Restauration d'image à l'aide d'algorithme stochastiques par chaînes de Markov

Code

Hide

```
#library(magrittr)
#library(imager)
library(stats)
library(survival)
library(ggplot2)
```

```
Keep up to date with changes at https://www.tidyverse.org/blog/
```

Différence entre l'approche bayésienne et non bayésienne

Pour une configuration initiale donnée, on va pouvoir considérer de nouvelles configurations candidates pour la remplacer. Il sagit alors de savoir quelle candidate est la meilleure. Deux manières de faire sont ici présentées: une approche bayésiennes, et non bayésiennes.
Pour une approche non bayésienne du problème, l'évaluation d'une configuration candidate est booléenne: soit on accepte le candidat, soit on le rejete, autrement dit la probabilité d'une nouvelle configuration vaut 0 ou 1, car on n'a aucune information apriori sur la configuration de l'image.
Différement, avec une analyse bayésienne d'une candidate, on dispose d'informations à priori sur cette dernière, et on lui attribuera ainsi une probabilité entre 0 et 1 d'être la meilleure.

$$\begin{aligned} P(X_s = x_s | X^s = x^s) &= \frac{P(X_s = x_s, X^s = x^s)}{P(X^s = x^s)} \text{ formule de Bayes} \\ &= \frac{P(X = x)}{P(X^s = x^s)} \\ &= \frac{e^{-U(x)}}{e^{-U(x^s)}} \end{aligned}$$

$$\text{On a } U(x^s) = U_{\bar{s}}(x) = \sum_{c \in C, s \notin c} U_c(x)$$

$$\begin{aligned} \text{Et } U(x) &= \sum_{c \in C} U_c(x) \\ &= \sum_{c \in C, s \in c} U_c(x) + \sum_{c \in C, s \notin c} U_c(x) \\ &= U_s(x_s, V_s) + U_{\bar{s}}(x) \end{aligned}$$

$$\begin{aligned} \text{Donc , } P(X_s = x_s | X^s = x^s) &= \frac{exp(-U_s(x_s, V_s) - U_{\bar{s}}(x))}{exp(-U_{\bar{s}}(x))} \\ &= \frac{exp(-U_s(x_s, V_s)}{exp(-U_{\bar{s}}(x) + U_{\bar{s}}(x))} \\ &= exp(-U_s(x_s, V_s)) \text{ ce qui justifie l'hypthère markovienne} \end{aligned}$$

Méthodes non bayésiennes

Algorithmes d'échantillonage RMF adaptés à la restauration d'image

Outil pour les algorithmes

Fonction de génération d'image

Hide

```
#====================================================================================
====================

imageGenerator=function(forme, N){
  #fonction permettant de générer une matrice-image de forme choisie, et de taille N*
N

  img <- matrix(-1, nrow= N, ncol= N)

  #----rectangle----
  if(forme== "rectangle"){
    img[(N/4):(N*3/4),(N/4):(N*3/4)] <- 1}

  #----losange----
  if(forme== "losange"){
    for (i in 1:N){
      for (j in 1:N){
        sigma0[i,j]=-1+2*(abs(i-j)<=N/4)*(abs(i+j-N)<=N/4)
      }
    }#for
  }#if

  #----damier----
  #----triangle----
  #----cercle----
  #----bande----


  return(img)
}#func

#====================================================================================
====================
```

fonction de bruitage d'image

Hide

```
#====================================================================================
====================

bruiteur=function(image, br){
  #fonction qui revoit l'image donnée en argument par la même image mais bruité par l
a méthode "br"
  imgBr <- image
  N <- #RECUP LONGEUR IMAGE
  bruit <- 0
  p <- 0.15
  #----binomiale----
  if(br== "rbinom"){bruit <- matrix(2*rbinom(N^2,1,1-p)-1,nrow=N,ncol=N)}
  #----poisson----
  #----normale----

  imgBr <- image*br
  return(imgBr)
}#func  A FINIR

#====================================================================================
====================
```

fonction de calcul de l'énergie générale d'une image

Hide

```
#====================================================================================
====================

energie=function(S){
  #fonction pour mesurer l'enrgie globale d'une image

  eligne=0
  for (i in 1:N) {for (j in 1:(N-1)){eligne=eligne+S[i,j]*S[i,j+1]}}
  ecolonne=0
  for (j in 1:N) {for (i in 1:(N-1)){ecolonne=ecolonne+S[i,j]*S[i+1,j]}}
  return(eligne+ecolonne)
}

#====================================================================================
====================
```

fonction de calcul de correspondance pixel par pixel entre deux images

Hide

```
correspondance=function(img0, img1, N){
  #----pourcentage de correspondance----
  nok <- 0 #nombre de pixels divergents
  correspondance <- 0 #pourcentage sur nb pixels total
  for (i in (1:N)){
    for (j in (1:N)){
      if(img0[i,j] != img1[i,j]){nok <- nok+1}
    }#for
  }#for
  correspondance <- ((N^2)-nok)/(N^2)*100
  return(correspondance)
}#func
```

Algorithme de métropolis (metropolisIsing)

Le parametre alpha détermine le poid donné à l'image bruitée initiale. - Si alpha est trop grand, les valeurs les plus probables seront toujours celles de l'image bruitée, et donc l'algorithme ne fera rien. - Si il est trop petit voir nulle, la configuration initiale va être oubliée petit à petit, et on va tendre vers l'image branche, soit donc la configuration d'énergie minimale ne tenant pas compte de l'image bruitée.

Le calcul du voisinage se fait grâce à la fonction Vstar, détaillée plus bas. Pour avoir le voisinage standard (les quatres pixels haut bas gauche droite), il suffit de mettre 1 comme valeur pour L.

Le parcours des sites peut se faire soit aléatoirement, soit ligne par ligne.

Hide

```
#=======================================================================================
====================

metropolisIsing=function(forme, N, beta, B, alpha, L, n, p){
  #fonction qui réalise l'algorithme de Metropolis sur une image img0 qui est bruité
en img1, et doit à la fin être restaurée en img2

  #----création de l'image parfaite img0----
  img0 = imageGenerator(forme, N)
  #image(img0)
  #----création d'une image auxiliaire img1, avec bord plus long et bruitage----
  img1 = matrix(0, nrow=N+2, ncol=N+2)
  bruit = matrix(2*rbinom(N^2,1,1-p)-1,nrow=N,ncol=N)
  img1[2:(N+1),2:(N+1)] = img0*bruit
  #image(img1)
  img2 = img1 #image tamponz
  img3 = img1 #mémoire du bruit initiale

  for(k in 1:n){
    #----choix d'un site---
    i <- 1+ceiling(N*runif(1)) # indice entre 2 et N+1
    j <- 1+ceiling(N*runif(1))
    #i <- 2 + k%%N #de 2 à N+1
    #j <- 2 + k%/%(N-1)%%N #de 2 à N+1
    #----tirage nouvelle valeure candidate----
    img2[i,j] = -img1[i,j]
    #----voisinage----
    #v = img1[i+1,j] + img1[i-1,j] + img1[i,j+1] + img1[i,j-1]
    v = Vstar(img1, N+2, i, j, L)
    #----potentiels----
    #U1 = -B*img1[i,j]-beta*img1[i,j]*v-alpha*img3[i,j]
    #U2 = -B*img2[i,j]-beta*img2[i,j]*v-alpha*img3[i,j]
    U1 = -B*img1[i,j]-img1[i,j]*(beta*v + alpha*img3[i,j])#potentiel actuel
    U2 = -B*img2[i,j]-img2[i,j]*(beta*v + alpha*img3[i,j])#potentiel candidat

    #----test----
    dU = U2-U1
    if(dU<0){img1[i,j]= -img1[i,j]}
    else{
      p <- exp(-dU)
      if(runif(1)<=p){img1[i,j]= -img1[i,j]}
      else{img2[i,j] <- img1[i,j]} #sinon on ne change pas
    }#else
  }#for
  #----affichage----
  #par(mfrow=c(1,3))
  #image(img1)

  corresp <- correspondance(img0, img1, N)
  return(corresp) #utile pour MCdeMC_Vstar
```

```
  #calcul de la correspondance
}



metropolisIsing("rectangle", 64, 1, 0, 1, 10, 10^4, 0.15)
```

```
[1] 92.65137
```

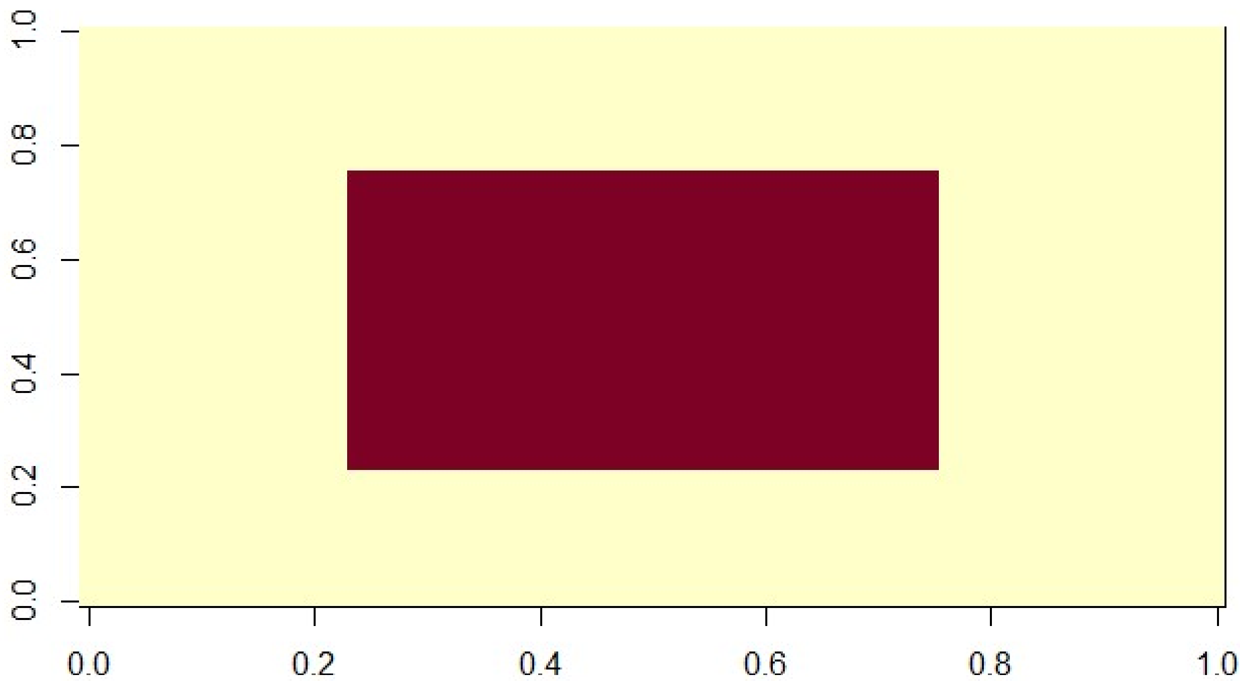
Hide

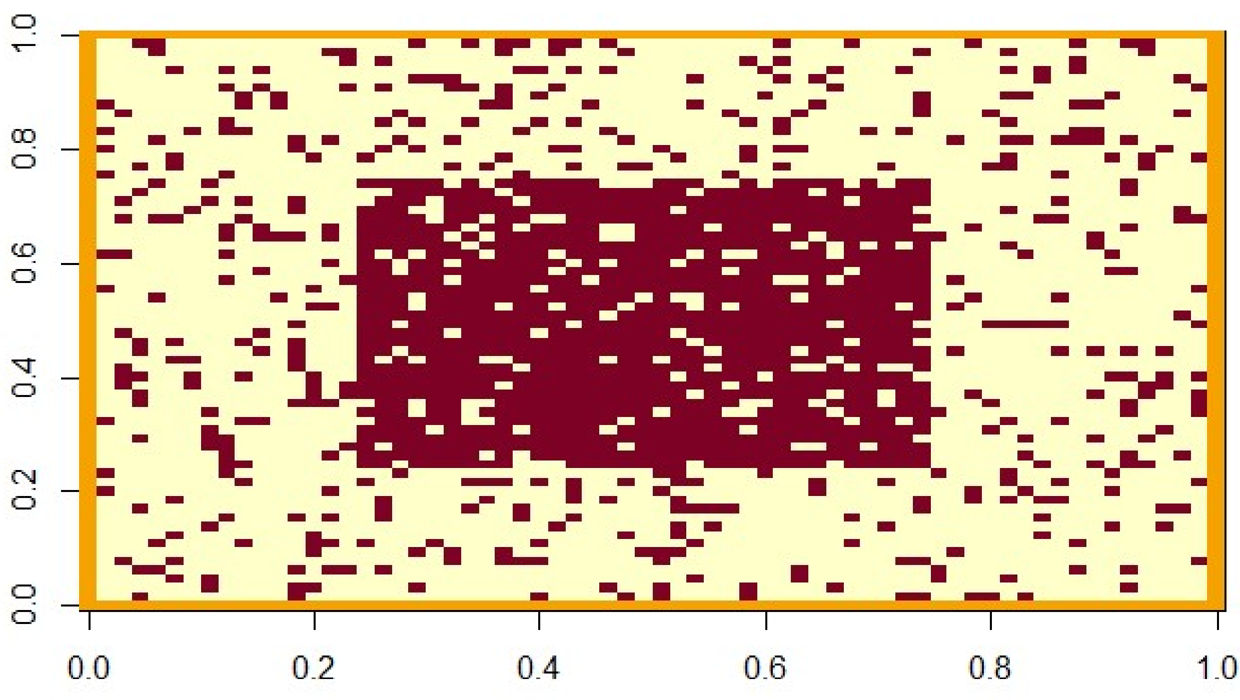
```
#metropolisIsing("rectangle", 64, 1, 0, 1, 5, 10^4)
#metropolisIsing("rectangle", 64, 1, 0, 1, 1, 10^4)

#=====================================================================================
====================
```

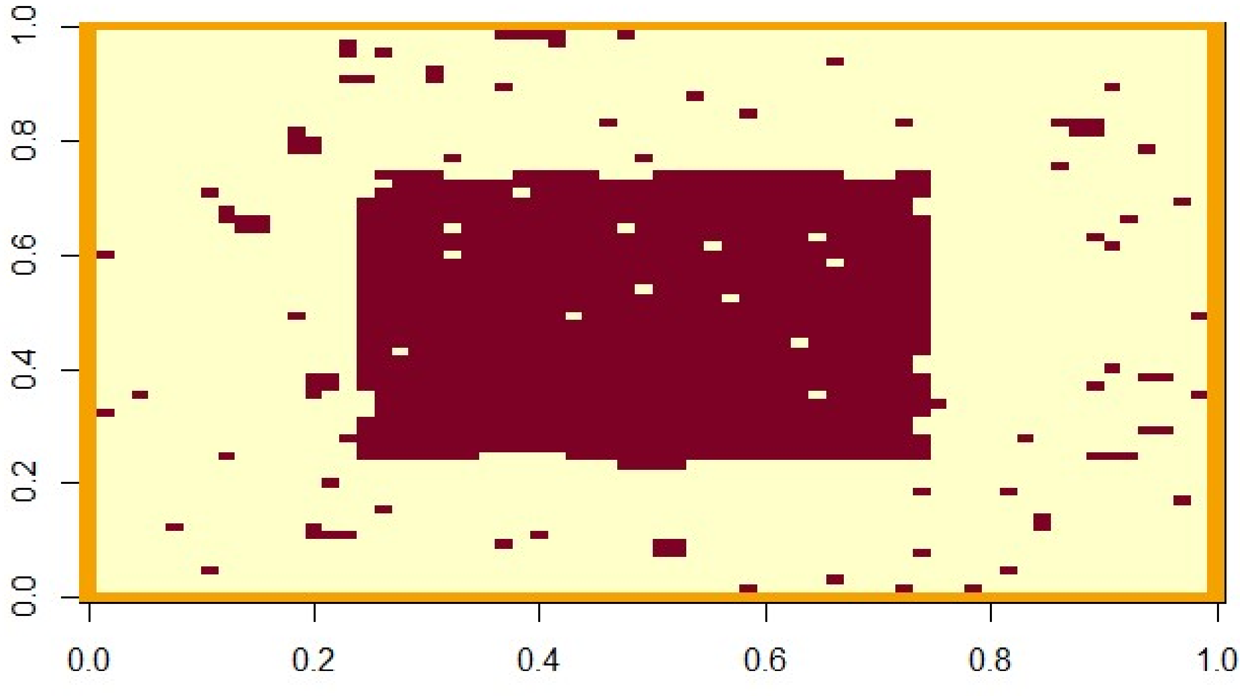
Algorithme de Gibbs (GibbsIsing)

Hide

```
#=====================================================================================
====================

GibbsIsing=function(forme, N, beta, B, alpha, n, p){
  #fonction qui réalise l'algorithme de Metropolis sur une image img0 qui est bruité
en img1, et doit à la fin être restaurée en img2

  #----création de l'image parfaite img0----
  img0 = imageGenerator(forme, N)
  image(img0)
  #----création d'une image auxiliaire img1, avec bord plus long et bruitage----
  img1 = matrix(0, nrow=N+2, ncol=N+2)
  bruit = matrix(2*rbinom(N^2,1,1-p)-1,nrow=N,ncol=N)
  img1[2:(N+1),2:(N+1)] = img0*bruit
  image(img1)
  img2 = img1 #image tampon
  img3 = img1

  for(k in 1:n){
    #----choix d'un site---
    i <- 1+ceiling(N*runif(1)) # indice entre 2 et N+1
    j <- 1+ceiling(N*runif(1))
    #iaux <- 2 + k%%N #de 2 à N+1
    #jaux <- 2 + k%/%(N-1)%%N #de 2 à N+1


    #----changement de valeur au site visé sur img2 (1ou-1) pour représenter E en ent
ier avec img1 et img2 (cas d'une image en noir et blanc)----
    img2[i,j] = -img1[i,j]
    #----voisinage----
    v = img1[i+1,j] + img1[i-1,j] + img1[i,j+1] + img1[i,j-1]
    #----potentiels----
    #U1 = -B*img1[i,j]-beta*img1[i,j]*v
    #U2 = -B*img2[i,j]-beta*img2[i,j]*v
    U1 = -B*img1[i,j]-img1[i,j]*(beta*v + alpha*img3[i,j])
    U2 = -B*img2[i,j]-img2[i,j]*(beta*v + alpha*img3[i,j])
    #----calcule de la proba conditionnelle locale----
    P <- exp(-U2)/(exp(-U1)+exp(-U2))
    #----test----
    if(runif(1)<= P){img1[i,j]= img2[i,j]}#img1 redevient identique à img2
    else{img2[i,j] <- img1[i,j]} #opération inverse, on ne retient pas le changement,
et img2 redevient identique à img1
  }#for
  #----affichage----
  #par(mfrow=c(1,3))
  image(img1)
}


GibbsIsing("rectangle", 64, 2, 0, 0.5, 10^4, 0.15)
```


0.0
0.2
0.4
0.6
0.8
1.0
0.0
0.2
0.4
0.6
0.8
1.0


0.0
0.2
0.4
0.6
0.8
1.0
0.0
0.2
0.4
0.6
0.8
1.0



Hide

```
#==========================================================================================
====================
```

Recuit simulé non bayésien

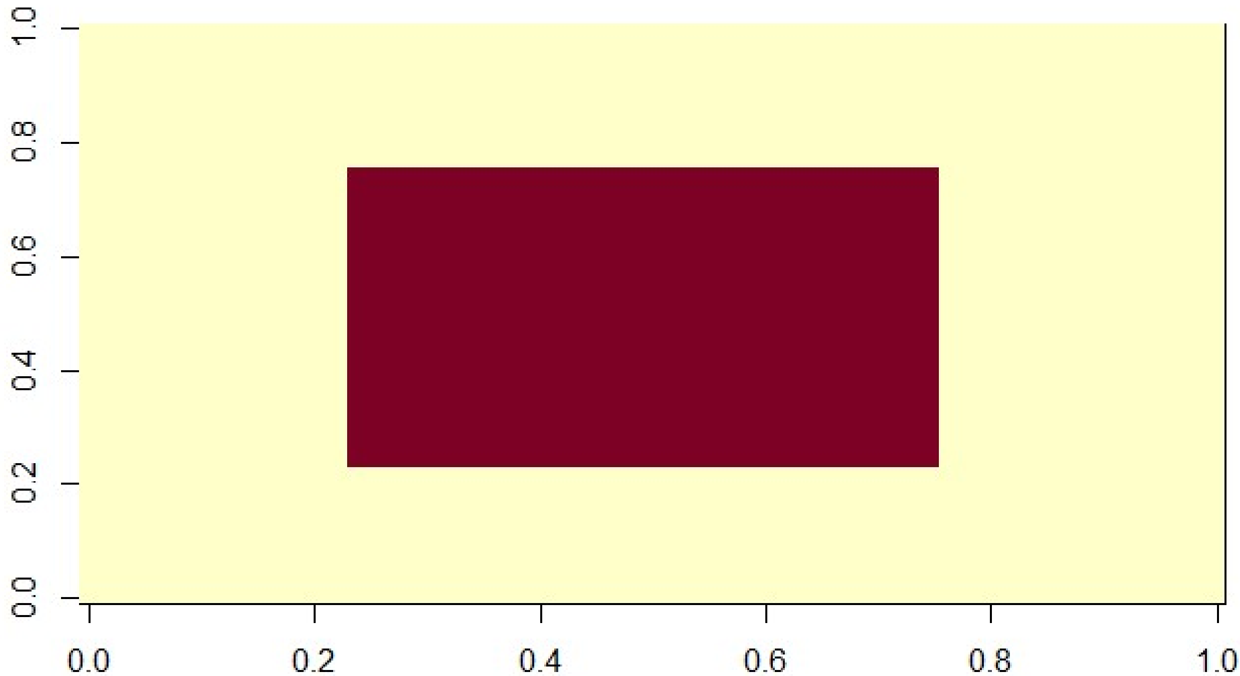
recuitMetro

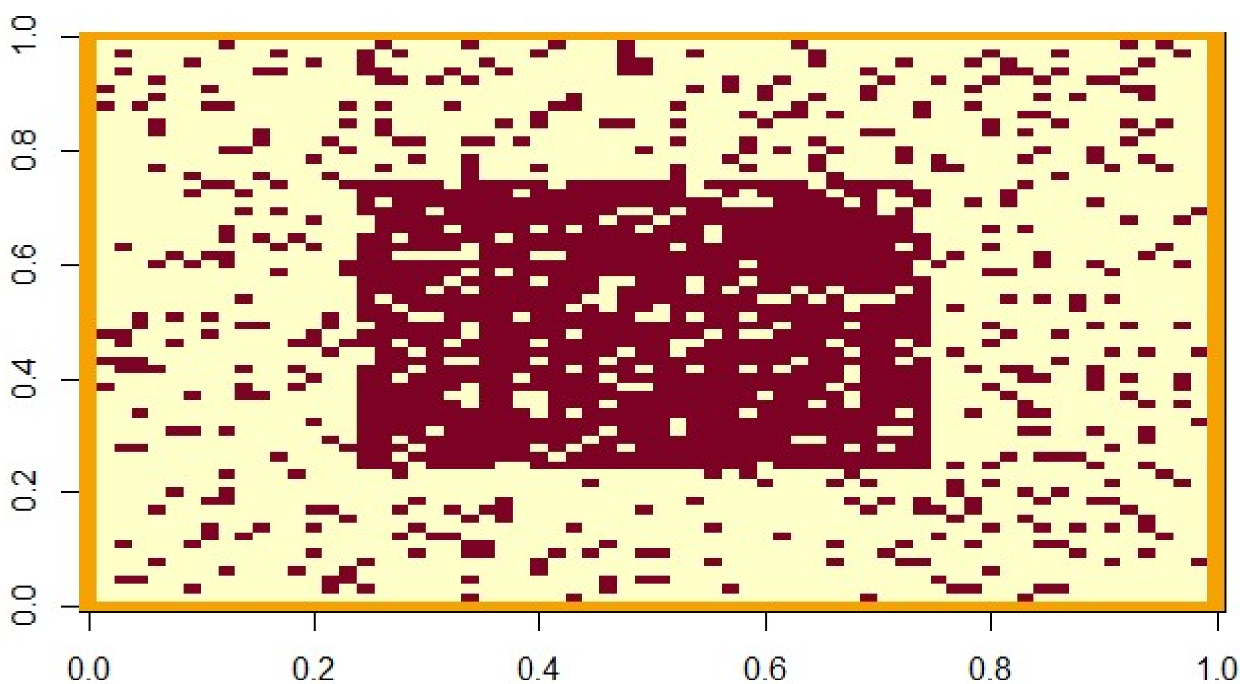
On réalise un recuit simulé en faisant lentement réduire les paramètres B, beta et alpha, par division successive par t la temperature, qui diminue lentement au cours des itérations.

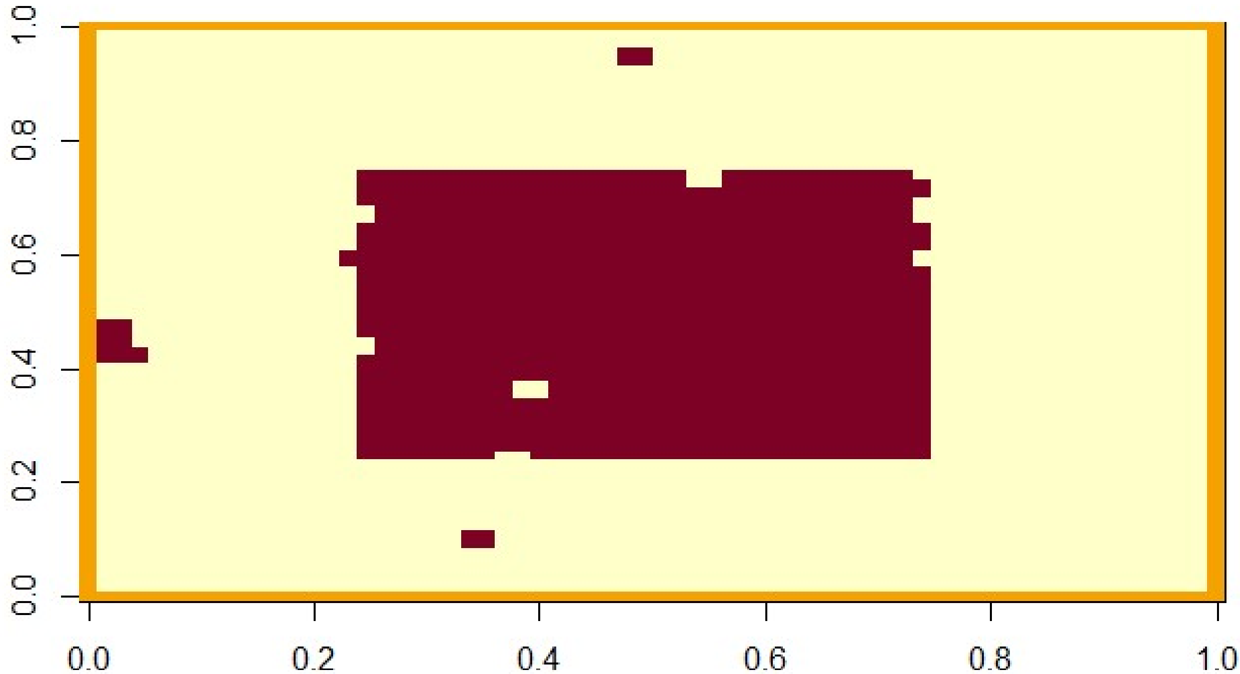
Hide

```
#========================================================================================
====================

recuitMetro=function(forme, N, beta, B, alpha, n, p){
  #fonction qui réalise l'algorithme de Metropolis sur une image img0 qui est bruité
en img1, et doit à la fin être restaurée en img2

  #----création de l'image parfaite img0----
  img0 = imageGenerator(forme, N)
  image(img0)
  #----création d'une image auxiliaire img1, avec bord plus long et bruitage----
  img1 = matrix(0, nrow=N+2, ncol=N+2)
  bruit = matrix(2*rbinom(N^2,1,1-p)-1,nrow=N,ncol=N)
  img1[2:(N+1),2:(N+1)] = img0*bruit
  image(img1)
  img2 = img1 #image tamponz
  img3 = img1 #mémoire du bruit initiale

  for(k in 1:n){
    t <- 1/k
    Br <- B/t
    betar <- beta/t
    alphar <- alpha/t
    #print(c(T,Br,betar,alphar))
    #----choix d'un site---
    i <- 1+ceiling(N*runif(1)) # indice entre 2 et N+1
    j <- 1+ceiling(N*runif(1))
    #i <- 2 + k%%N #de 2 à N+1
    #j <- 2 + k%/%(N-1)%%N #de 2 à N+1
    #----tirage nouvelle valeure candidate----
    img2[i,j] = -img1[i,j]
    #----voisinage----
    v = img1[i+1,j] + img1[i-1,j] + img1[i,j+1] + img1[i,j-1]
    #----potentiels----
    #U1 = -B*img1[i,j]-beta*img1[i,j]*v-alpha*img3[i,j]
    #U2 = -B*img2[i,j]-beta*img2[i,j]*v-alpha*img3[i,j]
    U1 = -Br*img1[i,j]-img1[i,j]*(betar*v + alphar*img3[i,j])#potentiel actuel
    U2 = -Br*img2[i,j]-img2[i,j]*(betar*v + alphar*img3[i,j])#potentiel candidat

    #----test----
    dU = U2-U1
    if(dU<0){img1[i,j]= -img1[i,j]}
    else{
      p <- exp(-dU)
      if(runif(1)<=p){img1[i,j]= -img1[i,j]}
      else{img2[i,j] <- img1[i,j]} #sinon on ne change pas
    }#else
  }#for
  #----affichage----
  #par(mfrow=c(1,3))
```

```
  image(img1)
}

recuitMetro("rectangle", 64, 1, 0, 0.3, 10^5, 0.15)
```







Hide

```
#==============================================================================
====================
```

méthodes bayésiennes

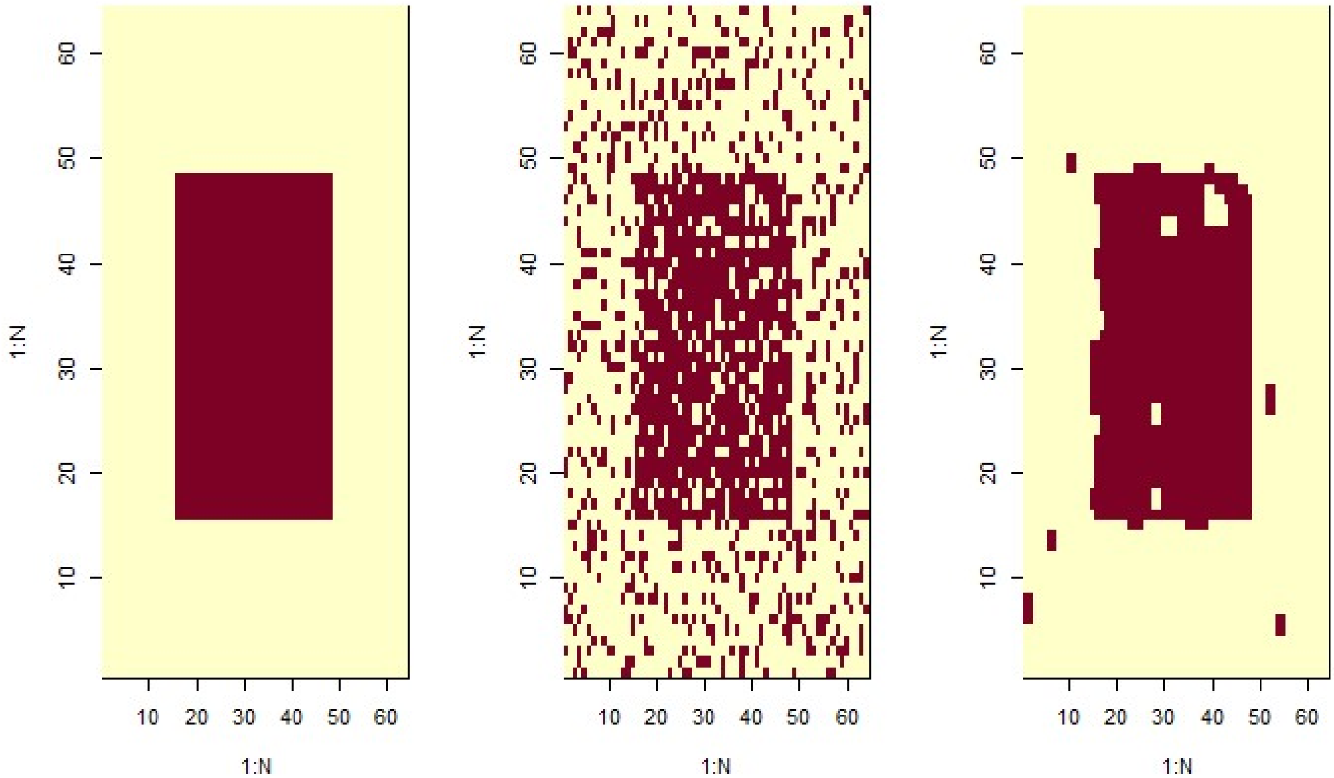
recuit simulé de Guyader

Hide

```
#=====================================================================================
====================


recuitSimule = function(n, beta, N, p, alpha, stop){

  #----création de sigma0 le rectangle initial----
  sigma0 <- matrix(-1,nrow=N,ncol=N)
  sigma0[(N/4):(N*3/4),(N/4):(N*3/4)] <- 1

  #----création de sigma le rectangle bruité----
  bruit <- matrix(2*rbinom(N^2,1,1-p)-1,nrow=N,ncol=N)
  sigma <- sigma0*bruit

  #----création image auxiliaire, avec contour suplémentaire pour calculer U plus fac
ilement----
  Saux <- matrix(0, nrow=N+2, ncol=N+2)
  Saux[2:(N+1),2:(N+1)] <- sigma
  sigmaaux <- Saux #sugmaaux = matrice auxiliaire pour le calcule de U, garde la vale
ur initiale de Saux (qui lui change)


  #--------le recuit simulé--------
  if(stop==0){ #cas où on s'arrête simplement après n itérations
    for (k in (1:n)) {
      #T <- 1/sqrt(sqrt((1+log(k))))
      T <- 1/k #T tend lentement vers 0
      iaux <- 1+ceiling(N*runif(1)) # indice entre 2 et N+1
      jaux <- 1+ceiling(N*runif(1))
      #iaux <- 2 + k%%N #de 2 à N+1
      #jaux <- 2 + k%/%(N-1)%%N #de 2 à N+1


      #----voisinage----
      #s <- Vstar(Saux, N, iaux, jaux)
      s <- Saux[iaux-1,jaux]+Saux[iaux+1,jaux]+Saux[iaux,jaux-1]+Saux[iaux,jaux+1]
      #print("s")
      #print(s)
      #----proba pour un recuit simulé, avec rapport de métropolis----
      r <- exp(-2*Saux[iaux,jaux]*(2*alpha*T*sigmaaux[iaux,jaux]+beta*s)/T)

      #----changement ou non de spin selon la proba r---
      if(runif(1)<r){Saux[iaux,jaux] <- -Saux[iaux,jaux]}
    }#for
  }#if

  if(stop==1){#cas ou on s'arrête après n itérations sans changement de spin
    X<-0#nombre d'itérations consécutives sans changement de spin
    it<-0
    while(X<=n) {
```

```
      it<-it+1
      #T <- 1/sqrt(sqrt((1+log(k))))
      T <- 1/k #T tend lentement vers 0
      iaux <- 1+ceiling(N*runif(1)) # indice entre 2 et N+1
      jaux <- 1+ceiling(N*runif(1))
      #iaux <- 2 + k%%N #de 2 à N+1
      #jaux <- 2 + k%/%(N-1)%%N #de 2 à N+1


      #----voisinage----
      s <- Saux[iaux-1,jaux]+Saux[iaux+1,jaux]+Saux[iaux,jaux-1]+Saux[iaux,jaux+1]

      #----proba pour un recuit simulé----
      r <- exp(-2*Saux[iaux,jaux]*(2*alpha*T*sigmaaux[iaux,jaux]+beta*s)/T)

      #----changement ou non de spin selon la proba r---
      if(runif(1)<r){
        Saux[iaux,jaux] <- -Saux[iaux,jaux]
        X <- 0#on vient de changer un spin, donc la condition d'arrêt retombe à 0
      }#if
      else{X <-X+1 }
    }#while
    print(it)
  }#if

  S <- Saux[2:(N+1),2:(N+1)] #image finale (on retire le contour auxiliaire)


  #----pourcentage de correspondance----
  nok <- 0 #nombre de pixels divergents
  correspondance <- 0 #pourcentage sur nb pixels total
  for (i in (1:N)){
    for (j in (1:N)){
      if(S[i,j] != sigma0[i,j]){nok <- nok+1}
    }#for
  }#for
  correspondance <- ((N^2)-nok)/(N^2)*100
  #print(nok)
  #print(correspondance)


  #----affichage----
  par(mfrow=c(1,3))
  image(1:N,1:N,sigma0)
  image(1:N,1:N,sigma)
  image(1:N,1:N,S)
  return(correspondance)
}#function


recuitSimule(10^5, 5, 64, 0.2, 3, 0)
```

```
[1] 97.63184
```



Hide

```
#===================================================================================
====================
```

Les estimateurs

> une autre approche pour déterminer la probabilité de X et Xs, c'est de travailler avec la notion de coût. On considère ainsi le coût représenter par le change d'un état par rapport à l'état précédent.

Estimateur MAP

$L(x, x') = 1 \text{ pour } x \neq x', \text{et } 0 \text{ sinon}$
$E[L(X, \phi(y))|y] = 1 - P(X = \phi(y)|y)$

estimateur MPM

estimateur TPM

Adaptabilité du calcul de potentiel

voisinage en étoile

On va essayer d’augmenter la précision des algorithmes précédents en jouant sur la définition du voisinage. On définit le voisinage possible comme une étoile à huit branches. La longueur de chaque branche (et donc son poid dans le calcul de U) va dépendre de la continuité des pixels constituant cette branche. Ainsi d’un pixel à l’autre on n’aura pas forcément un voisinage de même ampleur. La longeur d’un branche est égale au nombre de pixels côte à côte suivant l’orientation de la branche, de même valeur (pour une image en noir et blanc), partant de s. Le poid d’une branche de voisinage est alors proportionel à sa longeur. On aurait: $U_s = Bx_s + \beta x_s \sum_{i=1}^{8} a_i l_i$ avec (l_i) les 8 lead et (a_i) leur longueur.

On peut choisir laune longuer maximal pour les branches, avec comme longuer infini simplement une longeur supérieure ou égale à la largeur de l’image.

constitution du voisinage

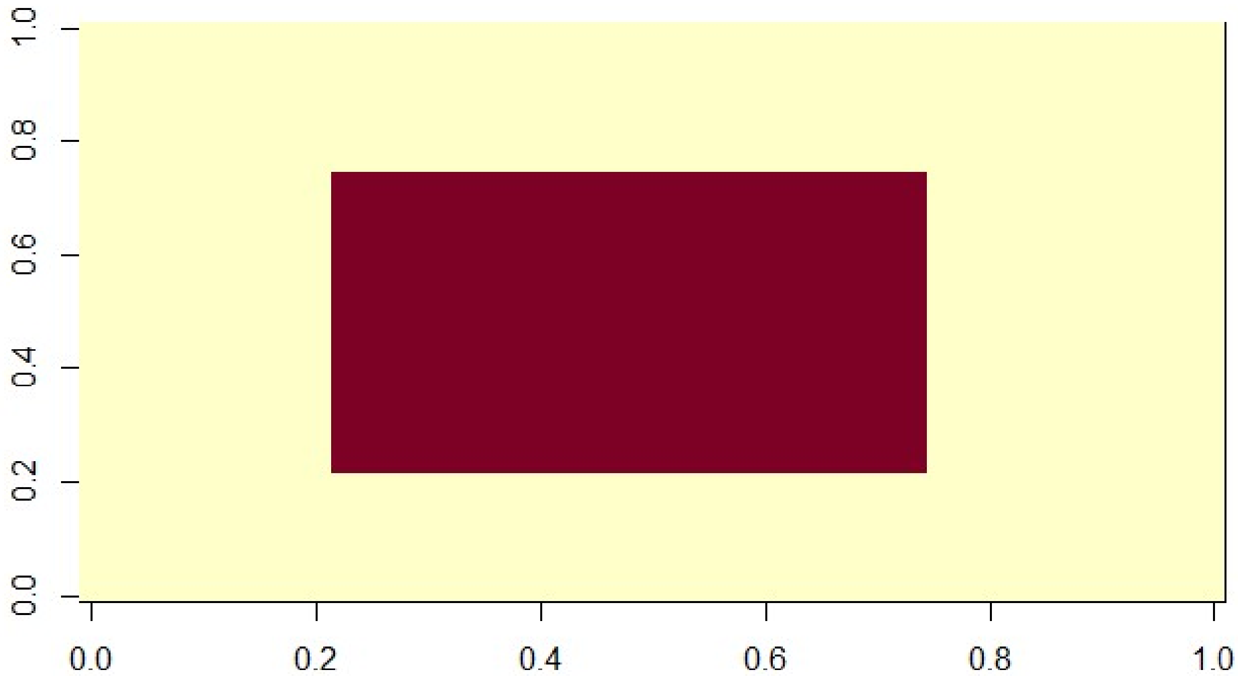
1. ajout des 4 plus proches voisins (habituel). On les note lead.
2. pour chacun des lead, on regarde la valeur du pixel qui le jouxe dans l’orientation de la flêche auquel il appartient, opposément à s biensûr. Si ce pixel est de même valeur que le lead, on l’intègre à la branche. Sinon, la branche est terminée (cas d’une image en noir et blanc).

Vbranch, Vstar

Image pour tester les fonctions Vbranch() et Vstar

Hide

```
img <- imageGenerator("rectangle", 50)

imgaux <- matrix(data=0,nrow=52,ncol=52)
imgaux[2:51,2:51] <- img
image(img)
```



Hide

```
#Vstar(imgaux, 50, 2, 2)
```

Vbranch()

Fonction qui calcule le poid d’une seule branche, d’orientation donnée.

Hide

```
#=====================================================================================
====================
vbranch=function(img, N, is, js, iv, jv, L){
  #la taille de l'image N doit être précisée (à remplacer par méthode info image)
  #L détermine la longueur maximale d'une branche
  #(iv, jv) appartient à [(0,1),(0,-1),(1,0),(-1,0)] ce qui donne le sens de la branc
he à partir de s
  v = c()
  vl = c()
  if(is<1 | is>N | js<1 | js>N){return("les coordonnées données sont en dehors de l'i
mage")}

  #----récupération des L sites de la branche potentielles sans condition----
  for(k in(1:L)){
    i <- is+k*iv
    j <- js+k*jv
    if(i>=1 && i<=N){ if(j>=1 && j<=N){ #teste si les coordonnées du candidat sont da
ns l'image
      vl <- c(vl,img[i,j])
    }}#if&if
  }#for
  #print(vl)
  #print(length(vl))
  if(is.null(vl[1])){return(0)}

  #----épuration selon la condition de valeur de site----
  lead <- vl[1]
  for(k in 1:length(vl)){ #parcours les L sites
    if(vl[k]==lead){  v <- c(v,vl[k])  }#condition pour faire partie de la branche.
  }#for

  #print(v)
  #print(length(v))
  return(length(v)*lead) #"poid" de la branche. length(v) fait office de facteur, com
me beta classiquement
}#func

#vbranch(img, 50, 1,1, 0, -1, 50 )
#=====================================================================================
====================
```

Vstar()

Fonction qui calcule le voisinage d'un site, selon le modèle en étoile à quatre branches.

Hide

```
Vstar=function(img, N, i, j, L){
  s <- 0
  s <- s + vbranch(img, N, i, j, -1, 0, L)
  s <- s + vbranch(img, N, i, j, 1, 0, L)
  s <- s + vbranch(img, N, i, j, 0, -1, L)
  s <- s + vbranch(img, N, i, j, 0, 1, L)
  return(s)
}#func

#Vstar(img, 50, 50,2 , 5)
#=====================================================================================
====================
```

comparaison des performances

On regarde comment évolue les performance de restauration en fonction de la taille du voisinage.

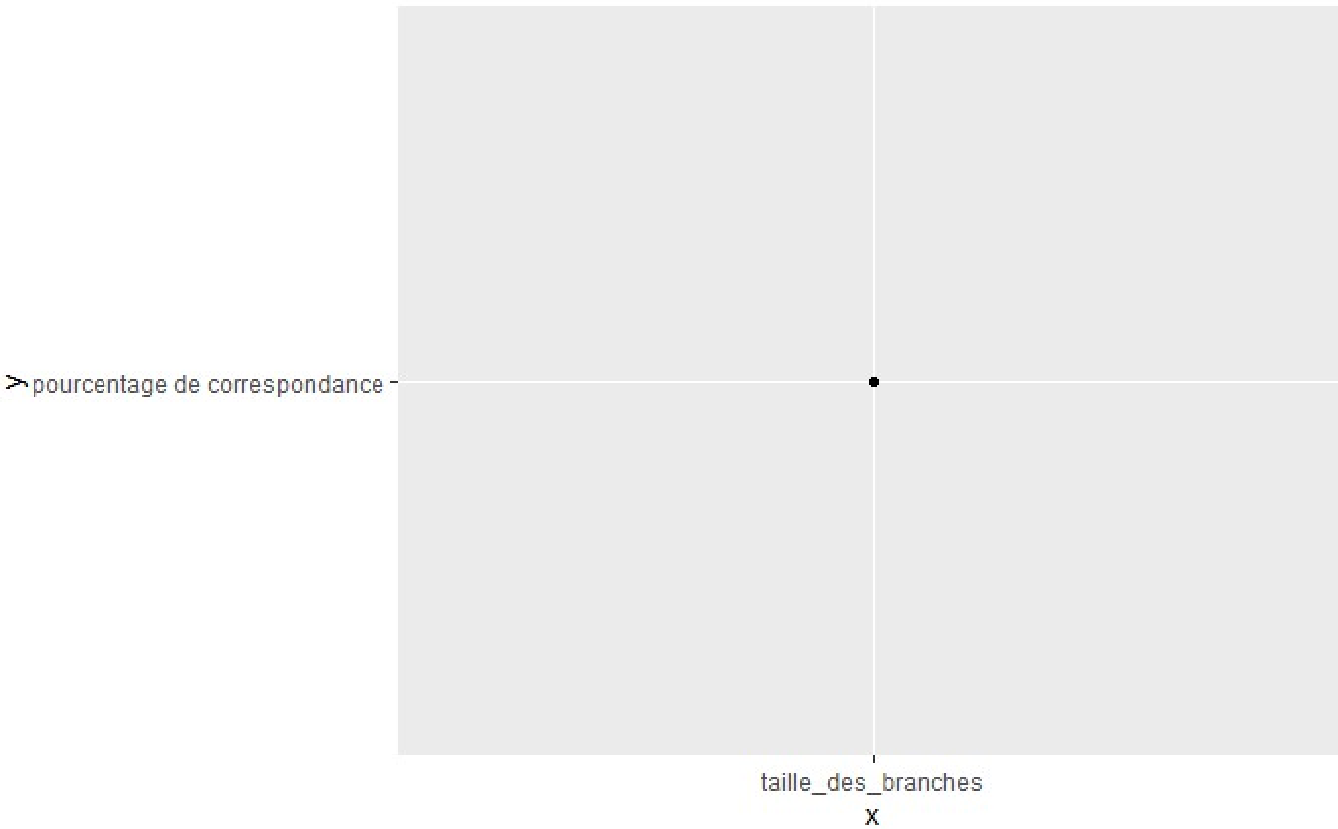
Hide

```
#=====================================================================================
====================

MCdeMC_Vstar=function(forme, N, beta, B, alpha, L, n, p){
  corresp = c(1,L)
  longueur = c(1:L)
  for( l in(1:L)){ corresp[l] = metropolisIsing(forme, N, beta, B, alpha, l, n, p) }#
for
  rev <- data.frame(longeur = c(1:L), correspondance = corresp )
  print(rev)
  ggplot(data = rev, aes(x="taille_des_branches", y="pourcentage de correspondance"))
+ geom_point() + geom_smooth() #+ geom_line()
}#func


MCdeMC_Vstar("rectangle", 64, 1, 0, 1, 20, 10^4, 0.15)
```

longeur <int>	correspondance <dbl>
1	91.79688
2	92.28516
3	92.84668
4	92.70020
5	92.23633
6	92.50488
7	92.67578

longeur <int>	correspondance <dbl>
8	93.11523
9	92.38281
10	92.65137

1-10 of 20 rows Previous **1** 2 Next



Hide

```
#=====================================================================================
====================
```

Il semble qu’une augmentation même faible de la longueur des branches permette déjà d’améliorer significativement le nombre de pixels bien restitués, mais que l’on ne peut pas augmenter encore ce nombre par un alongement de cette longeur de branche.

Probabilité du pourcentage de bon pixels bien représentés en sachant le nombre d’itérations

Fonction de répartition empirique du pourcentage

de pixels correctement restitué

> **remarque** On cherchera plutôt la probabilité de correspondance sachant le nombre de révolutions, plutôt que le nombre d'itération. Cela permet de s'afranchir de la taille de l'image. Par conséquent dans les algos on privélégiera les palaiyage déterministes, permettant d'assurer un balayage de tous les sites à chaque révolution.

> **hypothèse** à varifier/démontrer, mais, dans le cas où on a aucune information sur un pixel, ni sur ses voisin à priori, la proba de son état sur E suit la loi uniforme p=1/2.

> **remarque** Au niveau local avec un voisinage 2-connexité, on peut travailler avec l'espression de la mesure de Gibbs $\frac{1}{Z}exp(-U(x))$. Le voisinage d'un site ne peut avoir que $2^4 = 16$ configurations possibles. Et comme la configuration du voisinage conditionne directement l'état du site considéré, connaître la proba de Vs revient à connaître la proba de xs.

> **remarque** Il est surement plus simple de commencer à chercher à modéliser Nn après une première révolution. En effet on sait alors que la valeur en chaque site a été le fruit de l'algo, on dispose donc d'informations à posteriori sur ce site. Avant le première révolution, on ne peut pas prédire des valeurs des sites.

Une question intermédiaire: *quelle est la probabilité P_s que l'algorithme du recuit restitue correctement 1 pixel après un balayage ?* On pourra alors supposer que la proba qu'il restitue correctement l'image après 1 révolution est égale à P_s^N , soit le produit des proba locales de bonne restitution.

Probabiblité P_s de bonne restitution de s après N révolutions

Avant le balayage: x_s est correct, ou non (selon le bruitage) Après un balayage: x_s est correct (l'algo a fait le bon changement), ou non
Après N balayages: x_s est correct ou non (il est tout à fait possible que l'algo vienne à modifier faussement x_s à un N donné)
Quelle est la proba que l'algo corrige corrige correctement x_s? (avec une correction nulle dans le cas ou xs est déjà bon)
Si on considère le cas de l'échantilloneur de Gibbs, à la révolution n-1, la proba a priori que l'algo affecte la bonne valeur à s à la n-ième révolution, c'est la proba à psotériori que la valeur effictevement affecté à s par l'algo soit la bonne, soit la mesure de Gibbs.

recuit simulé fixe

Légère modification du recuit simulé de Guyader, affin que que l'on donner en argument non plus le nombre d'intération totale, mais le nombre de révolutions (parcours de tous les sites de l'image une fois). Cela a impliqué pour simplifier le code que le parcours des sites de soit plus aléatoire, mais ici ligne par ligne.

Hide

```
#==========================================================================================
====================
recuitSimuleFixe = function(n, beta, N, p, alpha, stop){

  #----cr?ation de sigma0 le rectangle initial----
  sigma0 <- matrix(-1,nrow=N,ncol=N)
  sigma0[(N/4):(N*3/4),(N/4):(N*3/4)] <- 1

  #----cr?ation de sigma le rectangle bruit?----
  bruit <- matrix(2*rbinom(N^2,1,1-p)-1,nrow=N,ncol=N)
  sigma <- sigma0*bruit

  #----cr?ation image auxiliaire, avec contour supl?mentaire pour calculer U plus fac
ilement----
  Saux <- matrix(0, nrow=N+2, ncol=N+2)
  Saux[2:(N+1),2:(N+1)] <- sigma
  sigmaaux <- Saux #sugmaaux = matrice auxiliaire pour le calcule de U, garde la vale
ur initiale de Saux (qui lui change)


  #--------le recuit simul?--------
  if(stop==0){ #cas o? on s'arr?te simplement apr?s n it?rations
    for (k in (1:(N*N*n))) {#N*N*n = nombre d'itération nécessaires pour faire n révo
lutions
      #T <- 1/sqrt(sqrt((1+log(k))))
      T <- 1/k #T tend lentement vers 0
      #iaux <- 1+ceiling(N*runif(1)) # indice entre 2 et N+1
      #jaux <- 1+ceiling(N*runif(1))
      iaux <- 2 + k%%N #de 2 ? N+1
      jaux <- 2 + k%/%(N-1)%%N #de 2 ? N+1



      #----voisinage----
      #s <- Vstar(Saux, iaux, jaux)
      s <- Saux[iaux-1,jaux]+Saux[iaux+1,jaux]+Saux[iaux,jaux-1]+Saux[iaux,jaux+1]

      #----proba pour un recuit simul?----
      r <- exp(-2*Saux[iaux,jaux]*(2*alpha*T*sigmaaux[iaux,jaux]+beta*s)/T)

      #----changement ou non de spin selon la proba r---
      #rmc[k]=r
      if(runif(1)<r){Saux[iaux,jaux] <- -Saux[iaux,jaux]}
    }#for
  }#if

  if(stop==1){#cas ou on s'arr?te apr?s n it?rations sans changement de spin
    X<-0 #nombre d'it?rations cons?cutives sans changement de spin
    it<-0 #nb itération
    revo<-0 #nb de révolution
    while(X<=n) {
```

```
      it<-it+1
      revo <- it%/%N*N
      #T <- 1/sqrt(sqrt((1+log(k))))
      T <- 1/it #T tend lentement vers 0
      iaux <- 1+ceiling(N*runif(1)) # indice entre 2 et N+1
      jaux <- 1+ceiling(N*runif(1))
      #iaux <- 2 + k%%N #de 2 ? N+1
      #jaux <- 2 + k%/%(N-1)%%N #de 2 ? N+1


      #----voisinage----
      #s <- Vstar(Saux, iaux, jaux)
      s <- Saux[iaux-1,jaux]+Saux[iaux+1,jaux]+Saux[iaux,jaux-1]+Saux[iaux,jaux+1]

      #----proba pour un recuit simul?----
      r <- exp(-2*Saux[iaux,jaux]*(2*alpha*T*sigmaaux[iaux,jaux]+beta*s)/T)

      #----changement ou non de spin selon la proba r---
      #rmc[k]=r
      if(runif(1)<r){
        Saux[iaux,jaux] <- -Saux[iaux,jaux]
        X <- 0#on vient de changer un spin, donc la condition d'arr?t retombe ? 0
      }#if
      else{X <-X+1 }
    }#while
    print(c("ItÃ©rations",it))
  }#if

  S <- Saux[2:(N+1),2:(N+1)] #image finale (on retire le contour auxiliaire)


  #----pourcentage de correspondance----
  corresp <- correspondance(S, sigma0, N)
  return(corresp)

}#function


recuitSimuleFixe(10,2/3,64,0.2,0.3,0)
```

```
[1] 98.75488
```

Hide

```
#===================================================================================
====================
```

recuit simulé (guyader)

Hide

```
#=================================================================================
===================

recuitSimule2 = function(n, beta, N, p, alpha, stop){

  #----cr?ation de sigma0 le rectangle initial----
  sigma0 <- matrix(-1,nrow=N,ncol=N)
  sigma0[(N/4):(N*3/4),(N/4):(N*3/4)] <- 1

  #----cr?ation de sigma le rectangle bruit?----
  bruit <- matrix(2*rbinom(N^2,1,1-p)-1,nrow=N,ncol=N)
  sigma <- sigma0*bruit

  #----cr?ation image auxiliaire, avec contour supl?mentaire pour calculer U plus fac
ilement----
  Saux <- matrix(0, nrow=N+2, ncol=N+2)
  Saux[2:(N+1),2:(N+1)] <- sigma
  sigmaaux <- Saux #sugmaaux = matrice auxiliaire pour le calcule de U, garde la vale
ur initiale de Saux (qui lui change)
  #rmc=rep(1/2,n)


  #--------le recuit simul?--------
  if(stop==0){ #cas o? on s'arr?te simplement apr?s n it?rations
    for (k in (1:n)) {
      #T <- 1/sqrt(sqrt((1+log(k))))
      T <- 1/k #T tend lentement vers 0
      #iaux <- 1+ceiling(N*runif(1)) # indice entre 2 et N+1
      #jaux <- 1+ceiling(N*runif(1))
      iaux <- 2 + k%%N #de 2 ? N+1
      jaux <- 2 + k%/%(N-1)%%N #de 2 ? N+1


      #----voisinage----
      #☻s <- Vstar(Saux, iaux, jaux)
      s <- Saux[iaux-1,jaux]+Saux[iaux+1,jaux]+Saux[iaux,jaux-1]+Saux[iaux,jaux+1]

      #----proba pour un recuit simul?----
      r <- exp(-2*Saux[iaux,jaux]*(2*alpha*T*sigmaaux[iaux,jaux]+beta*s)/T)

      #----changement ou non de spin selon la proba r---
      #rmc[k]=r
      if(runif(1)<r){Saux[iaux,jaux] <- -Saux[iaux,jaux]}
    }#for
  }#if

  if(stop==1){#cas ou on s'arr?te apr?s n it?rations sans changement de spin
    X<-0#nombre d'it?rations cons?cutives sans changement de spin
    it<-0
    while(X<=n) {
```

```
      it<-it+1
      #T <- 1/sqrt(sqrt((1+log(k))))
      T <- 1/it #T tend lentement vers 0
      iaux <- 1+ceiling(N*runif(1)) # indice entre 2 et N+1
      jaux <- 1+ceiling(N*runif(1))
      #iaux <- 2 + k%%N #de 2 ? N+1
      #jaux <- 2 + k%/%(N-1)%%N #de 2 ? N+1


      #----voisinage----
      s <- Vstar(Saux, iaux, jaux)
      #s <- Saux[iaux-1,jaux]+Saux[iaux+1,jaux]+Saux[iaux,jaux-1]+Saux[iaux,jaux+1]

      #----proba pour un recuit simul?----
      r <- exp(-2*Saux[iaux,jaux]*(2*alpha*T*sigmaaux[iaux,jaux]+beta*s)/T)

      #----changement ou non de spin selon la proba r---
      #rmc[k]=r
      if(runif(1)<r){
        Saux[iaux,jaux] <- -Saux[iaux,jaux]
        X <- 0#on vient de changer un spin, donc la condition d'arr?t retombe ? 0
      }#if
      else{X <-X+1 }
    }#while
    print(c("ItÃ©rations",it))
  }#if

  S <- Saux[2:(N+1),2:(N+1)] #image finale (on retire le contour auxiliaire)


  #----pourcentage de correspondance----
  nok <- 0 #nombre de pixels divergents
  correspondance <- 0 #pourcentage sur nb pixels total
  for (i in (1:N)){
    for (j in (1:N)){
      if(S[i,j] != sigma0[i,j]){nok <- nok+1}
    }#for
  }#for
  correspondance <- ((N^2)-nok)/(N^2)*100
  #print(nok)
  print(correspondance)


  #----affichage----
  #layout(matrix(c(1,4,2,5,3,0), nc=3))
  #image(1:N,1:N,sigma0)
  #image(1:N,1:N,sigma)
  #image(1:N,1:N,S)
  return(correspondance)
}#function
```

```
#=====================================================================================
====================
```

Montecarlo de Montecarlo (MCdeMC)

Hide

```
#=====================================================================================
====================


MCdeMC=function(pct, K, n, beta, N, p, alpha, stop){
  #algo de montecarlo sur celui de recuit simul?, pour estimer la proba de correspodn
ance
  #? pct% d'une image reconstitu?e, en en fonction du nombre d'it?rations dans le rec
uitsimul?
  P <- 0
  E <- 0
  corresp = rep(0,K)#génère une suite de K zero, qui va stoquer no résults pour affic
her ensuite avec ecfd()
  for (k in (1:K)) { #on fait K recuitSimul?
    corresp[k] = recuitSimuleFixe(n, beta, N, p, alpha, stop)
    if(corresp[k]>=pct){E <- E+1} #si le pourcentage de correspondance convient ? la
condition pct
  }

  P <- E/K  #les recuitSimule sont a priori ind?pendants les uns des autres, et r?ali
s?s identiquement
  print(c("probabilité de correspondance = ",P))

  #survie=function(x){1-ecdf(corresp)(x)}
  plot(ecdf(corresp))
  #x <- seq(min(corresp)-4,100,0.1)
  #plot.Surv(corresp)
  #plot(x,survie(x),type="S")
}

#=====================================================================================
====================
```

Hide

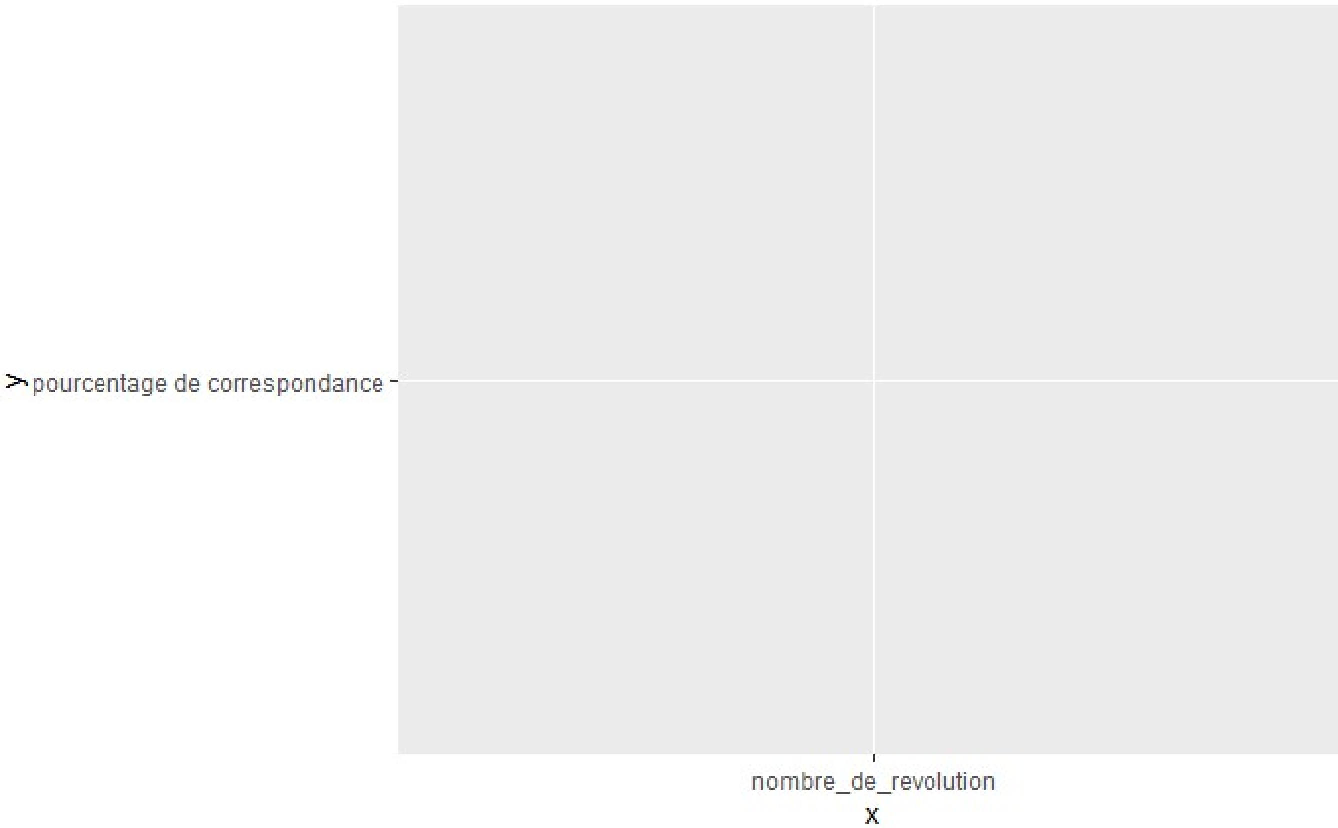
```
#MCdeMC(98,40,10,2/3,64,0.2,0.3,0)
```

Hide

```
MCdeMC_revo=function(n, beta, N, p, alpha, stop){
  corresp2 = c(1,n)
  revolution = c(1:n)
  for( revo in(1:n)){ #on va tester le recuit en faisant varier le nombre de révoluti
on de 1 à 50 (cas ou stop = 0)
    corresp2[revo] = recuitSimuleFixe(revo, beta, N, p, alpha, stop)
  }#for
  rev <- data.frame(revo = c(1:n), correspondance = corresp2 )
  print(rev)
  ggplot(data = rev, aes(x="nombre_de_revolution", y="pourcentage de correspondanc
e")) + geom_line()    # + geom_point()
    # + geom_smooth()
}#func

MCdeMC_revo(20, 2/3, 64, 0.2, 0.3, 0 )
```

revo <int>	correspondance <dbl>
1	95.45898
2	98.38867
3	97.85156
4	98.26660
5	98.65723
6	98.46191
7	98.68164
8	99.14551
9	98.97461
10	98.85254

1-10 of 20 rows Previous **1** 2 Next



Les paramètres et leur estimation

Modèle d’Ising

On rappelle, pour le modèle d’Ising, $E = -1, 1$, et $U(x_s) = \beta - x_s \sum_{t \in V_s} x_t - Bx_s$. Le choix du signe de beta est déterminant pour la valeur de $U_s(x_s)$. En effet, choisir un beta négatif va imposer que toutes les 2-cliques dont les pixels sont de signes différents auront pour potentiel $-1 \times 1 \times \beta$, donc un nombre positif, tandis que les 2-cliques de signes identiques auront un potentiel négatif. Or la mesure de Gibbs est telle que plus le potentiel pour un état est élevé, moins ce site est probable d’être dans cette état.

References