

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
1.1	Présentation du problème	1
1.2	Objectifs :	2
2	Maximum Sum Submatrix	2
2.1	Approche naïve	2
2.2	Approche améliorée (Méthode de Kadane)	2
2.3	Principe de l'algorithme	3
3	Un premier exemple	3
4	Comparaison R avec C++	4
4.1	Un essai	5
4.2	Simulations avec répétitions	6
4.3	Simulations avec <code>microbenchmark</code>	7
5	Evaluation de la complexité	8
6	Cas particulier des données presque triées	11
7	Cas particulier avec matrice symétrique	13

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 naïve et optimale
- Implémenter ces algorithmes en R et en C++ et évaluer le gain de temps
- Valider la complexité théorique attendue 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 :

$O((nm)^3)$

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 3x3, il y a 36 sous-matrices. Si l'on étend cela à une matrice 100x100, 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’algorithmes 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 :

```
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 :

```
# 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\n")
```

```
## time_naive: 995.95 ms
```

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

```
## time_kadane: 8.31 ms
```

4.1.2 Temps d'exécution en C++

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

```
# 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")
```

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

```
## time_naive_cpp: 87.15 ms
```

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

```
## time_kadane_cpp: 0.98 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

```
# 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++

```
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\n")
```

```
## le gain R vs cpp pour naif: 10.2 ms
```

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

```
## le gain R vs cpp pour Kadane: 10.8 ms
```

4.2.2 Gain Naif Versus Kadane en R et C++

```
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\n")
```

```
## le gain naif vs Kadane en R est: 100.55 ms
```

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

```
## le gain cpp est: 106.53 ms
```

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

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

```

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: 19.36 ms

Conclusion :

4.2.3 Performances C++ vs R :

- Naïf : C++ 10× plus rapide
- Kadane : C++ 11× plus rapide → 19× pour (n,m)=(80,60)

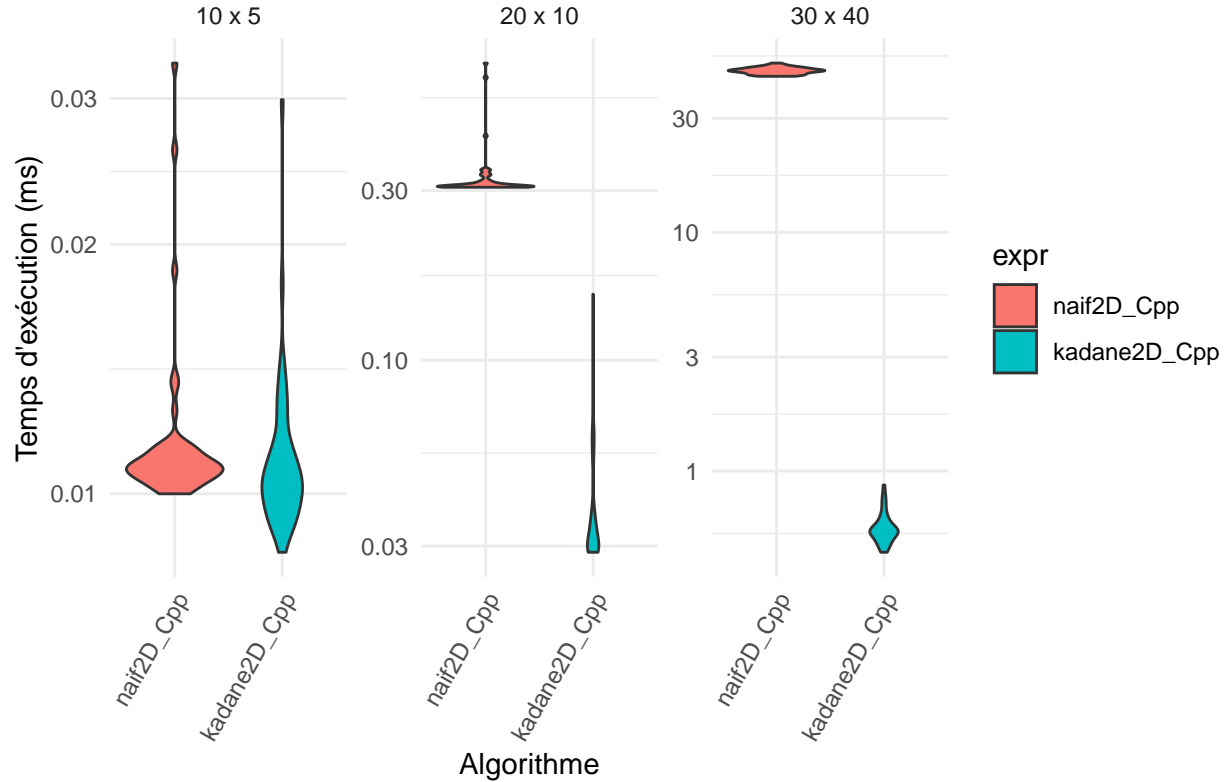
4.2.4 Efficacité algorithmique :

- Kadane 101× mieux que naïf en R
- Kadane 107× 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)

Comparaison des algorithmes Maximum Subarray 2D

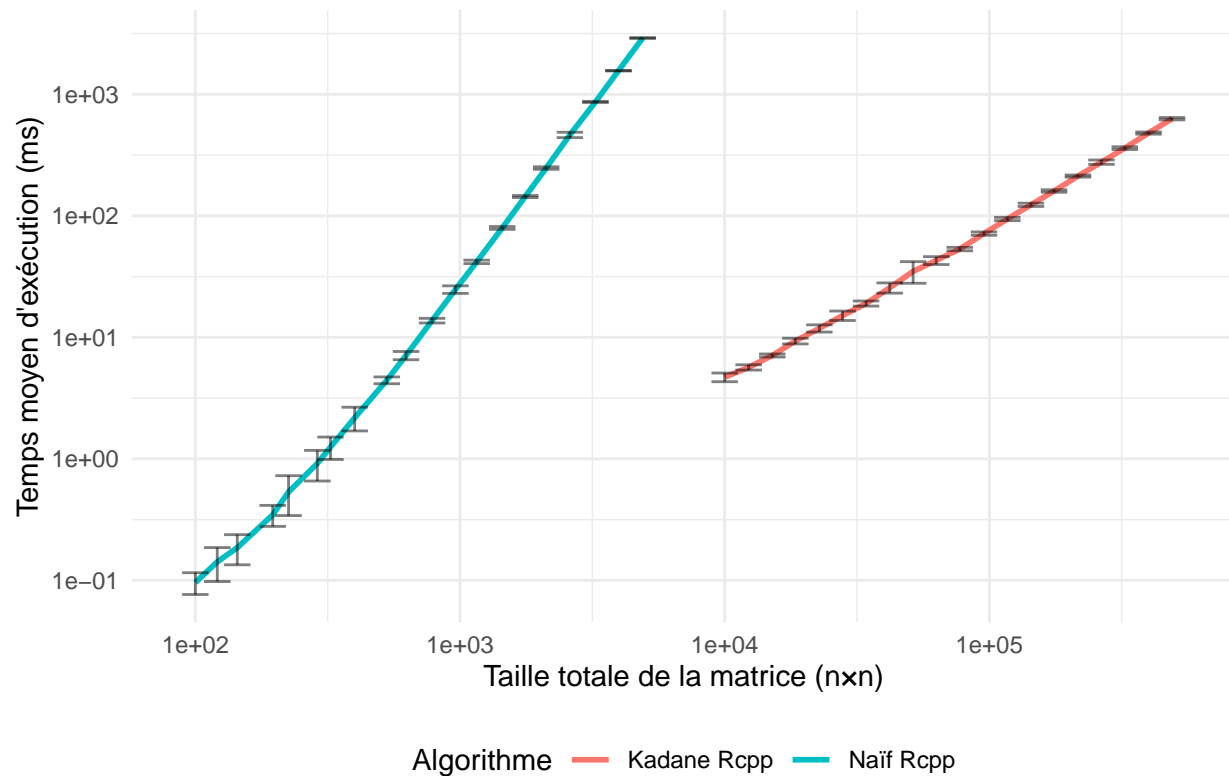


```
## # A tibble: 6 x 8
##   size expr      min_time q1_time median_time mean_time q3_time max_time
##   <dbl> <fct>      <dbl>   <dbl>      <dbl>     <dbl>   <dbl>   <dbl>
## 1    50 naif2D_Cpp    0.01    0.0105      0.0108    0.0118  0.0113  0.0331
## 2    50 kadane2D_Cpp  0.0085  0.00962    0.0104    0.0112  0.0114  0.0299
## 3   200 naif2D_Cpp    0.307   0.308      0.309     0.330   0.317   0.686
## 4   200 kadane2D_Cpp  0.0288  0.0296    0.0320    0.0397  0.0372  0.153
## 5  1200 naif2D_Cpp   45.0    46.4      47.5     47.5    48.6    51.1
## 6  1200 kadane2D_Cpp  0.457   0.523     0.558     0.579   0.609   0.874
```

5 Evaluation de la complexité

5.0.0.0.1 Complexité expérimentale (naif) D'après la littérature, et comme expliquée précédemment, l'algorithme naif a une complexité polynomiale en $O((nm)^3)$, ici nous essayons d'exp

Performance des algorithmes Maximum Subarray 2D



```
# 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	0.096	0.01955050	100
## 2	11	11	0.142	0.04417138	121
## 3	12	12	0.186	0.05168279	144
## 4	14	14	0.346	0.06801961	196
## 5	15	15	0.534	0.19265398	225
## 6	17	17	0.916	0.25872336	289
## 7	18	18	1.250	0.26115130	324
## 8	20	20	2.179	0.48098164	400
## 9	23	23	4.438	0.28549372	529
## 10	25	25	7.096	0.55349797	625
## 11	28	28	13.734	0.60415598	784
## 12	31	31	24.775	1.77162355	961
## 13	34	34	41.820	1.38223651	1156
## 14	38	38	79.585	2.10719535	1444
## 15	42	42	144.203	2.88942151	1764
## 16	46	46	247.426	5.94501509	2116
## 17	51	51	464.146	23.71130823	2601
## 18	57	57	866.553	9.57412723	3249
## 19	63	63	1564.906	7.49193967	3969

```
## 20 70 70 2907.218 25.18713199 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      4.699  0.3868520      10000
## 2  111 111      5.664  0.2813539      12321
## 3  123 123      7.092  0.2056318      15129
## 4  136 136      9.333  0.5034558      18496
## 5  151 151     11.866  0.8136229      22801
## 6  167 167     15.115  1.3503024      27889
## 7  185 185     19.008  0.9176516      34225
## 8  205 205     25.596  2.4731545      42025
## 9  227 227     34.937  7.0523236      51529
## 10 251 251     42.941  3.2107128      63001
## 11 278 278     53.335  1.7266297      77284
## 12 309 309     71.488  2.2730635      95481
## 13 342 342     94.241  2.9288392     116964
## 14 379 379    123.336  3.8497509     143641
## 15 419 419    160.155  3.7121011     175561
## 16 465 465    212.544  4.2603265     216225
## 17 515 515    276.896 11.4978067     265225
## 18 570 570    361.101  9.3000208     324900
## 19 632 632    480.394  9.7461060     399424
## 20 700 700    630.326 14.0494256    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
## -0.23073 -0.16716 -0.04825  0.12184  0.44446
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)    -15.2525     0.2465  -61.88  <2e-16 ***
## log(res_Naive_2D$n)  5.4133     0.0741   73.05  <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.1957 on 18 degrees of freedom
## Multiple R-squared:  0.9966, Adjusted R-squared:  0.9965
## F-statistic: 5336 on 1 and 18 DF, p-value: < 2.2e-16
## Exposant estimé (naïf): 5.413324
##
## Call:
## lm(formula = log(res_Kadane_2D$mean_time) ~ log(res_Kadane_2D$n))
##
```

```
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.065266 -0.023608 -0.008487  0.021311  0.108724
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)    -10.28647     0.08876  -115.9  <2e-16 ***
## log(res_Kadane_2D$n)  2.54607     0.01582   160.9  <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.04175 on 18 degrees of freedom
## Multiple R-squared:  0.9993, Adjusted R-squared:  0.9993
## F-statistic: 2.59e+04 on 1 and 18 DF,  p-value: < 2.2e-16
## Exposant estimé (Kadane): 2.546071
```

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

6 Cas particulier des données presque 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

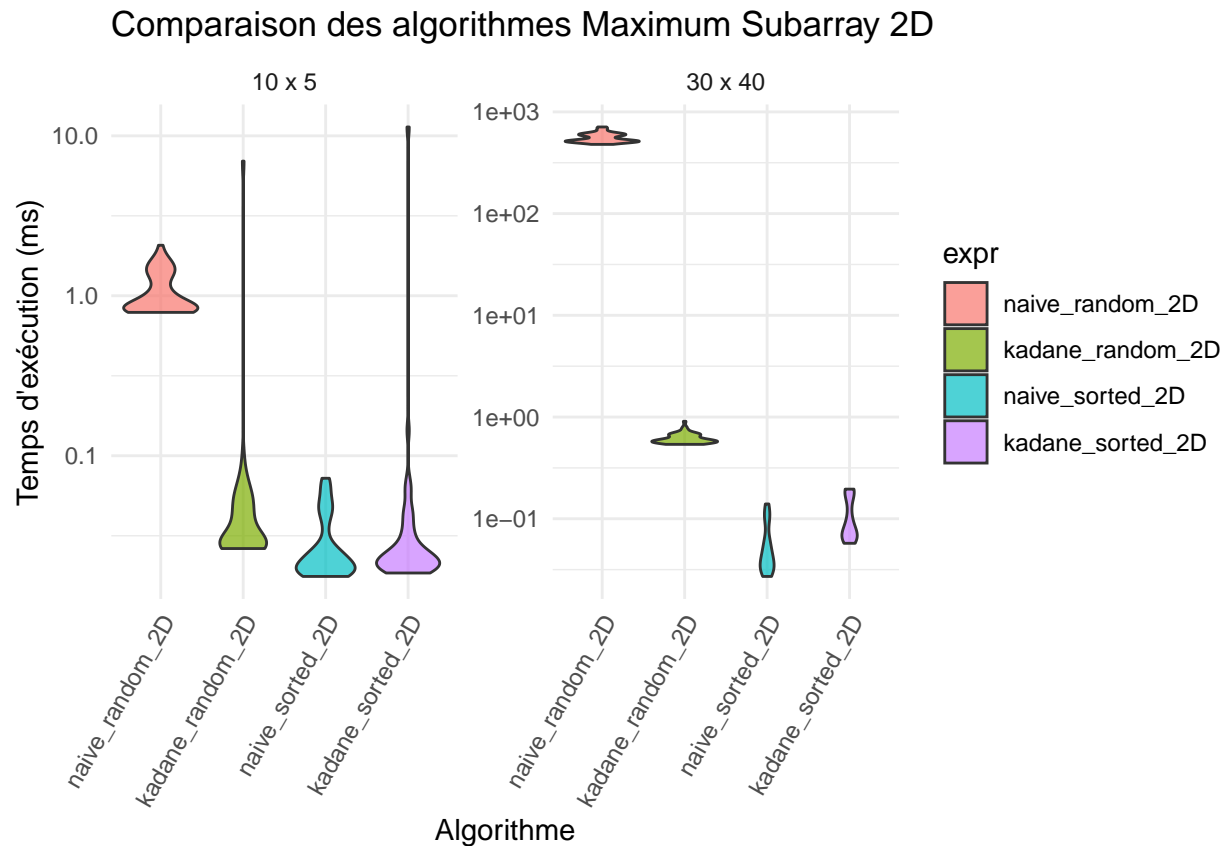
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## [1,]    1   11   21   31   41   51   61   71   81   91
## [2,]    2   12   22   32   42   52   62   72   82   92
## [3,]    3   13   23   33   18   53   63   73   83   93
## [4,]    4   14   24   34   44   54   64   74   84   94
## [5,]    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 == "Naive2D_cpp") return(max_subarray_rectangle_naive(mat))
  if (func == "Kadane2D_cpp") 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 == "Naive2D_cpp") return(max_subarray_rectangle_naive(mat))
  if (func == "Kadane2D_cpp") 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  0.787  8.25e-1    0.868    1.07   1.34e+0  2.07
## 2    50 kadane_random_2D 0.0263 2.75e-2    0.0308   0.177   4.91e-2  6.95
## 3    50 naive_sorted_2D  0.0176 1.92e-2    0.0219   0.0289  3.02e-2  0.0723
## 4    50 kadane_sorted_2D 0.0185 2.02e-2    0.0233   0.256   3.10e-2  11.4
## 5   1200 naive_random_2D 479.    5.12e+2    524.     558.    5.99e+2  711.
## 6   1200 kadane_random_2D 0.539  5.66e-1    0.591    0.617   6.71e-1  0.908
## 7   1200 naive_sorted_2D 0.0271 3.27e-2    0.0432   0.0566  6.15e-2  0.140
## 8   1200 kadane_sorted_2D 0.0574 6.60e-2    0.0812   0.110   1.77e-1  0.196

```

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.

7 Cas particulier avec matrice symétrique

On considère une matrice symétrique

Sur un exemple cela donne :

```
# Création d'une matrice triée
n <- 10 # nombre de lignes
m <- 10 # nombre de colonnes

v_sym <- sample(1:100, n * m, replace = TRUE)
mat <- matrix(v_sym, nrow = n, ncol = m)
mat[lower.tri(mat)] <- t(mat)[lower.tri(mat)]

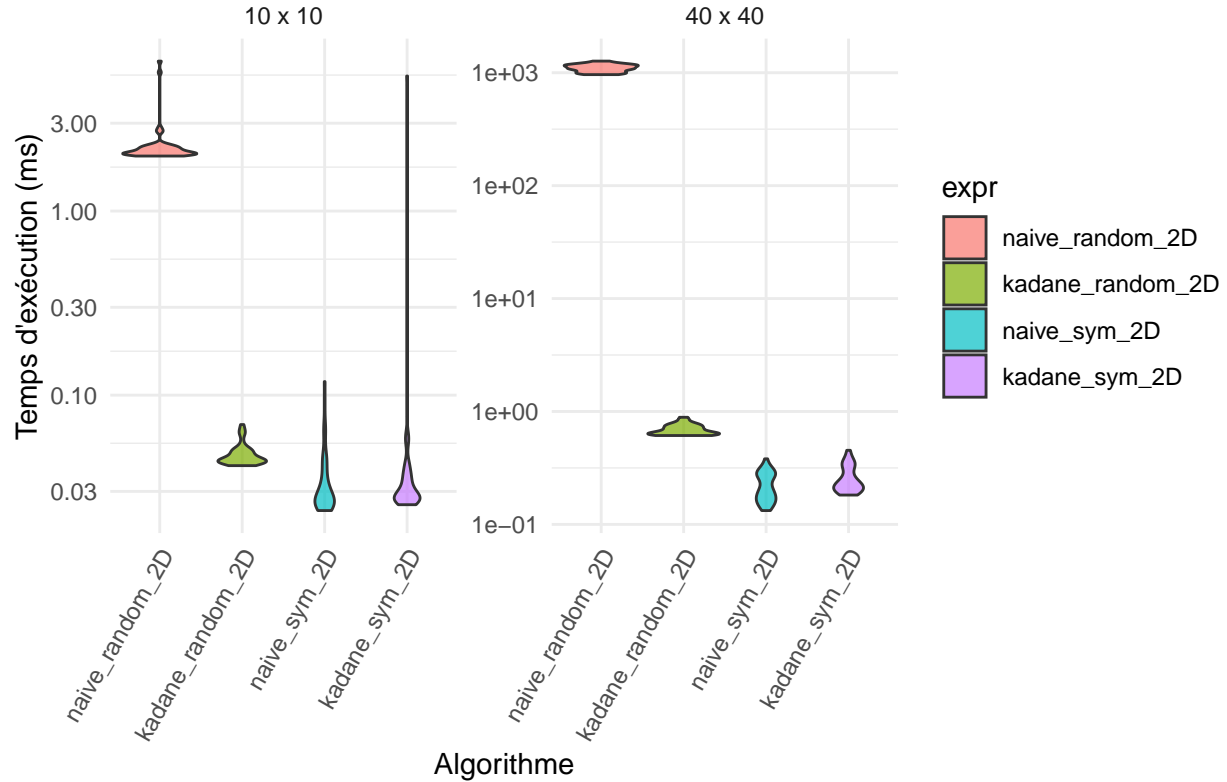
mat

##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## [1,]  50  59  16  85  17  37  38  98  37  92
## [2,]  59  52  11  76  85  92  8  24  89  14
## [3,]  16  11  70  12  28  69  82  49  80  96
## [4,]  85  76  12  86  32  16  81  41  6  56
## [5,]  17  85  28  32  39  58  76  40  28  24
## [6,]  37  92  69  16  58  93  44  16  24  90
## [7,]  38  8  82  81  76  44  12  91  65  18
## [8,]  98  24  49  41  40  16  91  15  65  38
## [9,]  37  89  80  6  28  24  65  65  80  48
## [10,] 92  14  96  56  24  90  18  38  48  23

# Fonctions de simulation
one.simu <- function(n,m, func) {
  mat <- matrix(sample(-100:100, n * m, replace = TRUE), nrow = n, ncol = m)
  if (func == "Naive2D_cpp") return(max_subarray_rectangle_naive(mat))
  if (func == "Kadane2D_cpp") return(max_subarray_rectangle_opt_Rcpp(mat))
}

one.simu2 <- function(n,m, func) {
  v_sym <- sample(1:100, n * m, replace = TRUE)
  mat <- matrix(v_sym, nrow = n, ncol = m)
  mat[lower.tri(mat)] <- t(mat)[lower.tri(mat)]
  if (func == "Naive2D_cpp") return(max_subarray_rectangle_naive(mat))
  if (func == "Kadane2D_cpp") return(max_subarray_rectangle_opt_Rcpp(mat))
}
```

Comparaison des algorithmes Maximum Subarray 2D



```
## # 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   100 naive_random_2D    1.99  2.04e+0         2.10      2.31  2.23e+0  6.52e+0
## 2   100 kadane_random_2D  0.0414 4.34e-2         0.045    0.0476 4.98e-2  6.92e-2
## 3   100 naive_sym_2D      0.0236 2.55e-2         0.0281    0.0353 3.70e-2  1.18e-1
## 4   100 kadane_sym_2D     0.0254 2.72e-2         0.0287    0.140  3.59e-2  5.41e+0
## 5  1600 naive_random_2D  962.    1.02e+3        1094.    1101.    1.17e+3  1.27e+3
## 6  1600 kadane_random_2D  0.612  6.32e-1         0.658    0.691  7.35e-1  8.88e-1
## 7  1600 naive_sym_2D     0.132  1.68e-1         0.205    0.220  2.81e-1  3.81e-1
## 8  1600 kadane_sym_2D     0.182  2.04e-1         0.226    0.258  3.14e-1  4.53e-1
```

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