OpenFLUID
Software Environment
for Modelling Fluxes in Landscapes

TP7 : Utilisation de la connectivité entre unités spatiales

Objectifs:	Utiliser les relations de connectivité entre unités spatiales pour obtenir des informations -variables, données d'entrées, ...- sur les unités connexes
Pré-requis:	TP3,TP4,TP5

Nous allons créer une nouvelle fonction de simulation (`training.all.transfer`), qui calcule le débit en sortie de chaque unité spatiale des classes SU et RS. Elle utilisera les variables produites par la fonction de simulation `training.su.prod`, produira le débit d'eau pour chaque exutoire d'unité spatiale, et se basera sur les connexions entre ces unités spatiales pour assurer le transfert de cette eau à travers la zone étudiée.

1 Informations préliminaires

Le débit en sortie prend en compte le temps de transfert au travers de chaque unité spatiale de classe SU ou RS.

Soit d la distance de transfert, t le temps de transfert, v la vitesse de transfert

$$v = \frac{d}{t}$$

$$t = \frac{d}{v}$$

Soit Δt la durée d'un pas de temps, n le nombre de pas de temps pour le transfert sur une unité donnée

$$n.\Delta t = \frac{d}{v}$$

$$n = \frac{d}{v.\Delta t}$$

Avec v équivalent à la racine carrée de la pente p, on obtient

$$n = \frac{d}{\sqrt{p}.\Delta t}$$

Le calcul du débit en sortie Q d'une unité spatiale sera égale à la somme des apports des unités amonts, auquel on ajoute le débit local. Dans le cadre de cet exercice, seules les unités

de types SU génèrent un débit local.

Soit Q le débit local en sortie d'une unité donnée de type SU, S la surface de cette unité, H la hauteur de ruissellement, Δt la durée d'un pas de temps

$$Q = \frac{H.S}{\Delta t}$$

Note: Les hypothèses suivantes sont utilisées au cours de ce TP :
- le transfert sur les SU est prioritaire sur les transfert sur les RS
- le mécanisme de prise en compte des apports amont est simplifié
- les fossés (RS) ne débordent pas

2 Code source

2.1 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.all.transfer`
- Fichier .cpp : `TransferFunc.cpp`
- Classe de la fonction : `TransferFunction`

2.2 Signature

Cette fonction génèrera des valeurs de débit à partir des débits amonts, ainsi que du ruissellement produit sur les unités de type SU. Nous allons donc déclarer la prise en compte de ce ruissellement (variable `water.surf.H.runoff`).
Ce calcul du débit et son transfert nécessitent la prise en compte de propriétés physiques (surface, pente et distance d'écoulement des SU, longueur et pente des RS) qui seront fournis sous forme d'inputdata :
- `area` pour la surface des SU
- `slope` pour la pente des SU ou des RS
- `flowdist` pour la distance d'écoulement des SU
- `length` pour la longueur des RS

Une fois complétée, la signature devrait être similaire à :

```
BEGIN_SIGNATURE_HOOK
  DECLARE_SIGNATURE_ID("training.all.transfer");
  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_INPUTDATA("area","SU","area␣of␣the␣SU","m2");
  DECLARE_REQUIRED_INPUTDATA("slope","SU","slope␣of␣the␣SU","m/m");
  DECLARE_REQUIRED_INPUTDATA("flowdist","SU","flow␣distance␣of␣the␣SU","m");

  DECLARE_REQUIRED_INPUTDATA("slope","RS","slope␣of␣the␣RS","m/m");
  DECLARE_REQUIRED_INPUTDATA("length","RS","length␣of␣the␣RS","m");

  DECLARE_REQUIRED_VAR("water.surf.H.runoff","SU",
                       "water␣runoff␣height␣on␣surface␣of␣SU","m");

  DECLARE_PRODUCED_VAR("water.surf.Q.downstream","SU",
                       "output␣volume␣at␣the␣outlet␣of␣the␣SU","m3/s");
  DECLARE_PRODUCED_VAR("water.surf.Q.downstream","RS",
                       "output␣volume␣at␣the␣outlet␣of␣the␣RS","m3/s");

END_SIGNATURE_HOOK
```

2.3 initializeRun()

Le nombre de pas de temps de transfert étant lié à la pente et à la distance d'écoulement (invariants tout au long de la simulation), nous allons donc les calculer une fois pour toutes dans la partie initializeRun(), et les stocker sous la forme d'un attribut privé. Pour cela, nous utiliserons une structure de données de type `openfluid::core::IDIntMap` permettant de stocker une valeur de type entier pour chacune des unités spatiales concernées.

Une fois complétés, les attributs privés devraient être similaires à :

```
private:
  openfluid::core::IDIntMap m_SUTransferSteps;
  openfluid::core::IDIntMap m_RSTransferSteps;
```

Pour calculer le nombre de pas de temps de tranfert, nous allons utiliser deux boucles spatiales, une sur les SU et une autre sur les RS. Afin d'obtenir des nombres de pas de temps entiers, nous arrondirons les calculs au nombre entier supérieur (instruction `std::ceil()`).

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

```
bool initializeRun(const openfluid::base::SimulationInfo* SimInfo)
{
  openfluid::core::DoubleValue Slope, Flowdist;
  openfluid::core::Unit* pUnit;
  DECLARE_UNITS_ORDERED_LOOP(1);
  DECLARE_UNITS_ORDERED_LOOP(2);

  BEGIN_UNITS_ORDERED_LOOP(1,"SU",pUnit)
    OPENFLUID_GetInputData(pUnit,"slope",Slope);
    OPENFLUID_GetInputData(pUnit,"flowdist",Flowdist);

    m_SUTransferSteps[pUnit->getID()] = std::ceil(Flowdist
            / (std::sqrt(Slope)*double(SimInfo->getTimeStep())));
  END_LOOP

  BEGIN_UNITS_ORDERED_LOOP(2,"RS",pUnit)
    OPENFLUID_GetInputData(pUnit,"slope",Slope);
```

```
    OPENFLUID_GetInputData(pUnit,"length",Flowdist);

    m_RSTransferSteps[pUnit->getID()] = std::ceil(Flowdist
            / (std::sqrt(Slope)*double(SimInfo->getTimeStep())));
  END_LOOP

  return true;
}
```

2.4 runStep()

Dans la méthode runStep(), nous allons i) déterminer les apports amonts à transférer sur chaque SU, ii) calculer et ajouter le débit local à chaque SU, iii) déterminer les apports amonts à transférer sur chaque RS. Les apports amonts à transférer, ainsi que le calcul du débit local doivent tenir compte du temps de transfert propre à chaque unité et calculé dans la partie initializeRun().

Pour déterminer l'ensemble des unités contributives amont, nous allons utiliser la méthode getFromUnits() pour chaque unité, et parcourir cette liste d'unités contributives avec l'instruction BEGIN_UNITS_LIST_LOOP().

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

```
bool runStep(const openfluid::base::SimulationStatus* SimStatus)
{
  int StepToTransfer;
  openfluid::core::DoubleValue QValue, UpQValue;
  openfluid::core::DoubleValue Area;
  openfluid::core::DoubleValue Runoff;
  openfluid::core::Unit* pUnit;
  openfluid::core::Unit* pUpUnit;
  openfluid::core::UnitsPtrList_t* UpUnitsList;
  int CurrentStep;
  DECLARE_UNITS_ORDERED_LOOP(1);
  DECLARE_UNITS_ORDERED_LOOP(2);
  DECLARE_UNITS_LIST_LOOP(10);


  CurrentStep = SimStatus->getCurrentStep();

  BEGIN_UNITS_ORDERED_LOOP(1,"SU",pUnit)

    StepToTransfer =  CurrentStep - m_SUTransferSteps[pUnit->getID()];

    QValue = 0.0;

    if (StepToTransfer >= 0)
    {
      UpUnitsList = pUnit->getFromUnits("SU");

      BEGIN_UNITS_LIST_LOOP(10,UpUnitsList,pUpUnit)
        OPENFLUID_GetVariable(pUpUnit,"water.surf.Q.downstream",StepToTransfer,UpQValue);
        QValue = QValue + UpQValue.get();
      END_LOOP

      OPENFLUID_GetInputData(pUnit,"area",Area);
      OPENFLUID_GetVariable(pUnit,"water.surf.H.runoff",StepToTransfer,Runoff);
```

```
      QValue = QValue + (Runoff * Area / double(SimStatus->getTimeStep()));
    }

    OPENFLUID_AppendVariable(pUnit,"water.surf.Q.downstream",QValue);
  END_LOOP


  BEGIN_UNITS_ORDERED_LOOP(2,"RS",pUnit)
    StepToTransfer = SimStatus->getCurrentStep() - m_RSTransferSteps[pUnit->getID()];
    QValue = 0.0;

    if (StepToTransfer >= 0)
    {
      UpUnitsList = pUnit->getFromUnits("SU");

      BEGIN_UNITS_LIST_LOOP(10,UpUnitsList,pUpUnit)
        OPENFLUID_GetVariable(pUpUnit,"water.surf.Q.downstream",StepToTransfer,UpQValue);
        QValue = QValue + UpQValue.get();
      END_LOOP

      UpUnitsList = pUnit->getFromUnits("RS");

      BEGIN_UNITS_LIST_LOOP(10,UpUnitsList,pUpUnit)
        OPENFLUID_GetVariable(pUpUnit,"water.surf.Q.downstream",StepToTransfer,UpQValue);
        QValue = QValue + UpQValue.get();
      END_LOOP
    }

    OPENFLUID_AppendVariable(pUnit,"water.surf.Q.downstream",QValue);
  END_LOOP

  return true;
}
```

3 Simulation

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

3.1 ... avec l'interface OpenFLUID-Builder

Afin de repartir du TP précédent, nous allons tout d'abord créer un projet OpenFLUID nommé TP7 (par exemple dans `/home/openfluid/formation/projects/TP7`) et y importer le jeu de données d'entrée du TP2, situé dans le sous-répertoire `IN` du projet TP5.

Ensuite, rajouter la fonction (`training.all.transfer`) dans le modèle du projet TP7, après la fonction déjà présente (training.su.prod).

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.

3.2 ... 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">
       <param name="s" value="1.5" />
     </function>
     <function fileID="training.all.transfer">
  </model>
</openfluid>
```

La commande à exécuter est donc :

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

(à taper sur une seule ligne)

Si tout s'est bien passé, les résultats de la simulation sont accessibles dans `/home/openfluid/formation/outputs/TP7`.