Langages

Informatique scientifique pour le Calcul
Ecoles Doctorales 2016

Vincent Miele

CNRS - Biométrie & Biologie Evolutive

Février 2016

Passer du papier à l'ordinateur

Quel(s) langage(s) ?

Quels objectifs ?

Du pragmatisme, pas de dogmatisme

- la performance
- la lisibilité/maintenabilité/POO
- la disponibilité de librairies (“package”, “toolbox”, “module”, “library”)
- l'adaptabilité au calcul parallele
- l'interopérabilité

Dans ce cours, on se concentrera sur les langages les plus courants en Calcul :

C/C++, (Fortran), Python, (Perl), R, Matlab (Scilab)

Quelles différences entre ces langages
pour quelles avantages/inconvénients ?

Haut vs 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.

Plus le niveau est bas, plus les performances sont importantes.

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.

Plus le niveau est bas, plus les performances sont importantes.

Fortan et C/C++ sont considérés de niveau intermédiaire (permet la gestion fine de la mémoire par exemple, 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").

Un langage est dit *compilé* quand le *code* ou programme *source* sous forme de texte est tout d'abord lu et traité par un autre programme appelé *compilateur* qui le convertit en langage machine.

Compilé vs interprété

Un langage est dit *compilé* quand le *code* ou programme *source* sous forme de texte est tout d'abord lu et traité par un autre programme appelé *compilateur* qui le convertit en langage machine.

Le compilateur signale les erreurs syntaxiques présentes dans le code source.

Cette phase de compilation peut parfois être très longue.

La compilation (et *l'édition des liens*, voir cours suivants) produit un exécutable autonome qui ne fonctionne que sur le type de machine (OS, 32/64 bits) où la compilation s'est déroulée.

C++, Fortran sont des langages compilés (avec un choix de compilateurs gratuits ou non).

Un programme en langage *interprété* nécessite pour fonctionner un interprète(eur) qui est un autre programme qui va vérifier la syntaxe et traduire directement, au fur et à mesure de son exécution, le programme source en langage machine (un peu comme un interprète durant une interview).

Un programme en langage *interprété* nécessite pour fonctionner un interprète(eur) qui est un autre programme qui va vérifier la syntaxe et traduire directement, au fur et à mesure de son exécution, le programme source en langage machine (un peu comme un interprète durant une interview).

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 sera plus simple car l'interprète signale à l'execution où se trouve l'erreur. Et le code source venant d'être écrit peut être directement testé.

NB : un programme dans un langage interprété est parfois appelé *script*.

Python, R, Matlab (pas si simple. . . , voir après) sont des langages interprétés.

	Included in Level			
Optimization	-O1	-O2	-Os	-O3
defer-pop	●	●	●	●
thread-jumps	●	●	●	●
branch-probabilities	●	●	●	●
cprop-registers	●	●	●	●
guess-branch-probability	●	●	●	●
omit-frame-pointer	●	●	●	●
align-loops	○	●	○	●
align-jumps	○	●	○	●
align-labels	○	●	○	●
align-functions	○	●	○	●
optimize-sibling-calls	○	●	●	●
cse-follow-jumps	○	●	●	●
cse-skip-blocks	○	●	●	●
gcse	○	●	●	●
expensive-optimizations	○	●	●	●
strength-reduce	○	●	●	●
rerun-cse-after-loop	○	●	●	●
rerun-loop-opt	○	●	●	●
caller-saves	○	●	●	●
force-mem	○	●	●	●
peephole2	○	●	●	●
regmove	○	●	●	●
strict-aliasing	○	●	●	●
delete-null-pointer-checks	○	●	●	●
reorder-blocks	○	●	●	●
schedule-insns	○	●	●	●
schedule-insns2	○	●	●	●
inline-functions	○	○	○	●
rename-registers	○	○	○	●

Le compilateur réalise (à la demande) des *optimisations* qui permettent de générer un code plus efficace. Toutefois, le compilateur “prend des risques” (il essaie de se mettre dans la tête du programmeur avec +/- de succès) et du temps.

- *loop unrolling* :

```
for (int i=0; i<n; i++)
{
  sum += data[i];
}
```

```
for (int i=0; i<n; i+=4)
{
  sum1 += data[i+0];
  sum2 += data[i+1];
  sum3 += data[i+2];
  sum4 += data[i+3];
}
sum = sum1 + sum2 + sum3 + sum4;
```

- *inlining* : Effective STL, Scott Meyers, item 46 C++ `sort` est plus + rapide que C `qsort` si on utilise un functor
- voir ci-contre. . .

Typologie des langages

Compilé vs interprété

```
1  #include <vector>
   #include <iostream>
3  using namespace std;

5  class Matrix
   {
7  private:
     int _s;
9    vector<double> _vinternal;
   public:
11   Matrix(int s)
       : _vinternal(s*s, 0), _s(s) {}
13   ~Matrix() {}
     inline double& element(int i, int j){
15     return _vinternal[i*_s+j];
     }
17 };

19 int main(int argc, char ** argv )
   {
21   int s = 20000;

23   Matrix m(s);
     // 10.30 secondes avec -O0
25   // 1.69 secondes avec -O3
     for (int i=0; i<s; i++)
27     for (int j=0; j<s; j++)
         m.element(i,j) = m.element(i,j)+m.element(i,j);
29 }
```

Typage statique vs dynamique

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.

Rigidité mais sécurité.

Fortran, C++ (NB : mot clé `auto` en C++11).

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.

Rigidité mais sécurité.

Fortran, C++ (NB : mot clé `auto` en C++11).

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 (moins *verbose*) mais des erreurs non detectables sont possibles (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* : enchainement de procédures/fonctions sur des données globales

- plusieurs fichiers, mais comment découper ?
- variables globales en début de programme, mais comment ne pas s'y perdre ? si même traitement sur 2 variables ?
- lisibilité de la liste de paramètres ?
  Exple : `fonction correction(nbanimaux, age, poids, taille, seuil, print, nbiterations, .....)`
- Difficile de modifier car tout est imbriqué, idem pour un tester une partie.Tout est accessible par tous (Couplage)
- Pbm de recopie de code (synchronisation !)
- Programme fragile, difficilement extensible

Langage *procédural* : enchainement de procédures/fonctions sur des données globales

- plusieurs fichiers, mais comment découper ?
- variables globales en début de programme, mais comment ne pas s'y perdre ? si même traitement sur 2 variables ?
- lisibilité de la liste de paramètres ?
  Exple : `fonction correction(nbanimaux, age, poids, taille, seuil, print, nbiterations, .....)`
- Difficile de modifier car tout est imbriqué, idem pour un tester une partie.Tout est accessible par tous (Couplage)
- Pbm de recopie de code (synchronisation !)
- Programme fragile, difficilement extensible

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

Langage *orienté objet* : regrouper les données et les fonctionnalités associés en enitités logicielles autonomes

- Identifier et grouper les ≪choses qui vont bien ensemble≫ en modules
- Les modules sont testés indépendemment pour assurer la robustesse du code (tests unitaires)
- Séparer les modules les uns des autres (modularité)
- Réduire au strict minimum la visibilité inter-modules (encapsulation)
- Si nécessaire : structuration hiérarchique des modules (héritage)
- Programme principal en Lego : chef d'orchestre entre modules

Langage *orienté objet* : regrouper les données et les fonctionnalités associés en enitités logicielles autonomes

- Identifier et grouper les ≪choses qui vont bien ensemble≫ en modules
- Les modules sont testés indépendemment pour assurer la robustesse du code (tests unitaires)
- Séparer les modules les uns des autres (modularité)
- Réduire au strict minimum la visibilité inter-modules (encapsulation)
- Si nécessaire : structuration hiérarchique des modules (héritage)
- Programme principal en Lego : chef d'orchestre entre modules

C++ et Python sont nativement des langages orientés objet

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. . .

▸ B1 ▸ B2 ▸ B3 ▸ B4

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...

▸ B1 ▸ B2 ▸ B3 ▸ B4

‘C++ rocks”, Python Numpy... lenteur du “pur” Python ou R. Mais est-ce aussi simple ?

Typologie des langages

Benchmarks, à prendre et à laisser ?

```python
k = 0
for i in range(10**7): # 4.5 seconds
  k = k+i
```

```c
#include <stdlib.h>
#include <math.h>
int main()
{
  int k = 0;
  for (int i=0; i<int(pow(10,7)); i++){ // poussieres de secondes
        k += k+i;
  }
}
```

```r
k <- 0
for(i in 1:10^7) # 15 seconds
  k <- k+i
```

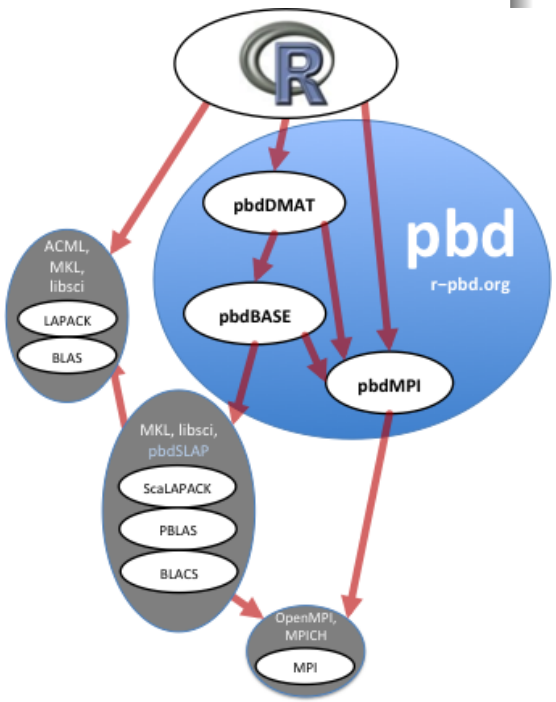
LA tendance : interfacer les langages pour prendre le meilleur de chacun, i.e. permettre la transmission des données entre plusieurs parties implementée dans différents langages.

LA tendance : 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.

L'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 :
$t[0][0], t[0][1] \ldots t[n][m-1], t[n][m]$ en C++
mais
$t(1,1), t(2,1), \ldots, t(m-1,n), t(m,n)$ en Fortran !



LA tendance : interfacer des langages en calcul.

Schéma classique d'un code de calcul (HNI : langage de haut niveau interprété, NIC : niveau intermédiaire compilé)

- lecture des données + pré-traitement HNI
- étapes de calcul MNC
- post-traitement, visualisation et/ou écriture des données HNI

See "Extending Python with C or C++" or "Writing R extensions with foreign language interfaces"

Perl is useful but painful !

Utilizing Pythons built-in methods or external modules can produce near or better than C++ performance.

Apprendre Python, c'est apprendre le meilleur du C++, du Java, du Fortran, etc...

Python ran an average of xxx slower than C++

The best thing about R is that it was developed by statisticians. The worst thing about R is that...it was developed by statisticians !

C++ is the only way to go for low level systems programming C++ is life ! It's beauty ! It's elegance and performance !

C/C++

+++	Langage très performant (si on fait du “vrai” C++, pas du C)
–	Apprentissage difficile et long (pour tirer le meilleur du langage), apprentissage de la compilation&co (voir cours suivants)
+++/-	Verbeux mais très sûr
–	Pas adapté pour le prototyping
+	Interface “tricky” avec Fortran
+++	POO nativement
+	Boost librairies, C++11
+++	Adapté à tous les niveaux de parallèlisme : classique (openMP, MPI) ou émergent (TBB, Silk)
-	E/S et graphiques

Python

Python remplace avantageusement Matlab (payant, code propriétaire)
`I used Matlab. Now I use Python.`

+/-	Performant ssi utilisation des modules ad-hoc (NumPy)
+++	Apprentissage facile. Adapté à la pédagogie de programmation
++	Rapidité de développement
+++	Prototyping
+++	Interface avec le C++ avec swig, cython, Boost.Python ou weave
+++	POO nativement
++/-	Jungle des modules
+	Modules pour le parallélisme classique (multiprocessing, MPI4py)
++	E/S et graphiques

R

+/-	Langage peu performant (pas de passage by reference)
	sauf si on chaîne les packages plus qu'on ne programme avec la syntaxe R
+/-	Apprentissage ambigu (facile pour les non-programmeurs)
++	Rapidité de développements de petits programmes
+++	Prototyping
+++	Interface avec C++ et Fortran (R-extensions)
-	POO non native et peu efficace (S4)
+++/-	Jungle des packages
+	Emergence du parallélisme avec R
+++	E/S et graphiques

Typologie des langages

exple-lang4

```
f = open('lang4.dat')
dico = {}

for line in f.xreadlines():
    elts = line.strip().split(' ')
    dico[elts[0]] = elts[1]

for key in dico.keys():
    print dico[key],' pour la cle ',key
```

```
PuyDeDome 63
Cantal 15
Hauteloire 43
Allier 03
```

```
vmiele@tulipe$ python lang4.py
43  pour la cle  Hauteloire
15  pour la cle  Cantal
03  pour la cle  Allier
63  pour la cle  PuyDeDome
```

Typologie des langages

exple-lang4

```
1  #include<fstream>
   #include<iostream>
3  #include<map>
   #include<string>
5  #include<sstream>
   using namespace std;
7  int main()
   {
9    ifstream f;
     f.open("lang4.dat");
11   map<string, int> dico;

13   string line;
     while (!f.eof()){
15     getline(f, line);
       istringstream linestream(line);
17     string a;
       int b;
19     linestream>>a;
       linestream>>b;
21     dico[a] = b;
     }
23
     for (map<string, int>::iterator iter=dico.begin();
25        iter!=dico.end(); iter++){
       cout<<iter->first<<" pour la cle "<<iter->second<<endl;
27   }
   }
```

Ouvrir des livres et des revues

+ faire des benchmarks et du profiling (voir cours suivants)

+ participer à des réseaux métiers/collègues

+ éviter tout dogmatisme

+ Lecture : C++ coding standards, Herb Sutter, item 6 “Correctness, simplicity and clarity come first”