UNIVERSITÉ PARIS 7 - DENIS DIDEROT

Comment faire communiquer C, Python, R ... ?

Objectif :

- Interfacer des programmes écrits dans différents langages

Application :

- Importer une librairie partagée C dans R
- Importer une librairie partagée C dans Python

Plan

I. Interface C / R :
   1. Lancer une librairie dynamique sous R
   2. Passer des arguments
   3. Modifier des valeurs
   4. Interfaçage pour un tableau 2D
   5. Fonctions d'interfaçage sous R

II. ***Swig***
   1. Généralités
   2. Application, interface C / Python

Plan

I. Interface C / R :
   1. Lancer une librairie dynamique sous R
   2. Passer des arguments
   3. Modifier des valeurs
   4. Interfaçage pour un tableau 2D
   5. Fonctions d'interfaçage sous R

II. ***Swig***
   1. Généralités
   2. Application, interface C / Python

Chargement d'une librairie C dans R

Rien de tel qu'un exemple pour comprendre :

```
$> cat fic.c
#include <stdio.h>
void hello (){
   printf("Test reussi\n");
}
$> gcc -c fic.c
$> gcc -shared fic.o -o libMBI.so
$> R
   > dyn.load("libMBI.so")
   > is.loaded("coucou")
   [1] FALSE
   > is.loaded("hello")
   [2] TRUE
   > .C("hello")
   Test reussi
   >
```

Attention aux subtilités :

- La fonction appelée par .C dans R ne doit pas retourner de valeur
- Lorsque l'on fait appel à une fonction par .C les objets passés en argument sont copiés !!!
- De même après l'appel, les objets sont à nouveau copiés pour être mis dans une liste

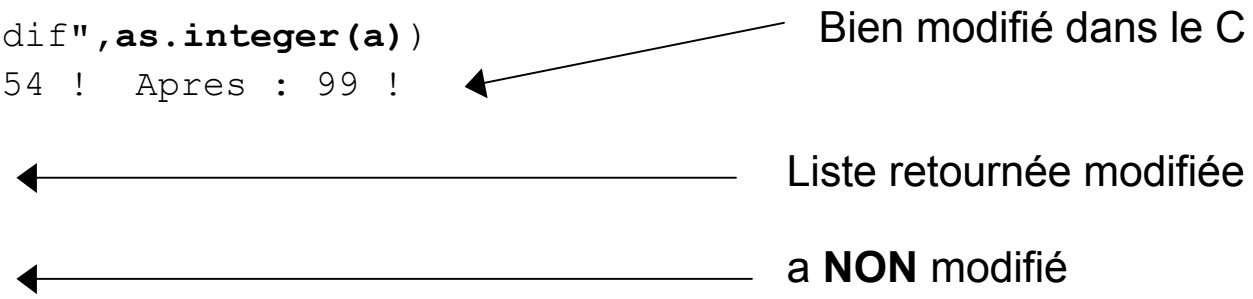
Passage d'arguments

- Les objets passés en arguments sont considérés comme des pointeurs dans la fonction C
- Comme les objets sont copiés lors de l'appel via .C,
  c'est l'adresse de la copie qui est transmise à la fonction

```
$> cat fic.c
#include <stdio.h>
void test (int *a){
      printf("Test reussi: %d !\n",*a);
}
$> gcc -c fic.c
$> gcc -shared fic.o -o libMBI.so
$> R
   > dyn.load("libMBI.so")
   > a = 54
   > .C("test",as.integer(a))
   Test reussi : 54 !
   [[1]]
   [1] 54
```

Modification d'une valeur

```
$> cat fic.c
   #include <stdio.h>
   void mod (int *a){
        printf("Avant: %d !  ",*a);
        *a = 99;
        printf("Apres: %d !\n",*a);
   }
$> gcc -c fic.c
$> gcc -shared fic.o -o libMBI.so
$> R
      > dyn.load("libMBI.so")
      > a = 54
      > .C("modif",as.integer(a))
      Avant : 54 !  Apres : 99 !
      [[1]]
      [1] 99
      > a
      [1] 54
```



Modification d'une valeur : Solution

```
$> cat fic.c
   #include <stdio.h>
   void mod (int *a){
        printf("Avant: %d !  ",*a);
        *a = 99;
        printf("Apres: %d !\n",*a);
   }
$> gcc -c fic.c
$> gcc -shared fic.o -o libMBI.so
$> R
      > dyn.load("libMBI.so")
      > a = 54
      > a = .C("modif",z = as.integer(a))$z
      Avant : 54 !  Apres : 99 !
      > a
      [1] 99
```

Une valeur est retournée dans a

Laquelle ? L'élément z de la liste

Passage d'un vecteur

- Les arguments sont considérés comme des pointeurs dans la fonction C
- En C les tableaux ne sont que des pointeurs

```
$> cat fic.c
#include <stdio.h>
void testvec (double *v, int *a){
       int i;
       for (i = 0; i < *a; i++){
          printf("v[%d] : %lf\n", i, v[i]);
          v[i] += 10;
       }
}
$> R
   > dyn.load("libMBI.so")
   > v = seq(1,4,1)
   > .C("testvec",as.double(v), as.integer(length(v)))
   v[0] 1.000000
   v[1] 2.000000
   v[2] 3.000000
   v[3] 4.000000
   [[1]]
   [1] 11 12 13 14
```

Passage d'une chaine de caractères

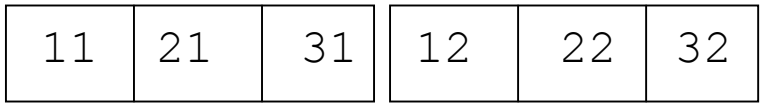
- En R, les chaines de caractères sont codés comme des tableaux de chaines
- Ce sont donc des **`char **`**

```
$> cat fic.c
   #include <stdio.h>
   void testchar (char **s, int *a){
        int i;
        for (i = 0; i < *a; i++){
           printf("s[%d] : %s\n", i, s[i]);
           v[i] += 10;
        }
   }
   $> R
      > dyn.load("libMBI.so")
      > s = "Bonjour !!!"
      > .C("testchar",as.character(s), as.integer(1))
      s[0] : Bonjour !!!
      [[1]]
      [1] "Bonjour !!!"
      [[2]]
      [1] 1
```

Interfacage d'une matrice

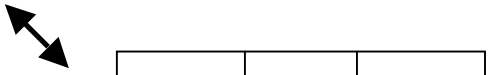
- Sous R :

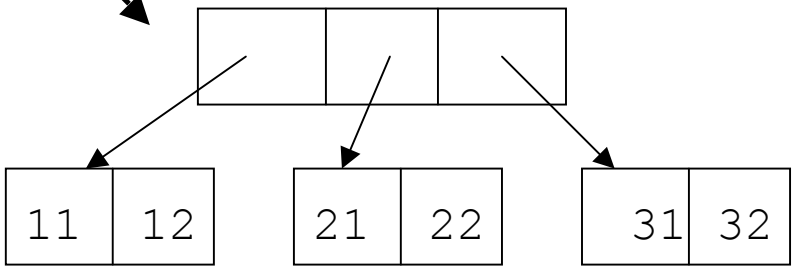
```
     [,1] [,2]
[1,]   11   12
[2,]   21   22
[3,]   31   32
```



En mémoire : 1 tableau → par colonnes

- Sous C :





En mémoire : 1 tableau de tableaux
→ par lignes

→ Besoin d'un programme d'interfacage qui convertit les matrices
!!! Matrice = pointeur simple (comme les vecteurs) !!!

Interfacage d'une matrice : Solution

```
$> cat fic.c
void tabint_R (int *m, int *nl, int *nc){
    int i;
    int **m2 = (int **) calloc (*nl, sizeof (int*));

    for (i = 0; i < *nl; i++){
          m2[i] = &m[i * (*nc)];
    }
   tabint (m2, *nl, *nc);  /* traitement de la matrice */
}
...
$> R
    > dyn.load("libMBI.so")
    > m = matrix(seq(1,9),3,3)
    > m = .C("tabint_R",as.integer(m),as.integer(3), as.integer(3))
    > m
    [1] 1 2 3 4 5 6 7 8 9
    > matrix(m,3,3)
         [,1]  [,2]  [,3]
    [1,]   1     2     3
    [2,]   4     5     6
    [3,]   7     8     9
```

Fonctions d'interfacage sous R

- Il y a quelques subtilités pour interfacer C et R :
  - Variables simples → pointeurs
  - Vecteurs, matrices → pointeurs
  - Chaines de caractères → pointeurs de pointeurs
  - Copie des variables, pas de retour de fonction, etc…
- Pas question de modifier les fonctions écrites en C :
  - Utilisation de la librairie pour des programmes C
- Création de fonctions d'interfacage qui adaptent les types donnés et appellent les "vraies" fonctions C
  - Convention : `maFonction_R`
  - Introduites dans le même fichier .c que la "vraie" fonction

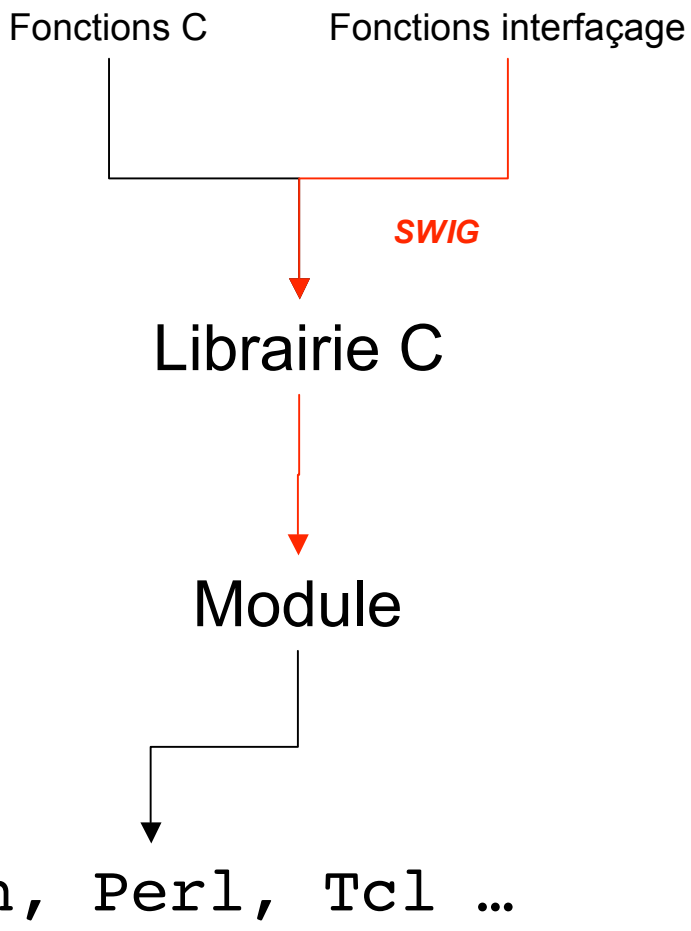
Plan

I. Interface C / `R` :
   1. Lancer une librairie dynamique sous `R`
   2. Passer des arguments
   3. Modifier des valeurs
   4. Interfaçage pour un tableau 2D
   5. Fonctions d'interfaçage sous R

II. ***Swig***
   1. Généralités
   2. Application, interface C / `Python`

SWIG (Simplified Wrapper and Interface Generator)

- Wrapper = emballage
- Permet d'interfacer des langages de script avec du C, C++ (perl, python, tcl …)
- Philosophie générale:




```
/* From Python To C */
long PyInt_AsLong(PyObject * obj );

/* From C To Python */
PyObject *PyInt_FromLong(long x);
```

Entrées / Sorties de ***SWIG***

```c
#include <stdio.h>

void hello( void )
{
  printf("Test reussi !\n");
}
```

hello.c

```
%module libAGM2
%{
/* Put headers and other
   declarations here */
%}

extern void hello( void );
```

hello.i

```
[…]
static PyObject *_wrap_hello(PyObject *self, PyObject *args) {
    PyObject *resultobj;

    if(!PyArg_ParseTuple(args,(char *)":hello")) goto fail;
    hello();

    Py_INCREF(Py_None); resultobj = Py_None;
    return resultobj;
    fail:
    return NULL;
}
[…]
```

hello_wrap.c

Howto …

```
> ls
README.Exemple1 hello.c
hello.i
> swig -python hello.i
> ls
README.Exemple1 hello.c
hello.i         hello_wrap.c
libAGM2.py
>
> gcc -c hello.c hello_wrap.c -
I/usr/include/python2.4
>
> ld -shared hello.o
hello_wrap.o -o _libAGM2.so
>
> python
[blah blah blah]
>>> import libAGM2
>>> libAGM2.hello()
Test reussi !
>>>
```

1. `swig -langage [fichier.i]`

   NB: si python a été choisi, le module python est créé à cette étape (libAGM2.py).

2. compilation des objets avec les headers python
3. création de la librairie partagée (`.so`)
4. Importation du module sous python

   Le module python fait la correspondance entre le .py et la librairie partagée .so

Passage d'arguments (I)

```c
#include <stdio.h>
#include <stdlib.h>

int test(int n)
{
        printf("=> n = %d\n", n);
}

void add( double x, double y,
   double *sum )
{
        *sum = x + y;
}

double add2( double x, double y )
{
        return x+y;
}
```

exemple.c

```
%module libAGM2
%{
/* Put headers and other
   declarations here */
%}

extern  int test(int n);
extern  void add( double x,
   double y, double *sum );
extern  double add2( double
   x, double y );
```

exemple.i

Passage d'arguments (II) *exemple_wrap.c*

```
[…]
static PyObject *_wrap_add(PyObject *self, PyObject *args) {
    PyObject *resultobj;
    double arg1 ;
    double arg2 ;
    double *arg3 = (double *) 0 ;
    PyObject * obj2 = 0 ;

    if(!PyArg_ParseTuple(args,(char *)"ddO:add",&arg1,&arg2,&obj2))
   goto fail;
    if ((SWIG_ConvertPtr(obj2,(void **) &arg3,
   SWIGTYPE_p_double,SWIG_POINTER_EXCEPTION | 0 )) == -1) SWIG_fail;
    add(arg1,arg2,arg3);

    Py_INCREF(Py_None); resultobj = Py_None;
    return resultobj;
    fail:
    return NULL;
}
[…]
```

Passage d'arguments (III) *exemple_wrap.c*

```
[…]
static PyObject *_wrap_add2(PyObject *self, PyObject *args) {
    PyObject *resultobj;
    double arg1 ;
    double arg2 ;
    double result;

    if(!PyArg_ParseTuple(args,(char *)"dd:add2",&arg1,&arg2)) goto
   fail;
    result = (double)add2(arg1,arg2);

    resultobj = PyFloat_FromDouble(result);
    return resultobj;
    fail:
    return NULL;
}
[…]
```

Bah alors …

A vous de jouer … bon courage !

et n'oubliez pas:

http://swig.sourceforge.net/