Projet techniques de simulation

Sommaire

Introduction

Ceci est le rapport écrit du projet de techniques de simulation soumis par **Mouad Sehli** et **Adrien Crastes**, le sujet avait pour but d'étudier l'impact de la modélisation de la météo sur les sinistres dans un modèle d'assurance moto.

Nous allons expliquer l'organisation du travail de groupe ainsi que du projet. Au début du projet nous avons discuté des différentes méthodes de simulation possible pour la loi de f, puis des fonctions demandées, nous avons après ceci constitué un code **mutualisé**, **sourcé**, et régulièrement soumis sur le site **dédié** pour pouvoir avoir la meilleur performance possible en prenant en compte le temps d'exécution calculé et leurs performances.

Ce mode de fonctionnement a permis l'élaboration de fonctions très performantes, notre projet est le fruit d'un effort de recherche et de travail.

0. Paramètrages (Retour au Sommaire)

Initialisation

on commence d'abord par initialiser les paramètres qui vont nous servir tout au long de l'étude

```
alpha <- 2
x0 <- 1
eta <- 3.1
N <- 1000
s <- 250
# Probabilités d'accident selon la météo
pacc <- c(1e-04, 2e-04, 5e-04)

# Matrice de transition de la chaine de Markov
ptrans2 <- c(0.3,0.2,0.3,0.4)
# soleil = 1 nuage = 2 pluie = 3
# Regrouper les paramètres dans une liste nommée 'params'
params <- list(
  alpha = alpha,
  x0 = x0,
  eta = eta,
  pacc = pacc,
  ptrans = ptrans2,
  N = N,
  s = s
)
```

Interpretation des paramètres

Nous avons la fonction de densité suivante $f^*(x) = exp(-\eta . log(\alpha + |x - x_0|))$

```
f<-function(x){
    return (ifelse(x>=0, exp(-eta*log(alpha + abs(x-x0))),0))
}
```

1. x_0 représente un point de référence par rapport auquel les écarts $|x - x_0|$ influencent la densité. Quand x s'éloigne de x_0, cela augmente le logarithme et le pic de la densité se déplace car $-\eta < 0$:
   - Cela signifie que les valeurs proches de x_0 sont **les moins probables**
   - **Dans un contexte d'assurance**, x_0 pourrait représenter une **franchise minimale** pour les montants de remboursements

   Démonstration :
   quand $x \to x_0$, la valeur absolue de $|x - x_0|$ décroît vers 0
   $\ln(\alpha + |x - x_0|)$ Donc décroît également. Par conséquent, le produit $-\eta \ln(\alpha + |x - x_0|)$ augmente car on a : $\eta > 0$
   Enfin, on a donc que $\exp(-\eta \ln(\alpha + |x - x_0|))$ augmente, ce qui augmente les chances que $f^*(x) > \epsilon$
2. α agit comme un terme d'ajustement pour la densité. Il déplace la contribution de $|x - x_0|$ dans le logarithme :
   - Si α est grand, même si l'écart $|x - x_0|$ est grand, son effet sera limité
   - Si α est petit, l'écart $|x - x_0|$ influence en majorité le logarithme

- **Dans un contexte d'assurance**, α pourrait représenter un **coût de base ou une pénalité** pour les montants de remboursements
  ce qui est logique car quand $x \to x_0$ on a $f^*(x) \to \alpha^{-\eta}$

3. η contrôle l'effet du logarithme sur la densité $f^*(x)$
   - Si η est élevé, le terme $-\eta.\, log(\alpha + |x - x_0|)$ devient plus petit, ce qui entraîne une décroissance plus rapide de la densité
   - η peut être vu comme un facteur de variabilité dans les remboursements. Un η élevé signifie que la distribution est très concentrée autour de x_0 et les valeurs éloignées sont peu probables

Intervalle pour les x

Problématique : tout ce qu'on sait à partir de l'énoncé, c'est que les x sont positifs, ce qui ne nous aide pas énormément vu que cela est à priori logique. Quand on rembourse un sinistre, ce n'est pas l'assuré qui nous paie. Il faut donc qu'on trouve un intervalle où sont compris la majorité des x.

> Pour trouver l'intervalle $[a; b]$ où $f^*(x)$ est le plus probable, on pose $\epsilon = 10^{-4}$ et on cherche l'intervalle pour lequel on a $f^*(x) > \epsilon$

$$\exp(-\eta \cdot \log(\alpha + |x - x_0|)) > \epsilon$$

$$-\eta \cdot \log(\alpha + |x - x_0|) > \log(\epsilon)$$

$$\log(\alpha + |x - x_0|) < -\frac{\log(\epsilon)}{\eta}.$$

$$|x - x_0| < \exp\left(-\frac{\log(\epsilon)}{\eta}\right) - \alpha.$$

$$x_0 - (\exp\left(-\frac{\log(\epsilon)}{\eta}\right) - \alpha) < x < x_0 + (\exp\left(-\frac{\log(\epsilon)}{\eta}\right) - \alpha).$$

En remplaçant les valeurs avec les paramètres que nous avons pris, on trouve $x \in [-16.5; 18.5]$. On exclut les valeurs négatives car on sait que $x > 0$, et donc on a $x \in [0; 18.5]$ pour vérifier.

```
Pourcentage <- integrate(f, lower = 0, upper = 18.5)$value / integrate(f, lower = 0, upper = Inf)$value

# Affichage du résultat
cat("Notre intervalle couvre donc :", Pourcentage * 100, "% des valeurs totales que peut prendre x ce qui est satisfaisan
t !")
```

```
## Notre intervalle couvre donc : 99.46729 % des valeurs totales que peut prendre x ce qui est satisfaisant !
```

```
## En tatonnant un peu nous avons remarqué que prendre l'intervalle [0,15] nous permettait de garder : 99.19265 % des val
eurs totales que peut prendre x. De plus cet intervalle permettait d'avoir des résulats qu'on jugeait meilleur.
```

1. Simulation de la loi de P_r (Retour au Sommaire)

Pour simuler la loi de P, plusieurs méthodes ont été testées. Tout d'abord, nous avons commencé par la méthode par inversion, en nous basant sur ce que nous avions vu en TD.

Première méthode : méthode par inversion

```
#Méthode par inversion
total<-integrate(f, -Inf, Inf)$value
frep.1 <-function (x) { #Fct de répartition
  if (x < 0) {
    return(0)
  }
  return (integrate(f, 0, x)$value / total)
}
frep<-Vectorize(frep.1)
g.1<-function (y){
  uniroot (function(u) frep(u)-y, c(0,1000), extendInt = "yes") $root
}
g<-Vectorize(g.1)

t<-system.time (x<-g(runif(1000)))
print (t)
```

```
##    user  system elapsed
##    2.05    0.03    3.11
```

Malheureusement, cette méthode a été vite écartée parce qu'elle prenait trop de temps au moment où nous avons essayé de la tester.

> Les méthodes de rejet par normale et mélange ne sont malheureusement pas possibles en raison de la structure de la fonction :
>
> - Pour pouvoir appliquer la méthode de rejet, il faut déterminer une constante c telle que
>
> $$c \cdot g(x) \geq f(x), \forall x$$
>
> dans l'ensemble de départ. Or, à cause de la forme logarithmique de la densité, cela est extrêmement difficile.
> - Pour appliquer la méthode par mélange, il faut que la densité de la fonction puisse être décomposée en une combinaison de distributions plus simples, par exemple des gaussiennes ou exponentielles pures. La présence du logarithme dans l'exponentielle complique cette approche.
>
> Naturellement, nous nous sommes donc tournés vers la deuxième méthode la plus naturelle et performante :

Deuxième méthode : fonction de répartition cumulative

Un autre candidat naturel était la fonction de répartition cumulative :

```
rremb <- function(n, params) {
  x_vals <- seq(0, 15, length.out = 600)  # Grille de valeurs pour x
  # Intégration totale
  total <- tryCatch(integrate(f, 0, Inf)$value, error = function(e) NA)
  if (is.na(total) || total == 0) stop("L'intégration de f a échoué ou total est nul.")

  # Calcul de la CDF
  cdf_vals <- sapply(x_vals, function(x) {
    tryCatch(integrate(f, 0, x)$value / total, error = function(e) NA)
  })

  if (any(is.na(cdf_vals))) stop("Les valeurs de cdf_vals contiennent des NA.")

  # Fonctions d'interpolation
  frep <- approxfun(x_vals, cdf_vals, rule = 2)
  g <- approxfun(cdf_vals, x_vals, rule = 2)

  # Génération des valeurs
  v <- g(runif(n))
  if (any(is.na(v))) stop("Des NA ont été générés dans les résultats.")
  return(v)
}
```

En faisant des recherches sur comment optimiser notre code, nous avons trouvé une méthode approchant celle par fonction de répartition cumulative qui est plus performante : celle des trapèzes (https://blogs.sas.com/content/iml/2011/06/01/the-trapezoidal-rule-of-integration.html). C'est après être tombés sur ce lien que nous avons approfondi nos recherches pour obtenir le code suivant.

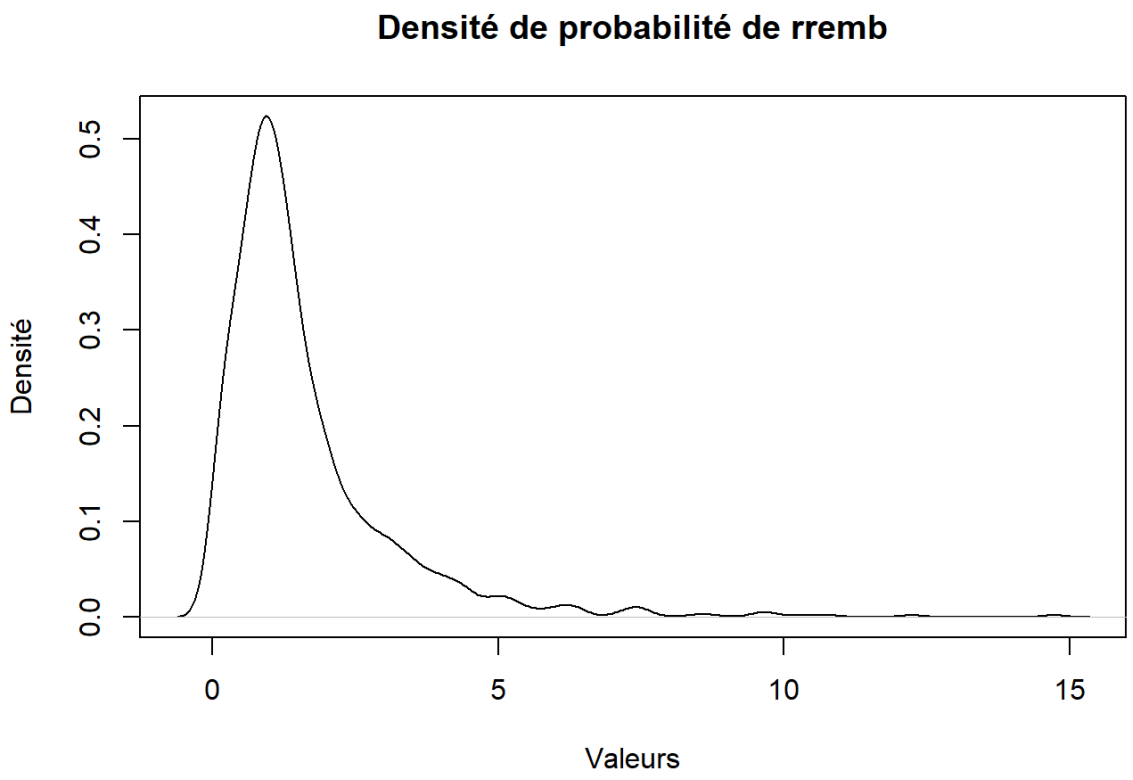
La méthode des trapèzes consiste à diviser l'intervalle d'intégration en plusieurs sous-intervalles, puis à approximer l'aire du trapèze sous la courbe de la fonction pour chaque sous-intervalle. Nous obtenons ainsi le code suivant :

```
rremb <- function(n, params) {
  alpha <- params$alpha
  x0 <- params$x0
  eta <- params$eta
  f <- function(x) {
    return(exp(-eta * log(alpha + abs(x - x0))))
  }
  x_vals <- seq(0, 15, length.out = 740)
  f_vals <- f(x_vals)
  dx <- diff(x_vals)
  cdf_vals <- cumsum((f_vals[-length(f_vals)] + f_vals[-1]) / 2 * dx)
  total <- cdf_vals[length(cdf_vals)]
  cdf_vals <- cdf_vals / total
  unique_indices <- which(!duplicated(cdf_vals))
  x_vals_unique <- x_vals[unique_indices]
  cdf_vals_unique <- cdf_vals[unique_indices]
  g <- approxfun(cdf_vals_unique, x_vals_unique, rule = 2)
  simulated_vals <- g(runif(n))
  return(simulated_vals)
}
```

la comparaison (le code en question) donne :

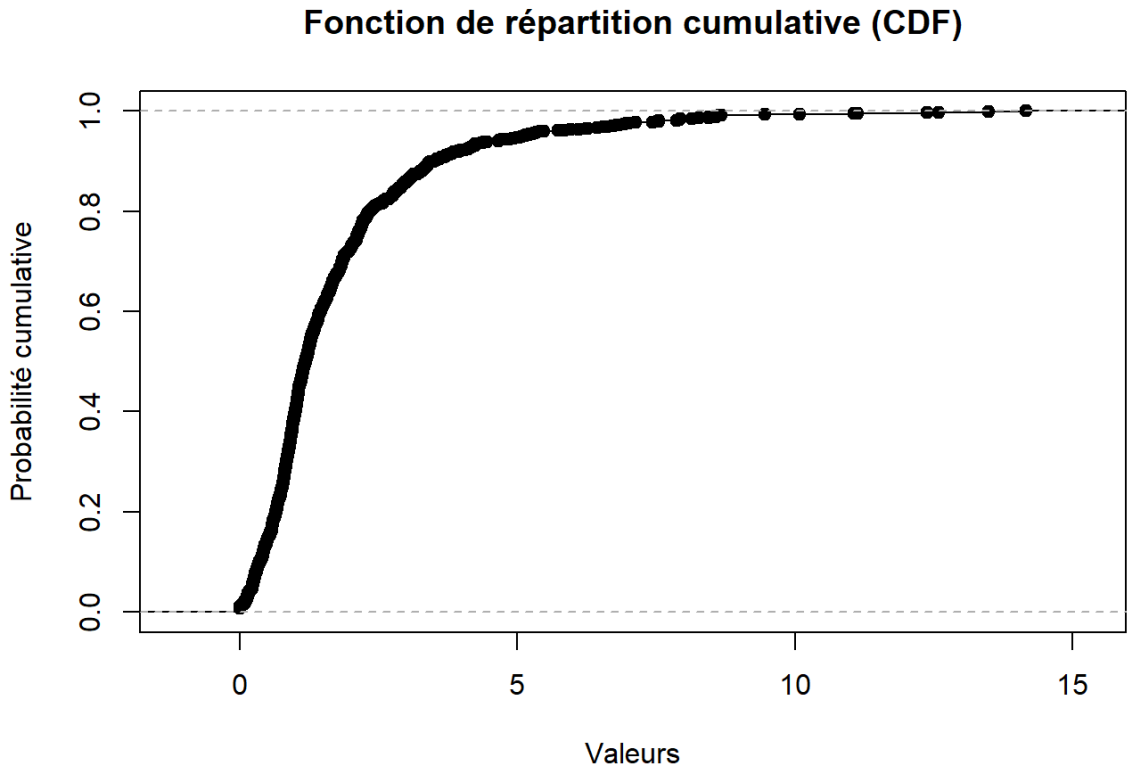
```
## Temps d'exécution de rremb1 qui utilise méthode des trapèzes : 2.17 secondes pour n=40 000 000
## Temps d'exécution de rremb2 qui utilise l'integration dans R : 2.11 secondes pour n=40 000 000
```

Code pour les fonctions



Ce graphique illustre la densité des valeurs simulées pour les remboursements. On observe une forte concentration autour de 1,7-1,8, indiquant que les petits montants sont plus probables et que la moyenne se situe probablement autour de cette valeur.

```
plot_cdf_rremb(1000, params)
```



La courbe de la fonction de répartition montre que la majorité des remboursements sont inférieurs à 5. La montée rapide initiale reflète les montants faibles très probables autour de la moyenne mentionnée précédemment.

Espérance des sinistres supérieurs à un seuil (Retour au Sommaire)

> Ici, nous devons calculer les deux valeurs suivantes :
>
> $$Msa = E(R - s|R > S)$$
>
> et
>
> $$Msb = E(R - s|R > S)$$
>
> , avec les hypothèses suivantes :
>
> - **Hypothèse A :** Tous les N motards ont la même météo. Cela facilite les calculs, car nous avons à simuler seulement **n.simul** météos de 365 jours.
> - **Hypothèse B :** Chaque motard a sa propre météo. Cela pose des difficultés techniques, car il faut simuler **n.simul** × **N** météos de 365 jours, ce qui fait rapidement monter la complexité.
>
> Nous avons essayé plusieurs approches que nous décrirons ci-dessous. Étant donné que la fonction est complexe et composée de plusieurs sous-fonctions différentes, nous avons utilisé la fonctionnalité **Rprof** au lieu de **system.time**. Celle-ci détaille explicitement quelles parties du programme prennent combien de temps. Nous nous intéresserons particulièrement au *sampling time* et au *by.self*.

Durant tout le projet, une seule fonction est restée la même : celle de la demi-largeur, que nous vous présentons ici. Il n'y a pas de particularité spécifique dans cette formule.

```
half_width <- function(values, cfdc = 0.95) {
  n <- length(values)
  mean_val <- mean(values)
  sd_val <- if (n > 1) sd(values) else 0
  a = qnorm((1 + cfdc) / 2)
  error_margin <- a * sd_val / sqrt(n)
  return(error_margin)
}
```

> Pour que le compte-rendu ne soit pas un bloc de code gigantesque et illisible, voici comment seront structurées les prochaines parties, car elles contiennent beaucoup de code. Tous les codes sont présents en annexe. Dans chaque paragraphe, je détaille une fonction ou une approche que nous avons adoptée ainsi que le temps d'exécution de chacune de ces fonctions pour les mêmes paramètres mentionnés au début.

2. Hypothèse A (Retour au Sommaire) (fonction MSA)

L'algorithme de notre fonction **msa** était le suivant :

- Générer la météo pour toutes les simulations.
- Générer tous les accidents survenus chaque jour pour chaque simulation.
- Faire la somme de tous ces accidents pour obtenir le nombre d'accidents par jour, puis refaire la somme pour obtenir le total par simulation.
- Simuler les montants remboursés, puis calculer la moyenne et la demi-largeur des simulations respectant les critères.

```r
# Transition chaine de markov
new_State<- function(current_state, ptrans) {
  transition_probs <- ptrans[current_state, ]
  next_state <- sample(1:3, size = 1, prob = transition_probs)
  return(next_state)
}
#Simulation de toutes les  météos
sim_Weather <- function(days, ptrans) {
  weather <- c(sample(1:3, size = 1))
  n_states <- 3
  for (i in 2:days) {
    weather[i] <- new_State(weather[i-1],ptrans)
  }
  return (weather)
}
#calcul des accidents survenus
accident_Matrix <- function(n.simul,  acc_probMatrix,N) {
  accidents_tot <- vector("list",n.simul)

  for (i in 1:n.simul){
    accidents_tot[[i]] <- matrix(
      rbinom(N *365, 1, rep(acc_probMatrix[i], each = N)),
      nrow = N,
      ncol = 365,
      byrow = TRUE
    )
  }
  return(accidents_tot)
}
msa1 <- function(n.simul,params){
  # Variables
  N <- params$N
  s <- params$s
  pacc <- params$pacc
  ## TRANSITION
  pSN <- params$ptrans[1]
  pNS <- params$ptrans[2]
  pNP <- params$ptrans[3]
  pPN <- params$ptrans[4]

  ptrans <-matrix(c(1 - pSN, pSN, 0.0,
                    pNS, 1- pNP - pNS, pNP,
                    0.0, pPN, 1-pPN), nrow = 3, byrow = TRUE)
  # initialisation vecteurs
  total_refund <- numeric(n.simul)
  weatherMatrix <- matrix(0, nrow = n.simul, ncol = 365)
  for (i in 1:n.simul) {
    weatherMatrix [i,] <- sim_Weather(365,ptrans)
  }
  acc_probMatrix <- matrix(pacc[weatherMatrix], nrow=n.simul, ncol=365)



  acc_total <- matrix(accident_Matrix(n.simul, acc_probMatrix, N))

  Nbr_acc <- sapply(acc_total, function(mat) sum(mat == 1))

  sinistre <- sapply(Nbr_acc, function(x) sum(rremb(x, params)))
  ms_values <- sinistre[sinistre > s] - s
  if (length(ms_values) == 0) {
    warning("Aucune valeur de R n'est supérieure à s.")
    return(list(ms = 0, demi.largeur = 0))
  }
```

```
  ms <- mean(ms_values)
  demi.largeur <- half_width(ms_values)
  return(list(
    ms = ms,
    demi.largeur = demi.largeur
  ))
}

Rprof("profiling.out")
d <- msa1(1000,params )
Rprof(NULL)

msa1_time <- list(summaryRprof("profiling.out")$by.self,summaryRprof("profiling.out")$sampling.time)
print(msa1_time)
```

```
## [[1]]
##                       self.time self.pct total.time total.pct
## "rbinom"                   8.82    73.38       8.82     73.38
## "matrix"                   0.64     5.32       9.48     78.87
## "FUN"                      0.64     5.32       0.94      7.82
## "sample.int"               0.36     3.00       0.96      7.99
## "stopifnot"                0.34     2.83       0.60      4.99
## "new_State"                0.30     2.50       1.38     11.48
## "sim_Weather"              0.20     1.66       1.58     13.14
## "...elt"                   0.18     1.50       0.20      1.66
## "sample"                   0.10     0.83       1.08      8.99
## "sum"                      0.10     0.83       0.10      0.83
## "diff.default"             0.06     0.50       0.06      0.50
## "anyNA"                    0.04     0.33       0.04      0.33
## "f"                        0.04     0.33       0.04      0.33
## "length"                   0.04     0.33       0.04      0.33
## "accident_Matrix"          0.02     0.17       9.48     78.87
## "rremb"                    0.02     0.17       0.20      1.66
## "diff"                     0.02     0.17       0.08      0.67
## "approxfun"                0.02     0.17       0.04      0.33
## "all"                      0.02     0.17       0.02      0.17
## "duplicated.default"       0.02     0.17       0.02      0.17
## "regularize.values"        0.02     0.17       0.02      0.17
## "w$get_new"                0.02     0.17       0.02      0.17
##
## [[2]]
## [1] 12.02
```

Comme vous pouvez le voir, nous obtenons le temps total d'exécution de la fonction ainsi que le pourcentage de temps utilisé par chaque fonction. Ce tableau est très important et a été très pertinent tout au long de notre travail, car il nous a permis de repérer, à chaque fois, quel était la fonction problématique.

Bien que l'on voit clairement que la fonction **rbinom** est la fonction la plus lourde, nous allons commencer par présenter d'abord celle de la simulation de météo dans ce rapport, pour deux raisons :

1. Une fois que nous avons trouvé une solution pour optimiser la fonction de génération des accidents, nous remarquons vite que la génération de météo est le deuxième processus à optimiser, car il prend également beaucoup de temps.
2. Il est impossible de tester la fonction de génération des accidents sans disposer d'un vecteur de météo valide.

Simulation météo

Nous avons essayé trois approches différentes avant d'être "satisfaits" du résultat :

- **Essai n°1 :** Nous avons défini une fonction qui calcule le nouvel état, puis nous l'appelons autant de fois que nécessaire dans la fonction qui retourne la météo sur 365 jours. Le premier état est choisi uniformément entre 1 et 3. (nous n'avions à ce stade pas remarqué que l'état initial est censé être le soleil) (cliquez ici pour voir le code)
- **Essai n°2 :** L'initialisation de l'état est tel X0 = Beau. La transition entre chaque état utilise la méthode cumulative directement à l'intérieur de la fonction, ce qui évite d'appeler à chaque fois la fonction "nouvel état". Nous remarquons (grâce au profilage) qu'une des fonctions "lourdes" de notre code est **runif**, appelée à plusieurs reprises. (cliquez ici pour voir le code)
- **Essai n°3 :** La troisième fonction génère en une seule fois un vecteur de 365 valeurs aléatoires pour tous les jours, contrairement aux deux premières fonctions où un nouveau nombre aléatoire est généré à chaque itération. (cliquez ici pour voir le code)

Voici les résultats des tests pour chacune de ces fonctions.

Si vous voulez voir le code qui a généré cette liste, vous pouvez cliquer ici.

```
## Le temps d'execution pour la fonction de météo numéro 1 pour n = 3000 est : 4.44 secondes
## Le temps d'execution pour la fonction de météo numéro 2 pour n = 3000 est : 1.14 secondes
## Le temps d'execution pour la fonction de météo numéro 3 pour n = 3000 est : 0.54 secondes
```

```
## Le 2ème code est 3.89 fois plus rapide que le 1er.
```

```
## Le 3ème code est 2.11 fois plus rapide que le 2ème.
```

```
## Le 3ème code est 8.22 fois plus rapide que le 1er.
```

Pour simuler la météo on avait donc décidé d'utiliser la fonction de l'essai 3 :

```
sim_Weather_3 <- function(days, ptrans) {
  rolls <- runif(days)
  # Weather vector
  weather <- numeric(days)
  weather[1] <- which.max(rolls[1] <= cumsum(ptrans[1, ]))
  for (i in 2:days) {
    current_state <- weather[i - 1]
    transition_probs <- ptrans[current_state, ]
    weather[i] <- which.max(rolls[i] <= cumsum(transition_probs))
  }

  return(weather)
}
#dans MSA plus tard pour avoir le vecteur des météos
#  for (i in 1:n.simul) {
#    weatherMatrix [i,] <- sim_Weather(365,ptranss)
#  }
```

une fois que cela était fait, on a défini une matrice de probabilité des accidents, qui contient la probabilité d'avoir un accident pour chaque jour:

```
# acc_probMatrix <- matrix(pacc[weatherMatrix], nrow=n.simul, ncol=365)
# Avec weatherMatrix, la matrice retournée par la fonction qui reitére la simulation de météo n.simul fois
```

- **Essai n°1 :** Notre premier code créait une liste où chaque élément est une matrice représentant les accidents pour une simulation (N motards par jour sur les 365 jours).
    1. Les accidents sont générés à l'aide de **rbinom**, qui simule des tirages binomiaux.
    2. Les probabilités pour chaque jour sont répétées pour chaque entité (N).
    3. Enfin, dans le code de msa, nous faisons la somme pour obtenir le nombre d'accidents par simulation.

  Le problème identifié est que cette fonction est très lente, car nous appelons **rbinom n.simul** fois et générons une matrice à chaque appel, ce qui devient coûteux à terme. (Cliquez ici pour voir le code ainsi que le test)

- **Essai n°2 :** Pour notre second code, nous avons essayé de simuler les accidents pour plusieurs simulations en utilisant une double boucle, ce qui réduisait la taille des matrices générées par **rbinom**.
    1. Initialisation d'une liste vide où nous stockons les résultats de chaque simulation.
    2. Boucle externe (i) : Pour chaque simulation, nous exécutons les étapes suivantes :
        - Initialiser un vecteur de 365 jours où seront stockés les accidents de chaque jour.
        - Boucle interne (j) : Pour chaque jour, générer un nombre d'accidents pour les N motards avec la probabilité associée.
        - Filtrer le vecteur **accidents_per_day** pour ne conserver que les jours avec des accidents, puis l'ajouter à la liste totale des accidents (réduisant ainsi le nombre d'éléments dans la liste, ce qui accélère le reste du programme).
        - Retourner le vecteur avec tous les accidents.

  Le problème identifié est que la boucle interne sur 365 jours est répétée pour chaque simulation, ce qui ralentit considérablement l'algorithme. De plus, chaque appel à **rbinom** est effectué individuellement, ce qui se traduit par **365 * n.simul** appels. (Cliquez ici pour voir le code ainsi que le test)

- **Essai n°3 (celui conservé) :** Pour notre troisième code, et version finale, nous avons remarqué que nous pouvions simplement simuler les accidents sur un long vecteur, puis réorganiser les résultats, ce qui évite d'appeler **rbinom** de manière itérative. Toutes les valeurs sont générées en une seule fois, puis réorganisées en une matrice.
    1. Initialisation d'une liste vide **accidents_tot** pour stocker les résultats.
    2. Simulation de tous les accidents en une seule opération vectorisée avec **rbinom**.
    3. Organisation des résultats dans une matrice de taille **(n.simul, 365)**.
    4. Calcul des sommes d'accidents directement en utilisant **rowSums** pour obtenir le total des accidents.
    5. Retour des totaux sous forme d'un vecteur.

  C'est ce code que nous avons décidé de conserver, car il est le plus performant. (Cliquez ici pour voir le code ainsi que le test)

Voici les résultats des tests pour les trois fonctions avec **n.simul = 1000** et **N = 1000** motards. On observe une nette amélioration entre les fonctions 1 et 2, malgré l'utilisation de deux boucles dans la deuxième fonction. Cependant, la troisième fonction, où toutes les probabilités sont générées en une seule fois avant d'être réorganisées, est de loin la plus efficace. Cliquez ici pour voir le code qui a généré cette liste

```
## Le temps d'execution pour la fonction de météo numéro 1 pour n = 1000, N = 1000 est : 9.50 secondes
## Le temps d'execution pour la fonction de météo numéro 2 pour n = 1000, N = 1000 est : 0.26 secondes
## Le temps d'execution pour la fonction de météo numéro 3 pour n = 1000, N = 1000 est : 0.02 secondes
```

```
## Le 2ème code est 36.54 fois plus rapide que le 1er.
```

```
## Le 3ème code est 13.00 fois plus rapide que le 2ème.
```

```
## Le 3ème code est 475.00 fois plus rapide que le 1er.
```

> Nous verrons plus tard qu'à la fin du développement de la fonction **msb** de l'hypothèse B, nous avons fait une découverte qui nous a permis de gagner énormément en efficacité. À ce stade du code, la fonction la plus rapide que nous avions, en utilisant la meilleure fonction pour la simulation de la météo et des accidents, avait le temps d'exécution suivant :

```
## Le temps d'execution pour pour n = 10000 est :  3.88 secondes
```

3 Hypothèse B (Retour au Sommaire) (fonction MSB)

> Pour l'hypothèse B, nous n'avons pas créé un grand nombre de fonctions, c'est pourquoi elles seront présentées directement ici. Nous n'avons pas beaucoup de fonctions car nous avons rapidement découvert une propriété qui nous a permis d'économiser énormément de temps.

Pour l'hypothèse B, le problème réside dans le fait que chaque motard a sa propre météo, ce qui entraîne rapidement la création d'une matrice très volumineuse. Notre première tentative consistait à utiliser une double boucle : pour chaque simulation, nous calculions la météo pour chaque motard et la placions dans la ligne associée à ce motard.

```
sim_Weather <- function(days, ptrans) {
  rolls <- runif(days)
  # Vecteur météo
  weather <- numeric(days)
  weather[1] <- which.max(rolls[1] <= cumsum(ptrans[1, ]))
  for (i in 2:days) {
    current_state <- weather[i - 1]
    transition_probs <- ptrans[current_state, ]
    weather[i] <- which.max(rolls[i] <= cumsum(transition_probs))
  }

  return(weather)
}

Rprof("profiling.out")
weather <- matrix(0, nrow = 100, ncol = 365 * N)
for (i in 1:100) {
  for (j in 1:N) {
    weather[i, ((j - 1) * 365 + 1):(j * 365)] <- sim_Weather(365, ptrans)
  }
}
Rprof(NULL)
Matrice <- summaryRprof("profiling.out")

cat("Pour n.simul = 100 et N = 1000 motards, le temps d'execution est ", Matrice$sampling.time, " secondes.\n")
```

```
## Pour n.simul = 100 et N = 1000 motards, le temps d'execution est  18.22  secondes.
```

Pour les accidents, nous avons suivi une approche similaire. Nous avons d'abord pris en entrée la matrice contenant les probabilités d'accidents pour chaque simulation, chaque jour, ainsi que le nombre de motards et de simulations. À l'aide de la fonction **rbinom**, nous avons initialisé un vecteur contenant tous les accidents. Ensuite, nous avons réorganisé ce

vecteur sous forme de matrice et effectué la somme sur les lignes de cette matrice, que nous avons retournée en sortie.

```
accident_MB <- function(n.simul, acc_probMatrix, N) {
  accidentstotal <- rbinom(length(acc_probMatrix), size = 1, prob = acc_probMatrix)
  result <- matrix(accidentstotal, nrow=n.simul,ncol=365*N)
  sums <- rowSums(result)
  return(sums)
}
```

On se retrouvait donc avec la fonction MSB suivante :

```
msb1 <- function(n.simul,params){
  # parameters
  N <- params$N
  s <- params$s
  pacc <- params$pacc
  ## Transition probabilities
  pSN <- params$ptrans[1]
  pNS <- params$ptrans[2]
  pNP <- params$ptrans[3]
  pPN <- params$ptrans[4]

  ptrans <-matrix(c(1 - pSN, pSN, 0.0,
                    pNS, 1- pNP - pNS, pNP,
                    0.0, pPN, 1-pPN), nrow = 3, byrow = TRUE)

  #Weather
  weather <- matrix(0, nrow = n.simul, ncol = 365 * N)
  for (i in 1:n.simul) {
    for (j in 1:N) {
      weather[i, ((j - 1) * 365 + 1):(j * 365)] <- sim_Weather(365, ptrans)
    }
  }
  #prob acc
  acc_prob <- pacc[weather]
  #accident total
  accident <- accident_MB(n.simul,acc_prob,N)

  sinistre <- sapply(accident, function(x) sum(rremb(x, params)))
  ms_values <- sinistre[sinistre > s] - s
  if (length(ms_values) == 0) {
    warning("Aucune valeur de R n'est supérieure à s.")
    return(list(ms = 0, demi.largeur = 0))
  }
  # mean + 95 % confidence
  ms <- mean(ms_values)
  demi.largeur <- half_width(ms_values)
  return(list(
    ms = ms,
    demi.largeur = demi.largeur
  ))
}
```

Mesure stationnaire

En revenant du contrôle continu de processus stochastiques et en cherchant un moyen d'optimiser **msb**, qui était extrêmement lente, je me suis rendu compte que le principal problème, que ce soit pour **msa** ou **msb**, était lié à la génération de la météo. L'état de chaque jour devait être généré jour par jour en suivant les probabilités de transition, ce qui rendait le code extrêmement lent dans les deux cas.

Cependant, je me suis rappelé d'une règle du cours de processus stochastiques :

1. Si la chaîne est irréductible,
2. Et qu'il y a un nombre fini d'états,

alors elle admet une **unique mesure stationnaire**.

De plus, si :

1. La chaîne est irréductible,
2. La chaîne est apériodique (ce qui est le cas ici car $P(S,S) > 0$, donc S est apériodique, et vu que la chaîne est irréductible, elle est également apériodique),
3. Et qu'elle admet une unique mesure stationnaire μ,

alors

$$X_n \xrightarrow{\mathcal{L}oi} \mu$$

J'ai donc défini la fonction ci-dessous pour calculer la mesure stationnaire, quels que soient les paramètres d'entrée.

```
measure_Stat <- function(ptrans) {
  P_trans <- t(ptrans)

  # Calcul des valeurs propres et vecteurs propres
  VP <- eigen(P_trans)

  # Identifier la valeur propre égale à 1
  VP_one <- which(abs(VP$values - 1) < 1e-10)


  stat_vec <- Re(VP$vectors[, VP_one])

  # Normaliser pour que la somme des éléments soit égale à 1
  measure_stat <- stat_vec / sum(stat_vec)

  return(measure_stat)
}
ptrans_stat <- measure_Stat(ptranss)
```

> Cela signifie qu'au lieu de générer la météo une par une, en prenant en compte l'état précédent, nous pouvions, en utilisant la mesure stationnaire, générer TOUS les états de la météo en une seule fois avec la fonction **sample**, puis les réordonner dans les dimensions souhaitées. Cette méthode est illustrée dans la fonction ci-dessous, qui est la version finale de génération de météo maintenue.
>
> Nous avons implémenté cette méthode pour **msa** et **msb**.

```
weatherMatrixA <- function(n.simul, ptrans_stat) {
  # Matrice 3D (0.s,365,1000)
  weather <- array(sample(1:3, n.simul * 365 , replace = TRUE, prob = ptrans_stat),
                   dim = c(n.simul, 365))
  return(weather)
}
```

```
## Le temps d'execution pour la fonction de météo numéro 1 pour n = 3000 est : 4.44 secondes
## Le temps d'execution pour la fonction de météo numéro 2 pour n = 3000 est : 1.14 secondes
## Le temps d'execution pour la fonction de météo numéro 3 pour n = 3000 est : 0.54 secondes
```

```
## Le temps d'execution pour la fonction de météo utilisant la mesure stationaire pour n = 300000 est : 1.58
##  secondes
```

On fait pareil pour msb

```
weatherMatrixB <- function(n.simul, ptrans_stat,N) {
  # Matrice 3D (0.s,365,1000)
  weather <- array(sample(1:3, n.simul * 365 * N , replace = TRUE, prob = ptrans_stat),
                   dim = c(n.simul, 365))
  return(weather)
}
```

> Malheureusement, à cause des limites de mémoire du site de programmation, cette méthode ne nous a pas été d'une grande utilité. Les vecteurs générés étant de très grande taille, nous ne pouvions pas dépasser un certain nombre de simulations (on a été limité à n = 300 pour msb). Nous avons tenté, en vain, de modifier notre fonction de météo à plusieurs reprises sans obtenir de résultats significatifs. Nous proposerons donc au manager un investissement en mémoire pour améliorer la performance et la précision de nos simulations !

Domaine de validité de la mesure stationnaire

Nous avons vu qu'en utilisant la chaîne de Markov stationnaire, nous obtenons une amélioration significative des performances. Cependant, il est nécessaire de vérifier si nous avons réellement le droit de l'utiliser. En effet, bien que la chaîne de Markov admette une mesure stationnaire, nous ne savons pas à partir de **quand** elle converge !

Pour cela, nous avons décidé de réaliser deux tests.

Test de convergence empirique

Nous allons procéder à deux tests empiriques. Dans le premier, nous générerons une longue chaîne de Markov, calculerons les fréquences de chaque occurrence telles que

$$f_i = \frac{|X_k = i, k \in [0, n]|}{n}$$

, et vérifierons dans quelle mesure cela se rapproche de la mesure stationnaire.

```
set.seed(170)
test_frequencies_empirical <- function(P, n_steps) {
  n_states <- nrow(P)
  states <- 1:n_states
  current_state <- 1
  state_counts <- numeric(n_states)
  for (i in 1:n_steps) {
    state_counts[current_state] <- state_counts[current_state] + 1
    current_state <- sample(states, 1, prob = P[current_state, ])
  }
  frequencies <- state_counts / sum(state_counts)
  return(frequencies)
}
cat("Fréquences empiriques pour n = 6000:", test_frequencies_empirical(ptranss,6000), "\n","Mesure stationnaire : ", meas
ure_Stat(ptranss) ,"\n")
```

```
## Fréquences empiriques pour n = 6000: 0.2718333 0.4176667 0.3105
##  Mesure stationnaire :  0.2758621 0.4137931 0.3103448
```

On observe que nous nous rapprochons beaucoup des fréquences empiriques. En effet, pour $n = 6000$, les différences sont déjà de l'ordre du centième de pourcent. Cependant, pour valider cela, nous allons procéder à un deuxième test.

Test du χ^2

Le test du χ^2 est une méthode statistique similaire au test empirique utilisant les fréquences calculées. Il consiste à poser les hypothèses suivantes :

- H_0 : La distribution empirique des états après N étapes correspond à la distribution stationnaire théorique π.
- H_1 : La distribution empirique des états après N étapes ne correspond pas à la distribution stationnaire théorique π.

Pour vérifier ces hypothèses, nous simulons N valeurs et calculons la statistique suivante :

$$\chi^2 = \sum_{i=1}^{3} \frac{(f_i - E_i)^2}{E_i}$$

où f_i représente les fréquences empiriques et E_i les fréquences théoriques attendues.

Ensuite, nous fixons un seuil $\alpha = 0.01$. Si nous trouvons $p < \alpha$, nous pouvons dire que l'hypothèse H_0 est acceptée avec un niveau de confiance de $1 - \alpha = 0.99$, soit 99 %.

Code de test pour le χ^2

```
## Chi-Squared Statistic: 236.1997
```

```
## p-value: 5.127225e-52
```

```
## Significant (alpha = 0.01 ): Yes
```

```
## Observed Counts: 613937 903155 672908
```

```
## Expected Counts: 604137.9 906206.9 679655.2
```

> On a vu que la mesure stationnaire est donc valable au moins à partir de $n > 6000$ de chaîne d'états simulés, ce qui équivaut pour l'hypothèse A à $n.simul_A \geq \lfloor \frac{6000}{365} \rfloor = 17$ et pour l'hypothèse B à $n.simul_B \geq \lfloor \frac{6000}{365*N} \rfloor = \lfloor \frac{6000}{365*1000} \rfloor = 1$. Dans les deux cas, cette condition est bien satisfaite, que ce soit sur le site de soumission sous les contraintes du site (https://isfa-srv3.univ-lyon1.fr/simulation/), ou bien sur nos machines.

> Nous allons comparer les performances des fonctions avec les mesures stationnaires contre les fonctions avec la mesure de base. Cliquez ici pour voir le code de comparaison
>
> ```
> ##
> ##  Comparaison entre msastat et msa pour n.simul = 20000, N = 1000 :
> ```

```
## Temps d'exécution de msastat : 6.13 secondes
## Temps d'exécution de msa1 : 11.35 secondes
## msastat est 185.15% plus rapide que msa1.
```

```
##
##  Comparaison entre msbstat et msb pour n.simul = 100, N = 1000 :
```

```
## Temps d'exécution de msbstat : 3.43 secondes
## Temps d'exécution de msb1 : 29.52 secondes
## msbstat est 860.64% plus rapide que msb1.
```

4. MSB.C (Retour au Sommaire)

Enfin, la 4ème et dernière fonction demandée était une fonction qui utilisait la bibliothèque Rcpp pour optimiser **msb** en utilisant C++ afin d'accélérer les fonctions considérées comme **lentes**.

Naturellement, pour pouvoir identifier quelles fonctions prennent le plus de temps, nous avons effectué un diagnostique avec **n = 600** (ceci sera le cas pour tous les tests suivants).

```
##                 self.time self.pct total.time total.pct
## "rbinom"             6.18    42.74       6.18     42.74
## "sample.int"         2.18    15.08       2.18     15.08
## "array"              1.86    12.86       4.72     32.64
## "aperm.default"      1.80    12.45       1.80     12.45
## "accident_MB2"       0.68     4.70       7.44     51.45
## "sample"             0.68     4.70       2.86     19.78
## "msbstat"            0.58     4.01      14.44     99.86
## "as.vector"          0.26     1.80       0.26      1.80
## "FUN"                0.08     0.55       0.22      1.52
## "f"                  0.04     0.28       0.04      0.28
## "rremb"              0.02     0.14       0.14      0.97
## "diff.default"       0.02     0.14       0.02      0.14
## "runif"              0.02     0.14       0.02      0.14
## "test_con"           0.02     0.14       0.02      0.14
## "unique.default"     0.02     0.14       0.02      0.14
## "which"              0.02     0.14       0.02      0.14
```

On peut directement remarquer que les fonctions qui prennent le plus de "temps" et qui ralentissent le plus notre programme sont les fonctions **rbinom** et **sample**, utilisées pour générer les accidents et la météo. Ce seront donc les fonctions que nous allons essayer d'implémenter en C++.

Fonction Binomiale en C++

Nous avons donc commencé par essayer de coder la fonction **sample** pour générer la météo en C++, ce qui nous donnait le code suivant :

```
library(Rcpp)
cppFunction('
#include <Rcpp.h>
#include <random>
using namespace Rcpp;

// [[Rcpp::export]]
IntegerVector rbinom_cpp(IntegerVector sizes, NumericVector probs) {
    int n = probs.size();
    IntegerVector results(n);

    std::mt19937 rng(std::random_device{}());  // Generateur de nombre aléatoire

    for (int i = 0; i < n; ++i) {
        std::bernoulli_distribution dist(probs[i]);
        results[i] = dist(rng);  // Genere le résultat binomiale
    }

    return results;
}
')
```

```
## Warning: No function found for Rcpp::export attribute at file62e0687178c8.cpp:5
```

Nous comparons notre nouvelle fonction avec la fonction **msbstat** que nous avons obtenue pour évaluer si ce changement améliore l'efficacité.

On constate que c'est le cas, donc nous décidons de maintenir ce changement.(Il faut regarder la différence entre **rbinom** et **rnibom_cpp** dans chaque profil) Code du test

```
## [1] "Profilling de msbstat pour n = 400 "
```

```
##                 self.time self.pct total.time total.pct
## "rbinom"             3.92    45.69       3.92     45.69
## "sample.int"         1.40    16.32       1.40     16.32
## "aperm.default"      1.04    12.12       1.04     12.12
## "array"              0.76     8.86       2.62     30.54
## "sample"             0.46     5.36       1.86     21.68
## "msbstat"            0.38     4.43       8.58    100.00
## "accident_MB2"       0.36     4.20       4.56     53.15
## "as.vector"          0.14     1.63       0.14      1.63
## "FUN"                0.06     0.70       0.12      1.40
## "diff.default"       0.04     0.47       0.04      0.47
## "approxfun"          0.02     0.23       0.02      0.23
```

```
## [1] "Profilling de msb.c1 pour n = 100"
```

```
##                       self.time self.pct total.time total.pct
## "sample.int"               1.40    22.51       1.40     22.51
## ".Call"                    0.98    15.76       0.98     15.76
## "array"                    0.94    15.11       2.78     44.69
## "aperm.default"            0.92    14.79       0.92     14.79
## "accident_MB2cppbinom"     0.50     8.04       2.16     34.73
## "sample"                   0.44     7.07       1.84     29.58
## "msb.c1"                   0.40     6.43       6.20     99.68
## "rbinom_cpp"               0.32     5.14       1.30     20.90
## "as.vector"                0.12     1.93       0.12      1.93
## "FUN"                      0.10     1.61       0.18      2.89
## "rremb"                    0.02     0.32       0.08      1.29
## ".approxfun"               0.02     0.32       0.02      0.32
## "diff.default"             0.02     0.32       0.02      0.32
## "exists"                   0.02     0.32       0.02      0.32
## "f"                        0.02     0.32       0.02      0.32
```

Météo en C++

Nous avons ensuite codé la fonction **accidents**, qui intègre la fonction **sample** (considérée comme gourmande en temps), en C++. Le code correspondant est présenté ci-dessous :

```
cppFunction('
#include <Rcpp.h>
#include <random>
using namespace Rcpp;

// [[Rcpp::export]]
IntegerVector sample_cpp(int n, IntegerVector values, NumericVector probs) {
    std::vector<int> results(n);
    std::mt19937 rng(std::random_device{}());
    std::discrete_distribution<> dist(probs.begin(), probs.end());

    for (int i = 0; i < n; ++i) {
        results[i] = values[dist(rng)];
    }

    return wrap(results);
}
')
```

```
## Warning: No function found for Rcpp::export attribute at file62e01dd34c9e.cpp:5
```

Nous procédons de la même manière : nous calculons et comparons les temps d’exécution des deux versions pour vérifier si ce changement améliore l’efficacité.

À noter : le code utilisé pour toutes les comparaisons en C++ reste le même que celui de la fonction précédente. Seules les fonctions comparées changent, et nous ne le répéterons donc pas ici.

```
## [1] "Profilling de msb.c2 pour n = 400"
```

```
##                       self.time self.pct total.time total.pct
## ".Call"                    2.26    33.33       2.26     33.33
## "rbinom_cpp"               1.34    19.76       1.34     19.76
## "aperm.default"            0.94    13.86       0.94     13.86
## "array"                    0.92    13.57       0.92     13.57
## "accident_MB2cppbinom"     0.64     9.44       2.32     34.22
## "msb.c2"                   0.36     5.31       6.78    100.00
## "findCenvVar"              0.12     1.77       0.12      1.77
## "as.vector"                0.10     1.47       0.12      1.77
## "FUN"                      0.02     0.29       0.10      1.47
## "rremb"                    0.02     0.29       0.08      1.18
## "approxfun"                0.02     0.29       0.04      0.59
## "as.list.default"          0.02     0.29       0.02      0.29
## "c"                        0.02     0.29       0.02      0.29
```

On observe le problème suivant : la fonction **.Call** est apparue et prend un temps considérable dans l'exécution de notre fonction.

> En approfondissant nos recherches, nous avons découvert que ce problème est dû à l'utilisation de **std::vector<int>** dans notre fonction. En effet, **std::vector** nécessite une conversion explicite avec la méthode **wrap**, ce qui entraîne de nombreux appels à **.Call**. Pour résoudre ce problème, nous avons décidé d'utiliser **IntegerVector**, natif à Rcpp, ce qui réduit les appels à **.Call** et améliore les performances.

Voici la nouvelle fonction C++ utilisant la méthode optimisée :

```
cppFunction('
#include <Rcpp.h>
#include <random>
using namespace Rcpp;

// [[Rcpp::export]]
IntegerVector sample_cpp(int n, IntegerVector values, NumericVector probs) {
    std::mt19937_64 rng(std::random_device{}());
    std::discrete_distribution<> dist(probs.begin(), probs.end());

    // Pré-allouer les résultats et générer directement
    IntegerVector results(n);
    for (int& result : results) {
        result = values[dist(rng)];
    }
    return results;
}
')
```

```
## Warning in msb.c2(400, params): Aucune valeur de R n'est supérieure à s.
```

```
## [1] "Profilling de msb.c2 pour n = 400"
```

```
##                       self.time self.pct total.time total.pct
## ".Call"                    1.60    27.30       1.60     27.30
## "rbinom_cpp"               1.28    21.84       1.28     21.84
## "aperm.default"            0.94    16.04       0.94     16.04
## "array"                    0.86    14.68       0.86     14.68
## "accident_MB2cppbinom"     0.56     9.56       2.20     37.54
## "msb.c2"                   0.38     6.48       5.84     99.66
## "as.vector"                0.14     2.39       0.14      2.39
## "FUN"                      0.02     0.34       0.10      1.71
## "cumsum"                   0.02     0.34       0.02      0.34
## "exists"                   0.02     0.34       0.02      0.34
## "f"                        0.02     0.34       0.02      0.34
## "runif"                    0.02     0.34       0.02      0.34
```

Résultat final

Étant globalement satisfaits de nos résultats, nous avons décidé de combiner directement les deux fonctions pour obtenir une fonction unique qui génère à la fois la météo et les accidents en une seule étape.

```
cppFunction('#include <Rcpp.h>
#include <random>
using namespace Rcpp;

// [[Rcpp::export]]
IntegerMatrix simulate_weather_and_accidents(int n_simul, int N, NumericMatrix ptrans, NumericVector pacc) {
    std::mt19937_64 rng(std::random_device{}()); // Using mt19937_64 for RNG
    std::vector<std::discrete_distribution<>> weather_distributions;
    for (int i = 0; i < 3; ++i) {
        weather_distributions.emplace_back(ptrans(i, _).begin(), ptrans(i, _).end());
    }
    IntegerMatrix accidents(n_simul, N);
    for (int sim = 0; sim < n_simul; ++sim) {
        for (int motard = 0; motard < N; ++motard) {
            int weather_state = 0; // Initial météo : soleil
            int total_accidents = 0;

            for (int day = 0; day < 365; ++day) {
                weather_state = weather_distributions[weather_state](rng);
                if (std::uniform_real_distribution<>(0.0, 1.0)(rng) < pacc[weather_state]) { // Optimized uniform random
generator
                    total_accidents++;
                }
            }
            accidents(sim, motard) = total_accidents;
        }
    }

    return accidents;
}
')
```

Nous obtenons donc notre fonction finale qui a ce temps d’exécution.

```
## [1] "Profilling de msb.c2 pour n = 100"
```

```
##                       self.time self.pct total.time total.pct
## "sample_cpp"               1.80    29.90       1.80     29.90
## "rbinom_cpp"               1.42    23.59       1.42     23.59
## "array"                    0.98    16.28       0.98     16.28
## "aperm.default"            0.80    13.29       0.80     13.29
## "msb.c2"                   0.38     6.31       6.00     99.67
## "accident_MB2cppbinom"     0.36     5.98       2.04     33.89
## "as.vector"                0.12     1.99       0.12      1.99
## "FUN"                      0.06     1.00       0.14      2.33
## "rremb"                    0.02     0.33       0.08      1.33
## "as.list.default"          0.02     0.33       0.02      0.33
## "dev.cur"                  0.02     0.33       0.02      0.33
## "stopifnot"                0.02     0.33       0.02      0.33
## "unique.default"           0.02     0.33       0.02      0.33
```

On peut enfin comparer le temps d'exécution des 3 différentes fonctions msb qui utilisent c++ et celui qui ne l'utilise pas

```
## # A tibble: 4 × 2
##   Nom                                                Sampling_Time
##   <chr>                                              <chr>
## 1 msb.c avec uniquement rbinom en C++                6.22 secondes
## 2 msb.c avec les 2 en C++ avec la mauvaise variable 6.78 secondes
## 3 msb.c avec les deux en C++ et la bonne variable   5.86 secondes
## 4 msb sans C++                                       14.46 secondes
```

En plus de cela, la nouvelle fonction msb.c n'utilise presque pas de mémoire contrairement aux 2 autres.

5 Influence des paramètres (Retour au Sommaire)

```
##
## Attaching package: 'dplyr'
```

```
## The following objects are masked from 'package:stats':
##
##     filter, lag
```

```
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

Nous allons utiliser les bibliothèques **dplyr** et **tibble** pour construire des tableaux décrivant l'évolution des valeurs de **rremb**, **msa**, **msb**, et **msb.c** en fonction des paramètres. Vous pouvez voir le code correspondant ici .

Variation de x_0

Dans cette section, nous analysons les variations de x_0 pour étudier l'impact des paramètres initiaux sur les valeurs générées par les différentes fonctions.

```
## Variation de la moyenne des sinistres ainsi que des montants de remboursement selon les variations de eta :
##  rremb n : 8000000
##  msa n = 30000
##  msb n = 300
##  msb.c n = 1000
```

```
## # A tibble: 5 × 8
##      x0 moyenne_sinistre   msa msa_demi_largeur   msb msb_demi_largeur msb.c
##   <dbl>            <dbl> <dbl>            <dbl> <dbl>            <dbl> <dbl>
## 1   0.1             1.43 12.7            10.0     0                0      0
## 2   1               1.77  8.65            2.11    0                0     14.7
## 3   1.5             2.12 13.9             0.570  14.6              7.09  13.0
## 4   2               2.51 23.4             0.336  24.2              3.62  23.1
## 5   2.9             3.27 69.3             0.409  71.6              4.08  70.3
##   msb.c_demi_largeur
##                <dbl>
## 1               0
## 2               5.97
## 3               3.03
## 4               1.74
## 5               2.23
```

L'augmentation de x_0 entraîne une augmentation de la moyenne, mais on observe également que l'écart entre la moyenne et x_0 diminue. Cela confirme bien le postulat initial selon lequel x_0 pourrait représenter une franchise minimale. De plus, on constate que les moyennes augmentent de manière cohérente. Les moyennes augmentent également ce qui est cohérent car avec notre seuil x_0 plus grand on a des dépassements plus fréquemment.

```
##            used  (Mb) gc trigger   (Mb)   max used   (Mb)
## Ncells   862083  46.1    1370522   73.2    1370522   73.2
## Vcells 92910599 708.9  461682813 3522.4 1081930998 8254.5
```

Variation de η

```
## Variation de la moyenne des sinistres ainsi que des montants de remboursement selon les variations de eta :
##  rremb n : 8000000
##  msa n = 30000
##  msb n = 300
##  msb.c n = 1000
```

```
## # A tibble: 5 × 8
##     eta moyenne_sinistre   msa msa_demi_largeur   msb msb_demi_largeur msb.c
##   <dbl>            <dbl> <dbl>            <dbl> <dbl>            <dbl> <dbl>
## 1   3.1             1.77 10.7              2.45  6.98             3.75  4.10
## 2   3.5             1.56  5.53             0     0                0     0
## 3   3.8             1.45  0                0     0                0     0
## 4   4               1.39  0                0     0                0     0
## 5   5               1.20  0                0     0                0     0
##   msb.c_demi_largeur
##                <dbl>
## 1               3.99
## 2               0
## 3               0
## 4               0
## 5               0
```

On observe que lorsque η augmente, même légèrement, la valeur de $\exp(-\eta \log(\alpha + |x - x_0|))$ décroît très rapidement. Cela entraîne une diminution du nombre de valeurs qui dépassent le seuil fixé initialement (pour la densité). Par conséquent, notre intervalle devient inadapté et devra être élargi pour couvrir un plus grand nombre de remboursements. Étant donné que notre intervalle capture un nombre réduit de remboursements, il est tout à fait logique et naturel que la moyenne des sinistres baisse progressivement. C'est ce phénomène qui explique la chute des valeurs de msa, msb, et $msb.c$ ainbsi que de la moyenne des sinistres..

```
##            used  (Mb) gc trigger   (Mb)   max used   (Mb)
## Ncells   862699  46.1    1370522   73.2    1370522   73.2
## Vcells 92913541 708.9  458790720 3500.3 1081930998 8254.5
```

Variation de la probabilité de transition

```
## Variation de la moyenne des sinistres ainsi que des montants de remboursement selon proba de transition :
##  rremb n : 8000000
##  msa n = 30000
##  msb n = 300
##  msb.c n = 1000
```

```
## # A tibble: 5 × 14
##     Psn   Pns   Pnp   Ppn    PS    PN    PP moyenne_sinistre   msa
##   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>            <dbl> <dbl>
## 1   0.3   0.2   0.3   0.4 0.276 0.414 0.310             1.77 10.7
## 2   0.3   0.1   0.1   0.5 0.276 0.414 0.310             1.77 10.2
## 3   0.5   0.1   0.3   0.2 0.276 0.414 0.310             1.77  9.26
## 4   0.1   0.2   0.5   0.2 0.276 0.414 0.310             1.77 11.3
## 5   0.1   0.5   0.3   0.1 0.276 0.414 0.310             1.77  9.67
##   msa_demi_largeur    msb msb_demi_largeur msb.c msb.c_demi_largeur
##              <dbl>  <dbl>            <dbl> <dbl>              <dbl>
## 1             2.45  6.98             3.75   9.54               8.07
## 2             2.12  0.600            0.554  4.65               2.26
## 3             1.89  0                0      9.14               0
## 4             2.07  0                0     49.2                0
## 5             1.89 22.8              0     12.1               10.1
```

En effet, comme nous avons eu recours à la mesure stationnaire, les modifications des matrices de transition n'ont eu aucun effet sur les probabilités. On observe que les espérances changent très légèrement, ce qui est dû au terme d'erreur présent dans les simulations aléatoires de $rremb$. Ces variations sont minimes, de l'ordre du centième ou du millième, et n'affectent pas significativement les résultats globaux.

```
##            used  (Mb) gc trigger   (Mb)   max used   (Mb)
## Ncells   863313  46.2    1370522   73.2    1370522   73.2
## Vcells 92918449 709.0  468108600 3571.4 1081930998 8254.5
```

Variation de la longueur de l'intervalle sur lequel on utilise la méthode de la fonction de répartition cumulée

```
## Variation de la moyenne des sinistres ainsi que des montants de remboursement selon la largeur de l'intervalle avec :
##  rremb n : 8000000
##  msa n = 30000
##  msb n = 300
##  msb.c n = 1000
```

```
## # A tibble: 8 × 8
##   x_len moyenne_sinistre   msa msa_demi_largeur   msb msb_demi_largeur  msb.c
##   <dbl>            <dbl> <dbl>            <dbl> <dbl>            <dbl>  <dbl>
## 1    30             1.30 10.4              1.82  0                 0   23.7
## 2    60             1.54 10.5              2.28  0                 0    1.15
## 3   150             1.69  8.84             1.68  0                 0    0.164
## 4   300             1.74  9.57             2.00  0                 0    2.28
## 5   450             1.75  9.25             1.76  9.96              0    3.97
## 6   600             1.76 10.1              1.74  6.55             12.3  9.93
## 7   750             1.77  9.09             1.72  0                 0    0
## 8   900             1.77 11.0              2.39  0                 0    5.11
##   msb.c_demi_largeur
##                <dbl>
## 1              0
## 2              0
## 3              0
## 4              0.242
## 5              3.03
## 6              9.17
## 7              0
## 8              1.62
```

Variation du seuil S

> Pour les seuils, ce que nous avons fait au début, c'est calculer msa avec $S = 0$, puis nous avons estimé approximativement $S \approx 171$. Ensuite, nous avons choisi les seuils 0, $0.5S$, S, $1.5S$, et $2S$ pour effectuer nos analyses.

```
## [1] "ici on affiche pas pour rremb, car à priori le seuil S n'est pas censé affecté la simulation du montant des sinis
tres"
```

```
## Variation de la moyenne des sinistres ainsi que des montants de remboursement selon les variations du seuil :
##   msa n = 30000
##  msb n = 300
##  msb.c n = 1000
```

```
## # A tibble: 5 × 7
##   Seuil   msa msa_demi_largeur    msb msb_demi_largeur msb.c msb.c_demi_largeur
##   <dbl> <dbl>            <dbl>  <dbl>            <dbl> <dbl>              <dbl>
## 1   0   171.             0.289 171.              2.91 171.               1.55
## 2  85.5  85.7            0.289  83.5             2.78  86.9              1.52
## 3 171    21.2            0.275  21.0             2.81  20.8              1.49
## 4 256.   10.9            2.40    6.49            1.03  11.5              0
## 5 342     0              0       0               0      0                0
```

Naturellement, à mesure que le seuil augmente, les valeurs de msa, msb, et $msb.c$ diminuent progressivement, jusqu'à atteindre un point où aucune espérance ne dépasse le seuil.

6 Propositions pour étudier / améliorer le modèle (Retour au Sommaire)

Études qu'on peut faire avec ce modèle

- **Coûts d'assurances**: On pourrait d'abord commencer par étudier la différence entre les valeurs de msa et msb pour des n plus grands. En effet, vu que le seuil choisi par le professeur est égal à 1.5 fois la moyenne de l'espérance, avec

peu de simulations, ce seuil est moins souvent dépassé. Il serait intéressant de voir ce qui se passe avec un plus grand nombre de simulations.

Comment rendre le modèle plus réaliste/précis ?

- **Améliorer la précision de la météo**: Les deux propositions suivantes augmenteraient considérablement les exigences en termes de mémoire :
  1. **Dépendance temporelle**: On pourrait utiliser une chaîne de Markov indexée sur le temps (période $[0, 365]$) avec des probabilités de transition variant selon la saison (hiver, été, etc.).
  2. **Dépendance géographique**: On pourrait créer plusieurs chaînes de Markov spécifiques à chaque région (plutôt qu'une par motard), puis effectuer les simulations en tenant compte de la répartition des motards par région.
  3. **États supplémentaires**: On pourrait ajouter des états météorologiques supplémentaires comme la neige, le brouillard, etc.
- **Améliorer le réalisme des accidents et remboursements**:
  1. Utiliser des densités différentes pour les montants à rembourser, en les liant à plusieurs facteurs (**météo, profil du motard, etc.**) pour mieux refléter la gravité des accidents.
  2. Introduire des **corrélations** entre les motards circulant dans une même région, en modélisant des scénarios où un accident peut en provoquer d'autres (par exemple, des chutes collectives d'un groupe).

Annexe : Codes utilisés au cours du projet (Retour au Sommaire)

Vous trouverez ci dessous, tous les codes décrits dans les essais, ainsi que les codes utilisés pour générer les temps et comparer les performances. ## Code météo

Premiere fonction (Retour au rapport)

```
new_State<- function(current_state, ptrans) {
  transition_probs <- ptrans[current_state, ]
  next_state <- sample(1:3, size = 1, prob = transition_probs)
  return(next_state)
}
sim_Weather1 <- function(days, ptrans) {
  weather <- c(sample(1:3, size = 1))
  n_states <- 3
  for (i in 2:days) {
    weather[i] <- new_State(weather[i-1],ptrans)
  }
  return (weather)
}

# dans msa plus tard
#  for (i in 1:n.simul) {
#    weatherMatrix [i,] <- sim_Weather(365,ptrans)
#  }
```

Deuxieme fonction (Retour au rapport)

```
sim_Weather2<- function(days, ptrans) {

  weather <- numeric(days)
  weather[1] <- which.max(runif(1) <= cumsum(ptrans[1, ]))


  for (i in 2:days) {
    current_state <- weather[i - 1]
    transition_probs <- ptrans[current_state, ]
    weather[i] <- which.max(runif(1) <= cumsum(transition_probs))
  }

  return(weather)
}
#dans msa plus tard
#  for (i in 1:n.simul) {
#    weatherMatrix [i,] <- sim_Weather(365,ptrans)
#  }
```

Troisième fonction (Retour au rapport)

```
sim_Weather3 <- function(days, ptrans) {
  rolls <- runif(days)
  # Weather vector
  weather <- numeric(days)
  weather[1] <- which.max(rolls[1] <= cumsum(ptrans[1, ]))
  for (i in 2:days) {
    current_state <- weather[i - 1]
    transition_probs <- ptrans[current_state, ]
    weather[i] <- which.max(rolls[i] <= cumsum(transition_probs))
  }

  return(weather)
}
#dans msa plus tard
#  for (i in 1:n.simul) {
#    weatherMatrix [i,] <- sim_Weather(365,ptrans)
#  }
```

Dernière fonction et celle maintenue

```
weatherMatrixA <- function(n.simul, ptrans_stat) {
  # Matrice 3D (0.s,365,1000)
  weather <- array(sample(1:3, n.simul * 365 , replace = TRUE, prob = ptrans_stat),
                   dim = c(n.simul, 365))
  return(weather)
}
```

Code accident (Retour au rapport)

Premiere fonction (Retour au rapport)

```
accident_Matrix <- function(n.simul,  acc_probMatrix,N) {
  accidents_tot <- vector("list",n.simul)

  for (i in 1:n.simul){
    accidents_tot[[i]] <- matrix(
      rbinom(N *365, 1, rep(acc_probMatrix[i], each = N)),
      nrow = N,
      ncol = 365,
      byrow = TRUE
    )
  }
  return(accidents_tot)
}
# Dans la fonctio msa plus tard
  #Nbr_acc <- numeric(length(acc_total))
  #Nbr_acc <- sapply(acc_total, sum)
```

deuxieme fonction (Retour au rapport)

```
accident_Matrix <- function(n.simul, acc_probMatrix, N) {
  accidents_tot <- vector("list", n.simul)

  for (i in 1:n.simul) {
    accidents_per_day <- numeric(365)

    for (j in 1:365) {

      accidents_per_day[j] <- rbinom(1, N, acc_probMatrix[i, j])
    }

    accidents_tot[[i]] <- accidents_per_day[accidents_per_day != 0]
  }

  return(accidents_tot)
}
# Dans la fonctio msa plus tard
  #Nbr_acc <- numeric(length(acc_total))
  #Nbr_acc <- sapply(acc_total, sum)
```

Derniere fonction et celle maintenue (Retour au rapport)

```
accident_MA <- function(n.simul,acc_probMatrix,N){
  accidents_tot <- vector("list", n.simul)
  accidentstotal <- rbinom(length(acc_probMatrix), size = N, prob = acc_probMatrix)
  result <- matrix(accidentstotal, nrow=n.simul,ncol=365)
  sums <- rowSums(result)
  return(sums)
}
```

Test rapidité des fonctions MSA/MSB

```
#Ce code est executé bien avant mais il est caché avec l'option de r markdown echo=FALSE eval=TRUE

weatherMatrixB <- function(n.simul, N, ptrans_stat) {
  weather <- array(sample(1:3, n.simul * 365 * N, replace = TRUE, prob = ptrans_stat),
                   dim = c(n.simul, 365, N))
  return(weather)
}

accident_MB2 <- function(n.simul, N, weather, pacc) {
  acc_prob <- pacc[as.vector(weather)]
  accidents <- rbinom(length(acc_prob), size = 1, prob = acc_prob)
  accidents <- array(accidents, dim = c(n.simul, 365, N))

  return(accidents)
}

msbstat <- function(n.simul,params){
  N <- params$N
  s <- params$s
  pacc <- params$pacc
  pSN <- params$ptrans[1]
  pNS <- params$ptrans[2]
  pNP <- params$ptrans[3]
  pPN <- params$ptrans[4]

  ptrans <-matrix(c(1 - pSN, pSN, 0.0,
                    pNS, 1- pNP - pNS, pNP,
                    0.0, pPN, 1-pPN), nrow = 3, byrow = TRUE)

  ptrans_stat <- measure_Stat(ptrans)
  weather <- weatherMatrixB(n.simul,N,ptrans_stat)
  acc_prob <- pacc[weather]
  accidents_sim_motard <- accident_MB2(n.simul, N, weather, pacc)
  accidents_sim <- apply(accidents_sim_motard, 1, sum)
  sinistre <- sapply(accidents_sim, function(x) sum(rremb(x, params)))
  ms_values <- sinistre[sinistre > s] - s
  if (length(ms_values) == 0) {
    warning("Aucune valeur de R n'est supérieure à s.")
    return(list(ms = 0, demi.largeur = 0))
  }
  ms <- mean(ms_values)
  demi.largeur <- half_width(ms_values)
  return(list(
    ms = ms,
    demi.largeur = demi.largeur
  ))
}
accident_MA <- function(n.simul,acc_probMatrix,N){
  accidents_tot <- vector("list", n.simul)
  accidentstotal <- rbinom(length(acc_probMatrix), size = N, prob = acc_probMatrix)
  result <- matrix(accidentstotal, nrow=n.simul,ncol=365)
  sums <- rowSums(result)
  return(sums)
}

msastat <- function(n.simul,params){

  # Variables
  N <- params$N
  s <- params$s
  pacc <- params$pacc
```

```
  ## TRANSITION
  pSN <- params$ptrans[1]
  pNS <- params$ptrans[2]
  pNP <- params$ptrans[3]
  pPN <- params$ptrans[4]

  ptrans <-matrix(c(1 - pSN, pSN, 0.0,
                    pNS, 1- pNP - pNS, pNP,
                    0.0, pPN, 1-pPN), nrow = 3, byrow = TRUE)
  ptrans_stat <- measure_Stat(ptrans)
  weather <- weatherMatrixA(n.simul,ptrans_stat)
  # proba d'accident pour chaque jour pour chaque simu (n,365)
  acc_probMatrix <- matrix(pacc[weather], nrow=n.simul, ncol=365)

  acc_total <- accident_MA(n.simul, acc_probMatrix, N)


  ##  simulation sinistre pour chaque simu

  sinistre <- sapply(acc_total, function(x) sum(rremb(x, params)))
  ms_values <- sinistre[sinistre > s] - s
  if (length(ms_values) == 0) {
    warning("Aucune valeur de R n'est supérieure à s.")
    return(list(ms = 0, demi.largeur = 0))
  }
  ms <- mean(ms_values)
  demi.largeur <- half_width(ms_values)
  return(list(
    ms = ms,
    demi.largeur = demi.largeur
  ))
}
```

Code pour comparer 2 fonctions quelconques on l'a utilisé pour rremb, msa,msastat etc etc

```
compare_functions <- function(f1, f2, params, n.simul) {
  #temps d'exécution pour la première fonction
  time_f1 <- system.time({
    result_f1 <- f1(n.simul, params)
  })["elapsed"]

  #temps d'exécution pour la deuxième fonction
  time_f2 <- system.time({
    result_f2 <- f2(n.simul, params)
  })["elapsed"]
  #résultats
  cat(sprintf("Temps d'exécution de %s : %.2f secondes\n", deparse(substitute(f1)), time_f1))
  cat(sprintf("Temps d'exécution de %s : %.2f secondes\n", deparse(substitute(f2)), time_f2))
  #Comparaison
  if (time_f1 < time_f2) {
    cat(sprintf("%s est %.2f%% plus rapide que %s.\n",
                deparse(substitute(f1)),
                100 * (1 - time_f1 / time_f2),
                deparse(substitute(f2))))
  } else if (time_f1 > time_f2) {
    cat(sprintf("%s est %.2f%% plus rapide que %s.\n",
                deparse(substitute(f2)),
                100 * (1 - time_f2 / time_f1),
                deparse(substitute(f1))))
  }
}
cat("\n Comparaison entre msbstat et msb pour n.simul = 20000, N = 1000 :\n")
compare_functions(msastat, msa1, params,n.simul = 20000)
cat("\n Comparaison entre msbstat et msb pour n.simul = 100, N = 1000 :\n")
compare_functions(msbstat, msb1, params,n.simul = 100)
```

Codes généraux

Code pour dessiner les fonctions

```
plot_density_rremb <- function(n, params) {
  values <- rremb(n, params)
  plot(density(values), main = "Densité de probabilité de rremb", xlab = "Valeurs", ylab = "Densité")
}
plot_cdf_rremb <- function(n, params) {
  values <- sort(rremb(n, params))
  cdf <- ecdf(values)
  plot(cdf, main = "Fonction de répartition cumulative (CDF)", xlab = "Valeurs", ylab = "Probabilité cumulative", vertica
ls = TRUE)
}
```

Comparaison des météos, Retour au rapport

```
pSN <- params$ptrans[1]
  pNS <- params$ptrans[2]
  pNP <- params$ptrans[3]
  pPN <- params$ptrans[4]

  ptrans <-matrix(c(1 - pSN, pSN, 0.0,
                    pNS, 1- pNP - pNS, pNP,
                    0.0, pPN, 1-pPN), nrow = 3, byrow = TRUE)

new_State<- function(current_state, ptrans) {
  transition_probs <- ptrans[current_state, ]
  next_state <- sample(1:3, size = 1, prob = transition_probs)
  return(next_state)
}
sim_Weather1 <- function(days, ptrans) {
  weather <- c(sample(1:3, size = 1))
  n_states <- 3
  for (i in 2:days) {
    weather[i] <- new_State(weather[i-1],ptrans)
  }
  return (weather)
}

sim_Weather2<- function(days, ptrans) {

  weather <- numeric(days)
  weather[1] <- which.max(runif(1) <= cumsum(ptrans[1, ]))


  for (i in 2:days) {
    current_state <- weather[i - 1]
    transition_probs <- ptrans[current_state, ]
    weather[i] <- which.max(runif(1) <= cumsum(transition_probs))
  }

  return(weather)
}

sim_Weather3 <- function(days, ptrans) {
  rolls <- runif(days)
  # Weather vector
  weather <- numeric(days)
  weather[1] <- which.max(rolls[1] <= cumsum(ptrans[1, ]))
  for (i in 2:days) {
    current_state <- weather[i - 1]
    transition_probs <- ptrans[current_state, ]
    weather[i] <- which.max(rolls[i] <= cumsum(transition_probs))
  }

  return(weather)
}
generate_weather_matrix1 <- function(n.simul, ptrans) {
  # Initialisation de la matrice
  weatherMatrix <- matrix(0, nrow = n.simul, ncol = 365)

  # Remplissage de la matrice avec une boucle for
  for (i in 1:n.simul) {
    weatherMatrix[i, ] <- sim_Weather1(365, ptrans)
  }

  # Retourner la matrice
```

```r
  return(weatherMatrix)
}

generate_weather_matrix2 <- function(n.simul, ptrans) {
  # Initialisation de la matrice
  weatherMatrix <- matrix(0, nrow = n.simul, ncol = 365)

  # Remplissage de la matrice avec une boucle for
  for (i in 1:n.simul) {
    weatherMatrix[i, ] <- sim_Weather2(365, ptrans)
  }

  # Retourner la matrice
  return(weatherMatrix)
}

generate_weather_matrix3 <- function(n.simul, ptrans) {
  # Initialisation de la matrice
  weatherMatrix <- matrix(0, nrow = n.simul, ncol = 365)

  # Remplissage de la matrice avec une boucle for
  for (i in 1:n.simul) {
    weatherMatrix[i, ] <- sim_Weather3(365, ptrans)
  }

  # Retourner la matrice
  return(weatherMatrix)
}

tempsparfonction <- function(func_prefix, num_funcs, n.simul, ptrans) {
  # Initialiser une liste pour stocker les résultats de profiling
  profiling_results <- list()

  # Parcourir chaque fonction de f1 à f<num_funcs>
  for (i in 1:num_funcs) {
    # Générer le nom de la fonction
    func_name <- paste0(func_prefix, i)

    if (exists(func_name, mode = "function")) {
      func <- get(func_name)  # Récupérer la fonction par son nom

      # Profiling de la fonction
      Rprof("profiling.out")  # Commencer le profiling
      b <- func(n.simul, ptrans)   # Appeler la fonction avec les paramètres
      Rprof(NULL)             # Arrêter le profiling

      # Résumé du profil
      profiling_summary <- summaryRprof("profiling.out")

      # Ajouter le temps d'échantillonnage à la liste
      profiling_results[[func_name]] <- profiling_summary$sampling.time
    } else {
      warning(paste("Function", func_name, "does not exist."))
    }
  }

  # Retourner la liste des résultats
  return(profiling_results)
}
test_meteo <- tempsparfonction("generate_weather_matrix",3,3000,ptrans)

for (i in seq_along(test_meteo)) {
  cat(sprintf("Le temps d'execution pour la fonction de météo numéro %d pour n = 3000 est : %.2f\n", i, test_meteo[i],"se
```

```
condes"))

}
gain_1_2 <- (test_meteo$generate_weather_matrix1 / test_meteo$generate_weather_matrix2)
gain_2_3 <-(test_meteo$generate_weather_matrix2 / test_meteo$generate_weather_matrix3)
gain_1_3 <- (test_meteo$generate_weather_matrix1/ test_meteo$generate_weather_matrix3)

# Génération des phrases
cat(sprintf("Le 2ème code est %.2f fois plus rapide que le 1er.\n", gain_1_2))
cat(sprintf("Le 3ème code est %.2f fois plus rapide que le 2ème.\n", gain_2_3))
cat(sprintf("Le 3ème code est %.2f fois plus rapide que le 1er.\n", gain_1_3))
```

Comparaison des fonctions accidents (Retour au rapport)

```
weatherMatrixA <- function(n.simul, ptrans_stat) {
  # Matrice 3D (0.s,365,1000)
  weather <- array(sample(1:3, n.simul * 365 , replace = TRUE, prob = ptrans_stat),
                   dim = c(n.simul, 365))
  return(weather)
}

tempsparfonction <- function(func_prefix, num_funcs, n.simul,N,pacc) {
  # Initialiser une liste pour stocker les résultats de profiling
  profiling_results <- list()
  weather_prob  <- pacc[weatherMatrixA(n.simul,c(0.2758621,0.4137931,0.3103448))]
  acc_probMatrix <- matrix(weather_prob, nrow = n.simul, ncol = 365)

  # Parcourir chaque fonction de f1 à f<num_funcs>
  for (i in 1:num_funcs) {
    # Générer le nom de la fonction
    func_name <- paste0(func_prefix, i)

    if (exists(func_name, mode = "function")) {
      func <- get(func_name)  # Récupérer la fonction par son nom

      # Profiling de la fonction
      Rprof("profiling.out")  # Commencer le profiling
      b <- func(n.simul, acc_probMatrix,N)   # Appeler la fonction avec les paramètres
      Rprof(NULL)             # Arrêter le profiling

      # Résumé du profil
      profiling_summary <- summaryRprof("profiling.out")

      # Ajouter le temps d'échantillonnage à la liste
      profiling_results[[func_name]] <- profiling_summary$sampling.time
    } else {
      warning(paste("Function", func_name, "does not exist."))
    }
  }

  # Retourner la liste des résultats
  return(profiling_results)
}
test <- tempsparfonction("accident_Matrix",3,1000,1000,pacc)

for (i in seq_along(test)) {
  cat(sprintf("Le temps d'execution pour la fonction de météo numéro %d pour n = 1000, N = 1000 est : %.2f secondes\n",
i, test[i]))
}
gain_1_2 <- test$accident_Matrix1 / test$accident_Matrix2
gain_2_3 <- test$accident_Matrix2 / test$accident_Matrix3
gain_1_3 <- test$accident_Matrix1 / test$accident_Matrix3

# Génération des phrases
cat(sprintf("Le 2ème code est %.2f%% plus rapide que le 1er.\n", gain_1_2))
cat(sprintf("Le 3ème code est %.2f%% plus rapide que le 2ème.\n", gain_2_3))
cat(sprintf("Le 3ème code est %.2f%% plus rapide que le 1er.\n", gain_1_3))
```

Comparaison des fonctions cpp (même code à chaque changement, on change juste les

fonctions)

```
weatherbiker2 <- function(n.simul, N) {
  # Matrice 3D (0.s,365,1000)
  weather <- array(sample(1:3, n.simul * 365 * N, replace = TRUE, prob = c(8/29,12/29,9/29)),
                   dim = c(n.simul, 365, N))
  return(weather)
}
accident_MB2cppbinom <- function(n.simul, N, weather, pacc) {
  # Mettez weather sous forme de vecteur pour calculer les probabilités directement
  acc_prob <- pacc[as.vector(weather)]
  accidents <- rbinom_cpp(rep(1, length(acc_prob)), acc_prob)

  # Reconstruire en matrice 3D (n.simul, 365, N)
  accidents <- array(accidents, dim = c(n.simul, 365, N))

  return(accidents)
}

msb.c1 <- function(n.simul,params){

  # parameters
  N <- params$N
  s <- params$s
  pacc <- params$pacc
  ## Transition probabilities
  pSN <- params$ptrans[1]
  pNS <- params$ptrans[2]
  pNP <- params$ptrans[3]
  pPN <- params$ptrans[4]

  ptrans <-matrix(c(1 - pSN, pSN, 0.0,
                    pNS, 1- pNP - pNS, pNP,
                    0.0, pPN, 1-pPN), nrow = 3, byrow = TRUE)

  #Weather
  weather <- weatherbiker2(n.simul,N)

  #prob acc
  acc_prob <- pacc[weather]
  #accident total
  accidents_sim_motard <- accident_MB2cppbinom(n.simul, N, weather, pacc)
  accidents_sim <- apply(accidents_sim_motard, 1, sum)
  sinistre <- sapply(accidents_sim, function(x) sum(rremb(x, params)))
  ms_values <- sinistre[sinistre > s] - s
  if (length(ms_values) == 0) {
    warning("Aucune valeur de R n'est supérieure à s.")
    return(list(ms = 0, demi.largeur = 0))
  }
  #returning the mean of the values when it's higher than s and the half width for a 95 % confidence
  ms <- mean(ms_values)
  demi.largeur <- half_width(ms_values)
  return(list(
    ms = ms,
    demi.largeur = demi.largeur
  ))
}

set.seed(150)
Rprof("msb.out")
result_msb <- msbstat(400, params)
Rprof(NULL)
profile_msb <- summaryRprof("msb.out")
```

```
Rprof("msb.c1.out")
result_msb <- msb.c1(400, params)
Rprof(NULL)
profile_msb.c1 <- summaryRprof("msb.c1.out")

print("Profilling de msbstat pour n = 400 ")
print(profile_msb$by.self)
print("Profilling de msb.c1 pour n = 100")
print(profile_msb.c1$by.self)
```

Comparaison de 2 fonctions quelconques (utilisée notamment pour rremb)

```
compare_functions <- function(f1, f2, params, n.simul) {
  # Mesurer le temps d'exécution pour la première fonction (méthode des trapèzes)
  time_f1 <- system.time({
    result_f1 <- f1(n.simul, params)
  })["elapsed"] #on prend seulement le temps écoulé
  time_f2 <- system.time({
    result_f2 <- f2(n.simul, params)
  })["elapsed"] #on prend seulement le temps écoulé
  cat(sprintf("Temps d'exécution de %s qui utilise méthode des trapèzes : %.2f secondes pour n=80000000 \n", deparse(subs
titute(f1)), time_f1)) #conversion du nom de la fonction en texte
  # Nous avons changé la description dans chacun des cas.
  cat(sprintf("Temps d'exécution de %s qui utilise l'integration dans R : %.2f secondes pour n=80000000 \n", deparse(subs
titute(f2)), time_f2)) #conversion du nom de la fonction en texte
}
```

Changement des variables. (Nous avons utilisé la même fonction en changeant à chaque fois les paramètres en questions.

On ne le montre pas ici mais à la fin de chaque tableau ou on calcule l’effet de la variation des variables nous avons rénitialisé les paramètres (mis ceux choisi au départ) et nettoyés la mémoire Retour au rapport

```
set.seed(170)
x0_values <- c(0.1, 1, 1.5, 2, 2.9)
# Calculs
results_x0 <- lapply(x0_values, function(x0) {
  params$x0 <- x0

  # Calcul des métriques
  mean_rremb <- mean(rremb(8000000, params))
  msa_res <- msa(30000, params)
  msb_res <- msb(300, params)
  msb.c_res <- msb.c(1000, params)

  tibble(
    x0 = x0,
    msa_demi_largeur = msa_res$demi.largeur,
    msb = msb_res$ms,
    msb_demi_largeur = msb_res$demi.largeur,
    msb.c = msb.c_res$ms,
    msb.c_demi_largeur = msb.c_res$demi.largeur
  )
})

# Compilation des résultats dans un tableau final
final_results_x0 <- bind_rows(results_x0)

# Affichage
cat("Variation de la moyenne des sinistres ainsi que des montants de remboursement selon x0 :\n rremb n : 8000000 \n msa
n = 30000 \n msb n = 300 \n msb.c n = 1000 \n")
print(final_results_x0,width=1000)
```

Tableau

Tableau pour les msb.c retour au rapport

```
library(tibble)
sampling_times <- tibble::tibble(
  Nom = c("msb.c avec uniquement rbinom en c++", "msb.c avec les 2 en c++ avec la mauvaise variable", "msb.c avec les deu
x en c++ et la bonne variable"),
 Sampling_Time = paste0(
    c(
      profile_msb.c1$sampling.time,
      profile_msb.c2$sampling.time,
      profile_msb.c3$sampling.time
    ),
    " secondes"
  )
)

print(sampling_times)
```

Test du chi2

Retour au rapport

```r
# On simule plusieurs météo et on compte le nombre d'occurences
simulate_multiple_weather <- function(n_simulations, days, ptrans) {
  n_states <- nrow(ptrans)
  state_counts <- numeric(n_states)

  for (sim in 1:n_simulations) {
    weather <- sim_Weather3(days, ptrans) #simulation de 365 jours
    state_counts <- state_counts + table(factor(weather, levels = 1:n_states)) #compter les occurences de chacuns des cas
  }

  return(state_counts)
}

# Test du Chi carré
test_chi2_stationarity_weather <- function(ptrans=ptranss, ptrans_stati = ptrans_stat, days=365, n_simulations=6000, alph
a = 0.01) {
  # Simuler les météos et faire le compte
  observed_counts <- simulate_multiple_weather(n_simulations, days, ptrans)
  # On calcule le nombre d'états qu'on attendait
  expected_counts <- ptrans_stat * sum(observed_counts)
  # On calcule notre Chi carré
  chi2_stat <- sum((observed_counts - expected_counts)^2 / expected_counts)
  # p-value à 2 degré de liberté car on a 3 états
  p_value <- pchisq(chi2_stat, df = length(ptrans_stat) - 1, lower.tail = FALSE)

  # Determine significance
  significant <- p_value <= alpha

  return(list(
    chi2_stat = chi2_stat,
    p_value = p_value,
    significant = significant,
    alpha = alpha,
    observed = observed_counts,
    expected = expected_counts
  ))
}

result <- test_chi2_stationarity_weather()

# Display Results
cat("Chi-Squared Statistic:", result$chi2_stat, "\n")
cat("p-value:", result$p_value, "\n")
cat("Significant (alpha =", result$alpha, "):", ifelse(result$significant, "Yes", "No"), "\n")
cat("Observed Counts:", result$observed, "\n")
cat("Expected Counts:", result$expected, "\n")
```

Code final des 4 fonctions (Retour au Sommaire)

```
Fonction <- function(params){
  alpha <- params$alpha
  x0 <- params$x0
  eta <- params$eta
  f <- function(x) {
    return(exp(-eta * log(alpha + abs(x - x0))))
  }
  x_vals <- seq(0, 31, length.out = 620)
  f_vals <- f(x_vals)
  dx <- diff(x_vals)
  cdf_vals <- cumsum((f_vals[-length(f_vals)] + f_vals[-1]) / 2 * dx)
  total <- cdf_vals[length(cdf_vals)]
  cdf_vals <- cdf_vals / total
  unique_indices <- which(!duplicated(cdf_vals))
  x_vals_unique <- x_vals[unique_indices]
  cdf_vals_unique <- cdf_vals[unique_indices]
  g <- approxfun(cdf_vals_unique, x_vals_unique, rule = 2)
  return(g)
}

rremb <- function(n, params) {
  if (!exists("fonction_direct_cache", envir = .GlobalEnv)) {
    assign("fonction_direct_cache", Fonction(params), envir = .GlobalEnv)
  }
  fonction_direct <- get("fonction_direct_cache", envir = .GlobalEnv)
  simulated_vals <- fonction_direct(runif(n))
  return(simulated_vals)
}


measure_Stat <- function(ptrans) {
  P_trans <- t(ptrans)
  VP <- eigen(P_trans)
  VP_one <- which(abs(VP$values - 1) < 1e-10)
  stat_vec <- Re(VP$vectors[, VP_one])
  measure_stat <- stat_vec / sum(stat_vec)
  return(measure_stat)
}

half_width <- function(values, cfdc = 0.95) {
  n <- length(values)
  mean_val <- mean(values)
  sd_val <- if (n > 1) sd(values) else 0
  a = qnorm((1 + cfdc) / 2)
  error_margin <- a * sd_val / sqrt(n)
  return(error_margin)
}

weatherMatrixA <- function(n.simul, ptrans_stat) {
  weather <- array(sample(1:3, n.simul * 365 , replace = TRUE, prob = ptrans_stat),
                   dim = c(n.simul, 365))
  return(weather)
}


accident_MA <- function(n.simul,acc_probMatrix,N){
  accidents_tot <- vector("list", n.simul)
  accidentstotal <- rbinom(length(acc_probMatrix), size = N, prob = acc_probMatrix)
  result <- matrix(accidentstotal, nrow=n.simul,ncol=365)
  sums <- rowSums(result)
  return(sums)
```

```r
}

msa <- function(n.simul,params){
  gc()
  N <- params$N
  s <- params$s
  pacc <- params$pacc
  pSN <- params$ptrans[1]
  pNS <- params$ptrans[2]
  pNP <- params$ptrans[3]
  pPN <- params$ptrans[4]

  ptrans <-matrix(c(1 - pSN, pSN, 0.0,
                    pNS, 1- pNP - pNS, pNP,
                    0.0, pPN, 1-pPN), nrow = 3, byrow = TRUE)
  ptrans_stat <- measure_Stat(ptrans)
  weather <- weatherMatrixA(n.simul,ptrans_stat)
  acc_probMatrix <- matrix(pacc[weather], nrow=n.simul, ncol=365)

  acc_total <- accident_MA(n.simul, acc_probMatrix, N)

  sinistre <- sapply(acc_total, function(x) sum(rremb(x, params)))
  ms_values <- sinistre[sinistre > s] - s
  if (length(ms_values) == 0) {
    warning("Aucune valeur de R n'est supérieure à s.")
    return(list(ms = 0, demi.largeur = 0))
  }
  ms <- mean(ms_values)
  demi.largeur <- half_width(ms_values)
  return(list(
    ms = ms,
    demi.largeur = demi.largeur
  ))
}

weatherMatrixB <- function(n.simul, N, ptrans_stat, block_size = 100) {
  weather_full <- array(0, dim = c(n.simul, 365, N))
  total_blocks <- ceiling(N / block_size)
  for (b in 1:total_blocks) {
    motard_start <- (b - 1) * block_size + 1
    motard_end <- min(b * block_size, N)
    block_motards <- motard_end - motard_start + 1
    weather_block <- array(
      sample(1:3, n.simul * 365 * block_motards, replace = TRUE, prob = ptrans_stat),
      dim = c(n.simul, 365, block_motards)
    )
    weather_full[, , motard_start:motard_end] <- weather_block
  }

  return(weather_full)
}

accident_MB2 <- function(n.simul, N, weather, pacc) {
  acc_prob <- pacc[as.vector(weather)]
  accidents <- rbinom(length(acc_prob), size = 1, prob = acc_prob)
  accidents <- array(accidents, dim = c(n.simul, 365, N))

  return(accidents)
}

msb <- function(n.simul,params){
  N <- params$N
  s <- params$s
```

```
  pacc <- params$pacc
  pSN <- params$ptrans[1]
  pNS <- params$ptrans[2]
  pNP <- params$ptrans[3]
  pPN <- params$ptrans[4]

  ptrans <-matrix(c(1 - pSN, pSN, 0.0,
                    pNS, 1- pNP - pNS, pNP,
                    0.0, pPN, 1-pPN), nrow = 3, byrow = TRUE)

  ptrans_stat <- measure_Stat(ptrans)
  weather <- weatherMatrixB(n.simul,N,ptrans_stat)
  acc_prob <- pacc[weather]
  accidents_sim_motard <- accident_MB2(n.simul, N, weather, pacc)
  accidents_sim <- apply(accidents_sim_motard, 1, sum)
  sinistre <- sapply(accidents_sim, function(x) sum(rremb(x, params)))
  ms_values <- sinistre[sinistre > s] - s
  if (length(ms_values) == 0) {
    warning("Aucune valeur de R n'est supérieure à s.")
    return(list(ms = 0, demi.largeur = 0))
  }
  ms <- mean(ms_values)
  demi.largeur <- half_width(ms_values)
  return(list(
    ms = ms,
    demi.largeur = demi.largeur
  ))
}

msb.c <- function(n.simul, params) {
  library(Rcpp)
  cppFunction('#include <Rcpp.h>
#include <random>
using namespace Rcpp;

// [[Rcpp::export]]
IntegerMatrix simulate_weather_and_accidents(int n_simul, int N, NumericMatrix ptrans, NumericVector pacc) {
    std::mt19937_64 rng(std::random_device{}()); // Using mt19937_64 for RNG
    std::vector<std::discrete_distribution<>> weather_distributions;
    for (int i = 0; i < 3; ++i) {
        weather_distributions.emplace_back(ptrans(i, _).begin(), ptrans(i, _).end());
    }
    IntegerMatrix accidents(n_simul, N);
    for (int sim = 0; sim < n_simul; ++sim) {
        for (int motard = 0; motard < N; ++motard) {
            int weather_state = 0; // Initial météo : soleil
            int total_accidents = 0;

            for (int day = 0; day < 365; ++day) {
                weather_state = weather_distributions[weather_state](rng);
                if (std::uniform_real_distribution<>(0.0, 1.0)(rng) < pacc[weather_state]) { // Optimized uniform random
generator
                    total_accidents++;
                }
            }
            accidents(sim, motard) = total_accidents;
        }
    }

    return accidents;
}
')
  N <- params$N
```

```r
  s <- params$s
  pacc <- params$pacc
  pSN <- params$ptrans[1]
  pNS <- params$ptrans[2]
  pNP <- params$ptrans[3]
  pPN <- params$ptrans[4]

  ptrans <-matrix(c(1 - pSN, pSN, 0.0,
                    pNS, 1- pNP - pNS, pNP,
                    0.0, pPN, 1-pPN), nrow = 3, byrow = TRUE)

  trans <- matrix(rep(measure_Stat(ptrans), each = 3), nrow = 3, byrow = TRUE)
  accidents <- simulate_weather_and_accidents(n.simul, N, trans , pacc)
  total_accidents_sim <- rowSums(accidents)
  sinistres <- sapply(total_accidents_sim, function(total_accidents) {
    sum(rremb(total_accidents, params))
  })
  ms_values <- sinistres[sinistres > s] - s
  if (length(ms_values) == 0) {
    warning("Aucune valeur de R n'est supérieure à s.")
    return(list(ms = 0, demi.largeur = 0))
  }
  ms <- mean(ms_values)
  demi.largeur <- half_width(ms_values)

  return(list(
    ms = ms,
    demi.largeur = demi.largeur
  ))
}
```