Application Programming Interface - 2022

Projet MOLONARI



# Table des matières

[**Table des matières**](#_escooim0l600) **2**

[**Introduction**](#_75r1z3w6ilri) **3**

[**IHM vers Calcul**](#_sefl8pjccw23) **4**

[**Classe Prior**](#_9q21r1946pa8) **4**

[**Classe Layer**](#_xu46ugr5dlk9) **4**

[**Fonction layersListCreator**](#_7y3huvb5dwzx) **5**

[**Classe Column**](#_p0sw5ikco3ks) **5**

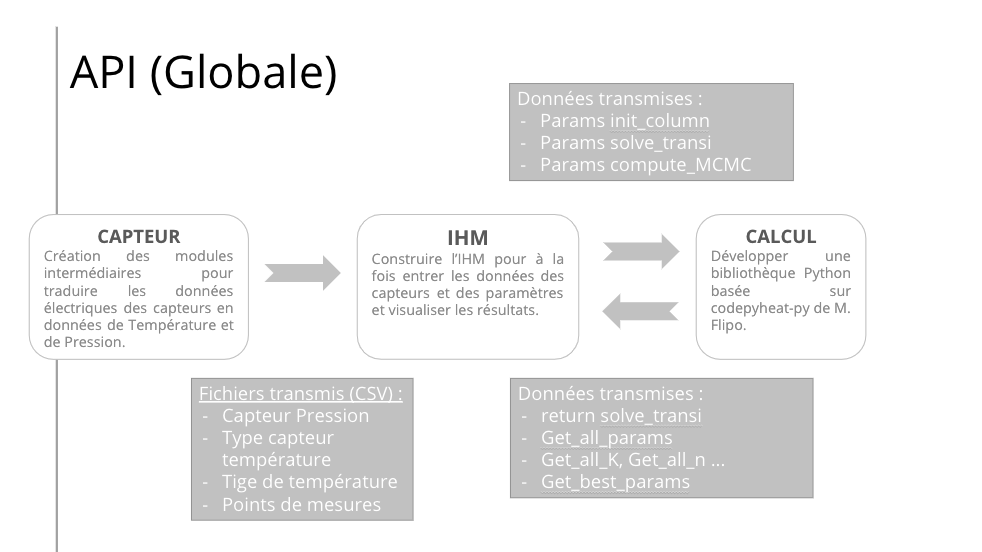
[**Calcul vers IHM**](#_qe537bpovk92) **9**

[**Méthodes disponibles après un appel du modèle direct**](#_120h7l1irp8o) **9**

[**Méthodes disponibles après un appel de la MCMC**](#_j8347gumpipi) **12**

# Introduction

Cette bibliothèque spécifie les interfaces d’échanges entre les trois pôles capteurs, calculs et IHM afin de mener le projet à bien.



# 

# **IHM vers Calcul**

Cette partie décrit les informations que IHM doit envoyer à Calcul pour définir les objets et utiliser les méthodes de calcul principales.

Les seuls objets que IHM a besoin de définir sont :

* Les instances de la classe **Column**. C’est la classe principale. Elle regroupe les mesures de charge et de températures réalisées sur un point de mesure, puis elle permet d’exécuter le modèle direct et d’exécuter une MCMC. À l’issue de ces calculs, elle contient les informations utiles, que l’on peut récupérer avec des méthodes décrites dans la partie suivante.
* Les instances de la classe **Layer**. Cette classe sert à définir une strate du milieu. Elle est utilisée par le modèle stratifié.
* Les instances de la classe **Prior**. Cette classe sert à définir, dans le cadre de la MCMC, les informations utiles pour un paramètre : son intervalle de variations (range), l’écart type de la marche aléatoire pour ce paramètre (sigma), et, dans le cas où sa loi a priori n’est pas uniforme, sa densité (density).

## **Classe Prior**

* **Définition d’un objet Prior :**

Paramètres :

* range : tuple, intervalle dans lequel le paramètre peut varier
* sigma : float, écart type de la gaussienne utilisée pour la marche aléatoire pour ce paramètre
* density : callable, densité de la loi a priori. Par défaut, c’est une fonction constante, correspondant à une loi uniforme.

**Remarque :** Le seul moment où l’utilisateur a potentiellement besoin de définir lui-même un Prior est pour la MCMC avec estimation du paramètre sigma2 (voir exemple ci-après).

## **Classe Layer**

* **Définition d’un objet Layer :**

Paramètres :

* name : string
* zLow : float, profondeur en m du bas de la couche (attention, repère orienté du haut vers le bas, donc il s’agit d’un nombre positif qui grandit à mesure que l’on s’enfonce dans le sol)
* moinslog10K : float, valeur de -log10(K) où K est la perméabilité
* n : float, porosité
* lambda\_s : float
* rhos\_cs : float

Exemple :

|  |
| --- |
| my\_layer = Layer(“Couche 1”, 0.2, 4.0, 0.1, 2.0, 4e6) |

Remarque : le groupe IHM créera des objets de la classe Layer uniquement via la fonction layersListCreator décrite ci-après.

## **Fonction layersListCreator**

Pour définir directement une liste d’objets de la classe Layer, on dispose de la fonction layersListCreator. Attention, ce n’est pas une méthode de la classe Layer.

* **Définition d’une liste d’objets de Layer avec la fonction layersListCreator**

Paramètres :

* layersListInput : list, liste de tuples où chaque tuple contient les six arguments nécessaires à la définition d’un objet Layer

Retourne :

* une liste d’objets de la classe Layer

Exemple :

|  |
| --- |
| my\_layer\_list = layersListCreator(  [(“Couche 1”, 0.1, 4, 0.1, 2, 4e6),  (“Couche 2”, 0.25, 4.2, 0.1, 3, 4e7)]  ) |

Remarque : c’est cette liste d’objets de la classe Layer qui est utilisée par la suite dans le code.

## **Classe Column**

* **Définition d’un objet Column avec la méthode de classe from\_dict :**

On peut définir un objet de la classe Column à partir d’un dictionnaire à l’aide de la méthode de classe from\_dict :

Paramètres : un dictionnaire de la forme (format clé : valeurs) :

* river\_bed : float, altitude du lit de la rivière, en m (inutilisé actuellement)
* depth\_sensors : Sequence[float], liste des profondeurs des capteurs de la tige (au nombre de 4), en m
* offset : float, enfoncement excédentaire de la tige, en m (si la tige est enfoncée 10 cm trop profondément, offset = 0.1)
* dH\_measures : list, liste contenant des tuples avec la date, la charge et la température au sommet de la colonne **(voir exemple)**
* T\_measures : list, liste contenant des tuples avec la date, les températures aux points de mesure sur la tige **(voir exemple)**
* sigma\_meas\_P : float, erreur de mesure sur la pression (inutilisé actuellement)
* sigma\_meas\_T : float, erreur de mesure sur la température (inutilisé actuellement)
* inter\_mode : str, définit le type d’interpolation utilisé pour calculer les températures initiales le long de la colonne. Deux modes possibles : “linear” (par défaut), et “lagrange”

Retourne : objet de type Column

Exemple :

|  |
| --- |
| dH\_measures = [  (Timestamp('2016-06-28 06:45:00'), (0.84556517, 287.165)),  (Timestamp('2016-06-28 07:00:00'), (0.84559872, 287.159)),  …  ]  T\_measures = [  (Timestamp('2016-06-28 06:45:00'), array([287.272, 287.344, 287.272, 287.08 ])),  (Timestamp('2016-06-28 07:00:00'), array([287.248, 287.32 , 287.272, 287.08 ]))  ]  col\_dict = {  “river\_bed”: 1.,  “depth\_sensors”: [.1, .2, .3, .4],  “offset”: .0,  “dH\_measures”: dH\_measures,  “T\_measures”: T\_measures,  “sigma\_meas\_P”: None, #float  “sigma\_meas\_T”: None, #float  “inter\_mode” : “linear”  }  col = Column.from\_dict(col\_dict) |

* **Appel du modèle direct via la méthode compute\_solve\_transi :**

Cette méthode exécute le modèle direct, en version stratifiée (si on donne une liste d’objets de type Layer en argument) ou en homogène (si on donne seulement un tuple contenant les 4 paramètres). Cette méthode met à jour les attributs de l’objet de type Column, elle ne renvoie rien. D’autres méthodes “getteurs” permettront de récupérer ces résultats.

Paramètres :

* layersList : liste d’objets de la classe Layer (créée par la fonction layersListCreator par exemple) *ou* tuple avec les 4 paramètres
* nb\_cells : int : nombre de cellules de la discrétisation
* verbose : booléen

Retourne :

* rien (calcul du modèle direct avec les paramètres associés aux layers de layersList)

Exemple :

|  |
| --- |
| # Appel en homogène avec un tuple de paramètres  col.compute\_solve\_transi((4, 0.1, 2, 4e6), 100)  # Appel en stratifié avec une liste d’objets Layer  col.compute\_solve\_transi(my\_layer\_list, 100) |

* **Calcul de l’inversion bayésienne via la méthode compute\_mcmc :**

Cette méthode réalise la MCMC. Elle ne renvoie rien, mais elle stocke la liste des états qu’elle a créés. Des “getteurs” permettront de récupérer les résultats.

Paramètres :

* nb\_iter : int, nombre d’itérations
* all\_priors : list, dont chaque élément est une liste de la forme [Nom de couche, zLow de la couche, et un dictionnaire décrivant les priors des paramètres de la couche]
* nb\_cells : int,
* quantile : tuple,
* sigma2 : float, None par défaut. Si sigma2 vaut None, la MCMC est lancée en estimant sigma2, dont la distribution a priori est donnée par sigma2\_temp\_prior. Sinon, la MCMC est lancée avec sigma2 fixé.
* sigma2\_temp\_prior : Prior, distribution a priori de sigma2

Retourne :

* rien (calcul de l’inversion bayésienne)

Exemple : pour une colonne à deux strates, de profondeur 0.4m

|  |
| --- |
| priors\_couche\_1 = {  “moinslog10K”: ((1.5, 6), .01),  “n”: ((.01, .25), .01),  “lambda\_s”: ((1, 5), .1),  “rhos\_cs”: ((1e6,1e7), 1e5),  }  priors\_couche\_2 = {  “moinslog10K”: ((1.5, 6), .01),  “n”: ((.01, .25), .01),  “lambda\_s”: ((1, 5), .1),  “rhos\_cs”: ((1e6,1e7), 1e5),  }  all\_priors = [  [“Couche 1”, 0.2, priors\_couche\_1],  [“Couche 2”, 0.4, priors\_couche\_2]  ]  # MCMC sans estimation de sigma2  col.compute\_mcmc(  nb\_iter = 5000,  all\_priors = all\_priors,  nb\_cells = 100,  quantile = (0.05, 0.5, 0.95),  sigma2 = 1.0  )  # MCMC avec estimation de sigma2  col.compute\_mcmc(  nb\_iter = 5000,  all\_priors = all\_priors,  nb\_cells = 100,  quantile = (0.05, 0.5, 0.95),  sigma2\_temp\_prior = Prior((0.1, np.inf), 0.1, lambda x : 1/x)  ) |

# 

# **Calcul vers IHM**

Cette partie décrit les méthodes que IHM peut utiliser sur un objet de type Column pour récupérer des informations issues des méthodes de calcul décrites précédemment.

## **Méthodes disponibles après un appel du modèle direct**

Ces méthodes permettent de récupérer les informations calculées par le **dernier appel du modèle direct,** que cet appel ait été lancé par l’utilisateur directement (via compute\_solve\_transi) ou indirectement (via compute\_mcmc). Les fonctions détaillées dans cette partie sont des méthodes de la classe Column, et elles ne peuvent pas être utilisées avant d’avoir appelé le modèle direct une première fois.

* **Récupérer les indices des cellules contenant un capteur : get\_id\_sensors**

Paramètres :

* aucun

Retourne :

* liste des indices des cellules contenant des capteurs (hors capteurs rivière et aquifère).

Exemple :

|  |
| --- |
| >>> col.get\_id\_sensors()  [24, 49, 74] |

* **Récupérer les RMSE : get\_RMSE**

Paramètres :

* aucun

Retourne :

* array contenant les RMSE pour chaque capteur, puis la RMSE globale.

Exemple :

|  |
| --- |
| >>> col.get\_RMSE()  array([0.0763115 , 0.09875992, 0.04563868, 0.0767243]) |

* **Récupérer les profondeurs des milieux des cellules : get\_depths\_solve**

Paramètres :

* aucun

Retourne :

* array contenant les profondeurs des milieux des cellules.

Exemple :

|  |
| --- |
| >>> col.get\_depths\_solve()  array([0.002, 0.006, 0.01 , 0.014, …, 0.394, 0.398]) |

* **Récupérer les temps : get\_times\_solve**

Paramètres :

* aucun

Retourne :

* self.\_times : liste contenant les dates des mesures (et donc les dates où les températures sont calculées)

Exemple :

|  |
| --- |
| >>> col.get\_times\_solve()  [Timestamp('2016-06-28 06:45:00'),  Timestamp('2016-06-28 07:00:00'),  …,  Timestamp('2016-07-11 21:45:00')] |

* **Récupérer les températures : get\_temps\_solve**

Paramètres :

* z : float, profondeur à laquelle on souhaite récupérer les températures calculées. Argument optionnel (voir ci-après).

Retourne :

* si z est précisé : array unidimensionnel contenant les températures au cours du temps à la profondeur z. Si z n’est pas fourni, renvoie le tableau bidimensionnel des températures pour toutes les profondeurs et tous les temps. **Une ligne de ce tableau correspond à une profondeur fixée, une colonne est une date fixée.**

Exemple :

|  |
| --- |
| >>> col.get\_temps\_solve(0.12)  array([287.29245, 287.16483, 287.1529 , ..., 289.0312 , 288.9903 ,  288.94864], dtype=float32) |

* **Récupérer les flux advectifs : get\_advec\_flows\_solve**

Paramètres :

* aucun

Retourne :

* array bidimensionnel contenant les flux advectifs. **Une ligne de ce tableau correspond à une profondeur fixée, une colonne est une date fixée.**

Exemple :

|  |
| --- |
| >>> col.get\_advec\_flows\_solve()  array([[254048.15919553, 254022.16654971, 254048.32720796, ...,  261296.83775776, 261262.13761012, 261198.36194832],  [254049.67349804, 254021.29928274, 254047.85319433, ...,  261298.47274618, 261265.90944973, 261197.28451545],  ...,  [253990.50543789, 254190.74094686, 254150.33887346, ...,  261391.05108739, 261367.21750146, 261317.35701802],  [253975.52151344, 254124.1002343 , 254101.98530373, ...,  261054.38735068, 261032.7279476 , 261001.42568484]]) |

* **Récupérer les flux conductifs : get\_conduc\_flows\_solve**

Paramètres :

* aucun

Retourne :

* array bidimensionnel contenant les flux advectifs. **Une ligne de ce tableau correspond à une profondeur fixée, une colonne est une date fixée.**

Exemple :

|  |
| --- |
| >>> col.get\_conduc\_flows\_solve(0.12)  array([[ 1.1068530e+00, -3.5365844e-01, 1.4527908e-01, ...,  3.7912127e-01, 3.8149682e-01, 3.6746627e-01],  [ 1.3292813e+00, -3.5493898e-01, 1.1135341e-01, ...,  3.6885816e-01, 3.6885816e-01, 3.6885816e-01],  ...,  [-3.8695312e+00, -1.1552916e+00, 8.0731219e-01, ...,  -1.7398970e-01, -7.6555468e-02, 0.0000000e+00],  [-3.8561687e+00, -6.3866383e+01, -5.0049278e+01, ...,  -3.3558313e+02, -3.3403745e+02, -3.1533536e+02]]) |

* **Récupérer les débits : get\_flows\_solve**

Paramètres :

* z : float, profondeur à laquelle on souhaite récupérer les débits calculés. Argument optionnel (voir ci-après).

Retourne :

* si z est précisé : array unidimensionnel contenant les débits au cours du temps à la profondeur z. Si z n’est pas fourni, renvoie le tableau bidimensionnel des débits pour toutes les profondeurs et tous les temps. **Une ligne de ce tableau correspond à une profondeur fixée, une colonne est une date fixée.**

Exemple :

|  |
| --- |
| >>> col.get\_conduc\_flows\_solve(0.12)  array([0.00021139, 0.00021139, 0.0002114 , ..., 0.00021605, 0.00021605,  0.00021603], dtype=float32) |

## **Méthodes disponibles après un appel de la MCMC**

Ces méthodes permettent de récupérer les informations calculées par la MCMC. Les fonctions détaillées dans cette partie sont des méthodes de la classe Column, et elles ne peuvent pas être utilisées avant d’avoir appelé la MCMC une première fois.

* **Récupérer les profondeurs des milieux des cellules : get\_depths\_mcmc**

Paramètres :

* aucun

Retourne :

* array contenant les profondeurs des milieux des cellules utilisées dans la MCMC.

Exemple :

|  |
| --- |
| >>> col.get\_depths\_mcmc()  array([0.002, 0.006, 0.01 , 0.014, …, 0.394, 0.398]) |

* **Récupérer les temps : get\_times\_mcmc**

Paramètres :

* aucun

Retourne :

* self.\_times : liste contenant les dates des mesures (et donc les dates où les températures sont calculées)

Exemple :

|  |
| --- |
| >>> col.get\_times\_mcmc()  [Timestamp('2016-06-28 06:45:00'),  Timestamp('2016-06-28 07:00:00'),  …,  Timestamp('2016-07-11 21:45:00')] |

* **Récupérer des paramètres aléatoirement : sample\_param**

Paramètres :

* aucun

Retourne :

* liste dont le n-ième élément est un named tuple contenant les 4 paramètres de la n-ième couche, pour un état de la MCMC choisi aléatoirement.

Exemple :

|  |
| --- |
| >>> col.sample\_param()  [Parametres(moinslog10K=5.829515999088784, n=0.13098899589086638, lambda\_s=4.575026012425696, rhos\_cs=1986067.5639966507),  Parametres(moinslog10K=5.977343374859448, n=0.02849317033859023, lambda\_s=3.6471541965577563, rhos\_cs=9082662.883385323),  Parametres(moinslog10K=5.9652309551915135, n=0.02789212956577023, lambda\_s=4.783013898493102, rhos\_cs=3841935.090947748)] |

* **Récupérer le meilleur sigma2 : get\_best\_sigma2**

Paramètres :

* aucun

Retourne :

* float correspondant au sigma2 du “meilleur” état de la MCMC : celui d’énergie minimale, le maximum a posteriori (MAP).

Exemple :

|  |
| --- |
| >>> col.get\_best\_sigma2()  1.0 |

* **Récupérer les meilleurs layers : get\_best\_layers**

Paramètres :

* aucun

Retourne :

* liste de layers, correspondant aux layers du “meilleur” état de la MCMC : celui d’énergie minimale, le maximum a posteriori (MAP).

Exemple :

|  |
| --- |
| >>> col.get\_best\_layers()  [Couche 1 : ends at 0.15 m. Parametres(moinslog10K=5.978658479245687, n=0.09435452260035626, lambda\_s=4.030363094208517, rhos\_cs=4535824.342980439),  Couche 2 : ends at 0.25 m. Parametres(moinslog10K=5.987688039256958, n=0.05392840931238012, lambda\_s=4.842472761269677, rhos\_cs=2980844.628387417),  Couche 3 : ends at 0.4 m. Parametres(moinslog10K=5.974161875812504, n=0.02942429514593753, lambda\_s=3.3324680740689945, rhos\_cs=9492712.639617197)] |

* **Récupérer tous les paramètres : get\_all\_params**

Paramètres :

* aucun

Retourne :

* liste dont le n-ième élément est une matrice correspondant au n-ième layer. Cette matrice a 4 colonnes, une pour chaque paramètre, qui contiennent donc les distributions de ce paramètre.

Exemple :

|  |
| --- |
| >>> col.get\_all\_params()[0]  [[8.56239173e+00 1.93875163e-01 5.37909268e+00 8.27597016e+06],  [8.46043895e+00 1.94392128e-01 5.42383332e+00 8.27329793e+06],  …,  [8.46043895e+00 1.94392128e-01 5.42383332e+00 8.27329793e+06],  [8.56239173e+00 1.93875163e-01 5.37909268e+00 8.27597016e+06]] |

* **Récupérer tous les moinslog10K : get\_all\_moinslog10K**

Paramètres :

* aucun

Retourne :

* liste dont le n-ième élément est une liste contenant les moinslog10K de chaque layer du n-ième état de la MCMC.

Exemple :

|  |
| --- |
| >>> col.get\_all\_moinslog10K()  [[5.975034735454937, 5.964062746233368, 5.933199009193335],  [5.975034735454937, 5.964062746233368, 5.933199009193335],  …,  [5.977080913309422, 5.974830357957307, 5.924773341826041]] |

* **Récupérer tous les n : get\_all\_n**

Paramètres :

* aucun

Retourne :

* liste dont le n-ième élément est une liste contenant les n de chaque layer du n-ième état de la MCMC.

Exemple :

|  |
| --- |
| >>> col.get\_all\_n()  [[0.09770101340142129, 0.05446545145441365, 0.03449298581974063],  [0.09770101340142129, 0.05446545145441365, 0.03449298581974063],  …,  [0.0980698732891378, 0.04571794145513558, 0.023430662346356088]] |

* **Récupérer tous les lambda\_s : get\_all\_lambda\_s**

Paramètres :

* aucun

Retourne :

* liste dont le n-ième élément est une liste contenant les lambda\_s de chaque layer du n-ième état de la MCMC.

Exemple :

|  |
| --- |
| >>> col.get\_all\_lambda\_s()  [[1.7652159243433618, 4.896081607512383, 3.3698558559669176],  [1.7652159243433618, 4.896081607512383, 3.3698558559669176],  …,  [1.8533628159142332, 4.931031104270697, 3.2884156594266707]] |

* **Récupérer tous les rhos\_cs : get\_all\_rhos\_cs**

Paramètres :

* aucun

Retourne :

* liste dont le n-ième élément est une liste contenant les rhos\_cs de chaque layer du n-ième état de la MCMC.

Exemple :

|  |
| --- |
| >>> col.get\_all\_rhos\_cs()  [[4163009.985443604, 2838640.4878082406, 1213370.888916834],  [4163009.985443604, 2838640.4878082406, 1213370.888916834],  …,  [4041559.6303990236, 2870287.58466214, 1242895.5038242743]] |

* **Récupérer tous les sigma2 : get\_all\_sigma2**

Paramètres :

* aucun

Retourne :

* liste dont le n-ième élément est le sigma2 du n-ième état de la MCMC.

Exemple :

|  |
| --- |
| >>> col.get\_all\_sigma2()  [1.0,  …,  1.0] |

* **Récupérer toutes les énergies : get\_all\_energy**

Paramètres :

* aucun

Retourne :

* liste dont le n-ième élément est l’énergie du n-ième état de la MCMC.

Exemple :

|  |
| --- |
| >>> col.get\_all\_energy()  [584.5139625238698,  …,  522.2160627948176] |

* **Récupérer tous les acceptance ratio : get\_all\_acceptance\_ratio**

Paramètres :

* aucun

Retourne :

* array dont le n-ième élément est l’acceptance ratio à la n-ième itération de la MCMC (la proportion d’états acceptés à cet instant).

Exemple :

|  |
| --- |
| >>> col.get\_all\_acceptance\_ratio()  array([0. , 0.5 , 0.33333333, 0.25 , 0.2 ,  …,  0.33333333, 0.28571429, 0.25 , 0.22222222, 0.2 ] |

* **Récupérer les quantiles : get\_quantiles**

Paramètres :

* aucun

Retourne :

* objet contenant les quantiles que l’utilisateur a donné lors de la MCMC.

Exemple :

|  |
| --- |
| >>> quantiles = col.get\_quantiles()  >>> for q in quantiles:  … print(q)  0.25  0.5  0.95 |

* **Récupérer un quantile de température : get\_temps\_quantile**

Paramètres :

* quantile : float, quantile que l’on souhaite récupérer. **Ce quantile devait faire partie de l’argument quantiles de la MCMC.**

Retourne :

* array bidimensionnel des températures pour toutes les profondeurs et tous les temps. **Une ligne de ce tableau correspond à une profondeur fixée, une colonne est une date fixée.**

Exemple :

|  |
| --- |
| >>> col.get\_temps\_quantile(0.5)  array([[287.16714478, 287.16143799, 287.1600647 , ..., 289.01049805,  288.96957397, 288.92938232],  [287.17141724, 287.16622925, 287.16427612, ..., 289.01473999,  288.9760437 , 288.93743896],  ...,  [287.09152222, 287.09268188, 287.09283447, ..., 287.99795532,  287.98400879, 287.99719238],  [287.08383179, 287.08425903, 287.08432007, ..., 287.97644043,  287.95648193, 287.97613525]]) |

* **Récupérer un quantile de débit : get\_flows\_quantile**

Paramètres :

* quantile : float, quantile que l’on souhaite récupérer. **Ce quantile devait faire partie de l’argument quantiles de la MCMC.**

Retourne :

* array bidimensionnel des débits pour toutes les profondeurs et tous les temps. **Une ligne de ce tableau correspond à une profondeur fixée, une colonne est une date fixée.**

Exemple :

|  |
| --- |
| >>> col.get\_flows\_quantile(0.5)  array([[3.22657797e-06, 3.22676533e-06, 3.22708661e-06, ...,  3.29926229e-06, 3.29849968e-06, 3.29725276e-06],  [3.22656911e-06, 3.22674373e-06, 3.22708479e-06, ...,  3.29921249e-06, 3.29854151e-06, 3.29726799e-06],  ...,  [2.26208385e-06, 2.26209750e-06, 2.26214274e-06, ...,  2.31107947e-06, 2.31133708e-06, 2.31154718e-06],  [2.26208385e-06, 2.26209750e-06, 2.26214229e-06, ...,  2.31107947e-06, 2.31133731e-06, 2.31154672e-06]]) |

* **Récupérer la RMSE d’un quantile : get\_RMSE\_quantile**

Paramètres :

* quantile : float, quantile dont on souhaite calculer la RMSE. **Ce quantile devait faire partie de l’argument quantiles de la MCMC.**

Retourne :

* array contenant les RMSE pour chaque capteur, puis la RMSE globale.

Exemple :

|  |
| --- |
| >>> col.get\_RMSE\_quantile(0.5)  array([0.14360852, 0.16400421, 0.08305708, 0.13468391]) |

* **détail données transmis / Return solve\_transi :**

|  |  |
| --- | --- |
| **Return get\_...\_solve()** | **Exemple** |
| pas de retours lors de l’appel de la fonction compute\_solve\_transi qui prend comme argument params compute\_solve\_transi mais on peut accéder à la liste des profondeurs / temps et matrices des Températures / débits en faisant comme dans l’exemple à droite   * Liste des profondeurs * Liste des Temps * Matrice des températures en fonction de la profondeur (en ligne) et du temps (en colonnes) * Matrice des débits en  fonction de la profondeur et du temps * Matrice des flux de chaleurs Advectifs * Matrice des flux de chaleurs conductifs   →   format numpy.array   * Pas de noms | #il faut compute\_solve\_transi avant et les valeurs récupérées après sont les dernières compute  **depths = get\_depths\_solve()**  [0, 02, 04, …], listes\_indices\_capteurs (unité m)  **times = get\_times\_solve()**  […](unité s)  **temps= get\_temps\_solve()**  [[123, 123, 123, …], température pour profondeur 0   [123, 123, 123, …], …   [123, 123, 123, …], en °C   …  ]  **flows = get\_flows\_solve()**  [[d1, d2 ….] débit à la profondeur 0   [...]   … en m^3.s-1  ]  **advec = get\_advec\_flows\_solve()**  [[1, 3, 1, …], température pour profondeur 0   [ …], … en °C.s-1   …  ]  **conduc = get\_conduc\_flows\_solve()**  [[1, 3, 1, …], température pour profondeur 0   [ …], … en °C.s-1   …  ] |

* **détail données transmis / get\_quantile() :**

|  |  |
| --- | --- |
| **Return get\_quantile(...)** | **Exemple** |
| Méthode à suivre pour récupérer les quantiles voulu   * Liste des profondeurs * Liste des temps * Matrices des températures pour les quantiles demandés * liste de Matrices des débits pour les quantiles demandés * Paramètres pour les quantiles   🡪 format numpy.array | #il faut compute\_mcmc avant avec les quantiles mis en paramètres lors du compute  **depths = get\_depths\_mcmc()**  [0, 02, 04, …] (unité m)  **times = get\_times\_mcmc()**  […](unité s)  **flows50 = get\_flows\_quantile(0.5)** pour le quantile à 50% des débits  [[d1, d2 ….] débit à la profondeur 0   [...]   … en m^3.s-1  ],  **temps95 = get\_temps\_quantile(0.95)** pour le quantile à 95% des températures  [[T1, T2 ….] température à la profondeur 0   [...]   … en K  ],  **moinslog10K\_50 = get\_moinslog10K\_quantile(0.5)** par ex pour K |

# **Générateur des données et visualisation de la solution analytique**

Cette partie décrit deux modules développés en 2022 qui pourront être ajoutés dans de futures versions de MOLONARI à l’IHM pour permettre à l’utilisateur de préparer des simulations de réponses de signaux.

Le module **gen\_test.py** permet de préparer des chroniques de signaux pour les conditions aux bords de l’objet column de **core.py**, c’est-à-dire des chroniques de température pour la rivière et pour l’aquifère et des chroniques de différence de charge qui permettent de simuler des conditions météo. Le format de sortie de la classe time\_series de **gen\_test.py** sont au bon format pour initialiser l’objet column.

L’utilisateur aura également accès aux solutions analytiques proposées par *Goto et al.* dans la classe **val\_analy.py** et il pourra comparer des sorties de température entre ces valeurs analytiques de température et ces valeurs de modèle direct.

Les deux modules sont aussi documentés dans deux notebook de démo : **demo\_gen\_test.ipynd** et **demo\_val\_direct.ipynb**. Le second présente l’accord en termes de RMSE entre les résultats du modèle direct et la solution analytique de Goto et al. selon une excitation bien particulière.

## **Générateur des données**

* **Définir un objet time\_series() :** Certains paramètres de l’init ressemblent à ceux de colonne. Si les prochains développeurs de calcul souhaitent changer cette rchitecture, c’est possible sans trop de difficulté.

Paramètres :

* offset : un float, même signification que pour Column
* depth\_sensors : la liste de float des profondeurs des capteurs, la même que pour Column
* param\_time\_dates : [t\_debut, t\_fin, dt], les paramètres de la fenêtre temporelle. Le format demandé pour t\_debut et t\_fin est celui-ci : (y,m,d,h,mn,s) compatible avec la librairie datetime, dt est un scalaire exprimé en seconde
* param\_dH\_signal : [dH\_amp, P\_dh, dH\_offset], avec des scalaires permettant de calibrer le signal sinusoïdal
* param\_T\_riv\_signal : [T\_riv\_amp, P\_T\_riv, T\_riv\_offset]
* param\_T\_aq\_signal : [T\_aq\_amp, P\_T\_aq, T\_aq\_offset]
* sigma\_meas\_P : float, indique l’erreur de mesure simulée par un bruit gaussien sur les chroniques de pression
* sigma\_meas\_T : float, indique l’erreur de mesure simulée par un bruit gaussien sur les chroniques de température

Retourne :

* un objet time\_series() avec de nombreux attribut qui seront explicités plus bas.

Exemple :

|  |
| --- |
| """Fenêtre temporelle"""  # le format demandé est celui-ci : (y,m,d,h,mn,s) compatible avec la librairie datetime  t\_debut = (2010, 1, 1)  t\_fin = (2010, 5, 30, 23, 59, 59)  dt = 15\*60 # pas de temps en (s) # A AMELIORER AVEC UN CONVERTISSEUR AUTOMATIQUE D'UNITES  """Conditions limites"""  # Température de la rivière  T\_riv\_amp = 5  T\_riv\_offset = 20 +K\_offset  P\_T\_riv = 28\*24\*4\*dt  # Température de l'aquifère  T\_aq\_amp = 0  T\_aq\_offset = 12 + K\_offset  P\_T\_aq = -9999 # à mettre dans le init  # Pression différentielle  dH\_amp = 0  dH\_offset = 1  P\_dh = -9999 #14\*24\*4\*dt  """Bruit de mesure"""  sigma\_meas\_P = 0.5  sigma\_meas\_T = 0.5  """Instanciation de l'objet Time\_series"""  # un dictionnaire qui facilite le paramétrage avec des variables globales définies plus haut  time\_series\_dict\_user1 = {  "offset":.0,  "depth\_sensors":[.25, .5, .75, 1],  "param\_time\_dates": [t\_debut, t\_fin, dt],  "param\_dH\_signal": [dH\_amp, P\_dh, dH\_offset],  "param\_T\_riv\_signal": [T\_riv\_amp, P\_T\_riv, T\_riv\_offset],  "param\_T\_aq\_signal": [T\_aq\_amp, P\_T\_aq, T\_aq\_offset],  "sigma\_meas\_P": sigma\_meas\_P,  "sigma\_meas\_T": sigma\_meas\_T, #float  }  # instanciation du simulateur de données  emu\_observ\_test\_user1 = Time\_series.from\_dict(time\_series\_dict\_user1) |

NB : le formatage du pas de temps est à revoir, donner par exemple le choix d’une variable dt\_day == un pas de temps en jour, dt\_hour == un pas de temps en heures, dt\_min == un pas de temps en minutes, dt\_s == un pas de temps en seconde. L’utilisateur choisit un pas de temps et le simulateur fait la somme des dt et fait les calculs avec un dt en secondes.

* **Générer les dates des chroniques au bon format :** generate\_dates\_series()

Paramètres :

* méthode sans paramètres, les attributs self.\_dates et self.\_time\_array se mettent à jour grâce au init

Retourne :

* l’objet time\_series avec les dates mises au bon format dans self.\_dates et self.\_time\_array

Exemple :

|  |
| --- |
| >>>emu\_observ\_test\_user1.\_dates # il n'y a pas de date encore  array([None], dtype=object)  >>>emu\_observ\_test\_user1.\_generate\_dates\_series()  emu\_observ\_test\_user1.\_dates # les dates sont remplies au format demandé par la classe colonne  >>>emu\_observ\_test\_user1.\_dates # retourne les dates sont remplies au format demandé par la classe colonne  array([datetime.datetime(2010, 1, 1, 0, 0),  datetime.datetime(2010, 1, 1, 0, 15),  datetime.datetime(2010, 1, 1, 0, 30), ...,  datetime.datetime(2010, 5, 30, 23, 15),  datetime.datetime(2010, 5, 30, 23, 30),  datetime.datetime(2010, 5, 30, 23, 45)], dtype=object)  >>>emu\_observ\_test\_user1.\_time\_array #retourne un array avec les dates depuis le début de l’expérience en seconde  array([ 0, 900, 1800, ..., 12957300, 12958200, 12959100]) |

* **Générer les chroniques de pression/charge** : \_generate\_dH\_series()

Paramètres :

* méthode sans paramètres

Retourne :

* l’objet time\_series avec les attributs self.\_dH et self.\_time\_array mis à jour au format array = chronique de charge au cours du temps.

Exemple :

|  |
| --- |
| >>>emu\_observ\_test\_user1.\_dH #ici on a mis une différence de charge constante au cours du temps  array([1., 1., 1., ..., 1., 1., 1.]) |

* **Générer les chroniques de température de la rivière** : \_generate\_Temp\_riv\_series()

Paramètres :

* méthode sans paramètres

Retourne :

* l’objet time\_series avec les attributs self.\_T\_riv, self.\_dates et self.\_time\_array mis à jour au format array = chronique de charge au cours du temps.

Exemple :

|  |
| --- |
| >>>emu\_observ\_test\_user1.\_T\_riv #ici on a mis une température sinusoïdale  array([293.15 , 293.16168746, 293.17337486, ..., 297.08092218,  297.07368868, 297.06643375]) # ici températures en Kelvin |

* **Générer le fichier de chroniques de pression/température du capteur de pression différentielle MOLONARI au format du datalogger**: \_generate\_T\_riv\_dH\_series()

Paramètres :

* méthode sans paramètres

Retourne :

* l’objet time\_series avec les attributs self.\_T\_riv, self.\_dH, self.\_dates et self.\_time\_array mis à jour au format array = chronique cours du temps.
* l’attribut self.\_T\_riv\_dH\_measures est mis à jour et peut être utilisé pour instancier un objet Column dans l’init “dH\_measure”

Exemple :

|  |
| --- |
| emu\_observ\_test\_user1.\_T\_riv\_dH\_measures  [(datetime.datetime(2010, 1, 1, 0, 0), (1.0, 293.15)),  ...  (datetime.datetime(2010, 1, 11, 9, 45), (1.0, 296.759124700894)),  ...] |

* **Générer les chroniques de températures sur les capteurs du shaft au format du datalogger** :\_generate\_Shaft\_Temp\_series()

Paramètres :

* méthode sans paramètres

Retourne :

* l’objet time\_series avec les attributs self.\_T\_shaft, self.\_dates et self.\_time\_array mis à jour au format array = chronique cours du temps.
* l’attribut self.\_T\_shaft\_measures, est mis à jour et peut être utilisé pour instancier un objet Column dans l’init “T\_measures”

Exemple :

|  |
| --- |
| >>>col.\_T\_measures # valeurs par défaut très élevée à 9e5 des températures des capteurs T1, T2 et T3  array([[2.91150e+02, 2.89150e+02, 2.87150e+02],  [9.59595e+05, 9.59595e+05, 9.59595e+05],  [9.59595e+05, 9.59595e+05, 9.59595e+05],  ...,  [9.59595e+05, 9.59595e+05, 9.59595e+05],  [9.59595e+05, 9.59595e+05, 9.59595e+05],  [9.59595e+05, 9.59595e+05, 9.59595e+05]])  # valeurs maj des températures de T\_shaft np.array([[T1, T2, T3, Taq](t=0), [T1, T2, T3, Taq](t=1), ... ])  >>>emu\_observ\_test\_user1.\_T\_Shaft  array([[291.15 , 289.15 , 287.15 , 285.15 ],  [291.19003296, 289.19003296, 287.19003296, 285.15 ],  [291.19006348, 289.19003296, 287.19009399, 285.15 ],  ...,  [294.45544434, 291.38296509, 288.27304077, 285.15 ],  [294.45306396, 291.38290405, 288.27349854, 285.15 ],  [294.45068359, 291.3828125 , 288.27386475, 285.15 ]])  >>>emu\_observ\_test\_user1.\_T\_Shaft\_measures # le fichier de température du shaft par défaut au format de la colonne  [(datetime.datetime(2010, 1, 1, 0, 0),  array([291.15, 289.15, 287.15, 285.15])),  (datetime.datetime(2010, 1, 1, 0, 15),  array([9.59595e+05, 9.59595e+05, 9.59595e+05, 2.85150e+02])),  (datetime.datetime(2010, 1, 1, 0, 30),  array([9.59595e+05, 9.59595e+05, 9.59595e+05, 2.85150e+02])),  (datetime.datetime(2010, 1, 1, 0, 45),  array([9.59595e+05, 9.59595e+05, 9.59595e+05, 2.85150e+02])),  (datetime.datetime(2010, 1, 1, 1, 0),  array([9.59595e+05, 9.59595e+05, 9.59595e+05, 2.85150e+02])),  (datetime.datetime(2010, 1, 1, 1, 15),  array([9.59595e+05, 9.59595e+05, 9.59595e+05, 2.85150e+02])),  …  (datetime.datetime(2010, 1, 1, 1, 30),  array([9.59595e+05, 9.59595e+05, 9.59595e+05, 2.85150e+02]))] |

* **Générer le fichier de chroniques de température du shaft MOLONARI au format du datalogger avec perturbations de mesures de température avec erreur de mesures de température simulée**: \_generate\_perturb\_Shaft\_Temp\_series()

Paramètres :

* méthode sans paramètres

Retourne :

* l’objet time\_series avec les attributs self.\_T\_shaft\_perturb, self.\_dates et self.\_time\_array mis à jour au format array = chronique cours du temps.
* l’attribut self.\_T\_shaft\_measures\_perturb, est mis à jour et peut être utilisé pour instancier un objet Column dans l’init “T\_measures”

Exemple :

|  |
| --- |
| >>>emu\_observ\_test\_user1.\_generate\_perturb\_Shaft\_Temp\_series()  >>>emu\_observ\_test\_user1.\_T\_shaft\_measures\_perturb  array([[18.31054887, 17.07740364, 15.58220471, 14.13629319],  [18.42680324, 16.85507814, 15.53947417, 13.90031993],  [18.45884315, 17.02728014, 15.62675485, 13.96020251],  ...,  [21.03653478, 18.56741759, 16.0042066 , 13.9411373 ],  [21.01353776, 18.28902244, 16.01057728, 14.14870168],  [21.14024195, 18.50122474, 15.95205776, 14.09108076]]) |

* **Générer le fichier de chroniques de température du capteur de pression/température MOLONARI au format du datalogger avec erreur de mesures de température/pression simulées:** \_generate\_perturb\_T\_riv\_dH\_series()

Paramètres :

* méthode sans paramètres

Retourne :

* l’objet time\_series avec les attributs self.\_dH\_perturb, self.\_T\_riv\_perturb, self.\_dates et self.\_time\_array mis à jour au format array = chronique cours du temps.
* l’attribut self.self.\_T\_riv\_dH\_perturb, est mis à jour et peut être utilisé pour instancier un objet Column dans l’init “T\_measures”

Exemple :

|  |
| --- |
| >>>emu\_observ\_test\_user1.\_generate\_perturb\_T\_riv\_dH\_series()  >>>emu\_observ\_test\_user1.\_T\_riv\_dH\_measures\_perturb() # ici charge constante à 1 m perturbée  [(datetime.datetime(2011, 8, 1, 0, 0),  (0.9662380752020394, 20.07225019743682)),  (datetime.datetime(2011, 8, 1, 0, 15),  (0.9855601212803974, 20.131963026647753)),  (datetime.datetime(2011, 8, 1, 0, 30),  (1.0164725331138886, 20.22847129485494)),  (datetime.datetime(2011, 8, 1, 0, 45),  (0.9891967134434072, 20.156322318820703)),  (datetime.datetime(2011, 8, 1, 1, 0),  (1.0297360996475038, 20.360987284654527)),  (datetime.datetime(2011, 8, 1, 1, 15),  (0.9688234643816331, 20.392391087765752)),  (datetime.datetime(2011, 8, 1, 1, 30),  . . .  (0.9719296831820816, 20.41472299672638))] |

* **Interaction entre time\_series et les objets Layers(), Column() et Time\_series() pour mettre à jour l’objet Column() avec les profils de température calculés par le modèle direct à partir des observations simulées** :\_measures\_column\_one\_layer()

Paramètres :

* Coulumn, un objet colonne
* liste de Layer,
* nb\_cells, le nombre de cellules pour le modèle direct

Retourne :

* aucun retour, les objets sont mis à jour notamment pour Time\_series qui met à jour avec les valeur de compute\_solve\_transi : self.\_T\_measures, self.\_T\_Shaft, self.\_T\_Shaft\_measures

Exemple :

|  |
| --- |
| >>>emu\_observ\_test\_user1.\_T\_Shaft\_measures # le fichier de température du shaft par défaut au format de la colonne  [(datetime.datetime(2010, 1, 1, 0, 0),  array([291.15, 289.15, 287.15, 285.15])),  (datetime.datetime(2010, 1, 1, 0, 15),  array([9.59595e+05, 9.59595e+05, 9.59595e+05, 2.85150e+02])),  (datetime.datetime(2010, 1, 1, 0, 30),  array([9.59595e+05, 9.59595e+05, 9.59595e+05, 2.85150e+02])),  (datetime.datetime(2010, 1, 1, 0, 45),  array([9.59595e+05, 9.59595e+05, 9.59595e+05, 2.85150e+02])),  (datetime.datetime(2010, 1, 1, 1, 0),  array([9.59595e+05, 9.59595e+05, 9.59595e+05, 2.85150e+02])),  (datetime.datetime(2010, 1, 1, 1, 15),  array([9.59595e+05, 9.59595e+05, 9.59595e+05, 2.85150e+02])),  …  (datetime.datetime(2010, 1, 1, 1, 30),  array([9.59595e+05, 9.59595e+05, 9.59595e+05, 2.85150e+02]))]  >>>emu\_observ\_test\_user1.\_measures\_column\_one\_layer(col, layers\_list, nb\_cells)  >>>emu\_observ\_test\_user1.\_T\_Shaft\_measures # les mesures d’observations sont bien mises à jour et peuvent être utilisées par la MCMC  [(datetime.datetime(2010, 1, 1, 0, 0),  array([291.15, 289.15, 287.15, 285.15])),  (datetime.datetime(2010, 1, 1, 0, 15),  array([291.19003296, 289.19003296, 287.19003296, 285.15 ])),  (datetime.datetime(2010, 1, 1, 0, 30),  array([291.19006348, 289.19003296, 287.19009399, 285.15 ])),  (datetime.datetime(2010, 1, 1, 0, 45),  array([291.19012451, 289.19006348, 287.19015503, 285.15 ])),  (datetime.datetime(2010, 1, 1, 1, 0),  array([291.19015503, 289.19006348, 287.19018555, 285.15 ])),  (datetime.datetime(2010, 1, 1, 1, 15),  array([291.19021606, 289.19012451, 287.19024658, 285.15 ])),  (datetime.datetime(2010, 1, 1, 1, 30),  array([291.19024658, 289.19015503, 287.19024658, 285.15 ])),  (datetime.datetime(2010, 1, 1, 1, 45), |

## **Solution analytique**

Ces méthodes permettent de visualiser la solution analytique et la comparer avec des données sorties du modèle direct. Les fonctions détaillées dans cette partie sont des méthodes de la classe Column, et elles ne peuvent pas être utilisées avant d’avoir appelé le modèle direct une première fois.

La solution analytique en cas général est :

avec , ,

La solution analytique en cas conductif est :

* **Définir un objet Analy\_Sol() :**

Paramètres :

* column\_exp
* time\_series
* monolayer
* nb\_cells

Retourne :

* un objet analy\_sol avec certains attributs.

Exemple :

|  |
| --- |
| analy\_sol\_dict = {  "column\_exp" : colonne\_exp,  "time\_series" : time\_series,  "monolayer" : monolayer,  "nb\_cells" : nb\_cells, |

* **Récupérer les coefficients utilisés dans la solution analytique : compute\_a, compute\_b**

Paramètres :

* aucun

Retourne :

* valeur des coefficients calculés d’après les propriétés hydro-thermiques et la période de signal.

Exemple :

|  |
| --- |
| >>> analy\_sol\_exp.compute\_a()  3.3823968938956313  >>> analy\_sol\_exp.compute\_b()  3.3825115862919444 |

* **Calculer la température calculée par la solution analytique en cas général : compute\_temps\_general**

Paramètres :

* aucun

Retourne :

* tableau des températures calculées analytiquement en cas général.

Exemple :

|  |
| --- |
| >>> analy\_sol\_exp.compute\_temp\_general()  array([[19.93348742, 19.97025726, 20.00702969, ..., 21.86397401,  21.83151255, 21.79889098],  [19.80717211, 19.842684 , 19.87820964, ..., 21.91417479,  21.88337573, 21.85241202],  [19.68954571, 19.72380338, 19.75808519, ..., 21.95685757,  21.9276803 , 21.89833451],  ...,  [20.02302351, 20.02165697, 20.02028854, ..., 19.91251124,  19.91362946, 19.91475523],  [20.02701995, 20.02570645, 20.0243907 , ..., 19.9115726 ,  19.91262644, 19.91368792],  [20.03069522, 20.02943418, 20.02817058, ..., 19.91089097,  19.91188231, 19.91288135]]) |

* **Calculer la température calculée par la solution analytique en cas conductif : compute\_temp\_cond**

Paramètres :

* aucun

Retourne :

* tableau des températures calculées analytiquement en cas conductif.

Exemple :

|  |
| --- |
| >>> analy\_sol\_exp.compute\_temp\_cond()  array([[19.93348746, 19.97025727, 20.00702969, ..., 21.86397294,  21.8315115 , 21.79888994],  [19.80717244, 19.84268427, 19.87820985, ..., 21.9141715 ,  21.88337249, 21.85240884],  [19.6895466 , 19.72380417, 19.75808588, ..., 21.95685196,  21.92767477, 21.89832907],  ...,  [20.02302094, 20.02165455, 20.02028627, ..., 19.91252102,  19.91363912, 19.91476476],  [20.0270169 , 20.02570355, 20.02438794, ..., 19.91158259,  19.91263631, 19.91369767],  [20.03069171, 20.02943082, 20.02816736, ..., 19.91090114,  19.91189236, 19.91289129]]) |

* **Calculer la RMSE du modèle direct par rapport aux solutions analytiques sur des conditions aux limites simples et dans un cas de monocouche : generate\_RMSE\_analytical**

Paramètres :

* time\_series : déjà expliqué dans la partie précédente
* colonne\_exp : un objet de Colonne
* monolayer : un objet de Layer

Retourne :

* array contenant les RMSE du modèle direct par rapport aux solutions analytiques.

Exemple :

|  |
| --- |
| """Colonne"""  col\_dict = {  "river\_bed": 1.,  "depth\_sensors": depth\_sensors, #En vrai y aura une 4e valeur ici mais ca prendra en charge pareil  "offset": .0,  "dH\_measures": time\_series.\_T\_riv\_dH\_measures,  "T\_measures": time\_series.\_T\_Shaft\_measures,  "sigma\_meas\_P": time\_series.\_sigma\_P, #float  "sigma\_meas\_T": time\_series.\_sigma\_T, #float  }  colonne\_exp = Column.from\_dict(col\_dict)  """Layer"""  monolayer\_dict = {  "name": "sable",  "zLow": depth\_sensors[-1],  "moinslog10K": moinslog10K,  "n": n,  "lambda\_s": lambda\_s,  "rhos\_cs": rho\_cs  }  monolayer = Layer.from\_dict(monolayer\_dict) |

|  |
| --- |
| >>> analy\_sol\_exp.generate\_RMSE\_analytical(time\_series, colonne\_exp, monolayer)  --- Compute Solve Transi ---  One layer : moinslog10K = 12, n = 0.1, lambda\_s = 2, rhos\_cs = 4000000.0  Done.  array([0.00123811, 0.00315148, 0.00548253, 0.00372034]) |

NB : Dans le module **val\_analy.py** nous n’avons pas créé des plotteurs. Cependant, nous avons créé une démo où nous faisons la comparaison visuelle du modèle direct et de la solution analytique avec une durée choisie.

Exemple :

|  |
| --- |
| >>> t\_fin =1500  >>> for i,id\_sens in enumerate(analy\_sol\_exp.\_id\_sensors) : plt.plot(analy\_sol\_exp.analy\_temp\_general[id\_sens,:t\_fin],label="Tanaly{}".format(i+1))  plt.plot(time\_series.\_T\_Shaft[:t\_fin,i],linestyle="--",label="Tmd{}".format(i+1))  plt.legend()  plt.grid() |