Astuce :
Créer des Heatmaps à partir de grosses matrices en R

jeu 11 Juin 2020 Guillaume DevaillyAstuce 0

En génomique, et sans doute dans tout un tas d'autres domaines *omiques* ou *big data*, nous essayons souvent de tracer des grosses matrices sous forme d'*heatmap*. Par *grosse matrice*, j'entends une matrice dont le nombre de lignes et/ou de colonnes est plus grand que le nombre de pixels sur l'écran que vous utilisez. Par exemples, si vous avez une matrice de 50 colonnes et de 20 000 lignes (cas assez fréquent quand il y a une ligne par gène), il y a de forte chances que cette matrice aura plus de lignes qu'il n'y a de pixels sur votre écran -- 1080 pixels verticaux sur un écran HD (à moins bien sûr que vous lisiez ceci dans un futur lointain d'hyper haute définition).

Le problème lorsqu'on affiche des matrices qui ont plus de lignes que de pixel à l'écran, c'est justement que chaque pixel va devoir représenter plusieurs cellules de la matrice, et que le comportement par défaut de R sur ce point-là n'est pas forcément optimal.

Un exemple de données numériques

Commençons par générer un faux jeu de données, imitant ce qu'on peut obtenir en épigénomique. J'essaye de produire un signal centré au milieu des rangs, de plus en plus fort, tout en gardant une part d'aléatoire. Je laisse le code pour que vous puissiez jouer chez vous à reproduire les figures de cet article, mais vous n'avez pas besoin de comprendre cette section pour comprendre la suite.

```
library(dplyr)
library(purrr)

Ncol <- 50
genmat <- map(
    1:20000,
    function(i) {
        runif(Ncol) + c(sort(abs(rnorm(50)))[1:25], rev(sort(abs(rnorm(50)))[1:25]))  * i/5000
    }
```

```
) %>% do.call(rbind, .)

genmat[1:5, 1:5]
##            [,1]      [,2]       [,3]      [,4]      [,5]
## [1,] 0.24750229 0.8144309 0.31405005 0.2787540 0.8435071
## [2,] 0.44266149 0.1147394 0.28464511 0.6437944 0.7597911
## [3,] 0.11495737 0.6750608 0.04393633 0.5712240 0.2088942
## [4,] 0.16660166 0.5508895 0.75274403 0.7340737 0.9325773
## [5,] 0.07285492 0.4573314 0.09437322 0.1534962 0.4939674

dim(genmat)
## [1] 20000    50
```

Il existe de nombreuses fonctions en R pour afficher cette matrice, par exemple `heatmap()`, `gplots::heatmap.2()`, `ggplot2::geom_raster()`, ou `ComplexHeatmap::Heatmap()`. La plupart de ces fonctions font appel à la fonction `image()` de plus bas niveau, qui est celle qui trace la matrice colorée. C'est cette fonction que nous allons utiliser dans ce billet:

```
oldpar <- par(mar = rep(0.2, 4)) # reducing plot margins
image(
    t(genmat), # image() has some weird opinions about how your matrix will be plotted
    axes = FALSE,
    col = colorRampPalette(c("white", "darkorange", "black"))(30), # our colour palette
    breaks = c(seq(0, 3, length.out = 30), 100) # colour-to-value mapping
)
box() # adding a box arround the heatmap
```

Figure 1: Une bien belle heatmap ?

On pourrait penser que tout va bien ici. On arrive bien à voir un signal, et on en tire la conclusion qu'il est au centre, plus fort en haut qu'en bas. Le souci c'est qu'avec 20 000 lignes dans notre matrice on devrait avoir une image beaucoup moins bruitée. Comme il n'y a que quelques centaines de pixels de hauteur dans le *png* (par défaut), R doit décider d'une manière ou d'une autre comment résumer l'information de plusieurs cellules en un seul pixel. Il semble que R choisisse plus ou moins au hasard une seule cellule à afficher par pixel (probablement la première ou la dernière dans la pile). Il y a donc un sous-échantillonnage important.

On pourrait imaginer générer un *png* de plus de 20 000 pixels de haut pour compenser, mais ça fait des fichiers lourds à manipuler, et il faut penser à agrandir d'autant la taille du texte et l'épaisseur des traits pour un résultat potable.

Autre idée, certaines *devices* graphiques (par exemple `pdf()`, mais pas `png()`) permettent jouer avec le paramètre `useRaster = TRUE` de la fonction `image()`, ce qui peut aider dans quelques situations. La rasterisation, d'après wikipedia, "*est un procédé qui consiste à convertir une image vectorielle en une image matricielle*". L'algorithme de rasterisation va donc essayer de convertir plusieurs lignes de données en un seul pixel.

```
pdf("big_hm_1.pdf")
layout(matrix(c(1, 2), nrow = 1)) # side by side plot

# Left plot, no rasterisation
image(
    t(genmat),
    axes = FALSE,
    col = colorRampPalette(c("white", "darkorange", "black"))(30),
    breaks = c(seq(0, 3, length.out = 30), 100),
    main = "Original matrix"
)

# Right plot, with rasterisation
image(
    t(genmat),
    axes = FALSE,
    col = colorRampPalette(c("white", "darkorange", "black"))(30),
    breaks = c(seq(0, 3, length.out = 30), 100),
    useRaster = TRUE,
    main = "With rasterization"
)

dev.off()
```

Le fichier *pdf* généré est disponible ici. Mais les différents lecteurs *pdf* n'affichent pas le même rendu des *plots* en questions :

Figure 2 : useRaster = TRUE, une solution loin d'être idéale

Acrobat, Edge et Okular donnent le rendu attendu: une représentation bien plus fine des données originales lorsque la rasterisation est activée. Evince et SumatraPDF inversent les rendus, et *voilent* la version "non rasterisée" ! Le lecteur de *pdf* de Firefox abandonne carrément (en tout cas sous Windows 10, sous GNU/Linux il affiche le même résultat qu'Acrobat, Edge et Okular). Si votre lecteur de *pdf* préféré n'est pas parmi ceux que j'ai testé, je serai curieux d'avoir le résultat que vous obtenez en commentaire.

Pour info, alors que le *pdf* fait 5 Mo, le même code exportant du *svg* génère un fichier de 200 Mo ! Je n'ai lâchement pas eu le courage de l'ouvrir pour voir le rendu obtenu...

Au final, la rasterisation essaie de résumer les informations contenues derrières chaque pixel en en faisant une moyenne. Mais c'est un processus qu'on peut essayer de faire nous même, ce qui a deux avantages : on s'affranchit des différences de rendus entre lecteurs de *pdf*, et ça marchera même sur les *devices* non vectoriels, du genre *png*, ce qui évite de générer des images trop lourdes.

L'idée est donc de redimensionner la matrice **avant** le *plot*, en la rendant plus petite et en appliquant une fonction qui "résumera" les cellules correspondantes à chaque pixel (par exemple la fonction `mean()`). Je vous propose cette petite fonction (aussi disponible sur canSnippet) :

```
# reduce matrix size, using a summarizing function (default, mean)
redim_matrix <- function(
    mat,
    target_height = 100,
    target_width = 100,
    summary_func = function(x) mean(x, na.rm = TRUE),
    output_type = 0.0, #vapply style
    n_core = 1 # parallel processing
    ) {

    if(target_height > nrow(mat) | target_width > ncol(mat)) {
        stop("Input matrix must be bigger than target width and height.")
    }

    seq_height <- round(seq(1, nrow(mat), length.out = target_height + 1))
    seq_width  <- round(seq(1, ncol(mat), length.out = target_width  + 1))

    # complicated way to write a double for loop
    do.call(rbind, parallel::mclapply(seq_len(target_height), function(i) { # i is row
        vapply(seq_len(target_width), function(j) { # j is column
            summary_func(
                mat[
                    seq(seq_height[i], seq_height[i + 1]),
                    seq(seq_width[j] , seq_width[j + 1] )
                    ]
            )
        }, output_type)
    }, mc.cores = n_core))
}

genmatred <- redim_matrix(genmat, target_height = 600, target_width = 50) # 600 is very rougthly the pixel height of the image.

genmatred[1:5, 1:5]
##           [,1]      [,2]      [,3]      [,4]      [,5]
## [1,] 0.4530226 0.4911097 0.4927123 0.5302643 0.5561331
## [2,] 0.5263392 0.5138786 0.5324716 0.5354325 0.5050932
## [3,] 0.4196155 0.4887105 0.5238630 0.5183627 0.5296764
## [4,] 0.5024431 0.5015508 0.5155568 0.5537814 0.5318501
## [5,] 0.5121447 0.5533040 0.4882006 0.4877140 0.5222805

dim(genmatred)
## [1] 600  50
```

Comparons le rendu Avant / Après :

```
layout(matrix(c(1, 2), nrow = 1))

# left plot, original matrix
image(
    t(genmat),
    axes = FALSE,
    col = colorRampPalette(c("white", "darkorange", "black"))(30),
    breaks = c(seq(0, 3, length.out = 30), 100),
    main = "Original matrix"
)
box()

# Right plot, reduced matrix
image(
    t(genmatred),
    axes = FALSE,
    col = colorRampPalette(c("white", "darkorange", "black"))(30),
    breaks = c(seq(0, 3, length.out = 30), 100),
    main = "Reduced matrix"
)
box()

par(oldpar) # restoring margin size to default values
```

Figure 3 : Un rendu bien plus fin lorsqu'on réduit la taille de la matrice nous même.

Paradoxalement, on "discerne" bien mieux les détails des 20 000 lignes de la matrice en réduisant la taille de la matrice nous même, plutôt qu'en laissant R (mal) afficher les 20 000 lignes.

Matrices creuses (*sparse*)

Dans certaines situations, faire la moyenne des cellules par pixel n'est pas la manière la plus maline de résumer les données. Par exemple, dans le cas de matrices creuses, on ne souhaite pas moyenner nos quelques valeurs isolées par tous les zéros les entourant. Dans ces cas-là, prendre la valeur maximale correspondra mieux à ce qu'on cherche à montrer.

J'ai rencontré ce cas dans une étude d'*eQTL* (analyse QTL en utilisant les niveaux d'expressions des gènes comme phénotypes). L'idée est d'identifier des *Single Nucleotide Polymorphisms* (SNP, des variants / mutations) qui sont associés à des changements d'expression des gènes. Pour cela, on fait un test statistique d'association entre chaque SNP et chaque niveau d'expression de gènes, ce qui nous donne autant de p-valeurs.

Nous avions l'expression d'environ 20 000 gènes, et environ 45 000 SNP, résultant en une matrice de 20 000 x 45 000 p-valeurs. La plupart des p-valeurs sont non significatives, et seule une minorité était très petite (ou très grande après transformation en *-log10(p-valeur)*). Or, ce qu'on souhaite c'est afficher les p-valeurs des SNP principaux (*lead SNP*). On va donc plutôt prendre le maximum des *-log10(p-valeur)* plutôt que leur moyenne :

```
# left matrix, we take the mean of the -log10 of the p-values
redim_matrix(
  eqtls,
  target_height = 600, target_width = 600,
  summary_func = function(x) mean(x, na.rm = TRUE),
  n_core = 14
)

# right matrix we take the maximum of the -log10 of the p-values
redim_matrix(
  eqtls,
  target_height = 600, target_width = 600,
  summary_func = function(x) max(x, na.rm = TRUE),
  n_core = 14
)
```

Figure 4: À gauche : On résume la grosse matrice en calculant la moyenne des p-valeurs par pixel. À droite : On prend la plus petite p-valeur (la plus grande après transformation -log10) pour mieux voir la significativité des lead SNP. Les p-valeurs réelles sont bien mieux représentés.

Données catégorielles

Dans le cas de données catégorielles, on ne peut pas vraiment prendre une moyenne des valeurs. Il faut plutôt faire **les moyennes des couleurs** associées à chaque catégorie (Ici, je le fais dans l'espace colorimétrique RGB, mais ça fonctionne peut-être encore mieux si la moyenne est faite en espace HCL ?). Pour afficher une matrice de couleurs, il faut utiliser `rasterImage()` au lieu d'`image()`.

```
# some fake data
mycolors <- matrix(c(
  sample(c("#0000FFFF", "#FFFFFFFF", "#FF0000FF"), size = 5000, replace = TRUE, prob = c(2, 1, 1)),
  sample(c("#0000FFFF", "#FFFFFFFF", "#FF0000FF"), size = 5000, replace = TRUE, prob = c(1, 2, 1)),
  sample(c("#0000FFFF", "#FFFFFFFF", "#FF0000FF"), size = 5000, replace = TRUE, prob = c(1, 1, 2))
), ncol = 1)
color_mat <- t(as.matrix(mycolors))

# custom funtion to average HTML colors
mean_color <- function(mycolors) {
    R     <- strtoi(x = substr(mycolors,2,3), base = 16)
    G     <- strtoi(x = substr(mycolors,4,5), base = 16)
    B     <- strtoi(x = substr(mycolors,6,7), base = 16)
    alpha <- strtoi(x = substr(mycolors,8,9), base = 16)

    return(
        rgb(
            red   = round(mean(R)),
            green = round(mean(G)),
            blue  = round(mean(B)),
            alpha = round(mean(alpha)),
            maxColorValue = 255
        )
    )
}

# Let's apply the redim_matrix() function using ou newly defined mean_color() function:
color_mat_red <- redim_matrix(
  color_mat,
  target_height = 1,
  target_width = 500,
  summary_func = mean_color,
  output_type = "string"
)

# And do the plotting
layout(matrix(c(1, 2), nrow = 2))

# left plot, original matrix
plot(c(0,1), c(0,1), axes = FALSE, type = "n",  xlab = "", ylab = "", xlim = c(0, 1), ylim = c(0,1), xaxs="i", yaxs="i", main = "Full matrix")
rasterImage(
    color_mat,
    xleft   = 0,
    xright  = 1,
    ybottom = 0,
    ytop    = 1
)
box()

# right plot, summarised matrix
plot(c(0,1), c(0,1), axes = FALSE, type = "n",  xlab = "", ylab = "", xlim = c(0, 1), ylim = c(0,1), xaxs="i", yaxs="i", main = "Reduced matrix")
rasterImage(
    color_mat_red,
    xleft   = 0,
    xright  = 1,
    ybottom = 0,
    ytop    = 1
)
box()
```

Figure 5: Réduire la taille d'une matrice catégorielle avant le *plot*, en faisant la moyenne des couleurs par pixel, permet de représenter plus fidèlement les données.

ggplot2

D'après mes tests, `ggplot2` est aussi affecté par ce souci d'overplotting, que ce soit `geom_tile()` ou `geom_raster()` (qui est une version optimisée de geom_tile() quand les cases sont régulières).

```
library(ggplot2)
library(patchwork)

# Wide to long transformation
data_for_ggplot <- as.data.frame(genmat) %>%
    mutate(row = rownames(.)) %>%
    tidyr::pivot_longer(-row, names_to = "col") %>%
    mutate(row = as.numeric(row), col = readr::parse_number(col))

# with geom_tile()
p1 <- ggplot(data_for_ggplot, aes(x = col, y = row, fill = value)) +
  geom_tile() +
  scale_fill_gradient2(
      low = "white", mid = "darkorange", high = "black",
      limits = c(0, 3), midpoint = 1.5, oob = scales::squish
  ) +
  labs(title = "geom_tile") +
  theme_void() +
  theme(legend.position = "none")

# with geom_raster()
p2 <- ggplot(data_for_ggplot, aes(x = col, y = row, fill = value)) +
  geom_raster() +
  scale_fill_gradient2(
      low = "white", mid = "darkorange", high = "black",
      limits = c(0, 3), midpoint = 1.5, oob = scales::squish
  ) +
  labs(title = "geom_raster") +
  theme_void()
  theme(legend.position = "none")

p1 + p2
```

Figure 6 : ggplot2 victime de l'overplotting. Notez de subtiles différences entre geom_tile() et geom_raster().

ComplexHeatmap

Le package Bioconductor ComplexHeatmap est vraiment top pour générer des Heatmaps un peu complexe, avec des annotations dans tous les sens.

Cela dit, mes quelques tests suggèrent qu'il souffre du même problème d'overplotting que les autres fonctions. Il réalise un sous-échantillonnage des cellules à afficher, au lieu de moyenner les données par pixel :

```
library(ComplexHeatmap)

Heatmap(
  genmat[nrow(genmat):1, ], # putting the top on top
  col = circlize::colorRamp2(c(0, 1.5, 3), c("white", "darkorange", "black")),
  cluster_rows = FALSE, cluster_columns = FALSE,
  show_heatmap_legend = FALSE,
  column_title = "No rasterisation"
)
```

Figure 7 : ComplexHeatmap nous déçoit.

La fonction `Heatmap()` a bien des paramètres qui permettent une *rasterization* dans le cas de grosses matrices, mais ils semblent plus utiles pour réduire le poids des fichiers vectoriels que pour résoudre le problème de sous-échantillonnage :

```
Heatmap(
  genmat[nrow(genmat):1, ],
  col = circlize::colorRamp2(c(0, 1.5, 3), c("white", "darkorange", "black")),
  cluster_rows = FALSE, cluster_columns = FALSE,
  show_heatmap_legend = FALSE,
  use_raster = TRUE,
  raster_resize = TRUE, raster_device = "png",
  column_title = "With rasterisation"
)
```

Figure 8: ComplexHeatmap nous déçoit même avec use_raster = TRUE

Conclusion

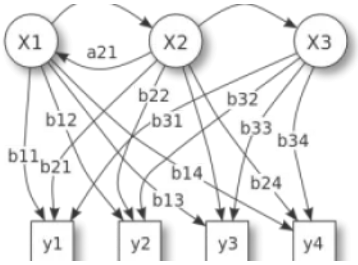
En R, réduisez vos grosses matrices avant de les afficher, vous verrez mieux les petits détails. Sinon vous obtiendrez des heatmaps un peu approximatives.

Le principal souci de cette solution, c'est qu'il faut faire les décorations (axes, barres de couleurs sur les côtés, dendrogrammes, *etc.*) à la main, ce qui est un peu laborieux.

Merci a mes talentueux relecteurs Mathurin, Gwenaëlle, et lhtd. Une version de cet article traduite en anglais sera publiée prochainement sur le blog de l'auteur.

Articles similaires

Les matrices de substitution
Dans les premiers billets de ce blog, nous avons présenté les alignements multiples de séquences d'une part du point de vue des logiciels et ensuite du calcul de la conservation. Je vous propose aujourd'hui de revenir sur un point important : les matrices de substitution. Commençons par
mar 16 Oct 2012
Dans "Découverte"



Suivez le guide : en quête de HMM
mer 3 Juil 2013
Dans "Suivez l'guide"


Cours de R pour débutant pressé (introduction)
Cours de R pour débutant pressé (introduction)
mer 19 Sep 2012
Dans "Suivez l'guide"

- **À propos de Guillaume Devailly**
- Après une thèse en cancérologie à Lyon et un postdoc en bioinformatique à Édimbourg, je suis chercheur à l'INRA Toulouse depuis fin 2017. Régulation transcriptionnelle et épigénétique. Twitter: @G_Devailly

Catégorie: Astuce | Tags: heatmaps, matrices, plot, R, visualisation
S'abonner au flux de commentaires de cet article

Laisser un commentaire

Entrez votre commentaire...

← Précédent Suivant →

Recherche

La boutique officielle de Bioinfo-fr.net !

Portez haut les couleurs de votre blog de bioinfo préféré !

Les profits nous aideront à maintenir le blog et à vous faire de beaux cadeaux :)

Les snippets de Bioinfo-fr

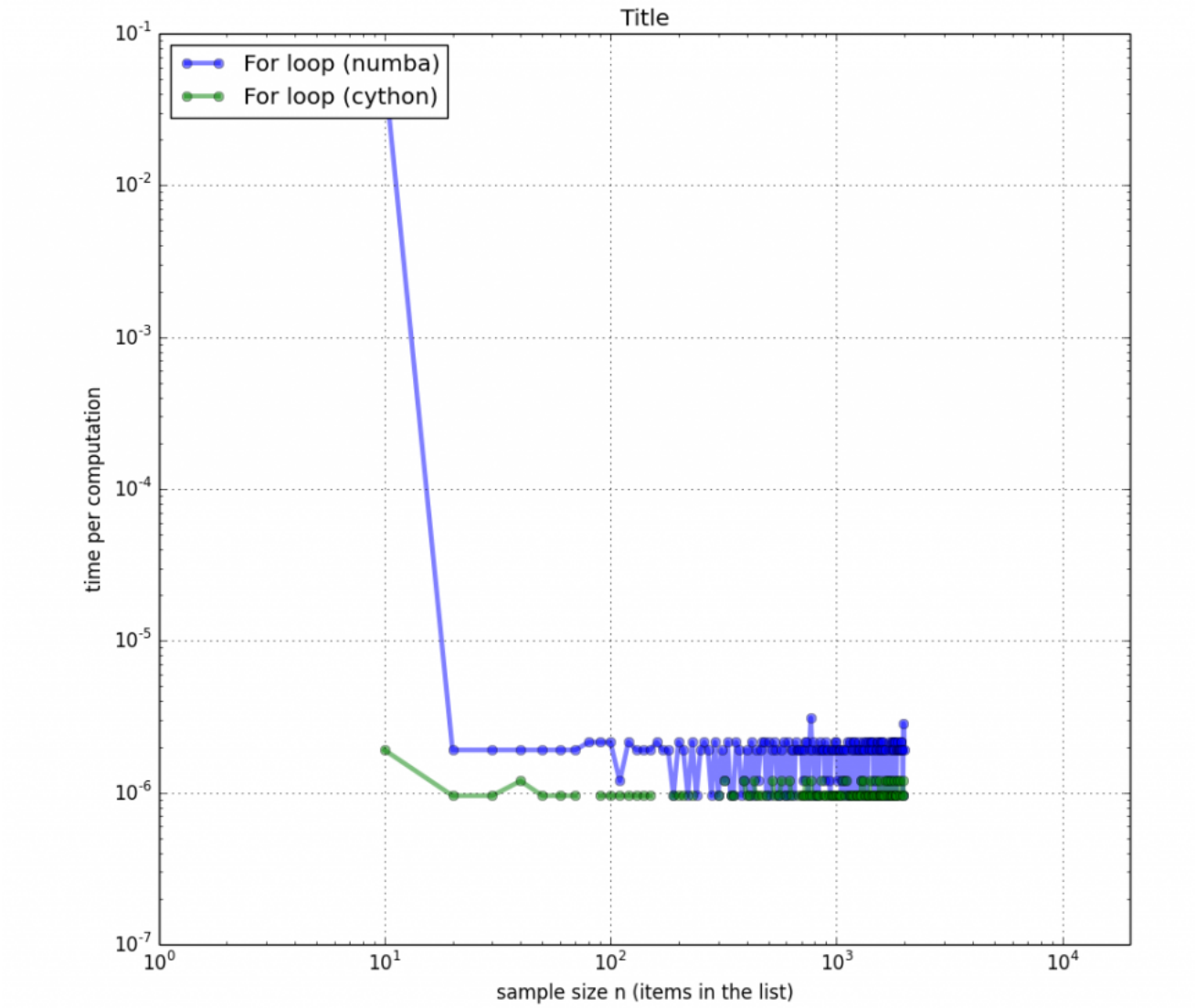
Partagez vos bouts de code !

canSnippet

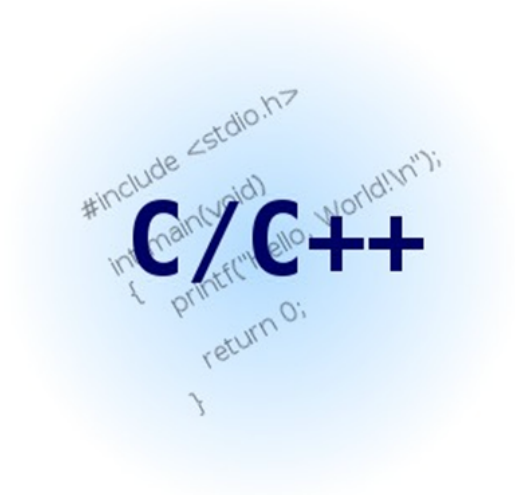
Articles au hasard

- Présentation de l'association JeBiF

ven 20 Avr 2012

- 

[Python fait la numba]
mer 26 Nov 2014

- 

[Les langages de programmation]
mer 29 Fév 2012

- 

[Cython: votre programme Python mais 100x plus vite]
mer 10 Sep 2014

Questions à… Gabriel Chandesris
mer 18 Mai 2016

Liens

- BioStar
- canSnippet
- JeBiF
- Les Bioinformations
- SFBI

Catégories

- Actualité (17)
- Astuce (36)
- Bioinformatique (1)
- Brèves (9)
- Conférence (13)
- Contribuer (1)
- Découverte (116)
- Didacticiel (60)
- Editorial (59)
- En image (11)
- Entretien (13)
- Formation (14)
- J'ai lu (9)
- Journal Club (5)
- Opinion (31)
- Suivez l'guide (28)

Commentaires récents

- Pierre-Edouard Guerin dans Ce qu'il faut voir sur une carte de contact chromosomique
- Ismael dans BLAST en pratique
- Pauline dans Télécharger des données de séquençage sur le NCBI.. pour les débutants!
- SOFIAN BATON dans LaTeX : les lettres (de motivation) !
- Gwenaelle dans Pourquoi et comment déposer un package R sur Bioconductor ?

Étiquettes

ADN analyse base de données bioinformatique code Communauté concours conférence débutant Découverte edito emploi formation GenBank Génomique Interview JeBiF JOBIM langage LaTeX master modélisation Métagénomique NCBI NGS outil Perl phylogénie pipeline programmation protéine Python R rentrée script SFBI SQL statistiques strip séquençage thèse Tips tutoriel vacances visualisation

Aller en haut