Parallel Computing for Data Science

1 – Good practices for coding

Jairo Cugliari

Master Informatique
Parcours Data Mining

Université Lumière Lyon 2
Université de Lyon

Quelques bonnes pratiques pour le calcul scientifique

Best Practices for Scientific Computing[1]

1. Write programs for people, not computers
2. Automate repetitive tasks
3. Use the computer to record history
4. Make incremental changes
5. Use version control
6. Don't repeat yourself (or others)
8. Plan for mistakes
   - Defensive programming
   - Write and run tests
   - Use a variety of oracles
   - Turn bugs into test cases
   - Use a symbolic debugger
9. Optimize software only after it works correctly
10. Document design and purpose, not mechanics
11. Collaborate

[1] G. Wilson et al. (2013) Download arXiv preprint

Quelques conseils

- Utiliser une IDE avec :
  - coloration syntactique
  - complétion automatique du code
- suivre les recommandations de style du langage (e.g. celles de Google ou Rstudio)
  - rend la lecture du code plus simple
  - permet de détecter plus facilement les erreurs
- mettre en place un système de gestion de versions
- en-tête de fichier descriptive du contenu
- penser au noms de variables et fonctions

Exemple : find runs of consecutive 1s in 0-1 vectors

The problem

- Input vector : $(1, 0, 0, 1, 1, 1, 0, 1, 1)$
- Function `findruns(vector, k)` should return the indexes of runs larger than k
- E.g. `findruns((1,0,0,1,1,1,0,1,1), 2)` should return (4, 5, 8)

(Very ugly) code

```
joe=function(x,k){
n=length(x)
r=NULL
for(i in 1:(n-k)) if(all(x[i:i+k-1]==1)) r<-c(r,i)
r
}
```

Make your code readable

Meaningful names, comments, correct code style & syntax highlight

```
## v1. Serial code ####
findruns <- function(x,k) {
  n <- length(x)
  runs <- NULL
  for (i in 1:(n - k)) {
    if (all(x[i:i + k - 1] == 1)) runs <- c(runs, i)
  }
  return(runs)
}
```

Précision numérique

Représentation de nombres entiers

Definition (Representation en base $\gamma \in \mathbb{N}$)

Soit $k \in \mathbb{N}$, alors

$$k = \sum_{j=0}^{m} a_j \gamma^j$$

pour un entier m unique (e.g. si $a_m \neq 0$) et une suite unique $a_0, a_1, \ldots$ d'entiers.

Example (193 peut être representé avec $m = 2$ en base 10)

$$193 = 1 \times 100 + 9 \times 10 + 3 \times 1$$

Example (et avec $m = 7$ en base 2)

$$193 = 1 \times 2^7 + 1 \times 2^6 + 1 \times 2^0$$

Représentation exacte de 193 avec l'octet 11000001 (approx. avec moins bits)

Recap: Approximation = limite à la précision

- transistor: 2 états (0: éteint; 1: allumé) => bit => byte = 8 bits (octet)
- Nb de transistors détermine la manière de représenter les nombres et la quantité (Combien d'entiers peut-on représenter avec un octet ?)

Sur R, la constante `.Machine$integer.max` contient le plus grand entier.

```
> .Machine$integer.max
[1] 2147483647
> .Machine$integer.max + 1
[1] 2147483648
> .Machine$integer.max + 1L
[1] NA
Warning message:
In .Machine$integer.max + 1L :
NA produit par débordement d'entier par le haut
```

Représentation de nombres réels

- Notation scientifique binaire (mantisse/exposant) avec $b_i = 0, 1$ ($b_0 = 1$ si $x \neq 0$):

$$x = b_0.b_1 b_2 \cdots \times 2^m$$

- En pratique on a un développement tronqué, e.g. en double précision (8o = 64b):
  - 1 bit pour le signe (+/-)
  - 52b pour la partie $b_1 b_2 \ldots b_5 2$ mantisse (b_0 dépend de m)
  - 11b pour l'exposant
- Epsilon de la machine: la plus petite quantité εtel que $1 + \varepsilon$ est différent de x : $\varepsilon = 2^{-52} \approx 2.220446 \times 10^{-16}$

```
> (1 + 2^-52) - 1
[1] 2.220446e-16
> (1 + 2^-53) - 1
[1] 0
```

Erreurs d'arrondi

- R est précis à 16 chiffres, mais pas plus

```
> 1.000000000000000123456 - 1
[1] 2.220446e-16
```

- Le système à virgule flottante a ses limites

```
> (1:10)/10 - (0:9)/10
[1] 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1
> (1:10)/10 - (0:9)/10 == 1/10
[1]  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE
[8] FALSE FALSE FALSE
> unique((1:10)/10 - (0:9)/10)
```

Le problème principale vient de l'annulation (différence de nombres à virgule flottantes)

Exemple

On veut calculer $sin(x) - x$. En utilisant le développement de Taylor ($\sin(x) = \sum_{n=0}^{\infty}(-1)^n x^{2n+1}/(2n+1)!$) on obtient

$$\sin(x) - x \approx -\frac{x^3}{6}\left(1 - \frac{x^2}{20}\right)$$

x_in_2	x_in_10	$\sin(x) - x$	"true"	erreur_rel
2^{-10}	9.765625e-04	-1.552204e-10	-1.552204e-10	4.656609e-11
2^{-20}	9.536743e-07	-1.445250e-19	-1.445603e-19	2.441406e-04
2^{-30}	9.313226e-10	0.000000e+00	-1.346323e-28	1.000000e+00
2^{-40}	9.094947e-13	0.000000e+00	-1.253861e-37	1.000000e+00

(Pseudo-)Aléatoire sur la machine

Simulation d'un échantillon iid uniforme

On veut créer une suite $x_1, x_2, \ldots, x_n$ issue d'une variable aléatoire uniforme sur l'intervalle $[0, 1]$

Definition (Générateurs congruentiels linéaires)

Soit $X_0 \in \{0, \ldots, m-1\}$ et deux nombres A et B, on définit la suite $X_n \in \{0, \ldots, m-1\}, n \in \mathbb{N}$, par

$$X_n = (AX_n + B) \mod m.$$

Alors, nous posons $U_n = X_n/m$ pour obtenir $U_n \in [0, 1), n \in \mathbb{N}$.

- Si l'on choisit bien m, A et B, alors la suite U_n a de propriétés proches à celles d'un suite aléatoire.
- La valeur X_0 est appelée la graine de la suite. Elle permet de reproduire le (pseudo-)aléatoire.

Génération (pseudo-)aléatoire sur R

- Plusieurs générateurs disponibles : Mersenne-Twister (MT) par défaut
- MT n'est pas un générateur linéaire congruentiel mais a besoin d'une graine

```
> runif(1)
[1] 0.8983897
> set.seed(2020)
> runif(1)
[1] 0.6469028
> set.seed(2020)
> runif(1)
[1] 0.6469028
```