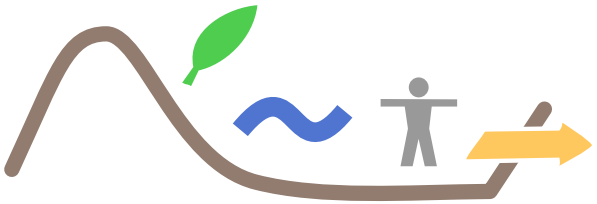


OpenFLUID

Software Environment
for Modelling Fluxes in Landscapes

Développement de fonctions de simulation

Jean-Christophe Fabre

LISAH
Laboratoire d'étude des Interactions
Sol-Agrosystème-Hydrosystème

Plan

Plan

Structure générale

Une fonction de simulation est un code en C++ comprenant deux parties distinctes

- La signature
- Le corps intégrant le code de calcul ("le modèle")

L'ensemble est intégré dans un fichier `.cpp` (et éventuellement d'autres fichiers `.cpp` et `.hpp` qui peuvent être inclus).

Ce(s) fichier(s) est(sont) compilé(s) sous la forme d'un plug-in indépendant pouvant être reconnu et utilisé par le framework OpenFLUID

- nom = identifiant de la fonction
- extension = `.sompi` (Unix/Linux), `.dylibmpi` (MacOSX), `.dllmpi` (Windows)

Signature

La signature contient une description de la fonction de simulation :

- identifiant de la fonction
- variables utilisées
- variables produites
- paramètres de fonction utilisés
- données d'entrée nécessaires
- évènements pris en compte
- méta-information : version, auteur(s), nom, description, ...

La signature doit refléter le comportement de la fonction de simulation

La signature peut être consultée en ligne de commande (`openfluid-engine -r`), ou via l'interface graphique.

Corps de la fonction

Le corps de la fonction de simulation est constitué d'une classe C++ unique.

Cette classe implémente des méthodes qui doivent être obligatoirement présentes (même si elles sont vides).

Ces méthodes correspondant à différentes phases de simulation

- Préparation
- Initialisation
- Exécution du calcul (à chaque pas de temps)
- Finalisation

Vue d'ensemble (pseudo-fonction)

Exemple

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

DECLARE_PLUGIN_HOOKS;

BEGIN_SIGNATURE_HOOK
  // ici la signature a completer
END_SIGNATURE_HOOK

class MyFunction : public openfluid::base::PluggableFunction
{
  // ici la classe a completer
  // avec les methodes obligatoires
  // incluant le code de calcul
}

DEFINE_FUNCTION_HOOK(MyFunction);
```

Interactions avec le framework OpenFLUID

Le framework charge dynamiquement les fonctions de simulation constituant le modèle couplé lors des simulations.

La signature de chaque fonction est utilisée par le framework pour

- connaître les entrées/sorties des fonctions
- vérifier la cohérence entre fonctions de simulation

Les méthodes définies par chaque fonction de simulation sont exécutées (appelées) à différentes phases de simulation.

Plan

API OpenFLUID

Application Programming Interface = ensemble des fonctionnalités disponibles pour développer des fonctions de simulation :

- manipulation de variables, des données d'entrée, des paramètres de fonctions, des évènements discrets, ...
- manipulation de l'espace
- information sur l'état de la simulation
- remontée d'avertissements et d'erreurs
- ...

3 espaces de nommage OpenFLUID

- `openfluid::core` (structures de données)
- `openfluid::base` (fonctionnement)
- `openfluid::tools` (outils de traitement de données)

Documentation

Disponible via l'espace communautaire OpenFLUID sur le web

http://www.umr-lisah.fr/openfluid/community/

- Documentation de l'API
- Guide de développement
- Guide de bonnes pratiques
- Manuel de l'utilisateur
- Code snippets ("Bouts de code")
- Information de migration (changement de version du moteur de calcul)
- Support

Environnement de développement préconisé

Compilateurs : suite GCC

- g++, gfortran

Système de construction/test/packaging

- CMake, CTest, CPack

Environnement de développement intégré (IDE)

- Eclipse + CDT + Plug-in Eclipse pour OpenFLUID
- Gestionnaire de version subversion

Support : canal IRC, mailing-list, espace SourceForge

- http://www.umr-lisah.fr/openfluid/community/index.php/Community_support

Plan

Identification de la fonction et autres informations

- L'identifiant de la fonction de simulation est obligatoire
- D'autres informations facultatives peuvent être ajoutées pour mieux décrire la fonction

Exemple

```
BEGIN_SIGNATURE_HOOK
  DECLARE_SIGNATURE_ID("formation.exemple");
  DECLARE_SIGNATURE_NAME("Fonction␣exemple");
  DECLARE_SIGNATURE_DESCRIPTION("Cette␣fonction␣est␣...");

  DECLARE_SIGNATURE_VERSION("10.04");
  DECLARE_SIGNATURE_STATUS(openfluid::base::EXPERIMENTAL);

  DECLARE_SIGNATURE_DOMAIN("formation");
  DECLARE_SIGNATURE_PROCESS("exemple");
  DECLARE_SIGNATURE_METHOD("simple");
  DECLARE_SIGNATURE_AUTHORNAME("Chuck␣Norris");
  DECLARE_SIGNATURE_AUTHOREMAIL("norris@overmail.chuck");
END_SIGNATURE_HOOK
```

Variables

Les variables déclarées doivent comporter

- un nom unique
- une classe d'unité à laquelle elles sont attachées
- une description et une unité SI

Exemple

```
BEGIN_SIGNATURE_HOOK
  DECLARE_SIGNATURE_ID("formation.exemple");

  // variable produite
  DECLARE_PRODUCED_VAR("exemple.var0","UnitsA","variable␣0","m");
  // variable requise
  DECLARE_REQUIRED_VAR("exemple.var1","UnitsA","variable␣1","kg");
  // variable utilisee si presente
  DECLARE_USED_VAR("exemple.var2","UnitsA","variable␣2","-");
  // variable requise a un pas de temps precedent
  DECLARE_REQUIRED_PREVVAR("exemple.var3","UnitsA","variable␣3","?");
  // variable utilisee a un pas de temps precedent
  DECLARE_USED_PREVVAR("exemple.var4","UnitsA","variable␣4","$\frac{1}{m}$");
END_SIGNATURE_HOOK
```

Variables et types de données

- Variable non typée : les valeurs peuvent être de n'importe quel type
- Variable typée : toutes les valeurs doivent être du type de la variable

La vérification de cohérence production/utilisation tient compte du typage/non-typage des variables

Exemple

```
BEGIN_SIGNATURE_HOOK
  DECLARE_SIGNATURE_ID("formation.exemple");
  // variable produite
  DECLARE_PRODUCED_VAR("exemple.var0","UnitsA","variable␣0","m");
  // variable produite de type matrice
  DECLARE_PRODUCED_VAR("exemple.var4[matrix]","UnitsA","variable␣4","-");
  // variable requise
  DECLARE_REQUIRED_VAR("exemple.var1","UnitsA","variable␣1","kg");
  // variable de type double requise a un pas de temps precedent
  DECLARE_REQUIRED_PREVVAR("exemple.var3[double]","UnitsA","variable␣3","?");
END_SIGNATURE_HOOK
```

Données d'entrée

Les données d'entrée déclarées doivent comporter

- un nom de donnée d'entrée existante
- une classe d'unité à laquelle elle sont attachées
- une description et une unité SI

Exemple

```
BEGIN_SIGNATURE_HOOK
  DECLARE_SIGNATURE_ID("formation.exemple");

  DECLARE_REQUIRED_INPUTDATA("input1","UnitsA","input␣1","m");
  DECLARE_USED_INPUTDATA("input2","UnitsA","input␣data␣2","-");
END_SIGNATURE_HOOK
```

Paramètres

Les paramètres de fonction déclarés doivent comporter

- un nom de paramètre
- une description et une unité SI

Exemple

```
BEGIN_SIGNATURE_HOOK
  DECLARE_SIGNATURE_ID("formation.exemple");

  DECLARE_FUNCTION_PARAM("param1","","-");
  DECLARE_FUNCTION_PARAM("param2","","m/s");
END_SIGNATURE_HOOK
```

Exemple complet

```
BEGIN_SIGNATURE_HOOK
  DECLARE_SIGNATURE_ID("formation.exemple");
  DECLARE_SIGNATURE_NAME("Fonction␣exemple");

  DECLARE_SIGNATURE_VERSION("10.04");
  DECLARE_SIGNATURE_STATUS(openfluid::base::BETA);
  DECLARE_SIGNATURE_DOMAIN("formation");
  DECLARE_SIGNATURE_AUTHORNAME("Chuck␣Norris");
  DECLARE_SIGNATURE_AUTHOREMAIL("norris@overmail.chuck");

  DECLARE_PRODUCED_VAR("var.exemple.var0","UnitsA","","m");
  DECLARE_REQUIRED_VAR("exemple.var1","UnitsA","","kg");
  DECLARE_USED_VAR("exemple.var2","UnitsA","var␣2","-");

  DECLARE_REQUIRED_INPUTDATA("input1","UnitsA","","m");
  DECLARE_USED_INPUTDATA("input2","UnitsA","input␣data␣2","-");

  DECLARE_FUNCTION_PARAM("param2","","m/s");
END_SIGNATURE_HOOK
```

Plan

Une fonction est une classe C++

Une classe définissant une fonction doit hériter de la classe `openfluid::base::PluggableFunction`

Elle doit redéfinir les méthodes publiques obligatoires suivantes:

- `initParams()`
- `prepareData()`
- `checkConsistency()`
- `initilizeRun()`
- `runStep()`
- `finalizeRun()`

Classe C++ de fonction "vide"

```
class MyFunction : public openfluid::base::PluggableFunction
{
  public:

    MyFunction() : PluggableFunction() {  }
    ~MyFunction() {  }

    bool initParams(openfluid::core::FuncParamsMap_t Params)
    { return true; }

    bool prepareData()
    { return true; }

    bool checkConsistency()
    { return true; }

    bool initializeRun(const openfluid::base::SimulationInfo* SimInfo)
    { return true; }

    bool runStep(const openfluid::base::SimulationStatus* SimStatus)
    { return true; }

    bool finalizeRun(const openfluid::base::SimulationInfo* SimInfo)
    { return true; }
};
```

Méthodes de la fonction de simulation (1/2)

`bool initParams(openfluid::core::FuncParamsMap_t Params)`

- récupération des paramètres de fonction depuis la section `<model>`, via le paramètre `Params`

`bool prepareData()`

- préparation des données avant vérification de cohérence

`bool checkConsistency()`

- vérifications de la cohérence interne de la fonction

Méthodes de la fonction de simulation (2/2)

```
bool initializeRun(const
openfluid::base::SimulationInfo* SimInfo)
```

- phase d'initialisation de la simulation : calcul d'invariants, mise en place de "lookup tables", chargement de fichiers propres, ...

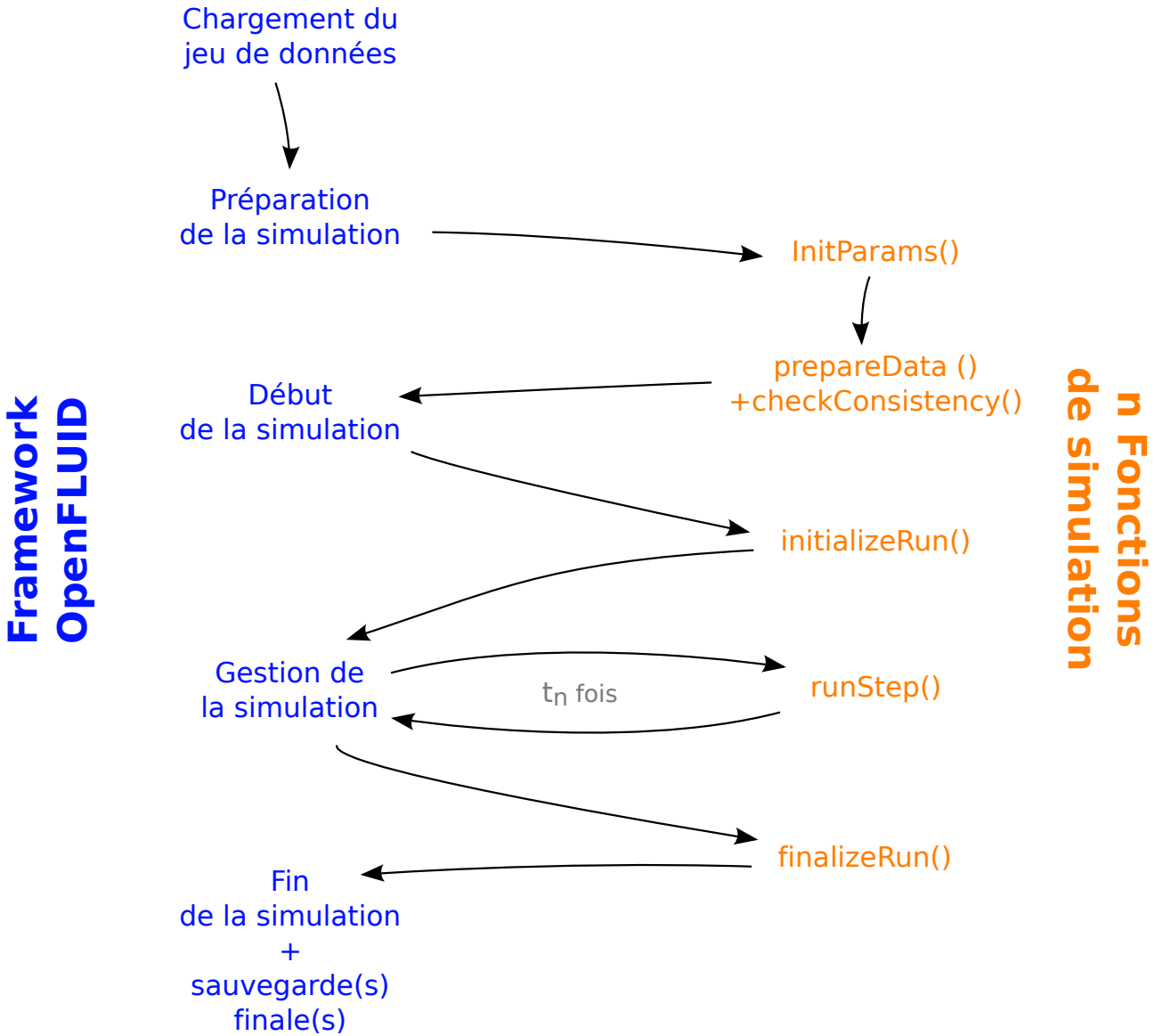
```
bool runStep(const openfluid::base::SimulationStatus*
SimStatus)
```

- exécution des calculs sur un pas de temps donné

```
bool finalizeRun(const
openfluid::base::SimulationInfo* SimInfo)
```

- phase de finalisation de la simulation : calcul de bilans, sauvegarde de fichiers propres, ...

Séquence d'interaction framework/fonctions de simu.



Boucles spatiales

Une boucle spatiale permet de parcourir l'ensemble des unités d'une même classe d'unité, selon leur ordre de traitement

- déclaration : `DECLARE_UNITS_ORDERED_LOOP()`
- début de boucle : `BEGIN_UNITS_ORDERED_LOOP()`
- fin de boucle : `END_LOOP`

Exemple

```
{
  openfluid::core::Unit* pUA;
  // declaration d'une boucle d'identifiant 7
  DECLARE_UNITS_ORDERED_LOOP(7);

  // boucle sur les unites de classe "UnitsA"
  // a chaque tour de boucle, pUA pointe sur l'unite courante
  BEGIN_UNITS_ORDERED_LOOP(7,"UnitsA",pUA);

        // effectuer ici les calculs sur l'unite courante

  END_LOOP;
}
```

Manipulation de variables

Les variables ne doivent être manipulées que depuis la méthode `runStep()`

- production : `OPENFLUID_AppendVariable()`
- récupération : `OPENFLUID_GetVariable()`
- mise à jour : `OPENFLUID_SetVariable()`

Exemple

```
bool runStep(const openfluid::base::SimulationStatus* SimStatus)
{
  openfluid::core::Unit* pUA;
  openfluid::core::DoubleValue Value;
  DECLARE_UNITS_ORDERED_LOOP(3);

  BEGIN_UNITS_ORDERED_LOOP(3,"UnitsA",pUA);
    OPENFLUID_GetVariable(pUA,"example.var",SimStatus->getCurrentStep(),&Value);
    Value = Value + 1;
    OPENFLUID_AppendVariable(pUA,"example.varplus",Value);
  END_LOOP;

  return true;
}
```

Manipulation de données d'entrée

Les données d'entrée ne doivent être manipulées que depuis les méthodes `initializeRun()`, `runStep()`, `finalizeRun()`

- récupération : `OPENFLUID_GetInputData()`

Exemple

```
bool runStep(const openfluid::base::SimulationStatus* SimStatus)
{
  openfluid::core::Unit* pUA;
  openfluid::core::DoubleValue Value,Data;
  DECLARE_UNITS_ORDERED_LOOP(3);

  BEGIN_UNITS_ORDERED_LOOP(3,"UnitsA",pUA);
    OPENFLUID_GetVariable(pUA,"example.var",SimStatus->getCurrentStep(),&Value);
    OPENFLUID_GetInputData(pUA,"idata",&Data);
    Value = Value + Data;
    OPENFLUID_AppendVariable(pUA,"example.varplusdata",Value);
  END_LOOP;

  return true;
}
```

Manipulation des paramètres de fonction

Les paramètres de fonction ne peuvent être manipulés que depuis la méthode `initParams()`

- récupération : `OPENFLUID_GetFunctionParameter()`
- afin de pouvoir être utilisées depuis d'autres méthodes, il est préférable de stocker les valeurs des paramètres dans des attributs privés de la classe C++

Exemple

```
bool initParams(openfluid::core::FuncParamsMap_t Params)
{
  OPENFLUID_GetFunctionParameter(Params,"paramA",&m_PA);
  OPENFLUID_GetFunctionParameter(Params,"paramB",&m_PB);

  return true;
}
```

Manipulation des connexions entre unités

Les connexions sont portées par les unités elles-mêmes, accessibles via les méthodes

- liste des unités connectées "vers" : `getToUnits()`
- liste des unités connectées "depuis" : `getFromUnits()`

Exemple

```
bool initializeRun(const openfluid::base::SimulationInfo* SimInfo)
{
  openfluid::core::Unit* pUA, pFromUA;
  openfluid::core::DoubleValue Area, AreaSum;
  DECLARE_UNITS_ORDERED_LOOP(2);
  DECLARE_UNITS_LIST_LOOP(9);

  BEGIN_UNITS_ORDERED_LOOP(2,"UnitsA",pUA);
    UpperAreaSum = 0;
    BEGIN_UNITS_LIST_LOOP(1,pUA->getFromUnits("UnitsA"),pFromUA)
      OPENFLUID_GetInputData(pFromUA,"area",&Area);
      UpperAreaSum = UpperAreaSum + Area;
    END_LOOP;
 END_LOOP;

  return true;
}
```

Messages d'erreurs et d'avertissements

Il est possible de faire remonter des messages informatifs

- avertissements non-bloquants : `OPENFLUID_RaiseWarning()`
- erreurs stoppant la simulation : `OPENFLUID_RaiseError()`

Exemple

```
bool runStep(const openfluid::base::SimulationStatus* SimStatus)
{
  openfluid::core::Unit* pUA;
  openfluid::core::IntegerValue Value;
  DECLARE_UNITS_ORDERED_LOOP(3);

  BEGIN_UNITS_ORDERED_LOOP(3,"UnitsA",pUA);
    OPENFLUID_GetVariable(pUA,"example.var",SimStatus->getCurrentStep(),&Value);

    if (Value < 0)
      OPENFLUID_RaiseError("function.example","runStep()",
                           SimStatus->getCurrentStep(),
                           "Error message from example function");
  END_LOOP;

  return true;
}
```

Pour en savoir plus. . .

Fonctionnalités non présentées

- Variables vecteurs
- Tests d'existence de variables, données d'entrée, . . .
- Utilisation d'évènements discrets
- Production d'évènements discrets
- Gestion de fichiers propres aux fonctions de simulation
- Outils intégrés pour l'interpolation des données
- Intégration de code en d'autres langages (Fortran, . . .)
- Utilisation de générateurs de données
- Données d'entrée de type chaînes de caractères
- Paramètres de fonctions communs
- Connectivité de type parents/enfants entre unités
- . . .

Références & Ressources


Jean-Christophe Fabre.
Guide du développeur LISAH.
fabrejc@supagro.inra.fr.


Eclipse IDE.
`http://www.eclipse.org/`.


Site web OpenFLUID.
`http://www.umr-lisah.fr/openfluid/`.


Site web OpenFLUID Community.
`http://www.umr-lisah.fr/openfluid/community/`.