Tutoriaux pour la plateforme d'assimilation de données

Jean-Philippe ARGAUD

Version du 28 janvier 2009

Table des matières

1 Choix généraux **3**
1.1 Utiliser le langage Python et de ses modules 3
1.2 Utiliser des vecteurs et des matrices 3
1.3 Utiliser des outils de calcul numérique, de statistiques... 4
1.4 Disposer de compléments d'informations au cours d'un calcul 4
1.5 Obtenir des informations sur la mémoire et les versions 6

2 Accéder aux données et aux résultats **8**
2.1 Les structures de données disponibles 8
2.2 Accéder à une donnée pour la construire et la remplir 8
2.2.1 Création 8
2.2.2 Remplissage 9
2.3 Accéder à un résultat pour l'exploiter 10
2.4 Les opérations sur une donnée structurée 10

3 Faire une étude simple d'assimilation **12**
3.1 L'étude à traiter, résultats attendus 12
3.2 La mise en place de l'étude et la conduite des calculs 13
3.3 L'exploitation des résultats 14
3.4 Compléments sur les méthodes d'un objet d'étude 14
3.5 Variante utile : faire plusieurs calculs ou de la sensibilité 15

4 Ecrire un nouvel outil de diagnostic **17**
4.1 Squelette standard d'outil de diagnostic et explications 17
4.2 Précisions sur les parties à modifier du squelette 20
4.3 Echanger et stocker des variables 20
4.4 Le code propre du diagnostic sur les données 21
4.5 Tester et utiliser le nouveau diagnostic 22

5 Ecrire un nouvel algorithme d'assimilation **23**
5.1 Squelette standard d'algorithme et explications 23
5.2 Précisions sur les parties à modifier du squelette 25
5.3 Echanger et stocker des variables 25
5.4 Le code propre de l'algorithme d'assimilation 26
5.5 Tester et utiliser le nouvel algorithme 27

1 Choix généraux

On indique brièvement ici les éléments de choix généraux qui peuvent être nécessaires pour l'utilisateur autant que pour le programmeur auxquels sont destinés les tutoriaux suivants.

1.1 Utiliser le langage Python et de ses modules

On choisit d'utiliser par défaut le langage Python, non seulement pour conduire les opérations par l'utilisateur, mais aussi pour travailler sur les données ou pour réaliser des calculs numériques efficaces. En effet, Python et ses modules scientifiques disposent de l'ensemble des fonctionnalités a priori nécessaires à la conduite de l'assimilation de données.

De manière pratique, au-delà du langage python et de ses modules standards, les seuls prérequis sont les modules `Numpy` et `Scipy`. On recommandera fortement deux modules supplémentaires pour le post-traitement, à savoir `Gnuplot` et `Matplotlib`. Leur absence ne doit pas bloquer le fonctionnement, mais les fonctions d'affichage sont dans ce cas fortement réduites.

Pour des tâches spécialisées comme par exemple la conduite de calculs parallèles ou le post-processing sur-mesure, on s'appuyera sur des outils complémentaires, souvent externes, et en particulier SALOME. Comme pour le post-traitement, l'absence de ces modules externes ne doit pas bloquer le fonctionnement.

Les modules utilisés sont donc, dans l'ordre :
- les modules standards de Python,
- les modules `Numpy` et `Scipy`,
- les modules `Gnuplot` et `Matplotlib`,
- les modules externes de type applicatif Python.

1.2 Utiliser des vecteurs et des matrices

L'utilisation des vecteurs et matrices est entièrement basée sur le recours à `Numpy` et en particulier à sa classe "`matrix`". Elle diffère de la classe habituelle "`array`" car elle produit des vecteurs orientés et non de simples vecteurs unidimensionnels. Les objets ainsi créés sont exactement similaires aux concepts normaux de vecteurs et matrices en algèbre linéaire. Dans ce cas, un vecteur et son transposé ne sont par exemple pas identiques.

Cela signifie que la création de l'un de ces objets se fait toujours explicitement en utilisant le constructeur standard de `Numpy`. Ainsi, toute donnée compatible avec ce constructeur permettra donc de créer le vecteur ou la matrice. De même, on requiert de toute donnée de création d'être compatible avec le constructeur standard de type "`matrix`" de `Numpy`.

On rappelle, pour un objet de type "`matrix`", les quelques méthodes indispensables pour l'utiliser :
- création d'un vecteur ou d'une matrice : `x = numpy.matrix(...)`
- transposition : `x.T`
- inversion d'une matrice : `x.I`
- conversion en un vecteur unidimensionnel : `x.A1`

La création de l'objet de type "`matrix`" peut se faire avec une liste de nombres (exemple pour un vecteur : `[0, 1, 2]`), une liste de liste de nombres (exemple pour une matrice : `[[1,0,0],[0,1,0],[0,0,1]]`), ou avec une chaîne de caractères qui sépare les éléments

d'une ligne par des espaces et les lignes par des points virgules (exemple pour une matrice : `"1 0 0;0 1 0;0 0 1"`).

1.3 Utiliser des outils de calcul numérique, de statistiques...

Tous les outils numériques sont atteignables par l'intermédiaire de `Numpy` et `Scipy`, et éventuellement directement dans les modules spécialisés, comme par exemple l'environnement SALOME ou le module d'interfaçage entre l'outil de statistiques R et Python.

`Numpy` contient l'ensemble du calcul numérique matriciel, avec les opérations de base sur les matrices, le traitement avancé de matrices et leur analyse, ainsi que la résolution de systèmes linéaires par exemple.

`Scipy` contient un grand nombre de possibilités de calculs (très similaires à ce dont l'on dispose dans Matlab ou Scilab) : optimisation, FFT, statistiques (par `R`, qui contient énormément d'outils pour le traitement statistique et l'estimation), matrices creuses, LAPACK et BLAS...

Notons enfin que tous les algorithmes disponibles à travers `Numpy` et `Scipy` sont optimisés en langage de type Fortran ou C, et doivent donc avoir de bonnes performances numériques pourvu qu'on les utilise correctement. Dans certains cas, et en particulier en algèbre linéaire optimisé, des modules supplémentaires améliorent encore les performances si nécessaire.

1.4 Disposer de compléments d'informations au cours d'un calcul

Des informations complémentaires peuvent être disponibles au cours d'un calcul à travers l'utilisation d'une démarche de "logging". Cela consite à disposer à la console de messages supplémentaires, prévus à divers endroits dans le code ou les calculs, et contrôlables de manière macroscopique en demandant un niveau donné d'affichage.

On indique simplement ici que les niveaux d'affichage préprogrammés sont les suivants, avec une valeur numérique correspondante à chaque constante :

Niveau	Valeur
CRITICAL	50
ERROR	40
WARNING	30
INFO	20
DEBUG	10
NOTSET	0

On peut se reporter à la documentation du module Python "`logging`" pour un point de vue plus approfondi. Il est fortement déconseillé (sauf dans un cas décrit plus loin) d'utiliser les valeurs numériques, et il est recommandé d'utiliser les constantes ci-dessus qui sont des attributs du module "`logging`" (avec une minuscule).

Initialisation

Par défaut, **le logging est configuré et activé dans les études d'assimilation standard**. Il suffit donc d'importer le module principal pour que l'initialisation avec les choix par défaut soit faite, dont en particulier le fait que seuls les messages de type WARNING ou au-delà sont affichés.

```
.... import AssimlationStudy
```

On peut aussi l'intégrer indépendamment en appelant directement le module propre de configuration "`Logging`" (avec un "L" majuscule), mais ce n'est pas recommandé. Pour cela, il faut instancier un objet du même nom :

```
.... import Logging
.... Logging.Logging()
```

Le niveau d'affichage se définit à l'aide du paramètre "`level`", affecté d'une valeur numérique ou d'un attribut de niveau du module "logging" (avec un "l" minuscule) :

```
.... Logging.Logging(level=20)
```

On peut en plus demander à l'objet de sortir conjointement les messages dans un fichier. Le nom par défaut du fichier de stockage est "AssimilationStudy.log" et il est rangé dans le répertoire courant de lancement de la commande python. Si on utilise encore les valeurs par défaut, alors il suffit d'écrire :

```
.... import Logging
.... Logging.Logging().setLogfile()
```

Configuration

Pour configurer le niveau global de messages ou le nom du fichier d'enregistrement des logs, il faut conserver l'instance de classe et lui appliquer des méthodes particulières :

```
.... import Logging
.... log = Logging.Logging()
.... import logging
.... log.setLevel(logging.DEBUG)
.... log.setLogfile(filename="toto.log", filemode="a", level=logging.WARNING)
```

et on peut changer le niveau d'affichage dans le fichier seul par la commande :

```
.... log.setLogfileLevel(logging.INFO)
```

Dans une application, à n'importe quel endroit et autant de fois qu'on veut, on peut changer le niveau global de message (et l'éventuel fichier ne contiendra pas les messages de niveau inférieur à niveau global défini) en utilisant par exemple :

```
import logging
logging.getLogger().setLevel(logging.DEBUG)
```

Utilisation

Ensuite, n'importe où dans les applications, il suffit d'utiliser le module "`logging`" (avec un petit "l"). Pour cela, on crée un objet de logging :

```
.... import logging
.... log = logging.getLogger()
```

Ensuite, pour un message quelconque, noté ici `"..."`, on peut l'afficher conditionnellement au niveau choisi par l'une des commandes suivantes :

```
.... log.critical("...")
.... log.error("...")
.... log.warning("...")
.... log.info("...")
.... log.debug("...")
```

De manière un peu moins propre mais plus claire, on peut écrire directement l'appel au module standard comme dans les lignes suivantes :

```
.... import logging
.... logging.info("...")
```

On recommande ce dernier usage.

1.5 Obtenir des informations sur la mémoire et les versions

Pour disposer d'informations sur les versions logicielles utilisées et sur la mémoire, on peut utiliser le module "`PlatformInfo`" disponible dans les outils d'assimilation. Il permet d'accéder de manière simple aux informations et de les afficher par autodiagnostic lorsqu'il est directement exécuté.

Informations sur les versions logicielles

Les informations sur les versions logicielles sont disponibles en instanciant un objet de la classe "`PlatformInfo`" et en utilisant les diverses méthodes prévues, qui sont les suivantes :

- `getName` : nom de l'application courante
- `getVersion` : version de l'application courante
- `getDate` : date de la version de l'application courante
- `getPythonVersion` : version de Python
- `getNumpyVersion` : version de Numpy
- `getScipyVersion` : version de Scipy

Un exemple d'utilisation est le suivant :

```
.... from PlatformInfo import PlatformInfo
.... print PlatformInfo()
.... print
.... p = PlatformInfo()
.... print "Les caractéristiques détaillées des applications et outils sont :"
.... print "  - Application.......:",p.getName()
.... print "  - Version...........:",p.getVersion()
.... print "  - Date Application..:",p.getDate()
.... print "  - Python............:",p.getPythonVersion()
.... print "  - Numpy.............:",p.getNumpyVersion()
.... print "  - Scipy.............:",p.getScipyVersion()
```

On peut noter que les champs de version renvoyés peuvent être utilisés pour de l'affichage ou pour des tests de compatibilité de version.

Informations sur la mémoire

Les informations sur la mémoire sont récupérées dynamiquement et peuvent donc être demandées à tout moment lors d'un calcul. Elles sont disponibles en instanciant un objet de la classe "`SystemUsage`" et en utilisant les diverses méthodes prévues, qui sont les suivantes :

- `getAvailableMemory` : indique la taille de la mémoire existante totale dans le système, i.e. la mémoire physique et la mémoire de swap.
- `getAvailablePhysicalMemory` : indique la taille de la mémoire physique existante.
- `getAvailableSwapMemory` : indique la taille de la mémoire de swap existante.
- `getUsableMemory` : calcul la taille de la mémoire potentiellement disponible pour un process. Attention, cette indication n'est pas forcément fiable, mais par contre elle indique usuellement la borne haute de la taille de mémoire utilisable.
- `getUsedMemory` : pour le process en cours dans lequel est appelé cette commande, indique la mémoire totale utilisée par le process.
- `getUsedResident` : pour le process en cours dans lequel est appelé cette commande, indique la mémoire RSS utilisée par le process.
- `getUsedStacksize` : pour le process en cours dans lequel est appelé cette commande, indique la mémoire de stack utilisée par le process.
- `getMaxUsedMemory` : renvoie la mémoire maximale mesurée lors des différents appels précédents à `getUsedMemory`.
- `getMaxUsedResident` : idem pour `getUsedResident`.
- `getMaxUsedStacksize` : idem pour `getUsedStacksize`.

Un exemple d'utilisation est le suivant :

```
.... from PlatformInfo import SystemUsage
.... m = SystemUsage()
.... print "La mémoire disponible est la suivante :"
.... print "  - mémoire totale....: %4.1f Mo"%m.getAvailableMemory("Mo")
.... print "  - mémoire physique..: %4.1f Mo"%m.getAvailablePhysicalMemory("Mo")
.... print "  - mémoire swap......: %4.1f Mo"%m.getAvailableSwapMemory("Mo")
.... print "  - utilisable........: %4.1f Mo"%m.getUsableMemory("Mo")
.... print "L'usage mémoire de cette exécution est le suivant :"
.... print "  - mémoire totale....: %4.1f Mo"%m.getUsedMemory("Mo")
.... print "  - mémoire résidente.: %4.1f Mo"%m.getUsedResident("Mo")
.... print "  - taille de stack...: %4.1f Mo"%m.getUsedStacksize("Mo")
```

Ces méthodes peuvent en particulier être appelées au cours du déroulement du code pour surveiller l'emprise mémoire temporaire ou maximale d'une exécution complète. Les méthodes "`getMax...`" permettent d'accéder aux valeurs maximales observées.

2 Accéder aux données et aux résultats

L'accès correct aux données et aux résultats est une part essentielle de la construction des études appliquées. Cette section décrit l'accès utilisateur aux informations et à leur structuration, ainsi que l'ensemble des outils de traitement direct ou indirect des données.

On appelle *donnée* un ensemble d'informations structurées dans un objet Python particulier. Cet objet dispose ensuite de *méthodes* pour l'alimenter, récupérer ou modifier des valeurs...

2.1 Les structures de données disponibles

Les données sont prévues pour être stockées à l'aide d'une classe de base nommée "`Persistence`", qui définit les structures de persistence et d'enregistrement de séries de valeurs, pour une analyse ultérieure ou une utilisation dans les calculs.

Par définition, une donnée est un ensemble de valeurs homogène ou hétérogène dont on dispose à des pas (de temps ou arbitraires) successifs. Par exemple, c'est un vecteur de valeur disponible dans le temps, ou une liste d'informations hétérogènes disponibles à chaque pas d'itération. On peut illustrer par le schéma 1, dans lequel les données sont `x`, `y`, `V1`, `V2`, `X`, `Y`... quelconques et intentionnellement différentes.

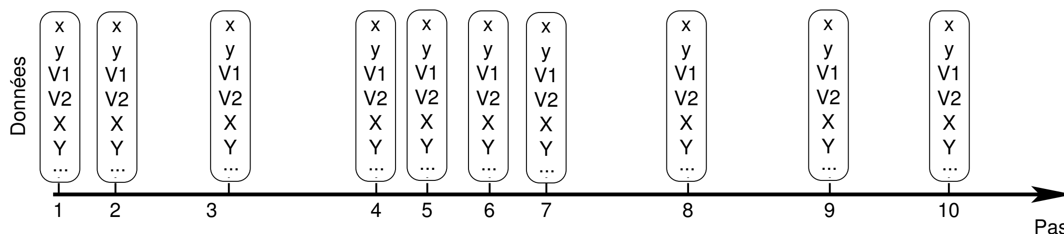

Fig. 1 – *Schématisation de données disponibles aux pas (de temps ou d'itération)*

Une donnée structurée de type "`Persistence`" présente trois groupes de méthodes d'accès permettant :

- la construction d'une nouvelle donnée et son stockage, décrits en section 2.2 (p.8),
- la récupération des valeurs de la donnée, décrite en section 2.3 (p.10),
- et les opérations sur les valeurs stockées pour la donnée, décrites en section 2.4 (p.10).

Dans la pratique, l'utilisateur est invité à utiliser dès que possible des objets spécialisés préparés sur la base de la classe "`Persistence`" pour faciliter la lecture du code écrit. Ces structures spécialisées sont les types suivants : "`OneScalar`" (pour stocker une valeur unique à chaque pas), "`OneVector`" (pour stocker un vecteur de valeurs homogènes à chaque pas), "`OneMatrix`" (pour stocker une matrice à chaque pas), "`OneList`" (pour stocker un vecteur de valeurs hétérogènes à chaque pas). Toutes les méthodes qui vont être décrites pour la classe "`Persistence`" sont disponibles pour chacune de ces classes spécialisées.

2.2 Accéder à une donnée pour la construire et la remplir

2.2.1 Création

La principale méthode de construction d'un objet de type "`Persistence`" est la méthode classique "`__init__`". Ses arguments sont les suivants :

- "name" permet de définir le nom de l'objet stocké, sous la forme d'une chaîne de caractères. Ce nom est informatif, mais il peut aussi servir à accéder à l'objet créé. Il faut donc veiller à ce que la chaîne soit discriminante.
- "unit" permet de définir l'unité de la donnée stockée à chaque pas. Dans ce cas aussi, c'est une chaîne de caractère à vocation informative, qui n'est pas vérifiée.
- "basetype" permet de définir le type Python de chaque donnée stockée. Cela peut être un type intrinsèque simple (comme float, int, bool...), mais peut aussi être un type complexe qui permette l'initialisation directe à l'aide de données fournies en argument. C'est le cas par exemple de "numpy.array" et "numpy.matrix". L'intérêt des types cités est de valider les données à leur initialisation. Mais on peut aussi profiter de classes utilisateurs, ou des "list/tuple" Python pour stocker des valeurs hétérogènes. Par défaut, en l'absence explicite de cet argument, le stockage se fait en chaîne de caractères "str".

On utilise cette classe (ou une des classes dérivées "OneScalar", "OneVector", "OneMatrix, "OneList") de la manière suivante pour créer un objet, qui est par exemple ici de type entier à chaque pas de stockage :

```
OBJET_DE_TEST = Persistence("My object", unit="", basetype=int)
```

Il est ensuite possible de modifier dynamiquement le type interne de stockage des données en utilisant la méthode "basetype". Lorsqu'elle est utilisée sans argument, elle renvoie le type Python de chaque donnée stockée. Si elle présente un argument nommé "basetype", il est utilisé pour modifier le type interne de stockage. On remarque bien que cette méthode est d'abord destinée à récupérer le type interne, plutôt qu'à le modifier (puisque cela peut être effectué en cours de remplissage des données, la modification peut éventuellement conduire à des conflits ultérieurs).

2.2.2 Remplissage

Pour remplir l'objet créé, on utilise la méthode "store" dont la fonction unique est de stocker une valeur, indiquée par l'argument nommé "value", à un pas, indiqué par l'argument optionnel "step". Si le pas n'est pas indiqué, le compteur interne de stockage est utilisé. Ce dernier démarre à 0. A partir de l'exemple ci-dessus de création, on peut donc compléter l'exemple avec trois pas de stockage par défaut :

```
OBJET_DE_TEST = Persistence("My object", unit="", basetype=int)
OBJET_DE_TEST.store(  1 )
OBJET_DE_TEST.store( -3 )
OBJET_DE_TEST.store(  7 )
```

Dans ce cas simple de stockage d'un scalaire, il a été prévu une classe plus explicite nommée "OneScalar", qui s'utilise strictement de la même manière. Son intérêt est exclusivement de rendre plus lisible le code produit :

```
OBJET_DE_TEST = OneScalar("Un entier", unit="", basetype=int)
OBJET_DE_TEST.store(  1 )
OBJET_DE_TEST.store( -3 )
OBJET_DE_TEST.store(  7 )
```

2.3 Accéder à un résultat pour l'exploiter

Les données sont stockées pour une liste de pas, donc, de manière générale, on peut disposer d'un résultat en allant chercher sa valeur à un pas particulier, ou de la liste complète des valeurs et des pas.

Un objet de type "`Persistence`" renvoie le nombre de pas de stockages par la méthode "`stepnumber`", sous la forme :

```
nombre_de_pas = OBJET_DE_TEST.stepnumber()
```

Un objet de type "`Persistence`" peut ensuite renvoyer la liste complète des pas ou des valeurs avec deux méthodes particulières : "`stepserie`" et "`valueserie`". Ces deux méthodes, lorsqu'elles sont appelées sans argument, renvoient une liste simple :

```
la_liste_des_pas     = OBJET_DE_TEST.stepserie()
la_liste_des_valeurs = OBJET_DE_TEST.valueserie()
```

Dans chaque liste retournée, les valeurs sont au type utilisé lors du stockage, sans transformation, ce qui rend un objet de type "`Persistence`" extrêmement souple pour stocker toutes sortes de valeurs utilisateurs.

Un objet de type "`Persistence`" peut enfin renvoyer des valeurs spécifiques de pas ou de valeurs. On utilise pour cela les mêmes méthodes "`stepserie`" et "`valueserie`", mais en leur donnant des arguments nommés "`step`" ou "`item`". L'argument de pas "`step`", si il existe dans la liste existante des pas de stockage, permet d'obtenir le pas ou la valeur stockée à ce pas. L'argument d'index "`item`", si il est inférieur au nombre de stockages (que l'on peut vérifier avec la méthode ci-dessus "`stepnumber`"), permet d'obtenir le pas ou la valeur stocké à cet index. Si un seul argument est fourni sans être nommé, il indique une valeur pour "`item`". Pour illustrer cela, voici des exemples d'appels :

```
un_pas     = OBJET_DE_TEST.stepserie(step = <un pas>)
une_valeur = OBJET_DE_TEST.valueserie(step = <un pas>)

le_troisieme_pas     = OBJET_DE_TEST.stepserie(item = 3)
la_troisieme_valeur = OBJET_DE_TEST.valueserie(item = 3)
la_troisieme_valeur = OBJET_DE_TEST.valueserie(step = le_troisieme_pas)

le_dernier_pas     = OBJET_DE_TEST.stepserie(-1) # Equivalent à item = -1
la_derniere_valeur = OBJET_DE_TEST.valueserie(-1)
```

Dans tous les cas, si le pas, ou la valeur, demandé (que ce soit par un "`step`" ou un "`item`") n'existe pas, c'est la liste complète des pas, ou des valeurs, qui est renvoyée. Si les arguments "`step`" et "`item`" sont simultanément remplis, c'est "`step`" qui est prioritaire. S'il n'est pas valide, "`item`" est ensuite pris en compte.

2.4 Les opérations sur une donnée structurée

Un certain nombre d'opérations peuvent être effectuées sur les valeurs stockées sans qu'il soit besoin pour cela de récupérer les valeurs elles-mêmes. Pour cela, il suffit de s'adresser à l'objet de type "`Persistence`" pour lui demander ces opérations prévues.

Les opérations prévues soit sont effectuées à chaque pas, soit donnent un résultat sur l'ensemble des pas de stockage. On décrit les deux groupes d'opérations successivement. Dans tous les cas, il faut que le type de base soit compatible avec les types élémentaires "`numpy`".

Les opérations pouvant être effectuées à chaque pas renvoient une liste de valeurs (les résultats de l'opération) associées aux pas de stockage. Les méthodes sont actuellement les suivantes :

- "`mean`" : renvoie la valeur moyenne des données.
- "`std`" : renvoie l'écart-type des données.
- "`sum`" : renvoie la somme des données.
- "`min`" : renvoie le minimum des données.
- "`max`" : renvoie le maximum des données.
- "`plot`" : renvoie un affichage Gnuplot pour chaque pas, sous la condition que la valeur à un pas soit compatible avec ce type de tracé. [1]

Ces méthodes ne prennent aucun argument, sauf la dernière. On remarque que ces méthodes simples sont "accrochées" directement aux objets de persistence pour simplifier l'usage des objets, mais elles sont conceptuellement tout à fait similaires aux diagnostics effectués sur un unique objet de persistence. L'usage de diagnostics permet néanmoins d'étendre les méthodes mono-objets sans intervenir sur la classe "`Persistence`".

Les opérations pouvant être effectuées sur l'ensemble des pas renvoient une valeur valable pour l'ensemble des pas, mais pas obligatoirement associées aux pas de stockage. Tous ces calculs se font sans tenir compte de la longueur des pas. Les méthodes sont actuellement les suivantes :

- "`stepmean`" : renvoie la moyenne sur toutes les valeurs.
- "`stepstd`" : renvoie l'écart-type de toutes les valeurs.
- "`stepsum`" : renvoie la somme de toutes les valeurs.
- "`stepmin`" : renvoie le minimum de toutes les valeurs.
- "`stepmax`" : renvoie le maximum de toutes les valeurs.
- "`cumsum`" : renvoie la somme cumulée de toutes les valeurs.
- "`stepplot`" : renvoie un affichage unique pour l'ensemble des valeurs à tous les pas. [2]

Ces méthodes ne prennent aucun argument, sauf la dernière.

[1] Les arguments de la méthode "`plot`" sont les suivants : "`item`" ou "`step`" permettent de sélectionner une seule valeur dans toute celles stockées, par un index de liste ou un index nommé respectivement. "`steps`" permet de donner explicitement une liste de valeurs pour l'affichage de l'axe des abscisses. "`title`", "`xlabel`" et "`ylabel`" permettent de donner des légendes à la figure et aux axes. "`ltitle`" permet de donner la légende de la ligne tracée. "`geometry`" impose la géométrie pour la fenêtre, en pixels, et la position du coin haut gauche, au format X11 : `LxH+X+Y` (par défaut, c'est "`600x400`"). "`filename`" donne la base de nom de fichier Postscript pour une sauvegarde par pas, qui est automatiquement complétée par le numéro du fichier calculé par incrément simple de compteur. "`persist`" rend la fenêtre permanente même après la sortie de l'interpréteur Python. "`pause`" permet de faire une pause après l'affichage d'une figure à chaque pas.

[2] Les arguments de la méthode "`stepplot`" sont les suivants : "`steps`" permet de donner explicitement une liste de valeurs pour l'affichage de l'axe des abscisses. "`title`", "`xlabel`" et "`ylabel`" permettent de donner des légendes à la figure et aux axes. "`ltitle`" permet de donner la légende de la ligne tracée, qui est automatiquement complétée par l'indication du pas. "`geometry`" impose la géométrie pour la fenêtre, en pixels, et la position du coin haut gauche, au format X11 : `LxH+X+Y` (par défaut, c'est "`600x400`"). "`filename`" donne le nom de fichier Postscript pour une sauvegarde. "`persist`" rend la fenêtre permanente même après la sortie de l'interpréteur Python. "`pause`" permet de faire une pause après l'affichage d'une figure.

3 Faire une étude simple d'assimilation

Le présent tutorial vise à expliciter l'environnement d'écriture et à donner un exemple simple d'étude complète. A priori, ce document est destiné à un utilisateur débutant, qui veut mettre en place une étude d'assimilation.

Le tutorial présente progressivement les points suivants :
- une brève description de l'étude que l'on veut traiter, pour comprendre l'objectif poursuivi dans la mise en place de l'algorithme et décrire les données disponibles,
- la mise en place de l'étude proprement dite pour conduire les calculs,
- les résultats et la manière de les exploiter,
- et enfin quelques compléments simples utiles pour étendre les calculs disponibles.

3.1 L'étude à traiter, résultats attendus

L'étude d'assimilation que l'on décrit ici se focalise sur l'analyse à l'aide d'une méthode d'assimilation. La physique du problème sous-jacent est complètement occultée pour insister sur le type de données et sur la conduite des calculs.

On considère un problème que l'on peut entièrement décrire par des matrices et des vecteurs. On s'intéresse ici à un problème statique d'interpolation optimale, donc qui relève d'une estimation de type BLUE.

Le vecteur des observations est :

$$\mathbf{y}^o = [0.5, 1.5, 2.5]$$

et le vecteur de l'estimation a priori (ébauche ou background) est :

$$\mathbf{x}^b = [0, 1, 2]$$

On suppose que les matrices de variance-covariance des erreurs d'ébauche $\mathbf{B}$ comme d'observations $\mathbf{R}$ sont diagonales unitaires, c'est-à-dire que les erreurs sont découplées et que la variance de chaque observation est arbitrairement égale à 1. De plus, on considère que l'on observe avec un opérateur $\mathbf{H}$ de sélection qui conserve toutes les valeurs d'ébauche puisque l'on en a le même nombre que d'observations. On a donc $\mathbf{B} = \mathbf{R} = \mathbf{H} = \mathbf{I}$.

Cette hypothèse n'est pas limitative car elle permet de connaître a priori l'interpolation optimale entre les observations et l'estimation a priori. Dans ce cas où les matrices de covariance sont égales à l'identité, la matrice de gain de Kalman K se réduit en effet à :

$$\mathbf{K} = \mathbf{B}.\mathbf{H}^t.(\mathbf{H}.\mathbf{B}.\mathbf{H}^t + \mathbf{R})^{-1} = \mathbf{H}^t.(\mathbf{H}.\mathbf{H}^t + \mathbf{I})^{-1} = \frac{1}{2}\mathbf{I}$$

Ainsi, le calcul BLUE de l'analyse $\mathbf{x}^a$ devient explicite, et vaut :

$$\mathbf{x}^a = \mathbf{x}^b + \mathbf{K}.(\mathbf{y}^o - \mathbf{H}.\mathbf{x}^b) = \mathbf{x}^b + \frac{1}{2}\mathbf{y}^o - \frac{1}{2}\mathbf{x}^b = \frac{1}{2}(\mathbf{y}^o + \mathbf{x}^b)$$

L'analyse $\mathbf{x}^a$ est donc simplement le milieu (composante par composante) des vecteurs d'observation $\mathbf{y}^o$ et d'ébauche $\mathbf{x}^b$:

$$\mathbf{x}^a = [0.25, 1.25, 2.25]$$

C'est ce résultat que l'on vérifiera à la fin de la mise en place de ce cas d'assimilation.

3.2 La mise en place de l'étude et la conduite des calculs

Une étude se met en place en trois phases :

1. définition des données "physiques" du problème d'assimilation,
2. paramétrage et lancement du calcul,
3. récupération des résultats et des analyses.

On va détailler chacune de ces phases (la dernière dans le paragraphe 3.3), après la vue d'ensemble fournie dans le squelette 2 pour l'étude complète. On remarque déjà la compacité de l'étude en tant que telle, même si la mise en données matricielles est évidemment plus importante dans un cas réel.

```
 1  #-*-coding:iso-8859-1-*-
 2  from AssimilationStudy import AssimilationStudy
 3  #
 4  ADD = AssimilationStudy("Ma premiere etude")
 5  #
 6  ADD.setBackground         (asVector     = [0,1,2])
 7  ADD.setBackgroundError    (asCovariance = "1 0 0;0 1 0;0 0 1")
 8  ADD.setObservation        (asVector     = [0.5,1.5,2.5])
 9  ADD.setObservationError   (asCovariance = "1 0 0;0 1 0;0 0 1")
10  ADD.setObservationOperator(asMatrix     = "1 0 0;0 1 0;0 0 1")
11  #
12  ADD.setAlgorithm(choice="Blue")
13  #
14  ADD.analyze()
15  #
16  Xa = ADD.get("Analysis")
```

FIG. 2 – Etude simple d'assimilation

On explique donc ce squelette 2 ligne à ligne ci-après.

En ligne 1, on indique que l'encodage des caractères du fichier lui permet de contenir en particulier des accents dans le texte.

En ligne 2, on requiert le module principal permettant de conduire les études d'assimilation. il peut être nécessaire de mettre préalablement en place le chemin qui permet à Python de le retrouver. En utilisant seulement Python, on réalise cela avec la ligne suivante :
`import sys ; sys.path.insert(0, "<chemin que l'on veut>/Sources/daCore")`

En ligne 4, on définit un objet d'étude d'assimilation qui va être l'interface générale pour l'étude à mener. Cet objet peut comporter un nom en clair, pour faciliter par exemple ensuite l'identification des résultats.

En ligne 6 à 10, on met en place les données physiques du problème à l'aide des méthodes prévues de l'objet d'assimilation. L'ensemble des méthodes est le suivant :

- `setBackground` : définir l'estimation a priori $\mathbf{x}^b$ (ébauche)
- `setBackgroundError` : définir la covariance des erreurs d'ébauche $\mathbf{B}$
- `setObservation` : définir les observations $\mathbf{y}^o$
- `setObservationError` : définir la covariance des erreurs d'observation
- `setObservationOperator` : définir un opérateur d'observation $\mathbf{H}$
- `setEvolutionModel` : définir un opérateur d'évolution $\mathbf{M}$

– `setEvolutionError` : définir la covariance des erreurs de modèle

Selon les cas, toutes les méthodes ne sont pas nécessaires. En particulier, dans le cas statique, l'opérateur d'évolution et ses covariances d'erreurs n'existent pas. Les informations sont passées sous forme vectorielle ou matricielle avec les paramètres nommés de manière adéquate par "`asVector`", "`asMatrix`", ou "`asCovariance`".

En ligne 12, on spécifie l'algorithme d'assimilation à utiliser sous la forme d'une chaîne de caractère. L'algorithme doit évidemment déjà exister et être disponible dans la plateforme. On peut obtenir la liste des algorithmes possibles en interrogeant l'étude par la commande suivante : "`ADD.get_available_algorithms()`".

En ligne 14, on lance l'analyse, ce qui permet d'activer l'algorithme d'assimilation pour effectuer la totalité des opérations d'analyse requises. Il n'y a aucun argument à cette méthode.

En ligne 16, on récupère un des résultats pour pouvoir l'exploiter, comme on le décrit dans la partie 3.3 qui suit.

3.3 L'exploitation des résultats

L'exploitation des résultats repose sur leur récupération en deux étapes, qu'il convient de bien séparer pour faciliter les traitements.

La **première étape** consiste à récupérer les résultats globaux de l'étude.

Ces résultats sont a priori rangés dans des objets de stockage de la plateforme, à savoir des objets qui stockent un type élémentaire (un réel, un vecteur, une matrice) selon des pas de temps ou d'itération. On peut récupérer les objets globaux qui sont stockés lors du déroulement de l'algorithme. L'analyse, qui est toujours stockée, est l'un de ces objets que l'on récupère donc en ligne 16.

De manière générale, l'étude présente une méthode "`get`" qui renvoie, lorsqu'elle est invoquée sans argument, tous les résultats globaux disponibles sous la forme d'un dictionnaire. Ce dictionnaire est une série de paires avec le nom de chaque résultat global et l'objet correspondant. Lorsque l'on donne un argument à cette méthode "`get`", alors la méthode cherche à renvoyer l'objet nommé.

La **seconde étape** consiste à analyser chaque objet global seul ou avec d'autres.

Un tel objet se comporte comme une liste dont on obtient la valeur élémentaire à un pas de temps ou d'itération par la méthode "`valueserie`" avec le pas en argument. De plus, cet objet dispose d'un nombre important de méthodes complémentaires pour obtenir des informations sur les pas ou pour mener les analyses prévues sur un objet isolé : moyenne, écart-type, minimum, maximum, somme pour chaque pas, ou la même chose pour l'ensemble des pas, etc.

L'analyse conjointe d'objets globaux relève des outils de diagnostics, mis en place justement pour réaliser des calculs faisant intervenir simultanément plusieurs objets de stockage. C'est par exemple le cas d'un calcul de RMS entre deux vecteurs à chaque pas d'itération.

3.4 Compléments sur les méthodes d'un objet d'étude

Il existe quelques méthodes supplémentaires de l'étude, qui n'ont pas été évoquées sur ce cas statique avec algorithme simple.

La méthode "`setControls`" permet de définir dans l'étude la valeur initiale des paramètres

de contrôle que l'on assimile dans un algorithme dynamique. Que ce soit pour une assimilation temporelle ou pour un algorithme itératif, il faut fournir une valeur initiale que cette méthode permet de donner sous la forme d'un vecteur par un argument nommé "`asVector`".

La méthode "`setAlgorithmParameters`" permet de définir dans l'étude les paramètres de contrôle de l'algorithme sous la forme d'un dictionnaire fournit dans l'argument nommé "`asDico`".

La méthode "`setDiagnostic`" permet d'associer à l'étude des calculs de diagnostics. Ils sont à définir en demandant un diagnostic particulier présent dans la plateforme (dont on peut obtenir la liste par "`get_available_diagnostics()`") et en lui donnant un nom qui servira ensuite à le calculer ou le restituer.

Pour mémoire, on rappelle que les méthodes d'interrogation suivantes permettent d'avoir des informations exhaustives sur les algorithmes ou sur les résultats de calculs. Pour un objet étude nommée "`ADD`", on a :
- "`ADD.get()`" permet de restituer le dictionnaire des résultats globaux disponibles dans l'étude ;
- "`ADD.get_available_algorithms()`" permet de restituer une liste des noms d'algorithmes disponibles ;
- "`ADD.get_available_diagnostics()`" permet de restituer une liste des noms de diagnostics disponibles.

3.5 Variante utile : faire plusieurs calculs ou de la sensibilité

Sur la base de l'algorithme décrit dans le squelette 2 page 13, on peut facilement enrichir le calcul pour faire plusieurs calculs successifs dans le même étude et en tirer simplement des informations de sensibilité par exemple.

Comme les objets globaux contenant les résultats permettent de stocker de manière sucessive le résultat d'un calcul, on peut procéder de deux manières.

Si les changements ne sont pas à stocker, mais seulement le résultat d'assimilation, il suffit de changer les paramètres physiques modifiables pour ensuite relancer le calcul. Par exemple, si l'on change la matrice de covariance des erreurs d'ébauche :

```
... <Début identique au squelette>
#
ADD.analyze()
#
ADD.setBackgroundError    (asCovariance = "2 0 0;0 2 0;0 0 2")
#
ADD.analyze()
#
Xa = ADD.get("Analysis")
```

Dans ce cas, l'objet résultat d'analyse "`Xa`" contient deux pas dont chacun a été obtenu par un calcul BLUE avec une matrice de covariance différente. On peut donc effectuer ensuite des calculs d'analyse sur l'ensemble des pas, comme par exemple obtenir le vecteur moyen sur l'ensemble des vecteurs d'analyse en invoquant simplement la méthode adéquate "`stepmean`" de l'objet "`Xa`".

Si les changements en entrée sont à stocker, alors il faut utiliser un objet de stockage de la plateforme pour alimenter les opérations décrites dans l'exemple ci-dessus.

Dans tous les cas, l'enchaînement explicite des étapes d'analyse par le concepteur de l'étude permet de disposer à la sortie de l'ensemble des résultats correspondant aux changements imposés, résultats stockés dans un même objet d'analyse. Les calculs de type sensibilité en sont donc ensuite facilités.

4 Ecrire un nouvel outil de diagnostic

Un outil de diagnostic désigne un ensemble d'opérations d'analyse des données disponibles à travers la plateforme d'assimilation. Le résultat de ces analyses permet d'interpréter les données. Cela peut être un calcul simple (comme une norme, une RMS ou un indicateur statistique) ou une analyse plus compliquée, comme un test décisionnel, mais aussi des traitements comme un affichage.

L'écriture d'un nouveau diagnostic se fait, comme pour les algorithmes d'assimilation, en profitant d'un cadre de définition d'interface préparé pour faciliter l'intégration. La variété des diagnostics envisageables est ensuite très importante, et cela peut d'ailleurs facilement ne pas être un diagnostic au sens courant (comme un affichage par exemple).

Le présent tutorial vise à expliciter l'environnement d'écriture et à donner un exemple simple de diagnostic déjà existant dans la plateforme. A priori, ce document n'est pas destiné à un utilisateur débutant, qui pourra lui s'appuyer sur les outils de diagnostic déjà existants. Le tutorial présente progressivement les points suivants :

- le squelette standard d'un diagnostic et les explications ligne par ligne,
- les indications précises de ce que le programmeur doit modifier dans le squelette,
- la manière d'échanger et de stocker des variables,
- l'écriture du code propre du diagnostic,
- et la mise en place de tests du nouvel outil de diagnostic ainsi que de son utilisation.

4.1 Squelette standard d'outil de diagnostic et explications

Pour commencer un outil de diagnostic en Python, il suffit de se baser sur le squelette 3 en page 18, en le recopiant directement dans un fichier Python en "`.py`". On propose ici un exemple particulier de diagnostic, sachant que d'importantes variations peuvent être rencontrées en fonction de la diversité des diagnostics.

Le nom que l'on choisit pour le fichier Python servira ensuite comme nom d'appel de ce diagnostic à travers la plateforme. Il est donc conseillé de le choisir de manière adaptée et en accord avec les noms déjà existants. Attention, si le nom est le même que celui d'un diagnostic déjà présent, un seul des deux pourra être utilisé en analyse (la détermination de celui qui sera activé dépend de l'ordre des répertoires dans le "`path`" complet Python). On déconseille donc fortement de choisir un nom déjà existant.

On explique donc ce squelette 3 ligne à ligne ci-après. Tous les mots écrits exclusivement en majuscule sont des commentaires destinés à être remplacés par vos textes.

En ligne 1, on indique que l'encodage des caractères du fichier lui permet de contenir en particulier des accents dans le texte.

En lignes 2 à 4, on indique en texte clair des information générales sur le diagnostic, usuellement sous la forme d'un titre et de commentaires généraux sur l'objectif du diagnostic et ses effets prévus. On peut y rassembler la description détaillée, mais cette dernière peut aussi être indiquée dans la méthode activée "`_formula`"

En ligne 5, on indique l'auteur et la date de création ou de modification dans la variable "`__author__`".

En ligne 7, on met en place le chemin qui permet à Python de retrouver les fichiers des modules demandés sur les lignes 8 et 9 suivantes. Si ces modules sont à un autre endroit que le répertoire supérieur, il convient de l'indiquer à cet endroit.

```
 1 #-*-coding:iso-8859-1-*-
 2 __doc__ = """
 3     TITRE
 4 """
 5 __author__ = "AUTEUR - DATE"
 6
 7 import sys ; sys.path.insert(0, "../daCore")
 8 import Persistence
 9 from BasicObjects import Diagnostic
10
11 # ==================================================================
12 class ElementaryDiagnostic(Diagnostic,Persistence.TYPE):
13     def __init__(self, name = "", unit = "", basetype = None, parameters = {}):
14         Diagnostic.__init__(self, name, parameters)
15         Persistence.TYPE.__init__( self, name, unit, basetype = ONETYPE )
16
17     def _formula(self, ARGUMENTS_F):
18         """
19         DESCRIPTION SUCCINCTE DU DIAGNOSTIC
20         """
21         #
22         # CALCUL MATHEMATIQUE DU DIAGNOSTIC
23         #
24         return VALEUR
25
26     def calculate(self, ARGUMENTS_C = None, step = None):
27         """
28         DESCRIPTION SUCCINCTE DES VERIFICATIONS PREALABLES
29         """
30         #
31         # TEST DES ARGUMENTS
32         #
33         # ACTIVATION DE LA FORMULE DE CALCUL
34         #
35         value = self._formula( ARGUMENTS_F )
36         #
37         # STOCKAGE DU RESULTAT
38         #
39         self.store( value = value, step = step)
40
41 # ==================================================================
42 if __name__ == "__main__":
43     print "\nAUTOTEST\n"
```

FIG. 3 – Squelette standard d'un outil de diagnostic

En ligne 8, on requiert le module de persistence qui sera utile si l'on veut rajouter des variables stockées. Il est conseillé de faire cet appel par défaut, car il est très souvent utile.

En ligne 9, on requiert cette fois la classe générale de diagnostic, dont doit hériter tout diagnostic élémentaire.

En ligne 12, on définit une classe dont le nom "`ElementaryDiagnostic`" est impératif. Les mécanismes de la plateforme cherchent ce nom-là exclusivement. Cette classe doit aussi impérativement hériter de la classe "`Diagnostic`". Elle hérite ici de plus d'une sous-classe particulière de persistence, que l'on précise en choisissant le "`TYPE`" parmi les classes existantes (essentiellement "`OneScalar`", "`OneVector`", "`OneMatrix`", "`OneList)`"). Ce dernier héritage présente ensuite l'intéret d'appliquer les méthodes standards de persistence directement sur l'objet diagnostic, mais on pourrait aussi stocker les valeurs persistentes dans une variable interne de l'objet pour l'utiliser ensuite.

En ligne 13, on définit la méthode "`__init__`" de la classe. Cette méthode doit contenir 4 paramètres en entrée et aucun en sortie. Tous les paramètres en entrée sont indiqués avec des valeurs par défaut qui ne présupposent pas de choix si elles ne sont pas données (chaine vide ou `None`). Le premier argument "`name`" permet de donner un nom à l'objet de diagnostic, nom qui sera ensuite utilisé pour accéder à ce diagnostic dans l'étude d'assimilation. Les deux arguments suivants "`unit`" et "`basetype`" sont utilisés pour construire l'objet de persistence. Ces arguments sont indispensables meme si la classe n'hérite pas d'un objet de persistence. Le dernier "`parameters`" est un dictionnaire quelconque, ce qui permet de passer en argument d'un diagnostic des variables nommées en nombre quelconque. Attention, ce dictionnaire ne doit par contre pas être une manière de passer les arguments de la formule de calcul.

En ligne 14, on appelle explicitement l'initialisation de la classe "`Diagnostic`" dont hérite la présente classe "`ElementaryDiagnostic`". Il y a deux arguments, une chaine de caractères donnant le nom dans "`name`", et un dictionnaire quelconque "`parameters`" dont le contenu est ensuite disponible dans la variable interne au diagnostic "`self.parameters`". Si cet argument est utilisé, il faut vérifier juste après si les arguments requis par le diagnostic sont bien dans le dictionnaire "`self.parameters`".

A l'issue de cette ligne 14, "`self`" dispose de deux attributs nouveaux "`name`" et "`self.parameters`".

En ligne 15, on effectue la meme opération d'initialisation explicite pour la seconde classe dont hérite le diagnostic, "`Persistence.TYPE`". C'est à cet endroit qu'il est impératif de déclarer le type "`ONETYPE`" de la variable stockée à chaque pas. Ce type peut être soit fixé en dur (`float`, `int`, `bool`...), soit reprendre l'argument de la méthode "`__init__`" en ligne 13.

En ligne 17 se trouve l'appel "`_formula`", qui est la méthode spécifique de calcul mathématique du diagnostic de la classe. Cette méthode ne doit pas être appellée sur l'objet car elle est activée en interne par la méthode "`calculate`", dont le but est de valider a priori les arguments. La séparation de la méthode "`_formula`" permet de bien identifier les formules de traitement ou de calcul sur les données, sous la forme la plus mathématique possible. Les arguments "`ARGUMENTS_F`" de cette méthode doivent être des objets adaptés (par exemple de type "`numpy`") au traitement mathématique développé ensuite.

En lignes 18 à 20 se trouve le commentaire général de la méthode, qu'il est vivement conseillé de remplir avec une description précise de l'aspect mathématique du diagnostic mis en oeuvre dans cette classe.

En lignes 21 à 23, on trouve l'emplacement qui contiend explicitement le calcul du diagnostic. Ce dernier peut utiliser en particulier les modules "`numpy`" et "`scipy`", qui contiennent beaucoup d'outils numériques, statistiques...

En ligne 24, on retourne la valeur calculée pour le diagnostic élémentaire. La valeur simple ou complexe retournée sera traitée pour stockage dans la méthode "`calculate`" d'appel interne, donc il suffit de la retourner ici.

En ligne 26 se trouve l'appel "`calculate`", qui est la méthode qui permet d'activer le diagnostic de la classe. Cette méthode doit contenir la vérification des variables qui sont passées

en argument par l'utilisateur, et faire les transformations nécessaires à leur usage de type mathématique dans la formule appellée en interne ensuite. Les arguments "`ARGUMENTS_C`" de cette méthode sont quelconques, mais il vaut mieux qu'ils permettent des tests Python significatifs (de type, de valeur, de présence...). L'argument "`step`" permet éventuellement de piloter le stockage pour le faire à des pas particuliers.

En ligne 31, on trouve l'emplacement des tests Python des arguments. Plus ces tests sont précis et développés, moins l'utilisateur aura de chance de rencontrer des cas délicats d'utilisation de diagnostic.

En lignes 33 à 35, on active de manière interne le calcul du diagnostic et on récupère sa valeur. Les arguments "`ARGUMENTS_F`" passés à la fonction "`_formula`" doivent avoir été préparés dans "`calculate`" pour permettre un usage simple dans "`_formula`".

En lignes 37 à 39 se trouve le stockage de la valeur du diagnostic, sous forme standard d'un objet de persistance associé à "`self`" puisque la classe présente dérive d'une classe de persistence aussi.

Enfin en lignes 42 et 43, on dispose de l'amorce simple de tests unitaires intégrés dans le fichier. Même si aucun test unitaire n'est directement intégré au fichier, il est conseillé de laisser ces deux lignes pour faciliter les diagnostics automatiques.

4.2 Précisions sur les parties à modifier du squelette

Explicitement, le programmeur **doit modifier le contenu des lignes 3, 5, 12, 15, 19, 21, 23 à 26, 28, 30, 32 à 40**.

En lignes 3, 5, 21 et 30, ce sont des commentaires clairs en français qui sont attendus.

En lignes 12 et 15, c'est le type de persistence qui est attendu.

En lignes 19 et 28, ce sont les arguments nommés, avec leurs valeurs par défaut, qui sont attendus.

En lignes 23 à 26, c'est le code mathématique proprement dit du diagnostic qui est attendu.

En lignes 32 à 40, ce sont les tests et les transformations des arguments, et le stockage des résultats, qui sont attendus.

Il peut aussi être nécessaire de compléter le code en rajoutant des informations après la ligne 17 (pour l'initalisation) et après la ligne 41 (pour mettre des tests). De plus, des diagnostics complexes peuvent conduire à rajouter des méthodes

4.3 Echanger et stocker des variables

Avant même de coder le diagnostic, il est important de savoir comment disposer des données nécessaires, et comment stocker les variables calculées utiles après le déroulement du calcul.

Il y a deux manières (détaillées juste après les deux points) pour échanger des informations, qui sont liées au fait que les données soient en entrée ou en sortie :

1. en **entrée**, les variables sont disponibles dans les arguments d'interface "`parameters`" de la méthode d'initialisation, et dans les arguments "`ARGUMENTS_C`" de la méthode "`calculate`" ;

2. en **sortie**, les variables peuvent être stockées dans un objet persistent qui est le diagnostic lui-même ici, ou dans une variable interne persistente du diagnostic.

Les arguments d'initialisation "`parameters`" sont à privilégier pour toute variable utile au diagnostic, et ne changeant pas à chaque calcul du diagnostic à un pas différent. Cela peut par exemple être un nombre de classes statistiques, un critère de convergence...

Les arguments "`ARGUMENTS_C`" de la méthode "`calculate`" doivent permettre à l'utilisateur de fournir à chaque pas de calcul nécessaire les données permettant d'évaluer le diagnostic. La vérification et la mise en forme de ces arguments est faite dans la méthode "`calculate`", pour les passer à la méthode "`_formula`".

Les arguments "`ARGUMENTS_F`" de la méthode "`_formula`" doivent par contre être adaptés à un calcul du diagnostic le plus simple et explicite possible mathématiquement. La lisibilité des formules codées dans "`_formula`" est un gage de vérification très appréciable.

On insiste sur le fait que la méthode "`calculate`" est la seule qui soit publique et doive être vue par l'utilisateur.

4.4 Le code propre du diagnostic sur les données

La partie proprement mathématique du diagnostic est à rajouter dans la méthode "`_formula`" à la ligne 24 déjà indiquée. Les arguments sont définis et traités dans l'entête de la méthode "`calculate`" et peuvent être quelconques.

Un exemple simple permet d'illustrer l'écriture d'un diagnostic, avec son stockage et les conversions initiales nécessaires au calcul. L'exemple traité est celui d'un calcul de RMS de l'écart entre deux vecteurs, que l'on extrait pour le commenter.

```
.... def _formula(self, V1, V2):
.... .... """
.... .... Fait un écart RMS entre deux vecteurs V1 et V2
.... .... """
.... .... rms = math.sqrt( ((V2 - V1)**2).sum() / float(V1.size) )
.... .... #
.... .... return rms
.... ....
.... def calculate(self, vector1 = None, vector2 = None, step = None):
.... .... """
.... .... Teste les arguments, active la formule et stocke le résultat
.... .... """
.... .... if vector1 is None or vector2 is None:
.... .... .... raise ValueError("Two vectors must be given to \
                                            calculate their RMS")
.... .... V1 = numpy.array(vector1)
.... .... V2 = numpy.array(vector2)
.... .... if V1.size < 1 or V2.size < 1:
.... .... .... raise ValueError("The given vectors must not be empty")
.... .... if V1.size != V2.size:
.... .... .... raise ValueError("The two given vectors must have the \
                        same size, or the vector types are incompatible")
.... .... #
.... .... value = self._formula( V1, V2 )
```

```
.... .... #
.... .... self.store( value = value, step = step)
```

La méthode "`_formula`" contient très simplement l'écriture mathématique de la RMS de l'écart des deux vecteurs V_1 et V_2 de longueur n :

$$\mathrm{RMS}(V_1, V_2) = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(V_2 - V_1)^2}$$

On remarque immédiatement l'intérêt de bien identifier le calcul du diagnostic sous forme mathématique car sa vérification est alors facile.

La méthode "`calculate`" contient elle la vérification de la présence et du type des arguments, qui doivent être compatibles avec le constructeur "`numpy.array`". Les arguments sont transformés pour être bien assuré de leur type, et le calcul est lancé à la fin, avec le résultat stocké dans l'objet "`self`".

4.5 Tester et utiliser le nouveau diagnostic

Puisque le module Python créé sur la base du squelette de la partie 3 en page 17, un de ses intérêts est d'être autoportant. Dans ce cas, on peut utiliser la démarche habituelle en Python pour effectuer des tests unitaires associés au module lui-même. C'est la raison des deux dernières lignes du squelette, qui permettent de lancer des tests situés après ces commandes lorsque le module est appelé seul. On peut aussi sans difficulté établir des tests externes se basant sur le module seul.

Pour utiliser ce nouveau diagnostic dans le cadre de la plateforme d'assimilation, il faut le placer dans l'un des répertoires "`daDiagnostics`" prévu par défaut pour le stockage des diagnostics élémentaires. Il suffit ensuite de passer son nom de fichier (sans le suffixe "`.py`"), suivi du nom du diagnostic, en argument de la méthode "`setDiagnostic`" d'une étude d'assimilation de type "`AssimilationStudy`".

5 Ecrire un nouvel algorithme d'assimilation

L'écriture d'un nouvel algorithme dans la plateforme d'assimilation se fait en profitant d'un cadre de définition d'interface préparé pour faciliter l'intégration. La variété des algorithmes envisageables est ensuite très importante, et cela peut d'ailleurs facilement ne pas être un algorithme d'assimilation.

Le présent tutorial vise à expliciter l'environnement d'écriture et à donner un exemple simple d'algorithme déjà existant dans la plateforme. A priori, ce document n'est pas destiné à un utilisateur débutant, qui pourra lui s'appuyer sur les algorithmes déjà existants. Le tutorial présente progressivement les points suivants :
- le squelette standard d'algorithme et les explications ligne par ligne,
- les indications précises de ce que le programmeur doit modifier dans le squelette,
- la manière d'échanger et et de stocker des variables,
- l'écriture du code propre de l'algorithme d'assimilation,
- et la mise en place de tests du nouvel algorithme ainsi que son utilisation.

5.1 Squelette standard d'algorithme et explications

Pour commencer un algorithme en Python, il suffit de se baser sur le squelette 4 en page 24, en le recopiant directement dans un fichier Python en "`.py`".

Le nom que l'on choisit pour le fichier Python servira ensuite comme nom d'appel de cet algorithme à travers la plateforme. Il est donc conseillé de le choisir de manière adaptée et en accord avec les noms déjà existants. Attention, si le nom est le même que celui d'un algorithme déjà présent, un seul des deux pourra être utilisé en analyse (la détermination de celui qui sera activé dépend de l'ordre des répertoires dans le "`path`" complet Python). On déconseille donc fortement de choisir un nom déjà existant.

On explique donc ce squelette 4 ligne à ligne ci-après. Tous les mots écrits exclusivement en majuscule sont des commentaires destinés à être remplacés par vos textes.

En ligne 1, on indique que l'encodage des caractères du fichier lui permet de contenir en particulier des accents dans le texte.

En lignes 2 à 4, on indique en texte clair des information générales sur l'algorithme, usuellement sous la forme d'un titre et de commentaires généraux sur l'objectif de l'algorithme et ses effets prévus. On peut y rassembler la description détaillée, mais cette dernière peut aussi être indiquée dans la méthode activée "`run`"

En ligne 5, on indique l'auteur et la date de création ou de modification dans la variable "`__author__`".

En ligne 7, on met en place le chemin qui permet à Python de retrouver les fichiers des modules demandés sur les lignes 8 et 9 suivantes. Si ces modules sont à un autre endroit que le répertoire supérieur, il convient de l'indiquer à cet endroit.

En ligne 8, on requiert le module de persistence qui sera utile si l'on veut rajouter des variables stockées. Il est conseillé de faire cet appel par défaut, car il est très souvent utile.

En ligne 9, on requiert cette fois la classe générale d'algorithme, dont doit hériter tout algorithme élémentaire.

En ligne 12, on définit une classe dont le nom "`ElementaryAlgorithm`" est impératif. Les mécanismes de la plateforme cherchent ce nom-là exclusivement. Cette classe doit aussi impérativement hériter de la classe "`Algorithm`".

```
 1 #-*-coding:iso-8859-1-*-
 2 __doc__ = """
 3     TITRE
 4 """
 5 __author__ = "AUTEUR - DATE"
 6
 7 import sys ; sys.path.insert(0, "../daCore")
 8 import Persistence
 9 from BasicObjects import Algorithm
10
11 # ===================================================================
12 class ElementaryAlgorithm(Algorithm):
13     def __init__(self):
14         Algorithm.__init__(self)
15         self._name = "MON_ALGORITHME"
16
17     def run(self,Xb=None,Y=None,H=None,M=None,R=None,B=None,Q=None,Par=None ):
18         """
19         DESCRIPTION SUCCINCTE DE L'ALGORITHME
20         """
21         #
22         # LE CODE
23         #
24         # STOCKAGE DES VARIABLES
25         #
26         return 0
27
28 # ===================================================================
29 if __name__ == "__main__":
30     print "\nAUTOTEST\n"
```

Fig. 4 – Squelette standard d'algorithme d'assimilation

En ligne 13, on définit la méthode "`__init__`" de la classe. Cette méthode ne doit pas contenir de paramètres en entrée ou en sortie. S'il y a des arguments en entrée, ils sont ignorés et non renseignés.

En ligne 14, on appelle explictement l'initialisation de la classe "`Algorithm`" dont hérite la présente classe "`ElementaryAlgorithm`".

En ligne 15, on indique en clair le nom succinct de l'algorithme. Ce champ est utilisé uniquement pour les commentaires, et il vaut mieux le choisir court.

En ligne 17 se trouve l'appel "`run`", qui est la principale méthode de la classe. C'est la seule qui sera activée par les mécanismes d'assimilation. Elle reçoit en argument un ensemble imposé de variables, indiqué dans la parenthèse et mises par défaut à la valeur vide "`None`". Les arguments indiqués sont impératifs.

En lignes 18 à 20 se trouve le commentaire général de la méthode, qu'il est vivement conseillé de remplir avec une description précise de l'algorithme mis en oeuvre dans cette classe.

En lignes 21 à 23, on trouve l'emplacement qui contiend explicitement le code de l'algo-

rithme. Ce dernier peut utiliser en particulier les modules "`numpy`" et "`scipy`", qui contiennent beaucoup d'outils numériques.

En ligne 24, il est proposé de socker de manière persistante (sur les pas de temps ou sur ceux d'itération sans aspect temporel) les variables nécessaires. On discurera plus loin (partie 5.3 en page 25) du stockage en détail.

En ligne 26 est indiqué en retour le code retour de la méthode, qui est 0 par défaut lorsque la méthode s'est correctement déroulée.

Enfin en lignes 29 et 30, on dispose de l'amorce simple de tests unitaires intégrés dans le fichier. Même si aucun test unitaire n'est directement intégré au fichier, il est conseillé de laisser ces deux lignes pour faciliter les diagnostics automatiques.

5.2 Précisions sur les parties à modifier du squelette

Explicitement, le programmeur **doit modifier le contenu des lignes 3, 5, 15, 19, et 22 à 24**.

En lignes 3, 5, 19, ce sont des commentaires clairs en français qui sont attendus.

En ligne 15, c'est un nom simple et signifiant pour l'algorithme qui est attendu, avec les caractères autorisés "`[a-zA-Z0-9]`".

En lignes 22 à 24, c'est le code proprement dit qui est attendu.

Il peut aussi être nécessaire de compléter le code en rajoutant des informations après la ligne 15 (pour l'initalisation) et après la ligne 30 (pour mettre des tests).

5.3 Echanger et stocker des variables

Avant même de coder l'algorithme dans la méthode "`run`", il est important de savoir comment disposer des données nécessaires, et comment stocker les variables calculées utiles après le déroulement du calcul.

Il y a deux manières (détaillées juste après les deux points) pour échanger des informations, qui sont liées au fait que les données soient en entrée ou en sortie :

1. en **entrée**, les variables sont disponibles dans la liste fixe des arguments d'interface de la méthode "`run`" ;
2. en **sortie**, les variables doivent être stockées dans des objets persistents, qui sont tous contenus dans le dictionnaire spécial "`self.StoredVariables`".

La liste des **variables en entrée** est limitative pour permettre son activation automatique par les mécanismes de la plateforme. Par contre, les arguments peuvent être compliqués, comme en particulier des variables dépendantes du temps. Par défaut, toutes les variables peuvent être temporelles. De plus, l'un des arguments est un dictionnaire extensible contenant les paramètres utilisateurs de l'algorithme. La liste des arguments nommés de la méthode "`run`" sont indiqués dans le tableau 5 suivant.

Pour les **variables en sortie**, la classe mère "`Algorithm`" prépare par défaut le dictionnaire "`self.StoredVariables`" pour le stockage d'un certain nombre de variables habituelles, décrites ci-aprés dans le tableau 6 en page 26. De nouvelles variables stockables peuvent être rajoutées, en complétant ce dictionnaire "`self.StoredVariables`" dans la méthode d'initialisation "`__init__`" de la classe "`ElementaryAlgorithm`", après l'appel de l'initialisation

Symbole	*Nom et contenu*
Xb	Vecteur d'ébauche
Y	Vecteur d'observations
H	Opérateur d'observation
M	Opérateur d'évolution (modèle)
R	Covariance des erreurs d'observation
B	Covariance des erreurs d'ébauche
Q	Covariance des erreurs du modèle d'évolution
Par	Dictionnaire de paramètres supplémentaires de l'algorithme

FIG. 5 – Inventaire des variables en entrée prévues par défaut

de la classe mère "`Algorithm`". L'accès à ces variables dans "`self.StoredVariables`" se fait à la manière d'un dictionnaire, par exemple "`self.StoredVariables["Analysis"]`".

Nom	*Symbole*	*Stockage à chaque pas d'algorithme*
Analysis	Xa	Vecteur d'analyse
Innovation	d	Vecteur d'innovation
...	...	...

FIG. 6 – Inventaire des variables persistentes prévues par défaut

En entrée comme en sortie, les variables sont des objets qui présentent des méthodes de transformation et d'adaptation. En particulier, ils peuvent se mettre à disposition sous une forme adaptée au calcul numérique algébrique. Pour ceux qui ne sont pas directement des objets "`numpy`", il existe par exemple une méthode "`asMatrix()`" pour les opérateurs.

De plus, la classe mère "`Algorithm`" donne à un algorithme quelconque une méthode "`get`" qui permet de restituer une variable stockée, ou leur ensemble si "`get`" n'a pas d'argument.

5.4 Le code propre de l'algorithme d'assimilation

La partie proprement assimilation du code est à rajouter dans la méthode "`run`" à la ligne 22 déjà indiquée.

Les objets en entrée se présentent sous les formes les plus adaptées au calcul mathématique que l'on rencontre dans les algorithmes théoriques, à savoir du calcul matriciel ou de l'algèbre d'opérateurs. En particulier, les opérateurs peuvent se présenter sous forme matricielle ou fonctionnelle. Il est donc recommandé d'utiliser la forme la plus adaptée à l'écriture naturelle de l'algorithme.

On peut en particulier utiliser des variables temporaires pour rendre le code plus explicite, sachant que le rangement d'une variable complexe (comme une liste ou un dictionnaire) dans une variable temporaire ne conduit pas à une copie, mais simplement utiliser le pointeur vers l'objet complexe dans la variable temporaire.

Enfin, de manière générale, un examen attentif du code d'assimilation peut être nécessaire pour assurer l'empreinte mémoire la plus réduite possible, en évitant les duplications inutiles, préjudiciables à l'éfficacité mémoire et même en temps de résolution.

Un exemple permet d'illustrer l'écriture d'un algorithme simple, avec son stockage et les

conversions initiales nécessaires au calcul (matriciel dans ce cas). L'exemple traité est celui de la partie essentielle d'un algorithme BLUE, que l'on extrait pour le commenter.

```
.... .... Hm = H["Direct"].asMatrix()
.... .... Ht = H["Adjoint"].asMatrix()
.... ....
.... .... K  = B * Ht * (Hm * B * Ht + R).I
.... .... d  = Y - Hm * Xb
.... .... Xa = Xb + K*d
.... ....
.... .... self.StoredVariables["Analysis"].store( Xa.A1 )
.... .... self.StoredVariables["Innovation"].store( d.A1 )
```

Les deux premières lignes permettent d'obtenir la forme matricielle des opérateurs d'observation direct "`Hm`" et adjoint "`Ht`".

Les trois lignes suivantes effectuent l'analyse BLUE proprement dite, pour obtenir la solution "`Xa`". Elles sont écrites de manière quasiment similaire à l'expression algébrique de l'analyse BLUE (la notation "`.I`" indique l'inversion d'une matrice). Les calculs présentés reposent entièrement sur le fait que ce sont des objets vectoriels ou matriciels "`numpy`" cohérents entre eux. Cette cohérence est celle que l'on obtient si l'on range les variables selon leurs caractéristiques "naturelles" d'algèbre linéaire.

Les deux lignes suivantes sont celles qui activent le stockage, en faisant intervenir un appel à la méthode "`store`" des objets persistents. Les deux variables nommées, "`Analysis`" et "`Innovation`", ont été prévues par défaut dans la classe mère "`Algorithm`", comme décrit dans le tableau 6 en page 26. La notation "`.A1`" indique que l'objet "`numpy`" est renvoyé sous la forme d'un "`1-d array`", ce qui est nécessaire pour le stockage d'un vecteur.

5.5 Tester et utiliser le nouvel algorithme

Puisque le module python créé sur la base du squelette de la partie 5.1 en page 23, un de ses intérêts est d'être autoportant. Dans ce cas, on peut utiliser la démarche habituelle en Python pour effectuer des tests unitaires associés au module lui-même. C'est la raison des deux dernières lignes du squelette, qui permettent de lancer des tests situés après ces commandes lorsque le module est appelé seul. On peut aussi sans difficulté établir des tests externes se basant sur le module seul.

Pour utiliser ce nouvel algorithme dans le cadre de la plateforme d'assimilation, il faut le placer dans l'un des répertoires "`daAlgorithms`" prévu par défaut pour le stockage des algorithmes élémentaires. Il suffit ensuite de passer son nom de fichier (sans le suffixe "`.py`") en argument de la méthode "`setAlgorithm`" d'une étude d'assimilation de type "`AssimilationStudy`".

Index

A

Algorithm 23
AsCovariance 14
AsDico 15
AsMatrix 14, 26
AssimilationStudy 22
AssimilationStudy.log 5
AsVector 14

B

basetype 9
BLUE 12

C

calculate 19
CRITICAL 4

D

DaAlgorithms 27
daDiagnostics 22
DEBUG 4
Diagnostic 17, 19

E

ElementaryAlgorithm 23
ElementaryDiagnostic 19
ERROR 4

F

formula 17

G

get 15
get_available_algorithms 14, 15
get_available_diagnostics 15
Gnuplot 3

I

INFO 4
item 10

L

Level 5
Logging 4

M

Matplotlib 3

N

name 9
Niveaux d'affichage 4
NOTSET 4
Numpy 3
numpy.array 9
numpy.matrix 9

O

OneList 8, 19
OneMatrix 8, 19
OneScalar 8, 9, 19
OneVector 8, 19

P

Persistence 8

S

SALOME 3
Scipy 3
setAlgorithmParameters 15
SetBackground 13
SetBackgroundError 13
setControls 14
setDiagnostic 15, 22
SetEvolutionError 14
SetEvolutionModel 13
SetObservation 13
SetObservationError 13
SetObservationOperator 13
step 9, 10
Stepmean 15
stepnumber 10
stepserie 10
StoredVariables 25

U

unit 9

V

value 9
Valueserie 14
valueserie 10

W

WARNING 4