Interfacing C with R

Jarrod Hadfield

University of Edinburgh

April 22, 2015

Pros of R and C

Pros of R

- Easy & fast to develop
- Documentation is good
- Code does not have to be compiled

Pros of R and C

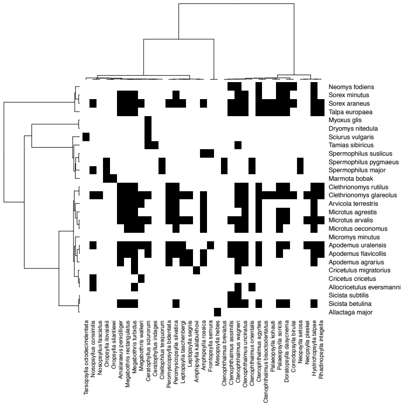
Pros of R

- Easy & fast to develop
- Documentation is good
- Code does not have to be compiled

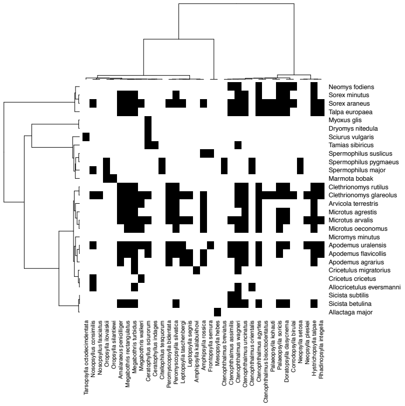
Pros of C

- Very fast to run
- Bugs are easier to detect (but harder to find)

An example: pure R



An example: pure R



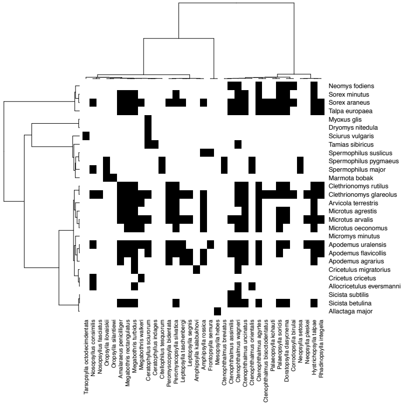
```
pos<-which(Y, arr.ind=TRUE)
N<-nrow(pos)

dh<-dp<-1:(N*(N+1)/2)

cnt<-1
for(i in 1:N){
   for(j in i:N){
      h1<-pos[i,1]
      h2<-pos[j,1]
      p1<-pos[i,2]
      p2<-pos[j,2]
      dh[cnt] <- ht[h1,h2]
      dp[cnt] <- pt[p1,p2]
      cnt<-cnt+1
   }
}

cor(dh,dp)
```

An example: R code calling C



```
pos<-which(Y, arr.ind=TRUE)
N<-nrow(pos)

dh<-dp<-1:(N*(N+1)/2)

output <- .C("hommola",
        as.integer(pos[,1]-1),
        as.integer(pos[,2]-1),
        as.integer(N),
        as.double(c(ht)),
        as.double(c(pt)),
        as.integer(nrow(ht)),
        as.integer(nrow(pt)),
        as.double(dh),
        as.double(dp)
)

cor(output[[8]],output[[9]])
```

An example: C code

```
# include <R.h>
void hommola(
        int *posh,      // Host identifier
        int *posp,      // Parasite identifier
        int *N,         // Number of interactions
        double *ht,     // Pairwise distances between hosts
        double *pt,     // Pairwise distances between paraistes
        int *nh,        // Number of hosts
        int *np,        // Number of parasites
        double *dh,
        double *dp
){

    int i, j, cnt;

    cnt=0;
    for(i=0; i<N[0]; i++){
      for(j=i; j<N[0]; j++){
        dh[cnt] = ht[posh[i]*nh[0]+posh[j]];
        dp[cnt] = pt[posp[i]*np[0]+posp[j]];
        cnt ++;
      }
    }
}
```

An example: C code

```
# include <R.h>
```

- .h files are header files that allow you to access other functions: think of this as library(R)
- Documentation: System and foreign language interfaces of Writing R Extensions provides documentation.

An example: C code

```
void hommola(
        int *posh,      // Host identifier
        int *posp,      // Parasite identifier
        int *N,         // Number of interactions
        double *ht,     // Pairwise distances between hosts
        double *pt,     // Pairwise distances between parasites
        int *nh,        // Number of hosts
        int *np,        // Number of parasites
        double *dh,
        double *dp
){
}
```

- The function 'hommola' should return nothing (void) and have arguments that are pointers (*)
- Pointers contain memory addresses (like a page number in an index)
- In C all variables have to be declared: integer, double ...
- Variables passed as pointers will change externally if modified, this is why the function does not need to output anything

An example: C code

```
int i, j, cnt;

cnt=0;
for(i=0; i<N[0]; i++){
  for(j=i; j<N[0]; j++){
    dh[cnt] = ht[posh[i]*nh[0]+posh[j]];
    dp[cnt] = pt[posp[i]*np[0]+posp[j]];
    cnt ++;
  }
}
```

- C indexing starts at 0 not 1 (This is a good thing!)
- `++ increments a variable by 1`

An example: C code

```c
# include <R.h>
void hommola(
        int *posh,     // Host identifier
        int *posp,     // Parasite identifier
        int *N,        // Number of interactions
        double *ht,    // Pairwise distances between hosts
        double *pt,    // Pairwise distances between paraistes
        int *nh,       // Number of hosts
        int *np,       // Number of parasites
        double *dh,
        double *dp
){

    int i, j, cnt;

    cnt=0;
    for(i=0; i<N[0]; i++){
      for(j=i; j<N[0]; j++){
        dh[cnt] = ht[posh[i]*nh[0]+posh[j]];
        dp[cnt] = pt[posp[i]*np[0]+posp[j]];
        cnt ++;
      }
    }
}
```

Making C code accesible to R

```
R CMD SHLIB filepath/hommola.c
```

- On the command line (i.e. outside R) R CMD SHLIB compiles the C code and creates a shared object hommola.so that R can use.
- You will need a C compiler (e.g. gcc, or install Xcode on Macs)

Making C code accesible to R

```
R CMD SHLIB filepath/hommola.c
```

- On the command line (i.e. outside R) R CMD SHLIB compiles the C code and creates a shared object hommola.so that R can use.
- You will need a C compiler (e.g. gcc, or install Xcode on Macs)

```
dyn.load("filepath/hommola.so")
```

- In R, `dyn.load` dynamicaly loads the shared object so that the C code can be used in R using the `.C` function:

```
output<-.C("hommola", ...)
```

Debugging

R is very forgiving/dangerous:

```
> x<-1:2
> x[3]<-1
> x
```

```
[1] 1 2 1
```

Debugging

R is very forgiving/dangerous:

```
> x<-1:2
> x[3]<-1
> x
```

```
[1] 1 2 1
```

C is pedantic:

```
int x[2];

x[0] = 1;
x[1] = 2;
x[2] = 1;
```

Debugging

R is very forgiving/dangerous:

```
> x<-1:2
> x[3]<-1
> x
[1] 1 2 1
```

C is pedantic:

```
int x[2];

x[0] = 1;
x[1] = 2;
x[2] = 1;
```

- Because x is declared to be of size 2, you have just overwritten a bit of memory that wasn't yours (x[2]; the third element of x).
- There will be no error message, and it will compile with out warning!

Debugging

```
int i, j, cnt;

cnt=0;
for(i=0; i<=N[0]; i++){
  for(j=i; j<N[0]; j++){
    dh[cnt] = ht[posh[i]*nh[0]+posh[j]];
    dp[cnt] = pt[posp[i]*np[0]+posp[j]];
    cnt ++;
  }
}
```

Debugging

```
int i, j, cnt;

cnt=0;
for(i=0; i<=N[0]; i++){
  for(j=i; j<N[0]; j++){
    dh[cnt] = ht[posh[i]*nh[0]+posh[j]];
    dp[cnt] = pt[posp[i]*np[0]+posp[j]];
    cnt ++;
  }
}
```

- At runtime, the outcome may be a segfault (if you're lucky) or infrequent strange behaviour (if you're unlucky)

Debugging

```
> *** caught segfault ***
> address 0x1000080, cause 'memory not mapped'
>
>  Traceback:
> 1: sys.parent()
> 2: sys.function(sys.parent())
> 3: formals(sys.function(sys.parent()))
...
```

Debugging: Print to R console from C

```
int i, j, cnt;

cnt=0;
for(i=0; i<=N[0]; i++){
  for(j=i; j<N[0]; j++){
    dh[cnt] = ht[posh[i]*nh[0]+posh[j]];
    dp[cnt] = pt[posp[i]*np[0]+posp[j]];
    Rprintf("%i\n", i);
    cnt ++;
  }
}
```

Debugging: Use valgrind

Open R on the command line using

```
R -d "valgrind --tool=memcheck --leak-check=full" --vanilla
```

valgrind is debugging software that will find many bugs (although not all - it may miss this bug for example).

Write an R library

In R:

```
package.skeleton("myLibrary", list=myFunctions)
```

or

```
package.skeleton("myLibrary", code_files="filepath/myFunctions")
```

will generate folders/files for your new R library which has a src folder for C code.

Write an R library

In R:

```
package.skeleton("myLibrary", list=myFunctions)
```

or

```
package.skeleton("myLibrary", code_files="filepath/myFunctions")
```

will generate folders/files for your new R library which has a src folder for C code.

On the command line, compile and install:

```
R CMD INSTALL filepath/myLibrary
```

and then use like a standard library.