__ in (None, ""):
        dossier_parent = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
        if dossier_parent not in sys.path:
            sys.path.insert(0, dossier_parent)

# Applique l'ajustement avant les imports
_ajuster_chemins_import()
```

```python
# Import conditionnel selon le mode
if __package__ in (None, ""):
    import hydro_pipeline.gui as gui
    import hydro_pipeline.logging_setup as logging_setup
    import hydro_pipeline.config_io as config_io
else:
    from . import gui, logging_setup, config_io

# ──────────────────────────────────────────────────────────────
# Fonction principale
# ──────────────────────────────────────────────────────────────

def analyser_arguments() -> argparse.Namespace:
    """Configure et analyse les arguments en ligne de commande."""
    analyseur = argparse.ArgumentParser(
        description="Pipeline Hydrologique - Interface de configuration"
    )
    analyseur.add_argument(
        "--config",
        type=str,
        help="Chemin vers le fichier config.yaml",
        default=None
    )
    analyseur.add_argument(
        "--default",
        type=str,
        help="Chemin vers le fichier default_config.yaml",
        default=None
    )
    return analyseur.parse_args()

def charger_configuration(config_path: str, default_path: str, logger: logging.Logger) -> Optional["Config"]:
    """Tente de charger une configuration à partir des fichiers YAML."""
    try:
        return config_io.from_yaml_pair(config_path, default_path)
    except Exception as e:
        logger.error("Échec du chargement des configurations YAML : %s", e)
        return None


def lancer_interface(configuration: Optional[gui.PipelineConfigGUI] = None):
    """Initialise et lance l'interface graphique."""
    application = gui.PipelineConfigGUI(cfg=configuration)
    application.mainloop()

def main() -> None:
    """Point d'entrée principal de l'application."""
    # Configuration initiale
    args = analyser_arguments()
    logger = logging_setup.setup_logging()

    # Gestion des différents cas d'utilisation
    if args.config and args.default:
        configuration = charger_configuration(args.config, args.default, logger)
        if configuration:
            logger.info("Configuration préchargée avec succès")
        else:
            logger.info("Lancement avec configuration par défaut")
    else:
        logger.info("Lancement sans configuration préchargée")
        configuration = None

    # Lancement de l'interface
    lancer_interface(configuration)

# ──────────────────────────────────────────────────────────────
# Point d'entrée du script
# ──────────────────────────────────────────────────────────────

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