Wrapping du code de DassFlow-2D

01 Septembre 2020

1 Introduction

Ce fichier a pour but de représenter la méthodologie du wrapping du code de DassFlow-2D en Python. La générateur utilisée ici est la `f90wrap`, une version plus forte de `f2py`. Alors pourquoi on a choisi cet outil?

Il y a des autres générateurs d'interface de Fortran-Python comme `f2py-2e`, `G3 f2py`, etc. ceux qui basent aussi sur la `f2py` (inclue dans le `numpy` de Python). Pourtant, ces outils permettent de résoudre seulement pour des codes simples, il y a une seule la `f90wrap` développée à partir de `f2py` qui fonctionne avec plus des cas complexes (le cas du type de dérivé, etc.). En plus, on peut trouver une solution de wrapper Python-Fortran via un autre langage comme C. Autrement dit, on peut faire le wrapping Fortran-C et puis C-Python (via `SWIG` par exemple), comme il est plus facile de traduire du code de Fortran à C et il y a plus des outils qui sont plus fortes pour faire le wrapping de C-Python. Par contre, ce moyen n'est pas optimal car on a besoin d'un troisième langage, ce qui rend beaucoup de complication dans la construction du code.

2 Relance du `Makefile`

2.1 Modification du chemin de PYTHONPATH

Avant de commencer, on a besoin de modifier un bon chemin pour la variable `PYTHONPATH` dans le fichier `/scripts/env.sh` pour exécuter le fichier Python qui appelle des fonctions de Fortran quand les modules de Python sont crées.

```
PYTHONPATH = Le chemin de votre machine
```

2.2 Run des commandes

Tout d'abord, on réinstalle les libs et recompile tous les codes du DassFlow-2D par la liste des commandes ci-dessous:

```
$ make cleanall
$ make lib
$ make bin_files
$ make tap_files
```

Une fois les codes sont recompilés, on passe à l'étape du wrapping et puis exécuter le fichier Python à tester pour vérifier que le paquet `dassflow2d` est bien importé depuis un fichier Python en tapant les commandes suivants:

```
$ make clean
$ make wrap
$ source ./scripts/env.sh
$ make testwrap
```

3 Modification du code de fichiers Fortran

La liste des fichiers modifiés ou rajoutés:

m_common.f90
m_linear_algebra.f90
m_mesh.f90
m_sw_mono.f90
m_user_data.f90
sw_pre.f90
m_linear_solver.f90
m_numeric.f90
m_obs.f90
boundary.f90
advance_time.f90
numeric_sw.f90
sw_flux.f90
euler_time_step_first.f90
euler_time_step_first_b1.f90
sw_post.f90
run_model_sw_mono.f90
call_run_model.f90 (rajouté)
geometry.f90
initialization.f90

Notons que tous les compilations conditionnelles (CPP_PASS) rajoutées dans les fichiers Fortran n'affectent qu'à la partie de wrapping.

3.1 Modification général

En général, on met la ligne USE m_mpi, tous les routines concernant le mpi comme Time_Init_Part, com_var_r, Time_End_Part et Stopping_Program_Sub et parfois, la ligne USE m_time_screen et l'appel de ses routines en commentaire. Voici un exemple pour les différents entre le fichier euler_time_step_first_b1.f90 wrappé et non wrappé (voir la figure 1):

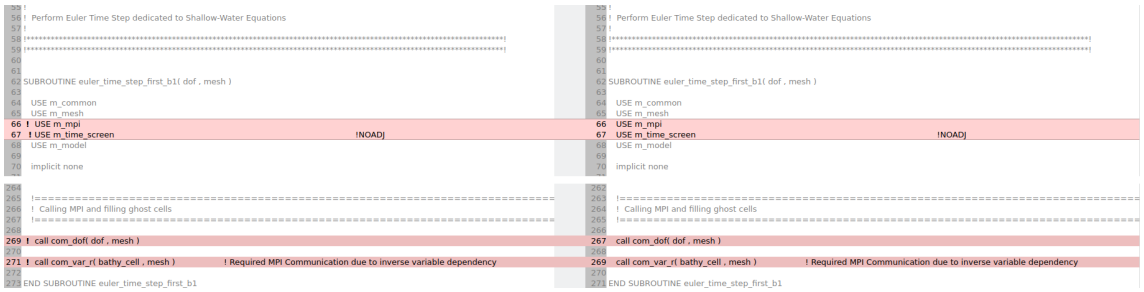

Figure 1: Différence entre les fichiers euler_time_step_first_b1.f90 wrappé et non wrappé

3.2 Modification spécifique

3.2.1 Fichier src/common/m_mesh.f90

On rajoute dans ce fichier les routines NodeType_initialise et test.

- Le content du routine NodeType_initialise est pris dans le routine Create_Cartesian_Mesh du fichier src/base/input.f90 mais il est modifié un peu pour adapter au fait de wrapping.
- Le routine test a pour but de tester si la variable du type msh est bien importée car il n'y aucun règle pour d'appeler directement les variables comme Node, Nodeb, etc. de ce type depuis un fichier Python.

3.2.2 Rajout du fichier src/sw_mono/call_run_model.f90

Le fichier call_run_model.f90 crée a pour but de faire une simulation directe à partir des Modules et des routines de la source. La simulation base sur les pas principaux ci-dessous:

- Initialiser la variable mdl du type Model contenant la donnée entrée msh du type Mesh, la position initiale dof du type unk et le coût de function cost.
- Initialiser et set pour les variables msh et dof de mdl
- Lancer le modèle appliqué pour mdl et le résultat est enregistrée dans res cost_func.

3.2.3 Lecture du fichier input.txt

L'idée c'est l'on lit directement les données à partir du fichier Python et puis les enregistrer dans une variable dont le type est lisible pour Fortran. Enfin, cette variable est utilisé dans un routine de Fortran pour définir les variables entrées comme: mesh_type, mesh_name, ts, dtw, etc. L'intérêt de cette méthode c'est qu'on peut re-compiler le fichier Python avec des différentes valeurs de variables entrées en modifiant le fichier input.txt dans src/wrappers/ et sans relancer la commande pour faire le wrapping make wrap.

Lecture de données avec Python. Dans les parties # Create a specific data file to read data from Python et # Convert variable type from the specific data file, on a lu des données dans le ficher src/wrappers/input.txt (il est recopié depuis bin/input.txt au cours de lancer la commande make wrap) et puis a converti le type pour chacun de paramètre, tous les paramètres lus sont enregistrés dans la liste input_data.

Création d'un type lisible pour Fortran. On veut maintenant définir la valeur des variables entrées (ts, dtw, bc_N, bc_S, etc.) dans Fortran en prenant la variable input_data créée par Python. Cependant, il n'existe pas une liste avec des éléments de types différents en Fortran, donc il est nécessaire de créer un nouveau type dans Fortran et initialiser la variable de ce nouveau type en Python. Dans le fichier m_sw_mono.f90, on crée ce nouveau type:

```
TYPE inputdata
character(len=lchar) :: mesh_type_
character(len=lchar) :: mesh_name_
real(rp)      :: lx_
real(rp)      :: ly_
integer(ip)       :: nx_
integer(ip)       :: ny_
character(len=lchar) :: bc_n_
character(len=lchar) :: bc_s_
character(len=lchar) :: bc_w_    character(len=lchar) :: bc_e_
real(rp)      :: ts_
real(rp)      :: dtw_
real(rp)      :: dtp_
real(rp)      :: dta_
character(len=lchar) :: temp_scheme_
```

```
character(len=lchar) :: spatial_scheme_
integer(ip)      :: adapt_dt_
real(rp)      :: cfl_
real(rp)      :: heps_
integer(ip)      :: friction_
real(rp)      :: g_
integer(ip)      :: w_tecplot_
integer(ip)      :: w_obs_
integer(ip)      :: use_obs_
real(rp)      :: eps_min_
integer(ip)      :: c_manning_
real(rp)      :: eps_manning_
real(rp)      :: regul_manning_
integer(ip)      :: c_bathy_
real(rp)      :: eps_bathy_
real(rp)      :: regul_bathy_
integer(ip)      :: c_hydrograph_
real(rp)       :: eps_hydrograph_
real(rp)      :: regul_hydrograph_
END TYPE inputdata
```

et écrit un routine pour qu'on puisse les appeler et les initialiser dans Python:

```
subroutine inputdata_initialise(inData)
  implicit none
  type(inputdata), intent(out) :: inData
end subroutine
```

Dans le fichier Python, on initialise la variable inData de type inputdata en utilisant la liste input_data:

```
# Set variable as a type of inputdata
inData = m_model.inputdata()
inData.mesh_type_ = input_data[0]
inData.mesh_name_ = input_data[1]
inData.lx_ = input_data[2]
inData.ly_ = input_data[3]
...
```

Définir la valeur pour les entrées dans Fortran. Finalement, on écrit un routine dans m_sw_mono.f90 pour définir les entrées avec la variable inData vient d'être initialisée par Python:

```
subroutine set_inputdata(inData)
  implicit none
  type(inputdata), intent(in) :: inData
  mesh_type= inData%mesh_type_
  mesh_name= inData%mesh_name_
  lx= inData%lx_
  ly= inData%ly_
...
end subroutine
```

3.2.4 Lecture des conditions limites

La variable bc de type bcs (dans m_sw_mono.f90) définit la condition limite de la simulation:

```
TYPE bcs
      integer(ip)  ::  nb , nb_in , nb_out
      character(len=lchar), dimension(:,:), allocatable  ::  typ
      integer(ip), dimension(:), allocatable  ::   grpf
      real(rp), dimension(:), allocatable  ::  inflow       real(rp), dimension(:), al
      type(hydrograph), dimension(:), allocatable  ::  hyd
      type(ratcurve), dimension(:), allocatable  ::  rat
      real(rp), dimension(:), allocatable  ::  sum_mass_flux
   END TYPE bcs
```

Lecture de données (non-wrappé): Les données nécessaires pour déterminer la condition limite sont enregistrées dans des fichiers dans /bin (dans ce cas hydrograph.txt et rating_curve.txt).

- hydrograph.txt: La variable bc est initialisée par le fichier initialization.f90 (la partie 'Boundary Condition Initialization'). Pour cela, on a besoin de lire les variables ci-dessous dans le fichier hygrograph.txt:
  - Le number de hydrographe: bc%nb_in = 1
  - Le number de point (ou la taille de t et q): j = 241
  - Les temps t: bc%hyd(i) %t(k)
  - Les discharges q: bc%hyd(i)%q(k)

avec i = 1,bc%nb_in et k = 1,j.

Remarque: La variable bc%hyd est de type hydrograph qui est défini dans m_sw_mono.f90:

```
TYPE hydrograph
   integer(ip)  ::  group
   real(rp), dimension(:), allocatable  ::  t , q
END TYPE
```

- rating_curve.txt:
  - Le number de ratcurve: bc%nb_out = 1
  - Le number de point: j = 101
  - bc%rat(i)%z_rat_ref = 0
  - h: bc%rat(i)%h(k)
  - q: bc%rat(i)%q(k)

avec i = 1,bc%nb_out et k = 1,j.

Remarque: La variable bc%rat est de type ratcurve qui est défini dans m_sw_mono.f90:

```
TYPE ratcurve
   integer(ip)  ::  group
   real(rp), dimension(:), allocatable  ::  h , q
   real(rp)  ::  z_rat_ref , zout , c1 , c2 , pow(2)
END TYPE
```

Autre façon afin de lire les données (wrappé):

- Création d'un nouveau type boundarycondition: Dans le fichier sw_mono.f90 on crée un nouveau type 'boundarycondition' contenant des variables qui sont importées par les fichiers .txt comme nb_in, nb_out, etc.. On définit également une nouvelle variable bc_ du type boundarycondition:

```
TYPE boundarycondition
integer(ip)  ::  nb_in_ , j_hyd
integer(ip)  ::  nb_out_ , j_rat
real(rp)     ::  rat_ref
real(rp), dimension(:), allocatable  ::  t_hyd , q_hyd
real(rp), dimension(:), allocatable  ::  h_rat , q_rat
   END TYPE boundarycondition
   type(boundarycondition) :: bc_
```

Cette variable bc_ va être utilisé pour remplacer aux variables qui sont importées par les fichiers .txt.

- set_boundarycondition: Maintenant, le routine set_boundarycondition() créé dans le fichier sw_mono.f90 permet de définir des valeurs pour tous les paramètres de bc_:

```
subroutine set_boundarycondition( t1 , q1 , h2 , q2 )
implicit none
real(rp), dimension(:), intent(in) :: t1 , q1
real(rp), dimension(:), intent(in) :: h2 , q2
bc_%nb_in_ = 1
bc_%j_hyd = 241
bc_%nb_out_ = 1
bc_%j_rat = 101
bc_%rat_ref = 0._rp
bc_%t_hyd = t1
bc_%q_hyd = q1
bc_%h_rat = h2
bc_%q_rat = q2
end subroutine
```

Notons que l'on définit uniquement des paramètres entiers et réel de bc_c, les 4 vecteurs t1 , q1 , h2 , q2 sont importés par le fichier Python.

- Modification du fichier initialization.f90: Dans ce fichier, on remplace les variables importées par Fortran par les paramètres de la variable bc_, par exemple:

```
read(10,*) bc%nb_out
```

remplacé par:

```
bc%nb_out = bc_%nb_out_
```

Ou un autre exemple:

```
read(10,*) j , bc%rat( i )%z_rat_ref
```

remplacé par:

```
j = bc_%j_rat
bc%rat( i )%z_rat_ref = bc_%rat_ref
```

- Lecture direct des paramètres par Python:

```
##### Import boudary conditions data ######     fname1 = os.path.join("hydrograph.txt"
hyd = np.loadtxt(fname1) # Import average monthly precip to numpy array
t1 = hyd[:,0]
q1 = hyd[:,1]
fname2 = os.path.join("rating_curve.txt")
rat = np.loadtxt(fname2) # Import average monthly precip to numpy array
h2 = rat[:,0]
q2 = rat[:,1]
```

3.3 Visualization de résultats de simulation

Afin de visualisiez les résultats (hauteur d'eau selon l'axe x et l'axe y), on a besoin de créer un nouveau type nommé points dans le fichier src/sw_mono/call_run_model. Une variable pts de ce type contient la coordonnées des points à tracer dans l'espace:

```
 TYPE points
real(rp), dimension(:), allocatable :: x_space
real(rp), dimension(:), allocatable :: y_space
 END TYPE
```

On rajoute une variable sortie pts dans le routine run_direct. Les informations concernant les points à tracer sont dans la variable node du type mesh (mdl%mesh%node). Pour chaque nœud, on recopie les infos de coordonnée dans pts:

```
do k = 1,mdl%mesh%nn
   pts%x_space(k) = mdl%mesh%node(k)%coord%x
   pts%y_space(k) = mdl%mesh%node(k)%coord%y
end do
```

Dans le fichier Python, après d'avoir enregistré les résultats de simulation dans les variables h, u, v, x et y (les x, y sont les coordonnées des points à tracer et h est son hauteur) à temps ts. On va ensuite créer un fichier data.plt pour enregistrer ces données:

```
 #####Create a data file for Plotting data#####
x=result[2].x_space
y=result[2].y_space
for j in range(size_):
    name = "t="+str(int(t[j]))
    filename = "%s.plt" % name
    filepath = os.path.join('./output_data', filename)
    if not os.path.exists('./output_data'):
        os.makedirs('./output_data')
    f=open(filepath,"w+")
    f.write('## VARIABLES  =  x  y  h  u  v ' + '\n')
    for i in range(len(x)):
    f.write( str(x[i]) + ' ' + str(y[i]) + ' ' + str(h[j][i]) + '  ' + str(u[j][i]) +
    f.close()
```

Une fois les données sont enregistrés dans les fichiers src/wrappers/t=xxx.plt, on utilise le gnuplot pour visualiser les résultats obtenus:

```
gnuplot> reset
gnuplot> set hidden3d
gnuplot> set dgrid3d 50,50 qnorm 2
gnuplot> set pm3d
gnuplot> splot "data.plt" using 1:2:3 title 'h' with lines
```