Compte-rendu du TP
“Evaluation empirique d’un programme”

Castex Adrien Polytech Informatique et Master Recherche IA
Benoit Vuillemin Polytech Informatique et Master Recherche IA

21 janvier 2016

Table des matières

1 Présentation sommaire du programme à évaluer

1.1 Données à fournir en entrée

1.2 Données fournies en sortie

1.3 Algorithme

— n représente le nombre de mots mesurés dans le fichier d'entrée.
— m représente le nombre de mots que l'on souhaite avoir en sortie.
— k représente le nombre de mots à passer +1 dans le texte d'entrée.

Algorithm 1 <your caption for this algorithm>

```
if i ≥ maxval then
    i ← 0
else
    if i + k ≤ maxval then
        i ← i + k
    end if
end if
```

2 Premières mesures du temps d'exécution

2.1 Paramètres utilisés dans toute cette section

2.2 Mise en évidence de la stochasticité de l'algorithme

2.3 Influence de la charge système

2.4 Influence des options de compilation

3 Opération dominante

3.1 Paramètres utilisés dans toute cette section

3.2 Identification de l'opération dominante

Pour déterminer l'opération dominante au sein du programme, nous avons utilisé gprof.

```
$ gcc -pg -o markovPG markov.c
$ ./markovPG.exe 3 10000 < ./don-quixote.txt > outPG.txt
$ gprof markovPG.exe gmon.out > pg2.out
```

% time	cumulative seconds	self seconds	calls	self ns/call	total ns/call	name
53.57	0.15	0.15	7746432	19.36	19.36	wordncmp
17.86	0.20	0.05				sortcmp
14.29	0.24	0.04				__fentry__
14.29	0.28	0.04				_mcount_private
0.00	0.28	0.00	30000	0.00	0.00	skip
0.00	0.28	0.00	10000	0.00	0.00	writeword

Nous pouvons voir que la fonction *wordncmp* est la plus appellée avec 7746432 appels.

3.3 Comptage des appels à l'opération dominante

Nous avons ajouté, de façon globale, un tableau de 3 cases de long int. Nous incrémentons ensuite à chaque appel la case du tableau correspondante.

```
/* [...] */
up = nword;
while(lo+1 != up)
{
  mid = (lo + up) / 2;
  nbCall[0]++;
  if(wordncmp(word[mid], phrase) < 0)
    lo = mid;
  else
    up = mid;
}
/* [...] */
```

```
/* [...] */
nbCall[1]++;
for(i = 0; wordncmp(phrase, word[up+i]) == 0; i++)
{
  if(rand() % (i+1) == 0)
    p = word[up+i];
  nbCall[1]++;
}
/* [...] */
```

Il ne faut pas oublier de mettre une incrémentation avant ou après la boucle for car celle-ci va faire x tours de boucle, impliquant d'avoir x fois la condition qui vaut *vrai* et 1 fois qui vaut $faux$. Nous avons donc $x + 1$ appels à *wordncmp*.

```
/* [...] */
/* called by system qsort */
int sortcmp(const void* p, const void* q)
{
  char** p1 = (char**)p;
  char** q1 = (char**)q;
```

```
7   nbCall[2]++;
8   return wordncmp(*p1, *q1);
9 }
10 /* [...] */
```

4 Analyse de l'expérience fournie par R. Thion

4.1 Plan d'expérience

4.1.1 Variables mesurées

4.1.2 Paramètres

20 tests par jeu de paramètre. m 100 1000000

Table 1 – .

Paramètre	Valeur min	Valeur max	Incrément
m	100	1000000	$*10$
k	2	7	$+1$

Table 2 – Valeurs de n choisies.

Fichier	Nombre de mots		
m	100	1000000	$*10$
k	2	7	$+1$

4.1.3 Mode de combinaison des valeurs

4.1.4 Nombre de runs

4200

4.1.5 Environnement de test

Table 3 – Informations sur la machine utilisée.

OS	4.2.0-23-generic #28-Ubuntu SMP x86_64 GNU/Linux (Ubuntu 15.10)
Processeur	Intel(R) Core(TM) i7-5600U CPU @ 2.60GHz (dual-core)
Memoire	8GB 1600MHz DDR3L
Disque	180GB M2 SATA-3 Solid State Drive
Compilateur	g++ (Ubuntu 5.2.1-22ubuntu2) 5.2.1 20151010

4.2 Analyse des résultats

En regardant les résultats obtenus sur les jeux de paramètres identiques que les résultats obtenus sont les mêmes. Cela peut s'expliquer si les tests ont été réalisés avec une graine aléatoire ($srand(...)$) en seconde telle que $time(0)$. En effet, cette dernière fonction retourne le temps en seconde, ce qui fourni à *srand* la même valeur tant que les tests sont réalisés la même seconde. Étant donné que l'exécution du programme a nécessité peu de temps et que ceux-ci ont été enchainés en moins d'une seconde, cela explique ce résultat. Par conséquent, il est préférable d'utiliser une graine qui dépend du nombre de cycles du processeur et non du temps lorsque l'on effectue de nombreux tests à la suite. Nous pouvons donc conclure qu'environ 210 résultats sur 4200 sont réellement intéressant, les autres étant des doublons.

5 Conception et réalisation de votre expérience

5.1 Plan d'expérience

5.2 Analyse des résultats

6 Annexe : Code réalisé pour le tutoriel sur les tests d'hypothèse

```
########################
# Cas ou H0 est vraie
########################

# On va considerer deux populations qui ont exactement
# les memes parametres : les deux sont uniformement
# distribuees entre 0 et 1. Les deux populations ont
# donc la meme moyenne, en l'occurrence 0.5.
# Puis nous allons faire une experience :
# - tirer un petit echantillon dans chaque population
# - calculer la moyenne observee dans l'echantillon 1,
#   puis celle observee dans l'echantillon 2. On
#   n'obtiendra pas exactement 0.5 ni pour l'une ni pour
#   l'autre.
# - calculer la difference entre les deux moyennes
#   d'echantillons. Appelons d cette difference.

tirerEchantillonPop1H0 <- function(n) {
  vraieMoyenne <- 0.5
  plageDeDispersion <- 1.0
  min <- vraieMoyenne - plageDeDispersion/2.0
  max <- vraieMoyenne + plageDeDispersion/2.0
  ech <- runif(n, min, max)
  return(ech)
}

tirerEchantillonPop2H0 <- function(n) {
  vraieMoyenne <- 0.5
  plageDeDispersion <- 1.0
  min <- vraieMoyenne - plageDeDispersion/2.0
  max <- vraieMoyenne + plageDeDispersion/2.0
  ech <- runif(n, min, max)
  return(ech)
}

# Ecrire ici le code permettant de faire une "experience"
# c'est a dire tirer un echantillon de taille 20 dans
# chaque population et mesurer la difference d observee
# entre les 2 moyennes d'echantillon.
pop1 <- tirerEchantillonPop1H0(20)
pop2 <- tirerEchantillonPop2H0(20)

mean(pop1)
mean(pop2)
abs(mean(pop1) - mean(pop2))
abs(mean(tirerEchantillonPop1H0(20)) - mean(tirerEchantillonPop2H0(20)))

# Si 1000 etudiants font independamment cette experience,
# chaque etudiant aura des echantillons differents et donc
# chacun aura une valeur differente pour d. Si nous
# mettons toutes ces 1000 valeurs ensemble dans un vecteur
# (appele diffMoyH0), quelle sera la moyenne de ce vecteur ?

diffMoyH0 = c()

for(i in  1:1000)
{
  diffMoyH0 = c(diffMoyH0, abs(mean(tirerEchantillonPop1H0(20)) - mean(tirerEchantillonPop2H0
      (20))))
}

# On affiche la moyenne totale.
mean(diffMoyH0)

# On affiche les 1000 valeurs sur un histogramme.
plot(diffMoyH0)
abline(h=mean(diffMoyH0), col="red")
```

On obtient une moyenne de 0,07.

```

```

```
########################
# Cas ou H1 est vraie
########################

# On va voir comment se comporte notre indicateur quand
# les deux populations n'ont pas la meme moyenne
# (mais ont tout de meme la meme dispersion)
# eg . pop1 comprise entre 0.0 et 1.0
#   et pop2 comprise entre 0.2 et 1.2

tirerEchantillonPop1H1 <- function(n) {
  vraieMoyenne <- 0.5
  plageDeDispersion <- 1.0
  min <- vraieMoyenne - plageDeDispersion/2.0
  max <- vraieMoyenne + plageDeDispersion/2.0
  ech <- runif(n, min, max)
  return(ech)
}

tirerEchantillonPop2H1 <- function(n) {
  vraieMoyenne <- 0.7
  plageDeDispersion <- 1.0
  min <- vraieMoyenne - plageDeDispersion/2.0
  max <- vraieMoyenne + plageDeDispersion/2.0
  ech <- runif(n, min, max)
  return(ech)
}


# Ecrire ici le code pour faire les 1000 experiences
# comme precedemment, en nommant cette fois le vecteur
# diffMoyH1. Calculez sa moyenne et tracez son histogramme.

diffMoyH1 = c()

for(i in  1:1000)
{
  diffMoyH1 = c(diffMoyH1, abs(mean(tirerEchantillonPop1H1(20)) - mean(tirerEchantillonPop2H1
      (20))))
}

# On affiche la moyenne totale.
mean(diffMoyH1)

# On affiche les 1000 valeurs sur un histogramme.
plot(diffMoyH1)
abline(h=mean(diffMoyH1), col="red")
```

On obtient une moyenne de 0,2.
Placons nous maintenant dans le cas où l'on ait fait une seule experience.

```
cMean <- function(v1, v2)
{
  return(abs(mean(v1) - mean(v2)))
}
sameMean <- function(v1, v2)
{
  return(cMean(v1, v2) < 0.1)
}

sameMean(tirerEchantillonPop1H0(20), tirerEchantillonPop2H0(20))
sameMean(tirerEchantillonPop1H1(20), tirerEchantillonPop2H1(20))

x1 <- c(0.97050525, 0.13734516, 0.80793033, 0.05207726, 0.62629180, 0.93485856,
0.58220744, 0.65935145, 0.76467195, 0.73512414, 0.45139560, 0.93225380,
0.53790595, 0.99845675, 0.31035081, 0.43082815, 0.15475353, 0.42647652,
0.65676067, 0.74186048)
x2 <- c(0.33565036, 0.28830545, 0.51556544, 0.93223089, 0.29192576, 0.43505823,
0.63127002, 0.86082799, 0.56533392, 0.19083212, 0.13087779, 0.09849703,
0.98921291, 0.91480756, 0.78556552, 0.33859160, 0.88482223, 0.76701274,
0.24190609, 0.46251866)

cMean(x1, x2)
sameMean(x1, x2)
```

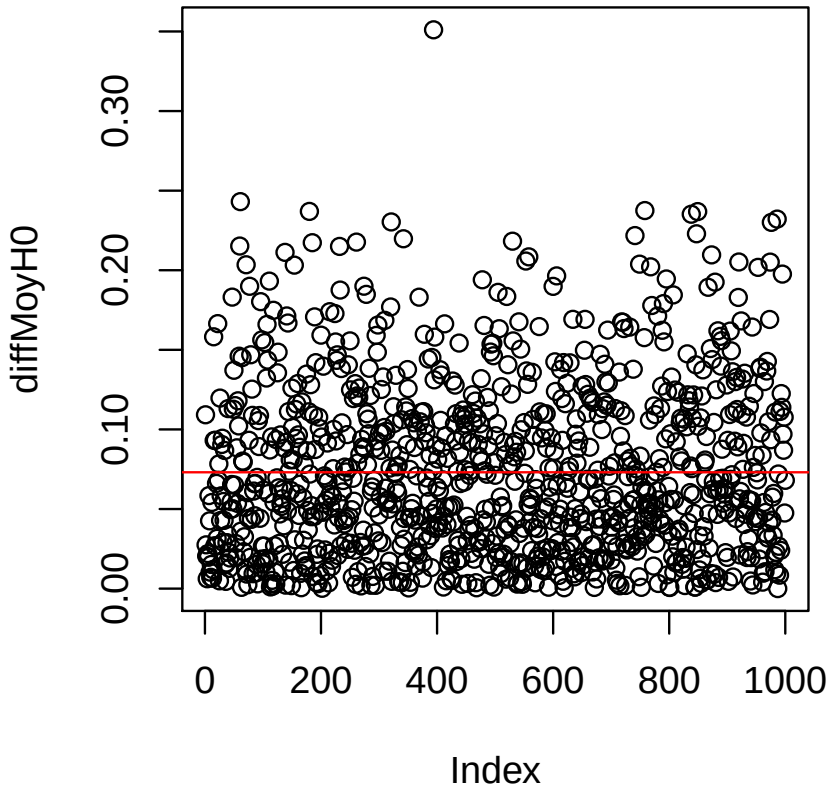

FIGURE 1 – Graphique représentant diffMoyH0 et diffMoyH1 ainsi que leur moyenne.

Ici, sameMean(x1, x2) retourne "TRUE", ce qui signifie que les deux échantillons ont une moyenne proche l'une de l'autre.

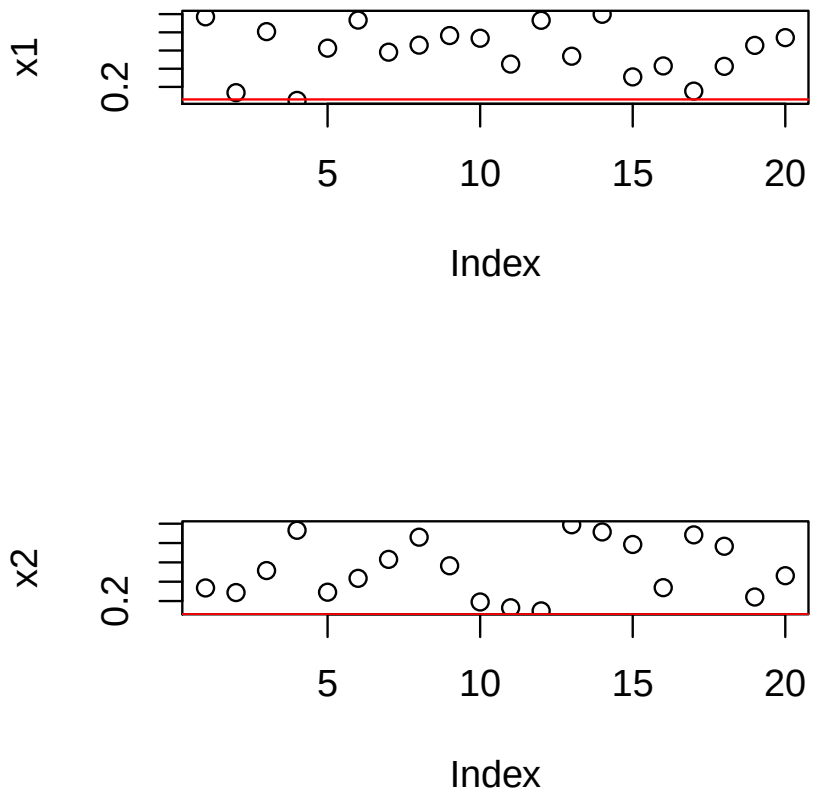

FIGURE 2 – Graphique représentant x1 and x2 ainsi que leur moyenne.

```
x1 <- c(0.41236444, 0.28422821, 0.15093798, 0.05885328, 0.25514435, 0.63026931,
0.56325462, 0.76304859, 0.56523993, 0.92535660, 0.10898729, 0.51579642,
0.07223967, 0.53483839, 0.52516575, 0.20250815, 0.89634680, 0.53879059,
0.58736912, 0.53945749)
x2 <- c(1.0809923, 0.5772333, 0.7252340, 1.1529082, 1.0924642, 0.6046166, 0.9495800,
0.3019857, 0.7701195, 1.0746508, 0.3928894, 1.0885017, 0.5101510, 1.1871599,
0.7953318, 0.5711237, 0.5505642, 0.9854085, 0.5105643, 0.9601635)

cMean(x1, x2)
sameMean(x1, x2)
```

Ici, sameMean(x1, x2) retourne "FALSE", ce qui signifie que les deux échantillons ont une moyenne bien différente l'une de l'autre.

Incorporez également les histogrammes obtenus sous forme de figures. Indiquez vos conclusions sur les deux paires d'échantillons.

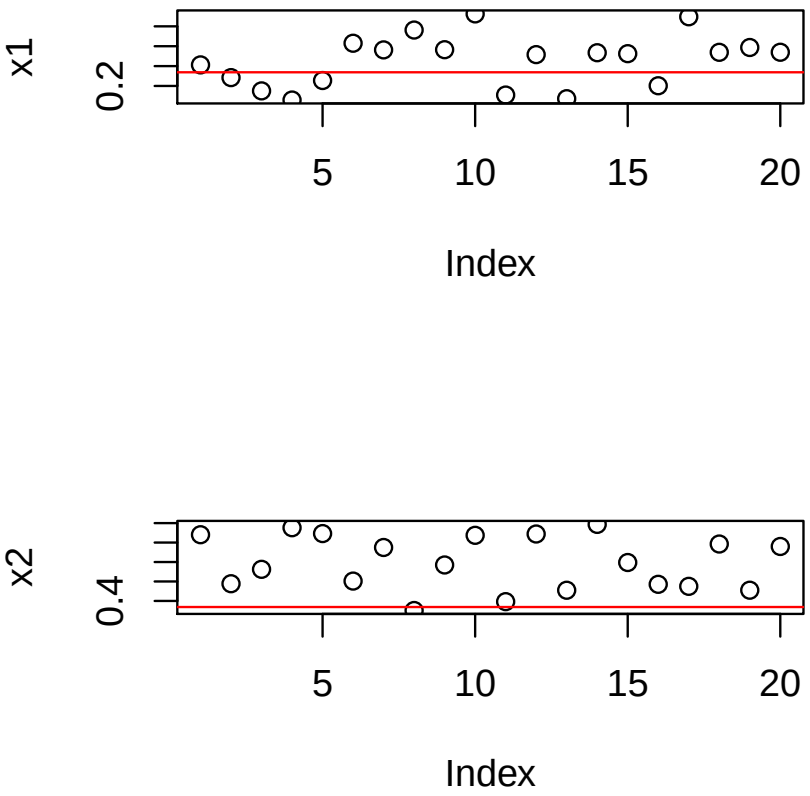

FIGURE 3 – Graphique représentant x1 and x2 ainsi que leur moyenne.