POLYTECHNIQUE
MONTRÉAL

L'environnement Python3 pour Dragon5 et Donjon5

A. Hébert

2020/07/17

POLYTECHNIQUE MONTRÉAL

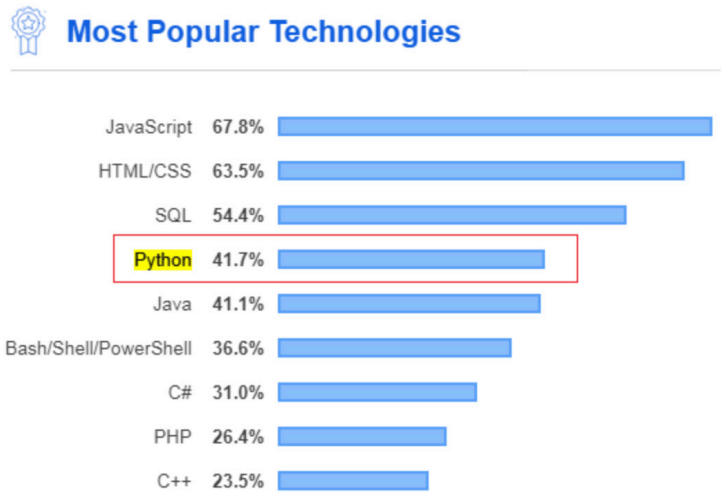
Table des matières

Le langage Python3
Les modules d'extension PyGan
Le module d'extension lcm
Le module d'extension lifo
Le module d'extension cle2000
Multiphysics calculations
PyBind

Le langage Python3

Python3 est un langage informatique Open Source ayant les caractéristiques suivantes:

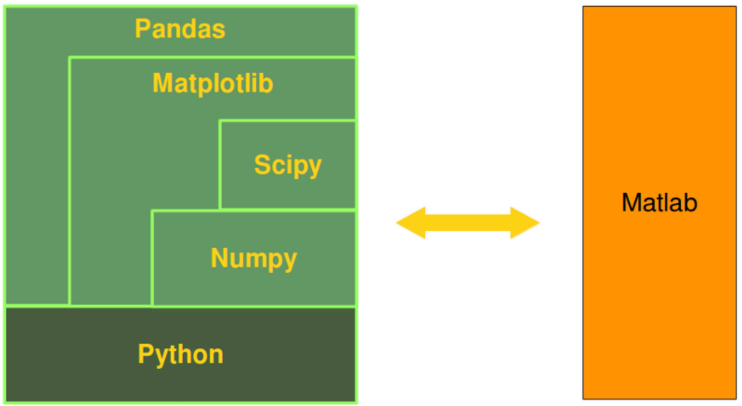
- langage de script (comme Matlab ou Perl) interprété (comme Gibiane, CLE-2000, Matlab ou Perl) au moyen d'une machine virtuelle
- orienté objet (modèle de classes UML, comme Java, Swift ou C++)
- à typage dynamique (comme Matlab, à l'opposé de CLE-2000)
- permet de superviser l'exécution de logiciels ou progiciels écrits dans des langages compilés (Fortran, C, C++)
  - Souvent associé à des codes écrits en C++ (OpenMOC, OpenMC, Cocagne, etc).
- largement répandu dans différentes disciplines, dont celles du génie
  - Utilisé au MIT (équipe de Benoit Forget) et à EDF/R&D (plateforme ODYSSEE)
  - Utilisé par Amazon, YouTube, Google, etc.
- Créé par Guido van Rossum et maintenue par la Python Software Foundation



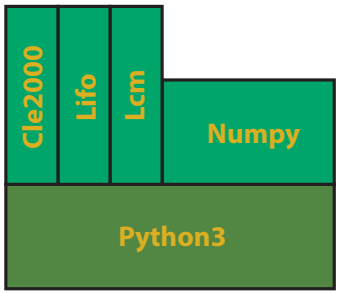
Le langage Python3

Utilisation de Python

- Comme alternative à Matlab?
  - Nécessite l'ajout de modules externes (Numpy, Scipy, Matplotlib, et Pandas)
  - Plus difficile à configurer que Matlab pour le service informatique



- Comme alternative à CLE-2000?
  - Trop lourd pour enchainer des modules de calcul individuels
  - Idéal pour enchainer des procédures globales (schémas de calcul neutroniques)



Le langage Python3

Avantages de Python

- Simplicité d'apprentissage et d'utilisation (similaire à Matlab)
- De nombreux partenaires de l'IGN utilisent Python (EDF/R&D, Framatome, COG (couplage Cathena), INL, etc.)
- Possibilité de créer des modules d'extension (en langage ANSI C) pour des usages spécifiques
  - Le but de ce séminaire est de présenter trois modules d'extension spécifiques aux codes Dragon5 et Donjon5

Défauts de Python

- Le langage n'est pas "upward-compatible":
  - Le code Python2 a de fortes probabilités de ne pas fonctionner en Python3. La version 3 est complètement différente de la version 2.
  - L'outil de conversion de Python2 à Python3 (`$ 2to3 example.py`) a une capacité limitée
  - Les modules d'extensions Python2 ne fonctionnent pas en Python3
  - La spécification du langage n'est pas encadrée par l'ISO
  - L'utilisation de Python2 est fortement déconseillée
- Numpy est indispensable à l'utilisation de Python en génie, mais n'est pas intégrée au langage (contrairement aux arrays de Java)
- La gestion des nombreuses extensions requises par le langage pour différents OS requiert un support informatique professionnel
  - Il peut être moins coûteux d'acquérir une licence commerciale Matlab

Les modules d'extension PyGan

Historique du développement

2001 Développement des bindings Python2-Ganlib

- Programmation par Laurent Plagne (EDF R&D) dans le câdre de l'opération Descartes

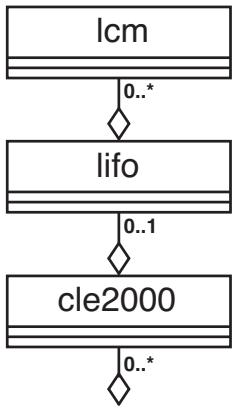
  L. Plagne, "PyLCM – Description des bindings Python," Rapport EDF-R&D-HI-76/01/010/A, 2001.

- Élément d'un prototype (Sunset) visant à évaluer le remplacement de Gibiane par Python

2008 Disponibilité de Python3

2020 Développement des bindings Python3-Ganlib

- Reprogrammation complète de 3 classes Python3 basées sur le modèle de classes de Skin++



Les modules d'extension PyGan

Installation

- Python3 et numpy sont prérequis
- Définir la variable d'environnement `FORTRANPATH` (de préférence dans votre script `.profile`) afin de localiser la bibliothèque `libgfortran.a`. Par exemple:

```
export FORTRANPATH=/usr/local/lib/gcc/9/ # directory with libgfortran.a
```

- Utiliser la commande `make` dans le répertoire `Version5/PyGan/`:

```
cd Version5/PyGan
make
```

Par défaut, des bindings sont créés pour Dragon5 et Donjon5. Il est possible d'installer des bindings uniquement pour Dragon5:

```
cd Version5/PyGan
make dragon
```

Les tests de non-régression sont exécutés par la commande

```
make tests
```

Un cas-test particulier est exécuté par la commande `rpython`:

```
.\rpython simplePOW.py
```

Le module d'extension lcm

Le module d'extension `lcm` permet un accès in-out aux objets LCM de DRAGON5/DONJON5 et à leur contenu au moyen d'un mapping de l'opérateur " `[]` " en Python. Voici un tutoriel interactif:

```
cd Version5_beta_ev2547/PyGan
export PYTHONPATH=lib/Linux_x86_64/python/
python3
>>> import lcm
>>> from assertS import *
>>> import numpy as np
>>> my_lcm=lcm.new('LCM_INP','nonfuel')
>>> my_lcm._impx=3
>>> my_lcm.lib()
>>> my_lcm.keys()
>>> sign=my_lcm['SIGNATURE']
>>> print('object signature=', sign)
>>> daughter=my_lcm['REFL']
>>> daughter.lib()
>>> o2=lcm.new('LCM_INP','new_dictionary',pyobj=daughter)
>>> state=o2['STATE-VECTOR']
>>> print('state vector=', state)
>>> o3=daughter['MIXTURES']
>>> ia=np.array([8, 7, 8, 4, 9, 1, 0, 4], dtype='i')
>>> ra=np.array([8.0,6.0,5.0,2.0,1.0], dtype='f')
>>> da=np.array([8.0,6.0,5.0,2.0,1.0], dtype='d')
>>> o2['key1']='new comments for this record'
>>> o2['key2']=ia
>>> o2['key3']=ra
>>> o2['key4']=da
```

`olcm=lcm.new(type[,name][,iact][,pyobj][,lrda][,impx])`

`olcm._impx`
`olcm._access`
`olcm._type`
`olcm._name`

`olcm.lib()`
`olcm.val()`
`olcm.close()`
`olcm.erase()`
`tuple=olcm.keys()`
`length=olcm.len()`
`o2lcm=olcm.rep({hkey|ikey},length)`
`o2lcm=olcm.lis({hkey|ikey}`
mapping: `olcm[hkey]`
mapping: `olcm[ikey]`

Le module d'extension lifo

Le module d'extension `lifo` permet un accès in-out aux objets lifo (stack "last in first out") utilisés par CLE-2000.

```
cd Version5_beta_ev2547/PyGan
export PYTHONPATH=lib/Linux_x86_64/python/
python3
>>> import lifo, lcm
>>> import numpy as np
>>> x=lifo.new()
>>> x._impx=1
>>> x.push('tata')
>>> x.push(int)
>>> x.push(12345678)
>>> x.push(2.5)
>>> x.push(3.5)
>>> x.push(float)
>>> x.pushEmpty('my_new_LCM_object','LCM')
>>> x.push('this is very looong text')
>>> print('val(3)=',x.node(3)
>>> val=x.pop()
>>> print('val(pop)=',val)
>>> ilen=x.getMax()
>>> print('stack length=',ilen)
>>> my_lcm=lcm.new('LCM_INP','nonfuel')
>>> x.push(my_lcm)
>>> x.push('more text')
>>> new_lcm=x.node('nonfuel')
>>> x.pop()
>>> x.pop()
>>> x.lib()
```

olifo=lifo.new([impx])
olifo._impx
olifo.lib() olifo.push(obj) olifo.pushEmpty(name[,type]) obj=olifo.pop() obj=olifo.node({ipos|name}) length=olifo.getMax() name=olifo.OSName(ipos)

Le module d'extension cle2000

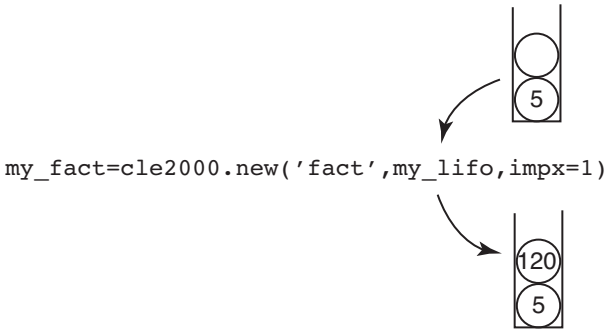
Le module d'extension `cle2000` permet d'encapsuler Ganlib5, Trivac5, Dragon5 ou Donjon5 et d'exécuter une procédure CLE-2000 appelant des modules de ces codes ou des sous-procédures CLE-2000.

```
cd Version5/PyGan
export PYTHONPATH=lib/Linux_x86_64/python/
python3
>>> import lifo,cle2000
>>> my_lifo=lifo.new()
>>> my_lifo.push(5)
>>> my_lifo.push(int)
>>> my_fact=cle2000.new('fact',my_lifo,1)
>>> my_fact.exec()
>>> print('factorial(5)=', my_lifo.node(1))
```

On rappelle la procédure CLE-2000 `fact.c2m`:

```
INTEGER   n n_fact prev_fact ;
 ::  >>n<< ;
IF n 1 = THEN
  EVALUATE n_fact := 1 ;
ELSE
  EVALUATE n := n 1 - ;
  ! Here, "fact" calls itself
  PROCEDURE fact ;
  fact ::   <<n>>  >>prev_fact<< ;
  EVALUATE n_fact := n 1 + prev_fact * ;
ENDIF ;
 :: <<n_fact>>  ;
QUIT " Recursive procedure *fact* XREF " .
```

ocle2000=cle2000.new(procname,olifo[,impx])
ocle2000._impx
ocle2000.exec() olifo=ocle2000.getLifo() ocle2000.putLifo(olifo)



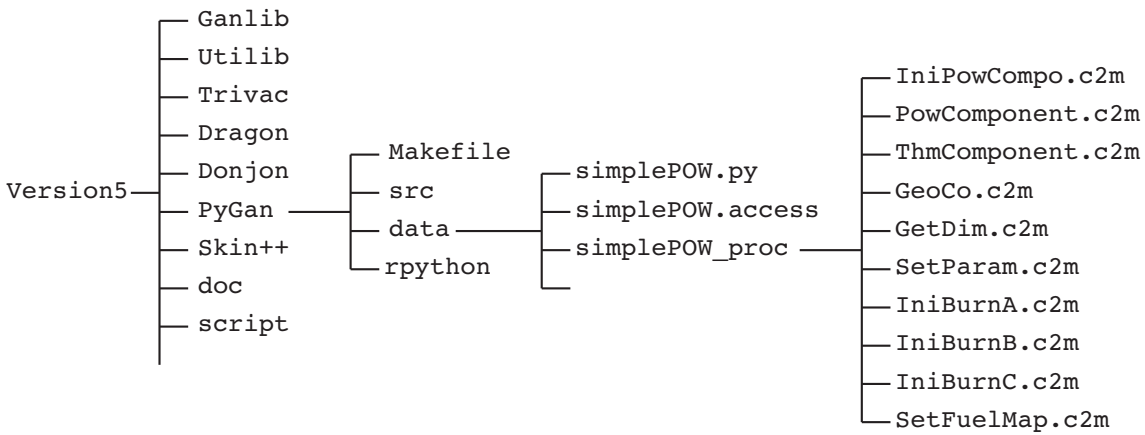
Le module d'extension cle2000

Remarque: L'équivalent Python3 de la procédure `fact.c2m` est:

```
def factorial(n):
    if n == 1:
        return 1
    else:
        return (n * factorial(n-1))
```

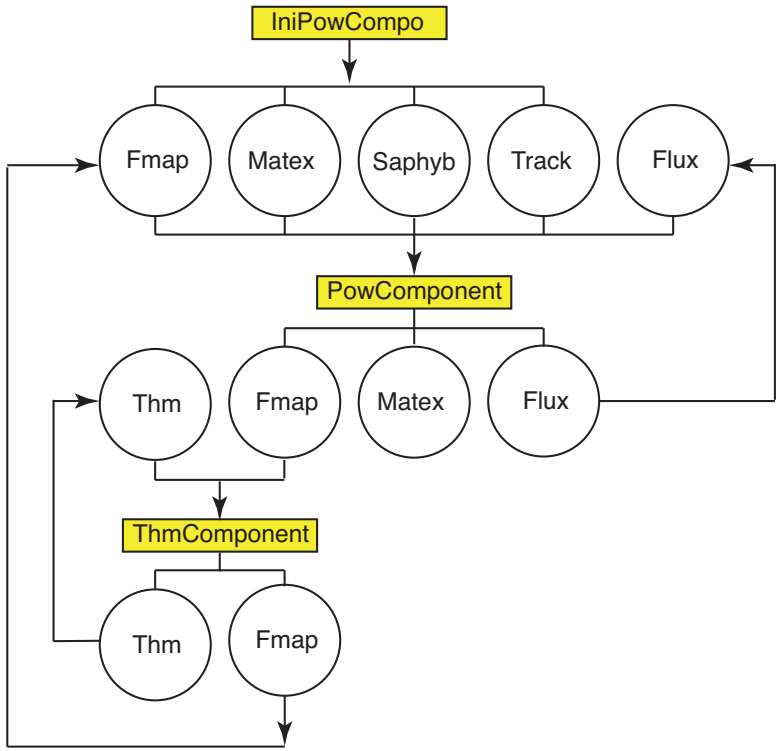
Multiphysics calculations

- Our multiphysics application involves reactor physics and simplified thermal-hydraulics in steady-state conditions.
- This application converges the assembly power distributions and thermal-hydraulics values for the fuel temperature, coolant temperature and density of a 900 MWe PWR.
- The multiphysics iteration is implemented
  - as a single while loop programmed in Python3
  - using three CLE-2000 procedures calling themselves other CLE-2000 procedures
- The `lifo` and `cle2000` extension modules are used to bind Donjon5 with Python3.
- This multiphysics testcase is implemented in `Version5/PyGan/data/simplePOW.py`:



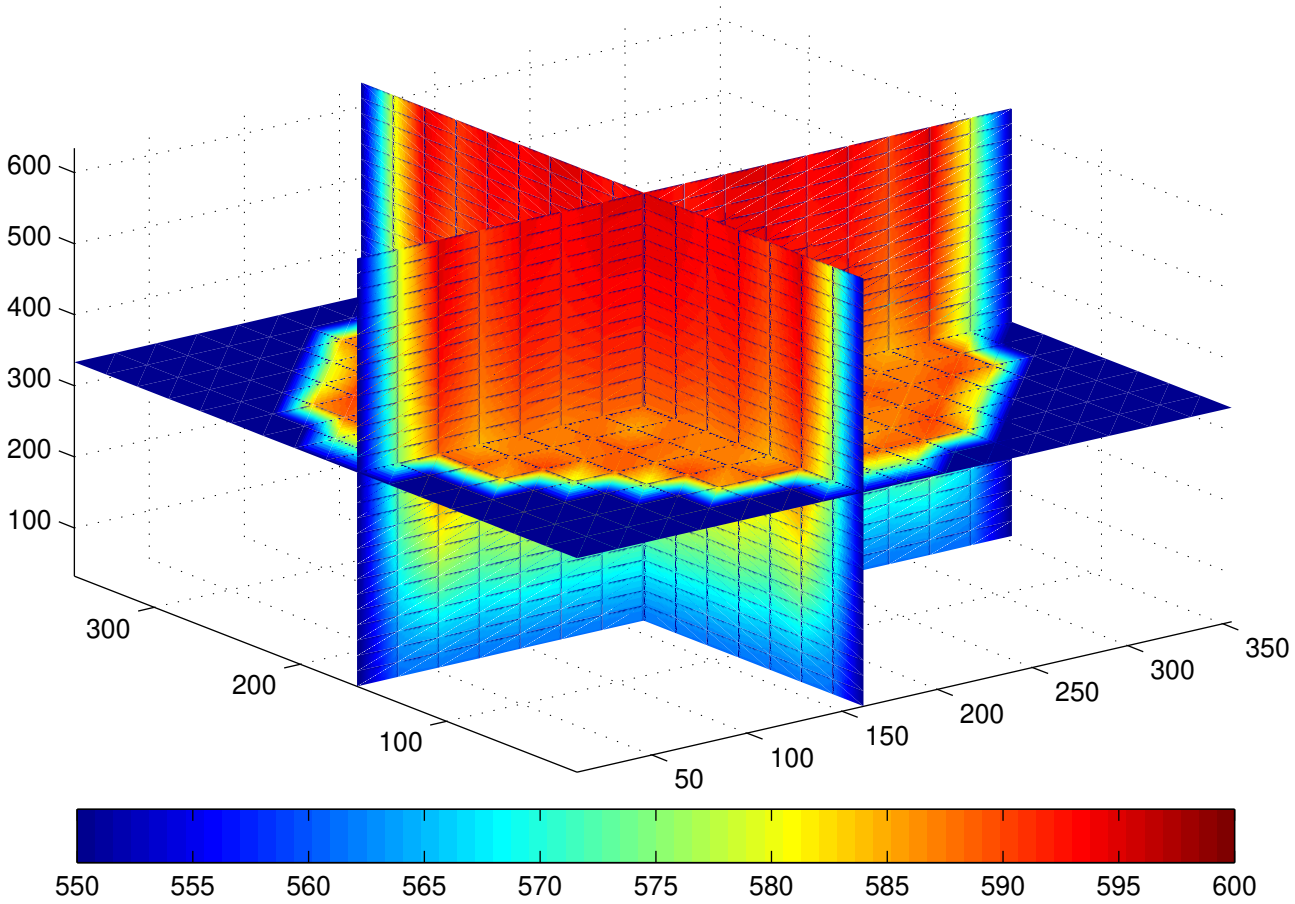
Multiphysics calculations

- The while loop made of three CLE-2000 procedures:
  - `IniPowCompo.c2m`: initialization of the calculation and setting of the initial conditions. The Fmap object contains the parameter distributions in power, burnup, fuel temperature, coolant temperature and coolant density.
  - `PowComponent.c2m`: neutron flux and power calculation involving a single finite-element full-core calculation in DONJON5
  - `ThmComponent.c2m`: simplified thermal-hydraulics calculation, made over each assembly, assuming steady-state conditions



Multiphysics calculations

The converged coolant temperature distribution in core



PyBind

PyBind permet d'exposer des classes C++ en Python

- Compatible avec la version C++11
- Offre une alternative propre aux bindings SWIG et Boost.Python
- Utilisé dans la plateforme ODYSSEE pour réaliser les bindings Python3
- W. Jakob, "pybind11 – Seamless operability between C++11 and Python.

exemple: addition de deux nombres en C++

```
#include <pybind11/pybind11.h>
int add(int i, int j) {
  return i + j;
}
PYBIND11_MODULE(example, m) {
m.doc() = "pybind11 example plugin"; // optional module docstring
m.def("add", &add, "A function which adds two numbers");
}
```

Compilation (les variables d'environnement `PYBIND_ROOT` et `PYTHON3_ROOT` indiquent la position des "include files"pour PyBind et Python3, respectivement):

```
#!/bin/sh
c++ example.cpp -o example‘python3-config --extension-suffix‘ -O3 -Wall \
-shared -std=c++11 -fPIC $python_lib -I $PYBIND_ROOT -I $PYTHON3_ROOT
```

Utilisation en Python3:

```
python3
>>> import example
>>> sum=example.add(1, 2)
>>> print('sum=', sum)
```