Faire des simulations au DMS

(lorsque l'on est statisticien)

P. Lafaye de Micheaux[1]

[1] Département de Mathématiques et de Statistique
Université de Montréal

Séminaire midi, 2010

Plan de la présentation

Pourquoi simuler ?

Pourquoi faire des simulations ?

- Pour “vérifier” à l’aide de l’ordinateur un résultat mathématique déjà connu.
- Pour “démontrer” à l’aide de l’ordinateur un résultat que l’on ne parvient pas à démontrer mathématiquement.
- Cela peut nous guider dans la démonstration d’un résultat mathématique difficile.
- C’est souvent exigé lorsque l’on essaye de publier un papier dans une revue.
- Cela permet de mieux raisonner comme un statisticien (voir le chapitre 10 de mon livre sur R).

Comment simuler ?

Comment bien faire des simulations (selon moi) ?

1. Bien cerner le (coeur du) problème.
2. Écrire la trame d'un algorithme permettant de résoudre le problème.
3. Transcrire cet algorithme dans un programme écrit dans un langage interprété comme R.
4. Tester ce programme pour s'assurer qu'il fonctionne correctement.
5. Traduire (éventuellement) le programme en utilisant un langage de plus bas niveau (compilé) comme C ou Fortran.

Avantages de cette approche

- R peut servir à faire des simulations (très correctement) et c'est aussi un logiciel statistique. Donc la plupart des fonctions statistiques sont déjà présentes dans le logiciel.
- En plus, vous apprenez à maitriser un excellent logiciel de statistique.
- Développer un code en R est plus facile qu'en C ou Fortran : consision du code source de votre programme = rapidité pour coder + limitation du risque d'erreurs.
- Traduire (une partie de) votre prog R en C s'il est trop lent.
- On en profite pour apprendre à "maitriser" des outils tels que le C, le calcul parallèle, le calcul sur carte graphique ce qui peut donner un plus sur le marché de l'emploi.

Fil rouge pour cette présentation

On va se donner un problème simple à résoudre, et on va appliquer la démarche dont je viens de parler.

On va voir comment résoudre le même problème

- avec R,
- avec R qui appelle C,
- avec R qui appelle Fortran77,
- avec R qui appelle C qui appelle R,
- avec R et l'utilisation d'un cluster,
- avec R et l'utilisation d'une carte graphique.

Fil rouge pour cette présentation

Note 1 : l'intérêt d'utiliser R comme maître est que tous les résultats des calculs se retrouvent directement accessibles dans R. On peut alors faire des graphiques avec les résultats, faire d'autres calculs sur les résultats, etc. Aussi, on n'est pas obligé de passer par des fichiers intermédaires.

Note 2 : on travaillera sous l'environnement Linux car tous les outils sont déjà présents et que cet OS est très stable. On peut faire la même chose sous Windows mais cela nécessite d'installer plusieurs autres outils au préalable.

La régression linéaire simple : rappels.

Soient

$$Y_i = \beta_0 + \beta_1 x_i + \epsilon_i \quad i = 1, \ldots, n$$

où les ϵ_i sont $i.i.d.$ de loi $\mathcal{L}$. On veut tester

$$H_0 : \beta_1 = 0 \qquad \text{versus} \qquad H_1 : \beta_1 \neq 0.$$

On peut utiliser la statistique du test de Student : $T = \frac{\hat{\beta}_1}{\hat{\sigma}_{\hat{\beta}_1}}$, avec

$$\hat{\beta}_1 = \frac{\overline{xy} - \overline{x} \times \overline{y}}{\overline{x^2} - \overline{x}^2}, \quad \hat{\beta}_0 = \overline{y} - \hat{\beta}_1 \overline{x}, \quad \hat{\sigma}_{\hat{\beta}_1} = \frac{\hat{\sigma}}{\sqrt{n(\overline{x^2} - \overline{x}^2)}} \text{ et}$$

$$\hat{\sigma} = \sqrt{n \left[\overline{y^2} + \hat{\beta}_0^2 + \hat{\beta}_1^2 \overline{x^2} - 2\hat{\beta}_0 \overline{y} - 2\hat{\beta}_1 \overline{xy} + 2\hat{\beta}_0 \hat{\beta}_1 \overline{x} \right] / (n-2)}.$$

La régression linéaire simple : rappels.

On sait que si $\mathcal{L} = N(0, \sigma^2)$, alors $T \sim Student(n-2)$. La procédure de test est alors d'accepter H_1 dès que

$$|t_{obs}| > t_{1-\alpha/2}^{(n-2)}$$

où $t_{1-\alpha/2}^{(n-2)}$ est le quantile d'ordre $1-\alpha/2$ d'une loi de $Student(n-2)$.

À cette procédure de test est associé un risque (dit de 1ère espèce) de se tromper défini par :

$$P[\text{décider } H_1 | H_1 \text{ est faux}].$$

Si $\mathcal{L} = N(0, \sigma^2)$, on peut montrer mathématiquement que l'utilisation de la procédure ci-dessus mène à un risque de 1ère espèce égal à α.

La régression linéaire simple : question de recherche.

Comment le risque de première espèce est-il modifié lorsque $\mathcal{L} \neq N(0, \sigma^2)$? Par exemple lorsque $\mathcal{L} = U(0, 1)$?

Il peut être difficile de répondre mathématiquement à ce genre de question.

Mais on peut utiliser une simulation de Monte Carlo.

Procédure de Monte Carlo : l'algorithme.

Fixons $n = 100$ et donnons nous n valeurs $x_1, \ldots, x_n$ fixées (par exemple $x_i = i$). Fixons également (arbitrairement) $\beta_0 = 3$, et fixons $\beta_1 = 0$ (donc on sait que H_1 est fausse).

Pour $m = 1, \ldots, M$ (M grand, disons 10,000) :

- Générer $y_i = \beta_0 + \epsilon_i$, $i = 1, \ldots,$n où ϵ_i $i.i.d.$ $U(0, 1)$.
- Ajuster le modèle $y_i = \beta_0 + \beta_1 x_i + \epsilon_i$, c'est-à-dire calculer les estimateurs des moindres carrés $\hat{\beta}_1$ et $\hat{\sigma}_{\hat{\beta}_1}$ basés sur les données (x_i, y_i), $i = 1, \ldots, n$.
- Calculer la statistique de test $t_{obs}^{(m)}$.
- Prendre la décision (ou non) d'accepter H_1 si $t_{obs}^{(m)} > q_{n-2}^{(1-\alpha/2)}$.

Procédure de Monte Carlo : l'algorithme.

On compte alors, parmis M, le nombre de fois d où l'on a accepté H_1 (à tort donc puisque H_1 est fausse).

On a alors :

$$P[\text{décider } H_1 | H_1 \text{ est faux}] \approx \frac{d}{M}$$

(pourvu que M soit assez grand).

On pourra alors comparer ce nombre à α pour voir si on sous-estime ou sur-estime le risque de 1ère espèce.

Avec le logiciel R : `reglinR.R`

```
Rfonc <- function(n=100,M=10000,beta0=3,
                        alpha=0.05,xvec=1:n) {
    d <- 0
    seuil <- qt(1-alpha/2,n-2)
    for (m in 1:M) {
      yvec <- beta0 + runif(n)
      tobs <- summary(lm(yvec~xvec))[4]$c[2,3]
      if (abs(tobs)>seuil) d <- d+1
    }
  return(d/M)
  }
system.time(Rfonc())
utilisateur      système       écoulé
     58.961        0.795       62.471
```

Références

- `http://www.dms.umontreal.ca/~stat/Logiciels/R/index.html`
- `http://www.biostatisticien.eu/springeR`

Outline

Avec le langage C/C++ : reglinC.cpp

```
#include <R.h>
#include "Rmath.h"
extern "C" {
void Cfonc(int *n, int *M, double *beta0,
   double *alpha, double *xvec, double *res) {
double qt(double p, double ndf, int lower_tail,
          int log_p);
double runif(double a, double b);
int d=0, m, i;
double seuil, num=0.0, denom=0.0, tobs,
        beta1hat, sighatbeta1hat;
seuil=qt(1.0-alpha[0]/2.0,(double)(n[0]-2),1,0);
double *yvec;
yvec=new double[n[0]];
```

Avec le langage C/C++ : reglinC.cpp

```
 GetRNGstate();
 for (m=1;m<=M[0];m++) {
  for (i=1;i<=n[0];i++) yvec[i-1]=beta0[0] +
                                  runif(0.0,1.0);
  num=...   denom=... // A faire bien sur!
  beta1hat=num/denom;
  sigbethat= ...  // A faire bien sur!
  tobs=beta1hat/sighatbeta1hat;
  if (fabs(tobs)>seuil) d=d+1;
 }
 PutRNGstate();
res[0]=(double)d;
delete[] yvec;
} // Fin Cfonc
}
```

Avec le langage C/C++ : `reglinC.cpp`

Vous aurez noté que l'on a utilisé des librairies externes (`R.h` et `Rmath.h`) qui contiennent les fonctions C `qt()`, `runif()`, `GetRNGstate()`, `PutRNGstate()`. Elles contiennent d'autres fonctions (voir le dossier nmath dans les sources de R).

Il existe plusieurs librairies de fonctions C (statistique, matrices, etc.). Certaines sont gratuites voire libres, d'autres non (Numerical recipies, IMSL, NAG).

Intérêt d'utiliser des outils issus du libre : vous pouvez ensuite distribuer votre travail sans problème.

Références

- http://clips.imag.fr/commun/bernard.cassagne/Introduction_ANSI_C/hyperdoc.html
- http://www.gnu.org/software/gsl/
- http://www.robertnz.net/nm10.htm
- http://www.robertnz.net/nr02doc.htm
- http://www.nrbook.com/a/bookcpdf.php
- http://www.scimath.com

Compiler le prog C/C++

Pour compiler le fichier `reglin.cpp` et créer une librairie partagée (.so=shared object, ou DLL=dynamic linked library sous Windows) on utilise :

```
R CMD SHLIB reglinC.cpp
```

Cela crée un fichier `reglinC.so`

Note : à cette étape, le compilateur nous informe des éventuelles erreurs à corriger dans notre programme (avec le numéro de ligne).

Voyons comment appeler ce programme depuis R.

Appeler un prog C/C++ depuis R : `reglinRC.R`

```
masimul1 <- function(n=100,M=10000,beta0=3,
                     alpha=0.05,xvec=1:n) {
 dyn.load("reglinC.so")
 res <- 0
 out <- .C("Cfonc",as.integer(n),as.integer(M),
           as.double(beta0),as.double(alpha),
           as.double(xvec),val=as.double(res))
 dyn.unload("reglinC.so")
return(out$val/M)
}
```

Appeler un prog C/C++ depuis R : `reglinRC.R`

```
source("reglinRC.R")
system.time(masimul1())
utilisateur     système      écoulé
      0.191       0.001       0.211
```

62.471/0.211=296 fois plus rapide !

Pour donner un ordre d'idée :
10mn en R/C *versus* 2 jours en R !

Appeler un prog R depuis un prog C/C++ depuis R

Voir les fichiers `callRfunc.cpp`, `callRfunc.R` et `calling.pdf`.

Références

- `http://www.biostatisticien.eu/fr/logiciels/interface-rc.html`

Outline

Appeler un prog Fortran77 depuis R : `reglinF.f`

Tout ce qu'on a vu avec le langage C est aussi possible avec le Fortran77.

Je ne le conseille pas car il est moins pratique à programmer que le C. Il y a tout de même 3 avantages à l'utiliser :

- C'est aussi rapide que le C.
- Certains profs du département l'utilisent.
- La librairie NAG contient beaucoup de fonctions statistiques.

Références

- http://www.dms.umontreal.ca/~stat/Logiciels/NAG/index.html
- http://www.idris.fr/data/cours/lang/fortran/f90/F77.html

Outline

Le calcul parallèle

La machine **simulation9** du DMS contient 32 processeurs. Il est possible d'utiliser R sur cette machine avec les packages R `snow`, `rsprng` et `rmpi`. Voilà comment modifier le programme précédent pour effectuer les calculs en parallèle.

Appeler un prog C/C++ depuis R et utiliser un cluster : reglinRCcl.R

```
masimul4 <- function(n=100,M=10000,beta0=3,
            alpha=0.05,xvec=1:n,nbclus=6) {
require(snow);require(rsprng);require(Rmpi)
cl <- makeCluster(nbclus, type = "MPI")
clusterSetupSPRNG(cl)
myfunc <- function(M) {
 dyn.load("reglinC.so")
 res <- 0
 out <- .C("Cfonc",as.integer(n),as.integer(M),
            as.double(beta0),as.double(alpha),
            as.double(xvec),val=as.double(res))
```

Appeler un prog C/C++ depuis R et utiliser un cluster : reglinRCcl.R

```
 dyn.unload("reglinC.so")
      }

out <- clusterCall(cl, myfunc, round(M/nbclus))
stopCluster(cl)
decision <- 0
for (clus in 1:nbclus) {
 decision <- decision + out[[clus]]$val
}
return(decision/(round(M/nbclus)*nbclus))
}
```

Appeler un prog C/C++ depuis R et utiliser un cluster : `reglinRCcl.R`

Faire un ssh sur **simulation9**.

```
bash
export PATH=/local/mpich2/bin/:$PATH
mpd -d
/local/R-2.9.2/bin/R
source("reglinRCcl.R")
system.time(masimul4(M=10000000,nbclus=30))
source("reglinRC.R")
system.time(masimul1(M=10000000))
```

Gain de temps : 221.996/22.274 = 10 fois

Appeler un prog C/C++ depuis R et utiliser un cluster : `reglinRCcl.R`

Sur ma machine :

```
> system.time(masimul1(M=10000000))
utilisateur     système      écoulé
     30.330       0.009      30.359
> system.time(masimul4(M=10000000,nbclus=6))
utilisateur     système      écoulé
      9.699       1.999      12.144
```

Gain de temps : 30.359/12.144 = 2.5 fois plus rapide

Références

- http://www.sfu.ca/~sblay/R/snow.html

Outline

Le calcul sur carte graphique

Les nouvelles machines dans le labo math du DMS (au 5ème) seront très bientôt équipées de cartes graphiques NVIDIA GeForce GTX 480 très performantes contenant chacune 480 processeurs. Il est possible d'utiliser le package R `gputools` pour faire des simulations en parallèle sur l'une de ces cartes graphiques.

Appeler un prog R qui utilise le GPU : `reglinRgpu.R`

```
Rfonc2 <- function(n=100,M=10000,beta0=3,
                   alpha=0.05,xvec=1:n) {
require(gputools)
    d <- 0
    seuil <- qt(1-alpha/2,n-2)
    for (m in 1:M) {
      yvec <- beta0 + runif(n)
      tobs <- summary(gpuLm(yvec~xvec))[4]$c[2,3]
      if (abs(tobs)>seuil) d <- d+1
    }
  return(d/M)
  }
```

Appeler un prog R qui utilise le GPU : `reglinRgpu.R`

```
source("reglinR.R")
system.time(Rfonc())
utilisateur      système       écoulé
     21.120        0.000       21.123
source("reglinRgpu.R")
system.time(Rfonc2())
system.time(Rfonc2())
utilisateur      système       écoulé
     77.944        0.000       77.956
```

Gain de temps médiocre pour cette fonction (à cause des temps d'accès) : 77.956/21.123 = 3.6 fois plus lent.

Appeler un prog R qui utilise le GPU : `reglinRgpu.R`

Mais essayez ceci :

```
require(gputools)
A <- matrix(runif(40000),nrow=200,ncol=200)
B <- matrix(runif(40000),nrow=200,ncol=200)
system.time(gpuCor(A, B, method="kendall"))
utilisateur     système      écoulé
      0.593       0.048       0.641
system.time(cor(A, B, method="kendall"))
utilisateur     système      écoulé
     29.772       0.000      29.780
```

Gain de temps : 29.780/0.641 = 46 fois plus rapide.

Références

- http://cran.r-project.org/web/packages/gputools/gputools.pdf
- http://developer.nvidia.com/object/cuda_training.html

Outils, création de packages

J'utilise l'éditeur de texte Emacs. Il facilite l'indentation et la coloration syntaxique. En passant, pensez à bien documenter vos codes !

Il est possible assez facilement de créer un vrai package R qui contiendra vos codes R et/ou C (ou Fortran77), des éventuels jeux de données, avec des fichiers d'aide sur chacune des fonctions du package.

Références

- Mon livre, chapitre 7.
- `http://www.biostatisticien.eu/fr/logiciels/interface-rc.html`