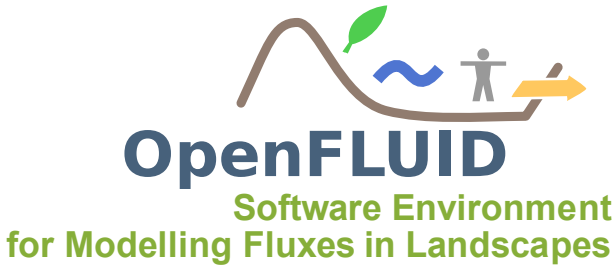

TP3 : Utilisation de variables en entrée

Objectifs:	Utiliser des variables produites par d'autres fonctions de simulation
Pré-requis:	TP1, TP2

Nous allons créer une fonction de simulation (`training.su.prod`) qui produit des variables de ruissellement et d'infiltration sur les SU à partir du signal de pluie, et en appliquant la méthode SCS.

1 Code source

1.1 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.su.prod`
- Fichier .cpp : `SUFunc.cpp`
- Classe de la fonction : `SUFunction`

1.2 Signature

Cette fonction génèrera des valeurs de ruissellement et d'infiltration à partir d'un signal de pluie. Nous allons donc déclarer la prise en compte de ce signal de pluie (variable `water.atm-surf.H.rain`) et la production des deux variables (`water.surf.H.runoff` pour le ruissellement et `water.surf.H.infiltration` pour l'infiltration).
La prise en compte d'une variable produite par une autre fonction sera déclarée au travers de l'instruction `DECLARE_REQUIRED_VAR`.

Une fois complétée, la signature devrait être similaire à :

```
BEGIN_SIGNATURE_HOOK
  DECLARE_SIGNATURE_ID("training.su.prod");
  DECLARE_SIGNATURE_NAME("");
  DECLARE_SIGNATURE_DESCRIPTION("");
```

```
  DECLARE_SIGNATURE_VERSION("12.03");
  DECLARE_SIGNATURE_SDKVERSION;
  DECLARE_SIGNATURE_STATUS(openfluid::base::EXPERIMENTAL);

  DECLARE_SIGNATURE_DOMAIN("");
  DECLARE_SIGNATURE_PROCESS("");
  DECLARE_SIGNATURE_METHOD("");
  DECLARE_SIGNATURE_AUTHORNAME("Doe␣J.");
  DECLARE_SIGNATURE_AUTHOREMAIL("doe@foobar.org");

  DECLARE_REQUIRED_VAR("water.atm-surf.H.rain","SU","rainfall␣height␣on␣the␣SU","m");

  DECLARE_PRODUCED_VAR("water.surf.H.runoff","SU","water␣runoff␣height␣on␣surface␣of␣SU","m");
  DECLARE_PRODUCED_VAR("water.surf.H.infiltration","SU",
                       "water␣infiltration␣height␣through␣the␣surface␣of␣SU","m");
END_SIGNATURE_HOOK
```

1.3 runStep()

Dans la méthode runStep(), nous allons calculer le partage ruissellement-infiltration à partir du signal de pluie en utilisant la méthode SCS.

Soit R le ruissellement, P la pluie, S le coefficient de rétention

$$0.0001 \leq S \leq 0.008$$

$$\text{Si } P \leq (0.2 \cdot S) \text{ alors } R = 0.0$$

$$\text{Si } P > (0.2 \cdot S) \text{ alors } R = \frac{(P - 0.2 \cdot S)^2}{(P + 0.8 \cdot S)}$$

Nous allons utiliser une boucle spatiale sur les SU, récupérer le signal de pluie au travers de l'instruction `OPENFLUID_GetVariable`, et produire le ruissellement et l'infiltration calculés à l'aide de deux instructions `OPENFLUID_AppendVariable`. Pour cet exercice nous fixerons la valeur de S à une valeur arbitraire choisie dans sa plage de validité.

Une fois complétée, la méthode runStep() devrait être similaire à :

```
bool runStep(const openfluid::base::SimulationStatus* SimStatus)
{
  openfluid::core::Unit* pSU;
  openfluid::core::DoubleValue RainValue;
  openfluid::core::DoubleValue RunoffValue;
  openfluid::core::DoubleValue InfiltrationValue;
  double S;
  DECLARE_UNITS_ORDERED_LOOP(5);

  BEGIN_UNITS_ORDERED_LOOP(5,"SU",pSU);
    S = 0.0035;  // Valeur du coeff. de retention fixe à 0,0035

    // recuperation de la valeur du signal de pluie
    OPENFLUID_GetVariable(pSU,"water.atm-surf.H.rain",SimStatus->getCurrentStep(),&RainValue);

    // calcul du ruissellement selon la methode SCS
    RunoffValue = 0.0;
    if (RainValue > (0.2*S))
    {
      RunoffValue = std::pow(RainValue-(0.2*S),2)/(RainValue+(0.8*S));
```

```
    }

    // calcul de l'infiltration par deduction
    InfiltrationValue = RainValue-RunoffValue;

    // production de l'infiltration et du ruissellement
    OPENFLUID_AppendVariable(pSU,"water.surf.H.infiltration",InfiltrationValue);
    OPENFLUID_AppendVariable(pSU,"water.surf.H.runoff",RunoffValue);
  END_LOOP;

  return true;
}
```

2 Simulation

Pour la simulation, nous allons compléter le jeu de données "Bassin versant TP" en ajoutant la fonction `training.su.prod` dans le modèle.

Attention: L'ordre des fonctions de simulation dans le modèle est important !

2.1 ... avec l'interface OpenFLUID-Builder

Afin de repartir du TP précédent, nous allons tout d'abord créer un projet OpenFLUID nommé TP3 (par exemple dans `/home/openfluid/formation/projects/TP3`) et y importer le jeu de données d'entrée du TP2, situé dans le sous-répertoire `IN` du projet TP2 à l'aide de l'onglet `import de données`, `Importer le jeu de données d'un projet existant`

Ensuite, rajouter la fonction (`training.su.prod`) dans le modèle du projet TP3, après la fonction déjà présente (training.signal.prod).

Attention: Il est nécessaire de remettre à jour la liste des fonctions disponibles si une de ces fonctions a été modifiée (recompilée par exemple). Pour cela, cliquer sur le bouton situé au dessus de la liste des fonctions disponibles.

Avant de procéder à la simulation, régler la configuration des sorties (*Outputs*) afin de prendre en compte les variables créées par la fonction de simulation nouvellement ajoutée.

2.2 ... en ligne de commande

Une fois complété, le fichier `model.fluidx` devrait être structuré comme suit :

```
<?xml version="1.0" encoding="UTF-8"?>
<openfluid>
  <model>
     <function fileID="training.signal.prod"/>
     <function fileID="training.su.prod" />
  </model>
</openfluid>
```

La commande à exécuter est donc :

```
openfluid-engine -i /home/openfluid/formation/datasets/TP1-TP7
  -o /home/openfluid/formation/outputs/TP3
```

(à taper sur une seule ligne)

Si tout s'est bien passé, les résultats de la simulation sont accessibles dans `/home/openfluid/formation/outputs/TP3`.