



Université  
de Rennes



**Fondation**  
Université de Rennes

# 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, 5 mai 2025

# Table des matières

---

<b>Table des matières</b>	<b>1</b>
<b>Table des figures</b>	<b>3</b>
<b>Liste des tableaux</b>	<b>4</b>
<b>1 Introduction</b>	<b>5</b>
1.1 Objectifs du document . . . . .	6
1.2 Contexte du projet . . . . .	6
1.3 Guide d'utilisation . . . . .	6
<b>2 Bibliographie</b>	<b>7</b>
2.1 Modflow . . . . .	8
2.1.1 Modflow NWT . . . . .	8
2.1.2 Package DRN (drain) . . . . .	8
2.1.3 Package SFR (streamflow-routing) . . . . .	8
2.2 Fuite du Lac (Leakage) . . . . .	8
<b>3 Données</b>	<b>9</b>
3.1 DEM . . . . .	10
3.1.1 BD-ALTI-75m . . . . .	10
3.2 Climatiques (passé) . . . . .	10
3.3 Projections climatiques . . . . .	10
3.4 Hydrologie . . . . .	10
3.4.1 Stations de jaugeage . . . . .	10
3.4.2 Hydrographie . . . . .	10

3.4.3	Intermittence . . . . .	10
3.5	Géologie . . . . .	10
3.6	Données EBR . . . . .	10
3.6.1	Abaque   Bathymétrie . . . . .	10
3.6.2	Données journalières . . . . .	10
3.6.3	Scénarios de gestion . . . . .	10
<b>4</b>	<b>Code - EBR</b>	<b>11</b>
4.1	App EBR commun.py . . . . .	12
4.1.1	Chargements des bibliothèques, modules et du dossier racines . . . . .	12
4.1.2	LogManager . . . . .	12
4.2	Initialisation de la classe climatiques . . . . .	12
4.2.1	Réanalyse Surfex . . . . .	12
4.2.2	Méthode de création d'un csv pour données climatiques . . . . .	12
4.3	Paramétrisation . . . . .	13
4.3.1	Simplex de Nelder-Mead . . . . .	13
<b>5</b>	<b>HydroModPy</b>	<b>19</b>
5.1	watershed_root.py . . . . .	20
5.2	toolbox.py . . . . .	20
5.2.1	class LogManager . . . . .	20
5.2.2	Paramétrisation par xml . . . . .	22
5.2.3	class Parameter . . . . .	22
5.2.4	class XmlConfigurator . . . . .	22
<b>6</b>	<b>Patch</b>	<b>23</b>
6.1	DeprecationWarnings . . . . .	24
6.2	Suppression des fichiers.chk . . . . .	24
<b>7</b>	<b>Python Tips</b>	<b>25</b>
7.1	Decorators . . . . .	26
7.1.1	@StaticMethod . . . . .	26
	<b>Références bibliographiques</b>	<b>27</b>

# Table des figures

---

# Liste des tableaux

---

---

# Chapitre 1

## Introduction

---

---

1.1	Objectifs du document . . . . .	6
1.2	Contexte du projet . . . . .	6
1.3	Guide d'utilisation . . . . .	6

---

## **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 . . . . .	8
2.1.1	Modflow NWT . . . . .	8
2.1.2	Package DRN (drain) . . . . .	8
2.1.3	Package SFR (streamflow-routing) . . . . .	8
2.2	Fuite du Lac (Leakage) . . . . .	8

---



## 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 . . . . .	10
3.1.1	BD-ALTI-75m . . . . .	10
3.2	Climatiques (passé) . . . . .	10
3.3	Projections climatiques . . . . .	10
3.4	Hydrologie . . . . .	10
3.4.1	Stations de jaugeage . . . . .	10
3.4.2	Hydrographie . . . . .	10
3.4.3	Intermittence . . . . .	10
3.5	Géologie . . . . .	10
3.6	Données EBR . . . . .	10
3.6.1	Abaque   Bathymétrie . . . . .	10
3.6.2	Données journalières . . . . .	10
3.6.3	Scénarios de gestion . . . . .	10

---

### **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 . . . . .	12
4.1.1	Chargements des bibliothèques, modules et du dossier racines . . . . .	12
4.1.2	LogManager . . . . .	12
4.2	Initialisation de la classe climatiques . . . . .	12
4.2.1	Réanalyse Surfex . . . . .	12
4.2.2	Méthode de création d'un csv pour données climatiques . . . . .	12
4.3	Paramétrisation . . . . .	13
4.3.1	Simplex de Nelder-Mead . . . . .	13

---

## 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 :

```

1  #=====
2  # Exportation des données climatiques
3  # =====
4  # df_climatic = pd.DataFrame({
5  #     'recharge': BV.climatic.recharge,
6  #     'runoff': BV.climatic.runoff,
7  #     'precip': BV.climatic.precip,
8  #     'evt': BV.climatic.evt,
9  #     'etp': BV.climatic.etp,
10 #     't': BV.climatic.t,
11 # })
12 # df_climatic.to_csv(os.path.join(data_path, 'Meteo', 'Historiques SIM2', 'climatic_data.csv'))

```

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

```

1 df_climatic = pd.read_csv(
2     os.path.join(data_path, 'Meteo', 'Historiques SIM2', 'climatic_data.csv'),
3     index_col=0, parse_dates=True
4 )
5 df_climatic.index = pd.to_datetime(df_climatic.index)
6 df_climatic = df_climatic.loc[
7     (df_climatic.index >= pd.Timestamp("01/01/{}".format(first_year))) &
8     (df_climatic.index <= pd.Timestamp("31/12/{}".format(last_year)))
9 ]
10
11 agg_dict = {
12     'recharge': 'sum',
13     'runoff': 'sum',
14     'precip': 'sum',
15     'evt': 'sum',
16     'etp': 'sum',
17     't': 'mean'
18 }
19 df_climatic = df_climatic.resample(freq_input).agg(agg_dict)
20
21 BV.climatic.recharge = df_climatic['recharge']
22 BV.climatic.runoff = df_climatic['runoff']
23 BV.climatic.precip = df_climatic['precip']
24 BV.climatic.evt = df_climatic['evt']
25 BV.climatic.etp = df_climatic['etp']
26 BV.climatic.t = df_climatic['t']
27
28 first_clim = BV.climatic.recharge[0]
29 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 (LAGARIAS et al., 1998 ; NELDER et MEAD, 1965).

#### 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 :

$$f(x_1) \leq f(x_2) \leq \dots \leq f(x_{N+1}) \quad (4.1)$$

Où  $x_1$  est le meilleur sommet (valeur de fonction la plus basse) et  $x_{N+1}$  le pire sommet (valeur de fonction la plus élevée). Le centroïde des  $N$  meilleurs sommets est calculé comme  $x_0 = \frac{1}{N} \sum_{i=1}^N x_i$ .

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

1. **Réflexion** (toujours effectuée en premier) :

Cette étape vise à explorer l'espace des solutions en reflétant le pire point du simplex (le sommet avec la plus mauvaise valeur de la fonction objectif) à travers le centre de gravité des autres points. Cela permet souvent de s'approcher d'une zone de meilleure performance.

- Calculer le point réfléchi :

$$x_r = x_0 + \alpha(x_0 - x_{N+1}) \quad (4.2)$$

où  $x_0$  est le centroïde des meilleurs sommets,  $x_{N+1}$  est le pire point, et  $\alpha$  est le coefficient de réflexion (généralement égal à 1). Ce coefficient détermine la distance à laquelle le point est réfléchi au-delà du centroïde, le nouveau point  $x_r$  est donc situé à une distance proportionnelle à la distance entre le centroïde et le pire point.

On évalue ensuite la fonction objectif au point réfléchi :

$$f_r = f(x_r) \quad (4.3)$$

Cette valeur déterminera l'étape suivante de l'algorithme.

2. **Décision** (une seule branche est suivie en fonction de la qualité du point réfléchi) :

Selon la valeur de  $f_r$ , plusieurs scénarios sont envisagés pour ajuster le simplex et continuer l'optimisation. Ce processus de décision permet à l'algorithme de s'adapter à la topologie locale de la fonction objectif.

- **Cas 1 : Acceptation simple**

Si le point réfléchi est meilleur qu'une majorité des points mais pas le meilleur :

$$f(x_1) \leq f_r < f(x_N) \quad (4.4)$$

alors on remplace simplement le pire point par le point réfléchi :

$$x_{N+1} \leftarrow x_r \quad (4.5)$$

Cette situation correspond à un progrès modéré dans la recherche de l'optimum, sans nécessiter d'exploration supplémentaire dans cette direction.



Cependant, lorsque ce même point deviendra le moins bon, l'exploration pourra reprendre dans cette direction.

- **Cas 2 : Expansion**

Si le point réfléchi est meilleur que le meilleur point actuel :

$$f_r < f(x_1) \quad (4.6)$$

on tente d'exploiter cette direction prometteuse en calculant un point encore plus éloigné. L'expansion permet de progresser plus rapidement vers l'optimum lorsqu'une direction favorable est identifiée :

$$x_e = x_0 + \beta(x_r - x_0) \quad (4.7)$$

où  $\beta$  est le coefficient d'expansion (supérieur à 1, égal à 2 dans le cas de l'utilisation de la méthode `minimize` dans `SciPy`). Ce coefficient détermine jusqu'où on étend la recherche dans la direction prometteuse.

- **Cas 3 : Contraction externe**

Si le point réfléchi est moins bon que la plupart des points mais meilleur que le pire :

$$f(x_N) \leq f_r < f(x_{N+1}) \quad (4.8)$$

on essaie un compromis plus modéré en effectuant une contraction externe. Cette opération permet d'explorer l'espace entre le centroïde et le point réfléchi :

$$x_c = x_0 + \gamma(x_r - x_0) \quad (4.9)$$

où  $\gamma$  est le coefficient de contraction. Ce coefficient restreint l'exploration à une zone plus proche du centroïde.

On évalue ensuite :

$$f_c = f(x_c) \quad (4.10)$$

- Si  $f_c \leq f_r$  : on accepte  $x_c$  (la contraction a trouvé un point meilleur) - Sinon : on procède à un rétrécissement du simplex (la contraction n'a pas été efficace)

- **Cas 4 : Contraction interne**

Si le point réfléchi est encore pire que le pire actuel :

$$f_r \geq f(x_{N+1}) \quad (4.11)$$

on tente une contraction plus prudente en explorant l'espace entre le centroïde et le pire point. Cette stratégie est adoptée lorsque la direction de réflexion s'avère défavorable :

$$x_c = x_0 + \gamma(x_{N+1} - x_0) \quad (4.12)$$

où  $\gamma$  est à nouveau le coefficient de contraction.

On évalue :

$$f_c = f(x_c) \quad (4.13)$$

- Si  $f_c < f(x_{N+1})$  : on remplace le pire point par  $x_c$  (la contraction interne a été bénéfique)  
 - Sinon : un rétrécissement complet du simplex devient nécessaire (la topologie locale est complexe et nécessite une restructuration)

### 3. Rétrécissement (seulement si la contraction a échoué) :

Cette étape drastique vise à « resserrer » le simplex autour du meilleur point trouvé pour éviter de rester bloqué dans des zones peu prometteuses. Le rétrécissement est une stratégie de dernier recours qui indique souvent que l'algorithme approche d'un minimum local ou rencontre une région difficile de la fonction objectif.

- Pour chaque  $i = 2, \dots, N + 1$  :

$$x_i \leftarrow x_1 + \delta(x_i - x_1) \quad (4.14)$$

où  $\delta$  est le coefficient de rétrécissement. Ce coefficient détermine à quel point le simplex se contracte autour du meilleur point.

Cette opération réduit la taille du simplex et recentre la recherche autour du meilleur point actuel, permettant une exploration plus fine et locale de l'espace des paramètres.



Où les coefficients standards sont  $\alpha = 1$  (réflexion),  $\beta = 2$  (expansion),  $\gamma = 0.5$  (contraction) et  $\delta = 0.5$  (rétrécissement). La méthode `minimize` de `Scipy` utilise une implémentation adaptative, ils dépendent de la dimension  $N$  du problème( basé sur les travaux de NELDER et MEAD, 1965 ; LAGARIAS et al., 1998 ; et GAO et HAN, 2012) :

$$\alpha = 1, \quad \beta = 1 + \frac{2}{N}, \quad \gamma = 0.75 - \frac{0.5}{N}, \quad \delta = 1 - \frac{1}{N} \quad (4.15)$$

Cette adaptation permet d'optimiser le comportement de l'algorithme en fonction de la dimensionnalité du problème. Pour les problèmes de grande dimension, les coefficients sont ajustés pour favoriser une exploration plus équilibrée de l'espace des paramètres.

#### **i Points importants à noter :**

- Une seule des branches de l'arbre de décision est suivie à chaque itération, ce qui rend l'algorithme efficace en termes de nombre d'évaluations de la fonction objectif. Cela signifie que l'algorithme choisit toujours le meilleur mouvement possible à chaque étape.
- Le rétrécissement n'est appliqué qu'en dernier recours, si les contractions échouent, car il s'agit d'une opération coûteuse nécessitant  $N$  évaluations supplémentaires de la fonction. Cette opération réduit la taille du simplex autour du meilleur point pour affiner la recherche localement.
- Pour les problèmes de grande dimension ( $>5$  paramètres), les opérations de réflexion deviennent dominantes mais moins efficaces. Les mouvements géométriques du simplex ne s'adaptent pas bien à la complexité croissante de l'espace de recherche, ce qui ralentit la convergence.
- L'algorithme ne nécessite pas le calcul de dérivées, ce qui le rend particulièrement utile pour l'optimisation de fonctions non différentiables, bruitées ou irrégulières. Cette caractéristique est idéale pour les modèles hydrogéologiques où les relations entre paramètres et performance peuvent présenter des discontinuités.

### **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 :

$$x_{norm} = \frac{x - x_{min}}{x_{max} - x_{min}} \quad (4.16)$$

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 :

$$K_{norm} = \frac{\log_{10}(K) - \log_{10}(K_{min})}{\log_{10}(K_{max}) - \log_{10}(K_{min})} \quad (4.17)$$

### **Implémentation dans le code :**

L'algorithme est intégré via la bibliothèque SciPy :

```

1  from scipy.optimize import minimize
2
3  # Normalisation des paramètres
4  def normalize(x, xmin, xmax):
5      """Normalise une valeur x selon les bornes xmin et xmax"""
6      return (x - xmin) / (xmax - xmin)
7
8  def denormalize(x_norm, xmin, xmax):
9      """Dénormalise une valeur x_norm selon les bornes xmin et xmax"""
10     return x_norm * (xmax - xmin) + xmin
11
12 # Fonction objectif normalisée
13 def erreur_modele_norm(params_norm):
14     # Dénormalisation des paramètres
15     log_hk_value = denormalize(params_norm[0], log_hk_min, log_hk_max)
16     hk_value = 10**log_hk_value
17     sy_value = denormalize(params_norm[1], sy_min, sy_max)
18     thick_value = denormalize(params_norm[2], thick_min, thick_max)
19
20     # Mise à jour du modèle avec les nouveaux paramètres
21     BV.hydraulic.update_hk(hk_value)
22     BV.hydraulic.update_sy(sy_value)
23     BV.hydraulic.update_thick(thick_value)
24
25     # Simulation du modèle avec ces paramètres
26     model_modflow = BV.preprocessing_modflow()
27     success_modflow = BV.processing_modflow(model_modflow)
28     BV.postprocessing_timeseries(model_modflow)
29
30     # Calcul du critère de Nash-Sutcliffe
31     nse = 1 - (numerator / denominator)
32     return 1 - nse # On minimise 1-NSE
33
34 # Exécution de l'optimisation
35 result = minimize(
36     erreur_modele_norm,
37     x0_norm, # Paramètres initiaux normalisés
38     method='Nelder-Mead',
39     options={
40         'xatol': 0.01, # Tolérance sur les paramètres
41         'fatol': 0.01, # Tolérance sur la fonction
42         'maxiter': 200, # Nombre max d'itérations
43         'disp': True # Affichage des informations
44     }
45 )

```

## Paramètres calibrés

Dans notre implémentation, trois paramètres hydrogéologiques fondamentaux sont calibrés, la conductivité hydraulique ( $K$ ), la porosité efficace ( $S_y$ ) et l'épaisseur de la couche aquifère ( $e$ ). Ces paramètres sont cruciaux pour simuler le comportement hydraulique du modèle.

## Fonction objectif et sélection des données

La fonction objectif utilisée dans notre cas est le critère de Nash-Sutcliffe - NSE (NASH et SUTCLIFFE, 1970), qui évalue la qualité de la simulation par rapport aux données observées. Le NSE est défini comme suit :

$$\text{NSE} = 1 - \frac{\sum_{t=1}^T (Q_{obs,t} - Q_{sim,t})^2}{\sum_{t=1}^T (Q_{obs,t} - \bar{Q}_{obs})^2} \quad (4.18)$$

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

```

1 def filter_dates(dates):
2     """Filtre les dates selon des critères temporels et saisonniers"""
3     mask = pd.Series(True, index=dates)
4
5     if use_time_filter:
6         mask = mask & (dates >= calib_start_date) & (dates <= calib_end_date)
7
8     if use_seasonal_filter:
9         def is_in_season(date):
10             start = pd.Timestamp(date.year, season_start_month, season_start_day)
11             if season_end_month < season_start_month:
12                 end = pd.Timestamp(date.year + 1, season_end_month, season_end_day)
13             else:
14                 end = pd.Timestamp(date.year, season_end_month, season_end_day)
15             return (date >= start) & (date <= end)
16
17         seasonal_mask = dates.map(is_in_season)
18         mask = mask & seasonal_mask
19
20     return mask

```

Cette approche permet de concentrer la calibration sur des périodes représentatives, en excluant si nécessaire des événements extrêmes ou des saisons particulières, pour optimiser les périodes d'étiage par exemple.

## 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.

Cependant, il est important de garder à l'esprit ses limitations, notamment la sensibilité à l'initialisation et la possibilité de convergence vers des minima locaux.

Pour éviter cela, **il est recommandé de tester plusieurs initialisations** afin de s'assurer que les différents résultats convergent vers des solutions similaires.

---

# Chapitre 5

## HydroModPy

---

---

5.1	watershed_root.py . . . . .	20
5.2	toolbox.py . . . . .	20
5.2.1	class LogManager . . . . .	20
5.2.2	Paramétrisation par xml . . . . .	22
5.2.3	class Parameter . . . . .	22
5.2.4	class XmlConfigurator . . . . .	22

---

## 5.1 watershed\_root.py

## 5.2 toolbox.py

### 5.2.1 class LogManager

Le `LogManager` est conçue pour configurer et gérer la journalisation 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 :

```
1 # Initialisation du LogManager en mode developpement
2 log_manager = toolbox.LogManager(
3     mode="dev", # Utilisez mode="verbose" pour afficher les logs INFO et superieurs, et mode="quiet"
4     # pour afficher les logs WARNING et superieurs
5     log_dir=root_dir, # Specifiez le repertoire de journalisation
6     overwrite=False, # Utilisez overwrite=True (par default) pour ecraser les fichiers de log
7     # existants
8     verbose_libraries=True # Utilisez verbose_libraries=True pour afficher les logs des bibliotheques
9     # (avertissements et superieurs, generalement masques)
10 )
```

**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 certaines bibliothèques externes pour éviter un terminal (kernel) surchargé. Voici la liste des bibliothèques dont les logs sont réduits au niveau CRITICAL :

```
1 libraries_to_silence = [
2     "fiona",
3     "rasterio",
4     "urllib3",
```

```
5 "geopy",
6 "matplotlib",
7 "PIL"
8 ]
```

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.

### Exceptions :

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 Spécificateurs 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.

### 5.2.2 Paramétrisation par xml

### 5.2.3 class Parameter

### 5.2.4 class XmlConfigurator

---

# Chapitre 6

## Patch

---

---


6.1	DeprecationWarnings . . . . .	24
6.2	Suppression des fichiers.chk . . . . .	24

---



## 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.

```
1 # Filtrer les avertissements (avant les imports)
2 import warnings
3 warnings.filterwarnings('ignore', category=DeprecationWarning)
4
5 import pkg_resources # A placer apres DeprecationWarning car elle meme obsolète...
6 warnings.filterwarnings('ignore', message='.*pkg_resources.*')
7 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**.

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

---

# Chapitre 7

## Python Tips

---

---

7.1	Decorators . . . . .	26
7.1.1	@StaticMethod . . . . .	26

---

## 7.1 Decorators

### 7.1.1 @staticmethod

`@staticmethod` est un décorateur qui transforme une méthode de classe en une méthode statique. Cela signifie que la méthode peut être appelée sur la classe elle-même, sans avoir besoin d'une instance de la classe. Les méthodes statiques n'ont pas accès à l'instance (`self`) ou à la classe (`cls`) et ne peuvent pas modifier l'état de l'objet ou de la classe.



Elle fonctionne comme une fonction normale, mais organisée dans l'espace de noms de la classe.

Exemple :

```
1 class MathUtils:
2     @staticmethod
3     def add(x, y):
4         return x + y
```



- Ne pas tenter d'accéder à `self` dans une méthode statique.
- Ne pas utiliser `@staticmethod` quand la fonction a besoin d'accéder aux données de l'objet.
- Ne pas confondre avec des fonctions globales - les méthodes statiques ont un lien conceptuel avec leur classe.

# Références bibliographiques

---

- GAO, F. et HAN, L. (jan. 2012). « Implementing the Nelder-Mead Simplex Algorithm with Adaptive Parameters ». Dans : *Computational Optimization and Applications* 51.1, 259-277. DOI : [10.1007/s10589-010-9329-3](https://doi.org/10.1007/s10589-010-9329-3). URL : <http://link.springer.com/10.1007/s10589-010-9329-3> (visité le 27/03/2025).
- LAGARIAS, J. C., REEDS, J. A., WRIGHT, M. H. et WRIGHT, P. E. (jan. 1998). « Convergence Properties of the Nelder-Mead Simplex Method in Low Dimensions ». Dans : *SIAM Journal on Optimization* 9.1, 112-147. DOI : [10.1137/S1052623496303470](https://doi.org/10.1137/S1052623496303470). URL : <http://epubs.siam.org/doi/10.1137/S1052623496303470> (visité le 02/05/2025).
- NASH, J. et SUTCLIFFE, J. (avr. 1970). « River Flow Forecasting through Conceptual Models Part I — A Discussion of Principles ». Dans : *Journal of Hydrology* 10.3, 282-290. DOI : [10.1016/0022-1694\(70\)90255-6](https://doi.org/10.1016/0022-1694(70)90255-6). URL : <https://linkinghub.elsevier.com/retrieve/pii/S0022169470902556> (visité le 02/05/2025).
- NELDER, J. A. et MEAD, R. (1<sup>er</sup> jan. 1965). « A Simplex Method for Function Minimization ». Dans : *The Computer Journal* 7.4, 308-313. DOI : [10.1093/comjnl/7.4.308](https://doi.org/10.1093/comjnl/7.4.308). URL : <https://academic.oup.com/comjnl/article-lookup/doi/10.1093/comjnl/7.4.308> (visité le 02/05/2025).