UE31

MOLONARI

rapport de projet

sur l'estimation de paramètres thermo-hydrologiques des échanges nappe-rivière

Tous mes remerciements à mes encadrants Nicolas Flipo et Thomas Romary, enseignants-chercheurs au centre de géosciences des Mines de Paris.

Dan MAUREL

27 décembre 2023

Table des matières

1 Stratégies d'estimation des paramètres physiques

1.1 Problématisation

1.1.1 Équations physique : échanges thermiques advecto-conductifs

La conservation de l'énergie thermique s'écrit

$$c_m \rho_m \frac{\partial \theta}{\partial t} + \operatorname{div}(\vec{\phi}) = 0$$

avec $\vec{\phi}$ le flux surfacique thermique instantané. Par la loi de Darcy et la loi de Fourier,

$$\vec{\phi} = -c_w \rho_w K \vec{\operatorname{grad}}(H)\theta - \lambda_m \vec{\operatorname{grad}}(\theta)$$

En 1D, l'équation thermique est

$$c_m \rho_m \frac{\partial \theta}{\partial t} = c_w \rho_w \left(K \frac{\partial^2 H}{\partial z^2} + \frac{dK}{dz}\frac{\partial H}{\partial z} \right) \theta + \left(\rho_w c_w K \frac{\partial H}{\partial z} + \frac{d\lambda_m}{dz} \right) \frac{\partial \theta}{\partial z} + \lambda_m \frac{\partial^2 \theta}{\partial z^2} \qquad (1)$$

En 1D, l'équation de diffusion hydraulique s'énonce

$$S \frac{\partial H}{\partial t} = \frac{dK}{dz}\frac{\partial H}{\partial z} + K \frac{\partial^2 H}{\partial z^2} \qquad (2)$$

Les deux équations se mettent sous la forme

$$y_t = \alpha(z) y + \kappa(z) y_z + D(z) y_{zz}$$

1.1.2 Problématique

Dans la continuité de [1], un objectif du projet MOLONARI consiste en l'estimation des paramètres physiques du sous-sol présents dans les équations (1) et (2) à partir des mesures en température et en charge hydraulique.

1.2 Détermination de (α, κ, D) par une méthode différentielle

Il est possible de déterminer localement (α, κ, D) à partir d'une solution θ à (1) lorsqu'il y a des variations de θ, θ_z et θ_{zz} suffisamment indépendantes.

1.2.1 Une condition de déterminabilité

Supposons chercher à déterminer (α, κ, D) pour une profondeur z fixée. S'il existe des temps t_1, t_2 et t_3 tels que les $v(z,t_i) = \begin{bmatrix} \theta(z,t_i) & \dfrac{\partial \theta}{\partial z}(z,t_i) & \dfrac{\partial^2 \theta}{\partial z^2}(z,t_i) \end{bmatrix}$ soient non-colinéaires, alors

$$\begin{pmatrix} \alpha(z) \\ \kappa(z) \\ D(z) \end{pmatrix} = \begin{bmatrix} v(z,t_1) \\ v(z,t_2) \\ v(z,t_3) \end{bmatrix}^{-1} \begin{pmatrix} \theta_t(z,t_1) \\ \theta_t(z,t_2) \\ \theta_t(z,t_3) \end{pmatrix} \qquad (3)$$

D'un point de vue physique, cette condition revient à considérer un système évoluant de manière suffisamment variée. En particulier, ceci exclut un cas de régime permanent.

1.2.2 La méthode différentielle

Si nous disposons d'estimations $(v(z,t_i))_{i\in[\![1:N]\!]}$ ainsi que $(\theta_t(z,t_i))_{i\in[\![1:N]\!]}$, nous pouvons chercher à résoudre le problème de minimisation

$$\min_{\alpha(z),\kappa(z),D(z)} \left\| \begin{pmatrix} v(z,t_1) \\ \vdots \\ v(z,t_N) \end{pmatrix} \begin{pmatrix} \alpha(z) \\ \kappa(z) \\ D(z) \end{pmatrix} - \begin{pmatrix} \theta_t(z,t_1) \\ \vdots \\ \theta_t(z,t_N) \end{pmatrix} \right\|^2$$

en incluant possiblement des contraintes physiques sur ($\alpha(z)$,$\kappa(z)$, $D(z)$).

1.2.3 Limitations pratiques

Avec la méthode différentielle il faut être capable d'obtenir des estimations satisfaisantes de θ_t, θ, θ_z et θ_{zz} : les trois dernières grandeurs sont particulièrement importantes car l'inversion (3) ne propage pas les erreurs de manière linéaire.

Le calcul des dérivées temporelles paraît facile à implémenter car il suffirait de choisir la fréquence d'échantillonnage adéquate. Un jour étant probablement le plus petit temps caractéristique de variation thermique, nous nous attendons à avoir en moyenne des θ_t faibles devant les écart-types induits par les erreurs de mesure.

Quant à θ, θ_z et θ_{zz}, ils nécessitent des mesures avec une bonne résolution spatiale, ce qui est potentiellement limité par notre capacité à déployer un grand nombre de capteurs sur une petite distance. Sans une résolution spatiale adéquate, des estimateurs de dérivées première et seconde comme $\frac{T_{i+1}-T_i}{dz}$ ou $\frac{T_{i+1}+T_{i-1}-2T_i}{dz^2}$ auront probablement des biais systématiques difficiles à corriger, en plus d'un bruitage lié aux erreurs de mesures. Ce sont ces biais systématiques qui rendent probablement non-pertinente la méthode différentielle lorsque la densité spatiale de capteurs est insuffisante.

1.3 Méthode d'estimation probabiliste

S'il est généralement envisageable de calculer (α, κ, D) à partir d'une solution θ de (1), ce n'est pas toujours le cas pour des sous-paramètres intervenant dans les expressions de α, κ et D. Par exemple, en milieu homogène avec H en régime permanent (donc $\alpha = 0$), pour l'équation (1) il existe typiquement une infinité de manières de choisir ($\rho_m c_m$, λ_m, K, H_z) afin d'obtenir (κ_0, D_0) et donc une solution y_0 fixée. Si l'on veut savoir quels jeux de sous-paramètres sont les plus pertinents, il nous faut alors quantifier cette pertinence, ce qui peut-être fait par l'introduction de densités probabilistes sur les sous-paramètres qualifiées d'a priori.

1.3.1 Approche bayésienne et méthode MCMC

Nous supposons disposer d'un modèle physique tel qu'en se donnant des paramètres du milieu X, nous pouvons par résolution numérique calculer des grandeurs $Y = F(X)$ (au delà d'un temps d'amortissement des erreurs en les conditions initiales lorsque ces dernières sont mal connues). Typiquement Y consiste ici en des températures à (z, t) fixés. Nous disposons par ailleurs de mesures réelles Y_{meas} de ces grandeurs. Si nous supposons que $Y_{\text{meas}} = Y + \varepsilon$ avec ε une erreur aléatoire de densité estimée f_ε indépendante de Y (si des paramètres de la loi de ε sont inconnus, nous pouvons les ajouter aux paramètres X), alors

$$f_{X|Y_{\text{meas}}} = \frac{f_{Y_{\text{meas}}|X} f_X}{f_{Y_{\text{meas}}}}$$

donc

$$f_{X|Y_{\text{meas}}=y_{\text{meas}}} \propto f_{F(X)+\varepsilon|X} f_X$$

Afin d'estimer la distribution probabiliste de $(X|Y_{\text{meas}} = y_{\text{meas}})$, nous supposons connue la densité a priori f_X ainsi que $f_{F(X)+\varepsilon|X} = f_{\varepsilon=y_{\text{meas}}-F(X)} = f_\varepsilon(y_{\text{meas}} - F(X))$. Il reste par exemple à calculer le coefficient de proportionnalité afin de mieux connaître $f_{X|Y_{\text{meas}}=y_{\text{meas}}}$. Nous pouvons essayer de calculer numériquement $\int f_{\varepsilon=y_{\text{meas}}-F(X)} f_X dX$ mais cette approche est peu adaptée à un domaine d'intégration $\mathcal{D}$ trop vaste.

Si l'on suppose que la distribution probabiliste cible $f_{X|Y_{\text{meas}}=y_{\text{meas}}}$ est concentrée en quelques îlots entourés d'espace où la densité y est quasiment nulle, une idée consiste à limiter notre espace de travail à ces zones d'importance. Il faut donc procéder à une exploration de $\mathcal{D}$ avec essentiellement pour seul outil la possibilité d'évaluer ponctuellement $f_{\varepsilon=y_{\text{meas}}-F(X)} f_X$. Pour se déplacer dans ce domaine sans avoir de connaissances a priori sur $f_{\varepsilon=y_{\text{meas}}-F(X)} f_X$, il paraît nécessaire de faire appel à une marche aléatoire. Il faut dans ce cas trouver la bonne règle stochastique prenant en compte la pseudo-densité $f_{\varepsilon=y_{\text{meas}}-F(X)} f_X$ afin de cibler les régions d'intérêt, mais assurant également une recherche suffisamment exhaustive.

A partir d'une marche aléatoire bien pensée $(*)$, il est alors envisageable d'estimer directement la distribution en jeu. Idéalement, au delà d'un certain temps après l'initialisation, le marcheur aléatoire

devrait avoir parcouru les zones d'importance tout en ayant perdu la mémoire de son origine et la probabilité qu'il soit présent en un voisinage $\mathcal{V}_{X_0}$ devrait être proportionnelle à $\int_{\mathcal{V}_{X_0}} f_{\varepsilon=y_{\text{meas}}-F(X)} f_X dX$: il suffit en pratique de calculer sa fréquence de présence en $\mathcal{V}_{X_0}$ pour estimer cette probabilité.

$(*)$ Une telle marche aléatoire existe : nous allons utiliser la méthode de Métropolis-Hasting qui admet les propriétés escomptées, notamment l'estimation d'une probabilité à partir d'une fréquence de passage.

1.3.2 Principes de génération d'un déplacement élémentaire stochastique

Pour générer le prochain pas d'un marcheur aléatoire, il est possible de procéder en deux étapes :

(i) proposer un déplacement élémentaire selon un processus stochastique.

(ii) décider d'accepter la proposition avec une probabilité calculée à la volée.

La proposition doit être suffisamment indépendante de l'histoire passée du marcheur pour amener à l'exploration de régions non-visitées. La probabilité d'acceptation doit quant à elle prendre en compte $f_{\varepsilon=y_{\text{meas}}-F(X)} f_X$ afin de favoriser les régions les plus prometteuses.

Les objets nécessaires à l'étape (i) sont donc le passé $\mathcal{P}$ du marcheur et un générateur aléatoire G tel que $\text{Prop} = G(\mathcal{P})$ soit une proposition de déplacement. Pour l'étape (ii), il faut le passé $\mathcal{P}$, une proposition Prop, la fonction $f_{\varepsilon=y_{\text{meas}}-F(X)} f_X$, une fonction de calcul de probabilité d'acceptation $A : \mathcal{P}, \text{Prop}, f \mapsto [0,1]$ et un générateur de réels tirés dans $[0,1]$ selon une loi uniforme. Le passé $\mathcal{P}$ du marcheur regroupe à un instant donné tous les états des itérations précédentes.

1.3.3 Énergie d'un état

Nous postulons une erreur de mesure gaussienne ϵ, invariante dans le temps et centrée. Dans le cas iid, la densité non-normalisée est

$$f_{\varepsilon=y_{\text{meas}}-F(X)} f_X = \exp\left(-\frac{\|y_{meas} - F(X)\|^2}{2\sigma^2}\right) f_X$$

en supposant connu l'écart-type σ de ϵ. Dans le cas où σ est lui-même un paramètre inconnu, il faut sortir le facteur $1/\sigma^n$ de la constance de normalisation implicite dans l'écriture $\propto$ car cette dernière ne doit pas dépendre des paramètres inconnus :

$$f_{\varepsilon=y_{\text{meas}}-F(X)} f_X = \sigma^{-n} \exp\left(-\frac{\|y_{meas} - F(X)\|^2}{2\sigma^2}\right) f_X$$

Nous pouvons réunir les deux cas précédents en posant

$$\pi_X = f_{\varepsilon=y_{\text{meas}}-F(X)} f_X = \exp\left(-\mathcal{E}(X)\right)$$

avec $\mathcal{E}(X)$ l'énergie associée aux paramètres inconnus X. Ce formalisme est d'ailleurs compatible avec un cas plus général en posant simplement $\mathcal{E}(X) = -\ln\left(f_{\varepsilon=y_{\text{meas}}-F(X)} f_X\right)$.

1.3.4 Probabilité d'acceptation

1.3.4.1 Acceptation de Metropolis-Hasting

En notant $q(X_0 \to X_1)$ la probabilité de proposer un état X_1 depuis X_0, la méthode de Metropolis-Hasting pour construire la probabilité d'acceptation de X_1 consiste en

$$\min\left(1, \frac{\pi(X_1)}{\pi(X_0)} \frac{q(X_1 \to X_0)}{q(X_0 \to X_1)}\right)$$

Dans le cas usuel symétrique (dit à incrément symétrique), l'expression se simplifie en

$$\min\left(1, \frac{\pi(X_1)}{\pi(X_0)}\right)$$

et avec l'énergie, l'équation devient

$$\min\left(1, \exp(\mathcal{E}(X_0) - \mathcal{E}(X_1))\frac{f(X_1)}{f(X_0)}\right)$$

1.3.4.2 Lois a priori uniformes

Pour un système physique inconnu, il est courant de ne pouvoir donner que des intervalles a priori $\mathcal{S}$ pour les paramètres inconnus, ce qui revient à considérer une densité f_X uniforme de support $\mathcal{S}$. Dans ce cas particulier, l'expression de la probabilité d'acceptation se simplifie en

$$A(X_0, X_1) = \left| \begin{array}{ccc} 0 & \text{if} & X_1 \notin \mathcal{S} \\ 1 & \text{elif} & \mathcal{E}(X_1) \leqslant \mathcal{E}(X_0) \\ \exp(\mathcal{E}(X_0) - \mathcal{E}(X_1)) & \text{else} & \end{array} \right.$$

Autrement dit, la marche aléatoire s'effectue sur $\mathcal{S}$ et un nouvel état proposé $X_1 \in \mathcal{S}$ est nécessairement accepté s'il diminue l'énergie courante, sinon sa probabilité d'acceptation décroît exponentiellement avec $|\mathcal{E}(X_0) - \mathcal{E}(X_1)|$.

1.3.5 Proposition stochastique : méthode à état courant délocalisé

La méthode la plus simple consiste à proposer un nouvel état X_1 à partir d'une distribution probabiliste unimodale centrée en l'état courant X_0. Le problème de cette approche est qu'elle favorise un cantonnement du marcheur aléatoire à un maximum local de $f_{X|Y_{\text{meas}}=y_{\text{meas}}}$. Une idée consiste alors à délocaliser l'état courant afin de permettre l'exploration de plusieurs maxima : dit autrement, il n'y a plus un seul mais plusieurs états courants depuis lesquels le marcheur aléatoire peut partir. Une autre manière de voir les choses consiste à dire qu'il y a plusieurs marcheurs aléatoires, ou encore plusieurs chaînes de Markov, bien que cela puisse faire perdre de vue la propriété markovienne.

La situation où nous considérons un état courant délocalisé $\mathcal{X}(t) = (X_1, \cdots, X_N)$ à l'itération t est bien compatible avec le formalisme antérieur de marche aléatoire en considérant désormais comme densité cible la densité produit

$$f_{\mathcal{X}|Y^n_{\text{meas}}=(y_{\text{meas}},\ldots,y_{\text{meas}})} : \mathcal{X} \mapsto \prod_{i=1}^{n} f_{X_i|Y_{\text{meas}}=y_{\text{meas}}}$$

avec une fonction de modèle direct qui est alors

$$\tilde{F} : \mathcal{X} \mapsto \begin{pmatrix} F(X_1) \\ \vdots \\ F(X_n) \end{pmatrix}$$

ce qui permet d'obtenir une sympathique expression pour l'énergie, dans le cas usuel où les ε_i sont gaussiennes iid :

$$\mathcal{E}(\mathcal{X}) = \sum_{i=1}^{n} \mathcal{E}(X_i)$$

Pour utiliser au mieux l'état délocalisé il paraît pertinent de proposer $\mathcal{X}(t+1)$ en faisant intervenir des termes $X_j(t)$ dans la génération de $X_i(t+1)$: c'est justement ce qui est fait dans la méthode DiffeRential Evolution and Adaptive Markov-chainS (DREAMS) (par exemple présenté dans [2]).

1.3.5.1 Méthode DREAMS

La proposition d'un nouvel état $\mathcal{X}(t+1)$ est constituée de $N-1$ sous-états précédents et d'une proposition d'un nouveau sous-état $X_i(t+1)$ avec i parcourant le cycle $[\![1; N]\!]$ au cours des propositions successives. En supposant que la méthode de proposition est symétrique, la probabilité d'acceptation se résume à

$$A(\mathcal{X}(t), \mathcal{X}(t+1)) = \left| \begin{array}{ccc} 0 & \text{if} & X_i(t+1) \notin \mathcal{S} \\ 1 & \text{elif} & \mathcal{E}(X_i(t+1)) \leqslant \mathcal{E}(X_i(t)) \\ \exp(\mathcal{E}(X_i(t)) - \mathcal{E}(X_i(t+1))) & \text{else} & \end{array} \right.$$

La proposition de $X_{i+1}(t+1)$ est générée par une variation élémentaire dX_i telle que

$$\left| \begin{array}{lcl} dX_{i,\bar{I}} & = & 0 \\ dX_{i,I} & = & \alpha_I + \beta_I \sum\limits_{(k,\ell)\in \mathrm{P}} (X_{k,I} - X_{\ell,I}) \end{array} \right.$$

où les variables aléatoires sont

I	un sous-ensemble de $[\![1;d]\!]$ avec d la dimension de X_i
α_I	une variable de saut
β_I	une variable d'intéraction
P	un ensemble de couples d'indices distincts de $[\![1;N]\!]$

L'ensemble I est construit :

• en tirant au hasard m parmi $[\![1,d]\!]$ selon des probabilités $p_{1\leqslant i\leqslant d}$.

• puis en tirant uniformément d entiers s_i dans $[\![1,d]\!]$ tels que si $s_i \leqslant m$ alors i est ajouté à I. Dans le cas où I est resté vide, on tire uniformément $i \in [\![1,d]\!]$ et l'on pose $I = \{i\}$.

Afin de favoriser les déplacements stochastiques admettant un bon taux d'acceptation, nous pouvons initialiser les p_i à $1/d$ puis une fois la phase de "burn-in" finie, poser

$$p_i = \frac{\sum\limits_{t=0}^{b} \|dX(t)\| \delta_{m=i}}{\sum\limits_{t=0}^{b} \|dX(t)\|}$$

L'ensemble P est généré en tirant de manière équiprobable $|P|$ couples de sous-états d'indices dans $[\![1;N]\!]\backslash\{i\}$ et finalement

$$\alpha_I \sim \mathcal{N}(0, cI_{|I|}) \qquad \beta \sim \mathcal{U}^{|I|}\left(\gamma\frac{1-c}{\sqrt{|P||I|}}, \gamma\frac{1+c}{\sqrt{|P||I|}}\right)$$

avec c et γ des paramètres fixés.

Le mode de génération d'une proposition de nouvel état consiste à modifier $X_i(t)$ en utilisant les autres sous-états $X_{j\neq i}(t)$, lesquels sont recopiés dans l'état $\mathcal{X}(t+1)$. Implicitement la densité q devrait prendre en compte le temps auquel s'effectue une proposition $\mathcal{X}_0 \to \mathcal{X}_1$, ce qui revient rigoureusement à considérer plutôt :

• $\overline{\mathcal{X}} = (\mathcal{X}, t)$ comme état

• $\overline{q}(\overline{\mathcal{X}}_0 \to \overline{\mathcal{X}}_1) = q(\mathcal{X}_0 \overset{r(\min(t_0,t_1),N)}{\longrightarrow} \mathcal{X}_1)\delta((t_0-t_1)^2-1)$ comme densité de transition

$r = r(\min(t_0,t_1),N)$ est le reste euclidien indiquant le numéro du sous-état cible X_r modifié par la proposition. Ainsi définie, $\overline{q}$ est symétrique, ce qui montre que DREAMS est une forme particulière de méthode de Metropolis-Hasting à incrément symétrique et justifie l'expression de la probabilité d'acceptation donnée plus tôt.

1.3.5.2 Test sur une densité bimodale

Nous considérons une distribution cible bimodale, ie comportant deux maxima.

argument	nom	choix pour le test
F	modèle	$x \mapsto x^2$
y_{meas}	mesure	$1+\epsilon$ avec $\epsilon \sim \mathcal{N}(0,\sigma)$
D	domaine d'exploration a priori	$[-2,2]$
σ	erreur caractéristique de mesure	0.1

Avec une implémentation python de DREAMS, nous obtenons bien une densité bimodale recherchée, avec deux pics centrés en -1 et 1 :

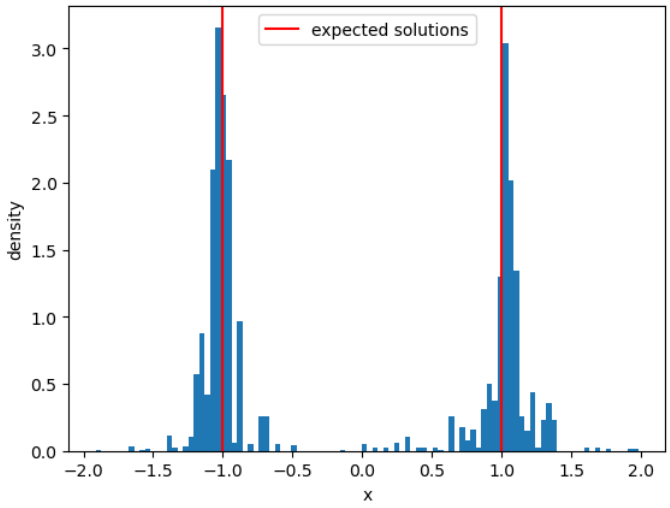

FIGURE 1 – test DREAMS sur densité a posteriori bimodale

1.3.5.3 Critère de Gelman-Rubin

A l'initialisation de la marche aléatoire fait suite une phase dit de "burn-in" pendant laquelle le marcheur découvre progressivement les régions d'importance de la distribution probabiliste. Ce n'est qu'une fois cette phase passée que le comportement du marcheur traduit correctement la connaissance acquise sur la densité cible. Dans DREAMS, une estimation de la fin du "burn-in" peut s'effectuer avec le critère de Gelman-Rubin. L'idée consiste à comparer les variances intra et inter chaînes : nous nous attendons à ce que ces dernières se stabilisent à des échelles différentes, typiquement une variance de paramètre intra-chaîne devrait être plus faible qu'une variance de paramètre inter-chaînes car l'espace exploré par une chaîne (ou sous-état) devrait être plus petit que celui exploré par l'ensemble des chaînes (ou sous-états). En pratique, nous calculons pour chaque paramètre k les variances

$$V_{k,intra} = \frac{2}{NT} \sum_{i=1}^{N} \sum_{t=\lfloor T/2 \rfloor}^{T} (X_{i,k}(t) - \overline{X_{i,k}}^{t})^2$$

$$V_{k,inter} = \frac{1}{N} \sum_{i=1}^{N} (\overline{X_{i,k}}^{t} - (\overline{X}^{t,i})_k)^2$$

avec $\overline{X_{i,k}}^{t}$ la moyenne empirique en temps sur $[\![\lfloor T/2 \rfloor, T]\!]$, et $\overline{X}^{t,i}$ la moyenne empirique de $\overline{X_{i,k}}^{t}$ sur les sous-états $\{1, \ldots, N\}$. Un indicateur de fin de "burn-in" est alors

$$R_k = \sqrt{1 + \frac{V_{k,intra}}{V_{k,inter}}}$$

et cette fin est usuellement considérée comme atteinte lorsque les R_k sont suffisamment petits. Le choix de $\lfloor T/2 \rfloor$ plutôt que 0 est généralement utilisé afin de limiter l'influence des premières itérations non-pertinentes pour mesurer une propriété asymptotique.

2 Modèle physique

Afin d'utiliser une méthode MCMC, nous devons disposer d'un modèle physique et des méthodes numériques appropriées pour calculer $F(X)$. Les équations fondamentales du modèle physique ont déjà été présentées, il reste donc à proposer une méthode de résolution de ces équations.

2.1 Modèle physique simplifié

Il est intéressant de considérer les équations (1) et (2) sous des hypothèse simplificatrices, permettant par exemple d'obtenir une solution analytique θ. Avec ce modèle simplifié, nous pouvons :

• proposer des critères d'échantillonnage spatiaux et temporels afin de pouvoir reconstituer θ efficacement à partir des mesures.

• estimer le temps de propagation des conditions limites dans le milieu.

• comparer les résultats des méthodes de résolution numérique aux solutions analytiques du modèle simplifié.

• interpréter des observations physiques.

• tenter d'estimer κ et D à partir des mesures en supposant les hypothèses simplificatrices vérifiées (dans ce modèle simplifié nous avons $\alpha = 0$).

2.1.1 Hypothèses

Nous considérons qu'en profondeur la température est constante, alors qu'en surface la température suit des variations journalières et annuelles. La charge hydraulique est en régime permanent. Nous supposons que les variations surfaciques en température se propagent dans un milieu homogène sous la forme d'une superposition de deux ondes thermiques (voir [3]) :

$$\theta(z,t) = \theta_{surf} + \theta_1 e^{-a_1 z}\cos\left(\frac{2\pi}{P_1}t - b_1 z\right) + \theta_2 e^{-a_2 z}\cos\left(\frac{2\pi}{P_2}t - b_2 z\right)$$

avec

• $a = \dfrac{1}{2D}\left(\sqrt{\dfrac{\sqrt{\kappa^4 + (8\pi D/P)^2} + \kappa^2}{2}} + \kappa\right)$

• $b = \dfrac{1}{2D}\sqrt{\dfrac{\sqrt{\kappa^4 + (8\pi D/P)^2} - \kappa^2}{2}}$

• $P_1 = 1$ an et $P_2 = 1$ jour

Pour plus de fidélité, nous pourrions prendre en compte une évolution de θ_2 au cours de l'année. Nous pourrions également ajouter un déphasage entre l'excitation journalière et annuelle, ou encore un écart entre la température moyenne de surface et la température moyenne en profondeur.

Pour une profondeur z fixée, à partir de la mesure de $t \mapsto \theta(z,t)$ il théoriquement possible de dissocier les contributions annuelle et journalière à l'aide d'une méthode de Fourier. Nous pouvons donc nous ramener à des études séparées des signaux

$$\begin{cases} \theta_1(z,t) &= \theta_{surf} + \theta_1 e^{-a_1 z}\cos\left(\dfrac{2\pi}{P_1}t - b_1 z\right) \\ \theta_2(z,t) &= \theta_2 e^{-a_2 z}\cos\left(\dfrac{2\pi}{P_2}t - b_2 z\right) \end{cases}$$

2.1.2 Échantillonnage spatial

Nous supposons disposer de capteurs consécutivement positionnés à des profondeurs $z_1, \ldots, z_N$, avec une incertitude σ sur les mesures.

2.1.2.1 Enveloppe exponentielle

Pour quantifier la décroissance exponentielle entre deux capteurs consécutifs i et $i+1$, nous voulons typiquement que

$$\theta\left(e^{-az_i} - e^{-az_{i+1}}\right) > \sigma$$

ce qui est équivalent à

$$\begin{cases} \theta e^{-az_i} > \sigma \\ -\dfrac{1}{a}\ln\left(1 - \dfrac{\sigma}{\theta}e^{az_i}\right) < z_{i+1} - z_i \end{cases}$$

2.1.2.2 Périodicité spatiale

Pour estimer la périodicité spatiale, nous devons respecter le critère de Shanon en ayant plus de deux mesures par période, c'est à dire

$$z_{i+1} - z_i < \frac{2\pi}{b}$$

2.1.3 Échantillonnage temporel

Nous supposons disposer de mesures à des temps consécutifs $t_1, \ldots, t_M$. Pour estimer la périodicité temporelle, il nous faut respecter encore une fois le critère de Shanon :

$$t_{i+1} - t_i < P$$

2.1.4 Exemple d'échantillonnage admissible

Pour $\theta_1 = 15°\text{C}$, $\theta_2 = 3°\text{C}$, $\lambda_m = 3\,\text{W}\cdot\text{m}^{-2}$, $c_m\rho_m = c_w\rho_w = 4\times10^6\,\text{J}\cdot\text{K}^{-1}\cdot\text{m}^{-3}$, $H_z = -\dfrac{0.05}{0.4}$ (cas infiltrant) et $\sigma = 0.1$ K, nous obtenons les figures 2 et 3.

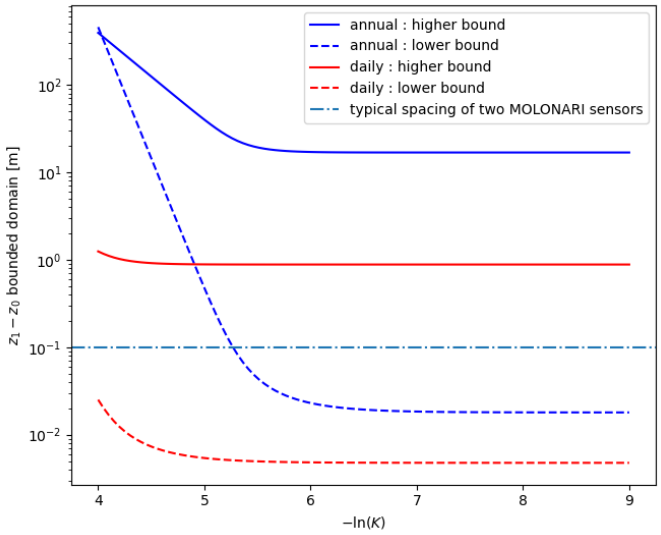

FIGURE 2 – $z_1 - z_0$ admissible

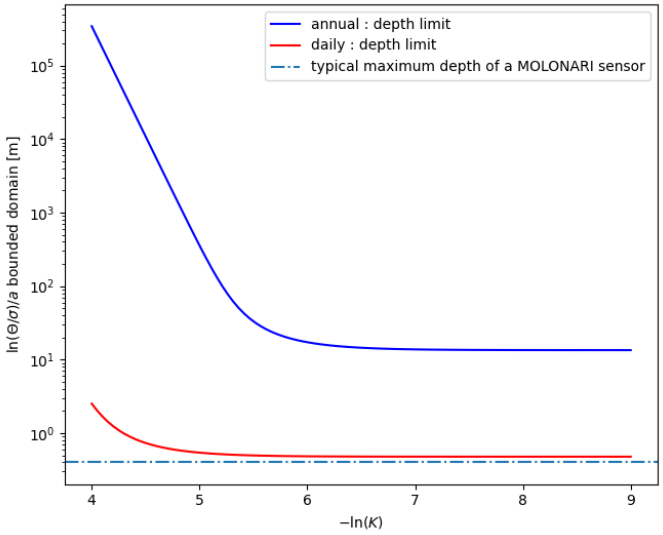

FIGURE 3 – profondeur limite z telle que $\theta e^{-az} = \sigma$

En particulier l'espacement typique de deux capteurs MOLONARI consécutifs vaut 10 cm et la profondeur maximale 40 cm. Pour les valeurs indiquées, un espacement des capteurs de 10 cm sur 40 cm paraît acceptable pour la contribution journalière, mais pour la contribution annuelle, une valeur de $-\ln(K)$ inférieure à 5 n'est a priori pas suffisante pour quantifier l'évolution de $z \mapsto \theta_1(z,t)$ sur 40 cm.

2.1.5 Temps de propagation sur une épaisseur

Un pic d'onde thermique sert de marqueur spatial de la perturbation et ainsi le temps que met le premier pic à atteindre la profondeur ℓ peut être considéré comme le temps de propagation de l'excitation à travers l'épaisseur de sol ℓ.

$$t_{\text{prop}} = \frac{bP}{2\pi}\ell$$

Si nous connaissons le temps caractéristique de variation des conditions limites, nous pouvons alors utiliser t_{prop} pour estimer le temps à partir duquel l'effet de ces conditions limites se sera propagé et aura modifié globalement le profil initial.

2.1.6 Lien entre (κ,D) et (a,b,P)

Si $z,t \mapsto e^{-az}\cos\left(\frac{2\pi}{P}t - bz\right)$ est solution de (1), alors $z,t \mapsto e^{-az}\sin\left(\frac{2\pi}{P}t - bz\right)$ l'est également, donc $z,t \mapsto \exp(-(a+ib)z+(2\pi i/P)t)$ est solution de (1), ce qui implique que $D(a+ib)^2-\kappa(a+ib) = i2\pi/P$. En séparant partie réelle et imaginaire, cette dernière équation se réécrit :

$$\begin{pmatrix} 2ab & -b \\ a^2-b^2 & -a \end{pmatrix}\begin{pmatrix} D \\ \kappa \end{pmatrix} = \begin{pmatrix} 2\pi/P \\ 0 \end{pmatrix}$$

Ce système admet toujours une solution lorsque a et b sont non-nuls car

$$\begin{vmatrix} 2ab & -b \\ a^2-b^2 & -a \end{vmatrix} = -2a^2b + ba^2 - b^3 = -b(a^2+b^2)$$

Finalement, la détermination de (a,b,P) est ici équivalente à la détermination de (κ,D).

2.2 Discrétisation des équations avec volumes finis

2.2.1 Discrétisation à partir des bilans de conservation

Nous découpons le milieu 1D en N+2 cellules :

• les cellules 0 et N+1 sont les cellules frontières auxquelles sont imposées les conditions limites.

• les cellules internes, indicées de 1 à N sont celles où s'opèrent les échanges advecto-conductifs que l'on cherche à simuler.

La discrétisation de l'équation de transfert de chaleur peut alors se mener à partir d'un bilan d'énergie échangée instantanément par une cellule avec ses voisines :

$$\frac{dU_n}{dt} = \Phi^c_{n-1,n} - \Phi^c_{n,n+1} + \Phi^a_{n-1,n} - \Phi^a_{n,n+1}$$

avec Φ^c un flux d'énergie de conduction et Φ^a un flux d'énergie d'advection. La double indexation par $(k-1,k)$ est utilisée pour une quantité échangée de la cellule $k-1$ vers la cellule k. De manière discrète, les lois de Fourier et de Darcy peuvent s'écrire

$$\Phi^c_{k,k+1} = -S_{k,k+1}\lambda_{k,k+1}\left(\frac{T_{k+1}-T_k}{\ell_{k,k+1}}\right)$$

$$\Phi^a_{k,k+1} = S_{k,k+1}q_{k,k+1}c_{k,k+1}T_{k,k+1}$$

donc

$$c_n\frac{dT_n}{dt} = -\lambda_{n-1,n}\left(\frac{T_n-T_{n-1}}{\ell_n\ell_{n-1,n}}\right) + \lambda_{n,n+1}\left(\frac{T_{n+1}-T_n}{\ell_n\ell_{n,n+1}}\right) + \left(\frac{q_{n-1,n}c_{n-1,n}T_{n-1,n} - q_{n,n+1}c_{n,n+1}T_{n,n+1}}{\ell_n}\right)$$

paramètre	sens physique	unité
$S_{k,k+1}$	la surface entre les cellules k et $k+1$	m^2
$\lambda_{k,k+1}$	une conductivité caractérisant la conduction entre k et $k+1$	$\mathrm{W}\cdot\mathrm{m}^{-1}\cdot\mathrm{K}^{-1}$
T_k et T_{k+1}	les températures des cellules k et $k+1$	K
$\ell_{k,k+1}$	la distance entre la cellule k et $k+1$	m
$q_{k,k+1}$	la vitesse de Darcy de l'eau de k vers $k+1$	$\mathrm{m}\cdot\mathrm{s}^{-1}$
$c_{k,k+1}$	la capacité volumique de l'eau échangée	$\mathrm{J}\cdot\mathrm{K}^{-1}\cdot\mathrm{m}^{-3}$
$T_{k,k+1}$	la température de l'eau échangée	K
c_n	la capacité volumique de la cellule n	$\mathrm{J}\cdot\mathrm{K}^{-1}\cdot\mathrm{m}^{-3}$
ℓ_n	l'épaisseur de la cellule n	m

2.2.2 Formulation tridiagonale

Repérer une cellule dans l'espace à partir de son centre amène à poser $\ell_{n,n+1} = \dfrac{\ell_n + \ell_{n+1}}{2}$.

Par soucis de symétrie d'un échange local, nous postulons que $T_{k,k+1}$ est une application symétrique des grandeurs de la cellule k et $k+1$. Par exemple,

$$T_{k,k+1} = \frac{\ell_{k+1}T_k + \ell_k T_{k+1}}{2\ell_{k,k+1}}$$

traduit bien le fait que si $\ell_k \ll \ell_{k+1}$ alors l'eau échangée à l'interface des cellules k et $k+1$ est de température proche de T_k. Dans ce cas, lorsque les cellules ont les mêmes dimensions, $T_{k,k+1} = \frac{T_k+T_{k+1}}{2}$ et en prenant des paramètres advecto-conductifs constants, nous obtenons l'expression classique en différences finies :

$$c_m\rho_m\frac{dT_n}{dt} = \left(\frac{\lambda_m}{\ell^2} + \frac{qc_w\rho_w}{2\ell}\right)T_{n-1} - \frac{2\lambda_m}{\ell^2}T_n + \left(\frac{\lambda_m}{\ell^2} - \frac{qc_w\rho_w}{2\ell}\right)T_{n+1}$$

De manière générale, il paraît pertinent de proposer une expression de $T_{k,k+1}$ comme barycentre de T_k et T_{k+1} afin d'assurer que $T_{k,k+1} \in [T_k, T_{k+1}]$. Le choix de $T_{k,k+1} = \dfrac{\ell_{k+1}T_k + \ell_k T_{k+1}}{2\ell_{k,k+1}}$ amène à

$$\begin{aligned}
\frac{dT_n}{dt} &= \frac{1}{c_n\rho_n\ell_n}\left(\frac{\lambda_{n-1,n}}{\ell_{n-1,n}} + q_{n-1,n}c_w\rho_w\frac{\ell_n}{2\ell_{n-1,n}}\right)T_{n-1} \\
&\quad -\frac{1}{c_n\rho_n\ell_n}\left(\frac{\lambda_{n-1,n}}{\ell_{n-1,n}} + \frac{\lambda_{n,n+1}}{\ell_{n,n+1}} - c_w\rho_w\left[q_{n-1,n}\frac{\ell_{n-1}}{2\ell_{n-1,n}} - q_{n,n+1}\frac{\ell_{n+1}}{2\ell_{n,n+1}}\right]\right)T_n \\
&\quad +\frac{1}{c_n\rho_n\ell_n}\left(\frac{\lambda_{n,n+1}}{\ell_{n,n+1}} - q_{n,n+1}c_w\rho_w\frac{\ell_n}{2\ell_{n,n+1}}\right)T_{n+1}
\end{aligned}$$

Ainsi, en notant $T = \begin{pmatrix} T_1 \\ \vdots \\ T_N \end{pmatrix}$, il vient que

$$\dot{T} = AT + B$$

avec $A = \begin{pmatrix} b_1 & c_1 & & & \\ a_2 & \ddots & \ddots & & \\ & \ddots & \ddots & \ddots & \\ & & \ddots & \ddots & c_{N-1} \\ & & & a_N & b_N \end{pmatrix} = \text{Tridiag}(a_n, b_n, c_n)$, $B = \begin{pmatrix} a_1T_0 \\ 0 \\ \vdots \\ 0 \\ c_NT_{N+1} \end{pmatrix}$ et

$$a_n = \frac{1}{c_n\rho_n\ell_n}\left(\frac{\lambda_{n-1,n}}{\ell_{n-1,n}} + q_{n-1,n}c_w\rho_w\frac{\ell_n}{2\ell_{n-1,n}}\right)$$

$$b_n = -\frac{1}{c_n\rho_n\ell_n}\left(\frac{\lambda_{n-1,n}}{\ell_{n-1,n}} + \frac{\lambda_{n,n+1}}{\ell_{n,n+1}} - c_w\rho_w\left[q_{n-1,n}\frac{\ell_{n-1}}{2\ell_{n-1,n}} - q_{n,n+1}\frac{\ell_{n+1}}{2\ell_{n,n+1}}\right]\right)$$

$$c_n = \frac{1}{c_n\rho_n\ell_n}\left(\frac{\lambda_{n,n+1}}{\ell_{n,n+1}} - q_{n,n+1}c_w\rho_w\frac{\ell_n}{2\ell_{n,n+1}}\right)$$

2.2.3 Couplage avec l'équation hydraulique

Les équations (1) et (2) ayant une structure identique, nous pouvons employer la même discrétisation pour la charge hydraulique et pour les températures, tout en veillant à ce que :

- $c_n\rho_n \leftrightarrow S_n$
- $\lambda_{n,n+1} \leftrightarrow K_{n,n+1}$
- $q_{n,n+1} = 0$ car il n'y a pas d'advection hydraulique

En posant $q_{n,n+1} = -K_{n,n+1}\dfrac{H_{n+1} - H_n}{\ell_{n,n+1}}$ nous obtenons deux équations couplées en T et H.

2.2.4 Décomposition U - V

Afin d'isoler la dépendance en H dans A_θ et B_θ ; nous pouvons écrire
$A_\theta = U_\theta + V_\theta$ avec $U_\theta = \text{Tridiag}(u_n^{(1)}, u_n^{(2)}, u_n^{(3)})$ et $V_\theta = \text{Tridiag}(v_n^{(1)}, v_n^{(2)}, v_n^{(3)})$ telles que

$$u_n^{(1)} = \frac{1}{c_n\rho_n\ell_n}\left(\frac{\lambda_{n-1,n}}{\ell_{n-1,n}}\right),\ u_n^{(2)} = -\frac{1}{c_n\rho_n\ell_n}\left(\frac{\lambda_{n-1,n}}{\ell_{n-1,n}} + \frac{\lambda_{n,n+1}}{\ell_{n,n+1}}\right) \text{ et } u_n^{(3)} = \frac{1}{c_n\rho_n\ell_n}\left(\frac{\lambda_{n,n+1}}{\ell_{n,n+1}}\right)$$

$$v_n^{(1)} = \frac{1}{c_n\rho_n\ell_n}\left(q_{n-1,n}c_w\rho_w\frac{\ell_n}{2\ell_{n-1,n}}\right)$$

$$v_n^{(2)} = \frac{1}{c_n\rho_n\ell_n}\left(c_w\rho_w\left[q_{n-1,n}\frac{\ell_{n-1}}{2\ell_{n-1,n}} - q_{n,n+1}\frac{\ell_{n+1}}{2\ell_{n,n+1}}\right]\right)$$

$$v_n^{(3)} = -\frac{1}{c_n\rho_n\ell_n}\left(q_{n,n+1}c_w\rho_w\frac{\ell_n}{2\ell_{n,n+1}}\right)$$

Les coefficients u et v admettent des expressions analogues, de la forme :

$$x_n^{(1)} = w_n j_n^{(1)} d_{n-1,n}e_{n-1,n}$$
$$x_n^{(2)} = \pm w_n\left(j_n^{(2-)}d_{n-1,n}e_{n-1,n} \pm j_n^{(2+)}d_{n,n+1}e_{n,n+1}\right)$$
$$x_n^{(3)} = \pm w_n j_n^{(3)} d_{n,n+1}e_{n,n+1}$$

A U et V nous pouvons associer des conditions aux limites, de la forme $B_U = u_1^{(1)}T_0e_1 + u_N^{(3)}T_Ne_N$.

2.2.5 Récapitulatif pratique (facilitant le recours massif au slicing)

	w_n	$d_{n,n+1}$	$e_{n,n+1}$	$j_n^{(1)}$	$j_n^{(2-)}$	$j_n^{(2+)}$	$j_n^{(3)}$
U_θ	$1/c_n\rho_n\ell_n$	$1/\ell_{n,n+1}$	$\lambda_{n,n+1}$	1	1	1	1
V_θ	$1/c_n\rho_n\ell_n$	$c_w\rho_w/2\ell_{n,n+1}$	$q_{n,n+1}$	ℓ_n	ℓ_{n-1}	ℓ_{n+1}	ℓ_n
U_H	$1/s_n\ell_n$	$1/\ell_{n,n+1}$	$K_{n,n+1}$	1	1	1	1
V_H	0	0	0	0	0	0	0

vecteurs	W	D	E	J
dimension	N	N+1	N+1	N+2
premier élément	w_1	$d_{0,1}$	$e_{0,1}$	j_0
dernier élément	w_N	$d_{N,N+1}$	$e_{N,N+1}$	j_{N+1}

En utilisant $\bullet$ le produit d'Hadamard et les notations Numpy de slicing :

$$U = W \bullet \text{Tridiag}(u^1, u^2, u^3) \quad \text{et} \quad V = W \bullet \text{Tridiag}(v^1, v^2, v^3)$$

avec

	$u^{(i)}$	$v^{(i)}$
1	$(D \bullet E)[1:N]$	$J[2:N+1] \bullet (D \bullet E)[1:N]$
2	$-\left((D \bullet E)[:N] + (D \bullet E)[1:]\right)$	$J[:N] \bullet (D \bullet E)[:N] - J[2:] \bullet (D \bullet E)[1:]$
3	$(D \bullet E)[1:N]$	$-J[1:N] \bullet (D \bullet E)[1:N]$

Enfin, le couplage thermo-hydraulique s'écrit

$$E_H = -K \bullet D \bullet (\tilde{H}[1:] - \tilde{H}[:-1])$$

avec $\tilde{H} = \begin{pmatrix} Hb[0] \\ H \\ Hb[1] \end{pmatrix}$ où les termes aux extrémités correspondent aux conditions limites.

Remarque : j'ai utilisé le slicing dans les premières implémentations python, mais il s'avère peu approprié pour les objets bas niveau en cython (par exemple les arrays créés avec malloc) dont l'atout principal est de permettre des opérations élémentaires plus rapides.

2.3 Résolution numérique

La discrétisation de l'équation amène à

$$\dot{Y} = AY + B$$

avec $A = U + V$ tridiagonale et B ne pouvant être non-nul qu'en ses extrémités.

En particulier,

$$\begin{array}{lrcl} (1) & \dot{T} & = & (U_T + V_T)T + B_T \\ (2) & \dot{H} & = & U_H H + B_H \end{array}$$

2.3.1 Résolutions séparées

Une première méthode consiste à résoudre (2) en stockant $(H(t_i))_{1 \leqslant i \leqslant N}$ afin ensuite de résoudre (1) aux temps (t_i). Cette approche comporte au moins quatre inconvénients :

• nous utilisons de l'espace mémoire en stockant les $H(t_i)$.

• c'est plus long de calculer T puis H, plutôt que de le faire simultanément.

• lors d'un calcul intermédiaire à $t_{i,i+1}$ entre t_i et t_{i+1} pour (1), nous ne connaissons pas $H(t_{i,i+1})$.

• nous allons faire deux appels au solveur pour (2) et (1), ce qui implique de la redondance ainsi qu'une difficulté à paramétrer correctement les deux appels.

2.3.2 Résolutions simultanées

Afin de ne pas avoir les défauts précédents, une idée est de rassembler (1) et (2) en une équation, par exemple en construisant un nouveau vecteur $X = \begin{pmatrix} T \\ H \end{pmatrix}$ solution de $\dot{X} = f(t, X)$.

2.3.3 Tests numériques

Nous considérons des conditions aux limites constantes. Asymptotiquement, nous devons donc retrouver des solutions de régimes permanents.

2.3.3.1 Milieu homogène

$H_\infty = \dfrac{H(\ell) - H(0)}{\ell} z + H(0)$
$T_\infty = \exp(-\gamma z)\,\alpha + \beta$ avec :
$\circ$ $\gamma = \dfrac{\rho_w c_w}{\lambda_m} \dfrac{K(H(\ell) - H(0))}{\ell}$
$\circ$ $\begin{pmatrix} 1 & 1 \\ \exp(\gamma h) & 1 \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \begin{pmatrix} \theta(0) \\ \theta(\ell) \end{pmatrix}$ donc $\alpha = \dfrac{\theta(0) - \theta(\ell)}{1 - \exp(\gamma\ell)}$ et $\beta = \dfrac{\theta(\ell) - \exp(\gamma\ell)\theta(0)}{1 - \exp(\gamma\ell)}$

2.3.3.2 Milieu hétérogène

Supposons que $K \mapsto K_{z,0} z + K_0$, les autres grandeurs en jeu restant constantes.

$$H_\infty = (H(\ell) - H(0)) \frac{\int_0^z \dfrac{1}{K_{z,0} z + K_0} dz}{\int_0^\ell \dfrac{1}{K_{z,0} z + K_0} dz} + H(0) = (H(\ell) - H(0)) \frac{\ln(K_{z,0} z + K_0) - \ln(K_0)}{\ln(K_{z,0}\ell + K_0) - \ln(K_0)} + H(0)$$

$$\theta_\infty = (\theta(\ell) - \theta(0)) \frac{\int_0^z \exp\left(-\frac{c_w \rho_w K H_{z,0}}{\lambda_m} x\right) dx}{\int_0^\ell \exp\left(-\frac{c_w \rho_w K H_{z,0}}{\lambda_m} x\right) dx} + \theta(0)$$

Avec $K_{z,0} = 10^{-5}$ et $K_0 = 10^{-5}$ m, $\ell = 0.4$ m, $\lambda_m = 3\,\mathrm{W} \cdot \mathrm{m}^{-2}$, $c_m \rho_m = c_w \rho_w = 4 \times 10^6\,\mathrm{J} \cdot \mathrm{K}^{-1} \cdot \mathrm{m}^{-3}$, nous obtenons un résultat satisfaisant :

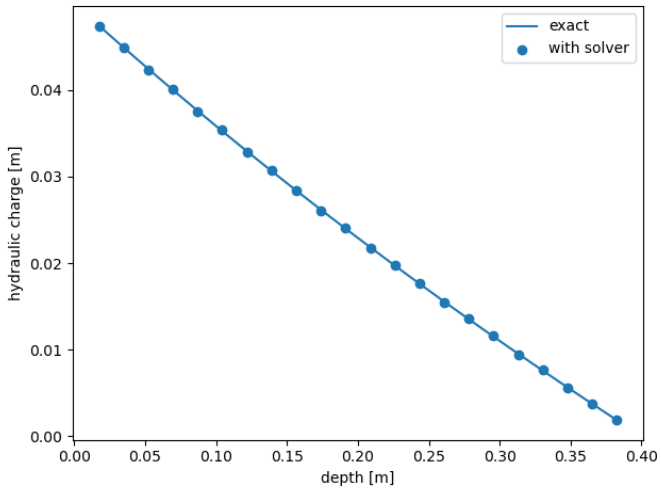

FIGURE 4 – charge hydraulique

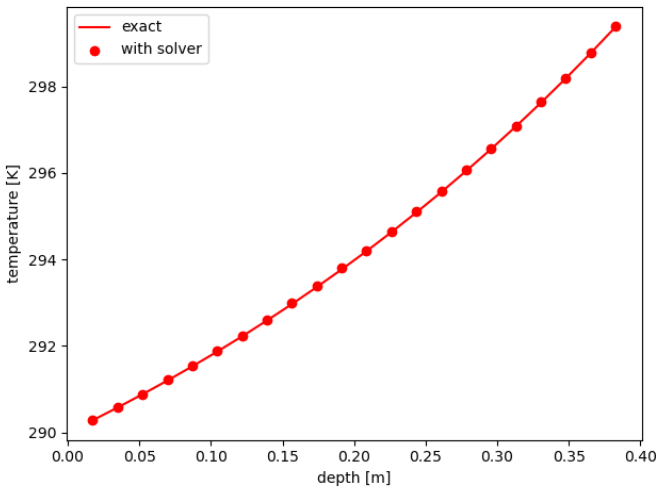

FIGURE 5 – température

2.4 Conditions limites hydrauliques

Les conditions limites en charge hydraulique sont nécessaires à la résolution de l'équation (2), ce qui permet alors d'estimer H_z intervenant dans l'équation (1). Or nous ne disposons actuellement que de la différence entre les charges hydrauliques $H(\ell)$ et $H(0)$ aux extrémités de la colonne de sol (dispositif décrit dans [1]). Il y a alors deux voies possibles : construire des conditions limites hydrauliques acceptables pour résoudre numériquement (2) ou bien estimer directement H_z à partir de ΔH.

2.4.1 Méthode de régression pour l'estimation de H_z

Afin d'estimer H_z à partir de ΔH, il faut usuellement connaître une forme analytique de $z \mapsto H_z$ paramétrée par ΔH. Ce cas de figure est réalisé en régime quasi-permanent où $H = \beta \int_0^z \frac{1}{K(u)} du + H_0$ donc $H_z = \frac{\beta}{K}$ avec β pouvant être fixé par $\Delta H = \beta \int_0^\ell \frac{1}{K(u)} du$.

Entre deux temps d'observations consécutifs t_1 et t_2, il reste à interpoler temporellement les profils $z \mapsto H_z(z, t_1)$ et $z \mapsto H_z(z, t_1)$. Par exemple :

$$H_z(z, \epsilon t_1 + (1-\epsilon) t_0) = H_z(z, t_0)\epsilon + (1-\epsilon) H_z(z, t_1), \qquad \epsilon \in [0, 1]$$

Un test de régime quasi-permanent consiste à comparer le temps d'évolution caractéristique des conditions aux limites au temps de propagation d'une onde de charge hydraulique dans le milieu, ce qui peut se faire en calculant

$$\frac{t_{\text{prop}}}{P} = \frac{1}{2\sqrt{\pi}} \sqrt{\frac{S}{PK}} \cdot \ell$$

Prenons comme valeurs de référence, $\ell = 60$ cm, $S = 0.2$, et t_{prop}/P en pourcentage :

t_{prop}/P	P	1 heure	1 jour	1 mois
$-\log(K)$				
3		3%	0.8 %	0.1%
4		13%	3 %	0.4%
5		40%	8 %	1%
6		126%	26 %	5%

2.4.2 Méthode d'obtention des conditions limites

Il est nécessaire de connaître les conditions limites $t \mapsto H(0,t)$ et $t \mapsto H(\ell,t)$ afin de pouvoir faire des prédictions via l'équation (2). Une connaissance de $t \mapsto \Delta H(t)$ n'est pas suffisante dans le cas général car nous avons

$$H(\ell,t) = H(0,t) + \Delta H(t)$$

et sans information autre que ΔH nous ne pouvons pas obtenir $H(0,t)$.

Considérons l'exemple ci-dessous pour comprendre que la connaissance de $t \mapsto \Delta H(t)$ ne suffit pas à prévoir l'évolution de H et donc de H_z. Plutôt que d'utiliser H, nous parlerons de température puisque les deux grandeurs suivent la même équation de diffusion.

Une tige en métal est coincée entre deux thermostats.
Initialement, les deux thermostats et la tige sont tous à T_0.
Nous réglons simultanément la température des deux thermostats à T_1.
Le champ des températures dans la tige évolue alors vers le nouvel équilibre et l'on observe transitoirement de forts gradients thermiques.

Dans cet exemple, à tout moment nous avons $\Delta T = 0$ aux extrémités de la tige, pour autant avec cette seule information il nous est impossible de dire si le métal reste à l'équilibre thermique ou bien si ce dernier subit un choc thermique.

Nous savons que la charge hydraulique de l'aquifère en profondeur est quasi-constante (du moins sur une période d'observation suffisamment courte). En supposant qu'à la profondeur ℓ, nous bénéficions déjà d'une charge hydraulique stable durant la période de mesure considérée, il est alors possible de poser $H(\ell,t) = 0$ et $H(0,t) = -\Delta H$ pour obtenir des conditions limites pertinentes. Cette hypothèse permettant de proposer des conditions limites à partir de $t \mapsto \Delta H(t)$ n'est pas toujours vérifiée, comme le montre le cas simple d'un milieu homogène parcouru par une onde de charge hydraulique de période temporelle P. L'amplitude amortie de l'onde de charge hydraulique à la profondeur z vaut

$$A(z) = \exp\left(-\sqrt{\frac{\pi S}{KP}}z\right)$$

En calculant cette amplitude en pourcentage de l'amplitude initiale (en $z = 0$), nous en déduisons

$A(\ell)$	P	1 heure	1 jour	1 mois
$-\log(K)$				
3		78%	95 %	99%
4		45%	85 %	97%
5		8%	60 %	91%
6		0.04%	20 %	74%

Sur les 3 premiers ordres de grandeur de K, la charge hydraulique en $z = \ell$ oscille avec une amplitude significative pour les 3 périodes P considérées et donc l'hypothèse de charge hydraulique constante en $z = \ell$ n'est pas satisfaisante.

Nous remarquons ici que les pires cas sont globalement ceux où l'hypothèse de régime quasi-permanent est valable : ceci pourrait nous amener à utiliser l'une ou l'autre méthode en fonction des conditions les plus favorables.

3 Implémentation : MCMC avec modèle direct

3.1 Choix de cython

L'implémentation la plus avancée est réalisée en cython qui est une extension permettant de compiler un code hybride mélangeant python et C. Les atouts de cython sont ses performances (le code est compilé en C), sa maturité (contrairement à numba), la richesse des outils disponibles (librairies C ou C++, objects cython comme les memoryviews ou les extended classes) ou encore sa facilité à créer des modules appelables depuis des scripts python classiques. Voici quelques tests de vitesse

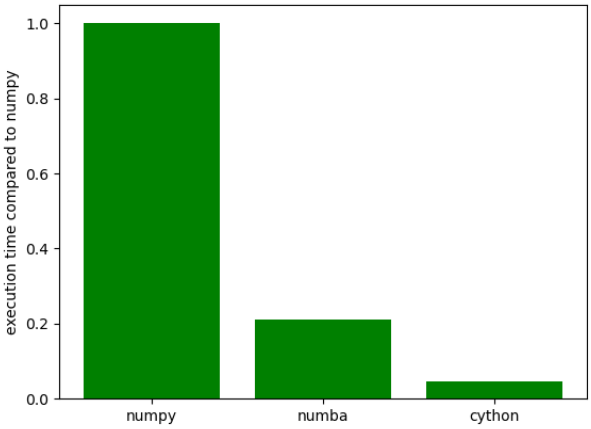

FIGURE 6 – création d'un tableau vide

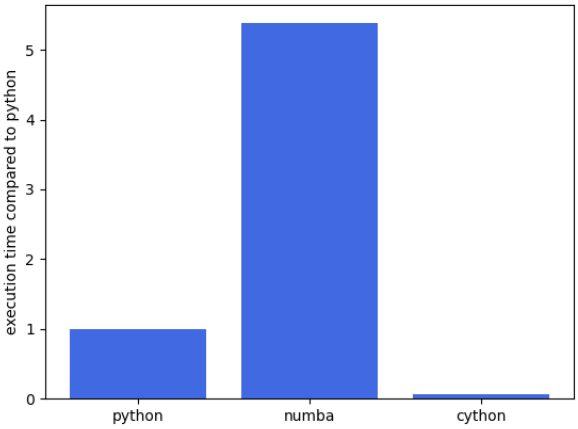

FIGURE 7 – création d'une classe

Afin d'accroître la vitesse d'exécution, il est conseillé avec cython de systématiquement privilégier les instructions C de base : typiquement malloc pour créer un array (voir [4]). Ce gain en rapidité a pour principal désagrément de nécessiter davantage de lignes de code : par exemple l'allocation manuelle de la mémoire avec malloc suppose une dés-allocation manuelle avec free, alors qu'en python cette dés-allocation est gérée automatiquement par le garbage collector.

3.2 Test en régime permanent

Pour un milieu homogène avec :

- une perméabilité de $10^{-5}\,\mathrm{m}^2$
- une résolution des équations (1) et (2) sur une semaine
- 5 chaînes MCMC
- 1000 itérations

Nous obtenons au bout de 3.1 secondes un estimation $\overline{K} = 1.7 \times 10^{-5}\,\mathrm{m}^2$ ainsi que l'histogramme ci-dessous. En particulier, en prenant comme option autostep=False, c'est à dire en désactivant la méthode de pas de temps adaptatif du solver (voir [5]), le temps d'exécution est multiplié par 10.

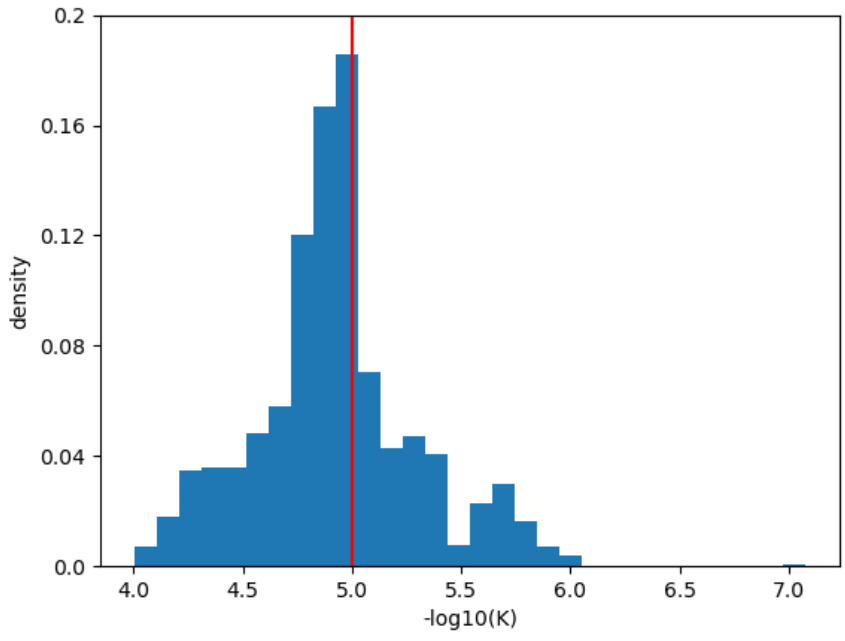

FIGURE 8 – histogramme de la perméabilité logarithmique

remarque : je n'ai pas utilisé un a priori uniforme sur $-\log(K)$ mais sur K dans la MCMC, car j'estime que ce dernier a priori est plus pertinent physiquement parlant.

3.3 Conclusion

Dans ce projet, nous nous sommes intéressé à l'estimation des paramètres d'une EDP en 1D à partir de mesures physiques d'une solution. Les mesures physiques parcellaires dont nous disposons nous ont fait privilégier une approche bayésienne de quantification de la vraisemblance des jeux de paramètres par rapport à un modèle physique et à des densités a priori. Plus précisément, nous avons utilisé une méthode classique de MCMC (Markov Chain Monte Carlo) où une marche aléatoire permet une exploration intelligente de l'espace des paramètres a priori. Le recourt à cette méthode a exigé l'élaboration d'un outil de résolution numérique d'EDP, obtenu à partir d'une interprétation discrète des lois physiques en jeu (∗). C'est en cherchant à améliorer les performances d'une implémentation python des algorithmes de MCMC et de résolution d'EDP que nous avons opté pour cython après avoir également testé numpy et numba.

D'un point de vue de l'implémentation, plusieurs progrès sont envisageables :
◦ tester de manière plus approfondie l'implémentation cython : typiquement avec des ondes thermiques et en milieu multi-strates. De manière plus générale, il faudrait créer une banque de tests de référence.
◦ créer une classe (ou fonction) permettant d'interfacer de manière générique les outils cython principaux (construits avec des instructions C) avec du python (pour rappel : les objets C définis avec cython, reconnaissables par leur déclaration utilisant "cdef", ne sont pas appelables depuis un script python).
◦ implémenter des fonctionnalités supplémentaires comme l'utilisation du critère de Gelman-Rubin ou le calcul des quantiles. Afin d'accélérer le calcul des quantiles, nous pourrions nous limiter à l'estimation des températures moyennes ainsi que d'un écart-type moyen.

Pour finir, voici quelques pistes de réflexion :
◇ faut-il revoir les dispositifs de mesure pour améliorer l'efficacité des algorithmes ? Par exemple, les fréquences d'échantillonnage (spatiale et temporelle) et les capteurs de charges hydrauliques.
◇ comment spécialiser la méthode MCMC pour augmenter les performances de résolution ?
◇ est-il intéressant d'utiliser une méthode autre que la MCMC ?
◇ comment estimer la validité d'un modèle physique 1D avec des strates horizontales ?
◇ quels gains peut-on obtenir en complexifiant le modèle physique (ex : prendre en compte des dépendances en température ou poser le problème en 2D voire 3D) ?

(∗) la méthode utilisée ici est fortement inspirée de ce que j'ai créé l'année dernière durant un projet de résolution numérique de l'équation de la chaleur non-linéaire 1D, encadré par Laurent Lacourt (ingénieur chez Framatome et professeur aux Mines de Paris).

Références

[1] Karina Cucchi, Agnès Rivière, Aurélien Baudin, Asma Berrhouma, Véronique Durand, Fayçal Rejiba, Yoram Rubin, Nicolas Flipo, LOMOS-mini : A coupled system quantifying transient water and heat exchanges in streambeds, Journal of Hydrology, 2018

[2] Jasper A. Vrugt, Markov chain Monte Carlo simulation using the DREAM software package : Theory, concepts, and MATLAB implementation, Environmental Modelling & Software, 2016

[3] Stallman, Robert William, Steady one-dimensional fluid flow in a semi-infinite porous medium with sinusoidal surface temperature, Journal of Geophysical Research 70, 1965

[4] What is the recommended way of allocating memory for a typed memory view ?, https ://stackoverflow.com/questions/18462785/what-is-the-recommended-way-of-allocating-memory-for-a-typed-memory-view

[5] Practical Error Estimation and Step Size Selection - EPFL, https ://www.epfl.ch/labs/anchp/wp-content/uploads/2018/05/HNW.pdf

4 Codes en annexe

4.1 cython : programmes avancés

4.1.1 code du module cython à compiler (MOLO_45.pyx)

```
import numpy as np
from libc.stdlib cimport malloc, free
from libc.stdlib cimport rand, srand, RAND_MAX
from libc.math cimport log, sqrt, exp
from libc.time cimport time

##Matrix objects and operations

#creates a malloc array from a memoryview
cdef double* numpy_to_C(double[:] X):
    cdef int size = X.shape[0]
    cdef double* Xr = <double*> malloc(size * sizeof(double))
    cdef int i
    for i in range(size):
        Xr[i] = X[i]
    return Xr

#creates a memoryview from a malloc array
cdef double[:] C_to_numpy(double* X, int size):
    cdef double[:] Xr = np.empty(size,dtype=np.double)
    cdef int i
    for i in range(size):
        Xr[i] = X[i]
    return Xr

#returns coefficient (i,j) of a triadiagonal matrix stored as a malloc array
cdef double tridget(double* coefs, int size, int i, int j):
    cdef int select = i-j
    #coef from the subdiagonal
    if select>0:
        return coefs[i-1]
    #coef from the diagonal
    elif select<0:
        return coefs[2*size-1+i]
    #coef from the superdiagonal
    else:
        return coefs[size-1+i]

#to build an ordinary (n,m) real matrix
cdef class Matrix:

    #matrix stored in a malloc array
    cdef double* mat
    #number of lines
    cdef int n
    #number of columns
    cdef int m

    cdef void init(self,int n, int m):
        self.n = n
        self.m = m
        self.mat = <double*> malloc((n*m) * sizeof(double))

    cdef void set(self, int i, int j, double val):
        self.mat[i*self.m+j] = val

    cdef double get(self, int i, int j):
        return self.mat[i*self.m+j]

    #assigns values of a contiguous row slice
```

```
    cdef void slicerowset(self,int i, int j1, int j2, double* row):
        cdef int j
        for j in range(j1,j2):
            self.set(i,j,row[j-j1])

    def convert_to_numpy(self):
        A = np.empty((self.n,self.m),dtype=np.double)
        cdef double[:, ::1] A_view = A
        cdef int i
        for i in range(self.n):
            for j in range(self.m):
                A_view[i,j] = self.get(i,j)
        return np.asarray(A)

    def extract_from_numpy(self, double[:, ::1] A):
        cdef int i
        for i in range(self.n):
            for j in range(self.m):
                self.set(i,j, A[i,j])

    cdef void free(self):
        free(self.mat)

#to build a tridiagonal (n,n) real matrix
cdef class Tridiag:

    #matrix stored in a malloc array
    cdef double* mat
    #number of lines or rows
    cdef int size

    cdef void init(self,int size):
        self.size = size
        self.mat = <double*> malloc((3*size-2) * sizeof(double))

    cdef void set(self,int i, int j, double val):
        cdef int select = i-j
        #subdiagonal
        if select>0:
            self.mat[i-1] = val
        #diagonal
        elif select<0:
            self.mat[2*self.size-1+i] = val
        #superdiagonal
        else:
            self.mat[self.size-1+i] = val

    #assigns identity matrix coef values
    cdef void identity(self):

        #first row
        self.set(0,0,1)
        self.set(0,1,0)

        cdef int i
        for i in range(1,self.size-1):
            self.set(i,i-1,0)
            self.set(i,i,1)
            self.set(i,i+1,0)

        #last row
        self.set(self.size-1,self.size-2,0)
        self.set(self.size-1,self.size-1,1)

    #assigns three given diagonals
    cdef void diagset(self,double* sub_diag, double* diag, double* sup_diag):

        #first row
```

```
        self.set(0,0,diag[0])
        self.set(0,1,sup_diag[0])

        cdef int i
        for i in range(1,self.size-1):
            self.set(i,i-1,sub_diag[i-1])
            self.set(i,i,diag[i])
            self.set(i,i+1,sup_diag[i])

        #last row
        self.set(self.size-1,self.size-2,sub_diag[self.size-2])
        self.set(self.size-1,self.size-1,diag[self.size-1])

    cdef double get(self,int i, int j):
        cdef int select = i-j
        #subdiagonal
        if select>0:
            return self.mat[i-1]
        #diagonal
        elif select<0:
            return self.mat[2*self.size-1+i]
        #superdiagonal
        else:
            return self.mat[self.size-1+i]

    #multiplies the matrix with a scalar
    cdef void scalarmul(self,double a):

        #first row
        self.set(0,0,a*self.get(0,0))
        self.set(0,1,a*self.get(0,1))

        cdef int i
        for i in range(1,self.size-1):
            self.set(i,i-1,a*self.get(i,i-1))
            self.set(i,i,a*self.get(i,i))
            self.set(i,i+1,a*self.get(i,i+1))

        #last row
        self.set(self.size-1,self.size-2,a*self.get(self.size-1,self.size-2))
        self.set(self.size-1,self.size-1,a*self.get(self.size-1,self.size-1))

    #returns the image of a vector
    #WARNING: it returns a new malloc array
    cdef double* dot(self,double* X):

        cdef double* Xr = <double*> malloc(self.size * sizeof(double))

        #first coef
        Xr[0] = self.get(0,0)*X[0] + self.get(0,1)*X[1]

        cdef int i
        for i in range(1,self.size-1):
            Xr[i] = self.get(i,i-1)*X[i-1] + self.get(i,i)*X[i] + self.get(i,i+1)*X[i
    +1]

        #last coef
        Xr[self.size-1] = self.get(self.size-1,self.size-2)*X[self.size-2] + self.get
    (self.size-1,self.size-1)*X[self.size-1]

        return Xr

    #copies the tridiagonal matrix coef values given as an array
    cdef void copy(self,double* A):

        #first row
        self.set(0,0, tridget(A,self.size,0,0))
        self.set(0,1, tridget(A,self.size,0,1))
```

```
195
196         cdef int i
197         for i in range(1,self.size-1):
198             self.set(i,i-1,tridget(A,self.size,i,i-1))
199             self.set(i,i,tridget(A,self.size,i,i))
200             self.set(i,i+1,tridget(A,self.size,i,i+1))
201
202         #last row
203         self.set(self.size-1,self.size-2,tridget(A,self.size,self.size-1,self.size-2)
    )
204         self.set(self.size-1,self.size-1,tridget(A,self.size,self.size-1,self.size-1)
    )
205
206     #assigns the sum of two tridiagonal matrices given as arrays
207     cdef void add(self,double* A, double* B):
208
209         #first row
210         self.set(0,0, tridget(A,self.size,0,0) + tridget(B,self.size,0,0))
211         self.set(0,1, tridget(A,self.size,0,1) + tridget(B,self.size,0,1))
212
213         cdef int i
214         for i in range(1,self.size-1):
215             self.set(i,i-1,tridget(A,self.size,i,i-1) + tridget(B,self.size,i,i-1))
216             self.set(i,i,tridget(A,self.size,i,i) + tridget(B,self.size,i,i))
217             self.set(i,i+1,tridget(A,self.size,i,i+1) + tridget(B,self.size,i,i+1))
218
219         #last row
220         self.set(self.size-1,self.size-2,tridget(A,self.size,self.size-1,self.size-2)
     + tridget(B,self.size,self.size-1,self.size-2))
221         self.set(self.size-1,self.size-1,tridget(A,self.size,self.size-1,self.size-1)
     + tridget(B,self.size,self.size-1,self.size-1))
222
223     #solves the linear system A*x = d
224     #uses Thomas algorithm
225     #WARNING: the solution is returned as a new malloc array
226     cdef double* solve(self,double* d):
227
228         cdef double* cp = <double*> malloc((self.size-1) * sizeof(double))
229         cdef double* dp = <double*> malloc(self.size * sizeof(double))
230         cdef double* s = <double*> malloc(self.size * sizeof(double))
231         cdef int i
232
233         cp[0] = self.get(0,1)*self.get(0,0)**(-1)
234         for i in range(1,self.size-1):
235             cp[i] = self.get(i,i+1)*(self.get(i,i)-self.get(i+1,i)*cp[i-1])**(-1)
236
237         dp[0] = d[0]*self.get(0,0)**(-1)
238         for i in range(1,self.size):
239             dp[i] = (d[i]-self.get(i,i-1)*dp[i-1])*(self.get(i,i)-self.get(i,i-1)*cp[
    i-1])**(-1)
240
241         s[self.size-1] = dp[self.size-1]
242         for i in range(1,self.size):
243             s[self.size-1-i] = dp[self.size-1-i] - cp[self.size-1-i]*s[self.size-i]
244
245         free(cp)
246         free(dp)
247
248         return s
249
250     cdef void free(self):
251         free(self.mat)
252
253     def convert_to_numpy(self):
254         A = np.zeros((self.size,self.size),dtype=np.double)
255         cdef double[:, ::1] A_view = A
256         cdef int i
257         A_view[0,0] = self.get(0,0)
```

```
        A_view[0,1] = self.get(0,1)
        for i in range(1,self.size-1):
            A_view[i,i-1] = self.get(i,i-1)
            A_view[i,i] = self.get(i,i)
            A_view[i,i+1] = self.get(i,i+1)
        A_view[self.size-1,self.size-2] = self.get(self.size-1,self.size-2)
        A_view[self.size-1,self.size-1] = self.get(self.size-1,self.size-1)
        return np.asarray(A)

##MOLONARI objects

###Physical model

#to build the discretized spatial structure of the studied ground layer
cdef class Geometry:
    #number of ground cells (or sublayers)
    cdef int N
    #height of the studied ground layer [m]
    cdef double h
    #cells widths [m]
    cdef double* width
    #spacings between consecutive cells [m]
    cdef double* spacing
    #cells interfaces depths [m]
    cdef double* depth_i
    #cells depths [m]
    cdef double* depth_c

    cdef void init(self,int N):
        self.N = N
        self.width = <double*> malloc((N+2) * sizeof(double))
        self.spacing = <double*> malloc((N+1) * sizeof(double))
        self.depth_c = <double*> malloc(N * sizeof(double))
        self.depth_i = <double*> malloc((N+1) * sizeof(double))

    #creates a regular spatial structure
    cdef void regular(self, double h):

        self.h = h
        cdef double l = h*(<double> self.N + 1)**(-1)

        cdef int i
        for i in range(self.N):
            self.width[i] = l
            self.spacing[i] = l
            self.depth_c[i] = (i+1)*l
            self.depth_i[i] = i*l+0.5*l

        self.width[self.N] = l
        self.spacing[self.N] = l
        self.depth_i[self.N] = h
        self.width[self.N+1] = l

    cdef void free(self):
        free(self.width)
        free(self.spacing)
        free(self.depth_c)
        free(self.depth_i)

#to build a conduction matrix for discretized equations
cdef class Umatrix:
    #number of main cells
    cdef int N
    cdef Tridiag Umat
    #auxiliary array used in boundary conditions
    cdef double* B_U
    cdef Geometry geo
    #conductivities
```

```
    cdef double* L
    #capacities
    cdef double* C

    cdef void init(self, int N, Geometry geo, double* Lambda, double* C):
        self.N = N
        self.Umat = Tridiag()
        self.Umat.init(N)
        self.B_U = <double*> malloc(2 * sizeof(double))
        self.geo = geo
        self.L = Lambda
        self.C = C

    #assigns matrix Umat coef values using formulas given in theoretical doc
    #also computes B_U
    cdef void compute(self):

        #first row
        self.Umat.set(0,0,
                      (-self.L[0]*self.geo.spacing[0]**(-1)-self.L[1]*self.geo.
    spacing[1]**(-1))
                      *(self.C[0]*self.geo.width[1])**(-1))
        self.Umat.set(0,1,
                      self.L[1]
                      *(self.C[0]*self.geo.width[1]*self.geo.spacing[1])**(-1))

        cdef int i
        for i in range(1,self.N-1):
            self.Umat.set(i,i-1,
                          self.L[i]
                          *(self.C[i]*self.geo.width[i+1]*self.geo.spacing[i])**(-1))
            self.Umat.set(i,i,
                          (-self.L[i]*self.geo.spacing[i]**(-1)-self.L[i+1]*self.geo.
    spacing[i+1]**(-1))
                          *(self.C[i]*self.geo.width[i+1])**(-1))
            self.Umat.set(i,i+1,
                          self.L[i+1]
                          *(self.C[i]*self.geo.width[i+1]*self.geo.spacing[i+1])
    **(-1))

        #last row
        self.Umat.set(self.N-1,self.N-2,
                      self.L[self.N-1]
                      *(self.C[self.N-1]*self.geo.width[self.N]*self.geo.spacing[self
    .N-1])**(-1))
        self.Umat.set(self.N-1,self.N-1,
                      (-self.L[self.N-1]*self.geo.spacing[self.N-1]**(-1)-self.L[self
    .N]*self.geo.spacing[self.N]**(-1))
                      *(self.C[self.N-1]*self.geo.width[self.N])**(-1))

        self.B_U[0] = (self.L[0]
                       *(self.C[0]*self.geo.width[1]*self.geo.spacing[0])**(-1))
        self.B_U[1] = (self.L[self.N]
                       *(self.C[self.N-1]*self.geo.width[self.N]*self.geo.spacing[
    self.N])**(-1))

    cdef void free(self):
        self.Umat.free()
        free(self.B_U)

#to build an advection matrix for discretized equations
cdef class Vmatrix:
    #number of mail cells
    cdef int N
    cdef Tridiag Vmat
    #auxiliary array used in boundary conditions
    cdef double* B_V
    cdef Geometry geo
```

```
    #conductivities
    cdef double* L
    #capacities
    cdef double* C
    #permeabilities
    cdef double* K
    #water heat capacity [J*kg^{-1}*K^{-1}]
    cdef double cw
    #water density [kg*m^{-3}]
    cdef double rhow

    cdef void init(self, int N, Geometry geo, double* Lambda, double* C, double* K):
        self.N = N
        self.Vmat = Tridiag()
        self.Vmat.init(N)
        self.B_V = <double*> malloc(2 * sizeof(double))
        self.geo = geo
        self.L = Lambda
        self.C = C
        self.K = K
        self.cw = 4180
        self.rhow = 1000

    #assigns matrix Vmat coef values using formulas given in theoretical doc
    #also computes B_V
    cdef void compute(self, double* grad):

        #first row
        self.Vmat.set(0,0,
                      self.cw*self.rhow*(-self.K[0]*grad[0]*self.geo.width[0]*(2*self.geo.spacing[0])**(-1)
                                         +self.K[1]*grad[1]*self.geo.width[2]*(2*self.geo.spacing[1])**(-1))
                      *(self.C[0]*self.geo.width[1])**(-1))
        self.Vmat.set(0,1,
                      self.cw*self.rhow*self.K[1]*grad[1]
                      *(2*self.C[0]*self.geo.spacing[1])**(-1))

        cdef int i
        for i in range(1,self.N-1):
            self.Vmat.set(i,i-1,
                          -self.cw*self.rhow*self.K[i]*grad[i]
                          *(2*self.C[i]*self.geo.spacing[i])**(-1))
            self.Vmat.set(i,i,
                          self.cw*self.rhow*(-self.K[i]*grad[i]*self.geo.width[i]*(2*self.geo.spacing[i])**(-1)
                                             +self.K[i+1]*grad[i+1]*self.geo.width[i+2]*(2*self.geo.spacing[i+1])**(-1))
                          *(self.C[i]*self.geo.width[i+1])**(-1))
            self.Vmat.set(i,i+1,
                          self.cw*self.rhow*self.K[i+1]*grad[i+1]
                          *(2*self.C[i]*self.geo.spacing[i+1])**(-1))

        #last row
        self.Vmat.set(self.N-1,self.N-2,
                      -self.cw*self.rhow*self.K[self.N-1]*grad[self.N-1]
                      *(2*self.C[self.N-1]*self.geo.spacing[self.N-1])**(-1))
        self.Vmat.set(self.N-1,self.N-1,
                      self.cw*self.rhow*(-self.K[self.N-1]*grad[self.N-1]*self.geo.width[self.N-1]*(2*self.geo.spacing[self.N-1])**(-1)
                                         +self.K[self.N]*grad[self.N]*self.geo.width[self.N+1]*(2*self.geo.spacing[self.N])**(-1))
                      *(self.C[self.N-1]*self.geo.width[self.N])**(-1))

        self.B_V[0] = (-self.cw*self.rhow*self.K[0]*grad[0]
                       *(2*self.geo.spacing[0]*self.C[0])**(-1))
        self.B_V[1] = (self.cw*self.rhow*self.K[self.N]*grad[self.N]
                       *(2*self.geo.spacing[self.N]*self.C[self.N-1])**(-1))
```

```
450
451     cdef void free(self):
452         self.Vmat.free()
453         free(self.B_V)
454
455 #to build an efficient linear data interpolation function
456 #requires data sampled with a constant sampling step
457 cdef class QuickInterpol:
458     #number of samples
459     cdef int Ntimes
460     #sampling step
461     cdef double dt
462     #sampling start
463     cdef double t0
464     #sampling end
465     cdef double tf
466     #array storing data to interpolate
467     cdef double* X
468
469     cdef void init(self,int Ntimes, double dt, double t0, double tf, double* X):
470         self.Ntimes = Ntimes
471         self.dt = dt
472         self.t0 = t0
473         self.tf = tf
474         self.X = X
475
476     #returns a linear interpolation at "time" t
477     cdef double eval(self,double t):
478         cdef int n
479         cdef double frac
480         cdef double e
481         #before sampling start
482         if t <= self.t0:
483             return self.X[0]
484         #after sampling end
485         elif t >= self.tf:
486             return self.X[self.Ntimes-1]
487         else:
488             frac = (t-self.t0)*(self.dt)**(-1)
489             n = <int>(frac)
490             e = frac-n
491             return self.X[n]*(1-e) + self.X[n+1]*e
492
493     cdef void free(self):
494         free(self.X)
495
496 #to build a linear data interpolation function
497 #with possibly a non constant sampling step
498 cdef class SlowInterpol:
499     #number of samples
500     cdef int Ntimes
501     #sampling times
502     cdef double* times
503     #sampling start
504     cdef double t0
505     #sampling end
506     cdef double tf
507     #array storing data to interpolate
508     cdef double* X
509
510     cdef void init(self,int Ntimes, double* times, double* X):
511         self.Ntimes = Ntimes
512         self.times = times
513         self.t0 = times[0]
514         self.tf = times[Ntimes-1]
515         self.X = X
516
517     #returns a linear interpolation at "time" t
```

```
    #uses dichotomic method
    cdef double eval(self,double t):
        cdef int a = 0
        cdef int b = self.Ntimes
        cdef int m
        cdef double e
        if t<self.t0:
            return self.X[0]
        elif t>=self.tf:
            return self.X[self.Ntimes-1]
        else:
            while b-a>1:
                m = (a+b)//2
                if t < self.times[m]:
                    b = m
                else:
                    a = m
            e = (t-self.X[a])*(self.X[a+1]-self.X[a])**(-1)
            return self.X[a]*(1-e) + self.X[a+1]*e

    cdef void free(self):
        free(self.X)
        free(self.times)

#tool to summarize physical properties of ground layers and sublayers
cdef class StratifiedParams:

    #number of layers
    cdef int Nl
    #depths of layers boundaries [m]
    #[right limit layer 0, right limit interlayer 01, right limit layer 1 ...]
    cdef double* border
    #conductivies in layers
    cdef double* Ll
    #conductivies in sublayers
    cdef double* L
    #capacities in layers
    cdef double* Cl
    #capacities in sublayers
    cdef double* C
    #permeabilities in layers
    cdef double* Kl
    #permeabilities in sublayers
    cdef double* K
    #specific capacity in layers
    cdef double* Sl
    #specific capacity in sublayers
    cdef double* S
    cdef Geometry geo

    cdef void init(self, int Nl, double* border, Geometry geo):
        self.Nl = Nl
        self.border = border
        self.L = <double*> malloc((geo.N+1) * sizeof(double))
        self.C = <double*> malloc(geo.N * sizeof(double))
        self.K = <double*> malloc((geo.N+1) * sizeof(double))
        self.S = <double*> malloc(geo.N * sizeof(double))
        self.geo = geo

    cdef void update_layers_params(self, double* Ll, double* Cl, double* Kl, double*
    Sl):
        self.Ll = Ll
        self.Cl = Cl
        self.Kl = Kl
        self.Sl = Sl

    #assigns sublayers parameters values according to the layer they belong to
    cdef void compute(self):
```

```
585
586         cdef int i
587         cdef int d = 0
588         cdef int layer_bool = 1
589         cdef int layer_id = 0
590         cdef double e
591
592         #conductive properties (specific to an interface)
593         for i in range(self.geo.N+1):
594
595             while self.geo.depth_i[i] > self.border[d]:
596                 d = d + 1
597                 if layer_bool == 0:
598                     layer_bool = 1
599                     layer_id = layer_id + 1
600                 else:
601                     layer_bool = 0
602
603             #in the layer
604             if layer_bool == 1:
605                 self.L[i] = self.Ll[layer_id]
606                 self.K[i] = self.Kl[layer_id]
607             #in the interlayer (or interface)
608             else:
609                 e = (self.border[d]-self.geo.depth_i[i])*(self.border[d]-self.border[
    d-1])**(-1)
610                 self.L[i] = e*self.Ll[layer_id] + (1-e)*self.Ll[layer_id+1]
611                 self.K[i] = e*self.Kl[layer_id] + (1-e)*self.Kl[layer_id+1]
612
613         d = 0
614         layer_bool = 1
615         layer_id = 0
616
617         #capacitive properties (specific to a cell or sublayer)
618         for i in range(self.geo.N):
619
620             while self.geo.depth_c[i] > self.border[d]:
621                 d = d + 1
622                 if layer_bool == 0:
623                     layer_bool = 1
624                     layer_id = layer_id + 1
625                 else:
626                     layer_bool = 0
627
628             #in the layer
629             if layer_bool == 1:
630                 self.C[i] = self.Cl[layer_id]
631                 self.S[i] = self.Sl[layer_id]
632             #in the interlayer
633             else:
634                 e = (self.border[d]-self.geo.depth_i[i])*(self.border[d]-self.border[
    d-1])**(-1)
635                 self.C[i] = e*self.Cl[layer_id] + (1-e)*self.Cl[layer_id+1]
636                 self.S[i] = e*self.Sl[layer_id] + (1-e)*self.Sl[layer_id+1]
637
638     cdef void free(self):
639         free(self.L)
640         free(self.Ll)
641         free(self.C)
642         free(self.Cl)
643         free(self.K)
644         free(self.Kl)
645         free(self.S)
646         free(self.Sl)
647         free(self.border)
648
649 cdef double norm(double* X, int n, int id = 2):
650     cdef double result = 0
```

```
    cdef int i
    for i in range(n):
        result = result + X[i]**(id)
    return result**((<double> id)**(-1))

cdef double distance(double* X, double* Y, int n, int id = 2):
    cdef double result = 0
    cdef int i
    for i in range(n):
        result = result + (X[i]-Y[i])**(id)
    return result**((<double> id)**(-1))

#tool to solve discretized equations
cdef class StratifiedModel:

    #number of main cells
    cdef int N
    cdef StratifiedParams params
    cdef Umatrix Umatrix_T
    cdef Vmatrix Vmatrix_T
    cdef Umatrix Umatrix_H
    cdef Geometry geo
    #array storing cells hydraulic charges [m]
    cdef double* H
    #array storing hydraulic gradient
    cdef double* Hz
    #array storing cells temperatures [K]
    cdef double* T

    #to get boundary conditions
    cdef QuickInterpol Hriv
    cdef QuickInterpol Haq
    cdef QuickInterpol Triv
    cdef QuickInterpol Taq

    #to facilitate computations
    cdef Tridiag I
    cdef Tridiag A

    cdef double* depths_meas
    cdef int nb_meas
    #location of the cells at measures depths
    cdef int* interpol_index
    #weights necessary for linear interpolation
    cdef double* interpol_weights
    cdef double norm_H0
    cdef double norm_H1
    cdef double norm_T0
    cdef double norm_T1
    cdef double atol_H
    cdef double rtol_H
    cdef double atol_T
    cdef double rtol_T
    cdef int norm_id

    cdef void init(self,
                   int N,
                   double h,
                   double* Hriv,
                   double* Haq,
                   double* Triv,
                   double* Taq,
                   int Ntimes,
                   double dt,
                   double t0,
                   double tf,
                   int Nl,
                   double* borders,
```

```
719                 double* depths_meas,
720                 int nb_meas):
721
722         self.N = N
723         self.geo = Geometry()
724         self.geo.init(N)
725         self.geo.regular(h)
726
727         self.Hriv = QuickInterpol()
728         self.Hriv.init(Ntimes,dt,t0,tf,Hriv)
729         self.Haq = QuickInterpol()
730         self.Haq.init(Ntimes,dt,t0,tf,Haq)
731         self.Triv = QuickInterpol()
732         self.Triv.init(Ntimes,dt,t0,tf,Triv)
733         self.Taq = QuickInterpol()
734         self.Taq.init(Ntimes,dt,t0,tf,Taq)
735
736         self.params = StratifiedParams()
737         self.params.init(Nl,borders,self.geo)
738
739         self.Umatrix_H = Umatrix()
740         self.Umatrix_T = Umatrix()
741         self.Vmatrix_T = Vmatrix()
742
743         self.Hz = <double*> malloc((self.N+1) * sizeof(double))
744         self.H = <double*> malloc(self.N * sizeof(double))
745         self.T = <double*> malloc(self.N * sizeof(double))
746
747         self.I = Tridiag()
748         self.I.init(N)
749         self.I.identity()
750
751         self.A = Tridiag()
752         self.A.init(N)
753
754         self.depths_meas = depths_meas
755         self.nb_meas = nb_meas
756
757     #updates static paremeters and associated objects
758     cdef void update_params(self,double* Ll, double* Cl, double* Kl, double* Sl):
759         self.params.update_layers_params(Ll,Cl,Kl,Sl)
760         self.params.compute()
761         self.Umatrix_T.init(self.N,self.geo,self.params.L,self.params.C)
762         self.Umatrix_T.compute()
763         self.Vmatrix_T.init(self.N,self.geo,self.params.L,self.params.C,self.params.K
    )
764         self.Umatrix_H.init(self.N,self.geo,self.params.K,self.params.S)
765         self.Umatrix_H.compute()
766
767     #computes interpol_index and interpol_weights
768     cdef void prepare_interpol_T(self):
769
770         self.interpol_index = <int*> malloc(self.nb_meas * sizeof(int))
771         self.interpol_weights = <double*> malloc(self.nb_meas * sizeof(double))
772         cdef int i, a, b, m
773         cdef double e, d
774
775         for i in range(self.nb_meas):
776
777             d = self.depths_meas[i]
778
779             if d <= self.geo.depth_c[0]:
780                 self.interpol_index[i] = 0
781                 self.interpol_weights[i] = 1
782             elif d >= self.geo.depth_c[self.N-1]:
783                 self.interpol_index[i] = self.N-2
784                 self.interpol_weights[i] = 0
785             else:
```

```
                a = 0
                b = self.N
                while b-a>1:
                    m = (a+b)//2
                    if self.depths_meas[i] < self.geo.depth_c[m]:
                        b = m
                    else:
                        a = m
                e = (d-self.geo.depth_c[a])*(self.geo.depth_c[a+1]-self.geo.depth_c[a
    ])**(-1)
                self.interpol_index[i] = a
                self.interpol_weights[i] = 1-e

    #computes hydraulic gradient
    cdef void compute_Hz(self,double hriv, double haq):
        self.Hz[0] = (self.H[0]-hriv)*(self.geo.spacing[0])**(-1)
        cdef int i
        for i in range(1,self.N):
            self.Hz[i] = (self.H[i]-self.H[i-1])*(self.geo.spacing[i])**(-1)
        self.Hz[self.N] = (haq-self.H[self.N-1])*(self.geo.spacing[self.N])**(-1)

    #One Step Implicit Euler
    #computes the next T and H at t + dt
    cdef void OSIEuler(self,double t, double dt):

        #boundaries parameters
        cdef double hriv = self.Hriv.eval(t+dt)
        cdef double haq = self.Haq.eval(t+dt)
        cdef double triv = self.Triv.eval(t+dt)
        cdef double taq = self.Taq.eval(t+dt)

        cdef double* aux = <double*> malloc(self.N * sizeof(double))
        cdef int i

        #hydraulic boundary conditions
        aux[0] = dt*self.Umatrix_H.B_U[0]*hriv + self.H[0]
        for i in range(1,self.N-1):
            aux[i] = self.H[i]
        aux[self.N-1] = dt*self.Umatrix_H.B_U[1]*haq + self.H[self.N-1]

        #hydraulic conduction matrix
        self.A.copy(self.Umatrix_H.Umat.mat)
        self.A.scalarmul(-dt)
        self.A.add(self.I.mat,self.A.mat)

        free(self.H)
        #computes H[t+dt]
        self.H = self.A.solve(aux)

        #heat exchange matrix
        self.compute_Hz(hriv,haq)
        self.Vmatrix_T.compute(self.Hz)
        self.A.add(self.Umatrix_T.Umat.mat,self.Vmatrix_T.Vmat.mat)
        self.A.scalarmul(-dt)
        self.A.add(self.I.mat,self.A.mat)

        #heat boundary conditions
        aux[0] = dt*(self.Umatrix_T.B_U[0]+self.Vmatrix_T.B_V[0])*triv + self.T[0]
        for i in range(1,self.N-1):
            aux[i] = self.T[i]
        aux[self.N-1] = dt*(self.Umatrix_T.B_U[1]+self.Vmatrix_T.B_V[1])*taq  + self.
    T[self.N-1]

        free(self.T)
        #computes T[t+dt]
        self.T = self.A.solve(aux)
        free(aux)

```

```
    #One Step Heun
    #computes the next T and H at t + dt
    cdef void OSHeun(self,double t, double dt):

        #boundaries parameters
        cdef double hriv0 = self.Hriv.eval(t)
        cdef double haq0 = self.Haq.eval(t)
        cdef double triv0 = self.Triv.eval(t)
        cdef double taq0 = self.Taq.eval(t)

        cdef double hriv = self.Hriv.eval(t+dt)
        cdef double haq = self.Haq.eval(t+dt)
        cdef double triv = self.Triv.eval(t+dt)
        cdef double taq = self.Taq.eval(t+dt)

        cdef double* aux = <double*> malloc(self.N * sizeof(double))
        cdef int i

        cdef double* f = self.Umatrix_H.Umat.dot(self.H)

        #hydraulic boundary conditions
        aux[0] = 0.5*dt*self.Umatrix_H.B_U[0]*(hriv0+hriv) + self.H[0] + 0.5*dt*f[0]
        for i in range(1,self.N-1):
            aux[i] = self.H[i] + 0.5*dt*f[i]
        aux[self.N-1] = 0.5*dt*self.Umatrix_H.B_U[1]*(haq0+haq) + self.H[self.N-1] +
    0.5*dt*f[self.N-1]
        free(f)

        #hydraulic conduction matrix
        self.A.copy(self.Umatrix_H.Umat.mat)
        self.A.scalarmul(-0.5*dt)
        self.A.add(self.I.mat,self.A.mat)

        free(self.H)
        #computes H[t+dt]
        self.H = self.A.solve(aux)

        #heat exchange matrix
        self.A.add(self.Umatrix_T.Umat.mat,self.Vmatrix_T.Vmat.mat)
        f = self.A.dot(self.T)

        cdef double B_U0 = self.Umatrix_T.B_U[0]
        cdef double B_V0 = self.Vmatrix_T.B_V[0]
        cdef double B_U = self.Umatrix_T.B_U[1]
        cdef double B_V = self.Vmatrix_T.B_V[1]

        self.compute_Hz(hriv,haq)
        self.Vmatrix_T.compute(self.Hz)
        self.A.add(self.Umatrix_T.Umat.mat,self.Vmatrix_T.Vmat.mat)
        self.A.scalarmul(-0.5*dt)
        self.A.add(self.I.mat,self.A.mat)

        #heat boundary conditions
        aux[0] = 0.5*dt*((B_U0+B_V0)*triv0+(self.Umatrix_T.B_U[0]+self.Vmatrix_T.B_V
    [0])*triv) + self.T[0] + 0.5*dt*f[0]
        for i in range(1,self.N-1):
            aux[i] = self.T[i] + 0.5*dt*f[i]
        aux[self.N-1] = 0.5*dt*((B_U+B_V)*taq0+(self.Umatrix_T.B_U[1]+self.Vmatrix_T.
    B_V[1])*taq)  + self.T[self.N-1] + 0.5*dt*f[self.N-1]
        free(f)
        free(self.T)
        #computes T[t+dt]
        self.T = self.A.solve(aux)
        free(aux)

    #One Time Solve
    #computes T and H at tf
    def OTSolve(self, double t0, double tf, double dt, bint autostep = False):
```

```
        cdef double t = t0
        while t < tf:
            if autostep:
                #adaptive step size parameters
                dt = self.stepscaler(t,dt)
                self.norm_H0 = self.norm_H1
                self.norm_H1 = norm(self.H,self.N,id=self.norm_id)
                self.norm_T0 = self.norm_T1
                self.norm_T1 = norm(self.T,self.N,id=self.norm_id)
            if t + dt > tf:
                dt = tf - t
                self.OSHeun(t,dt)
                t = tf + 1
            else:
                self.OSHeun(t,dt)
                t = t + dt

    cdef void set_H(self,double* H0):
        cdef int i
        for i in range(self.N):
            self.H[i] = H0[i]

    cdef void set_T(self,double* T0):
        cdef int i
        for i in range(self.N):
            self.T[i] = T0[i]

    def init_stepscaler(self, double atol_H, double rtol_H, double atol_T, double
    rtol_T, int norm_id):
        self.norm_id = norm_id
        self.norm_H0 = norm(self.H,self.N,id=norm_id)
        self.norm_T0 = norm(self.T,self.N,id=norm_id)
        self.norm_H1 = self.norm_H0
        self.norm_T1 = self.norm_T0
        self.atol_H = atol_H
        self.rtol_H = rtol_H
        self.atol_T = atol_T
        self.rtol_T = rtol_T

    #adapts solver time step
    def stepscaler(self, double t, double dt):

        cdef int i
        cdef double* aux0_H = <double*> malloc(self.N * sizeof(double))
        cdef double* aux0_T = <double*> malloc(self.N * sizeof(double))
        cdef double* aux1_H = <double*> malloc(self.N * sizeof(double))
        cdef double* aux1_T = <double*> malloc(self.N * sizeof(double))

        for i in range(self.N):
            aux0_H[i] = self.H[i]
            aux0_T[i] = self.T[i]
            aux1_H[i] = self.H[i]
            aux1_T[i] = self.T[i]

        #computes Euler predictions
        self.OSIEuler(t,dt)
        cdef double* H_IEuler = self.H
        cdef double* T_IEuler = self.T

        self.H = aux0_H
        self.T = aux0_T
        self.compute_Hz(self.Hriv.eval(t),self.Haq.eval(t))
        self.Vmatrix_T.compute(self.Hz)

        #computes Heun predictions
        self.OSHeun(t,dt)
        cdef double* H_Heun = self.H
        cdef double* T_Heun = self.T
```

```
984
985         self.H = aux1_H
986         self.T = aux1_T
987
988         #time step adapted according to the prediction error
989         #error estimated with the difference between Euler and Heun predictions
990         cdef double dt_H = dt*((self.atol_H + self.rtol_H*max(self.norm_H0,self.
    norm_H1)) * distance(H_IEuler,H_Heun,self.N,id=self.norm_id)**(-1))**(0.5)
991         cdef double dt_T = dt*((self.atol_T + self.rtol_T*max(self.norm_T0,self.
    norm_T1)) * distance(T_IEuler,T_Heun,self.N,id=self.norm_id)**(-1))**(0.5)
992
993         self.compute_Hz(self.Hriv.eval(t),self.Haq.eval(t))
994         self.Vmatrix_T.compute(self.Hz)
995
996         free(H_IEuler)
997         free(H_Heun)
998         free(T_IEuler)
999         free(T_Heun)
1000
1001        return min(dt_H,dt_T)
1002
1003    #computes H and T at different times
1004    cdef void solve(self, Matrix memo, int ntimes, double* times, double t0, double
    dt, bint interpol_T = False, bint autostep = False):
1005
1006        cdef double t = t0
1007        cdef double val, e
1008        cdef int i, j, a
1009
1010        self.compute_Hz(self.Hriv.eval(t0),self.Haq.eval(t0))
1011        self.Vmatrix_T.compute(self.Hz)
1012
1013        for i in range(ntimes):
1014
1015            self.OTSolve(t,times[i],dt,autostep = autostep)
1016
1017            #to only store temperatures at measures depths
1018            if interpol_T:
1019                for j in range(self.nb_meas):
1020                    a = self.interpol_index[j]
1021                    e = self.interpol_weights[j]
1022                    val = e*self.T[a] + (1-e)*self.T[a+1]
1023                    memo.set(i,j,val)
1024
1025            #to keep all temperatures and hydraulic charges
1026            else:
1027                memo.slicerowset(i,0,self.N,self.H)
1028                memo.slicerowset(i,self.N,2*self.N,self.T)
1029
1030            t = times[i]
1031
1032    cdef void free(self, bint interpol_T = False):
1033        self.Umatrix_T.free()
1034        self.Umatrix_H.free()
1035        self.Vmatrix_T.free()
1036        self.A.free()
1037        self.I.free()
1038        self.geo.free()
1039        self.params.free()
1040        self.Hriv.free()
1041        self.Haq.free()
1042        self.Triv.free()
1043        self.Taq.free()
1044        free(self.Hz)
1045        free(self.H)
1046        free(self.T)
1047        free(self.depths_meas)
1048        if interpol_T:
```

```
            free(self.interpol_index)
            free(self.interpol_weights)

#function to call in a python script to test StratifiedModel
cpdef test_SM(int N,
          double h,
          double[:] Hriv,
          double[:] Haq,
          double[:] Triv,
          double[:] Taq,
          int Ntimes,
          double dt,
          double t0,
          double tf,
          int Nl,
          double[:] borders,
          double[:] depths_meas,
          double[:] Ll,
          double[:] Cl,
          double[:] Kl,
          double[:] Sl,
          double[:] H0,
          double[:] T0,
          int ntimes,
          double[:] times,
          double t02,
          double step,
          double atol_H,
          double rtol_H,
          double atol_T,
          double rtol_T,
          bint interpol_T = False,
          bint autostep = False):
    #conversion of python objects into C objects
    cdef double* Hriv2 = numpy_to_C(Hriv)
    cdef double* Haq2 = numpy_to_C(Haq)
    cdef double* Triv2 = numpy_to_C(Triv)
    cdef double* Taq2 = numpy_to_C(Taq)
    cdef double* borders2 = numpy_to_C(borders)
    cdef double* depths_meas2 = numpy_to_C(depths_meas)
    cdef double* Ll2 = numpy_to_C(Ll)
    cdef double* Cl2 = numpy_to_C(Cl)
    cdef double* Kl2 = numpy_to_C(Kl)
    cdef double* Sl2 = numpy_to_C(Sl)
    cdef double* H02 = numpy_to_C(H0)
    cdef double* T02 = numpy_to_C(T0)
    cdef double* times2 = numpy_to_C(times)
    #solver tool
    cdef StratifiedModel sm = StratifiedModel()
    sm.init(N,h,Hriv2,Haq2,Triv2,Taq2,Ntimes,dt,t0,tf,Nl,borders2,depths_meas2,
   depths_meas.shape[0])
    sm.update_params(Ll2,Cl2,Kl2,Sl2)
    sm.set_T(T02)
    sm.set_H(H02)
    if autostep:
        sm.init_stepscaler(atol_H,rtol_H,atol_T,rtol_T,2)
    cdef Matrix memo = Matrix()
    if interpol_T:
        memo.init(ntimes,len(depths_meas))
        sm.prepare_interpol_T()
        sm.solve(memo,ntimes,times2,t02,step,interpol_T=True,autostep=autostep)
        sm.free(interpol_T=True)
    else:
        memo.init(ntimes,2*N)
        sm.solve(memo,ntimes,times2,t02,step,interpol_T=False,autostep=autostep)
        sm.free(interpol_T=False)
    free(times2)
    free(H02)
```

```
    free(T02)
    A =  memo.convert_to_numpy()
    memo.free()
    return A

###Bayesian inference

#sets the random seed of rand using the clock
srand(time(NULL))

#uniformly draws a sample from [0,1]
cdef double random_uniform_std():
    cdef double r = rand()
    return (<double> r) * (<double> RAND_MAX)**(-1)

#draws a standard gaussian sample
#uses a polar version of the Box-Muller transformation
cdef double random_normal_std():
    cdef double x1, x2, w
    w = 2.0
    while (w >= 1.0):
        x1 = 2.0 * random_uniform_std() - 1.0
        x2 = 2.0 * random_uniform_std() - 1.0
        w = x1 * x1 + x2 * x2
    w = ((-2.0 * log(w)) * w**(-1)) ** 0.5
    return x1 * w

#uniformly draws a sample from [a,b]
cdef double random_uniform(double a, double b):
    return a + (b-a)*random_uniform_std()

#draws a gaussian sample of law (mean, sigma)
cdef double random_normal(double mean, double sigma):
    return mean + sigma*random_normal_std()

#probability of a set of consecutively numbered events
cdef double slicesum(double* probas, int a, int b):
    cdef double prob = 0
    cdef int i
    for i in range(a,b):
        prob = prob + probas[i]
    return prob

#to efficiently draw a sample from a finite distribution
#the idea: decompose a draw in a sequence of binomial experiments
#in practice, we use a binary tree
cdef class DiscreteRandomVariable:
    cdef int nb_events
    #probabilities
    cdef double* proba
    cdef int tree_depth
    #probabilities tree stored in an array
    cdef double* tree
    cdef int result

    cdef void init(self, int nb_events, double* proba):
        self.nb_events = nb_events
        self.proba = proba
        self.tree_depth = 1 + <int> (log(nb_events) * log(2)**(-1))
        self.tree = <double*> malloc((2**(self.tree_depth+1)) * sizeof(double))

    #to build the tree
    cdef void update_leaf(self,int a, int b, int id, int level, double previous):
        cdef int m = (b+a)//2
        if level == -1:
            self.tree[0] = 1
        else:
            self.tree[2**(level+1)-1+id] = slicesum(self.proba,a,b) * previous**(-1)
```

```
        if b-a>1:
            self.update_leaf(a, m, 2*id, level+1, slicesum(self.proba,a,b))
            self.update_leaf(m, b, 2*id+1, level+1, slicesum(self.proba,a,b))
    cdef void update(self):
        self.update_leaf(0, self.nb_events, 0, -1, 1)

    #to use the tree for random generation
    cdef void moove(self, int a, int b, int id, int level):
        cdef int m = (a+b)//2
        cdef double p_left = self.tree[2**(level+2)-1+2*id]

        if b-a>1:
            if random_uniform_std()<p_left:
                self.moove(a,m,2*id,level+1)
            else:
                self.moove(m,b,2*id+1,level+1)
        else:
            self.result = a
    cdef int generate(self):
        self.moove(0,self.nb_events,0,-1)

    cdef void free(self):
        free(self.tree)

cdef class MCMC:
    cdef int nb_layers
    #total number of parameters (of all layers)
    cdef int nb_params
    #number of Markov chains
    cdef int nb_chains
    #number of random walk steps
    cdef int nb_iter
    #number of ordered pairs
    cdef int nb_opairs
    #number of crossover probabilities
    cdef int nCR
    #number of a state dimensions
    cdef int nb_dim
    #static paremeters for random generation
    cdef double cn
    cdef double c
    cdef double t0
    #boolean arrays to keep track of selected elements (ex:chains or dimensions)
    cdef bint* dim_selector
    cdef bint* aux_selector
    cdef int* opairs_selector
    #ranges of physical parameters
    cdef double* range_L
    cdef double* range_C
    cdef double* range_K
    cdef double* range_S
    cdef double* range_sigma
    #layers parameters
    cdef double* Ll
    cdef double* Cl
    cdef double* Kl
    cdef double* Sl
    #measure times
    cdef double* times
    #crossover probabilities
    cdef double* probas
    #records the random walk
    cdef Matrix past
    #stores real physical measures
    cdef Matrix meas
    #stores current temperature predictions given by SM
    cdef Matrix pred
    #stores chain energies
```

```
    cdef Matrix energy
    cdef StratifiedModel sm

    cdef void init(self,
                   StratifiedModel sm,
                   int nb_chains,
                   int nb_iter,
                   Matrix meas,
                   double* times,
                   double t0,
                   double cn, double c,
                   int nb_opairs,
                   int nCR,
                   double* range_L,
                   double* range_C,
                   double* range_K,
                   double* range_S,
                   double* range_sigma):
        self.nb_iter = nb_iter
        self.times = times
        self.t0 = t0
        self.sm = sm
        self.nb_layers = sm.params.Nl
        self.nb_params = 4*sm.params.Nl+1
        self.nb_chains = nb_chains
        self.past = Matrix()
        self.past.init(nb_iter, self.nb_params * nb_chains)
        self.meas = meas
        self.pred = Matrix()
        self.pred.init(meas.n,meas.m)
        self.cn = cn
        self.c = c
        self.nb_opairs = nb_opairs
        self.nCR = nCR
        self.probas = <double*> malloc(nCR * sizeof(double))
        self.range_L = range_L
        self.range_C = range_C
        self.range_K = range_K
        self.range_S = range_S
        self.range_sigma = range_sigma
        self.energy = Matrix()
        self.energy.init(nb_iter,nb_chains)
        self.dim_selector = <bint*> malloc(self.nb_params * sizeof(bint))
        self.opairs_selector = <int*> malloc(2*nb_opairs * sizeof(int))
        self.aux_selector = <bint*> malloc(nb_chains * sizeof(bint))
        self.Ll = <double*> malloc(self.nb_layers * sizeof(double))
        self.Cl = <double*> malloc(self.nb_layers * sizeof(double))
        self.Kl = <double*> malloc(self.nb_layers * sizeof(double))
        self.Sl = <double*> malloc(self.nb_layers * sizeof(double))

    #assigns parameter j of chain i at given time
    cdef void set_chain_param(self, int time, int i, int j, double val):
        self.past.set(time, i * self.nb_params + j, val)

    #returns parameter j of chain i at given time
    cdef double get_chain_param(self, int time, int i, int j):
        return self.past.get(time, i * self.nb_params + j)

    #updateq layer parameters
    cdef void params_for_SM(self,int time, int id_chain):
        cdef int i
        for i in range(self.nb_layers):
            #conductivity
            self.Ll[i] = self.get_chain_param(time, id_chain, 4*i + 1)
            #thermal capacity
            self.Cl[i] = self.get_chain_param(time, id_chain, 4*i + 2)
            #permeability
            self.Kl[i] = self.get_chain_param(time, id_chain, 4*i + 3)
```

```
            #hydraulic capacity
            self.Sl[i] = self.get_chain_param(time, id_chain, 4*i + 4)

    cdef void init_chains(self):
        cdef int i, j
        cdef double sigma, L, C, K, S
        for i in range(self.nb_layers):
            for j in range(self.nb_chains):
                #sigma
                sigma = random_uniform(self.range_sigma[0],self.range_sigma[1])
                self.set_chain_param(0, j, 0, sigma)
                #conductivity
                L = random_uniform(self.range_L[2*i],self.range_L[2*i+1])
                self.set_chain_param(0, j, 4*i + 1, L)
                #thermal capacity
                C = random_uniform(self.range_C[2*i],self.range_C[2*i+1])
                self.set_chain_param(0, j, 4*i + 2, C)
                #permeability
                K = random_uniform(self.range_K[2*i],self.range_K[2*i+1])
                self.set_chain_param(0, j, 4*i + 3, K)
                #hydraulic capacity
                S = random_uniform(self.range_S[2*i],self.range_S[2*i+1])
                self.set_chain_param(0, j, 4*i + 4, S)

    #keeps chains within the bounded domain defined by parameters ranges
    cdef void mod(self, int time, int id_chain):
        cdef int i
        cdef double sigma, L, C, K, S, r, frac
        cdef int n
        cdef int j = id_chain
        for i in range(self.nb_layers):
            #sigma
            sigma = self.get_chain_param(time, j, 0)
            r = abs(sigma - self.range_sigma[2*i]) * (self.range_sigma[2*i+1] - self.
range_sigma[2*i])**(-1)
            n = <int> r
            frac = r - n
            sigma = self.range_sigma[2*i] + frac*(self.range_sigma[2*i+1] - self.
range_sigma[2*i])
            self.set_chain_param(time, j, 0, sigma)
            #conductivity
            L = self.get_chain_param(time, j, 4*i + 1)
            r = abs(L - self.range_L[2*i]) * (self.range_L[2*i+1] - self.range_L[2*i
])**(-1)
            n = <int> r
            frac = r - n
            L = self.range_L[2*i] + frac*(self.range_L[2*i+1] - self.range_L[2*i])
            self.set_chain_param(time, j, 4*i + 1, L)
            #thermal capacity
            C = self.get_chain_param(time, j, 4*i + 2)
            r = abs(C - self.range_C[2*i]) * (self.range_C[2*i+1] - self.range_C[2*i
])**(-1)
            n = <int> r
            frac = r - n
            C = self.range_C[2*i] + frac*(self.range_C[2*i+1] - self.range_C[2*i])
            self.set_chain_param(time, j, 4*i + 2, C)
            #permeability
            K = self.get_chain_param(time, j, 4*i + 3)
            r = abs(K - self.range_K[2*i]) * (self.range_K[2*i+1] - self.range_K[2*i
])**(-1)
            n = <int> r
            frac = r - n
            K = self.range_K[2*i] + frac*(self.range_K[2*i+1] - self.range_K[2*i])
            self.set_chain_param(time, j, 4*i + 3, K)
            #hydraulic capacity
            S = self.get_chain_param(time, j, 4*i + 4)
            r = abs(S - self.range_S[2*i]) * (self.range_S[2*i+1] - self.range_S[2*i
])**(-1)
```

```
            n = <int> r
            frac = r - n
            S = self.range_S[2*i] + frac*(self.range_S[2*i+1] - self.range_S[2*i])
            self.set_chain_param(time, j, 4*i + 4, S)

    #init with approximated hydraulic steady state solution
    cdef void init_H(self):

        cdef int i
        cdef double Itot = 0
        for i in range(self.sm.N):
            Itot = Itot + self.sm.geo.width[i] * self.sm.params.K[i]**(-1)
        cdef double Hz0 = (self.sm.Haq.eval(self.t0) - self.sm.Hriv.eval(self.t0)) *
Itot**(-1)

        self.sm.H[0] = self.sm.Hriv.eval(self.t0) + Hz0 * self.sm.geo.width[0] * self
.sm.params.K[0]**(-1)
        for i in range(1,self.sm.N):
            self.sm.H[i] = self.sm.H[i-1] + Hz0 * self.sm.geo.width[i] * self.sm.
params.K[i]**(-1)

    #init with approximated thermal steady state solution
    cdef void init_T(self):

        cdef int i
        cdef double* exponan = <double*> malloc(self.sm.N * sizeof(double))
        cdef double Itot = 0
        cdef double cw = self.sm.Vmatrix_T.cw
        cdef double rhow = self.sm.Vmatrix_T.rhow

        exponan[0] = self.sm.geo.width[0]*(cw*rhow*self.sm.params.K[0]*(self.sm.H[0]-
self.sm.Hriv.eval(self.t0))
                        *(self.sm.geo.spacing[0]*self.sm.params.L[0])**(-1))

        for i in range(1,self.sm.N):
            exponan[i] = exponan[i-1] + self.sm.geo.width[i]*(cw*rhow*self.sm.params.
K[i]*(self.sm.H[i]-self.sm.H[i-1])
                                            *(self.sm.geo.spacing[i]*self.sm.params.L[i
])**(-1))

        for i in range(self.sm.N):
            Itot = Itot + self.sm.geo.width[i] * exp(-exponan[i])

        cdef double Tz0 = (self.sm.Taq.eval(self.t0) - self.sm.Triv.eval(self.t0)) *
Itot**(-1)

        self.sm.T[0] = self.sm.Triv.eval(self.t0) + Tz0 * self.sm.geo.width[0] * exp
(-exponan[0])

        for i in range(1,self.sm.N):
            self.sm.T[i] = self.sm.T[i-1] + Tz0 * self.sm.geo.width[i] * exp(-exponan
[i])

        free(exponan)

    #init crossover probabilities
    cdef void init_probas(self):
        cdef int i
        for i in range(self.nCR):
            self.probas[i] = (<double> self.nCR)**(-1)

    cdef void predict(self, bint autostep = False, double dt = 300):
        self.sm.solve(self.pred,self.meas.n,self.times,self.t0,dt,interpol_T=True,
autostep=autostep)

    cdef void compute_energy(self, int time, int id_chain):
        cdef double e = 0
        cdef int i, j
```

```
        cdef double sigma = self.get_chain_param(time,id_chain,0)
        for i in range(self.meas.n):
            for j in range(self.meas.m):
                e = e + (self.meas.get(i,j) - self.pred.get(i,j))**2
        self.energy.set(time,id_chain, 0.5*e*sigma**(-2)+self.meas.n*self.meas.m*log(
    sigma))

    cdef double compute_acceptancy(self, int time, int id_chain):
        cdef double e0 = self.energy.get(time-1,id_chain)
        cdef double e1 = self.energy.get(time,id_chain)
        if e1<=e0:
            return 1
        else:
            return exp(e0-e1)

    #select crossover dimensions
    cdef void crossover(self):

        cdef int a = 0
        cdef int b = self.nCR
        cdef int m, i
        cdef double rnd, ceil
        cdef bint onetrue = False

        #rmk: instead you could use DiscreteRandomVariable to replace this loop
        while b-a > 1:
            m = (a+b)//2
            p = slicesum(self.probas,a,m)*(slicesum(self.probas,a,b))**(-1)
            rnd = random_uniform_std()
            if rnd < p:
                b = m
            else:
                a = m
        ceil = (a + 1) * self.nCR**(-1)

        self.nb_dim = 0
        for i in range(self.nb_params):
            rnd = random_uniform_std()
            if rnd < ceil:
                self.dim_selector[i] = True
                onetrue = True
                self.nb_dim = self.nb_dim + 1
            else:
                self.dim_selector[i] = False

        if not onetrue:
            rnd = self.nb_params * random_uniform_std()
            self.dim_selector[<int> rnd] = True
            self.nb_dim = 1

    cdef void opairs_selection(self):

        cdef int i, j
        for i in range(self.nb_chains):
            self.aux_selector[i] = True

        for i in range(self.nb_opairs):

            j = <int> (self.nb_chains * random_uniform_std())
            while not self.aux_selector[j]:
                j = j + 1
                if j > self.nb_chains-1:
                    j = 0
            self.opairs_selector[2*i] = j
            self.aux_selector[j] = False

            j = <int> (self.nb_chains * random_uniform_std())
            while not self.aux_selector[j]:
```

```
                j = j + 1
                if j > self.nb_chains-1:
                    j = 0
            self.opairs_selector[2*i+1] = j
            self.aux_selector[j] = False

    #random generators used in DREAMS
    cdef double alpha(self):
        return random_normal(0,self.cn)
    cdef double beta(self,double gamma = 2.38*2**(-0.5)):
        cdef double inf = (1-self.c)*gamma*(self.nb_opairs*self.nb_dim)**(-0.5)
        cdef double sup = (1+self.c)*gamma*(self.nb_opairs*self.nb_dim)**(-0.5)
        return random_uniform(inf,sup)

    #new chain state generation
    cdef void generate(self,int time, int id_chain):
        cdef int i,j
        cdef double a, b, s
        cdef int id1, id2
        for i in range(self.nb_params):
            s = 0
            if self.dim_selector[i]:
                a = self.alpha()
                b = self.beta()
                for j in range(self.nb_opairs):
                    id1 = self.opairs_selector[2*j]
                    id2 = self.opairs_selector[2*j+1]
                    s = s + self.get_chain_param(time,id2,i) - self.get_chain_param(
time,id1,i)
                s = a + b*s
            s = s + self.get_chain_param(time,id_chain,i)
            self.set_chain_param(time+1,id_chain,i,s)

    #when new chain state is not accepted, rewrites the previous one
    cdef void rewrite(self,int time, int id_chain):
        cdef int i
        cdef double past_val
        for i in range(self.nb_params):
            past_val = self.get_chain_param(time-1,id_chain,i)
            self.set_chain_param(time,id_chain,i,past_val)

    #rmk: to rigorously respect symmetric increments
    #you should modify the last two intricated loops
    cdef void walk(self,bint autostep=False):

        cdef int i, k
        cdef double rnd
        cdef double accept = 0
        self.init_chains()
        self.init_probas()
        for i in range(self.nb_chains):
            self.params_for_SM(0,i)
            self.sm.update_params(self.Ll,self.Cl,self.Kl,self.Sl)
            self.init_H()
            self.init_T()
            self.predict(autostep=autostep)
            self.compute_energy(0,i)

        for k in range(self.nb_iter-1):
            for i in range(self.nb_chains):
                self.crossover()
                self.opairs_selection()
                self.generate(k,i)
                self.mod(k+1,i)
                self.params_for_SM(k+1,i)
                self.sm.update_params(self.Ll,self.Cl,self.Kl,self.Sl)
                self.init_H()
                self.init_T()
```

```
                self.predict(autostep=autostep)
                self.compute_energy(k+1,i)
                accept = self.compute_acceptancy(k+1,i)
                rnd = random_uniform_std()
                if rnd > accept:
                    self.rewrite(k+1,i)

    cdef void free(self):
        self.sm.free()
        self.past.free()
        self.pred.free()
        self.energy.free()
        free(self.probas)
        free(self.range_L)
        free(self.range_C)
        free(self.range_K)
        free(self.range_S)
        free(self.range_sigma)
        free(self.times)
        free(self.dim_selector)
        free(self.opairs_selector)
        free(self.aux_selector)

#function to call in a python script to test StratifiedModel
def test_MCMC(int N,
          double h,
          double[:] Hriv,
          double[:] Haq,
          double[:] Triv,
          double[:] Taq,
          int Ntimes,
          double dt,
          double t0,
          double tf,
          int Nl,
          double[:] borders,
          double[:] depths_meas,
          int nb_chains,
          int nb_iter,
          int nb_layers,
          double[:, ::1] meas,
          double[:] times,
          double t0_solve,
          double cn,
          double c,
          int nb_opairs,
          int nCR,
          double[:] range_L,
          double[:] range_C,
          double[:] range_K,
          double[:] range_S,
          double[:] range_sigma,
          double atol_H,
          double rtol_H,
          double atol_T,
          double rtol_T,
          bint autostep = False):

    #conversion to C arrays
    cdef double* Hriv_C = numpy_to_C(Hriv)
    cdef double* Haq_C = numpy_to_C(Haq)
    cdef double* Triv_C = numpy_to_C(Triv)
    cdef double* Taq_C = numpy_to_C(Taq)
    cdef double* borders_C = numpy_to_C(borders)
    cdef double* depths_meas_C = numpy_to_C(depths_meas)
    cdef Matrix meas_C = Matrix()
    meas_C.init(meas.shape[0],meas.shape[1])
    meas_C.extract_from_numpy(meas)
```

```
    cdef double* times_C = numpy_to_C(times)
    cdef double* range_L_C = numpy_to_C(range_L)
    cdef double* range_C_C = numpy_to_C(range_C)
    cdef double* range_K_C = numpy_to_C(range_K)
    cdef double* range_S_C = numpy_to_C(range_S)
    cdef double* range_sigma_C = numpy_to_C(range_sigma)

    #physical model
    cdef StratifiedModel sm = StratifiedModel()
    sm.init(N,h,Hriv_C,Haq_C,Triv_C,Taq_C,Ntimes,dt,t0,tf,Nl,borders_C,depths_meas_C,
    depths_meas.shape[0])
    if autostep:
        sm.init_stepscaler(atol_H,rtol_H,atol_T,rtol_T,2)
    sm.prepare_interpol_T()

    #MCMC random walk
    cdef MCMC mcmc = MCMC()
    mcmc.init(sm,nb_chains,nb_iter,meas_C,times_C,t0_solve,cn,c,nb_opairs,nCR,
    range_L_C,range_C_C,range_K_C,range_S_C,range_sigma_C)
    mcmc.walk(autostep=autostep)

    return mcmc.past.convert_to_numpy()
```

4.1.2 setup file (MOLO_45_setup.py)

```
from setuptools import Extension, setup
from Cython.Build import cythonize

ext_modules = [
    Extension("MOLO_45",
              sources=["MOLO_45.pyx"],
              libraries=["m"]  # Unix-like specific
              )
]

setup(name="MOLO_setup",
      ext_modules=cythonize(ext_modules))

#After installing cython ...
#... to compile MOLO_45.pyx using linux ...
#... execute the command: python3 setup.py build_ext --inplace
```

4.1.3 code de test du StratifiedModel (MOLO_45_test_SM.py)

```
from MOLO_45 import test_SM
import numpy as np
import matplotlib.pyplot as plt

#number of main cells
N = 10
#ground layer height [m]
h = 0.4
#constant boundary parameters
Hriv = np.array([0.05,0.05],dtype=np.double)
Haq = np.array([0,0],dtype=np.double)
Triv = np.array([300,300],dtype=np.double)
Taq = np.array([290,290],dtype=np.double)
#number of samples provided for boundary parameters
Ntimes = 2
#sampling start
t0 = 0
#sampling end
tf = 3600
#sampling step
dt = tf
#number of layers
Nl = 2
#two layers separated by a 10 cm thick interlayer
```

```
#the second layer stops at at depth of 1 m
borders = np.array([0.15,0.25,1],dtype=np.double)
#three main measures at the depths of 10, 20 and 30 cm
depths_meas = np.array([0.1,0.2,0.3],dtype=np.double)
#layer parameters
Ll = np.array([3,3],dtype=np.double)
Cl = np.array([4*10**(6),4*10**(6)],dtype=np.double)
Kl = np.array([10**(-5),10**(-7)],dtype=np.double)
Sl = np.array([0.2,0.2],dtype=np.double)
#initial hydraulic and thermal conditions
H0 = 0.025*np.ones(N,dtype=np.double)
T0 = 295*np.ones(N,dtype=np.double)
#number of times at which H and T must be predicted
ntimes = 1
times = np.array([24*3600],dtype=np.double)
t02 = 0
#solver time step (adapted if autostep)
step = 0.1
#error control parameters if autostep
atol_H = 10**(-6)
rtol_H = 10**(-4)
atol_T = 10**(-6)
rtol_T = 10**(-4)

#solving discretized equations
result = test_SM(N,h,Hriv,Haq,Triv,Taq,Ntimes,dt,t0,tf,Nl,borders,depths_meas,Ll,Cl,
    Kl,Sl,H0,T0,ntimes,times,t02,step,atol_H,rtol_H,atol_T,rtol_T,autostep=True)

#displaying hydraulic charge curve
plt.plot(np.linspace(0,h,N),result[0,:N])
plt.ylabel('hydraulic charge [m]')
plt.xlabel('depth [m]')
plt.show()

#displaying temperature curve
plt.figure()
plt.plot(np.linspace(0,h,N),result[0,N:],color='red')
plt.ylabel('temperature [K]')
plt.xlabel('depth [m]')
plt.show()
```

4.1.4 code de test MCMC (MOLO_45_test_MCMC.py)

```
from MOLO_45 import test_MCMC
import numpy as np
import matplotlib.pyplot as plt
import time

#number of main cells
N = 10
#ground layer height [m]
h = 0.4
#constant boundary parameters
Hriv = np.array([0.05,0.05],dtype=np.double)
Haq = np.array([0,0],dtype=np.double)
Triv = np.array([300,300],dtype=np.double)
Taq = np.array([290,290],dtype=np.double)
#number of samples provided for boundary parameters
Ntimes = 2
#sampling start
t0 = 0
#sampling end
tf = 3600
#sampling step
dt = tf
#number of layers
Nl = 1
#a layer 1 meter thick
```

```
borders = np.array([1],dtype=np.double)
#three main measures at the depths of 10, 20 and 30 cm
depths_meas = np.array([0.1,0.2,0.3],dtype=np.double)
#number of MCMC chains
nb_chains = 5
#nb of walk steps
nb_iter = 1000
#predictions after a week
times = np.array([7*24*3600],dtype = np.double)
t0_solve = 0
#constant parameters for random generators of DREAMS
cn = 0.01
c = 0.1
#number of ordered pairs used to compute dX in DREAMS
np_opairs = 2
#number of crossover probabilities
nCR = 10
#physical parameters ranges
range_L = np.array([2,4],dtype = np.double)
range_C = 10**(6)*np.array([3,5],dtype = np.double)
range_K = np.array([10**(-8),10**(-4)],dtype = np.double)
range_S = np.array([0.1,0.3],dtype = np.double)
range_sigma = np.array([0.01,0.4],dtype = np.double)
#error control parameters if autostep
atol_H = 10**(-6)
rtol_H = 10**(-4)
atol_T = 10**(-6)
rtol_T = 10**(-4)

#Steady State solution
def SS(h,depths_meas,Hriv,Haq,Triv,Taq,
       k_ref=10**(-5),
       lambda_ref=3,
       c_ref=4*10**(6),
       s_ref=0.2,
       cw = 4180.,
       rhow = 1000.):
    H_exact = (Haq[0]-Hriv[0])/h*depths_meas+Hriv[0]
    gamma = -cw*rhow*k_ref*(Haq[0]-Hriv[0])/(h*lambda_ref)
    alpha = (Triv[0]-Taq[0])/(1-np.exp(gamma*h))
    beta = (Taq[1]-Triv[0]*np.exp(gamma*h))/(1-np.exp(gamma*h))
    Tmeas = np.exp(gamma*depths_meas)*alpha+beta
    Tmeas = Tmeas.reshape(1,len(Tmeas))
    Tmeas = Tmeas.astype(np.double)
    return Tmeas
#simulated measures
meas = SS(h,depths_meas,Hriv,Haq,Triv,Taq)

#MCMC at work
t0 = time.time()
past = test_MCMC(N,h,Hriv,Haq,Triv,Taq,Ntimes,dt,t0,tf,Nl,borders,depths_meas,
    nb_chains,nb_iter,0,meas,times,t0_solve,cn,c,np_opairs,nCR,range_L,range_C,range_K
    ,range_S,range_sigma,atol_H,rtol_H,atol_T,rtol_T,autostep=False)
t1 = time.time()
print(f'execution time  = {t1-t0} s')

#computing the average permeability encountered in the random walk
Kpast = np.empty(nb_chains*nb_iter)
for i in range(nb_iter):
    for j in range(nb_chains):
        Kpast[i*nb_chains+j] = past[i,5*j+3]
print(f'average K = {np.average(Kpast)} m^2')

#displaying logarithmic permeability histogram
plt.hist(-np.log10(Kpast),bins=30)
plt.axvline(x=5,color='red')
plt.xlabel('-log10(K)')
plt.ylabel('density')
```

```
locs, _ = plt.yticks()
plt.yticks(locs,np.round(locs/len(Kpast),3))
plt.show()
```

4.2 python : premiers programmes

4.2.1 modèle physique avec slicing numpy (M_slicing_python.py)

```
import numpy as np
import time
import matplotlib.pyplot as plt
import scipy.integrate

##Discretization

def compute_U(D:np.array,
              E:np.array,
              W:np.array):
    aux = D*E
    sub_diag = aux[1:-1]
    diag = -aux[:-1]-aux[1:]
    sup_diag = aux[1:-1]

    return W*(np.diag(sub_diag,k=-1)+np.diag(diag,k=0)+np.diag(sup_diag,k=1))

def compute_V(D:np.array,
              E:np.array,
              W:np.array,
              J:np.array):
    aux = D*E
    sub_diag = J[2:-1]*aux[1:-1]
    diag = J[:-2]*aux[:-1]-J[2:]*aux[1:]
    sup_diag = -J[1:-2]*aux[1:-1]

    return W*(np.diag(sub_diag,k=-1)+np.diag(diag,k=0)+np.diag(sup_diag,k=1))

def compute_B_U_coefs(D:np.array,
                      E:np.array,
                      W:np.array):
    return W[0,0]*D[0]*E[0], W[-1,0]*D[-1]*E[-1]

def compute_B_V_coefs(D:np.array,
                      E:np.array,
                      W:np.array,
                      J:np.array):
    return J[1]*W[0,0]*D[0]*E[0], -J[-2]*W[-1,0]*D[-1]*E[-1]

def build_geometry(N:int,
                   sizes:np.array,
                   cw=4180.,
                   rhow=1000.):

    if len(sizes)!=N+2:
        print('non valid number of given cells sizes')
    else:
        D_U = 2*(sizes[:-1] + sizes[1:])**(-1)
        D_V = cw*rhow*(sizes[:-1] + sizes[1:])**(-1)
        e = np.zeros((N,2))
        e[0,0] = 1.
        e[N-1,1] = 1.
        return sizes, D_U, D_V, e

def build_W(C,J):
    return ((C*J[1:-1])**(-1)).reshape(len(C),1)

def build_E_V(K:np.array,
              D:np.array,
              H:np.array):
```

```
    return -K*D*(H[1:]-H[:-1])

def build_B_U(coefs_B_U:np.array,
              e:np.array,
              Yb:np.array):
    return coefs_B_U[0]*Yb[0]*e[:,0] + coefs_B_U[1]*Yb[1]*e[:,1]

def build_B_V(D:np.array,
              E:np.array,
              W:np.array,
              J:np.array,
              e:np.array,
              Yb:np.array):
    coef0, coef1 = compute_B_V_coefs(D,E,W,J)
    return coef0*Yb[0]*e[:,0] + coef1*Yb[1]*e[:,1]

def args_for_F_all(N:int,
               sizes:np.array,
               C_T:np.array,
               C_H:np.array,
               Lambda:np.array,
               K:np.array):

    J, D_U, D_V, e = build_geometry(N,sizes)

    W_T = build_W(C_T,J)
    W_H = build_W(C_H,J)

    U_T = compute_U(D_U,Lambda,W_T)
    U_H = compute_U(D_U,K,W_H)

    coefs_B_U_T = compute_B_U_coefs(D_U,Lambda,W_T)
    coefs_B_U_H = compute_B_U_coefs(D_U,K,W_H)

    return U_T, U_H, coefs_B_U_T, coefs_B_U_H, D_V, W_T

def F_all(U_T:np.array,
          U_H:np.array,
          coefs_B_U_T:np.array,
          coefs_B_U_H:np.array,
          D_U:np.array,
          D_V:np.array,
          W_T:np.array,
          K:np.array,
          J:np.array,
          e:np.array,
          Tb,
          Hb):

    def f(t,X):

        T = X[:len(K)-1]
        H = X[len(K)-1:]
        H_b = Hb(t)
        forHz = np.concatenate((np.array([H_b[0]]),H,np.array([H_b[1]])))

        E_V_T = build_E_V(K,D_U,forHz)

        V_T = compute_V(D_V,E_V_T,W_T,J)

        B_V_T = build_B_V(D_V,E_V_T,W_T,J,e,Tb(t))

        B_U_T = build_B_U(coefs_B_U_T,e,Tb(t))
        B_U_H = build_B_U(coefs_B_U_H,e,Hb(t))

        A_T = U_T + V_T
        B_T = B_U_T + B_V_T

```

```
        A_H = U_H
        B_H = B_U_H

        Tdot = np.dot(A_T,T) + B_T
        Hdot = np.dot(A_H,H) + B_H

        return np.concatenate((Tdot,Hdot))

    return f

##ODE

def RK5coefs(f,
             y:np.array,
             t:np.float64,
             h:np.float64):
    k = np.empty((len(y),6))
    k[:,0] = h*f(t,y)
    k[:,1] = h*f(t+1/4*h,y+1/4*k[:,0])
    k[:,2] = h*f(t+3/8*h,y+3/32*k[:,0]+9/32*k[:,1])
    k[:,3] = h*f(t+12/13*h,y+1932/2197*k[:,0]-7200/2197*k[:,1]+7296/2197*k[:,2])
    k[:,4] = h*f(t+h,y+439/216*k[:,0]-8*k[:,1]+3680/513*k[:,2]-845/4104*k[:,3])
    k[:,5] = h*f(t+1/2*h,y-8/27*k[:,0]+2*k[:,1]-3544/2565*k[:,2]+1859/4104*k
    [:,3]-11/40*k[:,4])
    return k

def step_scaler(k:np.array,
                tol:np.float64,
                RK4_cok:np.array,
                RK5_cok:np.array):
    twoN = len(k)
    s_T = (tol/(2*np.linalg.norm(np.dot(k[:twoN//2,:],RK5_cok-RK4_cok))))**(1/4)
    s_H = (tol/(2*np.linalg.norm(np.dot(k[twoN//2:,:],RK5_cok-RK4_cok))))**(1/4)
    return min(s_T,s_H)

def RKF45(f,
          y0:np.array,
          t0:np.float64,
          tf:np.float64,
          h:np.float64,
          tol:np.float64,
          RK4_cok:np.array,
          RK5_cok:np.array,
          RK5_cok_vec:np.array):

    t = t0
    while t<tf:

        k0 = RK5coefs(f,y0,t,h)

        h = h*step_scaler(k0,tol,RK4_cok,RK5_cok)
        if t+h>tf:
            h = tf-t
        t = t + h

        k1 = RK5coefs(f,y0,t,h)
        y0 = y0 + np.dot(k1,RK5_cok_vec)

    return y0

def timefunction(C:np.array,
                 dt:float,
                 t0:float,
                 tf:float):
    def f(t:float):
        if t<t0:
            return C[0]
        elif t>=tf:
```

```
            return C[-1]
        else:
            n,e = np.divmod(t-t0,dt)
            n = int(n)
            return (1-e/dt)*C[n] + e/dt*C[n+1]
    return f

def solver(f,
           y0:np.array,
           t0:float,
           tol:float,
           fsample,
           times:np.array,
           h=1,
           method='RKF45'):

    samples = np.empty((len(fsample(y0)),len(times)))

    if method=='RKF45':
        RK4_cok_vec = np.array([25/216,0,1408/2565,2197/4101,-1/5,0],dtype=np.float64
    )
        RK5_cok_vec = np.array([16/135,0,6656/12825,28561/56430,-9/50,2/55],dtype=np.
    float64)
        RK4_cok = RK4_cok_vec.reshape(6,1)
        RK5_cok = RK5_cok_vec.reshape(6,1)
        for i,t in enumerate(times):
            y1 = RKF45(f,y0,t0,t,h,tol,RK4_cok,RK5_cok,RK5_cok_vec)
            y0 = y1
            t0 = t
            samples[:,i] = fsample(y0)
        return samples
    else:
        print('unkown method')

##Test

def build_depths_and_sizes(h,N):
    l = h/(N+1)
    sizes = l*np.ones(N+2)
    Zmeas = np.empty(N)
    Zmeas[0] = (sizes[0] + sizes[1])/2
    for i in range(1,N):
        Zmeas[i] = Zmeas[i-1] + (sizes[i] + sizes[i+1])/2
    return Zmeas, sizes

def steady_state_test(t0,tf,Tss,Hss,h,T0,H0,N,K,Lambda,C,S,tol):

    Tb = timefunction(Tss,tf,t0,tf)
    Hb = timefunction(Hss,tf,t0,tf)

    Zmeas, sizes = build_depths_and_sizes(h,N)

    J, D_U, D_V, e = build_geometry(N,sizes)
    U_T, U_H, coefs_B_U_T, coefs_B_U_H, D_V, W_T = args_for_F_all(N,sizes,C,S,Lambda,
    K)
    f = F_all(U_T,U_H,coefs_B_U_T,coefs_B_U_H,D_U,D_V,W_T,K,J,e,Tb,Hb)

    y0 = np.concatenate((T0,H0))
    def fsample(y:np.array):
        return y

    return solver(f,y0,t0,tol,fsample,np.array([tf]),h=tf/10)

def homgeneous_test(tf,Tss,Hss,h,N,T0,H0,tol,
                    k_ref=10**(-5),
                    lambda_ref=3,
                    c_ref=4*10**(6),
                    s_ref=0.2,
```

```
                    cw = 4180.,
                    rhow = 1000.,
                    display=False):
    t0 = 0
    K = k_ref*np.ones(N+1)
    Lambda = lambda_ref*np.ones(N+1)
    C = c_ref*np.ones(N)
    S = s_ref*np.ones(N)

    result = steady_state_test(t0,tf,np.array([Tss,Tss]),np.array([Hss,Hss]),h,T0,H0,
    N,K,Lambda,C,S,tol)

    T_res = result[:N]
    H_res = result[N:]

    Zmeas, _ = build_depths_and_sizes(h,N)

    H_exact = (Hss[1]-Hss[0])/h*Zmeas+Hss[0]

    gamma = -cw*rhow*k_ref*(Hss[1]-Hss[0])/(h*lambda_ref)
    alpha = (Tss[0]-Tss[1])/(1-np.exp(gamma*h))
    beta = (Tss[1]-Tss[0]*np.exp(gamma*h))/(1-np.exp(gamma*h))
    T_exact = np.exp(gamma*Zmeas)*alpha+beta

    if display:
        plt.figure()
        plt.scatter(Zmeas,T_res,label='computed')
        plt.plot(Zmeas,T_exact,label='exact')
        plt.xlabel('depth [m]')
        plt.ylabel('temperature [K]')
        plt.legend()
        plt.show()

        plt.figure()
        plt.scatter(Zmeas,H_res,label='computed')
        plt.plot(Zmeas,H_exact,label='exact')
        plt.xlabel('depth [m]')
        plt.ylabel('hydraulic charge [H]')
        plt.legend()
        plt.show()
    return np.max(abs(T_res.reshape(T_exact.shape)-T_exact)), np.max(abs(H_res.
    reshape(H_exact.shape)-H_exact))

def heterogeneous_test(tf,Tss,Hss,h,N,T0,H0,tol,
                       kz_ref = 10**(-5),
                       k_ref = 10**(-5),
                       lambda_ref=3,
                       c_ref=4*10**(6),
                       s_ref=0.2,
                       cw = 4180.,
                       rhow = 1000.,
                       display=False):
    t0 = 0
    Zmeas, sizes = build_depths_and_sizes(h,N)
    ZK = Zmeas-sizes[0]/2
    ZK = np.concatenate((ZK,np.array([ZK[-1]+sizes[0]/2])))
    K = kz_ref*ZK + k_ref
    Lambda = lambda_ref*np.ones(N+1)
    C = c_ref*np.ones(N)
    S = s_ref*np.ones(N)

    result = steady_state_test(t0,tf,np.array([Tss,Tss]),np.array([Hss,Hss]),h,T0,H0,
    N,K,Lambda,C,S,tol)

    T_res = result[:N]
    H_res = result[N:]

    fK = lambda z : kz_ref*z + k_ref
```

```
    fH = lambda z :(Hss[1]-Hss[0])*(np.log(kz_ref*z+k_ref)-np.log(k_ref))/(np.log(kz_ref*h+k_ref)-np.log(k_ref))+Hss[0]
    fHz = lambda z : kz_ref*(Hss[1]-Hss[0])/((np.log(kz_ref*h+k_ref)-np.log(k_ref))*(kz_ref*z+k_ref))

    H_exact = np.vectorize(fH)(Zmeas)

    T_exact = np.empty(N)
    aux = lambda z : np.exp(-cw*rhow*fK(z)*fHz(z)*z/lambda_ref)

    normalizer = scipy.integrate.quad(aux,0,h)[0]
    for i,z in enumerate(Zmeas):
        T_exact[i] = (Tss[1]-Tss[0])*scipy.integrate.quad(aux,0,z)[0]/normalizer+Tss[0]

    if display:

        plt.figure()
        plt.scatter(Zmeas,T_res,label='computed')
        plt.plot(Zmeas,T_exact,label='exact')
        plt.xlabel('depth [m]')
        plt.ylabel('temperature [K]')
        plt.legend()
        plt.show()

        plt.figure()
        plt.scatter(Zmeas,H_res,label='computed')
        plt.plot(Zmeas,H_exact,label='exact')
        plt.xlabel('depth [m]')
        plt.ylabel('hydraulic charge [H]')
        plt.legend()
        plt.show()

    return np.max(abs(T_res.reshape(T_exact.shape)-T_exact)), np.max(abs(H_res.reshape(H_exact.shape)-H_exact))

def bistratified_test(tf,Tss,Hss,h,N,T0,H0,tol,
                      k1_ref = 10**(-5),
                      k2_ref = 10**(-7),
                      lambda_ref=3,
                      c_ref=4*10**(6),
                      s_ref=0.2,
                      cw = 4180.,
                      rhow = 1000.,
                      display=False):
    t0 = 0
    Zmeas, sizes = build_depths_and_sizes(h,N)
    ZK = Zmeas-sizes[0]/2
    ZK = np.concatenate((ZK,np.array([ZK[-1]+sizes[0]/2])))
    K = np.ones(N+1)
    K[:(N+1)//2] = k1_ref
    K[(N+1)//2] = (k1_ref + k2_ref)/2
    K[(N+1)//2+1:] = k2_ref
    Lambda = lambda_ref*np.ones(N+1)
    C = c_ref*np.ones(N)
    S = s_ref*np.ones(N)

    result = steady_state_test(t0,tf,np.array([Tss,Tss]),np.array([Hss,Hss]),h,T0,H0,N,K,Lambda,C,S,tol)

    T_res = result[:N]
    H_res = result[N:]

    if display:

        plt.figure()
        plt.scatter(Zmeas,T_res,label='computed')
        plt.xlabel('depth [m]')
```

```
        plt.ylabel('temperature [K]')
        plt.legend()
        plt.show()

        plt.figure()
        plt.scatter(Zmeas,H_res,label='computed')
        plt.xlabel('depth [m]')
        plt.ylabel('hydraulic charge [H]')
        plt.legend()
        plt.show()

def to_test(tf,Tss,Hss,h,N,T0,H0,tol,
                    k_ref=10**(-5),
                    lambda_ref=3,
                    c_ref=4*10**(6),
                    s_ref=0.2,
                    cw = 4180.,
                    rhow = 1000.,
                    display=False):
    t0 = 0
    K = k_ref*np.ones(N+1)
    Lambda = lambda_ref*np.ones(N+1)
    C = c_ref*np.ones(N)
    S = s_ref*np.ones(N)

    Tb = timefunction(np.array([Tss,Tss]),tf,t0,tf)
    Hb = timefunction(np.array([Hss,Hss]),tf,t0,tf)

    y0 = np.concatenate((T0,H0))

    Zmeas, sizes = build_depths_and_sizes(h,N)

    J, D_U, D_V, e = build_geometry(N,sizes)
    U_T, U_H, coefs_B_U_T, coefs_B_U_H, D_V, W_T = args_for_F_all(N,sizes,C,S,Lambda,
    K)
    f = F_all(U_T,U_H,coefs_B_U_T,coefs_B_U_H,D_U,D_V,W_T,K,J,e,Tb,Hb)

    return y0, f

tf = 1*24*3600
Tss = np.array([290.,300.])
Hss = np.array([0.05,0.])
htot = 0.4
N = 22
T0 = 295*np.ones(N)
H0 = 0.025*np.ones(N)
tol = 10**(-1)

errors_heterogeneous = heterogeneous_test(tf,Tss,Hss,htot,N,T0,H0,tol,display=True)
```

4.2.2 MCMC (MCMC_python.py)

```
import numpy as np
import matplotlib.pyplot as plt

##Elementary functions

def generate_uncorrelated_multivariate_normal(sigmas:np.array):
    Y = np.empty(len(sigmas))
    for i in range(len(sigmas)):
        Y[i] = np.random.normal(loc=0,scale=sigmas[i])
    return Y

def is_valid(D:np.array,
             Y:np.array):
    return np.all((D[:,0]<=Y)*(Y<=D[:,1]))

def energy(X:np.array,
```

```
          Ymeas:np.array,
          F,
          sigma:float):

    if sigma > 0:
        return np.sum((Ymeas-F(X))**2)/(2*sigma**2)
    else:
        sigma = X[-1]
        logsigma = len(Ymeas)*np.log(X[-1])
        return np.sum((Ymeas-F(X[:-1]))**2)/(2*sigma**2)+logsigma

def acceptancy(X0:np.array,
               X1:np.array,
               D:np.array,
               Ymeas:np.array,
               F,
               sigma:float):

    if is_valid(D,X1):

        eX0 = energy(X0,Ymeas,F,sigma)
        eX1 = energy(X1,Ymeas,F,sigma)

        if eX1<=eX0:
            return 1
        else:
            return np.exp(eX0-eX1)

    return 0

##DREAMS

def walk(Xall,D,Ymeas,sigma,F,nb_iter,pi,cn,c,Psize,gamma=2.38/(2)**(0.5)):

    past = np.empty((nb_iter,len(Xall)*len(Xall[0])))

    indices = make_indices(len(Xall[0]),pi)
    alpha = make_alpha(cn)
    beta = make_beta(c,Psize,gamma)
    pairs = make_pairs(len(Xall),Psize)

    for i in range(nb_iter):

        chain_id = i%len(Xall)
        Xp = propose(Xall,chain_id,D,Ymeas,F,alpha,beta,indices,pairs)
        p_accept = acceptancy(Xall[chain_id],Xp,D,Ymeas,F,sigma)

        if (np.random.random()<p_accept):
            Xall[chain_id] = Xp
            past[i,:] = Xall.reshape(len(Xall)*len(Xall[0]))
        else:
            past[i,:] = Xall.reshape(len(Xall)*len(Xall[0]))

    return past

def propose(Xall,chain_id,D,Ymeas,F,alpha,beta,indices,pairs):

    I = indices()
    a = alpha(len(I))
    b = beta(len(I))
    P = pairs(chain_id)
    SP = pairs_sum(Xall,P)

    dX = np.zeros(len(Xall[chain_id]))

    for i,index in enumerate(I):

        dX[index] = a[i] + b[i]*SP[index]
```

```
    Xp = D[:,0] + np.mod(Xall[chain_id] + dX - D[:,0],D[:,1]-D[:,0])

    return Xp

def make_indices(d,pi):

    def indices():

        sizes = np.random.multinomial(1,pi)

        for size in range(len(pi)):
            if sizes[size]==1:
                m = (size+1)/len(pi)
                break

        I = []
        for i in range(len(pi)):
            if np.random.random()<m:
                I.append(i)
        if I == []:
            I.append(int(np.random.random()*len(pi)))

        return np.array(I)

    return indices

def make_alpha(cn):

    def alpha(Isize):
        return np.random.normal(loc=0,scale=cn,size=Isize)

    return alpha

def make_beta(c,Psize,gamma):

    def beta(Isize):

        inf = (1-c)*gamma/(Psize*Isize)**(0.5)
        sup = (1+c)*gamma/(Psize*Isize)**(0.5)

        return np.random.uniform(low=inf,high=sup,size=Isize)

    return beta

def make_pairs(N,nb_pairs):

    tab = np.empty((N*(N-1)//2,2))
    for i in range(N):
        for j in range(i):
            tab[i*(i-1)//2+j] = np.array([i, j])

    def pairs(id_chain):

        choices = np.random.randint(0,(N-1)*(N-2)//2,size=nb_pairs)

        couples = np.empty((nb_pairs,2))

        for i in range(nb_pairs):

            a,b = tab[choices[i]]

            if a >= id_chain:
                a = a + 1
            if b >= id_chain:
                b = b + 1

            if np.random.random()<0.5:
```

```
153
154                 couples[i,0] = a
155                 couples[i,1] = b
156
157             else:
158                 couples[i,0] = b
159                 couples[i,1] = a
160
161         return couples
162
163     return pairs
164
165 def pairs_sum(Xall,P):
166
167     S = np.zeros(len(Xall[0]))
168
169     for p in range(len(P)):
170
171         S = S + Xall[int(P[p,0])]-Xall[int(P[p,1])]
172
173     return S
174
175 def init(D):
176     X = np.empty(len(D))
177     for i in range(len(D)):
178         X[i] = np.random.uniform(D[i][0],D[i][1])
179     return X
180
181 ##Test
182
183 F = lambda x : x**2
184 D = np.array([[-2,2]])
185 sigma = 0.1
186 Ymeas = np.array([1]) + np.random.normal(loc=0,scale=sigma,size=1)
187
188 N = 30
189 cn = 0.01
190 c = 0.1
191 Psize = 10
192 pi = np.ones(1)
193 Xall = np.array([init(D) for _ in range(N)])
194 nb_iter = 1000
195
196 past = walk(Xall,D,Ymeas,sigma,F,nb_iter,pi,cn,c,Psize)
197
198 past = past.reshape((nb_iter*N))
199
200 plt.hist(past,bins=100,density=True)
201 plt.axvline(x=1,color='red',label='expected solutions')
202 plt.axvline(x=-1,color='red')
203 plt.xlabel('x')
204 plt.ylabel('density')
205 plt.legend()
206 plt.show()
```