Analyse des algorithmes de Maximum Subarray 2D
M2 Data Science Algorithmique

Taqwa Ben Romdhane, Manal Derghal, Khalil Ounis

Mercredi 9 avril 2025

Table des matières

---

1 Description du problème et objectif

1.1 Présentation du problème

Dans ce document, nous étudions le problème du sous-tableau de somme maximale du point de vue de la complexité algorithmique. Ce problème est intéressant car il existe de nombreux algorithmes pour le résoudre, dont la complexité varie considérablement.

Dans ce rapport, nous abordons uniquement deux algorithmes :

un algorithme naïf utilisant le paradigme de Brute Force,

et un algorithme optimal basé sur la méthode de Kadane.

Nous étendons ensuite le problème à deux dimensions afin de déterminer la sous-matrice contiguë de somme maximale dans une matrice. Bien que ce cas soit plus complexe, la méthode de Kadane peut également être adaptée au cas 2D.

Comme pour le cas 1D, nous proposerons deux solutions pour résoudre le problème de la sous-matrice de somme maximale.

1.2 Objectifs :

Dans le cadre de ce projet, nous devons répondre aux objectifs suivants :

- Evaluer la performance en temps d'exécution des solutions naive et optimale
- Implémenter ces algorithmes en R et en C++ et évaluer le gain de temps
- Valider la complexité théorique attenue par une régression linéaire

2 Maximum Sum Submatrix

Étant donné un tableau 2D, la tâche consiste à trouver la sous-matrice de somme maximale qu'il contient.

Imaginez que vous avez cette matrice 4x5 :

Le rectangle en vert présente la sous-matrice avec la somme maximale égale à 29.

1	2	-1	-4	-20
-8	-3	4	2	1
3	8	10	1	3
-4	-1	1	7	-6

2.1 Approche naïve

Nous explorons tous les rectangles possibles dans le tableau 2D donné, en utilisant quatre variables : deux pour définir les limites gauche et droite (`left` & `right`), et deux autres pour définir les limites supérieure et inférieure (`up` & `down`). Nous calculons leurs sommes et gardons une trace de la somme maximale trouvée.

Complexité algorithmique :
$O\left((nm)^3\right)$
Nous parcourons toutes les limites du rectangle en $O((nm)^2)$, et pour chaque rectangle, nous calculons sa somme en $O(nm)$. Par conséquent, la complexité temporelle globale est $O((nm)^3)$, où n est le nombre de lignes et m le nombre de colonnes de la matrice.

Pour une matrice 3×3, il y a 36 sous-matrices. Si l'on étend cela à une matrice 100×100, on obtient des millions de possibilités. Il est donc clair que la méthode naïve de force brute ne fonctionne tout simplement pas en pratique.

2.2 Approche améliorée (Méthode de Kadane)

La nouvelle approche est basée sur l'algorithme de **Kadane**, qui est utilisé comme solution optimale dans le cas 1D.
Nous allons expliquer comment appliquer cette méthode à un tableau 2D.
L'idée est de **compresser la matrice 2D en une série de tableaux 1D** et de résoudre chacun d'eux à l'aide de l'algorithme de Kadane.

Complexité algorithmique :
La complexité temporelle de cette approche est de $O(n \cdot m^2)$, où n est le nombre de lignes et m le nombre de colonnes.
En effet, nous itérons sur toutes les paires de colonnes et, pour chaque paire, nous appliquons l'algorithme de Kadane, qui prend un temps de $O(n)$.
La complexité spatiale est de $O(m \cdot n)$ pour le stockage des sommes de préfixes.

2.3 Principe de l'algorithme

1. **Fixer les limites gauche et droite :**
   — Nous fixons les colonnes gauche (`left`) et droite (`right`) une par une.
2. **Calculer les sommes cumulatives :**
   — Pour chaque paire de colonnes `left` et `right`, nous calculons la somme des éléments de chaque ligne entre ces colonnes et stockons ces sommes dans un tableau auxiliaire `temp[]`.
3. **Appliquer l'algorithme de Kadane :**
   — En appliquant l'algorithme de Kadane sur le tableau `temp[]`, nous obtenons la somme maximale d'un sous-tableau de ce tableau, ce qui correspond à la somme maximale de la sous-matrice avec les limites de colonnes `left` et `right`.
4. **Déterminer la sous-matrice de somme maximale :**
   — Pour obtenir la somme maximale globale, nous prenons le maximum de toutes les sommes obtenues pour chaque paire de colonnes `left` et `right`.

2.3.1 Exemple

```
# Input :
mat <- matrix(c(
   1, -9, -10, 1,
  -1,  10,  10, 1,
   0,   9,   9, -9,
  -1,  -1,  -1, -1
), nrow = 4, byrow = TRUE)

# Output : 38
# Explication : (top, left) = (2,2), (down, right) = (3,3)
# Sous-matrice : [[10, 10], [9, 9]]
```

3 Un premier exemple

```
# installer le package
devtools::install_github("AMATERASU11/MaximumSubarray")
```

```
library(MaximumSubarray)
```

```
n <- 4   # nombre de lignes
m <- 5   # nombre de colonnes

# Générer la matrice avec des valeurs aléatoires entre -100 et 100
set.seed(123)
mat <- matrix(sample(-100:100, n * m, replace = TRUE), nrow = n, ncol = m)
```

```
mat
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]   58   69  -87  -10   -9
## [2,]   78  -51   17   96   36
## [3,]  -87   17   52  -10   -2
## [4,]   94  -58  -11   84  -29
```

```
max_subarray_rectangle_naive(mat)
```

```
## $sum
## [1] 251
```

```
##
## $submatrix
##      [,1] [,2] [,3] [,4]
## [1,]   58   69  -87  -10
## [2,]   78  -51   17   96
## [3,]  -87   17   52  -10
## [4,]   94  -58  -11   84
```

```
max_subarray_rectangle_naive_Rcpp(mat)
```

```
## $sum
## [1] 251
##
## $submatrix
##      [,1] [,2] [,3] [,4]
## [1,]   58   69  -87  -10
## [2,]   78  -51   17   96
## [3,]  -87   17   52  -10
## [4,]   94  -58  -11   84
```

```
max_subarray_rectangle_opt(mat)
```

```
## $sum
## [1] 251
##
## $submatrix
##      [,1] [,2] [,3] [,4]
## [1,]   58   69  -87  -10
## [2,]   78  -51   17   96
## [3,]  -87   17   52  -10
## [4,]   94  -58  -11   84
```

```
max_subarray_rectangle_opt_Rcpp(mat)
```

```
## $sum
## [1] 251
##
## $submatrix
##      [,1] [,2] [,3] [,4]
## [1,]   58   69  -87  -10
## [2,]   78  -51   17   96
## [3,]  -87   17   52  -10
## [4,]   94  -58  -11   84
```

---

4 Comparaison R avec C++

On va faire des comparaisons pour les deux types d'algorithme en R et C++ pour quantifier leur différence de performance.

La fonction `one.simu.time` retourne le temps recherché, et `one.simu` sera utilisé par `microbenchmark`, on retourne le temps en ms

sur une matrice de taille 30x50, on obtient les résultats suivants :

```r
library(microbenchmark)

# Simulation d'un test de performance sur une matrice aléatoire de taille n x m
one.simu.time_2D <- function(n, m, func) {

  mat <- matrix(sample(-100:100, n * m, replace = TRUE), nrow = n, ncol = m)

  if (func == "Naive2D") {
    t <- microbenchmark(max_subarray_rectangle_naive(mat), times = 1)$time / 1e6
  } else if (func == "Kadane2D") {
    t <- microbenchmark(max_subarray_rectangle_opt(mat), times = 1)$time / 1e6
  } else if (func == "Naive2D_cpp") {
    t <- microbenchmark(max_subarray_rectangle_naive_Rcpp(mat), times = 1)$time / 1e6
  } else if (func == "Kadane2D_cpp") {
    t <- microbenchmark(max_subarray_rectangle_opt_Rcpp(mat), times = 1)$time / 1e6
  } else {
    stop("fonction inconnue")
  }

  # Arrondi à 2 décimales
  return(round(t, 2))
}
```

4.1 Un essai

4.1.1 Temps d'exécution en R

Sur un exemple, on obtient :

```r
# Simulation sur une matrice de taille n*m
n <- 30
m <- 50

# Exécuter la simulation
res_naive <- one.simu.time_2D(n,m,"Naive2D")
res_kadane <- one.simu.time_2D(n,m,"Kadane2D")

# Afficher les résultats
cat("time_naive:", res_naive,"ms")
```

```
## time_naive: 943.79 ms
```

```r
cat("time_kadane:", res_kadane, "ms")
```

```
## time_kadane: 9.29 ms
```

4.1.2 Temps d'exécution en C++

sur une matrice de taille 30x50, on obtient les résultats suivants :

```r
# Simulation sur une matrice de taille n*m
n <- 30
m <- 50

res_naive_cpp <- one.simu.time_2D(n,m,"Naive2D_cpp")
res_Kadane_cpp <- one.simu.time_2D(n,m,"Kadane2D_cpp")
```

```r
# Afficher les résultats
cat("time_naive_cpp:" ,res_naive_cpp,"ms")
```

```
## time_naive_cpp: 90.31 ms
```

```r
cat("time_kadane_cpp:",res_Kadane_cpp, "ms")
```

```
## time_kadane_cpp: 0.75 ms
```

4.2 Simulations avec répétitions

On reproduit ces comparaisons de manière plus robuste : Nous calculons le gain de temps à partir de 10 répétitions de simulation

```r
# Simulation sur une matrice de taille n*m
n <- 30
m <- 50

nbSimus <- 10

time_naive <- rep(0, nbSimus); time_naive_cpp <- rep(0, nbSimus);
time_kadane <- rep(0, nbSimus); time_kadane_cpp <- rep(0, nbSimus)

for(i in 1:nbSimus){time_naive[i] <- one.simu.time_2D(n,m, func = "Naive2D")}
for(i in 1:nbSimus){time_naive_cpp[i] <- one.simu.time_2D(n,m, func = "Naive2D_cpp")}
for(i in 1:nbSimus){time_kadane[i] <- one.simu.time_2D(n,m, func = "Kadane2D")}
for(i in 1:nbSimus){time_kadane_cpp[i] <- one.simu.time_2D(n,m, func = "Kadane2D_cpp")}
```

4.2.1 Gain R versus C++

```r
naive_speedup_cpp <- mean(time_naive) / mean(time_naive_cpp)
kadane_speedup_cpp <- mean(time_kadane) / mean(time_kadane_cpp)
cat("le gain R vs cpp pour naif:", round(naive_speedup_cpp,2),"ms")
```

```
## le gain R vs cpp pour naif: 6.07 ms
```

```r
cat("le gain R vs cpp pour Kadane:", round(kadane_speedup_cpp,2),"ms")
```

```
## le gain R vs cpp pour Kadane: 11.26 ms
```

4.2.2 Gain Naif Versus Kadane en R et C++

```r
kadane_vs_naive_R <- mean(time_naive) / mean(time_kadane)
kadane_vs_naive_Rcpp <- mean(time_naive_cpp) / mean(time_kadane_cpp)
cat("le gain naif vs Kadane en R est:",round(kadane_vs_naive_R,2), "ms")
```

```
## le gain naif vs Kadane en R est: 60.68 ms
```

```r
cat("le gain cpp est:",round(kadane_vs_naive_Rcpp,2), "ms")
```

```
## le gain cpp est: 112.58 ms
```

On recommence avec `n = 80` et `m = 60` seulement pour le gain avec C++ pour Kadane

```r
n <- 80
m<-60
```

```r
nbSimus <- 10

time_kadane <- rep(0, nbSimus); time_kadane_cpp <- rep(0, nbSimus)
for(i in 1:nbSimus){time_kadane[i] <- one.simu.time_2D(n,m, func = "Kadane2D")}
for(i in 1:nbSimus){time_kadane_cpp[i] <- one.simu.time_2D(n,m, func = "Kadane2D_cpp")}
median_kadane_R_vs_Rcpp <- median(time_kadane) / median(time_kadane_cpp)
cat("le gain Kadane en R vs Kadane en C++ est:",round(median_kadane_R_vs_Rcpp,2), "ms")
```

```
## le gain Kadane en R vs Kadane en C++ est: 19.25 ms
```

Conclusion :

4.2.3 Performances C++ vs R :

— Naïf : C++ 6× plus rapide

— Kadane : C++ 11× plus rapide → 19× pour `(n,m)=(80,60)`

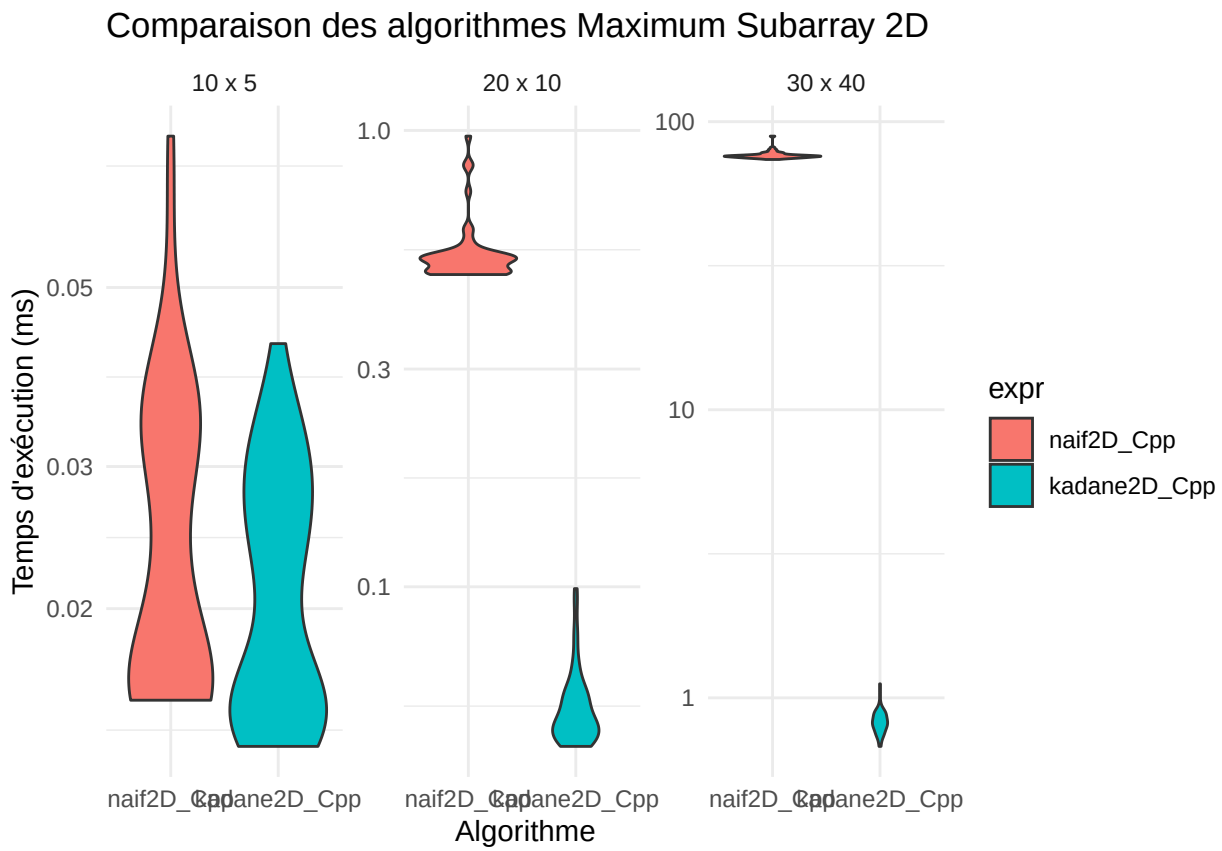
4.2.4 Efficacité algorithmique :

— Kadane 61× mieux que naïf en R

— Kadane 113× mieux que naïf en C++

4.3 Simulations avec `microbenchmark`

Vous avez besoin des packages `microbenchmark` et `ggplot2` pour exécuter les simulations et afficher les résultats (sous forme de diagrammes en violon). Nous comparons `naive_Rcpp` avec `opt_Rcpp` pour des tailles de données `(n,m) = (10,5)`, `(n,m) = (20,10)` et `(n,m) = (30,40)`

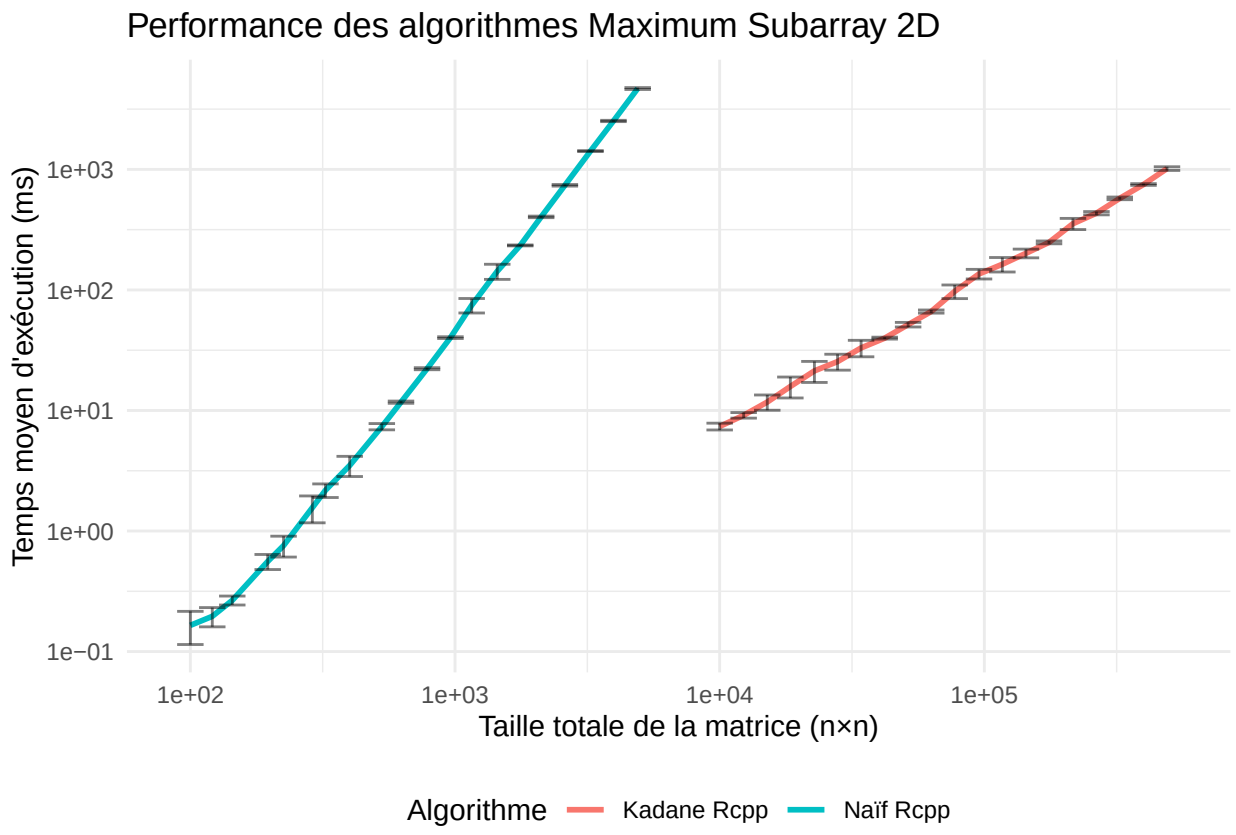



```
## # A tibble: 6 x 8
##    size expr         min_time q1_time median_time mean_time q3_time max_time
##   <dbl> <fct>           <dbl>   <dbl>       <dbl>     <dbl>   <dbl>    <dbl>
## 1    50 naif2D_Cpp     0.0154  0.0159      0.0217    0.0264  0.0342   0.0770
## 2    50 kadane2D_Cpp   0.0135  0.0150      0.0199    0.0220  0.0283   0.0426
## 3   200 naif2D_Cpp     0.483   0.490       0.521     0.544   0.533    0.972
## 4   200 kadane2D_Cpp   0.0447  0.0473      0.0507    0.0549  0.0578   0.0991
## 5  1200 naif2D_Cpp    73.9    75.3        76.0      76.5    76.7     89.1
## 6  1200 kadane2D_Cpp   0.679   0.782       0.823     0.825   0.874    1.12
```

---

5 Evaluation de la complexité

5.0.0.0.1 Complexité expériementale (naif) D'après la littérature, et comme expliquée précedemment, l'agorithme naif a une complexité polynomiale en O((nm)ˆ3), içi nous essayons d'exp




```
# Affichage des résultats
cat("Les résultats pour la solution naïve:")
```

```
## Les résultats pour la solution naïve:
```

```
res_Naive_2D
```

```
##     n  m mean_time     sd_time total_size
## 1  10 10     0.165  0.05060742        100
## 2  11 11     0.196  0.03565265        121
## 3  12 12     0.266  0.02270585        144
## 4  14 14     0.559  0.07978443        196
## 5  15 15     0.757  0.14892578        225
## 6  17 17     1.562  0.39233772        289
## 7  18 18     2.178  0.28141705        324
## 8  20 20     3.498  0.66676333        400
## 9  23 23     7.349  0.44016285        529
## 10 25 25    11.721  0.24843734        625
## 11 28 28    22.210  0.43230648        784
## 12 31 31    40.204  0.66367998        961
## 13 34 34    74.642 10.24031770       1156
## 14 38 38   142.807 20.48118977       1444
## 15 42 42   234.027  2.08537793       1764
## 16 46 46   403.600  6.09241970       2116
## 17 51 51   737.696 11.08930135       2601
## 18 57 57  1418.405 13.15217536       3249
## 19 63 63  2521.861 32.03512532       3969
```

```
## 20 70 70  4685.667 85.53927533       4900
```

```
cat("Les résultats pour la solution optimale:")
```

```
## Les résultats pour la solution optimale:
```

```
res_Kadane_2D
```

```
##      n   m mean_time    sd_time total_size
## 1  100 100     7.369  0.4771780      10000
## 2  111 111     9.124  0.4816915      12321
## 3  123 123    11.745  1.6973722      15129
## 4  136 136    15.820  3.1222072      18496
## 5  151 151    21.304  4.2273086      22801
## 6  167 167    25.440  3.8355414      27889
## 7  185 185    33.038  5.1589788      34225
## 8  205 205    39.837  0.7333947      42025
## 9  227 227    51.539  2.2519939      51529
## 10 251 251    66.198  1.9636802      63001
## 11 278 278    97.317 12.5969732      77284
## 12 309 309   135.725 12.4503942      95481
## 13 342 342   163.313 22.4270512     116964
## 14 379 379   201.343 16.7595638     143641
## 15 419 419   248.050  6.7056991     175561
## 16 465 465   354.498 37.6416438     216225
## 17 515 515   433.078 13.4269206     265225
## 18 570 570   575.083 14.5564503     324900
## 19 632 632   749.245 12.2654956     399424
## 20 700 700  1015.559 35.7560498     490000
```

On vérifie la valeur du coefficient directeur pour les deux méthodes :

```
##
## Call:
## lm(formula = log(res_Naive_2D$mean_time) ~ log(res_Naive_2D$n))
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.20284 -0.13428 -0.01958  0.07768  0.54047
##
## Coefficients:
##                     Estimate Std. Error t value Pr(>|t|)
## (Intercept)         -14.8923     0.2312  -64.42   <2e-16 ***
## log(res_Naive_2D$n)   5.4504     0.0695   78.42   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.1836 on 18 degrees of freedom
## Multiple R-squared:  0.9971, Adjusted R-squared:  0.9969
## F-statistic:  6150 on 1 and 18 DF,  p-value: < 2.2e-16
```

```
## Exposant estimé (naïf): 5.4504
```

```
##
## Call:
## lm(formula = log(res_Kadane_2D$mean_time) ~ log(res_Kadane_2D$n))
##
```

```
## Residuals:
##       Min        1Q    Median        3Q       Max
## -0.087289 -0.019932 -0.002617  0.034696  0.104749
##
## Coefficients:
##                       Estimate Std. Error t value Pr(>|t|)
## (Intercept)          -9.69907    0.11322  -85.67   <2e-16 ***
## log(res_Kadane_2D$n)  2.52993    0.02018  125.35   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.05326 on 18 degrees of freedom
## Multiple R-squared:  0.9989, Adjusted R-squared:  0.9988
## F-statistic: 1.571e+04 on 1 and 18 DF,  p-value: < 2.2e-16
```

```
## Exposant estimé (Kadane): 2.52993
```

Les coefficients directeurs trouvés sont bien ceux que l'on attendait. La valeur pour la méthode naïve 6 car $O\left((nm)^3\right) = O\left((n^2)^3\right)$ et 3 car $O(n \cdot m^2) = O(n \cdot n^2)$ pour l'algorithme de Kadane

---

6 Cas particulier des données presques triées

On considère des données triées avec 5% de valeurs échangées au hasard.

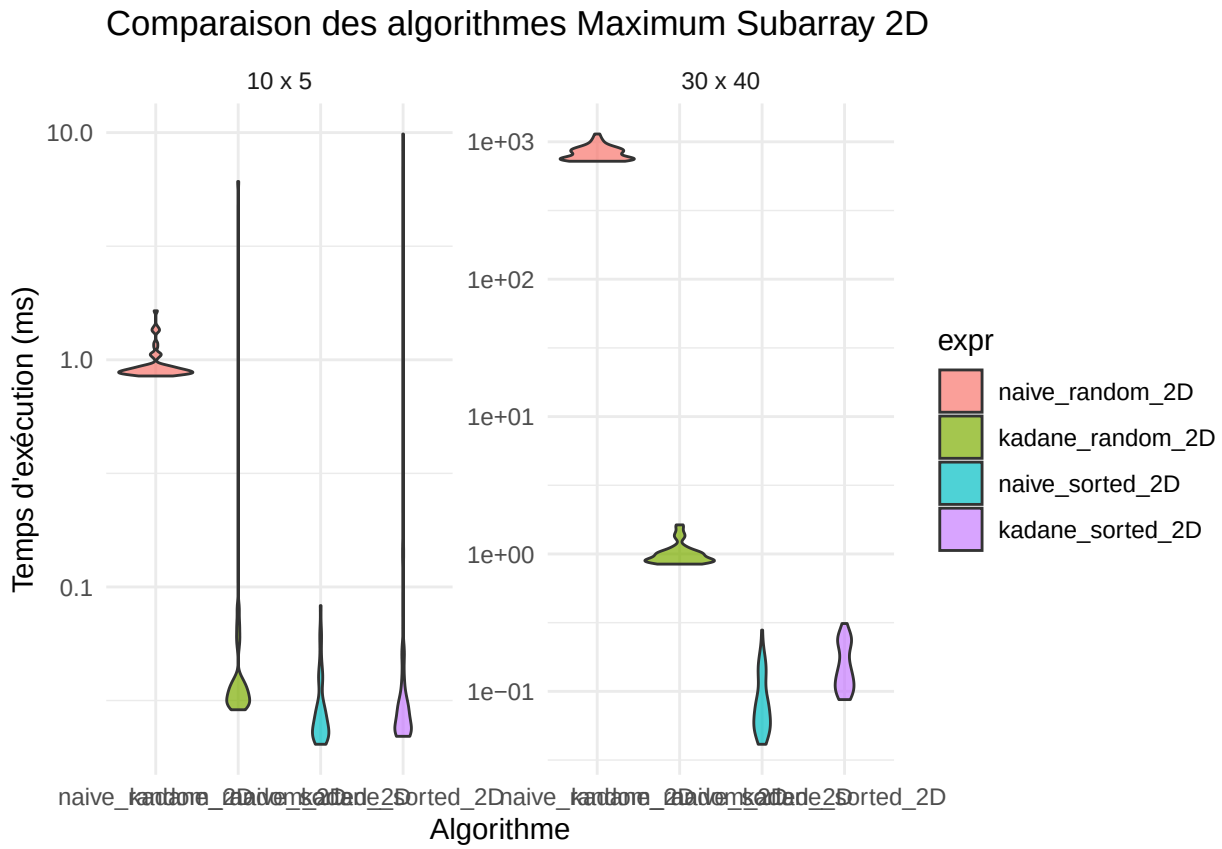
Sur un exemple cela donne :

```
# Création d'une matrice triée
n <- 10  # nombre de lignes
m <- 10  # nombre de colonnes
v <- 1:(n * m)
n_swap <- floor(0.05 * length(v))
swap_indices <- sample(length(v), n_swap)
v[swap_indices] <- sample(v[swap_indices])
mat <- matrix(v, nrow = n, ncol = m)
mat
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
##  [1,]    1   11   21   31   41   51   61   71   81    91
##  [2,]    2   12   22   32   42   52   62   72   82    92
##  [3,]    3   13   23   33   18   53   63   73   83    93
##  [4,]    4   14   24   34   44   54   64   74   84    94
##  [5,]    5   15   25   35   45   55   65   75   85    95
##  [6,]    6   16   26   36   46   43   66   76   86    96
##  [7,]    7   17   27   37   47   57   67   77   87    97
##  [8,]    8   56   28   38   48   58   68   78   88    98
##  [9,]    9   19   29   39   49   59   69   79   89    99
## [10,]   10   20   30   40   50   60   70   80   90   100
```

```
# Fonctions de simulation
one.simu <- function(n,m, func) {
  mat <- matrix(sample(-100:100, n * m, replace = TRUE), nrow = n, ncol = m)
  if (func == "naive_Rcpp") return(max_subarray_rectangle_naive(mat))
  if (func == "kadane_Rcpp") return(max_subarray_rectangle_opt_Rcpp(mat))
}
```

```
one.simu2 <- function(n,m, func) {
  v <- 1:(n * m)
  n_swap <- floor(0.05 * length(v))
  swap_indices <- sample(length(v), n_swap)
  v[swap_indices] <- sample(v[swap_indices])
  mat <- matrix(v, nrow = n, ncol = m)
  if (func == "naive_Rcpp") return(max_subarray_rectangle_naive(mat))
  if (func == "kadane_Rcpp") return(max_subarray_rectangle_opt_Rcpp(mat))
}
```




```
## # A tibble: 8 x 8
##    size expr             min_time q1_time median_time mean_time q3_time max_time
##   <dbl> <fct>               <dbl>   <dbl>       <dbl>     <dbl>   <dbl>    <dbl>
## 1    50 naive_random_2D    0.848  8.75e-1      0.899     0.958  9.37e-1  1.64e+0
## 2    50 kadane_random_2D   0.0288 3.10e-2      0.0341    0.160  3.85e-2  6.09e+0
## 3    50 naive_sorted_2D    0.0203 2.23e-2      0.0254    0.0302 3.03e-2  8.28e-2
## 4    50 kadane_sorted_2D   0.0220 2.33e-2      0.0269    0.230  3.19e-2  9.86e+0
## 5  1200 naive_random_2D  722.     7.45e+2    815.      831.     8.88e+2  1.14e+3
## 6  1200 kadane_random_2D   0.844  8.91e-1      0.956     1.02   1.04e+0  1.63e+0
## 7  1200 naive_sorted_2D    0.0412 5.51e-2      0.0752    0.0907 1.23e-1  2.79e-1
## 8  1200 kadane_sorted_2D   0.0871 1.08e-1      0.135     0.161  2.28e-1  3.12e-1
```

Pour `(n,m) = (10,5)`, le temps d'exécution est plus rapide que pour `(n,m) = (30,40)`. Kadane est toujours plus rapide que Naïf, avec un écart plus important à `(n,m) = (30,40)`. Lorsque les tableaux sont triés, Naïf et Kadane sont beaucoup plus rapides, avec un écart réduit entre les deux.