

# DOCUMENT 3

## ÉVOLUTION D'ALGORITHMES ET DE PROGRAMMES AVEC R SIMULATIONS ET CALCULS DE LOIS GÉOMÉTRIQUES TRONQUÉES ET DE LOIS BINOMIALES

Hubert RAYMONDAUD

LEGTA LOUIS GIRAUD – CARPENTRAS – SERRES – MINISTÈRE DE L'AGRICULTURE

### Table des matières

<b>A – SIMULER UNE LOI BINOMIALE.....</b>	<b>2</b>
1° Programme original tiré du manuel Nathan Hyperbole, <i>Première S</i> , page 312 n°30 partie 1.....	2
2° Première évolution : utilisation de <code>sample</code> au lieu de <code>runif()</code> ( <code>randon()</code> ).....	3
3° Deuxième évolution : utilisation de l'objet "urne" de R.....	3
4° Troisième évolution : la boucle <code>for</code> est remplacée par l'échantillonnage <code>sample</code> et le comptage des 1 dans la liste simulée.....	4
5° Quatrième évolution : introduction des paramètres.....	4
6° Cinquième évolution : on change de programme pour simuler une distribution.....	5
7° Sixième évolution : modèle d'urne avec boules non numérotées, tirage avec remise.....	5
8° Septième évolution : on simule la répétition d'une alternative de Bernoulli .....	6
9° Huitième évolution : utilisation de la fonction <code>rbinom()</code> , tableaux complets des probabilités et des fréquences, et superposition des représentations graphiques.....	7
10° Programmation de la partie 2 du T.P. Nathan Hyperbole, <i>Première S</i> , page 312 n°30.....	9
11° Quelques variations autour de la superposition de graphiques pour comparer fréquences simulées et probabilités.....	10
12° Un autre exemple de simulation d'une distribution : hyperbole page 321 n°71.....	11
13° Exemple d'algorithme et de programme R de calcul d'une probabilité cumulée avec une loi binomiale : hyperbole, <i>Première S</i> , page 338 n°41.....	12
<b>B – SIMULER UNE LOI GÉOMÉTRIQUE TRONQUÉE.....</b>	<b>13</b>
1° Programme original tiré du manuel Nathan Hyperbole, <i>Première S</i> , page 290 n°23 .....	13
2° Première évolution : utilisation de <code>sample</code> au lieu de <code>runif()</code> ( <code>randon()</code> ).....	14
3° Deuxième évolution : utilisation de l'objet R roulette.....	15
4° Troisième évolution : la boucle <code>while</code> est remplacée par l'échantillonnage <code>sample</code> et l'utilisation de <code>which</code> qui renvoie les rangs des composantes d'une liste.....	15
5° Quatrième évolution : introduction des paramètres.....	17
6° Cinquième évolution : on change de programme pour simuler une distribution.....	17
7° Sixième évolution : on remplace la roulette par une urne échantillonnée avec remise.....	18
8° Septième évolution : on simule des épreuves de Bernoulli identiques et indépendantes.....	18
9° Huitième évolution : tableaux complets des fréquences et des probabilités et superposition des représentations graphiques.....	19
10° Neuvième évolution : utilisation d'une fonction externe <code>geotronk2()</code> pour calculer les probabilités.....	21
11° Calcul numérique de l'espérance et de la variance d'une distribution géométrique tronquée.....	22
<b>C – FAC-SIMILÉ DES ÉNONCÉS.....</b>	<b>24</b>

# ÉVOLUTION D'ALGORITHMES ET DE PROGRAMMES AVEC R SIMULATIONS ET CALCULS DE LOIS GÉOMÉTRIQUES TRONQUÉES ET DE LOIS BINOMIALES

Hubert RAYMONDAUD

LEGTA LOUIS GIRAUD – CARPENTRAS – SERRES – MINISTÈRE DE L'AGRICULTURE

## A – SIMULER UNE LOI BINOMIALE

### 1° Programme original tiré du manuel Nathan Hyperbole, *Première S*, page 312 n°30 partie 1

#### Algorithmique



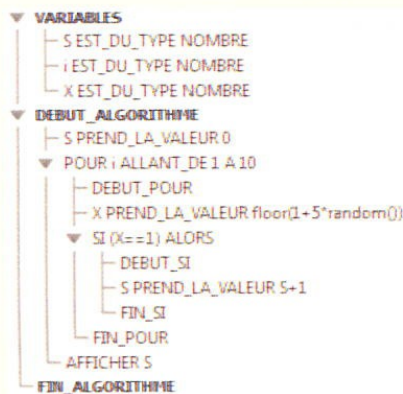
#### 30 Simuler la loi binomiale

**OBJECTIF** Simuler la loi binomiale à l'aide d'un programme.

##### 1. Une première expérience

Une urne contient cinq boules numérotées de 1 à 5. On tire au hasard, successivement et avec remise 10 boules de l'urne. On appelle succès, lors d'un tirage, l'apparition de la boule numérotée 1.

On simule cette expérience avec le programme suivant.



- `random ()` renvoie un nombre aléatoire  $x$  tel que  $0 \leq x < 1$ .
- `floor (1 + 5 * random ())` permet d'obtenir un nombre entier aléatoire compris entre 1 et 5 suivant la loi équirépartie.

On exécute pas à pas ce programme. Les variables et leur évolution sont résumées dans le tableau ci-dessous.

<b>i</b>	1	2	3	4	5	...
<b>X</b>	2	1	4	3	1	...
<b>S</b>	0	0	1	1	1	2

- Que représente la variable  $S$  ?
- Quelle est la loi de probabilité suivie par le nombre de succès ?

##### 2. Généralisation

L'urne contient  $B$  boules numérotées de 1 à  $B$ . On tire au hasard, successivement et avec remise  $n$  boules de l'urne.  $m$  est un entier compris entre 1 et  $B$ , on appelle succès, lors d'un tirage, l'apparition d'une boule dont le numéro est compris entre 1 et  $m$ .

- Écrire un algorithme afin de simuler cette expérience aléatoire.

La variable  $n$  sera lue en entrée. Les variables  $B$  et  $m$  seront initialisées au début de l'algorithme.

- Le nombre de succès suit une loi binomiale. Quels sont ses paramètres ?

- Écrire le programme associé à l'aide d'un logiciel ou d'une calculatrice et le tester.

Tous les codes **R** de ce chapitre figurent dans le fichier texte *EvolBinoAlgoHyp312\_30.r*. Il peut être ouvert avec l'éditeur spécifique à coloration syntaxique **RStudio**.

Ce programme utilise :

- Un générateur de nombres pseudo aléatoire uniforme entre 0 et 1, `runif ()` (qui est la transposition de `random()`), avec la fonction `floor ()`, pour simuler des entiers de distribution uniforme entre 1 et 5. Il est intéressant de noter que `runif (1, 0, 1)` nécessite de préciser le nombre de tirages effectués (1) et les bornes de l'intervalle de la loi uniforme utilisée (0, 1) ;
- Une boucle Pour (`for ()`) ;
- Avec un test Si (`if ()`) ;
- Un compteur manuel  $S$ , du nombre de tirage(s) de la boule numéro 1.

Ces difficultés rendent la construction du programme peu abordable pour une initiation avec des élèves de *Première*.

De plus, la syntaxe Algobox rend le programme peu lisible : les déclarations de variables sont inutiles pour les applications qu'en font les élèves, les indentations des boucles et des tests sont inutilement chargées. Un autre exemple, page 321 n°71, suivra, qui illustre mieux ce propos.

Sa traduction ou plutôt transposition en **R** en simplifie déjà significativement la lecture, comme on le voit ci-dessous :

```

#-- AlgoHyp312_30()--simulation d'une valeur
# algorithme de l'énoncé -- runif
# Modèle d'urne avec boules numérotées
# tirage avec remise
AlgoHyp312_30 <- function(){
  S <- 0
  for(i in 1:10){# 1:10 génère la liste
                #des entiers de 1 à 10
    x <- floor(1 + 5 * runif(1, 0, 1))
    if(x == 1) {S <- S + 1}
  }
# Affichage
  cat("\nNombre de boules n° 1=", S, "\n")
}

```

## Programme Texas sur schéma de Bernoulli

```

:Prompt N, P, R
:EffListe L5,L6
:For(I,1,R)
:0→S
:For(K,1,N)
:NbrAléa→A
:If A≤P
:Then
:S+1→S
:End
:End
:S→L6(I)
:End
:L6/N→L5
:Stats 1-Var L6
:EffEcr
:Disp n,  $\bar{x}$ , Med, Sx
:Pause
:Stats 1-Var L5
:EffEcr
:Disp n,  $\bar{x}$ , Med, Sx
:Pause
:GraphNAff
:0→Xmin:1→Xmax:.1→Xgrad
:0→Ymin:R/5→Ymax:(Ymax-Ymin)/10→Ygrad
:Graph1 (Histogramme, L5)
:AffGraph
:Pause
:GraphNAff
:Graph2 (GraphBoitMoust, L5)
:AffGraph

```

## 2° Première évolution : utilisation de `sample` au lieu de `runif()` (`randon()`)

- `sample(1:5, 1)` tire un échantillon aléatoire et simple de taille 1 dans la liste des entiers de 1 à 5.
- `1:5` génère la liste des nombres entiers de 1 à 5.
- le reste du programme est identique au précédent, avec les mêmes difficultés de la boucle et du test.

L'instruction `sample(1:5, 1)` est mieux comprise par les élèves que `floor(1+5*(runif(1, 0, 1)))`. De plus elle est directement liée à la notion d'échantillon aléatoire et simple, vue en cours.

```

#-- AlgoHyp312_30_1()--simulation d'une valeur avec sample 1
# Modèle d'urne avec boules numérotées, tirage avec remise
AlgoHyp312_30_1 <- function(){
  S <- 0
  for(i in 1:10){
    x <- sample(1:5, 1)
    if(x == 1) {S <- S + 1}
  }
# Affichage
  cat("Nombre de boules n° 1 =", S, "\n")
}

```

```

AlgoHyp312_30_1()
Nombre de boules n° 1 = 4

```

## 3° Deuxième évolution : utilisation de l'objet "urne" de R

- La seule différence avec le programme précédent est l'introduction de l'objet `urne` qui est la liste des entiers de 1 à 5 qui simule l'urne avec les 5 boules numérotées (`urne <- 1:5`). On simplifie ainsi la simulation du tirage dans l'urne, tout en le rapprochant de l'expérience réelle.

```

#-- AlgoHyp312_30_2()--simulation d'une valeur avec sample dans urne
# Modèle d'urne avec boules numérotées, tirage avec remise
AlgoHyp312_30_2 <- function(){
  S <- 0
  urne <- 1:5
  for(i in 1:10){
    x <- sample(urne, 1)
    if(x == 1) {S <- S + 1}
  }
# Affichage
cat("\nNombre de boules n° 1=", S, "\n")
}

```

#### 4° Troisième évolution : la boucle for est remplacée par l'échantillonnage sample et le comptage des 1 dans la liste simulée

- L'instruction **sample(urne, 10, replace = TRUE)** tire un échantillon de taille 10 avec remise, simulant ainsi les 10 tirages dans l'urne. Les résultats sont mis dans la liste **x**. On évite ainsi le recours à une boucle plus difficile à gérer et plus consommatrice de temps.
- Il s'agit maintenant de compter combien de fois la boule numérotée 1 est tirée. C'est fait par la commande **S <- sum(x == 1)** dans laquelle **x == 1** teste chaque composante de la liste par rapport à 1, générant une liste de TRUE ou FALSE. La fonction **sum()** transforme les TRUE FALSE en 1, 0 et en fait la somme.
- Il ne reste plus qu'à afficher le résultat contenu dans **S**, simulant une valeur d'une distribution binomiale.

La compréhension du programme s'en trouve simplifiée et on passe de 9 lignes à 7.

```

#-- AlgoHyp312_30_3()--simulation d'une valeur avec sample dans urne
# Modèle d'urne avec boules numérotées, tirage avec remise
AlgoHyp312_30_3 <- function(){
  urne <- 1:5
  x <- sample(urne, 10, replace = TRUE)
  S <- sum(x == 1)
# Affichage
cat("\nNombre de boules n° 1=", S, "\n")
}

```

La version en "lignes de commandes" permet de comprendre ce qui se passe :

```

#-- AlgoHyp312_30_3--simulation d'une valeur avec sample dans urne
# Modèle d'urne avec boules numérotées, tirage avec remise
> (urne <- 1:5)
[1] 1 2 3 4 5
> (x <- sample(urne, 10, replace = TRUE))
[1] 1 2 3 4 1 4 3 5 2 2
x == 1
[1] TRUE FALSE FALSE FALSE TRUE FALSE FALSE FALSE FALSE
> (S <- sum(x == 1))
[1] 2

```

#### 5° Quatrième évolution : introduction des paramètres

- Avec les programmes précédents, si l'on veut changer de valeur du nombre de boules, du nombre de tirages, du numéro de la boule dont on attend la sortie, il faut modifier ces valeurs en éditant (modifiant) le programme lui même, ce qui est long et source d'erreurs.
- Avec **R**, il suffit d'introduire ces valeurs en tant que paramètres en tête de la fonction programmée : **function(nbboules = 5, nbtirages = 10, numero = 1)**. Ce sont ces valeurs qui seront utilisées par défaut si aucune valeur n'est indiquée lors de l'exécution de la fonction. Sinon, il suffit de saisir les valeurs voulues, lors de l'exécution de la fonction : **AlgoHyp312\_30\_4(nbtirages = 15, nbboules = 10)**. On peut omettre les noms des paramètres si les valeurs sont saisies suivant l'ordre programmé dans la fonction.

```

#-- AlgoHyp312_30_4()-- simulation d'une valeur
# avec paramètres
# Modèle d'urne avec boules numérotées, tirage avec remise
AlgoHyp312_30_4 <- fonction(nbboules = 5, nbtirages = 10,
                           numero = 1){
  urne <- 1:nbboules
  x <- sample(urne, nbtirages, replace = TRUE)
  S <- sum(x == numero)
# Affichage
  cat("\nNombre de boules n°", numero, "=", S, "\n")
}

```

```

AlgoHyp312_30_4(8, 15, 3)
Nombre de boules n° 3 = 4

```

## 6° Cinquième évolution : on change de programme pour simuler une distribution

- On change d'objectif puisqu'il s'agit de simuler, non plus **une seule valeur** du nombre fois où la boule numéro 1 a été obtenue, mais la distribution de la variable **X**, nombre fois où la boule numéro 1 a été obtenue lors de 10 tirages.
- On va donc introduire une boucle de **nbsim = 2000** simulations. Le résultat de chacune de ces simulations est stocké dans la liste (vecteur) **seriesim** dont la composante **i** est **seriesim[i]**. À la fin des simulations **seriesim** contient 2000 valeurs de la variable **X**.
- Pour décrire cette série statistique, l'instruction **distsim <- table(seriesim)** en fait le tableau des effectifs et l'instruction **barplot(distsim / nbsim)** produit le diagramme en barre qui illustre le tableau des fréquences correspondant.
- On illustre ainsi une distribution simulée de la variable **X**, sans avoir besoin de connaître la distribution exacte (le modèle mathématique) de cette variable.
- On peut alors faire varier les différents paramètres introduits dans la fonction, pour en illustrer l'influence : si l'on change de **numéro**, la distribution va-t-elle changer "significativement" ?

```

#-- AlgoHyp312_30_5()-- Distribution simulée de
# la variable. Modèle d'urne avec boules numérotées
#, tirage avec remise
AlgoHyp312_30_5 <- fonction(nbboules = 5,
                             nbtirages = 10, numero = 1, nbsim = 2000){
  urne <- 1:nbboules
  seriesimul <- NULL
  for(i in 1:nbsim){
    x <- sample(urne, nbtirages, replace = TRUE)
    seriesimul[i] <- sum(x == numero)
  }
  tabloFreq <- table(seriesimul) / nbsim
# Affichages
  cat("Distribution simulée du nombre de boules n°",
      numero, ":\n")
  print(tabloFreq)
  barplot(tabloFreq)
}

```

```

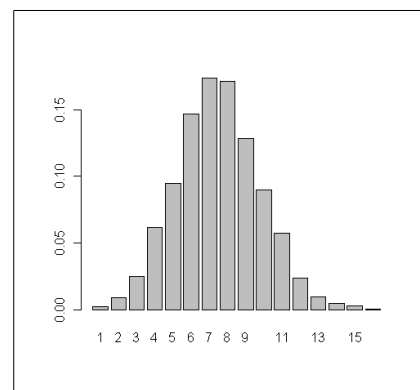
AlgoHyp312_30_5(4, 30, 3)

```

```

Distribution simulée du nombre de boules n° 3 :
seriesimul
  1     2     3     4     5     6     7     8     9
0.0020 0.0090 0.0250 0.0615 0.0945 0.1465 0.1735 0.1710 0.1285
 10    11    12    13    14    15    16
0.0895 0.0575 0.0240 0.0095 0.0045 0.0030 0.0005

```



## 7° Sixième évolution : modèle d'urne avec boules non numérotées, tirage avec remise

- Les boules numérotées sont remplacées par des boules de couleur, de composition fixée :  
**urne <- rep(c("Rouge", "Autre"), c(nbrouges, nbtot - nbrouges))**. Cette instruction crée une **urne** avec **nbrouges** boules **Rouge** et **(nbtot - nbrouges)** boules d'une



**Autre** couleur.

- Dans laquelle on effectue des tirages avec remise :  
**sample(urne, nbtirages, replace = TRUE)**.
- Les résultats des tirages sont donc **Rouge** ou **Autre** et non plus des numéros. Il faut donc effectuer des tests sur des chaînes de caractères, pour compter le nombre de boules rouges, ce qui ne pose aucun problème de programmation : **sum(x == "Rouge")**. Le reste du programme n'a pas changé.
- La simulation "mime", au plus près, une expérience réelle.

```
##-- AlgoHyp312_30_6-- Distribution simulée de la variable
# Modèle d'urne avec boules de couleur, tirage avec remise
AlgoHyp312_30_6 <- function(nbrouges = 1, nbtot = 5, nbtirages = 10,
  nbsim = 2000){
  urne <- rep(c("Rouge", "Autre"), c(nbrouges, nbtot - nbrouges))
  seriesimul <- NULL
  for(i in 1:nbsim){
    x <- sample(urne, nbtirages, replace = TRUE)
    seriesimul[i] <- sum(x == "Rouge")
  }
  tabloFreq <- table(seriesimul) / nbsim
# Affichages
cat("\nDistribution simulée du nombre de boules Rouges:\n")
print(tabloFreq)
barplot(tabloFreq)
}
```

La version en "lignes de commandes" permet de comprendre ce qui se passe :

```
##-- AlgoHyp312_30_6-- Distribution simulée de la variable
# Modèle d'urne avec boules de couleur, tirage avec remise
> nbrouges = 1 ; nbtot = 5 ; nbtirages = 10 ; nbsim = 20
> (urne <- rep(c("Rouge", "Autre"), c(nbrouges, nbtot - nbrouges)))
[1] "Rouge" "Autre" "Autre" "Autre" "Autre"
> seriesimul <- NULL
> (x <- sample(urne, nbtirages, replace = TRUE))
[1] "Rouge" "Autre" "Autre" "Autre" "Rouge" "Autre" "Autre"
[8] "Autre" "Autre" "Autre"
> (seriesimul[1] <- sum(x == "Rouge"))
[1] 2
> for(i in 1:nbsim){
+   x <- sample(urne, nbtirages, replace = TRUE)
+   seriesimul[i] <- sum(x == "Rouge")
+ }
> seriesimul
[1] 3 2 0 3 2 2 4 3 4 2 2 2 2 3 5 0 1 2 1
> (tabloFreq <- table(seriesimul) / nbsim)
seriesimul
 0  1  2  3  4  5
0.10 0.10 0.45 0.20 0.10 0.05
```

## 8° Septième évolution : on simule la répétition d'une alternative de Bernoulli

- Les deux issues de l'alternative sont "**succes**" et "**echec**",
- Le tirage avec remise est effectué en fonction des probabilités respectives **p** et **1 - p** par l'instruction :  
**sample(deuxalternatives, n, prob = c(p, 1 - p), replace = TRUE)**.
- On compte le nombre de succès avec : **sum(x == "succes")**
- Le reste du programme n'a pas changé.

```

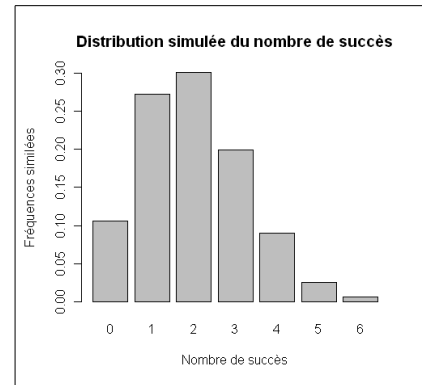
#-- AlgoHyp312_30_8()-- Distribution simulée de la
# variable
# n épreuves de Bernoulli répétées (indépendantes)
AlgoHyp312_30_8 <- function(n = 10, p = 1 / 5,
                           nbsim = 2000) {
  deuxalternatives = c("succes", "echec")
  seriesimul <- NULL
  for(i in 1:nbsim){
    x <- sample(deuxalternatives, n, prob = c(p, 1 - p),
               replace = TRUE)
    seriesimul[i] <- sum(x == "succes")
  }
  tabloFreq <- table(seriesimul) / nbsim
# Affichages
cat("Distribution simulée du nombre de succès :\n")
print(tabloFreq)
barplot(tabloFreq, ylab = "Fréquences simulées",
        xlab = "Nombre de succès",
        main = "Diagramme en barres")
}

```

```

AlgoHyp312_30_8()
Distribution simulée du nombre de succès :
seriesimul
0      1      2      3      4      5      6
0.1060 0.2725 0.3010 0.1990 0.0900 0.0255 0.0060

```



## 9° Huitième évolution : utilisation de la fonction `rbinom()`, tableaux complets des probabilités et des fréquences, et superposition des représentations graphiques

- On vient enfin d'identifier un modèle binomial, on peut donc utiliser les outils disponibles dans R concernant ce modèle.
- La fonction `R rbinom(nbsim, n, p)` génère nbsim nombres aléatoires de distribution binomiale. On n'a donc plus besoin de boucle nbsim, elle est comprise dans `rbinom()`, ce qui fait gagner en simplicité de programmation et en rapidité d'exécution.
- Par contre on ne peut plus utiliser d'objet urne, de boules, d'alternatives succès échec, ce qui diminue les qualités pédagogiques du programme, clarté, ressemblance avec une expérience réelle...
- Ayant économisé des lignes de programme sur la simulation, on peut en profiter pour améliorer sensiblement la représentation graphique : on va superposer la représentation graphique en points des valeurs de la distribution binomiale (`points(ValeursX, probabino, ...)`), à la représentation graphique en barres (lignes verticales), des fréquences simulées obtenues pour chacune des valeurs de la variable X : (`plot(ValeursX, tableauFreq, type = "h", ...)`).

La distribution de probabilité contenue dans la liste `probabino` est obtenue avec la fonction native `R dbinom(...)`.

- On bloque l'échelle des ordonnées (`ylim = c(0, echely)`), de façon à mieux observer la constance des valeurs de la distribution de la variable aléatoire X.
- Pour que la superposition graphique soit possible, il faut avoir toutes les valeurs possible de X dans le tableau des fréquences (`tableauFreq`), alors même que certaines valeurs de X peuvent ne pas faire partie des résultats de la simulation. Il faut donc gérer l'affectation des valeurs des fréquences simulées de façon particulière, c'est ce que fait la ligne :

```
tableauFreq[as.numeric(names(tableauFreq)) + 1] <- tableauFreq
```

- Voyons sur un exemple en lignes de commandes comment elle fonctionne.

```

> n = 6 ; p = .3 ; nbsim = 20
> (tableauFreq <- rep(0, n + 1))
[1] 0 0 0 0 0 0 0
> (ValeursX <- 0:n)
[1] 0 1 2 3 4 5 6
> (names(tableauFreq) <- ValeursX)
> tableauFreq
0 1 2 3 4 5 6
0 0 0 0 0 0 0
> seriesimul <- rbinom(nbsim, n, p)
> seriesimul
[1] 2 2 2 1 1 2 2 1 2 1 1 2 2 3 2
1 2 1 2 5

```

Que fait :

```
tableauFreq[as.numeric(names(tableauFreq)) + 1] <- tableauFreq
```

`names(tableauFreq)` repère les intitulés des composantes de la liste `tableauFreq`, qui sont "1" "2" "3" "5". Il manque "0" et "6".

`as.numeric(names(tableauFreq))` les transforme en nombres entiers, car les intitulés sont des caractères.

`[as.numeric(names(tableauFreq)) + 1]` les prend comme indices de la liste `tableauFreq` avec + 1 car il y a un décalage de 1 entre les valeurs de la variable X qui commencent à 0 et les indices de la liste qui commencent à 1.

`tableauFreq[as.numeric(names(tableauFreq)) + 1]` indique donc les

```
> (tabloFreq <-
table(series simul)/nbsim)
series simul
 1 2 3 5
0.35 0.55 0.05 0.05
```

indices des composantes de la liste `tableauFreq` auxquelles vont être affectées les valeurs de la liste `tabloFreq`.  
On obtient donc au final :

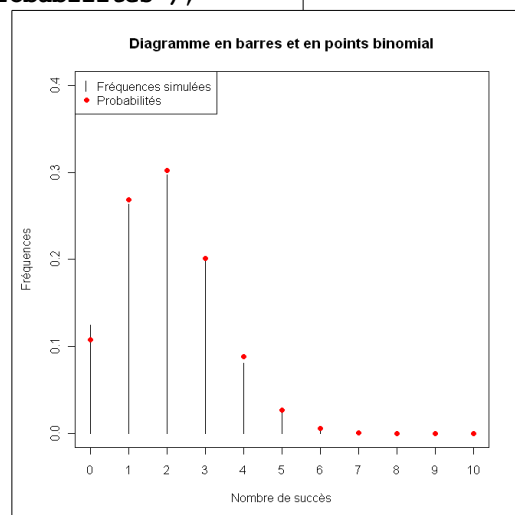
```
> tableauFreq[as.numeric(names(tabloFreq)) + 1] <- tabloFreq
> tableauFreq
 0 1 2 3 4 5 6
0.00 0.35 0.55 0.05 0.00 0.05 0.00
```

• Application dans la fonction `AlgoHyp312_30_9`.

```
##-- AlgoHyp312_30_9)-- Distribution simulée de la variable
# FONCTION rbinom() : plus besoin de boucle nbsim
# Superposition dans le graphique des valeurs du modèle binomial
# Il faut un tableau des fréquences simulées avec toutes les valeurs de X
AlgoHyp312_30_9 <- function(n = 10, p = 1 / 5, nbsim = 2000, echely = .4){
  series simul <- rbinom(nbsim, n, p)
  tabloFreq <- table(series simul) / nbsim
  ValeursX <- 0:n
  tableauFreq <- rep(0, n + 1)
  names(tableauFreq) <- ValeursX
  tableauFreq[as.numeric(names(tabloFreq)) + 1] <- tabloFreq
  probabino <- dbinom(ValeursX, n, p)
  names(probabino) <- ValeursX
# Affichage des résultats et des graphiques
  cat("Distribution simulée du nombre de succès :\n")
  print(tableauFreq)
  cat("\nDistribution du nombre de succès (modèle binomial) :\n")
  print(probabino)
  plot(ValeursX, tableauFreq, type = "h", xaxp = c(0, n, n), ylim = c(0, echely),
  ylab = "Fréquences", xlab = "Nombre de succès",
  main = "Diagramme en barres et en points binomial")
  points(ValeursX, probabino, pch = 21, col = "red", bg = "red")
  legend(x = "topleft", legend = c("Fréquences simulées", "Probabilités"),
  pch = c(124, 21), pt.cex = c(1, 1),
  col = c("black", "red"), pt.bg = c(NA, "red"))
}
```

```
##-- AlgoHyp312_30_9-- Exemple de résultats
> AlgoHyp312_30_9()
Distribution simulée du nombre de succès :
 0 1 2 3 4 5 6 7 8
0.1245 0.2635 0.2975 0.2040 0.0810 0.0245 0.0045 0.0005 0.0000
 9 10
0.0000 0.0000

Distribution du nombre de succès (modèle binomial) :
 0 1 2 3 4
0.1073741824 0.2684354560 0.3019898880 0.2013265920 0.0880803840
 5 6 7 8 9
0.0264241152 0.0055050240 0.0007864320 0.0000737280 0.0000040960
 10
0.0000001024
```





- On peut superposer à l'“histogramme” de la série discrète simulée, centrée réduite, la densité de la gaussienne centrée réduite. Ce traitement permet aussi d'illustrer graphiquement le théorème de Moivre-Laplace, dans lequel on centre et on réduit une variable binomiale. La distribution de cette variable centrée réduite converge vers un loi de Gauss lorsque n augmente.

<pre> #-- AlgoHyp312_30_9bis()-- Distribution simulée de la variable # FONCTION rbinom() : plus besoin de boucle nbsim # Superposition de l'histogramme et de la densité gaussienne AlgoHyp312_30_9bis &lt;- fonction(n = 10, p = 1 / 5, nbsim = 2000) {   EX &lt;- n * p ; EtX &lt;- sqrt(n * p * (1 - p))   bornes &lt;- seq(-.5, (n + .5), 1)   seriesimul &lt;- rbinom(nbsim, n, p)   ValeursXGauss &lt;- seq(-.5, (n + .5), length = 1000)   DensiteGauss &lt;- dnorm(ValeursXGauss, EX, EtX)   # Graphiques superposant histogramme et densité gaussienne   hist(seriesimul, breaks = bornes, freq = FALSE,     main = ""Histogramme" de la série simulée, densité gauss.",     xlab = "Nombre de succès")   points(ValeursXGauss, DensiteGauss, type = "l",     col = "red")   legend(x = "topright",     legend = c("Densités simulées", "Densités gaussiennes"),     bg = "grey90", pch = c(95, 95), pt.cex = c(1, 1.3),     col = c("black", "red")) } </pre>	<pre>&gt; AlgoHyp312_30_9bis()</pre>
---	--------------------------------------

## 10° Programmation de la partie 2 du T.P. Nathan Hyperbole, Première S, page 312 n°30

- La partie 2 est intitulée "généralisation". La seule différence avec la partie 1 est que le succès est l'apparition d'une boule dont le numéro est compris entre 1 et m. Il est demandé que "la variable n soit lue en entrée" et que "les variables B et m soient initialisées au début de l'algorithme".
- J'ai pensé plus pratique que les valeurs de B, n et m soient toutes mises en paramètres pour pouvoir être modifiées lors de l'utilisation du programme (fonction) R.
- À titre d'exercice, construire l'algorithme et la fonction R correspondante, en utilisant toutes les spécificités du langage R :

<pre> #-- AlgoHyp312_30_10() simulation d'une valeur ... avec paramètres # Modèle d'urne avec boules numérotées, tirage avec remise AlgoHyp312_30_10 &lt;- fonction(n = 10, B = 5, m = 2){   urne &lt;- 1:B   x &lt;- sample(urne, n, replace = TRUE)   S &lt;- sum(x &lt;= m)   # Affichage du résultat   cat("Nombre de boules de n° compris entre 1 et", m, "=", S, "\n\n") } </pre>	<pre> &gt; AlgoHyp312_30_10() Nombre de boules de n° compris entre 1 et 2 = 4  &gt; AlgoHyp312_30_10() Nombre de boules de n° compris entre 1 et 2 = 1 </pre>
---	---

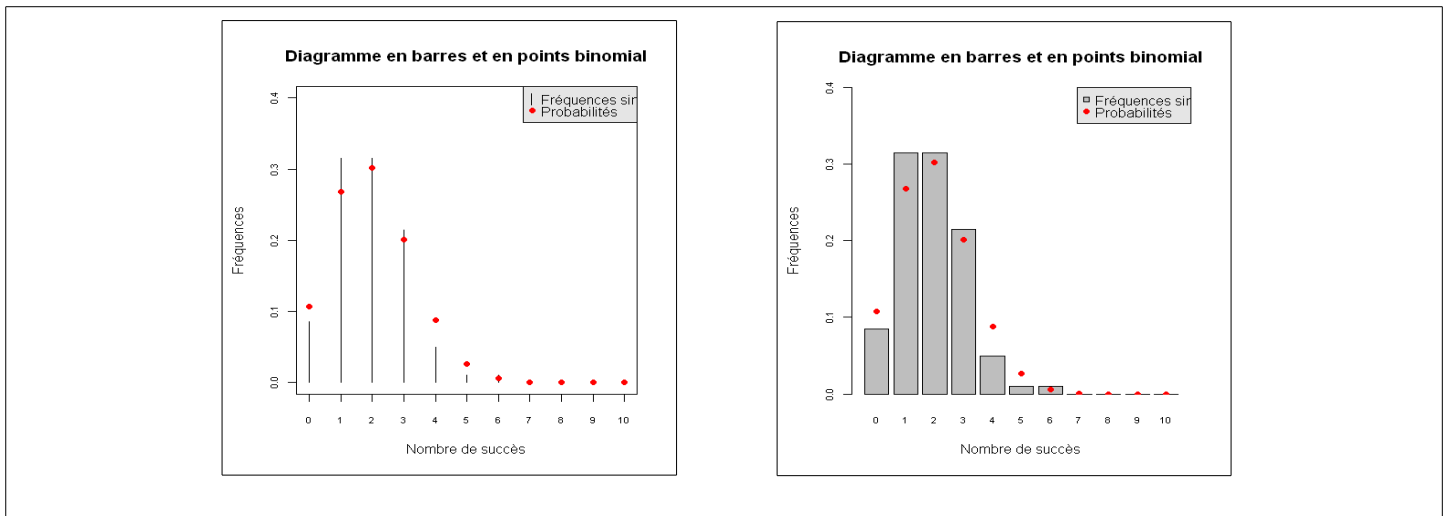
- **Suite de l'exercice** : utiliser, dans une boucle de 2000 simulations, cette fonction que vous adapterez légèrement, pour estimer la probabilité d'obtenir zéro boules de numéro entre 1 et 2. Comparer le résultat obtenu à la probabilité calculée avec la loi binomiale de paramètres n = 10 et p = 2 / 5 (utiliser `dbinom(k, n, p)`)

<pre> AlgoHyp312_30_10 &lt;- fonction(n = 10, B = 5, m = 2){   urne &lt;- 1:B   x &lt;- sample(urne, n, replace = TRUE)   S &lt;- sum(x &lt;= m)   return(S) }  probazero &lt;- fonction(){   z &lt;- 0   for(i in 1:2000) {z &lt;- z + sum(AlgoHyp312_30_10() == 0)}   cat("Une estimation de P(X = 0) :", z / 2000, "\n\n") } </pre>	<pre> &gt; probazero() Une estimation de P(X = 0) : 0.0065  &gt; probazero() Une estimation de P(X = 0) : 0.005  &gt; dbinom(0, 10, 2 / 5) [1] 0.006046618 </pre>
--	---

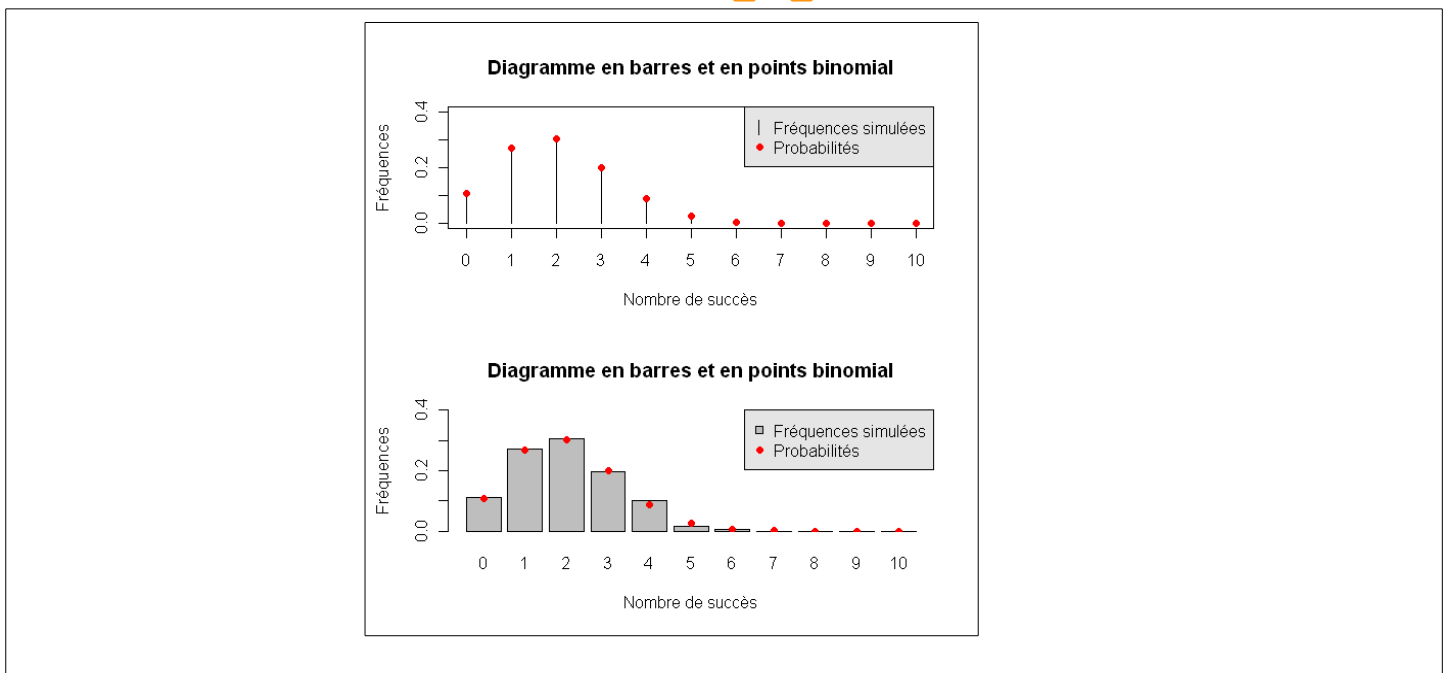
## 11° Quelques variations autour de la superposition de graphiques pour comparer fréquences simulées et probabilités

- Deux autres simulations en ligne de commande [AlgoHyp312\\_30\\_11](#), et [AlgoHyp312\\_30\\_12](#) proposent en plus du graphique plot-points, un graphique barplot-points, réalisés respectivement dans deux fenêtres graphiques ou dans une fenêtre graphique partagée en deux.

Ci-dessous le résultat graphique de [AlgoHyp312\\_30\\_11](#).



Ci-dessous le résultat graphique de [AlgoHyp312\\_30\\_12](#).



Ci-dessous les lignes de commande de [AlgoHyp312\\_30\\_11](#). Celles de [AlgoHyp312\\_30\\_12](#) figurent à la fin du fichier texte *EvolBinoAlgoHyp312\_30.r*.

```

***Lignes de commandes*****AlgoHyp312_30_11
#----Deux fenêtres graphiques pour plot-points et barplot-points
> nbsim <- 200 ; n <- 10 ; p <- .2 ; echelY <- .4
> valeurX <- 0:n
> bin1 <- rbinom(nbsim, n, p)
> (tablebin1 <- table(bin1, dnn = "tablebin1")) / nbsim
tablebin1
  0     1     2     3     4     5     6
0.085 0.315 0.315 0.215 0.050 0.010 0.010
> # attributes(tablebin1)
> (tableFreqX <- rep(0, n + 1))

```

```

[1] 0 0 0 0 0 0 0 0 0 0 0
> (names(tableFreqX) <- valeurX)
[1] 0 1 2 3 4 5 6 7 8 9 10
> tableFreqX[as.numeric(names(tablebin1)) + 1] <- tablebin1
> tableFreqX
  0    1    2    3    4    5    6
0.085 0.315 0.315 0.215 0.050 0.010 0.010
  7    8    9   10
0.000 0.000 0.000 0.000
> # attributes(tableFreqX)
> (binotheol <- dbinom(valeurX, n, p))
[1] 0.1073741824 0.2684354560 0.3019898880
[4] 0.2013265920 0.0880803840 0.0264241152
[7] 0.0055050240 0.0007864320 0.0000737280
[10] 0.0000040960 0.0000001024
> # On ferme tous les graphiques précédents
> graphics.off()
> # Graphiques superposant plot et points dans une nouvelle fenêtre
> plot(valeurX, tableFreqX, type = "h", xaxp = c(0, n, n), ylim = c(0, echelY),
+      ylab = "Fréquences", xlab = "Nombre de succès", cex.axis = .7,
+      main = "Diagramme en barres et en points binomial")
> points(valeurX, binotheol,
+        pch = 21, col = "red", bg = "red")
> legend(x = "topright", legend = c("Fréquences simulées", "Probabilités"),
+        bg = "grey90", pch = c(124, 21), pt.cex = c(1, 1),
+        col = c("black", "red"), pt.bg = c(NA, "red"))
> # On ouvre une nouvelle fenêtre graphique
> dev.new()
> # Graphiques superposant barplot et points dans cette nouvelle fenêtre
> barplot(tableFreqX, ylim = c(0, echelY), cex.axis = .7, cex.names = .7,
+         ylab = "Fréquences", xlab = "Nombre de succès",
+         main = "Diagramme en barres et en points binomial")
> points(barplot(1:(n + 1), plot = FALSE), binotheol,
+        pch = 21, col = "red", bg = "red")
> legend(x = "topright", legend = c("Fréquences simulées", "Probabilités"),
+        bg = "grey90", pch = c(22, 21), pt.cex = c(1, 1),

```

## 12° Un autre exemple de simulation d'une distribution : Nathan Hyperbole, *Première S* page 321 n°71

- Ce programme AlgoBox (cf. le fac-similé de l'énoncé en fin de document) produit une série simulée de valeurs à distribution binomiale.
- L'exercice porte mot pour mot le même intitulé "Simuler la loi binomiale avec un programme" que l'objectif de l'exercice page 312 n° 30, qui ne simule qu'une seule valeur. Cette imprécision n'est pas de nature à mettre en évidence l'importance de la notion de distribution (exacte ou simulée) d'une loi et la nature des outils à mettre en œuvre pour la simuler et décrire ensuite la série simulée.
- La fonction **R** suivante est la transcription du programme AlgoBox à laquelle j'ai ajouté un diagramme en barre.
- Cette transcription rend le programme plus lisible, ne serait-ce, entre autres, parce que l'on passe de 36 lignes AlgoBox à 17 lignes **R**.

```

#----- AlgoHyp321_71() algorithme de l'énoncé --- runif
AlgoHyp321_71 <- function(n = 10, p = .6, M = 10000){
  TAB <- NULL
  for(j in 1:(n + 1)){TAB[j] <- 0}
  for(k in 1:M){
    s <- 0
    for(i in 1:n){
      x <- runif(1, 0, 1)
      if(x < p) {s <- s + 1}
    }
    TAB[s + 1] <- TAB[s + 1] + 1
  }
  TABfreq <- TAB / M
# Affichages
  for(j in 1:(n + 1)){cat("\nTABfreq[" , j , " ] = fréquence(X =", j - 1, ") =", TABfreq[j], "\n")}
  barplot(TABfreq, names = 0:n, xlab = "valeurs de la variable",
          ylab = "fréquences", main = "Diagramme en barres")
}

```

► Ci-dessous j'ai fait évoluer la fonction **R** précédente afin d'utiliser quelques spécificités de **R** :

Toutes les valeurs de la variable n'étant pas forcément atteintes dans la simulation, le tableau des fréquences peut changer à chaque simulation.

```

#----- AlgoHyp321_71_1() algorithme n épreuves de Bernoulli
# identiques et indépendants, utilisation des opérateurs sur les vecteurs.
AlgoHyp321_71_1 <- function(n = 10, p = .6, M = 10000){
  deuxalternatives = c("succes", "echec")
  seriesimul <- NULL
  for(i in 1:M){
    x <- sample(deuxalternatives, n, prob = c(p, 1 - p), replace = TRUE)
    seriesimul[i] <- sum(x == "succes")
  }
  tabloFreq <- table(seriesimul) / M
# Affichages
  cat("\nDistribution simulée de la fréquence de succès :\n")
  print(tabloFreq)
  barplot(tabloFreq, ylab = "Fréquences simulées", xlab = "Nombre de succès",
          main = "Diagramme en barres")
}

```

► Ci-après sa variante pour avoir toutes les valeurs possibles de la variable dans le tableau :

```

#----- AlgoHyp321_71_2() algorithme n épreuves de Bernoulli
# Tableau des fréquences avec toutes les valeurs possibles de X
AlgoHyp321_71_2 <- function(n = 10, p = .6, M = 10000){
  deuxalternatives = c("succes", "echec")
  seriesimul <- NULL
  tabloFreq <- rep(0, n + 1) ; names(tabloFreq) <- 0:n
  for(i in 1:M){
    x <- sample(deuxalternatives, n, prob = c(p, 1 - p), replace = TRUE)
    seriesimul[i] <- sum(x == "succes")
  }
  tabloFreq[as.numeric(names(table(seriesimul))) + 1] <- table(seriesimul) / M
# Affichages
  cat("\nDistribution simulée de la fréquence de succès :\n")
  print(tabloFreq)
  barplot(tabloFreq, ylab = "Fréquences simulées", xlab = "Nombre de succès",
          main = "Diagramme en barres")
}

```

# 13° Exemple d'algorithme et de programme R de calcul d'une probabilité cumulée avec une loi binomiale : Nathan Hyperbole, *Première S*, page 338 n°41

## Algorithmique



### 41 Un algorithme

X est une variable aléatoire qui suit une loi binomiale de paramètres  $n$  et  $p$ .

La fonction `LOI.BINOMIALE(k;n;p;1)` du tableur renvoie la probabilité  $P(X \leq k)$ .

Écrire un algorithme qui calcule cette probabilité.

- L'énoncé demande d'écrire un algorithme pour calculer  $P(X \leq k)$ , X de loi binomiale de paramètres  $n$  et  $p$  et  $k$  entier donné entre 0 et  $n$ . Je vais présenter 5 stratégies possibles : l'utilisation classique d'une boucle `for()`, des utilisations de la série de la distribution de probabilité de X, et des utilisations des fonctions `R dbinom()` et `pbinom()`, dédiées aux lois binomiales.

```
*****Algorithme Hyperbole p338_41*****
# Algorithme pour calculer P(X = k), X de loi binomiale
# de paramètres n et p et k entier donné entre 0 et n.
#-----
#----AlgoHyp338_40_1() Utilisation de la boucle for()
AlgoHyp338_40_1 <- function(k = 4, n = 10, p = .3) {
  cumcrois <- 0
  for(i in 0:k) {
    cumcrois <- cumcrois + choose(n, i) * p^i * (1 - p)^(n - i)
  }
  cat("P(X <=", k, ")=", cumcrois, "\n\n")
}
> AlgoHyp338_40_1()
P(X <= 4 )= 0.8497317
#-----
#----AlgoHyp338_40_2() Utilisation de la liste de
# la distribution de X
AlgoHyp338_40_2 <- function(k = 4, n = 10, p = .3) {
  distribX <- NULL
  for(i in 1:(k + 1)){
    j = i - 1
    distribX[i] <- choose(n, j) * p^j * (1 - p)^(n - j)
  }
  cumcrois <- sum(distribX[1:(k + 1)])
  cat("P(X <=", k, ")=", cumcrois, "\n\n")
}
> AlgoHyp338_40_2()
P(X <= 4 )= 0.8497317
#-----
#----AlgoHyp338_40_3() Utilisation de la liste de
# la distribution de X et de la fonction R dbinom()
AlgoHyp338_40_3 <- function(k = 4, n = 10, p = .3) {
  distribX <- NULL
  for(i in 1:(k + 1)){
    j = i - 1
    distribX[i] <- dbinom(j, n, p)
  }
  cumcrois <- sum(distribX[1:(k + 1)])
  cat("P(X <=", k, ")=", cumcrois, "\n\n")
}
> AlgoHyp338_40_3()
P(X <= 4 )= 0.8497317

#--Lignes de commandes-----
#----AlgoHyp338_40_C Utilisation directe
# des fonctions R dbinom() (distribution) et
# pbinom() (répartition)
# Il n'y a plus besoin de programme
# par exemple avec n = 10, p = .3 et k = 4
# pour calculer P(X ≤ 4) on peut faire :
#-----
> sum(dbinom(x = 0:4, size = 10, prob = .3))
[1] 0.8497317
# ou plus simplement
> sum(dbinom(0:4, 10, .3))
[1] 0.8497317
#-----
> pbinom(4, 10, .3)
[1] 0.8497317
#
#-----
# On peut aussi faire le tableau de
# distribution de X :
> dbinom(0:10, 10, .3)
[1] 0.0282475249 0.1210608210 0.2334744405
[4] 0.2668279320 0.2001209490 0.1029193452
[7] 0.0367569090 0.0090016920 0.0014467005
[10] 0.0001377810 0.0000059049
#-----
# Et le tableau de la répartition de X
> cumsum(dbinom(0:10, 10, .3))
[1] 0.02824752 0.14930835 0.38278279
[4] 0.64961072 0.84973167 0.95265101
[7] 0.98940792 0.99840961 0.99985631
[10] 0.99999410 1.00000000
# ou plus simplement
> pbinom(0:10, 10, .3)
[1] 0.02824752 0.14930835 0.38278279
[4] 0.64961072 0.84973167 0.95265101
[7] 0.98940792 0.99840961 0.99985631
[10] 0.99999410 1.00000000
```



## B – SIMULER UNE LOI GÉOMÉTRIQUE TRONQUÉE.

### 1° Programme original tiré du manuel Nathan Hyperbole, *Première S*, page 290 n°23

#### 23 Simuler la loi géométrique tronquée

**OBJECTIF** Simuler la loi géométrique tronquée avec un algorithme.

Une roulette est divisée en 18 secteurs numérotés de 1 à 18.

Le croupier propose aux joueurs 20 parties successives. L'un des joueurs attend la sortie de son numéro fétiche : le 9.

#### 1. Simulation avec un programme

Voici un programme écrit avec le langage AlgoBox qui affiche le rang de la première apparition du numéro 9, ou bien qui affiche 0 si le 9 n'est pas sorti lors des 20 parties.

```
VARIABLES
- i EST_DU_TYPE NOMBRE
- x EST_DU_TYPE NOMBRE
DEBUT_ALGORITHME
- x PREND_LA_VALEUR floor(1+18*random())
- i PREND_LA_VALEUR 1
- TANT_QUE (x!=9 ET i<=20) FAIRE
  - DEBUT_TANT_QUE
  - x PREND_LA_VALEUR floor(1+18*random())
  - i PREND_LA_VALEUR i+1
  - FIN_TANT_QUE
- SI (i<=20) ALORS
  - DEBUT_SI
  - AFFICHER i
  - FIN_SI
- SINON
  - DEBUT_SINON
  - AFFICHER "0"
  - FIN_SINON
FIN_ALGORITHME
```

- Expliquer le fonctionnement de ce programme.
- Saisir le programme à l'ordinateur et tester son fonctionnement.

#### 2. Modélisation

On note  $X$  la variable aléatoire égale au rang de la première apparition du 9, ou égale à 0 si le 9 n'est pas apparu pendant les 20 parties.

- Déterminer la loi de probabilité de  $X$ .
- Calculer une valeur arrondie au dixième de l'espérance de  $X$ .

Que représente cette espérance ?

Tous les codes **R** de ce chapitre figurent dans le fichier texte *EvolGeoTronkAlgoHyp290\_23.r*. Il peut être ouvert avec l'éditeur spécifique à coloration syntaxique **RStudio**.

Ce programme utilise :

- Un générateur de nombres pseudo-aléatoires uniformes entre 0 et 1 (**runif()**) qui est la transposition de **random()**, avec la fonction **floor()**, pour simuler des entiers de distribution uniforme entre 1 et 18. Il est intéressant de noter que **runif(1, 0, 1)** nécessite de préciser le nombre de tirages effectués (**1**) et les bornes de l'intervalle de la loi uniforme utilisée (**0, 1**) ;
- Une boucle TantQue(**while()**) ;
- Avec un test (**if()**) utilisant le connecteur logique ET(**&**) ;
- Un compteur manuel de rang, **i** ;
- Un test pour le cas particulier du "rang" 0.

Ces difficultés rendent la construction du programme peu abordable avec des élèves de *Première*.

De plus, la syntaxe Algobox rend le programme peu lisible : les déclarations de variables sont inutiles dans les applications habituelles, les indentations des boucles et des tests sont inutilement chargées. Sa traduction ou plutôt transposition en **R** en simplifie déjà significativement la lecture, comme on le voit ci-dessous :



```

# SIMULATIONS de lois géométriques tronquées
# Une roulette est divisée en 18 secteurs
# identiques numéroté 1 à 18 On fait tourner
# la roue n fois (n>=1) et on note X la
# variable aléatoire prenant pour valeurs le
# rang de la première apparition du numéro 9
# ou bien 0 si le 9 n'est pas sorti. Les
# algorithmes suivants simulent cette
# expérience aléatoire. et généralisent des
# modèles de simulation.
# AlgoHyp290_23()-simuler un rang avec runif
# Modèle roulette 18 secteurs
AlgoHyp290_23 <- function(){
  x <- floor(1 + 18 * (runif(1, 0, 1)))
  i <- 1
  while(x != 9 & i <= 20){
    x <- floor(1 + 18 * (runif(1, 0, 1)))
    i <- i + 1
  }
# Affichages
if(i <= 20){
  cat("\nRang du premier 9 =", i, "\n")
} else {
  cat("\nRang du premier 9 =", 0, "\n")
}
}

```

## Programme Texas sur schéma de Bernoulli

```

:Prompt N, P, R
:EffListe L6
:For (I,1,R)
:1→K
:NbrAléa→A
:While K≤N et A≥P
:NbrAléa→A
:K+1→K
:End
:If K>N
:Then
:0→K
:End
:K→L6(I)
:End
:Stats 1-Var L6
:EffEcr
:Disp n,  $\bar{x}$ , Med, Sx
:Pause
:GraphNAff
:0→Xmin:N→Xmax:1→Xgrad
:0→Ymin:R/5→Ymax:(Ymax-Ymin)/10→Ygrad
:Graph1 (Histogramme,L6)
:AffGraph
:Pause
:GraphNAff
:Graph2 (GraphBoitMoust,L6)
:AffGraph

```

## 2° Première évolution : utilisation de sample au lieu de runif() (randon())

- `sample(1:18, 1)` tire un échantillon aléatoire et simple de taille 1 dans la liste des entiers de 1 à 18.
- `1:18` génère la liste des nombres entiers de 1 à 18.
- le reste du programme est identique au précédent, avec les mêmes difficultés.

L'instruction `sample(1:18, 1)` est mieux comprise par les élèves que `floor(1+18*(runif(1,0,1)))`.

De plus elle est directement liée à la notion d'échantillon aléatoire et simple, vue en cours.

```

#--AlgoHyp290_23_1()--simulation un rang--sample 1
# Modèle roulette 18 secteurs
AlgoHyp290_23_1 <- function(){
  x <- sample(1:18, 1)
  i <- 1
  while(x != 9 & i <= 20){
    x <- sample(1:18, 1)
    i <- i + 1
  }
# Affichages
if(i <= 20){
  cat("\nRang du premier 9 =", i, "\n")
} else {
  cat("\nRang du premier 9 =", 0, "\n")
}
}

```

```

AlgoHyp290_23_1()
Rang du premier 9 = 8

AlgoHyp290_23_1()
Rang du premier 9 = 6

```

## 3° Deuxième évolution : utilisation de l'objet R roulette

- La seule différence avec le programme précédent est l'introduction de l'objet roulette qui est la liste des entiers de 1 à 18, simulant la roulette à 18 secteurs (`roulette <- 1:18`). On simplifie ainsi la simulation des parties de roulette, tout en les rapprochant de l'expérience réelle.

```

#-----AlgoHyp290_23_2()----simulation un rang--roulette-sample 1
# Modèle roulette 18 secteurs
AlgoHyp290_23_2 <- fonction(){
  roulette <- 1:18
  x <- sample(roulette, 1)
  i <- 1
  while(x != 9 & i <= 20){
    x <- sample(roulette, 1)
    i <- i + 1
  }
# Affichages
if(i <= 20){
  cat("\nRang du premier 9 =", i, "\n")
} else {
  cat("\nRang du premier 9 =", 0, "\n")
}
}

```

#### 4° Troisième évolution : la boucle while est remplacée par l'échantillonnage sample et l'utilisation de which qui renvoie les rangs des composantes d'une liste

- L'instruction `sample(roulette, 20, replace = TRUE)` tire un échantillon de taille 20 avec remise, simulant ainsi les 20 parties. Les résultats sont mis dans la liste `x`. On évite ainsi le recours à une boucle plus difficile à gérer et plus consommatrice de temps.
- Il s'agit maintenant de repérer le rang du premier 9 de la liste `x`. L'instruction `which(x == 9)` renvoie les rangs de toutes les valeurs 9 de la liste. Le rang du premier 9 sera le minimum (`min`) de ces rangs.
- Afin de prendre en compte le cas où 9 ne fait pas partie de la liste, l'instruction `if(sum(x - 9 == 0) > 0)`, où `sum(x - 9 == 0)` compte le nombre de valeurs de `x` égales à 9, teste s'il y a au moins un 9 dans la liste `x`. Sinon c'est la valeur 0 qui est attribuée au rang.

Il n'est plus nécessaire d'utiliser le connecteur logique ET (&) dans une boucle while, ce qui simplifie la stratégie de l'algorithme. Dans ce cas, le test a uniquement pour fonction de gérer le cas du rang 0.

On peut facilement mettre en évidence le fonctionnement du programme en faisant afficher la liste `x` et le(s) rang(s) des 9 s'il y en a.

Le programme s'en trouve fortement simplifié : on passe de 14 lignes à 6 lignes !

```

#-----AlgoHyp290_23_3()----simulation un rang--roulette-sample 20
# Modèle roulette 18 secteurs
AlgoHyp290_23_3 <- fonction(){
  roulette <- 1:18
  x <- sample(roulette, 20, replace = TRUE)
  if(sum(x - 9 == 0) > 0){
    k <- min(which(x == 9))
  } else {
    k <- 0
  }
# Affichage
cat("\nRang du premier 9 =", k, "\n")
}

```

La version en “lignes de commandes” permet de comprendre ce qui se passe :

```
#-----AlgoHyp290_23_3-----simulation un rang--roulette-sample 20
# Modèle roulette 18 secteurs
> (roulette <- 1:18)
[1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
> (x <- sample(roulette, 20, replace = TRUE))
[1] 8 15 13 18 15 11 18 4 13 6 11 15 14 12 6 13 10 17 6 1
> sum(x - 9 == 0)
[1] 0
> (x <- sample(roulette, 20, replace = TRUE))
[1] 10 16 16 5 3 5 7 18 1 13 11 3 1 14 7 9 15 10 9 18
> sum(x - 9 == 0)
[1] 2
> which(x == 9)
[1] 16 19
> (k <- min(which(x == 9)))
[1] 16
```

## 5° Quatrième évolution : introduction des paramètres

- Avec les programmes précédents, si l'on veut changer de valeur du nombre de parties, du nombre de secteurs de la roulette, du nombre à jouer (dont on attend la sortie), il faut modifier ces valeurs en éditant (modifiant) le programme lui même, ce qui est long et souvent source d'erreurs.
- Avec **R**, il suffit d'introduire ces valeurs en tant que paramètres en tête de la fonction programmée : **function(nbperties = 20, nbsect = 18, numero = 9)** . Ce sont ces valeurs qui seront utilisées par défaut si aucune valeur n'est indiquée lors de l'exécution de la fonction. Sinon, il suffit de saisir les valeurs voulues, lors de l'exécution de la fonction : **AlgoHyp290\_23\_4(nbperties = 15, nbsect = 5)** . On peut omettre les noms des paramètres si les valeurs sont saisies suivant l'ordre programmé dans la fonction.

<pre>#-----AlgoHyp290_23_4()-----simulation un rang--roulette--paramètres # Modèle roulette nbsecteurs AlgoHyp290_23_4 &lt;- function(nbperties = 20, nbsect = 18, numero = 9){   roulette &lt;- 1:nbsect   x &lt;- sample(roulette, nbparties, replace = TRUE)   if(sum(x - numero == 0) &gt; 0){     k &lt;- min(which(x == numero))   } else {     k &lt;- 0   } # Affichage   cat("\nRang du premier", numero,"=", k, "\n") }</pre>	<pre>AlgoHyp290_23_4(numero = 2) Rang du premier 2 = 0</pre>
---	--

## 6° Cinquième évolution : on change de programme pour simuler une distribution

- On change d'objectif puisqu'il s'agit de simuler, non plus **une seule valeur** du rang du premier 9 obtenu, mais la distribution de la variable **X**, rang du premier 9 obtenu lors de 20 parties.
- On va donc introduire une boucle de **nbsim = 2000** simulations. Le résultat de chacune de ces simulations d'une valeur de **X** est stocké dans la liste (vecteur) **seriesim** dont la composante **i** est **seriesim[i]** . À la fin des simulations **seriesim** contient 2000 valeurs de la variable **X**.
- Pour décrire cette série statistique, l'instruction **distsim <- table(seriesim)** en fait le tableau des effectifs et l'instruction **barplot(distsim / nbsim)** produit le diagramme en barres qui illustre le tableau des fréquences correspondant.
- On illustre ainsi une distribution simulée de la variable **X**, sans avoir besoin de connaître la distribution exacte (le modèle mathématique) de cette variable.
- On peut alors faire varier les différents paramètres introduits dans la fonction, pour en illustrer l'influence : si l'on change de **numéro**, la distribution va-t-elle changer “significativement” ?

```

#--AlgoHyp290_23_5()--simulation distribution des rangs
# Modèle roulette nbsecteurs
AlgoHyp290_23_5 <- fonction(nbparties = 20,
                           nbsect = 18, numero = 9, nbsim = 2000){
  roulette <- 1:nbsect
  seriesim <- NULL
  for(i in 1:nbsim){
    x <- sample(roulette, nbparties, replace = TRUE)
    if(sum(x - numero == 0) > 0){
      seriesim[i] <- min(which(x == numero))
    } else {
      seriesim[i] <- 0
    }
  }
  distsim <- table(seriesim)
# Affichages
cat("\nTableau de distribution du Rang du premier",
    numero, "\n")
print(distsim / nbsim)
barplot(distsim / nbsim)
}

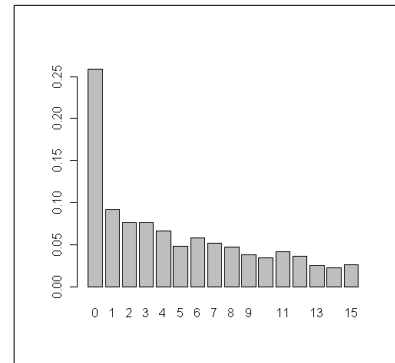
```

```
AlgoHyp290_23_5(15, 12, 6)
```

```

Tableau de distribution du Rang du premier 6
seriesim
  0      1      2      3      4      5      6      7      8
0.2585 0.0915 0.0760 0.0765 0.0660 0.0485 0.0585 0.0520 0.0475
  9      10     11     12     13     14     15
0.0385 0.0345 0.0415 0.0365 0.0255 0.0225 0.0260

```



## 7° Sixième évolution : on remplace la roulette par une urne échantillonnée avec remise

Il y a deux changements notables :

- La roulette est remplacée par une urne contenant des boules, de composition fixée :  
**urne <- rep(c("brouge", "bautre"), c(nbrouges, nbtot - nbrouges))** et dans laquelle on effectue des tirages avec remise :  
**sample(urne, nbtirages, replace = TRUE)**,
- L'urne contient des boules rouges **brouge** et d'autres couleurs **bautre**. Les résultats des tirages sont donc **brouge** ou **bautre** et non plus des numéros. Il faut donc effectuer des tests sur des chaînes de caractères, pour repérer les rangs d'apparition des boules rouges, ce qui ne pose aucun problème de programmation : **which(x == "brouge")**.
- La simulation "mime", au plus près, une expérience réelle.

```

#--AlgoHyp290_23_6()--simulation distribution des rangs--
# Modèle d'urnes nbboules rouges, nbttotal, tirages avec remise
AlgoHyp290_23_6 <- fonction(nbtirages = 20, nbrouges = 1, nbtot = 18,
                           nbsim = 2000){
  urne <- rep(c("brouge", "bautre"), c(nbrouges, nbtot - nbrouges))
  seriesim <- NULL
  for(i in 1:nbsim){
    x <- sample(urne, nbtirages, replace = TRUE)
    if(sum(x == "brouge") > 0){
      seriesim[i] <- min(which(x == "brouge"))
    } else {
      seriesim[i] <- 0
    }
  }
  distsim <- table(seriesim)
# Affichages
cat("\nTableau de distribution du rang de la première rouge\n")
print(distsim / nbsim)
barplot(distsim / nbsim)
}

```

## 8° Septième évolution : on simule des épreuves de Bernoulli identiques et indépendantes

- Les deux alternatives sont "**succes**" et "**echec**",
- Le tirage avec remise est effectué en fonction des probabilités respectives **p** et **1 - p** par l'instruction :  
**sample(deuxalternatives, n, prob = c(p, 1 - p), replace = TRUE)**.

- On compte le nombre de succès avec : `sum(x == "succes")`

```
#-----AlgoHyp290_23_7()----simulation distribution des rangs--
# Modèle épreuves de Bernoulli répétées (indépendantes) --n, p
AlgoHyp290_23_7 <- fonction(n = 20, p = 1 / 18, nbsim = 2000){
  deuxalternatives <- c("succes", "echec")
  seriesim <- NULL
  for(i in 1:nbsim){
    x <- sample(deuxalternatives, n, prob = c(p, 1 - p), replace = TRUE)
    if(sum(x == "succes") > 0){
      seriesim[i] <- min(which(x == "succes"))
    } else {
      seriesim[i] <- 0
    }
  }
  distsim <- table(seriesim)
# Affichages
cat("\nTableau de distribution du Rang du premier succès\n")
print(distsim / nbsim)
barplot(distsim / nbsim)
}
```

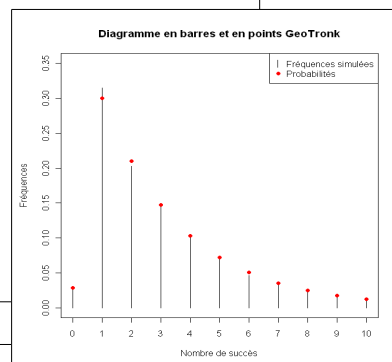
## 9° Huitième évolution : tableaux complets des fréquences et des probabilités et superposition des représentations graphiques

- Un tableau “complet” des fréquences simulées présente toutes les valeurs de la variable, ce qui est nécessaire lorsque l'on veut faire comparaisons et graphiques, par exemple des fréquences simulées et des probabilités.

```
#-----AlgoHyp290_23_8()----simulation distribution des rangs--
# Modèle épreuves de Bernoulli répétées (indépendantes) --n, p
# Graphiques des distributions simulées et calculées
AlgoHyp290_23_8 <- fonction(n = 20, p = 1 / 18, nbsim = 2000, echelY = .8){
  ValeursX <- 0:n
  DistGeoTronk <- NULL
  DistGeoTronk[1] <- (1 - p)^n
  DistGeoTronk[2:(n + 1)] <- (1 - p)^((1:n) - 1) * p
  names(DistGeoTronk) <- ValeursX
  deuxalternatives <- c("succes", "echec")
  seriesim <- NULL
  tableauFreq <- rep(0, n + 1)
  names(tableauFreq) <- ValeursX
  for(i in 1:nbsim){
    x <- sample(deuxalternatives, n, prob = c(p, 1 - p), replace = TRUE)
    if(sum(x == "succes") > 0){seriesim[i] <- min(which(x == "succes"))}
    } else {seriesim[i] <- 0}
  }
  distsim <- table(seriesim)
  tableauFreq[as.numeric(names(distsim)) + 1] <- distsim / nbsim
# Affichage des résultats et des graphiques
cat("Tableau des fréquences simulées du Rang du premier succès\n")
print(tableauFreq)
cat("\nTableau de probabilité du Rang du premier succès\n")
print(DistGeoTronk)
plot(ValeursX, tableauFreq, type = "h", xaxp = c(0, n, n), ylim = c(0, echelY),
      ylab = "Fréquences", xlab = "Nombre de succès",
      main = "Diagramme en barres et en points GeoTronk")
points(ValeursX, DistGeoTronk, pch = 21, col = "red", bg = "red")
legend(x = "topright", legend = c("Fréquences simulées", "Probabilités"),
       pch = c(124, 21), pt.cex = c(1, 1),
       col = c("black", "red"), pt.bg = c(NA, "red"))
}
> AlgoHyp290_23_8(n = 10, p = .3, echelY = .35)
Tableau des fréquences simulées du Rang du premier succès
  0      1      2      3      4      5      6      7      8      9     10
0.0255 0.3150 0.2025 0.1460 0.1010 0.0745 0.0465 0.0360 0.0230 0.0190 0.0110

Tableau de probabilité du Rang du premier succès
  0      1      2      3      4      5      6
0.02824752 0.30000000 0.21000000 0.14700000 0.10290000 0.07203000 0.05042100
  7      8      9     10
0.03529470 0.02470629 0.01729440 0.01210608
```

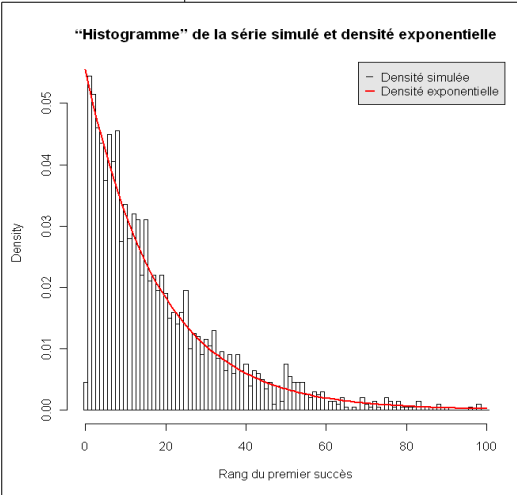
Voir ci-dessous le détail de l'obtention d'un tableau comprenant toutes les valeurs de X.



- C'est ce que fait la ligne de commande :  
`tableauFreq[as.numeric(names(distsim)) + 1] <- distsim / nbsim`
- Voyons comment elle fonctionne sur un exemple en "lignes de commandes".

<pre>&gt; n = 6 ; p = .4 ; nbsim = 20 &gt; (tableauFreq &lt;- rep(0, n + 1)) [1] 0 0 0 0 0 0 &gt; (ValeursX &lt;- 0:n) [1] 0 1 2 3 4 5 6 &gt; (names(tableauFreq) &lt;- ValeursX) &gt; tableauFreq 0 1 2 3 4 5 6 0 0 0 0 0 0 0 ... &gt; (distsim &lt;- table(seriesim)) seriesim 0 1 2 3 5 1 12 3 3 1 &gt; distsim / nbsim seriesim 0 1 2 3 5 0.05 0.60 0.15 0.15 0.05</pre>	<p>Que fait :</p> <pre>tableauFreq[as.numeric(names(distsim)) + 1] &lt;- distsim / nbsim</pre> <p><code>names(distsim)</code> repère les intitulés des composantes de la liste <code>distsim</code>, qui sont "0" "1" "2" "3" "5". Il manque "4" et "6".  <code>as.numeric(names(distsim))</code> les transforme en nombres entiers, car les intitulés sont des caractères.  <code>[as.numeric(names(distsim)) + 1]</code> les prend comme indices de la liste <code>tableauFreq</code> avec + 1 car il y a un décalage de 1 entre les valeurs de la variable X qui commencent à 0 et les indices de la liste qui commencent à 1.  <code>tableauFreq[as.numeric(names(distsim)) + 1]</code> indique donc les indices des composantes de la liste <code>tableauFreq</code> auxquelles vont être affectées les valeurs de la liste <code>distsim / nbsim</code>.  On obtient donc au final :</p> <pre>&gt; tableauFreq[as.numeric(names(distsim)) + 1] &lt;- distsim/nbsim &gt; tableauFreq 0 1 2 3 4 5 6 0.05 0.60 0.15 0.15 0.00 0.05 0.00</pre>
--	--

- Lorsque n augmente la loi géométrique tronquée converge vers la loi géométrique qui converge vers la loi exponentielle. C'est ce qu'illustre la fonction suivante, dans laquelle on prend 1/p comme valeur approchée du paramètre de la loi exponentielle. L'espérance de la loi géométrique tronquée est  $[1 - (1 + n*p)(1 - p)^n] / p$ .

<pre>#----AlgoHyp290_23_8bis()----simulation de la distribution des rangs-- # Modèle épreuves de Bernoulli répétées (indépendantes) --n, p # Graphiques des distributions simulées et Exponentielle AlgoHyp290_23_8bis &lt;- fonction(n = 100, p = 1 / 18, nbsim = 2000) {   Esperance &lt;- 1 / p   ValeursX &lt;- 0:n ; bornes &lt;- seq(-.5, (n + .5), 1)   DistExpo &lt;- exp(- ValeursX / Esperance) / Esperance   deuxalternatives &lt;- c("succes", "echec")   seriesim &lt;- NULL   for(i in 1:nbsim){     x &lt;- sample(deuxalternatives, n, prob = c(p, 1 - p),                replace = TRUE)     if(sum(x == "succes") &gt; 0){       seriesim[i] &lt;- min(which(x == "succes"))     } else {       seriesim[i] &lt;- 0     }   }    moyenne &lt;- mean(seriesim)   # Affichage des résultats et des graphiques   cat("Moyenne de la série simulée =", moyenne,       "\nEspérance =", Esperance, "\n\n")   hist(seriesim, breaks = bornes, freq = FALSE,        xlab = "Rang du premier succès",        main = "\"Histogramme\" de la série simulé et densité exponentielle")   points(ValeursX, DistExpo, type = "l", col = "red", lwd = 2)   legend(x = "topright",         legend = c("Densité simulée", "Densité exponentielle"),         bg = "grey90", pch = c(95, 95), pt.cex = c(1, 1.3),         col = c("black", "red")) } &gt; AlgoHyp290_23_8bis() Moyenne de la série simulée = 18.0785 Espérance de la loi exponentielle de paramètre 1/18 = 18</pre>	
--	--



# 10° Neuvième évolution : utilisation d'une fonction externe `geotronk2()` pour calculer les probabilités

- La seule différence avec [AlgoHyp290\\_23\\_8](#) est l'utilisation d'une fonction `geotronk2` externe pour calculer les probabilités. Cette fonction peut être utilisée seule comme les autres fonctions **R**.

```
#-----AlgoHyp290_23_9()---simulation distribution des rangs--
# Modèle épreuves de Bernoulli répétées (independantes) --n, p
# Graphiques des distributions simulées et calculées
# UTILISATION D'UNE FONCTION EXTERNE geotronk2()
geotronk2 = function(k, n, p){
  distribX <- rep(0, n + 1)
  names(distribX) <- 0:n
  distribX[1] <- (1 - p)^n
  distribX[2:(n + 1)] <- (1 - p)^((1:n) - 1) * p
  proba <- distribX[k + 1]
  return(proba)
}
AlgoHyp290_23_9 <- function(n = 20, p = 1 / 18, nbsim = 2000, echelY = .35){
  ValeursX <- 0:n
  DistGeoTronk <- geotronk2(ValeursX, n, p)
  deuxalternatives <- c("succes", "echec")
  seriesim <- NULL
  tableauFreq <- rep(0, n + 1)
  names(tableauFreq) <- ValeursX
  for(i in 1:nbsim){
    x <- sample(deuxalternatives, n, prob = c(p, 1 - p), replace = TRUE)
    if(sum(x == "succes") > 0){
      seriesim[i] <- min(which(x == "succes"))
    } else {
      seriesim[i] <- 0
    }
  }
  distsim <- table(seriesim)
  tableauFreq[as.numeric(names(distsim)) + 1] <- distsim / nbsim
# Affichage des résultats et des graphiques
cat("Tableau des fréquences simulées du Rang du premier succès\n")
print(tableauFreq)
cat("\nTableau de probabilité du Rang du premier succès\n")
print(DistGeoTronk)
plot(ValeursX, tableauFreq, type = "h", xaxp = c(0, n, n), ylim = c(0, echelY),
  ylab = "Fréquences", xlab = "Nombre de succès",
  main = "Diagramme en barres et en points GeoTronk")
points(ValeursX, DistGeoTronk, pch = 21, col = "red", bg = "red")
legend(x = "topright", legend = c("Fréquences simulées", "Probabilités"),
  bg = "grey90", pch = c(124, 21), pt.cex = c(1, 1),
  col = c("black", "red"), pt.bg = c(NA, "red"))
}
```

```
Exemple d'utilisation de la
fonction geotronk2 :
> geotronk2(0, 10, .3)
0
0.02824752

> geotronk2(2:5, 10, .3)
2 3 4 5
0.21000 0.14700 0.10290 0.07203

> sum(geotronk2(2:5, 10, .3))
[1] 0.53193

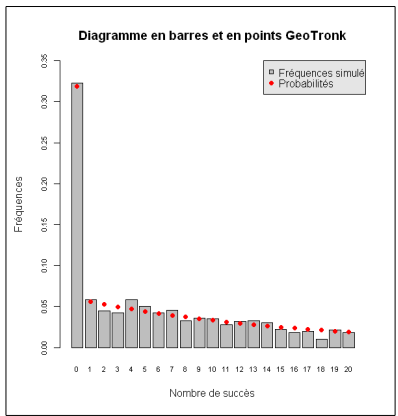
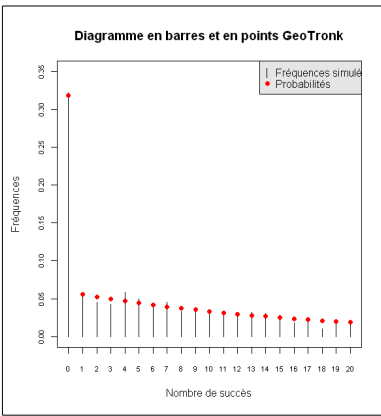
> sum(geotronk2(0:10, 10, .3))
[1] 1

> cumsum(geotronk2(0:2, 10, .3))
0 1 2
0.02824752 0.32824752 0.53824752
```

Cette fonction peut être utilisée aussi bien seule, qu'à l'intérieur d'une autre fonction, les variables des fonction étant locales par défaut puisque R est un langage "fonctionnel" (pas de souci pour les noms de variable).

- La version [AlgoHyp290\\_23\\_9](#) (sensiblement différente de la version ci-dessus) du fichier de code **R** `EvolGeoTronkAlgoHyp290_23.r` et la version [AlgoHyp290\\_23\\_10](#) proposent en plus du graphique plot-points, un graphique barplot-points, réalisés respectivement soit dans deux fenêtres graphiques soit dans une fenêtre graphique partagée en 2 :

```
> AlgoHyp290_23_9()
Fréquences simulées du Rang du premier succès
0 1 2 3 4 5
0.3225 0.0585 0.0450 0.0425 0.0580 0.0500
6 7 8 9 10 11
0.0420 0.0455 0.0330 0.0360 0.0350 0.0275
12 13 14 15 16 17
0.0320 0.0325 0.0305 0.0220 0.0180 0.0200
18 19 20
0.0105 0.0210 0.0180
Probabilité du Rang du premier succès
0 1 2 3
0.31880735 0.05555556 0.05246914 0.04955418
4 5 6 7
0.04680117 0.04420111 0.04174549 0.03942630
8 9 10 11
0.03723595 0.03516728 0.03321355 0.03136835
12 13 14 15
0.02962566 0.02797979 0.02642536 0.02495728
16 17 18 19
0.02357077 0.02226128 0.02102454 0.01985651
20
0.01875337
```



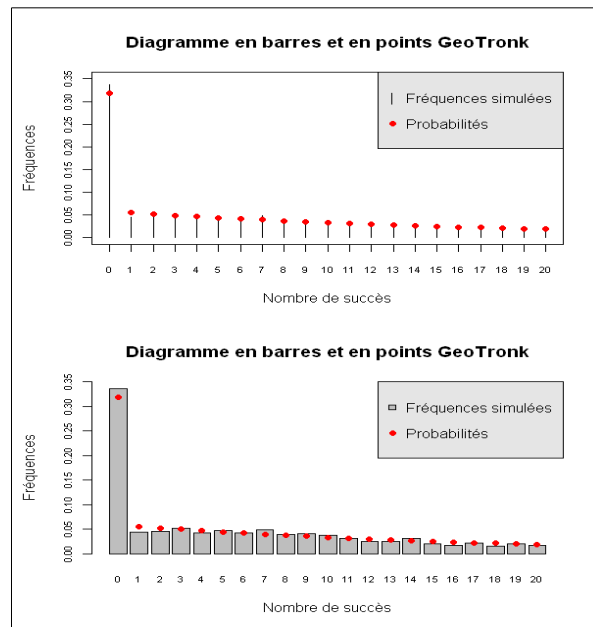
```
> AlgoHyp290_23_10()
```

Tableau des fréquences simulées du Rang du premier succès

0	1	2	3	4	5
0.3365	0.0445	0.0455	0.0525	0.0425	0.0475
6	7	8	9	10	11
0.0420	0.0490	0.0385	0.0410	0.0370	0.0315
12	13	14	15	16	17
0.0245	0.0255	0.0315	0.0200	0.0170	0.0210
18	19	20			
0.0160	0.0200	0.0165			

Tableau de probabilité du Rang du premier succès

0	1	2	3
0.31880735	0.05555556	0.05246914	0.04955418
4	5	6	7
0.04680117	0.04420111	0.04174549	0.03942630
8	9	10	11
0.03723595	0.03516728	0.03321355	0.03136835
12	13	14	15
0.02962566	0.02797979	0.02642536	0.02495728
16	17	18	19
0.02357077	0.02226128	0.02102454	0.01985651
20			
0.01875337			



## 11° Calcul numérique de l'espérance et de la variance d'une distribution géométrique tronquée.

- Il s'agit d'utiliser la fonction **R** `geotronk2()` que l'on a créée, pour calculer un valeur numérique de l'espérance et de la variance d'un distribution géométrique tronquée. Ce calcul est proposé dans le 2° de l'énoncé AlgoHyp296\_50. L'utilisation d'un logiciel de calcul numérique est suffisante pour faire les calculs demandés.

Je proposerai de faire les mêmes calculs pour la distribution obtenue dans AlgoHyp290\_23.

L'utilisation de la fonction `geotronk2()` se fait en ligne de commande :

```
# UTILISATION D'UNE FONCTION EXTERNE geotronk2()
# Calcul de la somme des probabilités de
# l'espérance et de la variance
geotronk2 = fonction(k, n, p){
  distribX <- rep(0, n + 1)
  names(distribX) <- 0:n
  distribX[1] <- (1 - p)^n
  distribX[2:(n + 1)] <- (1 - p)^((1:n) - 1) * p
  proba <- distribX[k + 1]
  return(proba)
}
> # Application numérique de l'énoncé AlgoHyp290_23
> # Somme numérique des probabilités
> sum(geotronk2(0:20, 20, 1 / 18))
[1] 1
> # Espérance numérique
> (somxi1pi <- sum(0:20 * geotronk2(0:20, 20, 1 / 18)))
[1] 5.885321
> # Variance numérique
> (somxi2pi <- sum((0:20)^2 * geotronk2(0:20, 20, 1 / 18)))
[1] 72.08714
> (vargeotronk <- somxi2pi - somxi1pi^2)
[1] 37.45014
-----
> # Application numérique de l'énoncé AlgoHyp296_50
> # Espérance numérique
> (somxi1pi <- sum(0:50 * geotronk2(0:50, 50, 1 / 5)))
[1] 4.999215
> # Variance numérique
> (somxi2pi <- sum((0:50)^2 * geotronk2(0:50, 50, 1 / 5)))
[1] 44.95654
```

```
> (vargeotronk <- somxi2pi - somxipi^2)
[1] 19.96439
-----
> # Espérance numérique
> (somxipi <- sum(0:100 * geotronk2(0:100, 100, 1 / 5)))
[1] 5
> # Variance numérique
> (somxi2pi <- sum((0:100)^2 * geotronk2(0:100, 100, 1 / 5)))
[1] 45
> (vargeotronk <- somxi2pi - somxipi^2)
[1] 20
```

Algorithmique



50



Travailler en groupe

Simuler à l'aide d'un algorithme

La roue d'une loterie est divisée en cinq secteurs identiques numérotés de 1 à 5.

On fait tourner la roue  $n$  fois ( $n$  entier,  $n \geq 1$ ) et on note  $X$  la variable aléatoire égale au rang de la première apparition du numéro 5, ou bien égale à 0 si le 5 n'est pas sorti.

1. On simule cette expérience aléatoire avec l'algorithme suivant :

**Entrée**

Saisir  $n$

**Initialisation**

$x$  prend la valeur d'un entier aléatoire de 1 à 5

$j$  prend la valeur 1

Tant que ... et ...

|  $x$  prend la valeur d'un entier aléatoire de 1 à 5

|  $j$  prend la valeur  $j + 1$

FinTantQue

**Sortie**

Si  $j \leq n$  alors

| Afficher ...

| sinon

| Afficher ...

FinSi

- a) Compléter cet algorithme.
- b) Écrire le programme associé dans un langage de programmation. Exécuter ce programme et vérifier son fonctionnement.

2. a) Déterminer la loi de probabilité de la variable aléatoire  $X$ .

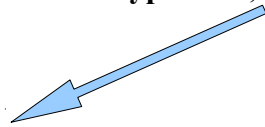
b) Vérifier que  $\sum_{j=0}^n P(X = j) = 1$ .

c) Établir que l'espérance de  $X$  est donnée par :

$$E(X) = \sum_{j=1}^n j \times 0,8^{j-1} \times 0,2$$

d) À l'aide d'un logiciel de calcul formel, calculer  $E(X)$  pour des valeurs assez grandes de  $n$ . Que remarque-t-on ?

$$\begin{aligned} & \sum_{j=1}^n (j \cdot (.8)^{j-1} \cdot .2) | n = 50 && 4.99921501377 \\ & \sum_{j=1}^n (j \cdot (.8)^{j-1} \cdot .2) | n = 100 && 4.99999997861 \end{aligned}$$



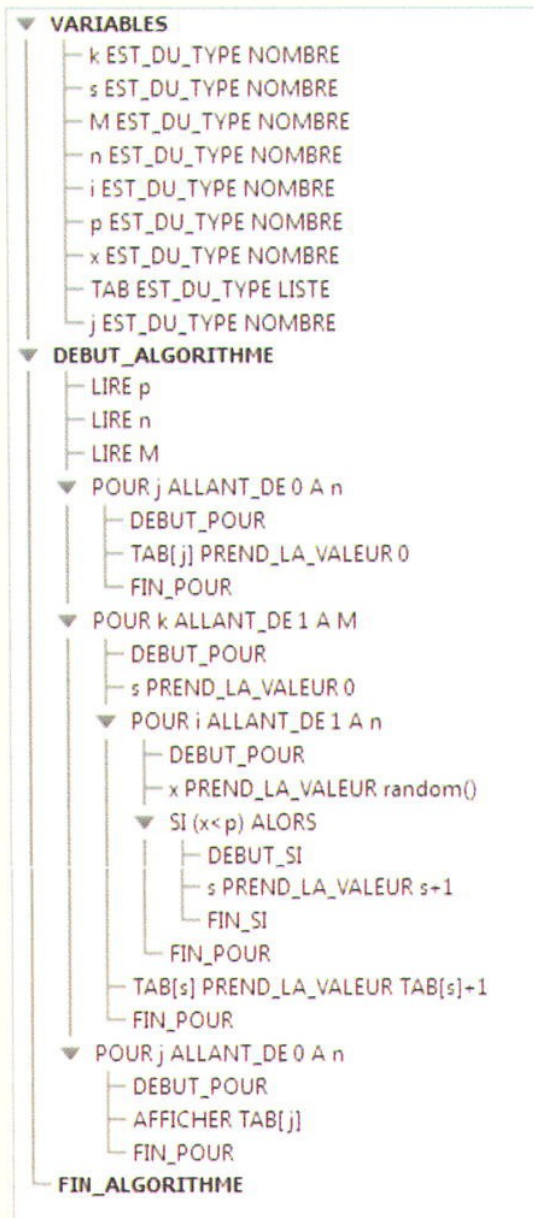
## 71 Simuler la loi binomiale avec un programme

Le programme suivant propose une simulation de la loi binomiale.

Le schéma de Bernoulli est la répétition de  $n$  épreuves de Bernoulli identiques et indépendantes de paramètre  $p$ .

Ce schéma est reproduit  $M$  fois dans le programme.

Pour  $k$  entier de 0 à  $n$ ,  $TAB[k]$  représente le nombre de fois, parmi les  $M$  reproductions, où il y a eu  $k$  succès.



a) Saisir ce programme à l'ordinateur.

b) On prend par exemple  $n = 10$  et  $p = 0,6$ .

Exécuter ce programme avec  $M = 10\ 000$ .

Comparer les valeurs données par le programme à celles fournies par le tableur pour la loi binomiale de paramètres  $n$  et  $p$ .