Code C++ pour la calibration FMP de codes de calcul

Juillet 2022

Table des matières

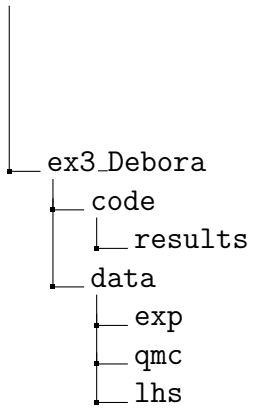
1 Résumé

Le présent document est la documentation du code fourni dans l'archive `archive`. Ce code permet l'application de la méthode FMP pour la calibration de codes de calcul avec erreur de modèle. La méthode est écrite dans les fichiers `densities.h` et `densities.cpp`. Les diverses fonctions composant la méthode sont ensuite utilisées dans le fichier `main.cpp`. Ce fichier inclut d'autres tâches afférentes à la calibration (construction d'un surrogate du code de calcul, construction d'un surrogate pour FMP, post-traitement des résultats, recherche du Maximum A Posteriori des paramètres, prédictions, etc.), qui sont présentes ou non selon les exemples traités.

2 Contenu de l'archive transmise

Voici l'architecture des répertoires contenus dans l'archive :

```
transfert.tar/
├── StochTk++
└── Code_calibration
    ├── ex1_GSobol
    │   └── code
    │       └── results
    └── ex2_MITB
        ├── code
        │   └── results
        └── data
```



Chacun des répertoires `code` contient le code C++ : `main.cpp`, `densities.cpp`, `densities.h`. Les fichiers *densities* sont identiques entre les 3 exemples. Les résultats de la calibration sont sauvegardés dans les répertoires `results` et leurs sous-répertoires. L'exemple Debora fait intervenir des fichiers de résultats de simulations CFD et des résultats expérimentaux dans le répertoire `data`, qui sont lus par le code.

3 Installation du code

Le code repose sur trois bibliothèques C++ : Eigen, NLopt, (publiques) et StochTk++ (privée). Il est nécessaire d'installer ces bibliothèques et de les compiler (pour NLopt et StochTk++) avant de pouvoir lancer le code.

Dans ce tutoriel d'installation, nous allons installer les bibliothèques dans le répertoire `/install`. Si vous souhaitez installer dans un autre répertoire, il sera nécessaire de modifier certains fichiers (indiqués le moment venu).

Les instructions d'installation sont données en commandes Bash, donc adaptées pour les systèmes Unix ou MacOS.

Commencer par créer le répertoire d'installation :

```
$ mkdir ~/install
```

3.1 Installation Eigen

Il faut télécharger Eigen au lien suivant : `http://eigen.tuxfamily.org/index.php?title=Main_Page#Download`. Ce tutoriel fonctionne avec la version 3.4.0 d'Eigen. Pour utiliser une autre version, il faudra modifier certains fichiers (indiqués le moment venu).

Télécharger l'archive, l'extraire, et la déplacer dans le répertoire d'installation :

```
$ mv eigen-3.4.0 ~/install/eigen-3.4.0
```

3.2 Installation NLopt

L'installation requiert les outils cmake et make. On réalise ce tutoriel avec la version 2.7.1. Télécharger la bibliothèque à l'adresse suivante : `https://nlopt.readthedocs.io/en/latest/#download-and-installation`.

Extraire l'archive, se déplacer dans le répertoire `nlopt-2.7.1`, puis exécuter les commandes suivantes :

```
$ mkdir build
$ cd build
$ cmake -DCMAKE_INSTALL_PREFIX=$HOME/install ..
$ make
$ sudo make install
```

Il est également nécessaire d'indiquer le chemin de la bibliothèque par la commande suivante, sous Linux :

```
$ export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:~/install/lib
```

ou sous MacOS :

```
$ export DYLD_LIBRARY_PATH=${DYLD_LIBRARY_PATH}:~/install/lib
```

Si le bon export n'est pas fait, le code ne pourra pas s'exécuter. Pour ma part, la commande export est recopiée dans mon `/.bashrc` pour être valide à chaque terminal.

3.3 Installation StochTk++

La bibliothèque StochTk++ est fournie dans l'archive `transfert.tar`. Déplacer le répertoire dans le répertoire d'installation :

```
$ mv StochTk++ ~/install/StochTk++
$ cd ~/install/StochTk++
```

Si le répertoire d'installation est différent, ou la version d'Eigen n'est pas la même, modifier les trois premières lignes du fichier `make.inc`.

Après les modifications, compiler la bibliothèque par :

```
$ ./do_libs
```

3.4 Compilation du code

Chaque exemple est compilé avec la commande `make`, effectuée dans le répertoire `code` correspondant. Dans chacun de ces répertoires se trouve un fichier `makefile`, nécessaire à la compilation. Les `makefile` des trois exemples sont identiques. Dans ces fichiers figurent en dur le chemin d'installation des bibliothèques et la version d'eigen. Si on a choisi un répertoire d'installation ou une version d'Eigen différente, modifier les `makefile` aux variables CPPFLAGS et LDFLAGS. Sinon, aucune modification n'est nécessaire.

Une fois que le `makefile` est correct, il suffit de se rendre par exemple dans `ex1_GSobol/code`, et de faire :

```
$ make
$ ./main.exe
```

4 Description des classes C++ de densities.h

Le code C++ permettant la calibration FMP et quelques fonctions annexes (sampling MCMC, construction de surrogates pour les hyperparamètres optimaux, prédictions a posteriori), est donné dans le header `densities.h`, codé dans `densities.cpp`.

Le fichier `densities.h` comporte trois classes :

- **DoE**, qui permet la construction d'un Design of Experiments dans l'espace des paramètres, selon 3 méthodes : Uniforme, LHS, ou QMC.
- **Density**, construite à partir d'un objet de type **DoE**, qui contient les méthodes de prédiction, de calcul de la vraisemblance, etc. Elle a été créee pour permettre la calibration avec une méthode quelconque (KOH, sans erreur de modèle, etc.).
- **DensityOpt**, construite à partir d'un objet de type **Density**. Cette classe contient tous les éléments nécessaires à la calibration FMP : calcul d'hyperparamètres optimaux, et construction des surrogates pour ces hyperparamètres.

Au début du fichier, quelques fonctions communes (appelées utility) sont codées par convénience. Elles sont décrites dans la section 4.1. On décrit ensuite les méthodes principales des trois classes de `densities.h`.

Le fichier comporte également des fonctions utility

4.1 Fonctions utility

— Les fonctions **ReadVector** et **WriteVector** permettent de lire et écrire des objets dans des fichiers, pour sauvegarde et écriture de résultats.
— Les **optroutine** sont utilisées dans les classes. Elles contiennent des instructions de la bibliothèque NLopt pour faire de l'optimisation.
— les **optfunc** sont diverses fonctions à optimiser. On trouve la maximisation Kennedy-O'Hagan, la maximisation FMP, la maximisation FMP avec calcul de gradient, et l'optimisation des hyperparamètres d'un processus Gaussien quelconque.
— la fonction **OptimizeGPBis** permet d'optimiser les hyperparamètres d'un objet de la classe GP (provenant de la bibliothèque StochTk++).
— La fonction **Run_MCMC** permet de faire tourner une chaîne MCMC. Elle renvoie la liste des étapes visitées. La fonction **Selfcor_diagnosis** permet de calculer l'autocorrélation d'une liste d'étapes d'une chaîne.

4.2 Classe DoE

Il faut choisir l'une des 3 manières suivantes pour créer un DoE :
— **Constructeur (VectorXd lb, VectorXd ub, int** n**)** : DoE uniforme. n est le nombre de points par dimension. Le DoE comportera au total $n \times \dim(\text{lb})$ points.
— **Constructeur (VectorXd lb, VectorXd ub, int npts, default_random_engine generator)** : DoE LHS. npts est le nombre de points dans le DoE.
— **Constructeur (VectorXd lb, VectorXd ub, int npts, int first_element)** : DoE QMC par suite de Halton. npts est le nombre de points dans le DoE. firest_element règle le premier élément de la suite utilisé pour la génération pseudo-aléatoire, il est conseillé de prendre la valeur 1.

4.3 Classe Density

Un objet de la classe Density est construit soit à partir d'un DoE, soit par copie d'un autre objet Density. Après la construction de l'objet, il est nécessaire d'appeler les méthodes suivantes :
— **SetObservations** : donner les données expérimentales pour la calibration.
— **SetFModel** : le modèle f à calibrer.

— **SetZKernel** : le noyau de l'erreur de modèle z.
— **SetZPriorMean** : la moyene a priori de l'erreur de modèle
— **SetLogPriorPars** : le log [1] du prior des paramètres du modèle.
— **SetLogPriorHpars** : le log du prior des hyperparamètres de l'erreur de modèle [2].
— **SetHparsBounds** : les bornes sup et inf des hyperparamètres.

Il faut appeler les méthodes suivantes si il y a présence d'erreur expérimentale dans le problème :

— **SetFixedOutputerr** ou **SetLearnedOutputerr** : appeler soit l'une, soit l'autre. Permettent de considérer une erreur de mesure, respectivement avec une valeur connue ou apprise pendant la calibration. Dans le second cas, il faut préciser l'indice correspondant dans le vecteur d'hyperparamètres. Notons que la valeur fixée doit être donnée en échelle log. Dans le code actuel, on considère la même erreur de mesure sur l'ensemble des observations. Si aucune de ces méthodes n'est appelée, on considère 0 erreur de mesure.
— **SetFixedInputerr** ou **SetLearnedInputerr** : fonctionnent de la même manière que les deux précédentes, avec de l'erreur sur le x expérimental, et non sur la quantité mesurée. Requièrent d'estimer au préalable les dérivées du "true process" aux points d'observations (voir exemple 6).

La méthode **HparsKOH** permet de calculer l'estimateur des hyperparamètres défini par Kennedy et O'Hagan.

Pour effectuer des prédictions de la quantité d'intérêt, il est nécessaire d'obtenir un échantillon de la densité a posteriori des paramètres par MCMC (voir les trois exemples pour l'usage de la fonction **Run_MCMC**), et de les inclure dans l'objet Density par les méthodes **SetNewSamples** et **SetNewHparsOfSamples** (voir exemples 6 et 7). Les prédictions s'effectuent avec les méthodes **WritePredictions** et **WriteSamplesFandZ**.

Pour utiliser **WritePredictions**, on spécifie le chemin d'un fichier de sortie dans lequel les prédictions seront écrites. Les lignes correspondent à chaque point x pour lequel on veut faire une prédiction, et les colonnes correspondent respectivement à : valeur de x - pred. moyenne de $f + z$ - std. de $f + z$ - pred. moyenne de f - quantile à 2.5% de f - quantile à 97.5% de f.

1. Dans tout le document, et dans le code, log désigne le logarithme naturel.

2. On regroupe dans un seul vecteur les hyperparamètres du noyau de l'erreur de modèle, les hyperparamètres de la moyenne à priori de l'erreur de modèle, et les erreurs d'observation. La densité a priori doit donc être formulée pour l'ensemble de ces variables.

Pour utiliser **WriteSamplesFandZ**, il faut également donner le chemin d'un fichier d'écriture. Dans cete méthode, on réalise le tirage de quelques valeurs de $\boldsymbol{\theta}$ parmi l'échantillon à posteriori. On calcule ensuite les prédictions du modèle f, et d'un tirage de z par valeur de $\boldsymbol{\theta}$, et on affiche le tout dans un fichier.

Les méthodes qui ne sont pas mentionnées dans ce document sont principalement dédiées à de l'usage interne (type **Gamma** pour le calcul de la matrice de covariance, **loglikelihood_theta** pour la log-vraisemblance), à l'accès externe de quantités (méthodes **GetObs**,...), ou sont facilement lisibles.

4.4 Classe DensityOpt

Comporte le nécessaire à la calibration FMP (Opt étant l'ancienne désignation de FMP). Cette classe hérite de la classe Density, donc tout objet DensityOpt peut utiliser les méthodes Density. Un objet de cette classe est nécessairement construit à partir d'un objet de la classe Density, ce qui réalise une copie de tous les éléments nécessaires à la calibration (modèle f, observations, etc.).

Il est possible de calculer les hyperparamètres optimaux FMP de deux manières, sans avoir recours au gradient de la logvraisemblance (**HparsOpt**), ou avec le gradient (**HparsOpt_withgrad**). La seconde méthode requiert au préalable de spécifier les dérivées du noyau de z et du prior des hyperparamètres, avec un appel aux méthodes **SetZKernelGrads** et **SetLogpriorHparsGrads**. Dans les exemples de ce tutoriel, nous utiliserons la version sans gradient.

Nous avons proposé dans [2] la construction de processus gaussiens (notés HGPs) pour les hyperparamètres FMP, pour réduire le coût de la calibration. Pour mettre cela en pratique, il faut au préalable calculer une base de données de $\boldsymbol{\theta}$, et les hyperparamètres optimaux sur cette base de donnée (par la méthode **HparsOpt**). Cela se fait hors de l'objet DensityOpt. Une fois cette base de données obtenue, elle est fournie à l'objet DensityOpt en utilisant la méthode **BuildHGPs**, dans laquelle il faut également spécifier un noyau pour les HGPs, et les dérivées de ce noyau. Ensuite, on utilise la méthode **OptimizeHGPs** pour estimer les hyperparamètres de ces HGPs par maximum de vraisemblance. Ensuite, on peut utiliser **EvaluateHparOpt** pour évaluer la prédiction moyenne des HGPs, ce qui fournit les hyperparamètres FMP plus rapidement que **HparsOpt**. A noter qu'il est également possible de réaliser des tirages des hyperparamètres FMP, plutôt que des prédictions

moyennes, par la méthode **SampleHparsOpt**.

La construction des HGPs et leur utilisation pour calibrer, sont présentées dans l'exemple GSobol (section 5).

5 Exemple 1 : GSobol

5.1 Description

Cet exemple se réfère à l'exemple numérique de [2], la calibration de la fonction GSobol $\mathrm{G}(x, \boldsymbol{\theta})$:

$$\mathrm{G}(\mathbf{x}, \boldsymbol{\theta}) = \prod_{i=1}^{6} \frac{|4x_i - 2| + \theta_i}{1 + \theta_i}, \tag{1}$$

à partir de 20 observations provenant de $\mathrm{G}(x, \boldsymbol{\theta}^{true})$, avec $\boldsymbol{\theta}^{true} = (0.55, 0.80, 0.3, 0.4, 0.6, 0.9)$.

Le `main.cpp` donné permet d'appliquer les 5 méthodes de calibration proposées dans [2] : FMP, HP-AS, HP-LHS, LL-AS, LL-LHS. Pour chaque méthode, on tire un échantillon de la densité à posteriori des paramètres par MCMC, on calcule la moyenne à posteriori ainsi que la matrice de covariance. On propose également de calculer le Maximum A Posteriori des paramètres. Enfin, on calcule la posterior-averaged log-likelihood error, comme dans l'article 2, à partir d'un échantillon FMP de référence (`reference_sample.gnu` dans le répertoire `code`).

5.2 Fichiers

Le fichier `sample_reference.gnu` contient l'échantillon de $\boldsymbol{\theta}$ issu de la calibration FMP. Il sert à calculer la posterior-averaged log-likelihood error.

Le répertoire `results` contient cinq répertoires, un pour chaque technique de calibration. Après exécution du code, chacun de ces répertoires contiendra :

— `samples.gnu`, l'échantillon de $\boldsymbol{\theta}$ provenant de la MCMC,
— `map.gnu`, contient le MAP, moyenne, et covariance a posteriori de $\boldsymbol{\theta}$,
— `autocor.gnu`, contient la corrélation des étapes de la chaîne,
— `score.gnu`, contient la posterior-averaged log-likelihood error.

5.3 main.cpp

On décrit maintenant le contenu du `main.cpp`, le jeu de données permettant l'exécution du code.

Les fonctions définies hors du main se rapportent aux différents noyaux, pour z ou les hGPs (**Kernel...**), aux priors, au calcul de la posterior-averaged log-likelihood error (**evaluate_hgp_surrogate** et **evaluate_ll_surrogate**), ou au calcul du MAP (**optfunc_MAP**, etc.).

Dans le main, on commence par définir les bornes des paramètres et hyperparamètres (lignes 309 à 333). Ensuite, on tire les observations nécessaires à la calibration et on les met dans le bon format (lignes 336-350). Ensuite, on construit l'objet Density. Notons qu'on se sert beaucoup des lambda functions pour leur flexibilité : c'est le cas des variables **lambda_model** et **lambda_priormean**. A partir de la ligne 382, on chosit les paramètres de la MCMC de calibration, ainsi que les bornes des hyperparamètres pour les processus gaussiens (lignes 390-423). On règle également la sensibilité des optimisations par le critère de tolérance relative sur la fonction objectif (lignes 425-427). Dans les lignes 430-443, on lit l'échantillon de référence pour le calcul de l'erreur.

La mise en oeuvre des méthodes de calibration commence à partir de la ligne 445. On a séparé les cinq méthodes par des accolades : HP-AS (l. 447), HP-LHS (l.598), LL-LHS (l.677), LL-AS (l.752), FMP (l.893). Je détaille plus précisément la méthode HP-AS.

On commence l. 447 par déterminer les paramètres de l'algorithme (voir commentaires sur le main). Ensuite l.464, on créé le DoE initial, par LHS. Ensuite l. 469, on définit les fonctions lambda qui servent à calculer la logvraisemblance (**get_score**), les prédictions du HGP (**get_hpars_and_var**), à rajouter des points au training set (**add_points**), et à évaluer l'erreur du HGP (**write_performance_hgps**). L. 544, on construit les HGPs avec le DoE initial, on les optimise et on évalue l'erreur. A partir de L. 552, on procède à la construction itérative (adaptive sampling). à partir de L. 563, on procède à la calibration (MCMC à la ligne 578), à l'écriture des résultats et au calcul du MAP.

La méthode HP-LHS procède de la même manière mais plus épurée, car la phase d'adaptive sampling disparaît. La méthode LL-AS procède de la même manière également, à ceci près qu'un seul GP est utilisé pour la logvraisemblance (variable **ll_surrogate**, l. 757). Il est donc optimisé avec la fonction **OptimizeGPBis** (l.837 et 855). La fct lambda **add_points** est aussi sim-

plifiée pour le calcul des poids, car elle requiert simplement un tirage du GP ll_surrogate.

La calibration FMP est la plus simple d'entre toutes, sa spécificité réside dans le fait que la fonction de calcul des hyperparamètres (**get_hpars_opti**) fait intervenir **HparsOpt** directement.

5.4 Résultats

Puisque la génération aléatoire dépend de l'ordinateur utilisé, il ne sera pas possible d'obtenir les mêmes résultats que moi, étant donné que les observations sont bruitées aléatoirement. Voici le MAP obtenu par la méthode FMP dans mon cas : $(0.615199, 1, 0.270912, 0.481236, 0.647954, 0.850912)$. La valeur obtenue ne doit pas être trop éloignée de $\boldsymbol{\theta}^{true}$ (voir description du problème).

Sur un MacBook Pro, le code tourne en une dizaine de minutes.

6 Exemple 2 : MIT Boiling Model

6.1 Description

On réalise la calibration du MIT Boiling Model (voir chap. 5 de [1]). Le modèle comporte 3 paramètres à calibrer. Le modèle MITB est implémenté en C++ dans le `main.cpp`. Les données expérimentales sont lues à partir des fichiers .csv dans le répertoire data. Les conditions expérimentales (pression, débit, etc.) sont codées en dur dans le `main.cpp`.

Je propose de pouvoir régler, par une variable, l'ensemble des expériences que l'on souhaite utiliser pour la calibration (13 expériences en tout). Cela illustre la flexibilité des lambda functions. On traite également de l'incertitude expérimentale en input, par estimation des dérivées du true process via un fit polynomial sur les observations. Sa valeur est fixée et non apprise. Enfin, on effectue des prédictions avec le modèle corrigé. On procède à deux calibrations : KOH, et HP-QMC (qui consiste en FMP, avec des HGPs calculés sur un design QMC).

6.2 Fichiers

Les données d'input sont dans le répertoire `data`, avec un .csv par expérience. Chaque csv contient les points de wall temperature, et la mesure du flux thermique correspondant.

Dans le répertoire `results`, on affiche les observations (`obs"N".gnu` avec N le numéro du cas). Les sous-répertoires `hpqmc` et `koh` contiennent les résultats de chaque calibration : l'échantillon de paramètres, `samples.gnu`, et les prédictions pour chaque cas (`preds"N".gnu`).

6.3 main.cpp

La variable **map_expe** contient les conditions expérimentales. La fonction **ravik_model** contient l'implémentation c++ du MITB. Les fonctions **optfuncfitpol** et **Compute_derivatives_y** permettent de fitter les observations pour estimer les dérivées, pour l'incertitude expérimentale. Les fonctions **GPtoR** et **RtoGP** servent à transférer les paramètres de l'espace réel à $[0, 1]$. Les fonctions **optfuncKOH_pooled** et **HparsKOH_pooled** permettent de calculer les hyperparamètres KOH.

On commence le main par définir les paramètres de la MCMC, des optimisations, et de l'input error (l. 504). On définit ensuite les bornes des paramètres et hyperparamètres (l. 514 à 560). Le vecteur **cases** (l. 563) contient les numéros des expériences qui seront utilisées pour la calibration. On construit un vecteur d'objets Density, nommé **vDens** (l. 583), qui contient une Density par expérience. On construit chaque Density individuellement dans la boucle de la ligne 585.

La calibration KOH commence à partir de la ligne 623. La fonction **HparsKOH_pooled** calcule les hyperparamètres KOH pour l'ensemble des expériences, et lance la MCMC (l. 649) après définition des lambda functions. Ensuite, l'échantillon de la chaîne est trié pour être ajouté à chaque Density par les méthodes **SetNewSamples** (l. 658) et **SetNewhparsofsamples** (l.664), ce qui permet de faire des prédictions sur les cas observés avec la méthode **WritePredictions** (l. 679). La calibration HP-QMC fonctionne comme la calibration HP-LHS de l'exemple précédent.

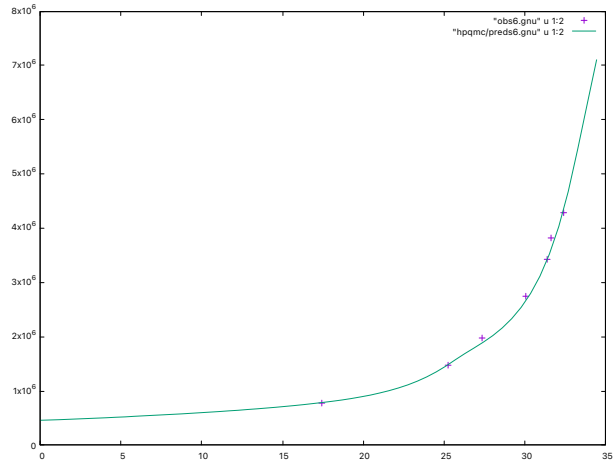


FIGURE 1 – Prédictions HP-QMC pour exemple MITB, expérience 6

6.4 Résultats

Le calcul tourne en quelques secondes lorsqu'on ne considère qu'une seule expérience (cas 6). En affichant les observations (obsN.gnu) en face des prédictions (colonnes 1 et 2 de predsN.gnu), on vérifie que le modèle corrigé fitte bien les observations (figure 1).

7 Exemple 3 : Neptune_ CFD et expériences DEBORA

7.1 Description

Cet exemple reprend la calibration du code de calcul NaptuneCFD, avec les données de l'expérience Debora. Se référer au chapitre 6 de [1] pour le contexte. Dans cet exemple, on propose une calibration avec deux observables, le taux de vide et le diamètre de bulle. On effectue dans le `main.cpp` la construction d'un surrogate de Neptune CFD, par Processus Gaussiens sur les Composantes Principales des données. On propose également des prédictions avec le modèle corrigé. Les méthodes de calibration FMP et HP-AS sont proposées sur ce cas.

7.2 Fichiers

Le répertoire `data` contient l'ensemble des données de calcul CFD. Un design pour construire le surrogate (répertoire `qmc`), et un design pour le valider (répertoire `qmc`). Les données CFD sont sous le format d'un fichier .dat, où les trois premieres colonnes sont l'abscisse radiale, le taux de vide, et le diamètre de bulle. Les valeurs de paramètres utilisées pour lancer ces calculs sont stockeés dans les fichiers `code/design_qmc_full.dat` et `code/design_lhs_full.dat`. L'ensemble est lu dans le `main.cpp`. Enfin, les observations sont dans `data/exp`.

Les fichiers `hpars_gp_deb_alpha.gnu` et `hpars_gp_deb_diam.gnu` dans le répertoire `code` sont lus par le code, ils contiennent les hyperparamètres des processus gaussiens utilisés pour le surrogate de NeptuneCFD. Cela évite de procéder à leur optimisation à chaque exécution du code.

7.3 main.cpp

Les fonctions définies en début de fichier correspondent à la lecture des données de calcul (**read_results_qmc** par exemple), à la construction du surrogate de NeptuneCFD (**PerformPCA** par exemple), ou à l'évaluation de son erreur (**compute_erreurs_validation**).

Le main commence par la lecture des données expérimentales et de calcul (l. 556-560), et la construction du surrogate pour NeptuneCFD (l. 567-610 pour le taux de vide alpha, l.612-655 pour le diamètre D). On extrapole les résultats de calcul sur les points expérimentaux (l. 663-664), et on définit les bornes des paramètres/hyperparamètres (l. 681-720). On construit ensuite 2 objets Density, l'une pour le taux de vide et l'autre pour le diamètre. Cela requiert deux fonctions lambda pour jour le rôle de modèle f (**lambda_model_alpha** et **lambda_model_diam**). Les deux Density sont créées des lignes 754 à 772. Notons l'usage de **SetLearnedOutputerr**, où l'on donne en argument l'indice "1", qui représente la place de l'erreur expérimentale dans le vecteur d'hyperparamètres.

Les deux calibration proposées sont la FMP (l. 780) et HP-AS (l. 883), qui s'effectuent de la même manière que dans les deux exemples précédents.

7.4 Résultats

L'exécution du code prend moins de 10 minutes. On obtient entre autres, pour fmp et hpas, un échantillon de la densité à posteriori (`samples.gnu`),

le map (`map.gnu`), et la prédiction du surrogate de NeptuneCFD au map (`predmap.gnu`).

J'obtiens pour le MAP FMP : $(0.244115, 0.531301, 0.272722, 1, 0.473508)$, et pour le MAP HP-AS : $(0.246799, 0.550598, 0.278985, 0.99727, 0.478874)$.

Références

[1] Nicolas Leoni. "Bayesian Inference of Model Error for the Calibration of Two-Phase CFD Codes". Institut Polytechnique de Paris, 2022.

[2] Nicolas Leoni et al. "Surrogate-Based Strategies for Accelerated Bayesian Calibration of Computer Codes with Full Maximum a Posteriori Estimation of Model Error". In : *Submitted to Reliability Engineering & System Safety* (2022).