Les langages de programmation pour le calcul scientifique

très largement inspiré du cours de Vincent Miele (LBBE, 2016)

Anne Cadiou

Laboratoire de Mécanique des Fluides et d'Acoustique

Séminaires et ateliers pour le calcul scientifique
PMCS2I - LMFA
Vendredi 6 octobre 2017

Pourquoi utiliser un langage de programmation ?

Le langage machine est composé d'instructions et de données codées en binaire (composé de 0 et de 1). Chaque type de processeur possède son propre jeu d'instruction, donc est spécifique au processeur sur lequel on travaille.

Par exemple, un processeur x86 reconnaît une instruction du type

```
10110000 01100001
```

En langage assembleur, langage proche de la machine, cette instruction est représentée par un équivalent plus facile à comprendre pour le programmeur mais étroitement lié à la machine, donc pas portable d'une machine à l'autre.

```
movb $0x61, %al
```

Ces instructions disent à la machine d'écrire le nombre 97
dans le registre AL. La valeur est donnée en hexadécimal : 61 en base 16, c'est 97 (base 10). en binaire c'est 1100001.
(*d'après Violaine Louvet, ICJ, 2014*)

Les langages de programmation servent à faciliter l'écriture des instructions à faire exécuter à la machine.

```cpp
int main()
{
    int a=97;

    return 0;
}
```

Programme C++ `prog.cpp`

```
    .file "prog.cpp"
    .text
    .globl  main
    .type main, @function
main:
.LFB0:
    .cfi_startproc
    pushq %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq  %rsp, %rbp
    .cfi_def_cfa_register 6
    movl  $97, -4(%rbp)
    movl  $0, %eax
    popq  %rbp
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFE0:
    .size main, .-main
    .ident  "GCC: (Ubuntu 5.4.0-6ubuntu1
        ~16.04.4) 5.4.0 20160609"
    .section  .note.GNU-stack,"",@progbits
```

Programme correspondant en assembleur

Il existe de nombreux langages...

pour différents usages. Ils peuvent être classés suivant leur typologie.

Quel(s) langage(s) ?

Quels objectifs ?

Du pragmatisme, pas de dogmatisme

Critères à prendre en compte pour le calcul scientifique

- la performance
- la lisibilité/maintenabilité/programmation orientée objet
- la disponibilité d'API (*package*, *toolbox*, *module*, *library*)
- la possibilité de calcul parallèle
- l'interopérabilité
- la portabilité
- la réutilisabilité
- l'habitude de la communauté

Langages les plus courants en calcul :

Fortran, C, C++, Python, Matlab, R, Java

Quelles différences entre ces langages
pour quelles avantages/inconvénients ?

sont aussi utilisés
Maple, Mathematica, Scilab, Octave, LabView, Delphi, Julia, Pythran, Cython, Ruby, Haskell, OCaml, ...

avec en bonus
make, bash, awk, Perl, LaTeX...

Un peu d'histoire pour le calcul scientifique

- 1840 Premier programme au monde : Ada Lovelace
- 1936 Machine de Turing
- 1947 Assembleur : langage machine
- 1951 A-0 : premier langage compilé
- 1956 FORTRAN (FORmula TRANslator) : premier langage de haut niveau
- 1968 ALGOL (ALGOrithmic Language) : théorique, peu utilisé
- 1971 Pascal : usage pédagogique
- 1972 C : à l'intention des informaticiens, développement d'Unix
- 1978 FORTRAN77 : normalisation importante du Fortran
- 1979 VISICALC (VISIble CALCulator) : premier tableur, ancêtre d'EXCEL
- 1981 C++ : dérivé du C, concept objet
- 1984 Matlab : langage de type script
- 1987 CAML : Langage fonctionnel
- 1990 FORTRAN90 : FORTRAN intégrant l'allocation dynamique
- 1991 Python : objet, interprété
- 1993 R : orienté objet, procédural
- 1994 JAVA : langage objet, simplification du C++, multi-plateforme
- 1995 FORTRAN95 : évolution du FORTRAN90
- 1995 Delphi : basé sur Borland Pascal
- 1996 OCAML (Objective CAML) : Caml objet
- 1996 ANSI C++ : Normalisation du C++
- 2000 C# : C++ simplifié, concurrent du JAVA (Microsoft)
- 2009 Julia : influencé par Matlab, R, Ruby, Perl, Python

d'après Eric Lunéville, ENSTA

Haut niveau versus bas niveau

On parle de *niveau* d'un langage en fonction de la nécessité imposée au programmeur de connaître le fonctionnement d'un ordinateur.

Le langage de très bas niveau est le langage machine binaire.
Assembleur est un langage proche du langage machine. Plus le niveau est bas,

plus les performances sont importantes.

Fortan, C et C++ sont considérés de niveau intermédiaire (permettent la gestion fine de la mémoire et donc de la performance).

Python ou Perl sont des langages de haut niveau, de même que Matlab/Scilab ou R (peu généralistes donc parfois appelés "environnements de programmation scientifique").

Haut niveau versus bas niveau

On parle de *niveau* d'un langage en fonction de la nécessité imposée au programmeur de connaître le fonctionnement d'un ordinateur.

Le langage de très bas niveau est le langage machine binaire.
Assembleur est un langage proche du langage machine. Plus le niveau est bas,

plus les performances sont importantes.

Fortan, C et C++ sont considérés de niveau intermédiaire
(permettent la gestion fine de la mémoire et donc de la performance).

Python ou Perl sont des langages de haut niveau, de même que Matlab/Scilab ou R (peu généralistes donc parfois appelés "environnements de programmation scientifique").

Langage compilé

Un langage est dit *compilé* quand le *code* ou programme *source* sous forme de texte est lu et traité par un autre programme, appelé *compilateur* qui le transforme une fois pour toute en exécutable pour la machine.

Langage interprété

Un programme en langage *interprété* nécessite pour fonctionner un interprète(eur) qui est un autre programme qui va vérifier la syntaxe et traduire en permanence les différentes lignes du programme source, à la volée, au fur et à mesure de son avancement dans le code (un peu comme un interprète durant une interview).

Langage compilé

Un langage est dit *compilé* quand le *code* ou programme *source* sous forme de texte est lu et traité par un autre programme, appelé *compilateur* qui le transforme une fois pour toute en exécutable pour la machine.

Langage interprété

Un programme en langage *interprété* nécessite pour fonctionner un interprète(eur) qui est un autre programme qui va vérifier la syntaxe et traduire en permanence les différentes lignes du programme source, à la volée, au fur et à mesure de son avancement dans le code (un peu comme un interprète durant une interview).

Langage compilé vs interprété : des usages différents

La compilation et l'édition des liens produisent un exécutable autonome qui ne fonctionne que sur le type de machine où la compilation s'est déroulée.

C, C++, Fortran sont des langages compilés
(avec différents compilateurs possibles, gratuits ou non).

Un programme interprété sera plus lent qu'un programme compilé du fait de la traduction dynamique. Quand une ligne du programme doit être exécutée un grand nombre de fois, l'interpréteur la traduit autant de fois qu'elle est exécutée.
Néanmoins la correction des erreurs est plus interactive, car l'interprète signale à l'execution où se trouve l'erreur, et le code source venant d'être écrit peut être directement testé.
Un programme dans un langage interprété est parfois appelé *script*.

Python, R, Matlab sont des langages interprétés (qui peuvent aussi être compilés dans certains cas...)

Langages statiques

Les langages de typage fort dit *statique* imposent la déclaration précise de toutes les variables (type, signe, taille) et les éventuelles conversions doivent être explicites.

Strict mais sûr.

Fortran, C++

Langages dynamiques

Les langages non typés ou de typage *dynamique* sont très souples avec les variables : pas de déclaration et possibilité de changement de type à la volée.

Le temps de développement est réduit mais des erreurs non détectables par le compilateur ou l'interpréteur sont possibles (e.g. faute de frappe dans le nom de la variable).

La grande flexibilité que permet le typage dynamique se paye en général par une surconsommation de mémoire correspondant à l'encodage du type dans la valeur.

Python, R, Matlab

Langage *procédural*

Codes basés sur un enchainement de procédures/fonctions sur des variables globales

- développement de plusieurs fichiers, découpage complexe
- variables globales en début de programme
- manque parfois de lisibilité de la liste des arguments
- difficile à modifier car tout est imbriqué
- difficile de ne tester qu'une partie du code car tout est couplé
- code difficilement extensible à des usages non prévus au départ

C, Fortran, Matlab ou R sont des langages procéduraux (malgré des évolutions)

Langage *orienté objet*

Code permettant de définir des nouveaux types de variables (objets) auxquels sont attachés des données (ou d'autres objets), des fonctions ou des opérateurs

- nécessite d'identifier et regrouper les choses qui vont bien ensemble en modules
- facilite le test des modules indépendamment les uns des autres
- permet un niveau de modularité élevé avec la séparation des modules (concepts)
- permet de réduire au strict minimum la visibilité inter-modules (encapsulation)
- possibilité de structuration hiérarchique des modules (héritage)

C++, Python, Java, sont nativement des langages orientés objet

$\Rightarrow$ un code écrit en Fortran ou C++ ne se conçoit pas de la même façon

Comparatif et lieux communs

Fortran

- langage de référence du calcul scientifique
- très rapide
- nombreuses bibliothèques optimisées
- allocation dynamique, calcul matriciel, couche objet

C++

- langage près du système (héritage du C)
- couche objet complète (héritage, surcharge, prog. générique)
- pas de variables native de type matrice
- plus difficile à maîtriser que le Fortran

Python

- langage du prototypage
- couche objet
- facile à prendre en main
- facile à coupler avec C, C++ ou Fortran
- possède des bibliothèques de calcul scientifique
- plus lent que le Fortran, le C ou le C++
- interopérable
- richesse de fonctionnalités (bibliothèques numériques, graphique,...)

R

- peu adapté au calcul intensif
- couche objet
- facile à prendre en main
- facile à coupler avec C, C++ ou Fortran
- richesse de fonctionnalités (bibliothèques numériques, graphique,...)

Syntaxes proches

Fortran

```
do i = 1,10,2
   print *, i
end do
```

C++

```
for (int i = 1; i < 10; i += 2)
  std::cout << i << std::endl;
```

Python 2.x

```
for i in xrange(2, 9, 2):
    print "%d," % i
```

Python 3.x

```
for i in range(2, 9, 2):
    print("%d, " % i, end="")
```

Julia

```
for i in 2:2:8
    print(i, ", ")
end
```

R

```
for(a in seq(2,8,2)) {
  cat(a, ", ")
}
```

http://rosettacode.org/wiki/Rosetta_Code

Benchmarks : à prendre et à laisser ?

La jungle des benchmarks du web est aussi la jungle des biais (sauf à réunir les hyper-spécialistes de chaque langage), mais elle apporte des tendances avérées. . .

- nasa.gov
- julialang
- stackoverflow
- matrix multiplication
- forums

```
k=0
for i in range(10**8):
    k=k+i
```

4.5 secondes

```
#include <stdlib.h>
#include <math.h>
int main()
{
    int k=0;
    for (int i=0; i>int(pow(10,8)); i++){
        k += k+i;
    }
}
```

poussières de secondes

```
k <- 0
for(i in &:10^8)
    k <- k+i
```

15 secondes

L'interopérabilité

Tendance actuelle : interfacer les langages pour prendre le meilleur de chacun, i.e. permettre la transmission des données entre plusieurs parties implementée dans différents langages.

Les modules Python (packages R) les + efficaces sont des interfaces Python (R resp.) vers C/C++ ou Fortran.

Exemple de NumPy "To build the module, you'll need a C compiler. Various NumPy modules use Fortran 77 libraries, so you'll also need a Fortran 77 compiler"

De très nombreux codes C++ sont interfacés avec les routines Fortran 77 de référence (BLAS, LAPACK, ARPACK) mais attention à l'ordre en mémoire :

$t[0][0], t[0][1] \dots t[n][m-1], t[n][m]$
en C++

$t(1,1), t(2,1), \dots, t(m-1,n), t(m,n)$
en Fortran

Réalisation de simulations numériques

La mise en oeuvre de simulations numériques fait intervenir à différentes étapes différents types de langages, interfacés ou chaînés (haut niveau : interprété, bas niveau : compilé).

Classiquement les étapes suivantes sont présentes :

- mise en données (e.g. haut niveau)
- définition des campagnes de simulation (configuration, mise en données) (e.g. haut niveau)
- étape de calcul (solveur, plutôt bas niveau)
- analyse, post-traitement, visualisation, I/O (e.g. haut niveau)

et dans les phases de développement, le prototypage d'un solveur peut être exploré avec un langage interprété, qui peut être optimisé ou conduire au développement d'un code de calcul en langage compilé.

Pour faire un choix, répondre aux questions suivantes :

- quel(s) langage(s) connaissez-vous déjà ?
- quelle est la taille du problème à traiter ?
- quel est le degré et le type de parallélisme nécessaire pour traiter le problème donné ?
- quelle généricité doit avoir le code (diversité des problèmes à traiter) ?
- est-ce que l'algorithme visé peut bénéficier de l'usage de pointeurs ?
- de quelles bibliothèques disposez-vous déjà ?

Pour se lancer

- suivre des bonnes pratiques de développement (versionnement, tests unitaires, ...)
- faire du profiling, utiliser des métriques
- suivre l'évolution des techniques par une veille technologique (littérature, réseaux métiers, forum experts...)