Final Assignment Report : Monte carlo simulations for Insurance Portfolio

Mouad Sehli

January 2025

Simulation Techniques

Msc applied mathematics and actuarial science

Contents

1 Introduction

This report is the result of an assignment for the course **Simulation Techniques**, taught by **Mr.Alexis Bienvenue**. It presents the application of simulation methods, with a particular focus on **Monte Carlo techniques**, to address a problem involving **stochastic modeling** in an **insurance context**.

Simulation techniques are a fundamental pillar of modern **computational** and **statistical analysis**, especially in fields like **finance**, **insurance**, and **risk management**. Through this project, we explored how to represent and analyze **random phenomena**—such as the **occurrence** and **financial impact** of claims—using tools like **pseudorandom number generators**, **probabilistic models**, and **advanced statistical techniques**.

The purpose of this work was to explore and implement advanced **simulation methodologies** to analyze **risks** and **dependencies** in a given system. This project offered an opportunity to deepen our understanding of **simulation techniques**, refine our **coding skills in R**, and apply **theoretical knowledge** to **practical problems**.

The report highlights the steps undertaken to **design**, **implement**, and **validate** the simulations, as well as to interpret the results obtained. Additionally, it reflects on the **computational strategies** employed to optimize the process, including leveraging **hybrid programming approaches**.

Note : All the codes for the various functions and tests discussed in this document are available on my GitHub repository. For convenience, only the final version of each function, reflecting the most optimized and retained implementations, is displayed at the end of this document.

2 Background and context

2.1 Context

The assignment focuses on modeling and simulating risks associated with motorcycle insurance. The objective is to study the impact of weather conditions on accident probabilities and insurance reimbursements using stochastic Markov chains and Monte Carlo simulations. The model accounts for different weather states Sunny(S), Cloudy(N), Raining(R) and their transitions as showcased below, as well as their influence on accident likelihoods and claim amounts.

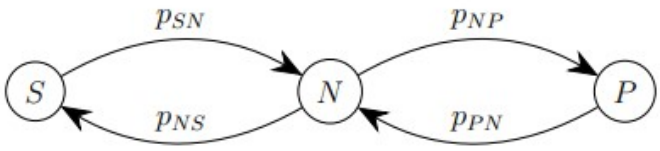

Figure 1: Weather Markov chain transition diagram

Tasks

- **Weather Modeling:** Use a three-state Markov chain to simulate daily weather conditions over a specified period.
- **Claim Distribution Simulation:** Simulate claim amounts according to the specified probability law, using parameters such as α, x_0, and η.
- **Monte Carlo Approximation:** Estimate the conditional expectation $m(s) = E(R - s \mid R > s)$, where R is the total reimbursement, under two hypotheses:
  1. **Hypothesis A:** All motorcyclists experience the same weather.
  2. **Hypothesis B:** Each motorcyclist has independent weather conditions.
- **Efficiency Competition:** Write efficient R code to simulate the models and compute $m(s)$, without using external libraries. (Code was regularly submitted to a test website)
- **C++ Integration:** Implement parts of the simulation in C++ to optimize performance.
- **Parameter Analysis:** Study the influence of key parameters, such as weather probabilities and thresholds, on the results.

3 Parameters Setup

3.1 Interpretation of Parameters

We have the following density function:

$$f^*(x) = \exp(-\eta \cdot \log(\alpha + |x - x_0|))$$

The parameters $\mathbf{x_0}$, η, and α intervene in the density function. Additionnaly we have the weather transition probabilities ($\mathbf{p_{SN}}$, $\mathbf{p_{NS}}$, $\mathbf{p_{NP}}$, $\mathbf{p_{PN}}$) and the accident probabilities ($\mathbf{p_S}$, $\mathbf{p_N}$, $\mathbf{p_P}$) which are key components of defining the behavior of the model.

1. $\mathbf{x_0}$ represents a reference point relative to which the deviations $|\mathbf{x} - \mathbf{x_0}|$ influence the density. When x moves away from $\mathbf{x_0}$, the logarithm increases and the peak of the density shifts because η:
   - This means that values close to $\mathbf{x_0}$ are the least probable.
   - In an insurance context, $\mathbf{x_0}$ could represent a **minimum deductible** for claim amounts.
2. α acts as an adjustment term for the density. It shifts the contribution of $|\mathbf{x} - \mathbf{x_0}|$ in the logarithm:
   - If α is large, even if the deviation $|\mathbf{x} - \mathbf{x_0}|$ is significant, its effect is limited.
   - If α is small, the deviation α primarily influences the logarithm.
   - In an insurance context, α could represent a **base cost or penalty** for claim amounts.
3. η controls the effect of the logarithm on the density $f^*(x)$:
   - If η is large, the term $-\eta \cdot \log(\alpha + |\mathbf{x} - \mathbf{x_0}|)$ becomes smaller, leading to a faster decay of the density.
   - η can be seen as a factor for tail behavior in claim distributions. A higher η means the distribution is highly concentrated around η, and distant values are less probable.

Parameter	Value	Description
α	2	Adjustment term in the density function
x_0	1	Reference point for claim amounts
η	3.1	Scaling factor for the density function
N	1000	Number of insured motorcyclists
s	250	Threshold for conditional expectation
$\mathbf{p_{acc}}$	$[10^{-4}, 2 \cdot 10^{-4}, 5 \cdot 10^{-4}]$	Accident probabilities for weather states
$\mathbf{p_{trans}}$	$[0.3, 0.2, 0.3, 0.4]$	Transition probabilities for the Markov chain

Table 1: Parameters Used in this assignement

3.2 Interval for X

The problem is as follows: all we know from the statement is that x is positive, which is not very helpful since this is obvious. When a claim is reimbursed, it is not the insured party who pays it. Therefore, it is necessary to find an interval where the majority of the x values are located.
To find the interval $[a; b]$ where $f^*(x)$ is the most probable, we set $\varepsilon = 10^{-4}$ and look for the interval where $f^*(x) > \varepsilon$:

$$\exp(-\eta \cdot \log(\alpha + |x - x_0|)) > \varepsilon$$
$$-\eta \cdot \log(\alpha + |x - x_0|) > \log(\varepsilon)$$
$$\log(\alpha + |x - x_0|) < -\frac{\log(\varepsilon)}{\eta}$$
$$|x - x_0| < \exp\left(-\frac{\log(\varepsilon)}{\eta}\right) - \alpha$$
$$x_0 - \left(\exp\left(-\frac{\log(\varepsilon)}{\eta}\right) - \alpha\right) < x < x_0 + \left(\exp\left(-\frac{\log(\varepsilon)}{\eta}\right) - \alpha\right)$$

By substituting the parameter values provided, we find $x \in [-16.5; 18.5]$. Negative values are excluded since $x > 0$, and thus the interval is $x \in [0; 18.5]$ for verification. This interval covers 99.46729 % of the possible values of x.
Note: Two interpretations can be made here:

1. Values above 18.5 are generally very high and can therefore be considered as outliers due to the shape of the density.
2. These values represent extremely rare cases with very valuable claims that are important to consider.

We choose to consider the first case here, as we lack information about the type of insurance contracts involved, and the first case makes the code more efficient and easier to implement.

4 Distribution of the law of P_r

To simulate the law of P_r, several methods were tested. We initially started with the inversion method, relying on techniques we had covered during practical sessions. Unfortunately, This method was quickly set aside as it required excessive computation time, and achieving the best performance and efficiency was one of the requirements.

Rejection and Mixture Methods:
Rejection and mixture methods were unfortunately deemed unsuitable due to the structure of the density function:

- For the rejection method to work, it is necessary to find a constant c such that:
$$c \cdot g(x) \geq f(x), \forall x$$
over the domain of interest. However, due to the logarithmic form of the density, this process is extremely difficult.

- For the mixture method, the density function must be decomposable into a combination of simpler distributions, such as Gaussians or pure exponentials. The presence of the logarithmic term in the exponential complicates this approach significantly.

Consequently,we turned to the second method, which is both more intuitive and effective. The cumulative distribution function (CDF) method, which proved to be more natural and efficient. Further optimizations, such as the trapezoidal rule, were later introduced to enhance the accuracy and performance of the simulations.

Method used	n (amount of claims computed)	Computing time
Inversion method	1000	3.25 seconds
Cdf method	40 000 000	2.17 seconds
Cdf optimized	40 000 000	2.11 seconds

Table 2: Computing time comparaison for each method

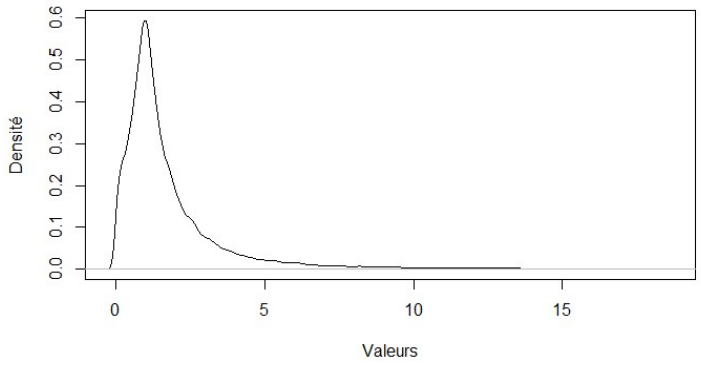

Figure 2: Probability Density Function (PDF) of Simulated Reimbursements

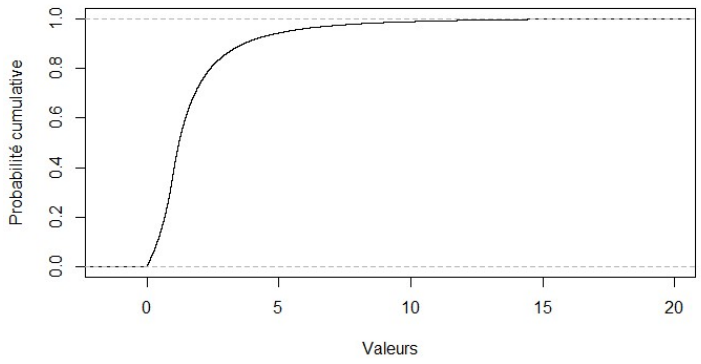

Figure 3: Cumulative Distribution Function (CDF) of Simulated Values

5 Hypothesis A

Under Hypothesis A, all N motorcyclists experience the same weather conditions. This simplifies the calculations, as we only need to simulate a single weather sequence of 365 days for $n.simul$ iterations. By assuming common weather, the dependency structure is straightforward, reducing computational complexity while maintaining the realism needed for the simulation.
The first algorithm followed this structure :

1. Generate the weather conditions for all simulations.
2. Generate all accidents occurring each day for each simulation.
3. Sum all these accidents to obtain the number of accidents per day, then sum again to get the total for each simulation.
4. Simulate the reimbursement amounts, then calculate the mean and the half-width of the confidence interval for simulations meeting the criteria.

5.1 Weather simulations

We tested three different approaches before being "satisfied" with the results:

- **Test n°1:** We defined a function that calculates the new state and called it as many times as needed within the function that returns the weather over 365 days. The initial state was chosen uniformly between 1 and 3 (at this stage, we had not yet realized that the initial state was supposed to be sunny).
- **Test n°2:** The initialization of the state is such that $X_0 = \text{Sunny}$. The transition between each state uses the cumulative method directly within the function, which avoids repeatedly calling the "new state" function.

We noticed (through profiling) that one of the "heavy" functions in our code is `runif` (generation of random uniform numbers), which is called multiple times.

- **Test n°3:** The third function generates, in a single step, a vector of 365 random values for all days, unlike the first two functions where a new random number is generated at each iteration.

Test	n_{simul}	Computing time
Test 1	3000	4.44 seconds
Test 2	3000	1.14 seconds
Test 3	3000	0.54 seconds

Table 3: Computing time of the weather functions for each method

5.2 Accident simulations

- **Test n°1:** Our first code created a list where each element is a matrix representing accidents for a simulation (N motorcyclists per day over 365 days).

  1. Accidents were generated using `rbinom`, which simulates binomial draws.
  2. Probabilities for each day were repeated for each entity (N).
  3. Finally, in the code for `msa`, we summed the results to obtain the total number of accidents per simulation.

  The identified problem is that this function is too slow because `rbinom` is called $n.simul$ times, generating a matrix for each call, which becomes computationally expensive in the long term.

- **Test n°2:** For our second code, we attempted to simulate accidents for multiple simulations using a double loop, which reduced the size of matrices generated by `rbinom`.

  1. Initialize an empty list where the results of each simulation are stored.
  2. Outer loop (i): For each simulation, the following steps are executed:
     - Initialize a vector of 365 days to store accidents for each day.
     - Inner loop (j): For each day, generate a number of accidents for the N motorcyclists with the associated probability.
  3. Filter the vector `accidents_per_day` to keep only days with accidents, then add it to the total list of accidents (reducing the number of elements in the list, which accelerates the rest of the program).
  4. Return the vector with all accidents.

The identified problem is that the inner loop for 365 days is repeated for each simulation, significantly slowing down the algorithm. Additionally, each call to `rbinom` is executed individually, resulting in $365 \times n.simul$ calls.

- **Test n°3 (final version):** For our third code, and the final version, we realized we could simply simulate accidents on a long vector with the following **size :** $365 * n_{simul} * N$, reorganize the results, and avoid iterative calls to `rbinom`.

  1. Initialize a list `accidents_tot` to store the results.
  2. Simulate all accidents in a single operation vectorized with `rbinom`.
  3. Organize the results in a matrix of size $(n.simul, 365)$.
  4. Compute the sums of accidents directly using `rowSums` to get the total number of accidents.
  5. Return the totals in vector form.

  This is the code we decided to keep, as it is the most efficient.

Test	n_{simul}	N (number of bikers)	Computing time
Test 1	1000	1000	0.50 seconds
Test 2	1000	1000	0.26 seconds
Test 3	1000	1000	0.02 seconds

Table 4: Computing time of the accident functions for each method

6 Hypothesis B

For Hypothesis B, the problem lies in the fact that each motorcyclist has their own weather conditions, which quickly leads to the creation of a very large weather matrix.
Our first attempt involved using a double loop: for each simulation, we calculated the weather for each motorcyclist and placed it in the row corresponding to that motorcyclist.
For accidents, we followed a similar approach. We first used as input the matrix containing accident probabilities for each simulation and each day, as well as the number of motorcyclists and simulations. Using the **rbinom** function, we initialized a vector containing all accidents. Then, we reorganized this vector into a matrix and calculated the row sums of this matrix.

Function	n_{simul}	N (number of bikers)	Computing time
Weather	100	1000	18.22 seconds
Accidents	100	1000	1.35 seconds

Table 5: Computing times for both functions

We didn't dig much deeper afterwards as we found a solution that allowed us to considerably save time for weather generation as it is the biggest problem in our MSB function.

7 Stationary Measure and validity domain

7.1 Stationary Measure

While looking for a way to optimize *msb*, which was extremely slow, we realized that the main problem—whether for *msa* or *msb*—was related to the generation of weather. The state for each day had to be generated day by day following the transition probabilities, which made the code extremely slow in both cases. However, we know that :

1. If the chain is irreducible, and
2. It has a finite number of states,

then it admits a **unique stationary measure**.
Moreover, if:

1. The chain is irreducible,
2. The chain is aperiodic (which is the case here since $P(S, S) > 0$, so S is aperiodic, and since the chain is irreducible, it is also aperiodic)
3. It admits a unique stationary measure μ,

then:

$$X_n \xrightarrow{\text{Law}} \mu$$

We can then make a function that computes the stationary measure based on the initial parameters and use it to generate weather. This means that instead of generating the weather states one by one, taking into account the previous state, we could use the stationary measure to generate **all weather states at once** with the `sample` function, and then reorder them into the desired dimensions. This approach provides the following results

Test	n_{simul}	Number of bikers	Computing time
Test 1 H_A	3000	1000	4.44 seconds
Test 2 H_A	3000	1000	1.14 seconds
Test 3 H_A	3000	1000	0.54 seconds
Stationary measure for A	100 000	1000	0.46 seconds
Test 4 H_B	100	1000	18.22 seconds
Stationary measure for B	100	10 000	4.60 seconds

Table 6: Computing time of the weather functions compared to the function using the stationary measure

So we decided to implement this method for both *msa* and *msb*. **Note :** the number of bikers has no influence on the computing time of the weather generating function as they all share the same weather.

7.2 Validity domain : Gelman-Rubin criteria

We observed that using the stationary Markov chain leads to a significant improvement in performance. However, it is necessary to verify whether we are actually justified in using it. Indeed, although the Markov chain admits a stationary measure, we do not know from which point it converges!

The **Gelman-Rubin criterion** (commonly referred to as the **potential scale reduction factor**, $\hat{R}$) is a statistical tool designed to evaluate the convergence of multiple Markov chains in simulations, particularly within the context of **Markov Chain Monte Carlo (MCMC)** methods. It works by comparing the **inter-chain variance** (variance between chains) with the **intra-chain variance** (variance within each chain). The underlying principle is that, once the chains have converged, these variances should become comparable, signifying that all chains are sampling from the same stationary distribution.

The computation of the criterion involves running several independent chains and, at each iteration, calculating both the variance within each chain and the variance between the chains. The $\hat{R}$ statistic is derived by comparing these two

variances, and is given by the following formula:

$$\hat{R} = \sqrt{\frac{N-1}{N} + \frac{B}{W \cdot N}}$$

where N is the number of chains, B is the variance between chains, and W is the variance within chains.

A value of $\hat{R}$ close to 1 (typically less than 1.1) indicates that the chains have likely converged to the stationary distribution. Conversely, an $\hat{R}$ value significantly greater than 1 suggests that the chains have not yet reached convergence.

In our specific case, for Markov chains with 365 states and 1500 simulations, we find:

$$\hat{R} = 0.9999178 \approx 1$$

We observed that the stationary measure is valid for $n \geq 1500$ simulated chains of states, which corresponds to:

- For Hypothesis A: $n.simul_A \geq 1500$,
- For Hypothesis B: $n.simul_B \geq \left\lfloor \frac{1500}{N} \right\rfloor = 2$.

In both cases, this condition is well satisfied, and we can confidently say that the stationary measure is valid, whether on the submission site (under its constraints) or on our own computer.

8 Hypothesis B with C++

Finally, the last part of the assignment involved using the `Rcpp` library and C++ to speed up the function. To achieve this, we performed a diagnostic on `msb` for $n_{\text{simul}} = 600$ and $N = 1000$ (the Monte Carlo function of Hypothesis B with the stationary measure) and found the following:

Function	**self.time**	**self.pct**
`rbinom`	6.18	42.74
`sample.int`	2.18	15.08
`array`	1.86	12.86
`aperm.default`	1.80	12.45
`accident_MB2`	0.68	4.70
`sample`	0.68	4.70
`as.vector`	0.26	1.80

Table 7: Partial Diagnosis of MSB function (some functions were ommitted due to irrelevance

We can directly observe that the functions that take the most "time" and slow down our program the most are the `rbinom` and `sample` functions, used to generate the accidents and the weather. These will therefore be the functions we will attempt to implement in C++.

8.1 Rbinom in C++

In order to optimize the performance of the simulation, we implemented the `rbinom` function in C++. This function is used to generate binomial random variables, which are critical in simulating the accidents.

```
#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;
}
```

Function	Computing time
`Rbinom in R`	3.92 seconds
`Rbinom_cpp`	0.32 seconds

Table 8: Rbinom in R vs C++ for $n_{\text{simul}} = 400$

We can see a great improvement in performance and we thus decide to keep this change.

8.2 Sample in C++

We then coded the accident function, which integrates the sample function (considered time-consuming), in C++. The corresponding code (for the sample function) is presented below

```
#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);
}
```

We proceed in the same manner : we calculate and compare the execution times of both versions to verify if this change improves efficiency. We observed the following issue: the `.Call` function appeared and it takes a considerable amount of time during the execution of our function.
Upon further investigation, we discovered that this problem is caused by the use of `std::vector<int>` in our function. Indeed, `std::vector` requires an explicit conversion with the `wrap` method, which leads to many calls to `.Call`. To resolve this issue, we decided to use `IntegerVector`, which is native to `Rcpp`, reducing the calls to `.Call` and improving performance.

```
#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;
}
```

Function	**Sampling Time**
`MSB without C++`	14.46 seconds
`MSB with only Rbinom in C++`	6.22 seconds
`MSB with Rbinom and first attempt at sample in C++`	6.78 seconds
`MSB with Rbinom and second attempt at sample in C++`	5.86 seconds

Table 9: Sampling Time for Different MSB Implementations for $n_{simul} = 400$

9 Suggestions

Insurance Costs : We could first start by studying the difference between the values of `MSA` and `MSB` for the larger n_{simuls}. Indeed, the threshold chosen by the professor is equal to 1.5 times the mean of the expectation due to the limits of the submition website, this threshold is less often exceeded. It would be interesting to see what happens with a larger number of simulations.

How to Make the Model More Realistic/Precise?

- Improve memory limits on the website for submissions : the website had strong memory limits as it would stop if a vector required more than 554.2 MO of memories, in this context, where the weather matrix of B contains $n_{simul} \times 365 \times N$ states it seems contradictory to have such a short limit.
- **Improve the accuracy of the weather**: The following suggestions would significantly increase the model but also amplify the memory constraints :
    1. **Temporal dependence**: We could use a time-indexed Markov chain (period [0, 365]) with transition probabilities varying by season (winter, summer, etc.).
    2. **Geographical dependence**: We could create several Markov chains for each region (rather than per motorcyclist), performing simulations while accounting for the distribution of motorcyclists by region.
    3. **Additional states**: We could add extra meteorological states like snow, fog, etc.
- **Improve the realism of accidents and reimbursements**:
    1. Use different densities to reimburse, linking them to various factors (weather, motorcyclist profile, etc.) to better reflect the severity of accidents.
    2. Introduce **correlations** between accidents, assuming they are correlated, by modeling scenarios where one accident can cause others (for example, a group fall).

10 Conclusion and Performance

This project offered a focused exploration of Monte Carlo simulations applied to motorcycle insurance, deepening my understanding of modeling and simulating accident risks and reimbursements.
By working with Markov chains, I gained insights into using stochastic processes for complex systems and improved simulation efficiency through C++ and Rcpp. Experimenting with different assumptions (not detailed here), such as weather and accident probabilities, helped me understand how parameter choices influence outcomes.
The project emphasized the importance of balancing model accuracy and computational feasibility, highlighting the trade-offs between computational limits and model complexity that we had to navigate throughout.

The following code ranked as follows :

Function	n_{simul}	Time	Rank
Density	63 095 734	8.9 seconds	4/41
MSA (Hypothesis A)	50119	24.1 seconds	6/41
MSB (Hypothesis B)	316	27.5 seconds	1/41
MSB.C (Hypothesis B using C++)	1585	55.4 seconds	4/41

Table 10: Performance of my final code

11 Final Code

```
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, 18.5, 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)
}
```

```
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)
}

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) {
  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)
}

msb <- 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 <- 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, Numer
    std::mt19937 rng(std::random_device{}());
    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::generate_canonical<double, 10>(rng) < pacc[weather_state]) {
                total_accidents++;
              }
```

```
}
accidents(sim, motard) = total_accidents;
}
}

return accidents;
}
')
  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)

  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
  ))
}
```