OpenFLUID
Software Environment
for Modelling Fluxes in Landscapes

TP8 : Encapsulation de codes existants

Objectifs:	Convertir un code, existant et autonome (Fortran90 ou C), en une fonction de simulation OpenFLUID
Pré-requis:	TP1 à TP5

1 Introduction

Dans ce TP, deux codes existants (Fortran90 et C) sont considérés. Ils permettent de représenter l'évolution de l'humidité d'une colonne de sol en fonction de sa profondeur par analogie réservoir. Il s'agit de convertir un de ces codes de calculs (au choix Fortran90 ou C) en fonctions de simulation OpenFLUID. Il sera ainsi possible de les coupler par la suite à d'autres fonctions de simulations au sein d'un même modèle.

2 Présentation des codes existants

2.1 Principe du modèle réservoir

Le principe de ce modèle est de considérer une colonne de sol comme un empilement de réservoirs. Quand une quantité d'eau est apportée en surface du sol, le premier réservoir se remplit jusqu'à atteindre sa capacité au champ (`th_cc` dans le code), puis le surplus est drainé vers le second réservoir. Le processus est itéré tant qu'il reste un volume d'eau à transmettre. Le volume restant, une fois tous les réservoirs saturés, correspond au volume drainé sous la colonne (voir figure 1).

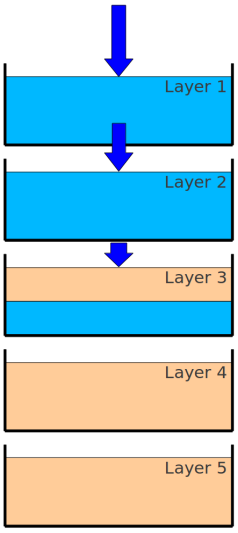

Figure 1 – illustration des principes du modèle réservoir. Ici, les 5 réservoirs ont tous la même capacité et l'apport d'eau correspond à la moitié de la capacité totale de la colonne : le premier et le deuxième réservoir se remplissent jusqu'à atteindre leur capacité maximale et le reste de l'apport d'eau draine dans le troisième réservoir.

Le modèle réservoir est parmi les modèles les plus simples représentant l'infiltration d'une quantité d'eau dans le sol. Il est développé dans un contexte 1D, n'est pas spatialisé et ne dépend pas directement du temps (le temps intervient indirectement au travers des chroniques d'apport d'eau en surface). Les fonctionnalités d'OpenFLUID seront exploitées pour distribuer ce modèle sur différentes unités spatiales (SU) et les faire évoluer dans un contexte temporel plus rigoureux.

2.2 Compilation et lancement des codes sources

Les codes sources sont disponibles dans les fichiers fournis :

- dans `src/training.encapsulation.reservoir-fortran/ReservoirFortran.f90` pour le code Fortran,
- dans `src/training.encapsulation.reservoir-c/ReservoirC.c` pour le code C.

Ils lisent en entrée les fichiers :

- `datasets/TP8/input.txt` renseignant les caractéristiques du sol et
- `datasets/TP8/Pluvio_3_1997_06_05.txt` renseignant la chronique des apports d'eau dans le temps.

Ils produiront en sortie, une fois compilé et lancé, le fichier :

- `outputs/TP8/out_fortran.txt` ou `outputs/TP8/out_c.txt` décrivant l'humidité du sol et le drainage cumulé en fin de simulation.

La commande suivante (illustrée ici, et dans le reste du document, pour le code C, mais la démarche est équivalente pour le code Fortran90) permet de compiler et lancer ces codes de calculs :

```
cd /home/openfluid/formation/src/training.encapsulation.reservoir-c
make run
```

Action à réaliser : lancer le code et observer les fichiers d'entrée, source et de sortie

2.3 Analyse et préparation des codes existants

Afin de permettre une communication cohérente entre les différentes fonctions à coupler, OpenFLUID prend en charge la gestion des boucles de temps, la distribution des actions de simulations sur les unités spatiales, la lecture et l'écriture des données d'entrée et de sortie. Tous ces éléments doivent donc être éliminés du code à encapsuler et remplacer par des appels et des déclarations OpenFLUID (voir figure 2).

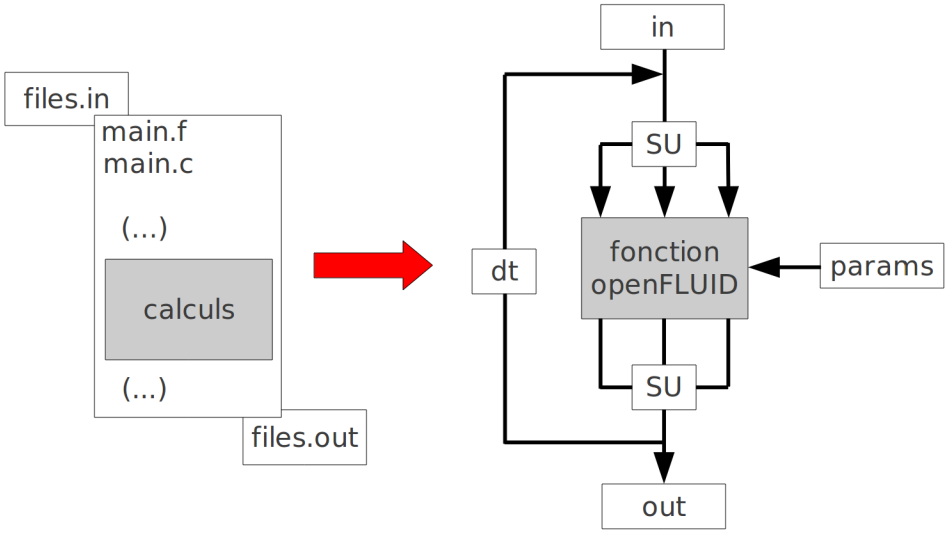

Figure 2 – le code original (à gauche) est analysé et décomposé de manière à isoler la partie "calculs". Seule cette partie "calculs" du code original est conservée dans la fonction de simulation OpenFLUID, le reste du code original (pré-, post-traitement, initialisation, boucle sur le temps et les unités spatiales,...) est confié au contexte OpenFLUID (à droite).

Cette analyse est une étape primordiale de l'encapsulation, elle permet d'isoler la boucle de calcul et de définir les paramètres, les variables requises et produites de la fonction de simulation OpenFLUID. Pour réaliser pas à pas cette étape, le code de calcul sera dans un premier temps modulariser dans son propre langage.

Action à réaliser : isoler la partie "calculs" dans une fonction indépendante `ReservoirCCalc.c` et réorganiser le code source initial en un programme principal `ReservoirCMain.c` faisant appel à cette fonction.

Une version modularisée du code est disponible au même emplacement que le code initial. Elle peut être compilée et exécutée par la commande :

```
cd /home/openfluid/formation/src/training.encapsulation.reservoir-c
make run2
```

Nous disposons désormais d'une fonction indépendante `ReservoirCCalc.c` qui gère uniquement la partie calculs du modèle. Il s'agit maintenant de développer une fonction de simulation OpenFLUID qui renseignera les arguments nécessaires à cette fonction, y fera appel et rendra

disponible les sorties du modèle à d'éventuelles autres fonctions de simulation.

3 Création de la fonction de simulation OpenFLUID

3.1 Stratégie retenue pour l'encapsulation du modèle réservoir

De nombreuses solutions sont possibles pour réaliser l'encapsulation du modèle réservoir décrit plus haut. La solution proposée ici est une solution relativement simple qui peut être mise en oeuvre rapidement dans la plate-forme OpenFLUID.

3.2 Génération de la fonction

Nous allons créer un projet Eclipse et générer la fonction au travers du plugin OpenFLUID pour Eclipse, en appliquant la démarche proposée dans le TP1. Cette fonction devra avoir comme caractéristiques :

- ID : `training.encapsulation.reservoir-c`
- Fichier .cpp : `ReservoirCFunc.cpp`
- Classe de la fonction : `ReservoirCFunction`

3.3 Signature

Cette fonction nécessite en entrée et produit en sortie les mêmes paramètres et variables que le code initial, à savoir en paramètres d'entrée :

- `th_cc` pour la capacité au champ des SU
- `thetai` pour la distribution de l'humidité initiale des SU
- `dz` pour la distribution des épaisseurs des réservoirs des SU

et en variables de sortie :

- `theta` pour la distribution de l'humidité par SU
- `drainage` pour la drainage sous la colonne par SU
- `storage` pour le stock par SU

Une fois complétée, la fin de la signature pourra être similaire à :

```
BEGIN_SIGNATURE_HOOK
  ...
  DECLARE_REQUIRED_INPUTDATA("th_cc","SU","Field␣capacity","m3/m3")
  DECLARE_REQUIRED_INPUTDATA("thetai","SU","Initial␣water␣content","m3/m3")
  DECLARE_REQUIRED_INPUTDATA("dz","SU","Soil␣layer␣depth","m")

  DECLARE_PRODUCED_VAR("water.surf-uz.theta[]","SU","Soil␣moisture␣profile","m3/m3")
  DECLARE_PRODUCED_VAR("water.surf-uz.drainage","SU","Drainage␣below␣soil␣profile","m")
  DECLARE_PRODUCED_VAR("water.surf-uz.storage","SU","Water␣storage␣in␣the␣whole␣soil␣column","m")
  DECLARE_REQUIRED_VAR("water.atm-surf.H.rain","SU","Produced␣variable,␣distributed␣on␣SU␣-␣rainfall␣he

END_SIGNATURE_HOOK
```

Note: notez la présence des crochets "[]" pour la déclaration d'une variable de type vecteur et nous verrons plus tard dans ce document pourquoi ils ne sont pas nécessaires lors de la lecture des paramètres spatialisés

3.4 Le développement de la fonction

Le TP proposé ici vous laisse une plus grande autonome pour le développement. En vous aidant des TPs précédents et des précisions apportées dans la suite de ce document, vous serez en mesure d'encapsuler le modèle réservoir dans une fonction OpenFLUID.

3.4.1 La lecture des données d'entrée du modèle

Nous avons tout d'abord choisi de définir un certain nombre de variables du modèle comme attribut de la fonction de simulation `ReservoirCFunction`. Il s'agit respectivement du nombre de réservoirs, de la capacité au champ, de l'humidité initiale et de l'épaisseur des réservoirs de chaque SU :

```
class ReservoirCFunction : public openfluid::base::PluggableFunction
{
private:

        int m_nCouche;
        openfluid::core::IDDoubleValueMap m_th_cc;
        openfluid::core::IDVectorValueMap m_thCI;
        openfluid::core::IDVectorValueMap m_dz;
}
```

La définition de ces nouveaux types introduits est disponible sur la documentation de l'API de développement (http://www.umr-lisah.fr/openfluid/resources/docs/manuals/en/openfluid/1.7.0/sdk/index.html). Ces nouveaux types permettent de spatialiser certaines données du modèle (nous n'avons pas jugé utile de spatialiser le nombre de réservoirs par SU). Ainsi, l'accès à la capacité au champ de la SU 2 et l'épaisseur du 1er réservoir de la SU 5 peuvent se faire en appelant :

```
m_th_cc[2] = 0.4;
m_dz[5][1] = 0.1;
```

Les valeurs à attribuer à ces variables sont lus par OpenFLUID dans les fichiers décrivant le domaine d'étude. Les applications de ce TP s'appuyeront sur les SUs du "Bassin Versant TP" utilisé dans les TPs précédents. Vous trouverez ces données dans `datasets/TP8`. Le fichier `SU.ddata.fluidx` a été complété avec les caractéristiques des réservoirs (épaisseur `dz`, capacité au champ `th_cc` et humidité initiale `thetai`).

```
<?xml version="1.0" standalone="yes"?>
<openfluid>
 <domain>
  <inputdata unitclass="SU" colorder="area;slope;flowdist;s;dz;th_cc;thetai" >
1  54.7  0.2  1.6  1.5  0.1;0.1;0.1;0.1;0.2  0.4  0.3;0.3;0.3;0.2;0.1
...
  </inputdata>
 </domain>
</openfluid>
```

Les champs `dz` et `thetai` sont renseignés sous forme de chaînes de caractères qu'OpenFLUID interprétera sous forme de vecteurs. Les commandes suivantes permettent de réaliser cette opération :

```
openfluid::core::StringValue str_dz;
openfluid::core::VectorValue dzVect;

OPENFLUID_GetInputData(SU,"dz",str_dz);
```

```
str_dz.toVectorValue(";",dzVect);

m_nCouche = dzVect.getSize();
```

Enfin, l'apport d'eau en surface est donné par une fonction de simulation OpenFLUID de type générateur. Son utilisation dans un modèle couplé peut soit se faire grâce à l'interface graphique dans l'onglet "Model", soit en ajoutant les lignes suivantes dans les fichiers `.fluidx` :

```
<model>
  <generator varname="water.atm-surf.H.rain" unitclass="SU" method="interp">
    <param name="distribution" value="SUraindistri.dat"/>
    <param name="sources" value="rainsources.xml"/>
  </generator>
  <function fileID="training.encapsulation.reservoir-fortran">
  </function>
</model>
```

Cette fonction prend en argument deux fichiers qui sont fournis dans le dataset du TP et renvoie au fichier `Pluvio_3_1997_06_05.txt` utilisé par le code initial. Pour plus de précision, on pourra se référer à la documentation de la fonction de simulation `water.atm-surf.rain-su.files` sur le dépôt de fonctions (http://www.umr-lisah.fr/repos2web/) qui fonctionne de manière équivalente.

3.4.2 Particularités de l'encapsulation d'un code C

Pour encapsuler un code C, des déclarations spécifiques sont utilisées :

- déclaration d'une fonction externe, à insérer après la liste des headers :

```
#include <openfluid/base.hpp>
#include <openfluid/core.hpp>

extern "C"
{
#include "ReservoirCCalc.h"
}
```

- appel de la fonction dans la méthode `runStep()` :

```
ReservoirCCalc(<liste des arguments>);
```

- ajout de la fonction externe dans la liste des fichiers à compiler dans le fichier `CMake.in.config` :

```
# list of CPP files. the func2doc tag must be contained in the first one
SET(FUNC_CPP ReservoirCCalc.c ReservoirCFunc.cpp)
```

Note: pour encapsuler un code écrit en C, il est également possible d'insérer le code initial directement dans la méthode `runStep()` de la fonction de simulation OpenFLUID.

3.4.3 Particularités de l'encapsulation d'un code Fortran

Pour encapsuler un code Fortran, des déclarations spécifiques sont utilisées :

- ajout d'une librairie dans la liste des headers :

```
#include <openfluid/base.hpp>
#include <openfluid/core.hpp>

#include <openfluid/tools/FortranCPP.hpp>
```

- déclaration d'une fonction externe, à insérer après la déclaration de la signature :

```
BEGIN_EXTERN_FORTRAN
        EXTERN_FSUBROUTINE(<nom de la subroutine>)
                (<listes des types et des arguments,ex:FINT *n,...>);
END_EXTERN_FORTRAN
```

- syntaxe spécifique pour appeler dans la méthode `runStep()` une subroutine Fortran (n'utilisant aucun module) :

```
CALL_FSUBROUTINE(<nom de la subroutine>)
        (<liste des arguments, ex:&n,...>);
```

- déclaration dans `CMake.in.config` du fichier source de la subroutine externe :

```
# list of Fortran files, if any
SET(FUNC_FORTRAN <nom du fichier source Fortran>)
```

- passage de vecteur en argument d'une subroutine.
  L'indexation des vecteurs est différente en Fortran et en C. Il faut donc passer en argument à la subroutine Fortran le premier élément du vecteur C déréférencé.

  ```
  CALL_FSUBROUTINE(<nom de la subroutine>)(<édbut du vecteur C, ex:&vect[0]>);
  ```

  De plus, la taille des vecteurs doit être déclarée de manière explicite dans l'entête de la subroutine Fortran.

  ```
  subroutine <nom de la subroutine> (<liste des arguments, ex:n,vect>)
  integer,intent(in)      :: n
  real(kind=8),intent(in) :: vect(n)
  ```

3.5 Résultats

Une fois développée, vous devez compiler et tester cette fonction. Il est possible d'utiliser l'application Google Earth™ pour visualiser les résultats scalaires (`water.surf-uz.storage` et `water.surf-uz.drainage`) de la fonction de simulation.

Note: une solution possible est proposée dans `src/training.encapsulation.reservoir-C/ReservoirCFunc.cpp`.