Documentation LakeRes

Suivi de thèse de Bastien Boivin

Version Draft – Document de Travail

Auteur : Bastien Boivin

Email (pro) : bastien.boivin@univ-rennes.fr
Email (perso) : bastien.boivin@proton.me

Directeur de thèse :
Jean-Raynald de Dreuzy, Directeur de recherche CNRS, Géosciences Rennes

Co-directeur de thèse :
Luc Aquilina, Professeur des universités, Géosciences Rennes

Partenaire industriel :
Jean-Yves Gaubert, Directeur du pôle R&D, Eau du Bassin Rennais

Rennes, 30 avril 2025

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

1.1 Objectifs du document

Ce document a pour but de fournir une documentation technique dans le cadre de mon doctorat. Il est conçu pour expliquer les concepts, les méthodes et les résultats de mes recherches, en passant par la bibliographie, les résultats, les concepts ainsi que l'explication du code développé au sein d'HydroModPy, initié par Alexandre Coche.

1.2 Contexte du projet

1.3 Guide d'utilisation

Chapitre 2

Bibliographie

2.1 Modflow

2.1.1 Modflow NWT

Modflow NWT est une version de Modflow qui intègre un solveur non linéaire pour simuler des conditions de flux d'eau souterraine. Il est particulièrement utile pour modéliser des aquifères avec des conditions de recharge variable et des niveaux d'eau fluctuants.

2.1.2 Package DRN (drain)

2.1.3 Package SFR (streamflow-routing)

2.2 Fuite du Lac (Leakage)

Chapitre 3

Données

3.1 DEM

3.1.1 BD-ALTI-75m

3.2 Climatiques (passé)

3.3 Projections climatiques

3.4 Hydrologie

3.4.1 Stations de jaugeage

3.4.2 Hydrographie

3.4.3 Intermittence

3.5 Géologie

3.6 Données EBR

3.6.1 Abaque | Bathymétrie

3.6.2 Données journalières

3.6.3 Scénarios de gestion

Chapitre 4

Code - EBR

4.1 App EBR commun.py

4.1.1 Chargements des bibliothèques, modules et du dossier racines

Cette section permet l'importation de l'ensemble des librairies utilisées par le code, dont celles de Python, celles de librairies externes et les codes d'HydroModPy fonctionnant en POO (programmation orientée objet). Ces différentes librairies sont toutes incluses dans l'environnement `Hydromodpy-0.1` préalablement installé.

En amont de ces librairies, une section `## Filtrer les avertissements` est à renseigner à chaque début de code afin que les alertes de `DeprecationWarnings` ne s'affichent pas, voir 6.1.

4.1.2 LogManager

La `class LogManager` permet de gérer l'interface verbale entre l'utilisateur et le code, en faisant remonter des logs selon différentes classes avec plus ou moins de précisions et de messages selon le mode choisi. Pour paramétrer le `LogManager`, voir la section 5.2.1.

4.2 Initialisation de la classe climatiques

4.2.1 Réanalyse Surfex

4.2.2 Méthode de création d'un csv pour données climatiques

En temps normal, HydroModPy (à l'échelle de la France) fonctionne automatiquement avec les données SIM2. Pour la Bretagne, la recharge et le runoff sont modifiés à partir des données de réanalyse. Ici, des données ISBA brutes issues du serveur FTP de Météo-France sont utilisées directement.

Ce procédé nécessite de fusionner des fichiers NetCDF à chaque itération, ce qui est coûteux en calcul. De plus, les données de réanalyses doivent être extraites dans chaque dossier de sortie, sauf si elles sont externalisées au préalable.

Une méthode plus simple consiste à exécuter une dernière fois la méthode classique, puis à créer un `DataFrame` pour exporter l'ensemble des données climatiques, comme ci-dessous :

```
#============================================================================
# Exportation des données climatiques
# ============================================================================
# df_climatic = pd.DataFrame({
#     'recharge': BV.climatic.recharge,
#     'runoff': BV.climatic.runoff,
#     'precip': BV.climatic.precip,
#     'evt': BV.climatic.evt,
#     'etp': BV.climatic.etp,
#     't': BV.climatic.t,
# })
# df_climatic.to_csv(os.path.join(data_path, 'Meteo', 'Historiques SIM2', 'climatic_data.csv'))
```

Ensuite, toute la classe climatique peut etre mise en commentaire afin de ne garder que la lecture du CSV précédemment créé, comme ci-dessous :

```python
df_climatic = pd.read_csv(
    os.path.join(data_path, 'Meteo', 'Historiques SIM2', 'climatic_data.csv'),
    index_col=0, parse_dates=True
)
df_climatic.index = pd.to_datetime(df_climatic.index)
df_climatic = df_climatic.loc[
    (df_climatic.index >= pd.Timestamp("01/01/{}".format(first_year))) &
    (df_climatic.index <= pd.Timestamp("31/12/{}".format(last_year)))
]

agg_dict = {
    'recharge': 'sum',
    'runoff': 'sum',
    'precip': 'sum',
    'evt': 'sum',
    'etp': 'sum',
    't': 'mean'
}
df_climatic = df_climatic.resample(freq_input).agg(agg_dict)

BV.climatic.recharge = df_climatic['recharge']
BV.climatic.runoff = df_climatic['runoff']
BV.climatic.precip = df_climatic['precip']
BV.climatic.evt = df_climatic['evt']
BV.climatic.etp = df_climatic['etp']
BV.climatic.t = df_climatic['t']

first_clim = BV.climatic.recharge[0]
BV.climatic.update_first_clim(first_clim)
```

> **Remarque :** Il est conseillé d'exporter le fichier en données journalières, puis de procéder à la réanalyse (hebdomadaire, mensuelle, etc.) lors de l'import. La sélection automatique des dates minimale et maximale peut être réalisée à l'aide des arguments déjà renseignés.

4.3 Paramétrisation

4.3.1 Simplex de Nelder-Mead

Le Simplex de Nelder-Mead est un algorithme d'optimisation non-linéaire adapté aux problèmes où le calcul des dérivées est complexe. Son principe repose sur la manipulation d'une figure géométrique à $N+1$ sommets dans un espace à N dimensions.

Principe et enchaînement des opérations

L'algorithme utilise quatre opérations géométriques principales qui s'enchaînent selon un arbre de décision précis. À chaque itération, les valeurs de la fonction objectif aux sommets sont d'abord ordonnées :

$$f(x_1) \leq f(x_2) \leq \ldots \leq f(x_{N+1}) \tag{4.1}$$

Où x_1 est le meilleur sommet et x_{N+1} le pire. Le centroïde des N meilleurs sommets est calculé comme $x_0 = \frac{1}{N}\sum_{i=1}^{N} x_i$.

L'enchaînement des opérations suit alors la logique suivante :

Enchaînement des opérations dans une itération du Simplex de Nelder-Mead :

1. **Réflexion** (toujours effectuée en premier) :
   - Calculer $x_r = x_0 + \alpha(x_0 - x_{N+1})$ et évaluer $f_r = f(x_r)$
2. **Décision** (une seule branche est suivie) :
   - Si $f(x_1) \leq f_r < f(x_N)$:
     Remplacer x_{N+1} par x_r (simple acceptation de la réflexion)
   - Si $f_r < f(x_1)$:
     Expansion : Calculer $x_e = x_0 + \beta(x_r - x_0)$ et évaluer $f_e = f(x_e)$
     Si $f_e < f_r$: Remplacer x_{N+1} par x_e
     Sinon : Remplacer x_{N+1} par x_r
   - Si $f_r \geq f(x_N)$ et $f_r < f(x_{N+1})$:
     Contraction externe : Calculer $x_c = x_0 + \gamma(x_r - x_0)$ et évaluer $f_c = f(x_c)$
     Si $f_c \leq f_r$: Remplacer x_{N+1} par x_c
     Sinon : Appliquer **Rétrécissement**
   - Si $f_r \geq f(x_{N+1})$:
     Contraction interne : Calculer $x_c = x_0 + \gamma(x_{N+1} - x_0)$ et évaluer $f_c = f(x_c)$
     Si $f_c < f(x_{N+1})$: Remplacer x_{N+1} par x_c
     Sinon : Appliquer **Rétrécissement**
3. **Rétrécissement** (seulement si la contraction a échoué) :
   - Pour $i = 2, \ldots, N+1$, remplacer x_i par $x_1 + \delta(x_i - x_1)$

Où les coefficients standards sont $\alpha = 1$ (réflexion), $\beta = 2$ (expansion), $\gamma = 0.5$ (contraction) et $\delta = 0.5$ (rétrécissement). Dans notre implémentation adaptative, ils dépendent de la dimension N du problème :

$$\alpha = 1, \quad \beta = 1 + \frac{2}{N}, \quad \gamma = 0.75 - \frac{0.5}{N}, \quad \delta = 1 - \frac{1}{N} \tag{4.2}$$

Points importants à noter :

- Une seule des branches de l'arbre de décision est suivie à chaque itération
- Le rétrécissement n'est appliqué qu'en dernier recours, si les contractions échouent
- Pour les problèmes de grande dimension, les opérations de réflexion deviennent dominantes, réduisant l'efficacité de l'algorithme

Normalisation et mise à l'échelle des paramètres

Pour garantir une convergence efficace, nous normalisons tous les paramètres dans l'intervalle [0,1] avant optimisation :

$$x_{norm} = \frac{x - x_{min}}{x_{max} - x_{min}} \tag{4.3}$$

Cette normalisation est particulièrement importante pour la conductivité hydraulique (K) qui varie sur plusieurs ordres de grandeur. Pour ce paramètre, nous utilisons une échelle logarithmique :

$$K_{norm} = \frac{\log_{10}(K) - \log_{10}(K_{min})}{\log_{10}(K_{max}) - \log_{10}(K_{min})} \tag{4.4}$$

Implémentation dans le code

L'algorithme est intégré via la bibliothèque SciPy :

```python
from scipy.optimize import minimize

# Normalisation des paramètres
def normalize(x, xmin, xmax):
    """Normalise une valeur x selon les bornes xmin et xmax"""
    return (x - xmin) / (xmax - xmin)

def denormalize(x_norm, xmin, xmax):
    """Dénormalise une valeur x_norm selon les bornes xmin et xmax"""
    return x_norm * (xmax - xmin) + xmin

# Fonction objectif normalisée
def erreur_modele_norm(params_norm):
    # Dénormalisation des paramètres
    log_hk_value = denormalize(params_norm[0], log_hk_min, log_hk_max)
    hk_value = 10**log_hk_value
    sy_value = denormalize(params_norm[1], sy_min, sy_max)
    thick_value = denormalize(params_norm[2], thick_min, thick_max)

    # Mise à jour du modèle avec les nouveaux paramètres
    BV.hydraulic.update_hk(hk_value)
    BV.hydraulic.update_sy(sy_value)
    BV.hydraulic.update_thick(thick_value)

    # Simulation du modèle avec ces paramètres
    model_modflow = BV.preprocessing_modflow()
    success_modflow = BV.processing_modflow(model_modflow)
    BV.postprocessing_timeseries(model_modflow)

    # Calcul du critère de Nash-Sutcliffe
    nse = 1 - (numerator / denominator)
    return 1 - nse  # On minimise 1-NSE

# Exécution de l'optimisation
result = minimize(
    erreur_modele_norm,
    x0_norm,  # Paramètres initiaux normalisés
    method='Nelder-Mead',
    options={
        'xatol': 0.01,  # Tolérance sur les paramètres
        'fatol': 0.01,  # Tolérance sur la fonction
        'maxiter': 200,  # Nombre max d'itérations
        'disp': True     # Affichage des informations
    }
)
```

Paramètres calibrés

Dans notre implémentation, trois paramètres hydrogéologiques fondamentaux sont calibrés :

Table 4.1 Paramètres hydrogéologiques calibrés et leurs bornes

Paramètre	Minimum	Maximum	Unité	Influence
Conductivité (K)	10^{-6}	10^{-3}	m/s	Capacité de l'aquifère à transmettre l'eau
Porosité efficace (S_y)	0.001	0.1	-	Capacité de stockage
Épaisseur (e)	20	40	m	Volume de l'aquifère disponible

Le Simplex est particulièrement adapté pour ce nombre limité de paramètres. Sa convergence reste efficace et rapide pour des problèmes de dimension inférieure à 5.

Fonction objectif et sélection des données

La fonction objectif utilisée est le critère de Nash-Sutcliffe (NSE) :

$$\text{NSE} = 1 - \frac{\sum_{t=1}^{T}(Q_{obs,t} - Q_{sim,t})^2}{\sum_{t=1}^{T}(Q_{obs,t} - \bar{Q}_{obs})^2} \tag{4.5}$$

Notre implémentation permet de sélectionner précisément les données utilisées pour la calibration :

```python
def filter_dates(dates):
    """Filtre les dates selon des critères temporels et saisonniers"""
    mask = pd.Series(True, index=dates)

    if use_time_filter:
        mask = mask & (dates >= calib_start_date) & (dates <= calib_end_date)

        if use_seasonal_filter:
            def is_in_season(date):
                start = pd.Timestamp(date.year, season_start_month, season_start_day)
                if season_end_month < season_start_month:
                    end = pd.Timestamp(date.year + 1, season_end_month, season_end_day)
                else:
                    end = pd.Timestamp(date.year, season_end_month, season_end_day)
                return (date >= start) & (date <= end)

            seasonal_mask = dates.map(is_in_season)
            mask = mask & seasonal_mask

    return mask
```

Cette approche permet de concentrer la calibration sur des périodes représentatives, en excluant si nécessaire des événements extrêmes ou des saisons particulières.

Avantages et limitations

Dans notre contexte hydrogéologique, le Simplex offre un excellent compromis entre simplicité d'implémentation et efficacité de calibration pour les principaux paramètres qui contrôlent le comportement hydraulique du modèle.

Table 4.2 Avantages et limitations du Simplex dans notre contexte

Avantages	**Limitations**
Ne nécessite pas le calcul des dérivées	Peut converger vers des minima locaux
Robuste face aux irrégularités de la fonction objectif	Nombre d'itérations potentiellement élevé
Implémentation simple via SciPy	Sensible à l'initialisation du simplexe
Adapté aux problèmes avec peu de paramètres	Performances réduites au-delà de 5 paramètres

Chapitre 5

HydroModPy

5.1 watershed_root.py

5.2 toolbox.py

5.2.1 class LogManager

Le `LogManager` est conçue pour configurer et gérer la journalsiation de l'application de manière flexible et adaptable.

Initialisation du LogManager : Pour intégrer le `LogManager` dans un script, il suffit d'insérer les lignes suivantes :

```
# Initialisation du LogManager en mode developpement
log_manager = toolbox.LogManager(
    mode="dev", # Utilisez mode="verbose" pour afficher les logs INFO et superieurs, et mode="quiet" pour afficher les logs WARNING et superieurs
    log_dir=root_dir, # Specifiez le repertoire de journalisation
    overwrite=False, # Utilisez overwrite=True (par defaut) pour ecraser les fichiers de log existants
    verbose_libraries=True # Utilisez verbose_libraries=True pour afficher les logs des bibliotheques (avertissements et superieurs, generalement masques)
)
```

Mode de fonctionnement :

- Mode `dev` :
  - Console : Affiche tous les messages de niveau DEBUG et supérieur (DEBUG, INFO, WARNING, ERROR, CRITICAL).
  - Format : `\%([levelname)s] [\%(name)s] [\%(module)s:\%(lineno)d] \%(message)s`
- Mode `verbose` :
  - Console : Affiche tous les messages de niveau INFO et supérieur (INFO, WARNING, ERROR, CRITICAL).
  - Format : `\%([levelname)s] \%(message)s`
- Mode `quiet` :
  - Console : Affiche uniquement les messages de niveau WARNING et supérieur (WARNING, ERROR, CRITICAL).
  - Format : `\%([levelname)s] \%(message)s`

Gestion des bibliothèques externes :

Par défaut, le `LogManager` supprime les logs provenant de certianes bbliothèques externes pour éviter un terminal (kernel) surchargé. Voici la liste des bibliothèques dont les logs sont réduits au niveau CRITICAL :

```
libraries_to_silence = [
    "fiona",
    "rasterio",
    "urllib3",
```

```
    "geopy",
    "matplotlib",
    "PIL"
]
```

Vous pouvez activer les logs des bibliothèques externes en définissant `verbose_libraries=True` lors de l'initialisation. Dans ce cas, les messages de niveau WARNING et supérieur seront affichés pour ces bibliothèques.

Sauvegarde des Logs :

- **Fichier de log** : Un fichier `dev.log` est automatiquement sauvegardé dans le dossier `dev.log` à la racine du projet.
- **Format** : Les logs sont enregistrés dans le format `dev` pour inclure la provenance des messages (fichier et numéro de ligne).
- **Écrasement** : Par défaut, le fichier est écrasé à chaque nouvelle exécution. Pour ajouter les logs successifs, utilisez `overwrite=False`.

Logique des niveaux de Logging :

Les scripts situés dans `src/` ont été mis à jour pour respecter la logique suivante :

- `logging.debug` : Points d'étape détaillés (peut générer beaucoup de lignes, notamment dans les boucles).
- `logging.info` : Messages classiques équivalents aux `print`.
- `logging.warning` : Avertissements nécessitant une attention particulière de l'utilisateur ou signalant une erreur mineure sans arrêt du code.
- `logging.error` : Erreurs mettant fin à l'exécution du script.
- `logging.critical` : Actuellement non utilisé.

Execptions :

Certains `print` sont conservés pour des raisons spécifiques :

- Affichage du logo d'HydroModPy.
- Décompte des étapes (ex. "Étape 1/51") afin de ne pas surcharger le terminal.

Actuellement, les `print` dans les fichiers d'exécution, comme les exemples, n'ont pas été mis à jour. Il reste à discuter si nous les conservons en tant que `print` ou si nous les remplaçons par des logs de niveau `logging.info()`.

Changement de syntaxe pour le Logging

La syntaxe utilisée pour les messages de logs a été modifiée, car le module `logging` ne permet pas d'insérer directement plusieurs variables dans une chaîne de caractères, comme c'est possible avec un simple `print` (par exemple : `print("Exemple" + A + B)` ou `print("Exemple", A, B)`). Pour formater les messages dans le contexte de logging, deux approches sont possibles :

- Utilisation des f-strings :

  - `logging.debug(f"Etape : {i} / {len(x)}")`
- Utilisation des Specificateurs de Format, associés aux variables dans l'ordre :
  - `logging.debug("Etape : \%s / \%s", i, len(x))`
    * Liste des principaux spécificateurs utiles :
      — `\%s` : Pour les chaînes de caractères.
      — `\%d` : Pour les entiers.
      — `\%f` : Pour les nombres à virgule flottante.

Chapitre 6

Patch

6.1 DeprecationWarnings

Les `DeprecationWarning` sont affichés dans le kernel lorsque des méthodes ou définitions d'une bibliothèque Python sont appelées et que ces dernières vont être supprimées dans une prochaine version. HydroModPy étant actuellement basé sur une version 3.8.10 de Python (version actuelle 3.13), beaucoup de `DeprecationWarning` apparaissent. Pour éviter cela, les quatre lignes ci-dessous sont à inclure en début de script.

> Supprimer l'affichage de ces messages ne pose aucun problème de fonctionnement à l'exécution du code.

```
# Filtrer les avertissements (avant les imports)
import warnings
warnings.filterwarnings('ignore', category=DeprecationWarning)

import pkg_resources # A placer apres DeprecationWarning car elle meme obsolète...
warnings.filterwarnings('ignore', message='.*pkg_resources.*')
warnings.filterwarnings('ignore', message='.*declare_namespace.*')
```

6.2 Suppression des fichiers.chk

À ce jour, je n'ai trouvé aucune information dans la bibliographie de Flopy permettant de désactiver la création des fichiers `*.chk`. Ces fichiers sont générés directement par le solveur et non par Flopy lui-même. Seules des variantes faites maison permettent de contourner la création de ces fichiers. Deux solutions sont donc possibles :

1. La première serait de simplement ajouter ces fichiers dans le `.gitignore` pour éviter leur synchronisation.
2. Sinon, créer un script qui supprime tous les fichiers se terminant par `*.chk`, sous la forme d'une fonction `def` dans la `toolbox`, appelée à la fin des `post-traitements` de **Modflow** et **Modpath**.

```
clean_root = [dirname(root_dir), self.watershed_folder]
for clean_root in clean_root:
    for dirpath, dirnames, filenames in os.walk(clean_root):
        print(dirpath, filenames, dirnames)
        for filename in filenames:
            if filename.endswith('.chk'):
                os.remove(os.path.join(dirpath, filename))
                print(f"Delete {filename} file")
```