QUELQUES ACTIVITÉS AVEC R
LOGICIEL POLYVALENT PROFESSIONNEL DE STATISTIQUE
Illustrations en géométrie, analyse, probabilité et statistique au lycée.

Hubert RAYMONDAUD - LEGTA Louis Giraud à Carpentras-Serres

I – INTRODUCTION

A – PRÉSENTATION GÉNÉRALE

Cette présentation se fera en cinq parties. Les deux premières auront pour support l'analyse exploratoire des données, dont les outils sont la statistique descriptive univariée et bivariée, la troisième portera sur l'analyse mathématique, la quatrième sur la géométrie et la cinquième sur les probabilités et la statistique inférentielle.

Toutes ces activités mettent en œuvre les outils mathématiques de la troisième à la terminale, à l'aide d'algorithmes qui sont traduits en langage **R**. Elles sont utilisables telles quelles (clés en main) mais elles sont "libres" c'est à dire que l'on peut se les approprier, modifier facilement le code des fonctions **R** associées pour les réinvestir dans des activités personnelles.

Ces activités comprennent des illustrations de certains aspects du cours, permettant d'en surmonter les obstacles didactiques. J'ai cependant favorisé autant que possible des exemples pour lesquels on peut faire de l'algorithmique un véritable outil de détermination de solutions approchées et même plus, de résolution de problème, en complément des méthodes mathématiques classiques.

Je suis convaincu que c'est en favorisant ce statut, **tant intellectuellement que matériellement**, qu'on fera sortir l'algorithmique et l'utilisation des TICE en mathématiques, du simple rôle d'exercice d'école imposé au bac et pour lequel, tant les enseignants que les élèves, "optimisent" leur investissement.

POURQUOI R ? R est un logiciel professionnel de statistique, et bien que je l'utilise ponctuellement pour des applications très spécialisées (analyse de variances de plans d'expérimentations agronomiques, ajustements non linéaires), ça n'est pas à lui que je pensais préférentiellement lorsqu'en 2011 j'ai été sollicité pour contribuer au document ressource du nouveau programme de Terminale S. Je n'utilisais alors, en classe et pour mes préparations, que les outils classiques libres, tableurs, GeoGebra, Xcas, Scilab. J'avais pratiqué la programmation de tests statistiques par simulation (tests bootstrap) avec des logiciels propriétaires spécialisés et hors de portée des finances d'un lycée généraliste (SAS, StatXact, Resampling Stats …). Pour mettre en œuvre les simulations et les algorithmes proposés dans les programmes, il me fallait trouver un logiciel libre, polyvalent et dont la programmation soit facilement accessible. J'ai donc essayé les quelques langages de programmation scientifique libres disponibles, Xcas, Python, Scilab, et **R**.

C'est avec **R**[1] (après deux demi journées de formation auprès de Claude Bruchou, ingénieur de recherche à l'INRA d'Avignon) que j'ai réussi le plus rapidement à mettre en œuvre les algorithmes les plus variés sans être obligé de faire appel à des "modules" supplémentaires. Les principaux avantages pratiques de **R** c'est une application de base complète, avec un moteur graphique très performant, une multitude d'outils statistiques et graphiques, un langage de programmation moderne simple à mettre en œuvre[2] grâce à une syntaxe classique et claire et qui tolère une certaine souplesse (symbole d'affectation qui peut être différent du symbole égalité, blocs délimités par des accolades, indentation et espaces, libres …).

De fait, R s'est facilement laissé détourner de son objet initial, pour devenir un outil polyvalent au service d'algorithmes très variés, comme nous allons le montrer tout au long de ce document.

Un autre avantage de poids c'est que **R** est un outil que l'on rencontre quasiment partout dans l'enseignement

[1]Les tableurs et autres GeoGebra ne permettent pas de mettre en œuvre des algorithmes "classique", entrée sorties (avec un tableur les entrées sont en engagement direct, GeoGebra a des cuseurs), boucles, appels à des sous programmes ou fonctions au sens informatique …, tels qu'on les trouve dans la littérature et dont on exige la production de la part des élèves.

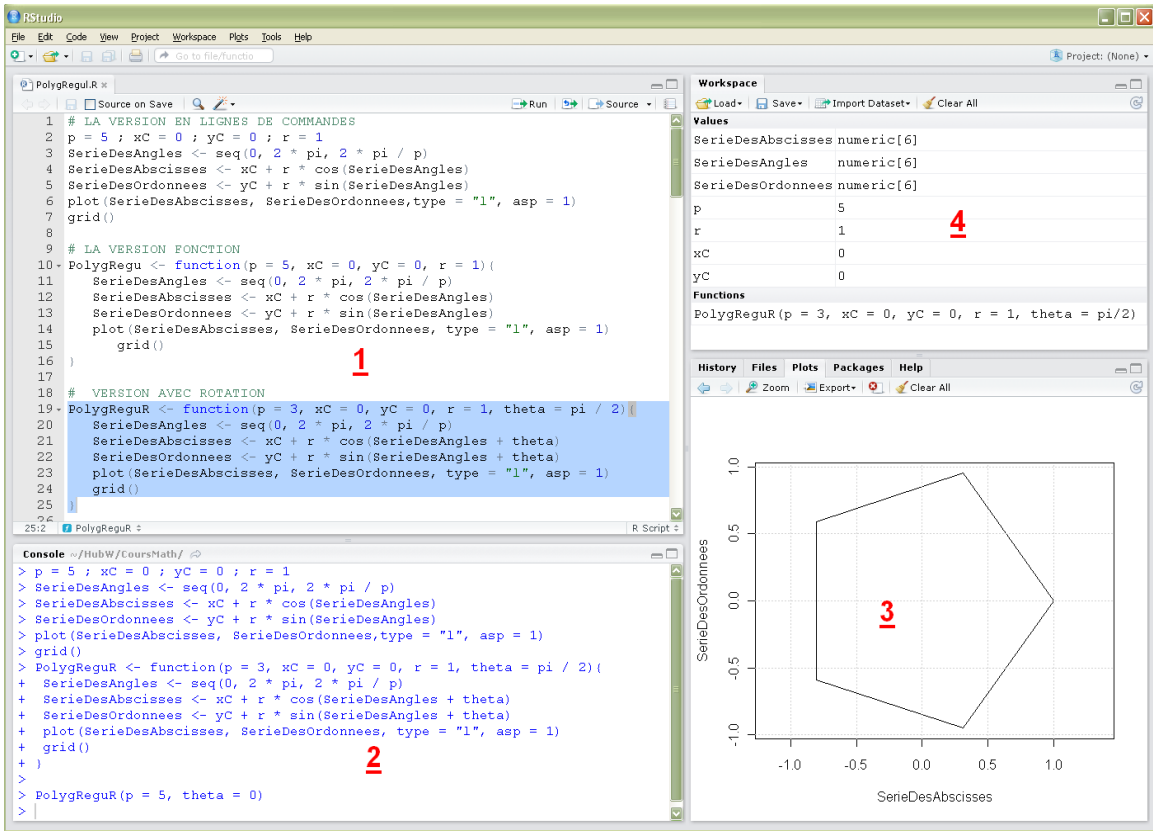
[2]Un bon exemple de cette simplicité est celui de la programmation en **R** (pourtant non spécialisé en géométrie) du tracé d'un polygone (cf page 24), comparée à ce qui est proposé en Algobox dans l'hyperbole de première (2011, page 252). Dans le choix d'un langage il semble intéressant de disposer d'éléments de comparaison – avantages-inconvénients-inadéquation – des diverses possibilités.

supérieur dispensant des cours de statistique[3]. Il est aussi utilisé dans de nombreux instituts de recherche, INRA, INSERM, CNRS, CIRAD, MNHN, qui mettent en ligne de nombreux documents de formation et des forum d'utilisateurs de **R**[4].

B – INSTALLATION ET UTILISATION DE R VIA RSTUDIO

R, comme tous les langages de programmation, s'écrit[5] avec un éditeur de code à coloration syntaxique. Ces éditeurs sont insérés dans un environnement de programmation offrant divers outils rendant plus pratique la manipulation du langage. Il existe deux principaux éditeurs dédiés à **R** : TinnR et RStudio, coloration syntaxique, appariement des délimiteurs (guillemets, parenthèses, accolades …), indentation, complétion et numérotation automatique des lignes, gestion des fenêtres, de l'aide, des fichiers, des packages (modules supplémentaires)... Personnellement je préfère RStudio pour l'utilisation en salle de formation.

Il faut d'abord installer **R** (téléchargé sur http://cran.univ-lyon1.fr/) puis ensuite RStudio (téléchargé sur http://www.rstudio.com/ide/download/). Une fois RStudio installé, il n'y a pas besoin de lancer **R** indépendamment. RStudio gère tous les outils nécessaires. L'environnement de RStudio se partage en quatre principales fenêtres :



La fenêtre **1** est celle du traitement de texte à coloration syntaxique.

Pour exécuter une ligne de commande(s) il suffit de positionner le curseur dans cette ligne (de la fenêtre **1**) et de cliquer sur l'icône "Run". Les lignes sont alors **copiées** dans la console (fenêtre **2**) puis **exécutées, automatiquement.** Le curseur passe alors automatiquement à la prochaine ligne de code.

Pour exécuter plusieurs lignes de commandes il suffit de les sélectionner et de cliquer sur l'icône "Run". Les lignes restent sélectionnées et on peut les exécuter à nouveau (pratique pour des simulations).

On peut aussi saisir des commandes directement dans la console, mais ça n'est pas très utile.

[3] http://pbil.univ-lyon1.fr/R/ ; http://perso.math.univ-toulouse.fr/dejean/2012/09/24/actualites/ ; http://www.biostat.fr/docs/cours1.pdf ;
[4] MNHN : semin-r ; Groupe des utilisateurs du logiciel R CIRAD ; http://math.agrocampus-ouest.fr/infoglueDeliverLive/membres/Francois.Husson ; http://stat.genopole.cnrs.fr/members/jchiquet/teachings/initiation_r ;
[5] Ça n'est pas une obligation technique, on peut très bien utiliser un traitement de texte classique, mais dès que le code devient conséquent il devient très difficile de lire et de corriger les programmes sans ces outils. D'un point de vue pédagogique, il me semble indispensable d'utiliser ces outils de facilitation pour l'apprentissage, l'écriture et la lecture de tels langages.

Lorsque l'on veut exécuter l'ensemble des lignes de code du fichier de la fenêtre de script active, il suffit de cliquer sur l'icône "Source".

Les objets **R** (les constantes, vecteurs, fonctions ...), créés en mémoire vive apparaissent dans la fenêtre **4**. Cette fenêtre est très utile pour détecter les éventuelles erreurs.

Les résultats graphiques apparaissent dans la fenêtre **3**. Les trente derniers graphiques effectués lors d'une session sont enregistrés et on peut les faire défiler en cliquant sur les flèches situées en haut à gauche de la fenêtre.

Les résultats numériques des lignes de commandes que l'on exécute apparaissent dans la console. Par contre lorsque ces lignes concernent la création d'un fonction, celle-ci est seulement stockée en mémoire, ce que l'on peut voir dans la fenêtre **4**, comme par exemple pour la fonction **`PolygRegu(...)`**.

Pour utiliser une fonction que l'on a créée, il faut saisir son nom dans la console, suivi des valeurs des paramètres que l'on veut utiliser. Une caractéristique intéressante des fonctions dans **R** c'est que l'on peut affecter des valeurs par défaut aux paramètres, ce qui est particulièrement utile dans les programmes de simulation, dans lesquels on utilise plusieurs fois le même programme sans changer les valeurs des paramètres.

Pour illustrer cela, prenons l'exemple apparaissant dans la copie d'écran. Glané dans l'hyperbole de première S (2011, page 252), c'est un thème d'approche algorithmique et géométrie dans lequel il s'agit de programmer la construction d'un polygone régulier à p = 3 côtés. Un programme Algobox de 35 lignes (!) est proposé au décryptage.

Je présente le programme **R** suivant, de 6 lignes, correspondant à la construction d'un polygone à p côtés, dont certaines grandeurs sont paramétrables :

```
1 # LA VERSION EN LIGNES DE COMMANDES
2 p <- 5 ; xC <- 0 ; yC <- 0 ; r <- 1
3 SerieDesAngles <- seq(from = 0, to = 2 * pi, by = 2 * pi / p)
4 SerieDesAbscisses <- xC + r * cos(SerieDesAngles)
5 SerieDesOrdonnees <- yC + r * sin(SerieDesAngles)
6 plot(SerieDesAbscisses, SerieDesOrdonnees, type = "l", asp = 1)
7    grid()
8
9 # LA VERSION FONCTION
10 PolygRegu <- function(p = 5, xC = 0, yC = 0, r = 1){
11    SerieDesAngles <- seq(from = 0, to = 2 * pi, by = 2 * pi / p)
12    SerieDesAbscisses <- xC + r * cos(SerieDesAngles)
13    SerieDesOrdonnees <- yC + r * sin(SerieDesAngles)
14    plot(SerieDesAbscisses, SerieDesOrdonnees, type = "l", asp = 1)
15       grid()
16}

Si l'on exécute PolygRegu() on obtiendra un polygone à p = 5 côtés,
"centré" en (xC = 0, yC = 0), de "rayon" de construction r = 1.
Si l'on veut un polygone à 7 côtés, on peut exécuter PolygRegu(p = 7) ou
bien PolygRegu(7). Si c'est le "rayon" seul que l'on veut modifier on
exécute PolygRegu(r = 3). Si l'on veut modifier p et r on exécute
PolygRegu(p = 9, r = 2) ou bien PolygRegu(9,,,2).

Les lignes 1 à 16 sont enregistrées dans le fichier texte "PolygRegul.R".

On peut donc, dans un tel fichier, enregistrer des procédures sous forme
de lignes de commandes ou bien des procédures sous forme de fonctions.
Avec un traitement de texte classique, ces lignes apparaîtraient comme du
texte normal. Avec RStudio la coloration syntaxique fera ressortir les
commandes et l'on y disposera d'autres aides à la programmation.
```

Les numéros de ligne ne font pas partie du programme.
Les lignes de commentaires commence par un #.
Ligne 2 : Contient les initialisations de paramètres. Quand il y a plusieurs commandes sur une même ligne elles sont séparées par des point-virgules.
<- est l'opérateur d'affectation. On peut aussi utiliser -> ou =.
Ligne 3 : seq(...) crée une suite de p valeurs de 0 à 2×pi, de 2×pi/p en 2×pi/p.
Ligne 4 : Calcul de la série des p valeurs des abscisses des sommets du polygone.
Ligne 5 : Calcul de la série des p valeurs des ordonnées des sommets du polygone.
Ligne 6 : Tracé des segments (type = "l") reliant les p sommets. asp = 1 sert à réaliser un repère orthonormé
Ligne 7 : Tracé du quadrillage sur le graphique fait par plot.
Dans le code d'une fonction, l'affectation des valeurs par défaut pour les paramètres se fait par l'opérateur = (et non <-) : (p = 5, xC = 0 ...)

On appelle procédure **R** une suite de lignes d'instructions en code **R** permettant de mettre en œuvre des méthodes et outils mathématiques, graphiques et informatiques. Un ligne d'instruction peut contenir plusieurs commandes séparées par des point-virgules. Les commandes **R** sont toujours des fonctions **R**. On dit que c'est un langage fonctionnel[6]. Les lignes d'instructions peuvent être exécutées seules (on parle alors de lignes de commandes) ou à l'intérieur d'une fonction que l'on construit. Les blocs d'instructions sont délimités par des accolades {…}. **R est sensible à la casse !**

[6]Sur ce sujet, on pourra consulter l'articles de Guillaume Conan sur un autre langage fonctionnel CAML, http://revue.sesamath.net/spip.php?article216 , et un article plus général sur l'interaction algèbre-informatique, http://revue.sesamath.net/spip.php?article497.

Dans les procédures **R,** les lignes de commandes sont en `orange` ou `noir`, les fonctions **R** et leurs paramètres en `bordeaux`, les résultats en *`vert italique`*. Le nom des fonctions créées par programmation sont en `bleu`. Les couleurs sont importantes, voire indispensables avec les élèves, pour la lisibilité des programmes.

Il existe un certain nombre de “cartes de références” résumant les principales fonctions **R** avec leurs syntaxes, bien utiles comme aides mémoires. J'en propose quelque-unes en documents joints.

RStudio fourmille d'autres fonctionnalités de programmeurs, très pratiques, que l'on peut découvrir dans http://www.rstudio.com/ide/docs/ et qui facilitent l'édition et la gestion des fichiers de code.

II – RÉINVESTIR LES OUTILS DES STATISTIQUES DESCRIPTIVES À UNE VARIABLE

TRAVAIL DIRIGÉ SUR L'ÉTUDE DE LA SATISFACTION DES CLIENTS DES GÎTES RURAUX EN MEURTHE ET MOSELLE

A – INTRODUCTION DE LA PROBLÉMATIQUE ET DONNÉES

Afin de faire le point sur la qualité de ses services et de les améliorer, la fédération des gîtes ruraux de Meurthe et Moselle (54) a lancé une grande enquête auprès des hôtes des gîtes ruraux du département. Le questionnaire comprend des questions de détails sur la qualité du gîte, de l'accueil et des services proposés.

À partir des appréciations obtenues aux différentes questions, on calcule une note globale de satisfaction, entre 0 et 10, pour l'ensemble des conditions du séjour. Un extrait (non représentatif et non aléatoire) des notes de l'enquête figure dans le tableau suivant, où l'on a noté l'origine géographique de la personne enquêtée. En effet, cette origine est importante car les attentes varient souvent sensiblement selon l'origine géographique des hôtes.

NoteEst	5,23	4,20	8,65	6,75	1,75	6,51	7,16	7,80	8,26	7,39
NoteSudOuest	8,10	4,48	4,66	8,38	4,25	4,33	8,47	8,21	8,65	4,17
NoteSudEst	4,60	7,39	9,20	8,41	4,27	9,64	5,03	5,74	4,06	5,36

L'objectif d'un premier traitement est de comparer la satisfaction des hôtes en fonction de la région d'origine.

L'échantillon n'ayant pas été choisi au hasard, il n'est pas représentatif du panel de l'enquête, ni de la population enquêtée. Les résultats des traitements ne seront donc valables que pour l'extrait utilisé.

B – NOTION DE STRUCTURE DU TABLEAU DE DONNÉES

Les données ne doivent pas être saisies avec la structure ci-dessus, 3 lignes et 11 colonnes ou bien la structure “transposée”, 11 lignes et 3 colonnes car ça ne correspond pas à un tableau individu statistique × variable. En fait nous avons deux variables, la variable “note” et la variable “origine” et 30 individus statistiques (clients des gîtes ruraux). Le tableau de données aura donc 2 colonnes et 30 lignes.

Cette notion de structure de données est importante car elle va conditionner le traitement par les logiciels de statistique (un tableur n'est pas un logiciel de statistique).

Elle a aussi un intérêt pédagogique dans la mesure où elle oblige à identifier correctement les variables, à déterminer leur “nature”, quantitative, qualitative et à préciser leur rôle dans le traitement, variable étudiée (note, origine), variable permettant de faire des groupes (origine). Il est intéressant de remarquer qu'une variable peut avoir les deux rôles.

Les données peuvent être saisies soit dans un tableur classique, soit directement dans **R** .

Cette séquence ne met en œuvre que des lignes de commande **R**.

C – IMPORTATION DES DONNÉES DEPUIS UN FICHIER TEXTE AU FORMAT ".CSV"

1° Les données sont saisies dans deux colonnes (cf. ci-dessous), dans une feuille (unique) d'un classeur de tableur. **La saisie devra obligatoirement commencer cellule A1**. Le fichier **"TroisDistribEx1T2.csv"** est enregistré au format "csv". On prendra le point-virgule comme séparateur de champs (les colonnes).

2° Importer les données du fichier, avec la procédure suivante, en adaptant le chemin du dossier. La fonction **`setwd(...)`** précise l'adresse du dossier actif. **`read.table(...)`** importe le tableau, **`sep`** indique le séparateur de champ (les variables en colonne), **`header`** indique la présence d'une ligne de noms de variables et **`dec`** le séparateur décimal.

```
setwd("E:/HubW/CoursMath/CoursStatDescrip/DesUniv")
(satis <- read.table("TroisDistribEx1T2.csv", sep = ";", header = TRUE, dec = ","))
```

	origine	note		origine	note		origine	note		origine	note
1	est	5.23	9	est	8.26	17	sudouest	8.47	25	sudest	4.27
2	est	4.20	10	est	7.39	18	sudouest	8.21	26	sudest	9.64
3	est	8.65	11	sudouest	8.10	19	sudouest	8.65	27	sudest	5.03
4	est	6.75	12	sudouest	4.48	20	sudouest	4.17	28	sudest	5.74
5	est	1.75	13	sudouest	4.66	21	sudest	4.60	29	sudest	4.06
6	est	6.51	14	sudouest	8.38	22	sudest	7.39	30	sudest	5.36
7	est	7.16	15	sudouest	4.25	23	sudest	9.20			
8	est	7.80	16	sudouest	4.33	24	sudest	8.41			

3° Vérifications du contenu importé dans le "data.frame" **`satis`**. Un data.frame est un objet **R,** en mémoire vive, qui contient un tableau de données. Les variables (colonnes) qu'il contient peuvent être de différents types, quantitatives ou qualitatives. On vérifie les noms de variables, les dimensions du tableau et le type des variables. La variable "note" est quantitative, la variable "origine" est qualitative, importée automatiquement comme "factor" par **R**.

```
names(satis)
[1] "origine" "note"

str(satis)
'data.frame':   30 obs. of  2 variables :
 $ origine: Factor w/ 3 levels "est","sudest",..: 1 1 1 1 1 1 1 1 1 1 ...
 $ note   : num  5.23 4.2 8.65 6.75 1.75 6.51 7.16 7.8 8.26 7.39 ...
```

D – DESCRIPTION NUMÉRIQUE ET GRAPHIQUE DE LA VARIABLE QUALITATIVE "origine" SE PARTAGEANT EN MODALITÉS

1° Description numérique

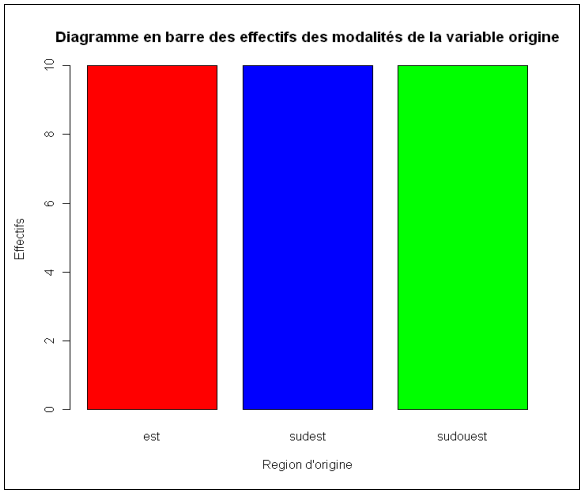
Il s'agit de faire un tableau des effectifs des modalités de la variable "origine". On dit que l'on effectue un tri à plat de la variable "origine".

```
table(satis$origine)
     est   sudest sudouest
      10       10       10
```

2° Description graphique

La fonction **`plot(...)`** comprend la construction du tableau des effectifs, nécessaire au diagramme en barres.

```
plot(satis$origine,
   main = "Diagramme en barre des effectifs des
modalités de la variable origine",
   xlab = "Region d'origine",
   ylab = "Effectifs",
   col = c("red", "blue", "green"))
```



D'autres variantes d'utilisation de la fonction **`plot(...)`** et d'autres exemples de fonctions graphiques figurent dans le fichier **"R_TroisDistrib.r"** et peuvent être expérimentées directement.

E – DESCRIPTION GRAPHIQUE ET NUMÉRIQUE DE LA VARIABLE QUANTITATIVE “note” *CROISÉE AVEC LA VARIABLE QUALITATIVE “origine”* *(groupes d'individus statistiques)*

Il s'agit de **juxtaposer** des représentations permettant de faciliter la description et la comparaison des distributions des variables dans les séries de notes des différentes régions. Il existe un grand nombre de graphiques permettant de résumer graphiquement les séries statistiques. Nous en verrons trois : les “branches et feuilles”, les “boites à pattes” (= “boites de dispersion” = “boites à moustaches”), et les histogrammes. Nous verrons ensuite les résumés numériques plus classiques.

1° Juxtaposer des “branches et feuilles” (“stem and leaf”)

C'est une représentation semi-graphique qui peut revêtir des aspects très variés. Dans l'exemple suivant le paramètre `scale` permet de changer d'échelle de représentation.

Les sorties fournies par **R** doivent être complétées et mises en forme pour afficher la présentation suivante :

```
stem(note[origine == "sudouest"],
scale = 1)
  The decimal point is at the |
  0 |.......................
  1 |.......................
  2 |.......................
  3 |.......................
  4 | 23357.................
  5 |.......................
  6 |.......................
  7 |.......................
  8 |.12457.................
  9 |.......................
```

```
stem(note[origine == "sudest"],
scale = 2)
  The decimal point is at the |
  0 |.......................
  1 |.......................
  2 |.......................
  3 |.......................
  4 | 136...................
  5 | 047...................
  6 | .......................
  7 |.4.....................
  8 |.4.....................
  9 |.26....................
```

```
stem(note[origine == "est"],
scale = 2)
  The decimal point is at the |
  0 |.......................
  1 |.8......................
  2 |........;..............
  3 |.......................
  4 |.2......................
  5 |.2......................
  6 | 58.....................
  7 | 248....................
  8 |.37.....................
  9 |.......................
```

Pour faciliter la compréhension de ces représentations, on peut trier et ordonner les différentes séries.

`note[origine == "sudouest"]` extrait les valeurs de la variable note des individus statistiques correspondant à la modalité sudouest de la variable origine.

```
sort(note[origine == "est"])
 [1] 1.75 4.20 5.23 6.51 6.75 7.16 7.39 7.80 8.26 8.65
sort(note[origine == "sudest"])
 [1] 4.06 4.27 4.60 5.03 5.36 5.74 7.39 8.41 9.20 9.64
sort(note[origine == "sudouest"])
 [1] 4.17 4.25 4.33 4.48 4.66 8.10 8.21 8.38 8.47 8.65
```

2° Juxtaposer des graphiques en “boites” (=“boites à moustaches”)

Tout comme les “branches et feuilles”, il en existe de multiples déclinaisons qui sont activées par l'intermédiaire des paramètres de la fonction `plot(...)`. Il est intéressant de noter que selon le type des données qu'on lui fournit, `plot` propose des représentations graphiques différentes et adaptées. On peut aussi utiliser la fonction `boxplot(...)` avec une syntaxe sensiblement différente.

`attach(...)` sert à mettre en mémoire le nom du data.frame (tableau de données), pour éviter de répéter son nom quand on utilise ses variables :

`plot(satis$origine, satis$note, …)` se simplifie alors en `plot(origine, note, …)`.

Le paramètre `varwidth = TRUE` construit des boites dont la largeur est proportionnelle à la racine carrée de l'effectif de la modalité représentée.

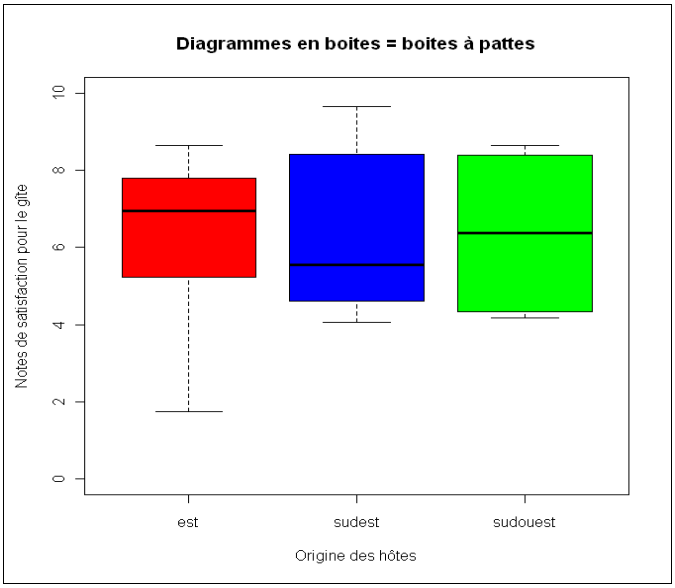
```
attach(satis)
plot(origine, note, horizontal = FALSE,
  col = c("red", "blue", "green"),
  main = "Diagrammes en boites = boites à pattes",
  xlab = "Origine des hôtes",
  ylab = "Notes de satisfaction pour le gîte",
  ylim = c(0, 10),
  varwidth = TRUE)
```

Comment est construite une boite à pattes ?
À l'aide des paramètres suivants:

```
plot(origine, note, plot = FALSE)$stats
      [,1] [,2] [,3]
[1,] 1.750 4.06 4.17
[2,] 5.230 4.60 4.33
[3,] 6.955 5.55 6.38
[4,] 7.800 8.41 8.38
[5,] 8.650 9.64 8.65
```

On les retrouve avec la fonction `quantile(...)` de type 2 (cf. définition** page 8) :

```
quantile(note[origine == "est"], type = 2)
   0%   25%   50%   75%  100%
1.750 5.230 6.955 7.800 8.650
quantile(note[origine == "sudest"],
         type = 2)
  0%  25%  50%  75% 100%
4.06 4.60 5.55 8.41 9.64
```



Le même résultat est donné par la fonction suivante :

```
boxplotplot(note ~ origine, horizontal = FALSE,
  col = c("red", "blue", "green"),
  main = "Diagrammes en boites = boites à pattes",
  xlab = "Origine des hôtes",
  ylab = "Notes de satisfaction pour le gîte",
  ylim = c(0, 10),
  varwidth = TRUE)
```

horizontal = TRUE fait des boites horizontales.

```
quantile(note[origine == "sudouest"],
         type = 2)
  0%  25%  50%  75% 100%
4.17 4.33 6.38 8.38 8.65
```

note[origine == "sudouest"] : […] extrait de la série des notes celles correspondant aux individus dont la modalité de la variable origine est sudouest.

R propose **9 "types"** c'est à dire façons de calculer des quantiles[7], dont le détail figure dans l'aide (saisir **? quantile** dans la console).
De même il existe plusieurs façons de tracer les moustaches des boites. Par défaut **R** propose le modèle original inventé par Tukey[8], dans lequel la moustache supérieure monte jusqu'à la plus grande valeur de la série inférieure à $Q_3 + 1{,}5 \times (Q_3 - Q_1)$, la moustache inférieure descend jusqu'à la plus petite valeur de la série supérieure à $Q_1 - 1{,}5 \times (Q_3 - Q_1)$ et dans lequel les valeurs au delà des moustaches sont marquées par des points.
Malheureusement les documents ressources n'ont pas suivi cette définition d'origine.
On peut facilement construire une fonction correspondant aux définitions rencontrées dans les documents ressources.

** Les quartiles de type 2 correspondent à ceux du mode de calcul proposé originellement par Tukey : Dans une série notée $x_{(n)}$ dans l'ordre croissant et $x^{(n)}$ dans l'ordre décroissant, $Q_1 = x_{((ent(n/2)+1)/2)}$, $Q_2 = x_{(n/2)} = x^{(n/2)}$, $Q_3 = x^{((ent(n/2)+1)/2)}$.

3° Juxtaposer des histogrammes.

par(mfrow = c(3, 1) partage la fenêtre graphique en 3 lignes et 1 colonne définissant des régions dans lesquelles viendront se juxtaposer les 3 histogrammes.

La fonction **hist(...)** est pilotée par de nombreux paramètres qui sont l'occasion d'illustrer les modes de construction de ce graphique et de donner du sens à ses caractéristiques.

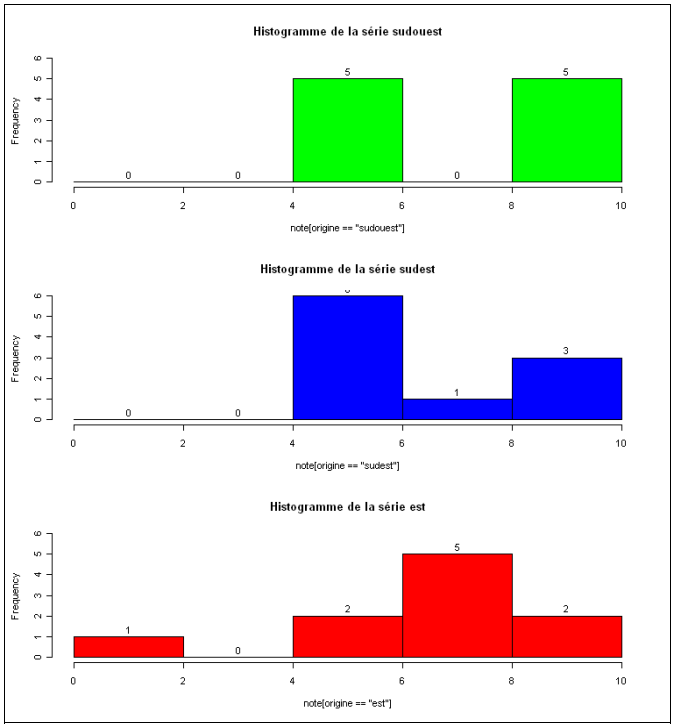
- **breaks(...)** fixe les valeurs des bornes des classes.
- **right = FALSE** fixe des intervalles du type [a ; b[(par défaut ce sont des classes à l'anglo-saxone] a ; b].
- **labels = TRUE** fait figurer les valeurs représentées, au dessus des rectangles.
- par défaut et pour des classes d'égale étendue, l'axe des ordonnées représente les effectifs (= frequency en anglais, attention au faux ami) des classes. Lorsque les classes sont d'étendues inégales ou lorsque l'on force le paramètre **freq = FALSE**, l'axe des ordonnées représente la densité de classe c'est à dire l'effectif divisé par l'étendue de la classe.
- **include.lowest = TRUE** (par défaut) ferme soit le premier soit le dernier intervalle (selon **que right = TRUE** ou **FALSE**), qui prend la forme [a ; b].

[7]Hyndman, R. J. and Fan, Y. (1996) Sample quantiles in statistical packages, American Statistician, 50, 361–365.
[8]Tukey, John Wilder (1977). Exploratory Data Analysis. Addison-Wesley. ISBN 0-201-07616-0.

Lorsque l'on ne précise aucun paramètre, les valeurs par défaut sont automatiquement utilisées.

Un exemple numérique supplémentaire, avec des valeurs entières illustrant l'effet de **right** et de **freq** figure dans le fichier **"R_TroisDistrib.r"** contenant le code **R** des exemples de ce document.

```
(bornes2 <- seq(0, 10, 2))
par(mfrow = c(3, 1))

hist(note[satis == "sudouest"], col = "green",
   main = "Histogramme de la série sudouest",
   breaks = bornes2,
   right = FALSE,
   ylim = c(0, 6),
   labels = TRUE)

hist(note[satis == "sudest"],col = "blue",
   main = "Histogramme de la série sudest",
   breaks = bornes2,
   right = FALSE,
   ylim = c(0, 6),
   labels = TRUE)

hist(note[satis == "est"], col = "red",
   main = "Histogramme de la série est",
   breaks = bornes2,
   right = FALSE,
   ylim = c(0, 6),
   labels = TRUE)
```



Ces différents outils graphiques mettent bien en évidence les différences des distributions observées de la variable note dans les trois séries.

La suite va nous montrer que l'exploration graphique des données est une étape indispensable de l'analyse exploratoire des données, et que les résumés numériques contiennent souvent des pièges que les méthodes graphiques permettent facilement de déjouer.

4° Résumés numériques par modalité d'origine

Dernière étape de l'exploration élémentaire des données, elle consiste à calculer quelques paramètres numériques classiques (quantiles, moyenne, écart type) pour chacune des trois séries correspondant aux trois modalités de la variable "origine". La fonction **aggregate(...)** va nous permettre d'obtenir ces résultats.

Les deux premiers paramètres doivent être des listes, la première est la variable quantitative à résumer, la deuxième concerne la variable contenant les modalités. Le dernier paramètre **FUN** doit être une fonction indiquant le paramètre à calculer. Ce peut être une fonction native de **R** (length, mean, sd, quantile …) mais ce peut être toute fonction créée par l'utilisateur. Une fonction (un objet fonction de **R**) est créée par la fonction **function(...)**. Selon la syntaxe suivante :

mafonction <- function(variable(s) = valeurpardéfaut ou rien){procédure définissant la fonction} .

► **length** est l'effectif des modalités de la variable "origine".

```
aggregate(list(NOTE = note), by = list(ORIGINE = origine), FUN = length)
   ORIGINE NOTE
1      est   10
2   sudest   10
3 sudouest   10
```

► **quantile** calcule les quantiles d'ordre 0 %, 25 %, 50 %, 75 % et 100 %, de type 7 (par défaut).

```
aggregate(list(NOTE = note), by = list(ORIGINE = origine), FUN = quantile)
   ORIGINE NOTE.0% NOTE.25% NOTE.50% NOTE.75% NOTE.100%
1      est  1.7500   5.5500   6.9550   7.6975    8.6500
2   sudest  4.0600   4.7075   5.5500   8.1550    9.6400
3 sudouest  4.1700   4.3675   6.3800   8.3375    8.6500
```

► **mean** calcule la moyenne.

```
aggregate(list(NOTE = note), by = list(ORIGINE = origine), FUN = mean)
   ORIGINE NOTE
1      est 6.37
2   sudest 6.37
3 sudouest 6.37
```

► **SDn**, que l'on définit avec **function(..){...}**, calcule l'écart type de la série à partir de la fonction sd de **R** qui, elle, calcule l'écart type estimé (racine(SCE/(n-1))) alors que l'écart type de la série vaut (racine(SCE/n)), SCE : Somme des Carrés des Écarts (à la moyenne).

```
SDn <- function(x){sqrt(sd(x)^2 * (length(x) - 1) / length(x))}
aggregate(list(NOTE = note), by = list(ORIGINE = origine), FUN = SDn)
   ORIGINE     NOTE
1      est 1.999920
2   sudest 1.999885
3 sudouest 2.000480
```

► **QuantBox** permet de calculer les quantiles selon la méthode de type 2 (méthode de Tukey).

```
QuantBox <- function(x){quantile(x, type = 2)}
aggregate(list(NOTE = note), by = list(ORIGINE = origine), FUN = QuantBox)
   ORIGINE NOTE.0% NOTE.25% NOTE.50% NOTE.75% NOTE.100%
1      est   1.750    5.230    6.955    7.800     8.650
2   sudest   4.060    4.600    5.550    8.410     9.640
3 sudouest   4.170    4.330    6.380    8.380     8.650
```

On remarque que les trois séries ont la même valeur de la moyenne et de l'écart type, au millième près. Se contenter de ces simples résumés numériques, comme c'est trop souvent le cas dans la pratique, ne permet pas de mettre en évidence les différences entre ces trois distributions observées.

R propose plusieurs autres outils graphiques que l'on peut découvrir par l'intermédiaire de nombreux documents en ligne et de divers forum d'utilisateurs d'institutions d'enseignement et de recherche déjà cités.

L'article Le logiciel R comme outil d'initiation à la statistique descriptive : enquête sur les dépenses des ménages de la revue en ligne "Statistique et Enseignement", propose des exemples concrets de réinvestissement des outils de la statistique descriptive du collège et du lycée.

Lorsque l'on a fini d'exploiter les données du data.frame "satis" il ne faut pas oublier de le "détacher" en faisant **detach(satis)**.

III – DESCRIPTION ET MODÉLISATION DE LA RELATION ENTRE DEUX VARIABLES QUANTITATIVES

RELATION ENTRE LE VOLUME FILTRÉ ET LE TEMPS DE FILTRATION COMPARAISON DE PLUSIEURS STRATÉGIES D'AJUSTEMENT AVEC R

Il s'agit d'étudier la relation entre un volume (en centilitres) de liquide filtré et le temps (en secondes) de filtration, dans un processus de fabrication d'un liquide physiologique stérile. (ringer). Dans cette séquence on crée des fonctions **R** avec les fonctions mathématiques ajustées sur les nuages observés.

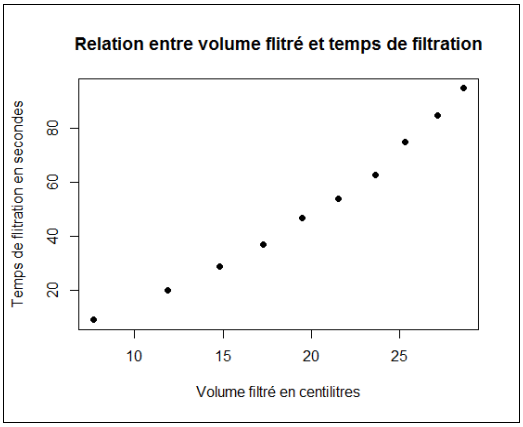
► Saisie des données sous forme de tableau et tracé du nuage de points (volu ; temps) :

Les parenthèses encadrant la totalité de la commande permettent d'afficher directement le contenu de l'objet créé. On peut ainsi éviter de saisir à nouveau le nom de l'objet, pour faire afficher son contenu.

c est l'opérateur de concaténation, il permet de construire les "vecteurs" (objets de **R** correspondants à une liste indicée) de données :

```
(filtra <- data.frame(
   volu = c(7.7, 11.9, 14.8, 17.3, 19.5, 21.5, 23.6, 25.3, 27.1, 28.6),
   temps = c(9, 20, 29, 37, 47, 54, 63, 75, 85, 95)))

attach(filtra)
```

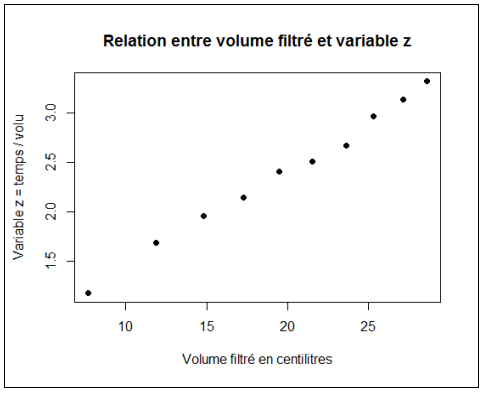
```
Extrait du tableau R nommé filtra
  volu temps
1  7.7     9
2 11.9    20
3 14.8    29
4 17.3    37
5 19.5    47
...

plot(volu, temps, pch = 19,
  xlab = "Volume filtré en centilitres",
  ylab = "Temps de filtration en secondes",
  main = "Relation entre volume filtré et temps
de filtration")

Le nuage de points ne s'allonge pas autour d'une
droite. Il faut donc rechercher une
transformation. pch = 19, code la forme du point
```



► On créé une variable z = temps/volu, on l'affiche et on en fait la représentation graphique :

Les commandes et les résultats :

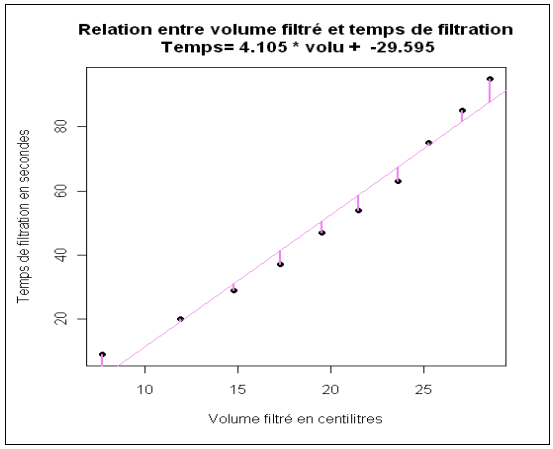
```
(z <- temps / volu)
Les 3 premières valeurs de la variable z
[1] 1.168831 1.680672 1.959459

plot(volu, z, pch = 19,
   xlab = "Volume filtré en centilitres",
   ylab = "Variable z = temps / volu",
   main = "Relation entre volume filtré et
variable z")

Le nuage de points s'allonge autour d'une
droite. On peut donc utiliser le modèle
affine pour résumer la relation.
```



► L'ajustement du nuage (volu ; temps) : un mauvais modèle

● calcul des paramètres :

La fonction `lsfit(...)` enregistre ses résultats dans une liste dont on fait afficher les éléments en faisant précéder leur nom par `$`.




```
ModAfY <- lsfit(volu, temps)

ModAfY$coefficients
  Intercept           X
-29.594564    4.105148
```

Le modèle s'écrit :

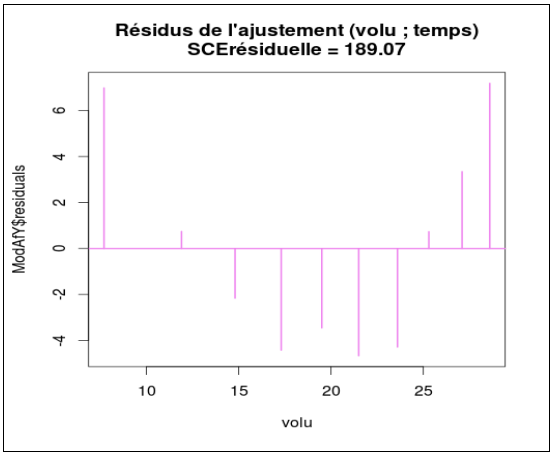
temps = 4,1051148 × volu – 29,594564

`plot` trace le nuage, `abline` ajoute la droite sur le nuage, `paste` permet d'afficher des messages avec le contenu de variables :

```
plot(volu, temps, pch = 21, col = "black", bg = "black",
    xlab = "Volume filtré en centilitres",
    ylab = "Temps de filtration en secondes",
    main = paste("Relation entre volume filtré et temps de filtration\n",
      "Temps=", round(ModAfY$coefficients[2], 3),
      "* volu + ", round(ModAfY$coefficients[1], 3)))
  abline(ModAfY$coefficients, col = "violet")
```

(Cette procédure ne trace pas les segments représentant les résidus, sur le nuage. La procédure ayant permis ce tracé figure dans le fichier "R_Filtrat.r" contenant tous les codes de cette séquence.)

● La représentation graphique du nuage des résidus



Les commandes et les résultats :

```
ModAfY$residuals
 [1]  6.9849267  0.7433064 -2.1616219
...
```

On peut aussi tracer les résidus sur le nuage (volu ; temps), cf. fichier "R_Filtrat.r".

```
(SCErY <- sum(ModAfY$residuals^2))
[1] 189.0699

(cor(volu, temps))^2
[1] 0.9738868
```

```
plot(volu, ModAfY$residuals, type = "h", lwd = 2,
      main = paste("Résidus de l'ajustement (volu ; temps)\nSCErésiduelle =",
        round(SCErY, 3)))
   abline(h = 0, lwd = 2)    #  lwd = 2 multiplie l'épaisseur du trait par 2.
```

● Calcul des valeurs estimées par un mauvais modèle

Les commandes et les résultats : On crée une fonction `TempsEst(...)` avec le modèle ajusté pour la série double (volu ; temps)

```
TempsEst <- function(x){
    y <- ModAfY$coefficients[2] * x + ModAfY$coefficients[1]
    return(y)
}

TempsEst(20) = 52.50839
```

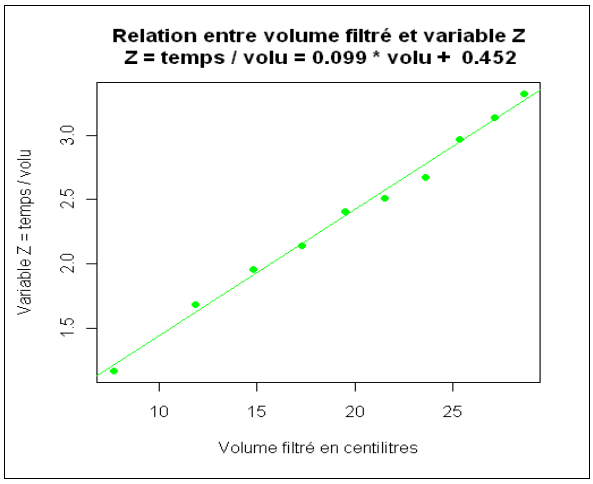
► L'ajustement du nuage (volu ; temps / volu) = (volu ; z) : un meilleur modèle

● calcul des paramètres :

Les commandes et les résultats :

```
ModAfZ <- lsfit(volu, z)


ModAfZ$coefficients
  Intercept           X
 0.45176953  0.09855047
```

Le modèle s'écrit :



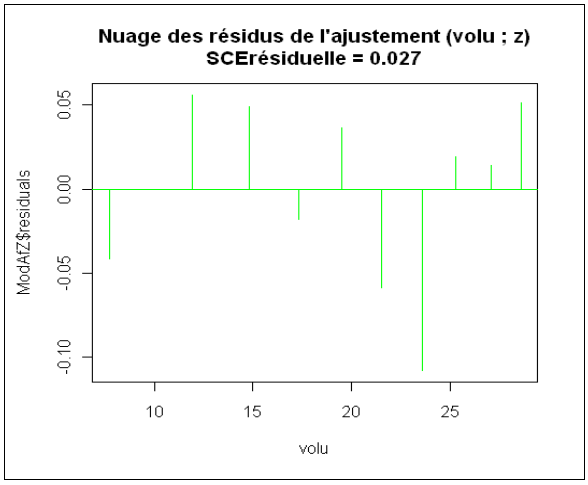
● On trace la droite sur le nuage (volu ; z) :

```
plot(volu, z, pch = 21, col = "green", bg = "green",
    xlab = "Volume filtré en centilitres",
    ylab = "Variable Z",
    main = paste("Relation entre volume filtré et variable Z\n",
      "Temps=", round(ModAfZ$coefficients[2], 3),
      "* volu + ", round(ModAfZ$coefficients[1], 3)))
  abline(ModAfZ$coefficients, col = "green")
```

● La représentation graphique du nuage des résidus de l'ajustement (volu ; z) :

Les commandes et les résultats :

```
ModAfZ$residuals
[1] -0.04177701  0.05615211  0.04914293
...


sum(ModAfZ$residuals^2)
[1] 0.02735338

(cor(volu, z))^2
[1] 0.993314
```




```
plot(volu, ModAfZ$residuals, type = "h", col = "green",
   main = paste("Nuage des résidus de l'ajustement (volu ; z)\nSCErésiduelle =",
     round(SCErZ, 3)))
   abline(h = 0, col = "green")
```

● Détermination du modèle temps = f(volu) à partir de l'ajustement affine (volu ; z). Attention : ce passage par la fonction réciproque a pour conséquence que ce modèle n'est pas optimal quand à la SCE résiduelle, alors que celui de (volu ; z) l'est.

Les commandes et les résultats : On crée une fonction `TempsZest(...)`, temps = f(volu), à partir du modèle ajusté pour la série double (volu ; z)

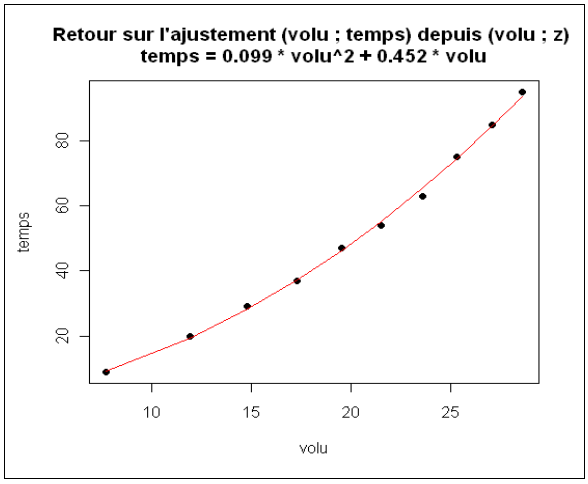
```
TempsZest <- function(x){
    yest <- (ModAfZ$coefficients[2] * x^2 + ModAfZ$coefficients[1] * x)
    return(yest)
}
```

On calcule, avec ce modèle, la SCE attachée à Y

```
(SCErYz <- sum((temps - TempsZest(volu))^2))
[1] 12.34384
```

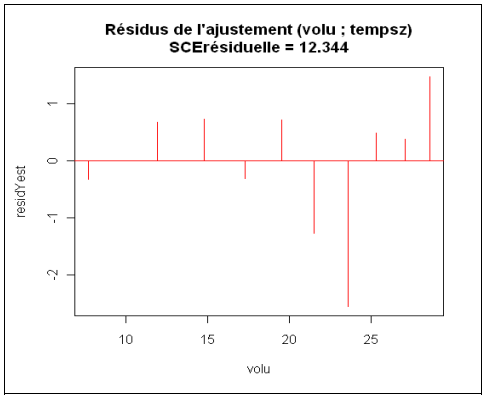
On trace la courbe ajustée sur le nuage observé. **`lines(SérieDesX, SérieDesY, ...)`** permet de tracer la courbe reliant des points repérés par leurs coordonnées cartésiennes,

```
plot(volu, temps, pch = 21, bg = "black",
    main = paste("Retour sur l'ajustement (volu ; temps) depuis (volu ; z)",
      "\ntemps =", round(ModAfZ$coefficients[2], 3), "* volu^2 +",
      round(ModAfZ$coefficients[1], 3), "* volu"))
  lines(volu, TempsZest(volu), col = "red")
```

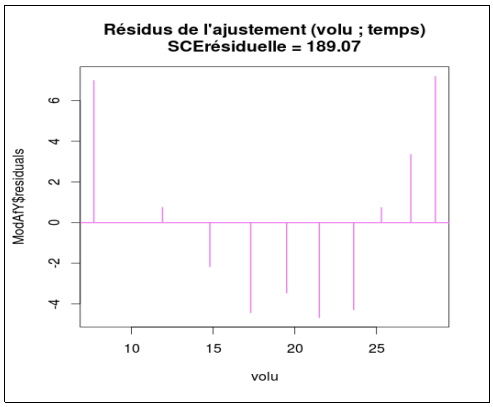


On calcule les résidus et on en fait une représentation graphique

```
residYest <- temps - TempsZest(volu)

plot(volu, residYest, type = "h", col = "red",
    main = paste("Résidus de l'ajustement (volu ; tempsZest)\nSCErésiduelle =",
      round(SCErYz, 3)))
  abline(h = 0, col = "red")
```



à comparer avec :



● **Le calcul des valeurs estimées par le modèle** de l'ajustement temps = f(volu), déduit de l'ajustement (volu ; z) se fait avec la fonction **`TempsZest`** que l'on a créée précédemment :

```
temps = 0.09855047 * volu^2 + 0.45176953 * volu
```

```
TempsZest(20) = 48.45558
```

► Preuve que la transformation de variable n'est pas la stratégie optimale

● La stratégie par transformation de variable a permis d'obtenir le modèle suivant :

*temps = 0.09855047 * volu^2 + 0.45176953 * volu,* avec *SCErYz = 12.34384*

● On va ajuster directement un modèle **temps = a * volu^2 + b * volu** par la méthode des moindres carrés linéaires multiples (les deux variables explicatives sont volu² et volu), assurée par la même fonction **R** **lsfit(...)**.

On prépare le tableau (une matrice avec la fonction **matrix**) des variables explicatives puis on fait l'ajustement :

```
xvol <- matrix(c(volu, volu^2), ncol = 2)

ModAfYxx <- lsfit(xvol, temps, intercept = FALSE)

ModAfYxx$coefficients
       X1        X2
0.4174244 0.1000438
```

Un extrait de la matrice créée :

```
xvol[1:4,]
     [,1]   [,2]
[1,]  7.7  59.29
[2,] 11.9 141.61
[3,] 14.8 219.04
[4,] 17.3 299.29
...
```

● On créé la fonction (**TempsYxx**) du modèle polynomial ajusté à partir de la série double (volu ; temps), et on calcule la SCE résiduelle obtenue avec ce modèle :

```
TempsYxx <- function(x){
    yest <- ModAfYxx$coefficients[1] * x + ModAfYxx$coefficients[2] * x^2
    return(yest)
}

(SCErYxx <- sum((temps - TempsYxx(volu))^2))
[1] 12.114
```

Le modèle est donc :

*temps = 0.1000438 * volu^2 + 0.4174244 * volu,* avec *SCErYxx = 12.114*

On a donc bien obtenu un meilleur ajustement quant à la SCE résiduelle.

On pourrait obtenir encore un SCE résiduelle plus petite en prenant x, x^2, x^3 comme variables explicatives :

*temps = 0.0009059614 * volu^3 + 0.0621767536 * volu^2 + 0.7862258293 * volu*

avec *SCErYx3 = 9.226121*

Avec un nombre de degré suffisamment grand, la SCEr tend vers 0 : la courbe passera par tous les points.

Autant de stratégies que **R** rend faciles à mettre en œuvre.

Mais le critère de la meilleure SCE résiduelle est loin d'être le critère le plus pertinent : On préfère souvent un modèle avec une moins bonne SCEr mais dont les paramètres ont une signification concrète, ce qui est rarement la cas pour les coefficients d'un modèle polynomial.

Lorsque vous changez de data.frame, ne pas oublier de faire : **detach(filtra).**

IV – QUELQUES EXERCICES EN ANALYSE

Il s'agit, pour l'essentiel de mettre en œuvre des algorithmes glanés dans des ouvrages ou des annales de bac, qui illustrent des notions de cours ou permette de trouver des solutions numériques approchées à certains problèmes.

A – APPROXIMATION DE LA SOLUTION D'UNE ÉQUATION PAR DICHOTOMIE

Je suis parti de l'exercice 3 du bac L 2009 étranger pour ensuite proposer trois prolongements, le premier montrant comment on peut construire le tableau des valeurs demandé dans l'énoncé, le second généralisant l'algorithme à un intervalle quelconque [a ; b] sur lequel la fonction est strictement monotone. Le troisième propose une algorithme alternatif plus simple.

EXERCICE 3 **5 points**

Dans cet exercice, toute trace de recherche, même incomplète, ou d'initiative, même non fructueuse, pourra être prise en compte dans l'évaluation.

La fonction f est définie pour tout nombre réel x de l'intervalle [2 ; 1] par

$$f(x) = xe^x - 1.$$

1. Montrer que la fonction dérivée f' de la fonction f est telle que, pour tout nombre réel x de [−2 ; 1], $f'(x) = e^x(1 + x)$.
2. **a.** Étudier le signe de $f'(x)$ pour tout réel x de [−2 ; 1].
   b. Dresser le tableau de variations de la fonction f sur [−2 ; 1].
   c. En vous appuyant sur le tableau de variations de la fonction f, justifier que, sur [−2 ; 1], l'équation $f(x) = 0$ admet une unique solution α et que cette solution appartient à l'intervalle [0 ; 1].
3. On considère l'algorithme suivant :

Entrée : Introduire un nombre entier naturel n
Initialisation : Affecter à N la valeur n.
Affecter à a la valeur 0
Affecter à b la valeur 1.
Traitement : Tant que $b - a > 10^{-N}$
  Affecter à m la valeur $\frac{a+b}{2}$
  Affecter à P le produit $f(a) \times f(m)$
  Si $P > 0$, affecter à a la valeur de m.
  Si $P \leqslant 0$, affecter à b la valeur m.
Sortie : Afficher a
Afficher b.

   a. On a fait fonctionner cet algorithme pour $n = 2$. Compléter le tableau de l'annexe 1 donnant les différentes étapes.
   b. Cet algorithme détermine un encadrement de la solution α de l'équation $f(x) = 0$ sur l'intervalle [0 ; 1]. Quelle influence le nombre entier n, introduit au début de l'algorithme, a-t-il sur l'encadrement obtenu ?

Exercice 3

	m	P	a	b	$b - a$
Initialisation			0	1	
Étape 1					
Étape 2					
Étape 3	0,625	−0,029 446 59	0,5	0,625	0,125
Étape 4	0,562 5	0,002 244 98	0,562 5	0,625	0,062 5
Étape 5	0,593 75	−0,000 960 45	0,562 5	0,593 75	0,031 25
Étape 6	0,578 125	−0,000 391 37	0,562 5	0,578 125	0,015 625
Étape 7	0,570 312 5	−0,000 112 22	0,562 5	0,570 312 5	0,007 812 5

1° L'algorithme proposé ne contient pas la fonction utilisée et se limite à l'intervalle [0 ; 1], ce qui n'est pas cohérent. Si l'on change de fonction il faut pouvoir saisir les bornes correspondantes.

Il faut créer un objet **R** qui contient la fonction étudiée `f` et exécuter son code pour qu'elle soit disponible en mémoire, pour la fonction `Etranger2009` qui met en œuvre l'algorithme. `function(...){...}` Crée la fonction. `cat(...)` Affiche des "messages" et le contenu des objets concernés. `\n` force le retour à la ligne.

Une caractéristique intéressante du langage **R** est que l'on peut proposer des valeurs par défaut pour les variables dans les fonctions. Cette caractéristique est particulièrement utile dans les simulations, comme nous le verrons dans la partie du document sur ce thème.

```
f <- function(x){x * exp(x) - 1}

Etranger2009 <- function(n = 3){
   N <- n ; a <- 0 ; b <- 1
   while(b - a > 10^-N){
      m <- (a + b) / 2
      P <- f(a) * f(m)
      if(P > 0){a <- m} else {b <- m}
   }
   cat("a =", a, "\nb =", b, "\n\n")
}
```

```
Etranger2009()
a = 0.5664062
b = 0.5673828

Etranger2009(6)
a = 0.5671425
b = 0.5671434
```

La variable N est inutile. Le passage de n à N ne sert à rien.

2° Premier prolongement montrant la construction du tableau de valeurs demandé.

`matrix(...)` construit un tableau de base de 6 colonnes et 1 ligne de 0. Ensuite la première ligne prend pour valeurs les valeurs initiales (`tablo[1,]`), l'indiçage des éléments du tableau est fait par `[sériedenumérosde lignes, seriesdenumérosdecolonnes]`, l'absence d'indice désignant tous les éléments : `[1,]` désigne la ligne 1 pour toutes les colonnes. `rbind(...)` ajoute des lignes supplémentaires au fur et à mesure de l'avancement des boucles de calcul.

Il est intéressant de faire expérimenter et remarquer que le nombre de boucles de calculs (étapes) dépend de la précision demandée avec le paramètre n. Une précision à 10^{-2} nécessite 7 boucles de calcul (7 étapes).

```
Etranger2009_1 <- function(n){
   N <- n ; a <- 0 ; b <- 1 ; k <- 1
   tablo <- matrix(0, ncol = 6,
      dimnames = list(NULL,
        c("etapes", "m", "P", "a",
          "b","b - a")))
   tablo[1,] <- c(0, NA, NA, a, b,
                  b - a)
   f <- function(x){x * exp(x) - 1}
   while(b - a > 10^-N){
      m <- (a + b) / 2
      P <- f(a) * f(m)
      if(P > 0){a <- m} else {b <- m}
      tablo <- rbind(tablo, c(k, m, P,
                     a, b, b - a))
      k <- k + 1
   }
   print(tablo)
   cat("\na =", a, "\nb =", b, "\n\n")
}
```

```
Avec la même fonction f :
> Etranger2009_1(2)
     etapes         m             P      a         b     b - a
[1,]      0        NA            NA 0.0000 1.0000000 1.0000000
[2,]      1 0.5000000  0.1756393646 0.5000 1.0000000 0.5000000
[3,]      2 0.7500000 -0.1032320388 0.5000 0.7500000 0.2500000
[4,]      3 0.6250000 -0.0294465935 0.5000 0.6250000 0.1250000
[5,]      4 0.5625000  0.0022449794 0.5625 0.6250000 0.0625000
[6,]      5 0.5937500 -0.0009604512 0.5625 0.5937500 0.0312500
[7,]      6 0.5781250 -0.0003913677 0.5625 0.5781250 0.0156250
[8,]      7 0.5703125 -0.0001122238 0.5625 0.5703125 0.0078125

a = 0.5625
b = 0.5703125
```

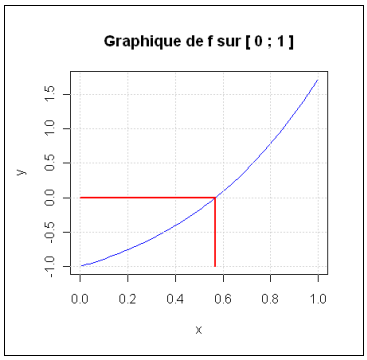
3° Deuxième prolongement dans lequel je vais généraliser l'algorithme précédent à une fonction strictement monotone sur un intervalle [A ; B], avec f(A) × f(B) < 0. À l'utilisation du programme il faut avoir exécuté la fonction concernée. Le programme demande de saisir les valeurs des deux bornes. La précision est à 10^{-4} par défaut, si on veut la modifier, il suffit de saisir la précision (N) lors de l'utilisation.

L'illustration graphique du résultat permet de le valider. On peut y détecter d'éventuelles erreurs d'utilisation.

`plot(f, Ainit, Binit)` trace la courbe représentative de f sur l'intervalle [Ainit ; Binit]. `grid()` trace le quadrillage. `lines(SérieDesX, SérieDesY, ...)` trace les segments de droite rouges du repérage, avec les coordonnées de toutes les extrémités les définissant. J'ai pris le milieu de [a ; b] comme abscisse de la représentation de la solution. Le paramètre `lwd` sert à gérer l'épaisseur des segments.

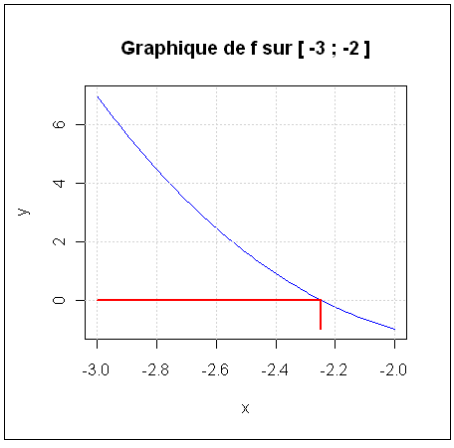
```
Etranger2009_2 <- function(Ainit, Binit, N = 4){
   a <- Ainit ; b <- Binit
   while (b - a > 10^-N){
      m <- (a + b) / 2
      P <- f(a) * f(m)
      if (P > 0) {a <- m} else {b <- m}
   }
   cat("Solution approchée de f(x) = 0 :\n",
      "[", a, ";", b, "]\n\n")
   plot(f, Ainit, Binit, col = "blue",
         xlab = "x", ylab = "y",
         main = paste("Graphique de f sur [",Ainit,
                      ";", Binit, "]"))
      grid()
      lines(c(Ainit, (a + b) / 2, (a + b) / 2),
         c(0, 0, min(c(f(Ainit), f(Binit)))),
         col = "red", lwd = 2)
}
```

```
Avec la fonction :
f <- function(x){x * exp(x) - 1}

Etranger2009_2(0, 1, 2)
Solution approchée de f(x) = 0 :
[ 0.5625 ; 0.5703125 ]
```

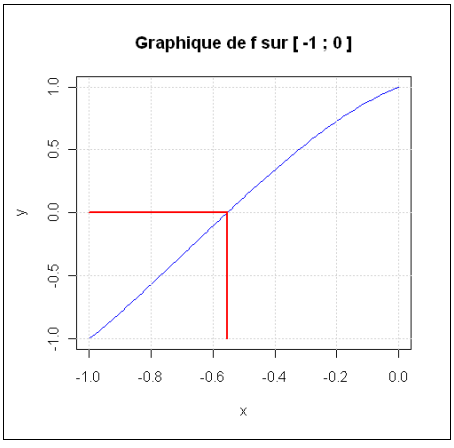


Parmi ces trois exemples, il y a une mauvaise utilisation du programme :

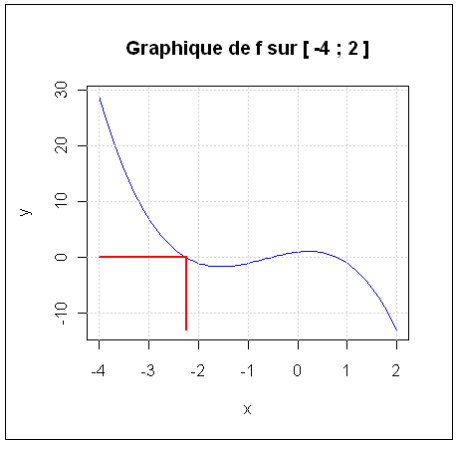
```
Avec :
f <- function(x){- x^3 - 2 * x^2
+ x + 1}

Etranger2009_2(-3, -2)
Solution approchée de f(x) = 0 :
[ -2.247009 ; -2.246948 ]
```




```
Avec :
f <- function(x){- x^3 - 2 * x^2
+ x + 1}

Etranger2009_2(-1, 0)
Solution approchée de f(x) = 0 :
[ -0.5549927 ; -0.5549316 ]
```




```
Avec :
f <- function(x){- x^3 - 2 * x^2
+ x + 1}

Etranger2009_2(-4, 2)
Solution approchée de f(x) = 0 :
[ -2.24704 ; -2.246948 ]
```



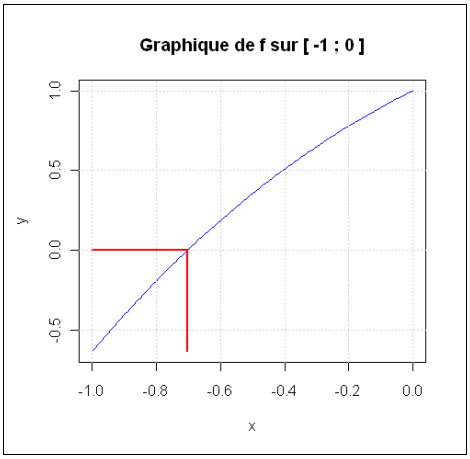
4° Troisième prolongement : une alternative didactique

Les élèves à qui j'ai proposé l'exercice ont eu beaucoup de mal à comprendre l'algorithme, en particulier le fait qu'il marche aussi bien pour une fonction strictement croissante qu'une fonction strictement décroissante.

J'ai donc proposé un autre algorithme (cf. la fonction **`dicot(...)`** dans le fichier **"Dicotomie1.R"**) dans lequel on effectue un traitement différencié pour les deux cas de sens de variation de la fonction. Cet algorithme permet de réfléchir à la méthode de détection du sens de variation de la fonction à faire en début de programme. Il est aussi l'occasion de réfléchir aux critères d'efficacité d'un programme, nombre de lignes, vitesse d'exécution …

```
dicot <- function(A, B, E = .01){
   Ainit <- A ; Binit <- B
   if (f(A) < f(B)){
      while (B - A > E){
         M <- (A + B) / 2
         if (f(M) > 0){B <- M} else {A <- M}
      }
   } else {
      while (B - A > E){
         M <- (A + B) / 2
            if (f(M) > 0){A <- M} else {B <- M}
      }
   }
   # Affichage des résultats et illustration graphique
   cat("Solution approchée de f(x) = 0 :\n",
       "[", A, ";", B, "]\n\n")
   plot(f, Ainit, Binit, col = "blue",
         xlab = "x", ylab = "y",
         main = paste("Graphique de f sur [",Ainit, ";",
                      Binit, "]"))
      grid()
      lines(c(Ainit, (A + B) / 2, (A + B) / 2),
            c(0, 0, min(c(f(Ainit), f(Binit)))),
            col = "red", lwd = 2)
}
```

```
f <- function(x){exp(x) - x^2}

dicot(-1, 0, .001)
Solution approchée de f(x) = 0 :
[ -0.7041016 ; -0.703125 ]
```



Il existe d'autres méthodes numériques, plus efficaces (par exemple plus rapides) de résolution d'équations, comme la méthode de Newton[9].

[9] http://fr.wikipedia.org/wiki/M%C3%A9thode_de_Newton

B – LA SUITE DE SYRACUSE COMME VOUS NE L'AVEZ JAMAIS VUE ?

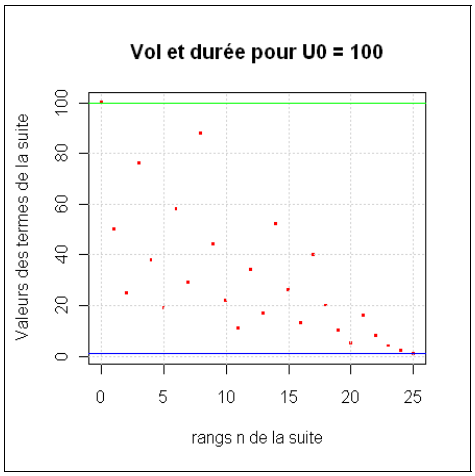
1° L'aspect chaotique du comportement de la suite est bien mis en évidence par la représentation graphique.

`|` est le connecteur logique ou. `!=` est l'opérateur logique "différent de". `...%%k` est l'opérateur modulo k. `c(vecteur, u)` permet d'ajouter l'objet `u` à la fin de l'objet `vecteur`. `which(TestSurUnVecteur)` renvoie tous les indices des éléments du "vecteur" dont le test renvoie "TRUE". L'indiçage des "vecteurs" dans **R** commence à 1. `0:k` génère la suite des nombres entiers de 0 à k. `pch` gère le type de symbole marquant les points, `cex` règle la grosseur des symboles.

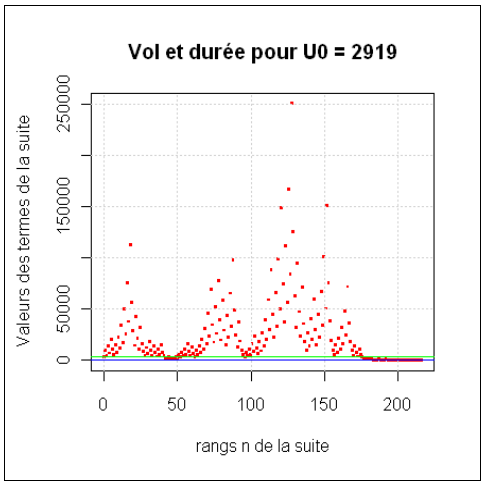
```
# Cette fonction calcule, en fonction de U0, le nombre
# d'itérations nécessaires pour atteindre la valeur 1
# La plus petite valeur d'indice tel que Un+1 < U0
#.ainsi que la valeur maximale de Un. Graphique de Un par n.
# Allez explorer le voisinage de U0 = 2919 **
syracn = function(U0 = 100){
   if (U0 <= 0 | trunc(U0) != U0) {
      cat("U0 doit être entier naturel différent de 0\n")
      stop()
   }
   u <- U0 ; vectu <- U0
   while (u != 1) {
      if (u %% 2 == 0) {u <- u / 2} else {u <- 3 * u + 1}
      vectu <- c(vectu, u)
   }
   k <- length(vectu) - 1
   Min_nU0 <- min(which(vectu < U0))
   MaxUn <- max(vectu)
#***** Affichage des résultats *************
   cat("\nDurée k du vol =", k, "\n")
   cat("La plus petite valeur de n telle que Un < U0, vaut :",
       Min_nU0, "\n")
   cat("La valeur maximale de Un vaut :", MaxUn, "\n")
# **** Affichage des graphiques ************
   plot(0:k, vectu,
         main = paste("Vol et durée pour U0 =",U0)
         xlab = "rangs n de la suite",
         ylab = "Valeurs des termes de la suite",
         pch = ".", cex = 4, col = "red")
      grid()
      abline(h = U0, col = "green")
      abline(h = 1, col = "blue")
}
```

```
syracn()

Durée k du vol = 25
La plus petite valeur de n
telle que Un < 100 vaut : 1
La valeur maximale de Un
vaut : 100

syracn(2919)

Durée k du vol = 216
La plus petite valeur de n
telle que Un < 2919 vaut :
42
La valeur maximale de Un
vaut : 250504
```

```
syracn()
```




```
syracn(2919)
```



2° Réinvestir les statistiques descriptives pour décrire graphiquement le comportement de la suite.

L'efficacité de **R** permet de calculer rapidement toutes les durées de vol pour des séries de valeurs de U0. Ces durées sont enregistrées au fur et à mesure dans le vecteur `vectn` . Dans le graphique représentant les valeurs de k en fonction de U0, la position des axes a été permutée de façon à ce qu'il y ait correspondance avec le diagramme en barre au dessous. `table(vectn)` réalise le tableau des effectifs des valeurs de k de la série contenue dans `vectn`. Tableau que l'on illustre graphiquement par un diagramme en barres.

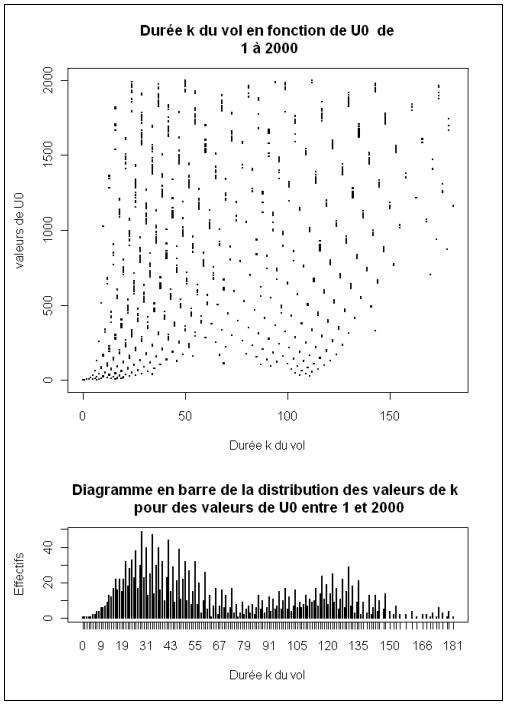
Il est intéressant de noter la "polyvalence" de la fonction `plot` qui, selon le type des données qui lui sont fournies, sait trouver un type de graphique adapté, nuages de points, diagrammes en barres, diagrammes en boites. Lorsqu'on lui fournit un tableau des effectifs, par exemple dans `plot(effec,...)`, `plot` réalise un diagramme en barres. Lorsqu'on lui fournit deux séries numériques, par exemple dans `plot(vectn, U0Inf:U0Sup,...)`, `plot` réalise un nuage de points.

`layout` découpe la fenêtre graphique en 1 colonne et 2 lignes : `(c(1, 2))`, et construit la partie du haut deux fois plus haute que celle du bas : `heights = c(2, 1).` Les deux graphiques sont juxtaposés de façon que les échelles des abscisses correspondent.

On visualise ainsi, grâce à un programme simple, deux aspects différents de la distribution des valeurs de k.

```
# Cette fonction calcule, pour un intervalle de valeurs de
# U0, les durées de vol, k. Sur les graphiques de k en
# fonction de U0 on constate que des valeurs de k se répètent.
# D'où l'idée de déterminer la distribution des valeurs de k,
# pour un intervalle de valeurs de U0 donné et d'en faire un
# diagramme en barre...
syr = function(U0Inf = 1, U0Sup = 100){
   vectn <- NULL
   for (i in U0Inf:U0Sup) {
      k <- 0 ; u <- i
      while (u != 1) {
         if (u %% 2 == 0) {u <- u / 2} else {u <- 3 * u + 1}
         k <- k + 1
      }
      vectn <- c(vectn, k)
   }
   effec <- table(vectn)
# *********Affichage des résultats et des graphiques ****
   layout(c(1, 2), heights = c(2, 1))
   plot(vectn, U0Inf:U0Sup,
      main = paste("Durée k du vol en fonction de U0", "de\n",
         U0Inf, "à", U0Sup),
      ylab = "valeurs de U0",
      xlab = "Durée k du vol",
      pch = ".", cex = 3)
   plot(effec,
      main = paste("Diagramme en barre de la distribution",
        "des valeurs de k", "\npour des valeurs de U0 entre",
         U0Inf, "et", U0Sup),
      xlab = "Durée k du vol",
      ylab = "Effectifs")
}
```

```
Syr(1, 2000)
```



C – APPROXIMATION NUMÉRIQUE D'UNE INTÉGRALE PAR LA MÉTHODE DES RECTANGLES

Je suis parti de l'activité d'approche de l'Hyperbole (Nathan) Terminale S 2012 page 199, dans laquelle il s'agit de calculer une valeur approchée de $\int_0^1 e^t dt$ par encadrement par la méthode des rectangles. Les sommes Sn et sn sont préprogrammées sur GeoGebra, mais comme l'algorithme n'est pas accessible, on en reste au niveau d'une simple illustration.

C'est l'occasion de mettre en œuvre plusieurs algorithmes : **1°** Une boucle pour le calcul de Sn et sn avec la fonction exponentielle et d'autres fonctions continues positives strictement croissantes sur [0 ; 1] ; **2°** On remplace la boucle par des listes obtenues avec la fonction **`seq(...)`** ; **3°** Deux algorithmes généralisant le calcul de Sn et sn à un intervalle [a ; b], dans le cas d'une fonction strictement croissante et d'une fonction strictement décroissante. Une illustration graphique est proposée dans chaque programme.

1° Une boucle pour le calcul de Sn et sn avec une fonction continue positive, strictement croissante

`5L` force le type entier. **`Grand_S`** contient $\sum_{i=0}^{n-1} f(\frac{i}{n})\times(\frac{1}{n})$ et **`Petit_s`** contient $\sum_{i=0}^{n-1} f(\frac{i+1}{n})\times(\frac{1}{n})$.

La version pour les fonctions strictement décroissantes figure dans le fichier **"R_Riemann.R"**.

```
# Définition de la fonction utilisée
f <- function(x){exp(x)}

# VERSION EN LIGNES DE COMMANDES
# Sur un intervalle [0 ; 1] où f est strictement
# CROISSANTE. On peut choisir la valeur de n
n <- 5L
Grand_S <- 0 ; Petit_s <- 0
for (i in 0:(n - 1)){
   Grand_S <- Grand_S + f(i / n) * 1 / n
   Petit_s <- Petit_s + f((i + 1) / n) * 1 / n
}
cat("Grand_S", Grand_S, "\nPetit_s", Petit_s, "\n\n")

Grand_S 1.552177
Petit_s 1.895834
```

```
Avec :
n <- 50L
...
...

Grand_S 1.701156
Petit_s 1.735522
```

2° Un version plus compacte (sans boucle interprétée) du même algorithme.

`Seq(...)/n` créé les suites des subdivisions de x, **`sum(...)`** fait la somme des aires des rectangles correspondants. L'avantage du format compact des commandes est qu'elles sont plus rapides et simples à programmer et qu'elles s'exécutent plus rapidement Le passage d'une forme à l'autre ne coule pas de source, elle nécessite donc d'être détaillée. La version pour les fonctions strictement décroissantes figure dans le fichier **"R_Riemann.R"**.

```
# Définition de la fonction utilisée
f <- function(x){exp(x)}

# VERSION EN LIGNES DE COMMANDES
# Sur un intervalle [0 ; 1] où f est strictement
# CROISSANTE. On peut choisir la valeur de n
n <- 5L
subGrand_S <- seq(0, n - 1, 1) / n
subPetit_s <- seq(1, n, 1) / n

Grand_S <- sum(f(subGrand_S) * 1 / n)
Petit_s <- sum(f(subPetit_s) * 1 / n)
cat("Grand_S", Grand_S, "\nPetit_s", Petit_s, "\n\n")

Grand_S 1.552177
Petit_s 1.895834
```

```
Avec :
n <- 50L
...
...

Grand_S 1.701156
Petit_s 1.735522
```

3° Généralisation à une fonction continue positive, strictement croissante sur un intervalle [a ; b]

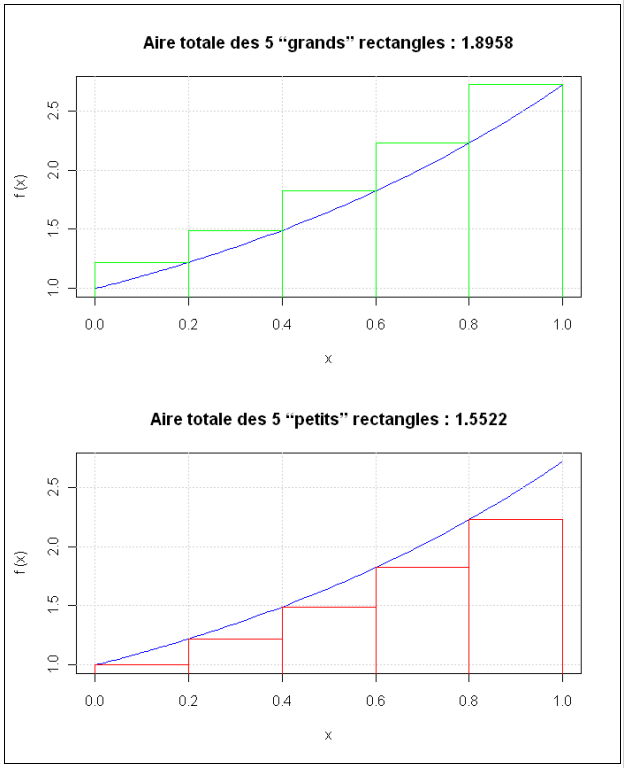
C'est la commande **`lines(...)`** incluse dans des boucles qui trace les segments de droite constituant les rectangles. Les paramètres de la fonction sont les bornes a et b de l'intervalle et le nombre n de subdivisions.

La version pour les fonctions strictement décroissantes figure dans le fichier **"R_Riemann.R"**.

Y figure aussi une version simplifiée **"`riemannC0_1(...)`"** opérant sur l'intervalle [0 ; 1], plus accessible pour un exercice avec les élèves.

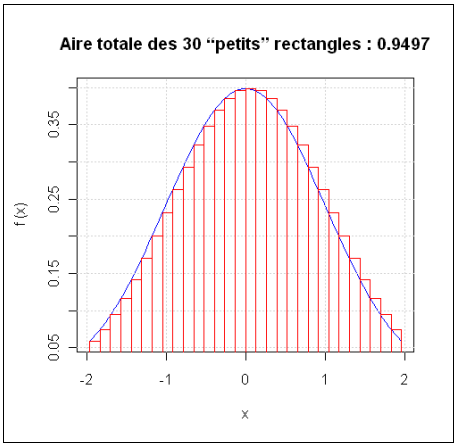
```
# Définition de la fonction utilisée
f <- function(x){exp(x)}

# VERSION FONCTION
# Sur un intervalle [a ; b] où f est strictement
# CROISSANTE. On peut choisir la valeur de n
riemannC <- function(a =0, b = 1, n = 5){
   sub <- (b - a) / n
   Grand_S <- 0 ; Petit_s <- 0
   for (i in 0:(n - 1)){
      Grand_S <- Grand_S + f(a + (i + 1) * sub) * sub
      Petit_s <- Petit_s + f(a + i * sub) * sub
   }
   cat("Grand_S", Grand_S, "\nPetit_s", Petit_s,
       "\nS - s =", Grand_S - Petit_s, "\n\n")
   # Les représentations graphiques
   par(mfrow = c(2, 1))
   # Les rectangles Grand_S
   plot(f, a, b, col = "blue",
      main = paste("Aire totale des", n,
         ""grands" rectangles :", round(Grand_S, 4)))
   grid()
   for (i in 0:(n - 1)){
      lines(c(a + i * sub, a + i * sub,
              a + (i + 1) * sub, a + (i + 1) * sub),
            c(0, f(a + (i + 1) * sub),
              f(a + (i + 1) * sub), 0),
              col = "green")
   }
   # Les rectangles Petit_s
   plot(f, a, b, col = "blue",
      main = paste("Aire totale des", n,
         ""petits" rectangles :", round(Petit_s, 4)))
   grid()
   for (i in 0:(n - 1)){
      lines(c(a + i * sub, a + i * sub,
              a + (i + 1) * sub, a + (i + 1) * sub),
            c(0, f(a + i * sub), f(a + i * sub), 0),
              col = "red")
   }
}
```

```
riemannC(0, 1, 5)
Grand_S 1.895834
Petit_s 1.552177
S - s = 0.3436564
```



On peut mener une réflexion sur la possibilité d'utiliser ces programmes dans le cas de fonction continues positives mais non monotones. Les illustrations issues de quelques essais permettront de poser les conditions de validation d'une telle utilisation :

```
f <- function(x){exp(-(x*x)/2) / sqrt(2 * pi)}
riemannC(-1.96, 1.96, 30)
Grand_S 0.9496783
Petit_s 0.9496783
S - s = -1.110223e-16
```



V – UN EXEMPLE DE GÉOMÉTRIE REPÉRÉE

A – CONSTRUIRE DES POLYGONES RÉGULIERS

Je suis parti d'un thème d'approche algorithmique et géométrie glané dans l'hyperbole de première S (2011, page 252), dans lequel il s'agit de programmer la construction d'un polygone régulier à p = 3 côtés. Un programme Algobox de 35 lignes (!) est proposé au décryptage.

Le polygone est tracé en reliant, par des segments, ses sommets repérés par leurs coordonnées polaires, dans un repère judicieusement choisi.

Nous allons voir comment les capacités graphiques de **R** rendent l'exercice facile et attrayant : 6 lignes suffisent, avec le “centre” et le “rayon”du polygone paramétrables.

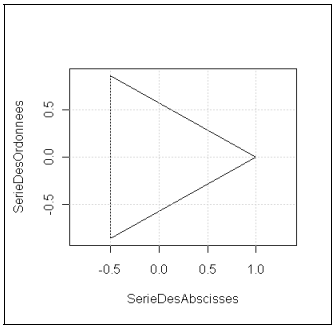
```
1 # LA VERSION EN LIGNES DE COMMANDES
2 p = 5 ; xC = 0 ; yC = 0 ; r = 1
3 SerieDesAngles <- seq(from = 0, to = 2 * pi, by = 2 * pi / p)
4 SerieDesAbscisses <- xC + r * cos(SerieDesAngles)
5 SerieDesOrdonnees <- yC + r * sin(SerieDesAngles)
6 plot(SerieDesAbscisses, SerieDesOrdonnees, type = "l", asp = 1)
7    grid()

# LA VERSION FONCTION
PolygRegu <- function(p = 5, xC = 0, yC = 0, r = 1){
   SerieDesAngles <- seq(from = 0, to = 2 * pi, by = 2 * pi / p)
   SerieDesAbscisses <- xC + r * cos(SerieDesAngles)
   SerieDesOrdonnees <- yC + r * sin(SerieDesAngles)
   plot(SerieDesAbscisses, SerieDesOrdonnees, type = "l", asp = 1)
      grid()
}

# LA VERSION FONCTION AVEC ROTATION D'ANGLE theta
PolygReguR <- function(p = 3, xC = 0, yC = 0, r = 1, theta = pi / 2){
   SerieDesAngles <- seq(0, 2 * pi, 2 * pi / p)
   SerieDesAbscisses <- xC + r * cos(SerieDesAngles + theta)
   SerieDesOrdonnees <- yC + r * sin(SerieDesAngles + theta)
   plot(SerieDesAbscisses, SerieDesOrdonnees, type = "l", asp = 1)
      grid()
}
```

Si l'on exécute PolygRegu() on obtiendra un polygone à p = 5 côtés, "centré" en (xC = 0, yC = 0), de "rayon" de construction r = 1. Si l'on veut un polygone à 7 côtés, on peut exécuter PolygRegu(p = 7) ou bien PolygRegu(7). Si c'est le "rayon" seul que l'on veut modifier on exécute PolygRegu(r = 3). Si l'on veut modifier p et r on exécute PolygRegu(p = 9, r = 2) ou bien PolygRegu(9,,,2).

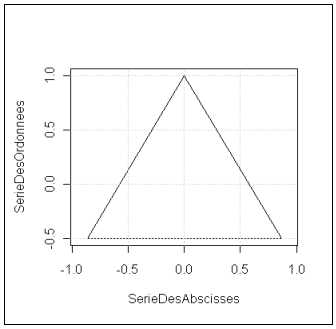
Il est intéressant d'observer ce qui se passe lorsque n devient grand.

Les numéros de ligne ne font pas partie du programme. Les lignes de commentaires commence par un #.
Ligne 2 : Figurent les initialisations de paramètres. Quand il y a plusieurs commandes sur une même ligne elles sont séparées par des point-virgules.
<- est l'opérateur d'affectation (on peut aussi utiliser -> ou =). Dans Ligne 3 : seq(...) créé une suite de p valeurs de 0 à 2×pi, de 2×pi/p en 2×pi/p.
Ligne 4 : Calcule la série des p valeurs des abscisses des sommets du polygone.
Ligne 5 : Calcule la série des p valeurs des ordonnées des sommets du polygone.
Ligne 6 : Trace les segments (type = "l") reliant les p sommets. asp = 1 sert à réaliser un repère orthonormé
Ligne 7 : Trace le quadrillage sur le graphique fait par plot.
Dans le code d'une fonction, l'affectation des valeurs par défaut pour les paramètres se fait par l'opérateur = (et non <-) : (p = 5, xC = 0 ...)

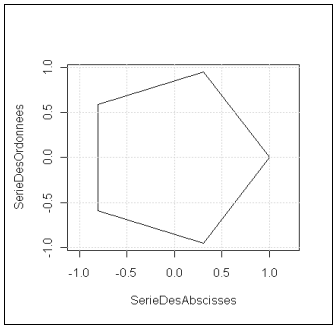
PolygReguR()



PolygReguR(theta = 0)



PolygReguR(p = 5, theta = 0)



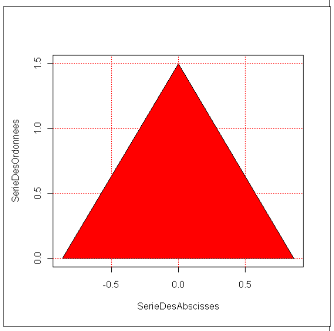
B – DES POLYGONES AUX TRIANGLES DE SIERPINSKI

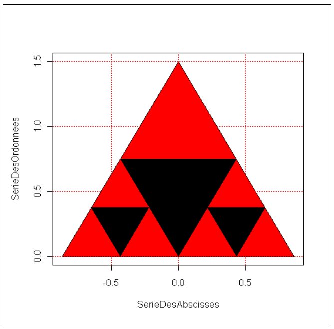
Les triangles équilatéraux sont des cas particuliers de polygones réguliers. Ça m'a fait penser aux triangles de Sierpinski. Je me suis donc demandé si l'on pouvait utiliser les algorithmes précédents pour construire des triangles de Sierpinsky.

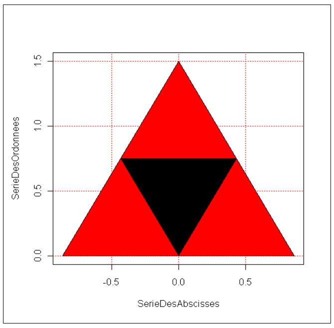
1° J'ai d'abord fait quelques essais en ligne de commande pour repérer les "répétitions". **R** possède deux types de fonctions graphiques, celles de "haut niveau" (high level) qui génèrent des graphiques complets et celles de "bas niveau" (low level) qui complètent un graphique existant, créé par une fonction de "haut niveau". `polygon(...)` étant une fonction graphique de "bas niveau" il faut créer un graphique avec `plot(...)` sur lequel se superposera le polygone. `type = "n"` effectue un tracé "vide".

J'ai superposé des triangles noirs sur le triangle initial rouge. On peut facilement faire des jeux de couleur.

```
# LE PREMIER TRIANGLE
# CAS PARTICULIER DE POLYGONE : LES TRIANGLES DE SIERPINSKI
Sierp1 <- function(xC = 0, yC = 0, r = 1, theta = pi / 2){
   SerieDesAngles <- seq(0, 2 * pi, 2 * pi / 3)
   SerieDesAbscisses <- xC + r * cos(SerieDesAngles + theta)
   SerieDesOrdonnees <- yC + r * sin(SerieDesAngles + theta)
   plot(SerieDesAbscisses, SerieDesOrdonnees, type = "n", asp = 1)
      polygon(SerieDesAbscisses, SerieDesOrdonnees, col = "red")
      grid(col = "red")
}

# POUR SUPERPOSER DES TRIANGLES
SierpN <- function(xC = 0, yC = 0, r = .5, theta = -pi / 2){
   SerieDesAngles <- seq(0, 2 * pi, 2 * pi / 3)
   SerieDesAbscisses <- xC + r * cos(SerieDesAngles + theta)
   SerieDesOrdonnees <- yC + r * sin(SerieDesAngles + theta)
      polygon(SerieDesAbscisses, SerieDesOrdonnees, col = "black")
}

Sierp1(xC = 0, yC = 0, r = 1)
#--------------------
SierpN(xC = 0 * sqrt(3) / 2, yC = 0 / 2, r = 1 / 2)

SierpN(xC = -sqrt(3) / 4, yC = -1 / 4, r = 1 / 4)
SierpN(xC = sqrt(3) / 4, yC = -1 / 4, r = 1 / 4)

SierpN(xC = 0 * sqrt(3) / 4, yC = 2 / 4, r = 1 / 4)

#--------------------
```

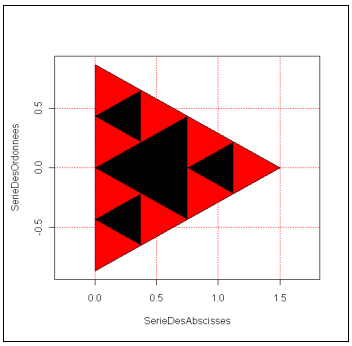







```
#--------------------
```

```
#---AVEC ROTATION-----------------
Sierp1(theta = 0, xC = 1 / 2, yC = 0)

SierpN(theta = pi, r = 1 / 2, xC = 1 / 2, yC = 0)

SierpN(theta = pi, r = 1 / 4, xC = 1 / 4, yC = - 1 * sqrt(3) / 4)
SierpN(theta = pi, r = 1 / 4, xC = 1 / 4, yC = 1 * sqrt(3) / 4)
SierpN(theta = pi, r = 1 / 4, xC = 4 / 4, yC = 0 * sqrt(3) / 4)
```

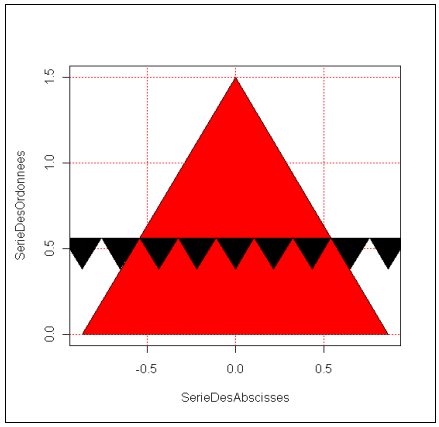


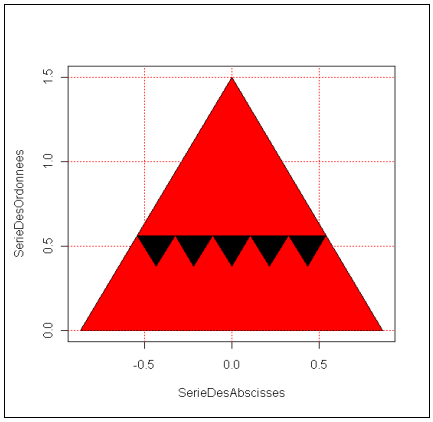
Il ne reste plus qu'à trouver le bon algorithme pour faire tout cela automathiquement !!!

2° Pour une première approche, j'ai simplifié le problème en superposant des rangées de triangles noirs. Cela évite de gérer les superpositions multiples, noir sur noir faisant toujours noir.

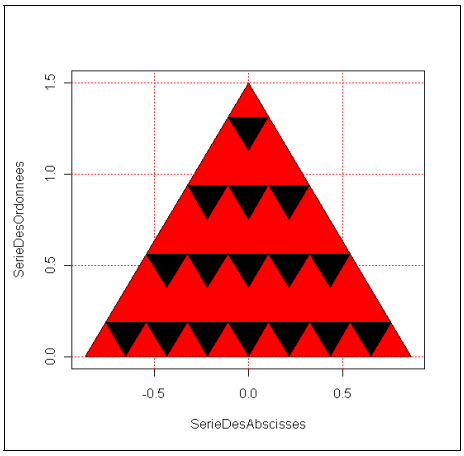
J'ai détaillé quelques étapes de la recherche qui m'a amené à l'algorithme final.

```
#  CAS PARTICULIER DES TRIANGLES DE SIERPINSKI
Sierp1 <- (xC = 0, yC = .5, r = 1, theta = pi / 2){
   SerieDesAngles <- seq(0, 2 * pi, 2 * pi / 3)
   SerieDesAbscisses <- xC + r * cos(SerieDesAngles + theta)
   SerieDesOrdonnees <- yC + r * sin(SerieDesAngles + theta)
   plot(SerieDesAbscisses, SerieDesOrdonnees, type = "n", asp = 1)
      polygon(SerieDesAbscisses, SerieDesOrdonnees, col = "red")
      grid(col = "red")
}

# POUR SUPERPOSER DES TRIANGLES
SierpN <- function(xC = 0, yC = .5, r = .5, theta = -pi / 2){
   SerieDesAngles <- seq(0, 2 * pi, 2 * pi / 3)
   SerieDesAbscisses <- xC + r * cos(SerieDesAngles + theta)
   SerieDesOrdonnees <- yC + r * sin(SerieDesAngles + theta)
      polygon(SerieDesAbscisses, SerieDesOrdonnees,
      col = "black", border = NA)
}

#-----------------------------------------------------------
#--UNE BOUCLE POUR LES TRIANGLES D'UNE RANGÉE D'UN NIVEAU
# UN NIVEAU C'EST UNE TAILLE DE TRIANGLE
#-----------------------------------------------------------
n <- 3
k <- 4 / 2^n
Sierp1(xC = 0, yC = .5, r = 1)
for (i in (-2^(n - 1)):(2^(n - 1))) {
   SierpN(i / 2^n * sqrt(3), k, 1 / 2^n)
}

# Il y a des triangles qui "sortent" du grand rouge ...
# On se limite aux triangles de l'intérieur du grand rouge.
#---------------------------------------
n <- 3 ; r <- 1
k <- (1 + r * 3) / 2^n
Sierp1(xC = 0, yC = .5, r = 1)
for (i in (-2^(n - 1) + r + 1):(2^(n - 1) - r - 1)) {
   SierpN(i / 2^n * sqrt(3), k, 1 / 2^n)
}
```

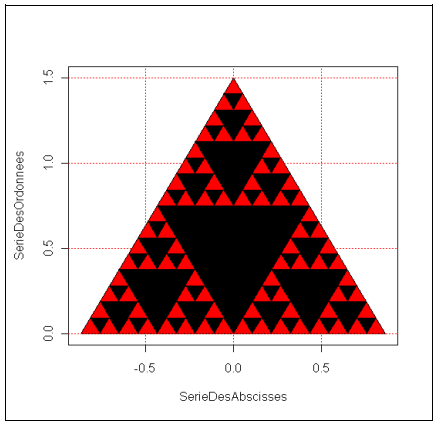




```
#-------------------------------------------------------------
#--DEUX BOUCLES POUR LES TRIANGLES DE TOUTES LES RANGÉES D'UN
# NIVEAU EN RESTANT À L'INTÉRIEUR DU GRAND ROUGE
#-------------------------------------------------------------
n <- 3
Sierp1(xC = 0, yC = .5, r = 1)
h <- seq(1 / 2^n, 3 / 2, 3 / 2^n)
for (k in h) {
   r <- which(h == k)
   for (i in (-2^(n - 1) + r):(2^(n - 1) - r)) {
      SierpN(i / 2^n * sqrt(3), k, 1 / 2^n)
   }
}

#-------------------------------------------------------------
# TROIS BOUCLES POUR LES TRIANGLES DE TOUTES LES RANGÉES DE
# TOUS LES NIVEAUX ---- UNE FONCTION -----
#-------------------------------------------------------------
sierpinski <- function(n = 3){
   if (n == 0) {
      Sierp1(xC = 0, yC = .5, r = 1)
   } else {
      Sierp1(xC = 0, yC = .5, r = 1)
      for (j in 1:n) {
         h <- seq(1 / 2^j, 3 / 2, 3 / 2^j)
         for (k in h) {
            r <- which(h == k)
            for (i in (-2^(j - 1) + r):(2^(j - 1) - r)) {
               SierpN(i / 2^j * sqrt(3), k, 1 / 2^j)
            }
         }
      }
   }
}
```



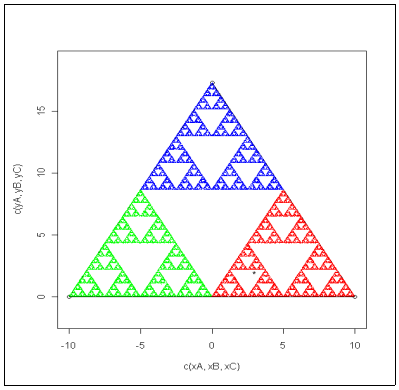
sierpinski(4)



Au lieu de charger les fonctions **`Sierp1(…)`** et **`SierpN(…)`** à partir de l'éditeur de code, on peut les appeler directement dans la fonction **`Sierpinski(…)`** en appelant le nom de fichier dans lequel elles sont enregistrées. La commande est : **`source("Unité:/CheminDuDossier/NomFichier.R")`**.

3° Voici un algorithme complètement différent, qui construit des triangles de Sierpinsky à partir d'une procédure faisant intervenir des nombres au hasard.

```
# Tracer le triangle A(-10;0), B(10;0), C(0;10v3). Placer un point M0(x;y)
# quelconque dans ce triangle. Pour placer les points Mi suivants :
# Tirer un nombre entier a au hasard entre 1 et 3. Si  a=1 le point Mi
# sera le milieu du segment AMi-1, si a=2 le point Mi sera le milieu du
# segment BMi-1, si a=2 le point Mi sera le milieu du segment CMi-1, est
# ainsi de suite à l'infini ...
SierpAlea <- function(nbsim = 50000, xM = 3, yM = 2){
   xA <- -10 ; yA <- 0 ; xB <- 10 ; yB <- 0 ; xC <- 0 ; yC <- 10 * sqrt(3)
   plot(c(xA, xB, xC), c(yA, yB, yC), asp = 1)
      lines(c(xA, xB, xC, xA), c(yA, yB, yC, yA))
      points(xM, yM, pch = "*") ; x <- xM ; y <- yM
      for (i in 1:nbsim) {
         a <- sample(1:3, 1)
         if (a == 1) {
            x <- (xA + x) / 2 ; y <- (yA + y) / 2
            points(x, y, pch = ".", col = "green")
         } else {
            if (a == 2) {
               x <- (xB + x) / 2 ; y <- (yB + y) / 2
               points(x, y, pch = ".", col = "red")
            } else {
               x <- (xC + x) / 2 ; y <- (yC + y) / 2
               points(x, y, pch = ".", col = "blue")
            }
         }
      }
}
```



VI – ALGORITHMES DE SIMULATION ET DE CALCULS EN PROBABILITÉ ET STATISTIQUE

A – PARTICULARITÉS DE L'ALGORITHMIQUE EN PROBABILITÉ ET STATISTIQUE

Les probabilités et la statistique constituent un terrain privilégié de mise en œuvre de l'algorithmique des programmes du lycée, tant pour le calcul et l'illustration des distributions que pour la pratique des simulations.

Les notions de distribution de fréquences et de probabilité constituent les matériaux de base de la statistique. À partir de quelques exemples nous verrons comment les outils spécifiques de **R** permettent de travailler avec des distributions variées et comment on peut faire de la simulation en probabilité un véritable outil de résolution* de problèmes, * signifiant qu'elle permet de trouver des solutions asymptotiquement exactes (en vertu de la loi des grands nombres qui assure la convergence des fréquences vers les probabilités).

Les outils spécifiques de **R** permettent de simuler facilement des situations concrètes très variées, ils permettent de décrire, de résumer et d'illustrer graphiquement les séries simulées, de les comparer aux distributions de probabilités que l'on prend comme modèle, de construire des algorithmes prenant en compte successivement, tous ces aspects du traitement d'une expérience aléatoire.

La pratique de la simulation en classe permet aussi de surmonter des obstacles didactiques rencontrés dans l'enseignement des probabilités, et plus particulièrement dans son approche fréquentiste, comme le montre bien l'article[10] de Michel Henry dans cette même revue en ligne. C'est d'ailleurs certains de ses exemples que j'ai repris pour les illustrer en **R**.

B – PROBLÈME HISTORIQUE D'UN DUC DE TOSCANE VERSION TRICHEUR

1° La simulation permet de surmonter l'obstacle didactique posé par la détermination de la distribution de probabilité de la variable aléatoire somme des valeurs associées au faces obtenues en jetant trois fois un dé.

Afin de passer en revue les principales caractéristiques des fonctions **R** concernant la simulation et le traitement des données simulées, j'ai choisi de faire tricher le duc de Toscane en lui faisant utiliser un dé pipé, dont la probabilité d'obtention de chaque face est proportionnelle au nombre qu'elle porte. Il est bien entendu que ce choix n'est pas pédagogique. On peut facilement transformer les fonctions proposées pour qu'elles simulent un dé équilibré.

Quelques explications pour l'algorithme de la simulation et du calcul de la distribution simulée :

`de <- 1:6` : `de` prend pour valeur la série des entiers de 1 à 6. `ProbaDe <- de / sum(de)` : `ProbaDe` prend pour valeurs la distribution de probabilité sur le dé pipé (1/21, 2/21, …, 6/21).

`jeu <- sample(de, 3, replace = TRUE, prob = ProbaDe)` : `sample` fait un tirage aléatoire, avec remise, dans `de`, avec une distribution de probabilité `ProbaDe` et met les 3 entiers obtenus dans le vecteur `jeu`. `s <- sum(jeu)` en fait la somme.

`serieSomNbjets <- c(serieSomNbjets, s)` construit la série des valeurs des sommes simulées.

`tabloFreqS <- table(serieSomNbjets) / nbsim` construit le tableau des fréquences de la série simulée et l'affecte à la table `tabloFreqS`. Une table ainsi construite est constitué des noms (names) des valeurs, “au dessus” des fréquences correspondantes. Les noms sont des chaines de caractères, c'est pour cela que l'on fait `as.numeric(names(tabloFreqS))` pour récupérer les valeurs de la variable.

Quelques explications pour l'algorithme de calcul de la distribution de probabilité :

La stratégie consiste à créer un dé à 21 faces dont une numéroté 1, deux numérotées 2, …, six numérotées 6 : `dai <- rep(1:6, 1:6)`.

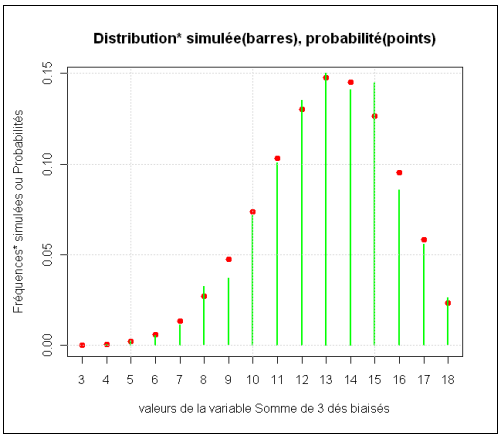
`s <- array(0, dim = c(21, 21, 21))` crée un tableau à trois dimensions, chacune de taille 21 dans lequel seront mises les 21^3 valeurs des sommes des faces obtenues par trois boucles imbriquées.
`table(s) / 21^3` construit le tableau de la distribution de probabilité de la variable aléatoire S.

[10]Simulations d'expériences aléatoires en classe. Un enjeu didactique pour comprendre la notion de modèle probabiliste, un outil de résolution de problèmes. (http://revue.sesamath.net/spip.php?article353)

Les deux fonctions **ToscSimul3Biais** et **ToscProba3Biais** doivent être chargées en mémoire pour utiliser la fonction **Toscane3Biais**.

```
# DISTRIBUTION DES FRÉQUENCES SIMULÉES DE LA VARIABLE S
ToscSimul3Biais <- function(nbsim){
   de <- 1:6 ; ProbaDe <- de / sum(de)
   serieSomNbjets <- NULL
   for (i in 1:nbsim) {
      jeu <- sample(de, 3, replace = TRUE, prob = ProbaDe)
      s <- sum(jeu)
   serieSomNbjets <- c(serieSomNbjets, s)
   }
   tabloFreqS <- table(serieSomNbjets) / nbsim
   return(tabloFreqS)
}
# DISTRIBUTION DE PROBABILITÉ DE LA VARIABLE S
ToscProba3Biais <- function(){
   dai <- rep(1:6, 1:6)
   s <- array(0, dim = c(21, 21, 21))
   for (i in 1:21){
      for (j in 1:21){
         for (k in 1:21){s[i, j, k] <- dai[i] + dai[j] + dai[k]}
      }
   }
   tabloProbaS <- table(s) / 21^3
   return(tabloProbaS)
}
# SUPERPOSITION DES DISTRIBUTIONS DE PROBABILITÉ ET
# DES FRÉQUENCES SIMULÉES
Toscane3Biais <- function(nbsim = 2000){
   DistribProbaS <- ToscProba3Biais()
   DistribFreqS <- ToscSimul3Biais(nbsim = nbsim)
   EsperanceS <- sum(as.numeric(names(DistribProbaS)) * DistribProbaS)
   MoyenneS <- sum(as.numeric(names(DistribFreqS)) * DistribFreqS)
   # Affichage des résultats et des graphiques
   cat("Tableau des fréquences* simulées de la variable S :\n")
   print(DistribFreqS)
   cat("\nEspérance de la variable S :", EsperanceS, "\n\n")
   cat("Moyenne de la distrib. simulée de S :", MoyenneS, "\n\n")
   plot(DistribProbaS, type = "p", pch = 19, col = "red",
         xlab = "valeurs de la variable Somme de 3 dés biaisés",
         ylab = "Fréquences* simulées ou Probabilités",
         main = "Distribution* simulée(barres), probabilité(points)")
      points(DistribFreqS, type = "h", col = "green")
      grid()
}
```

```
> ToscaneBiais(2000)
Tableau des fréquences* simulées de la
    variable S :
serieSomNbjets
     4      5      6      7      8      9
0.0005 0.0025 0.0055 0.0115 0.0325 0.0370
    10     11     12     13     14     15
0.0715 0.1005 0.1350 0.1500 0.1410 0.1445
    16     17     18
0.0855 0.0560 0.0265

Espérance de la variable S : 13

Moyenne de la distribution simulée de S :
    13.05
```



La fonction **plot** représente la distribution de probabilité par des points pleins de couleur rouge : (**type = "p", pch = 19, col = "red"**).

point y superpose un diagramme en barres vertes (**type = "h", col = "green"**) représentant les fréquences des valeurs simulées de la variable S. **grid()** y superpose le quadrillage.

Le fichier **"ToscaneBiaiS3d.r"** contient aussi le code de deux fonctions permettant, séparément, de faire et d'illustrer la simulation et de calculer et d'illustrer la distribution de probabilité de S.

La simulation permet de choisir un nombre de jets variable. Par contre pour mettre le nombre de jets en paramètre dans le calcul de la distribution de probabilité, il faut un algorithme un peu plus compliqué. Avis aux amateurs !

La simplicité de l'algorithme permis par les fonctions spécialisées de **R** facilite la lecture et la compréhension des stratégies de simulation, ce qui constitue un avantage indéniable par rapport aux simulations faites avec un tableur.

C – INTERVALLE DE FLUCTUATION SIMULÉ -- INTERVALLE DE FLUCTUATION CALCULÉ ET PROLONGEMENT

L'intervalle de fluctuation binomial est au cœur du programme de probabilité de première S. C'est un outil privilégié pour prendre des décisions en situation d'incertitude. C'est le début de l'initiation à l'inférence statistique au lycée.

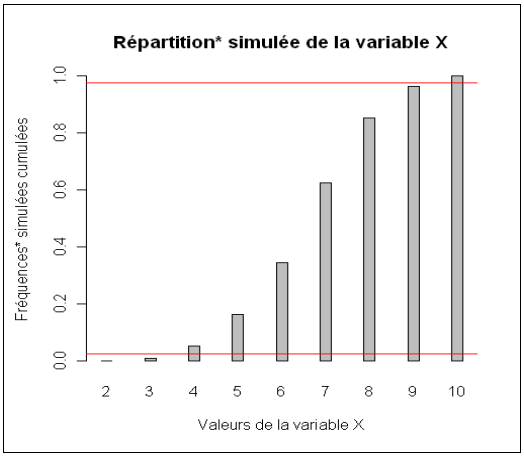
Nous allons voir comment **R** permet d'obtenir un intervalle de fluctuation par simulation (d'ailleurs, quelle que soit la distribution de la variable dans la population parent) puis comment il permet de mettre en œuvre un algorithme qui est l'application directe de la définition de l'IF figurant dans le document ressource du programme de première S. Nous terminerons par la construction de la courbe d'efficacité d'une règle de décision basée sur un IF binomial.

1° Intervalle de fluctuation simulé

- On simule **nbsim** = 2000 échantillons (tirages) avec remise, de taille **n** = 10 dans une urne de **nbtot** = 100 boules dont **nbrouges** = 70 rouges. **X** est la variable aléatoire prenant pour valeur le nombre de boules rouges obtenue. On obtient ainsi 2000 valeurs de **X** dans la liste **serieX**.
- L'intervalle de fluctuation bilatéral (au sens du programme de première S actuel) simulé, de probabilité 0,95, de la variable **X** est déterminé par les deux **Q**uantiles à 2,5 % et 97,5 % de la série **serieX**. On notera $IF^*_{0,95}$ de X = $[Q_{2,5\%} ; Q_{97,5\%}]$ et $IF^*_{0,95}$ de **X / n** = $[Q_{2,5\%} / n ; Q_{97,5\%} / n]$
- Un premier enjeu didactique est de montrer que l'on peut établir un intervalle de fluctuation simulé sans se servir du modèle binomial. On a simplement utilisé le **modèle équiprobable** généré par la fonction **sample()**.

```
# ALGO ---Intervalle de fluctuation bilatéral SIMULÉ d'une
# variable X nombre de boules rouges dans un échantillon de n la
# et de variable fréquence X/n, de probabilité minimale 1 - e
# Le principe est de simuler nbsim répétitions du tirage d'un
# échantillon de taille n
# On obtient une série statistique de nbsim valeurs.
# Les valeurs simulées de a et b seront les quantiles d'ordre
# e/2 et - e/2 de cette série.
#----------------Version de base simulIF_1()
IF_Simul = function(n = 10, nbrouges = 70, nbtot = 100,
                    e = .05, nbsim = 2000){
   urne <- rep(c("rouge", "autre"), c(nbrouges, nbtot - nbrouges))
   serieX <- NULL
   for (i in 1:nbsim) {
      tirage <- sample(urne, n, replace = TRUE)
      x <- sum(tirage == "rouge")
      serieX <- c(serieX, x)
   }
   tablFreqX <- table(serieX) / nbsim
   tablFreqCumX <- cumsum(tablFreqX)
   quantserieX <- quantile(serieX, probs = c(e / 2, 1 - e / 2),
                           type = 2)
   propIF_sim <- sum(serieX >= quantserieX[1] &
                     serieX <= quantserieX[2]) / nbsim
  # Affichage des résultats
   barplot(tablFreqCumX, space = 3,
         xlab = "Valeurs de la variable X",
         ylab = "Fréquences* simulées cumulées",
         main = "Répartition* simulée de X")
      abline(h = c(e / 2, 1 - e / 2), col = "red")
   cat("Distribution simulée de la variable X\n")
   print(tablFreqX)
   cat("\nQuantiles série X\n")
   print(quantserieX)
   cat("\nQuantiles série X / n\n")
   print(quantserieX / n)
   cat("\nPourcentage de valeurs de la série comprises dans l'IF :",
       propIF_sim, "\n\n")
}
```

```
> IF_Simul()
Distribution* simulée de la variable X
serieX
     2      3      4      5      6
0.0025 0.0090 0.0420 0.1095 0.1820
     7      8      9     10
0.2795 0.2285 0.1100 0.0370

Quantiles série X
 2.5% 97.5%
    4    10

Quantiles série X / n
 2.5% 97.5%
  0.4   1.0

Pourcentage de valeurs de la série comprises
dans l'IF : 0.9885
```



- Un deuxième enjeu didactique est de réinvestir les quantiles, outils de statistique descriptive, objet de nombreuses “controverses” quant à leurs calculs, que l'on a signalé dans le chapitre II note 7, page 8.

2° Intervalle de fluctuation calculé

Le document ressource de première S propose deux modes de calcul d'un IF binomial dont le suivant :

L'intervalle de fluctuation (**IF**) de probabilité 1 – e, d'une variable aléatoire *X* de loi binomiale de paramètres n et p, est l'intervalle [a ; b] défini par :

- ♦ *a* est le plus petit entier tel que $P(X \leq a) > e / 2$;
- ♦ *b* est le plus petit entier tel que $P(X \leq b) \geq 1 - e / 2$.

`repartX <- pbinom(0:n, n, p)` place la répartition binomiale dans la liste `repartX`.
`which(repartX > e / 2)` renvoie la liste des indices des valeurs de la liste supérieures à `e / 2`.
`min` en renvoie la plus petite valeur. Il y a un décalage de 1 entre indices de la liste et valeurs de la variable.

L'algorithme est ainsi l'application direct du mode de calcul proposé dans le document d'accompagnement, ce qui est plus simple à mettre en œuvre que les algorithmes classiquement construits avec des boucles while.

```
#  Des opérations sur une liste indicée remplacent les
# boucles while
IF_CalculBino <- function(n = 10, p = .7, e = .05){
   repartX <- pbinom(0:n, n, p)
   rang_a <- min(which(repartX > e / 2))
   a <- rang_a - 1
   rang_b <- min(which(repartX >= 1 - e / 2))
   b <- rang_b - 1
   # Affichages
   if (a != 0) {
      cat("P(X <=", a - 1, ")=", repartX[rang_a - 1],
          "\n")
      cat("P(X <=", a, ")=", repartX[rang_a], "\n\n")
   } else {
      cat("P(X <=", a, ")=", repartX[rang_a], "\n\n")
   }
   cat("P(X <=", b - 1, ")=", repartX[rang_b - 1], "\n")
   cat("P(X <=", b, ")=", repartX[rang_b], "\n\n")
   cat("Avec une valeur nominale de probabilité de",
       1 - e, "\n")
   cat("L'intervalle de fluctuation de X est :[", a,
       ";", b, "]\n")
   cat("L'intervalle de fluctuation de X/n est :[",
       a / n,";", b / n, "]\n")
   cat("Sa probabilité réelle est de", sum(dbinom((a:b),
       n, p)), "\n\n")
}
```

```
> IF_CalculBino(n = 10, p = .7)
P(X <= 3 )= 0.01059208
P(X <= 4 )= 0.04734899

P(X <= 9 )= 0.9717525
P(X <= 10 )= 1

Avec une valeur nominale de probabilité de 0.95
L'intervalle de fluctuation de X est :[ 4 ; 10 ]
L'intervalle de fluctuation de X/n est :[ 0.4 ; 1 ]
Sa probabilité réelle est de 0.9894079

> IF_CalculBino(n = 1000, p = .7)
P(X <= 670 )= 0.02158184
P(X <= 671 )= 0.02533364

P(X <= 727 )= 0.9719601
P(X <= 728 )= 0.9761905

Avec une valeur nominale de probabilité de 0.95
L'intervalle de fluctuation de X est :[ 671 ; 728 ]
L'intervalle de fluctuation de X/n est :
[ 0.671 ; 0.728 ]
Sa probabilité réelle est de 0.9546087
```

3° Courbe d'efficacité de la règle de décision basée sur un IF binomial.

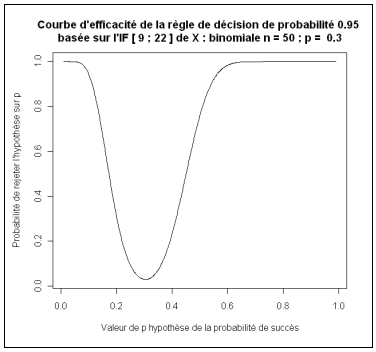
L'intervalle de fluctuation d'une variable fréquence X / n, au seuil de 1 – e (avec, en général, e = 5%), permet d'établir une règle de décision quant à une hypothèse sur une valeur p_0 du paramètre p d'une variable binomiale X : Si p_0 n'appartient pas à l'IF, on rejette l'hypothèse $p = p_0$, avec un risque (de première espèce) de se tromper (que p vaille p_0) d'au plus e. Si p_0 appartient à l'IF alors on dit que l'échantillon ne nous a pas permis de rejeter l'hypothèse $p = p_0$. Le risque que l'on prend à garder $p = p_0$ dépend alors de la "vraie" valeur de p.

Ce risque (de seconde espèce) peut être indirectement illustré par la courbe d'efficacité d'une règle de décision binomiale, ce que réalise la fonction **"`EfficaciteIF`"**.

Cette courbe d'efficacité représente la probabilité de rejeter l'hypothèse sur p en fonction des valeurs de p dans]0 ; 1[. La fonction **"`IF_Bino`"** version simplifiée de "`IF_CalculBino`" renvoie l'IF. L'algorithme est simple, par contre l'interprétation de la courbe est plus délicat, car selon les valeurs de p, on passe de la probabilité de rejeter l'hypothèse à raison (efficacité de la règle de décision) au risque de la rejeter à tors.

```
# Calcul et représentation graphique de la courbe d'efficacité
EfficaciteIF <- function(n = 100, p = .3, e = .05){
   IF <- IF_Bino(n, p, e) ; a <- IF[1] ; b <- IF[2]
   vectp <- seq(.01, .99, length.out = 1000) ; vecteffi <- NULL
   for (i in 1:1000) {
      ProbaRejet <- sum(dbinom(0:(a - 1), n, vectp[i])) +
                    sum(dbinom((b + 1):n, n, vectp[i]))
      vecteffi <- c(vecteffi, ProbaRejet)
   }
   # Affichage des résultats et graphiques
   plot(vectp, vecteffi, type = "l",
      xlab = "Valeur de p hypothèse de la probabilité de succès",
      ylab = "Probabilité de rejeter l'hypothèse sur p",
      main = paste("Courbe d'efficacité de la règle de décision de",
         "probabilité", 1 - e, "\nbasée sur l'IF [", a, ";", b,
         "] de X : binomiale n =", n, "; p = ",p))
   cat("Avec une valeur nominale de probabilité de", 1 - e, "\n")
   cat("L'intervalle de fluctuation de X est :[", a,";", b, "]\n")
   cat("L'intervalle de fluctuation de X/n est :[",
        a / n,";", b / n, "]\n")
   cat("Sa probabilité réelle est de", sum(dbinom((a:b), n, p)), "\n\n")
}
```

```
> EfficaciteIF(n = 50, p = .3)
Avec une valeur nominale de
   probabilité de 0.95
L'intervalle de fluctuation de X/n
   est :[ 0.18 ; 0.44 ]
Sa probabilité réelle est de
   0.9694705
```



D – HISTOIRE DE LA MARCHE ALÉATOIRE D'UNE PUCE, DÉCRITE SOUS PLUSIEURS ANGLES

Un puce se déplace aléatoirement par sauts, sur un axe gradué, en partant de l'origine. Elle se déplace vers la gauche ou vers la droite, d'une unité avec les proba 0,5 et 0,5, (p, 1-p). Il s'agit de simuler la coordonnée du point d'arrivée de marches aléatoires de 30 (nbSaut) sauts et de décrire les séries obtenues par simulation.

On est dans un cas (le plus souvent rencontré dans les exemples des manuels) où la simple simulation d'une valeur de la position finale de la puce n'a que peu d'intérêt car ne mettant en œuvre qu'une simple stratégie numérique.

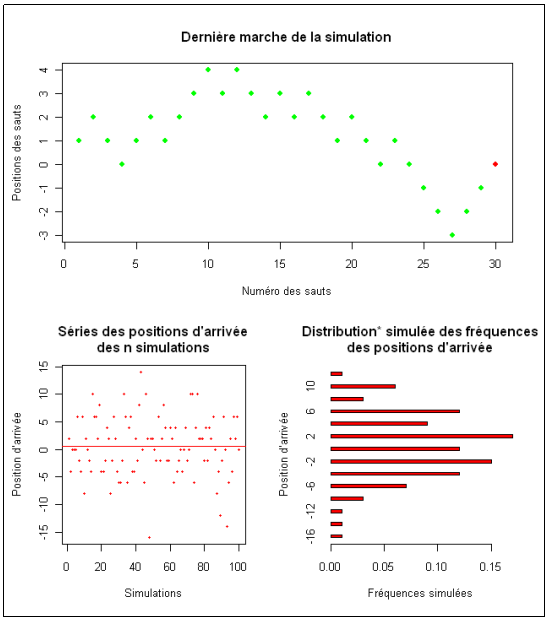
Or il est intéressant pour donner du sens à l'aspect aléatoire de la marche de la décrire sous différents aspects, en particulier d'illustrer le chemin parcouru, et de marquer la position d'arrivée sur ce chemin.

La simulation permet d'aborder l'aspect probabiliste de la marche par l'intermédiaire du graphique des positions d'arrivée et de la distribution de leurs fréquences. Ce sont autant d'outils graphiques différenciés à mettre en œuvre de façon pertinente.

Les trois graphiques sont positionnés dans une seule fenêtre graphique partagée en trois par la fonction `layout(partageFgraph)` où `partageFgraph <- matrix(c(1, 1, 2, 3), nrow = 2, byrow = TRUE)` est une matrice pilotant la disposition des graphiques dans la fenêtre. Le graphique 1 (le premier dans l'ordre de l'algorithme) occupera la partie du haut, le 2 (le suivant dans l'algorithme) occupera la partie en bas à gauche, le 3 occupera la partie en bas à droite.

```r
marcheB3 <- function(Nbsim = 100, nbSaut = 30, p = .5){
   serieArrivees <- NULL ; choix <- c("droite", "gauche")
   DerniereMarche <- NULL
   for (i in 1:Nbsim) {
      x <- 0
      for (j in 1:nbSaut) {
         direction <- sample(choix, 1, prob = c(p, 1 - p))
         if(direction == "gauche") {x <- x - 1} else {x <- x + 1}
         if (i == Nbsim) {DerniereMarche <- c(DerniereMarche, x)}
      }
      serieArrivees <- c(serieArrivees, x)
   }
   posimoy <- mean(serieArrivees)
   distFreqPosi <- table(serieArrivees) / Nbsim
   # Affichage des résultats et des graphiques
   cat("Moyenne des positions d'arrivée simulées\n")
   print(posimoy)
   partageFgraph <- matrix(c(1, 1, 2, 3), nrow = 2, byrow = TRUE)
   layout(partageFgraph)
   plot(DerniereMarche, pch = 19,
        col = rep(c("green", "red"), c(29, 1)),
        main = "Dernière marche de la simulation",
        xlab = "Numéro des sauts",
        ylab = "Positions des sauts")
   plot(1:Nbsim, serieArrivees, pch = 19, cex = .5, col = "red",
         main = "Séries des positions d'arrivée\ndes n
   simulations",
         xlab = "Simulations",
         ylab = "Position d'arrivée")
      abline(h = posimoy, col = "red")
   barplot(distFreqPosi, horiz = TRUE, col = "red", space = 3,
      main = "Distribution* simulée des fréquences\ndes positions
   d'arrivée",
      ylab = "Position d'arrivée",
      xlab = "Fréquences simulées")
}
```

```
> marcheB3()
Moyenne des positions d'arrivée simulées
[1] 0.54
```

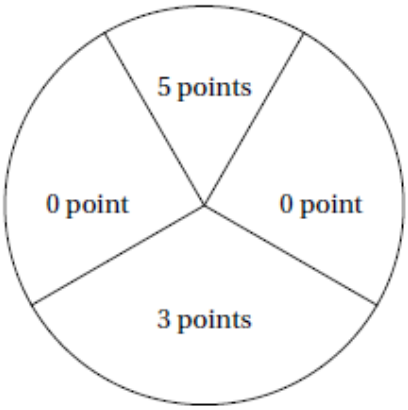


La juxtaposition des graphiques permet de donner du sens aux différentes représentations, les résultats des 30 sauts, les 100 simulations, et la distribution des fréquences qui s'y rapportent. En exécutant plusieurs fois de suite la fonction, on observe le caractère chaotique du cheminement à comparer au caractère plus stable de la distribution, illustrant une loi du hasard. Autant d'aspect qui n'apparaissent pas lors d'une simulation unique.

La résolution mathématique des problèmes de marches aléatoires font souvent appel aux chaînes de Markov, elles ne sont donc en majorité pas accessibles aux élèves de lycée.

E – LA SIMULATION : UN VÉRITABLE OUTIL DE RÉSOLUTION DE PROBLÈME, EXEMPLE D'UN EXERCICE DE BAC

J'ai entrepris de proposer une solution* par simulation à des exercices de probabilité du bac glanés dans les annales depuis 2009. Dans le plupart des cas la simulation permet de trouver des solutions* numériques approchées "asymptotiquement exactes". En voici un exemple (Pondichéry avril 2011) :



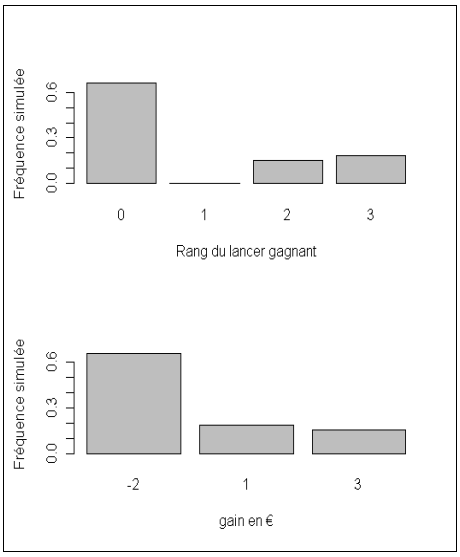
On suppose que les lancers sont indépendants et que le joueur touche la cible à tous les coups.

1. Le joueur lance une fléchette.
   On note p_0 la probabilité d'obtenir 0 point.
   On note p_3 la probabilité d'obtenir 3 points.
   On note p_5 la probabilité d'obtenir 5 points.
   On a donc $p_0 + p_3 + p_5 = 1$. Sachant que $p_5 = \frac{1}{2}p_3$ et que $p_5 = \frac{1}{3}p_0$ déterminer les valeurs de p_0, p_3 et p_5.
2. Une partie de ce jeu consiste à lancer trois fléchettes au maximum. Le joueur gagne la partie s'il obtient un total (pour les 3 lancers) supérieur ou égal à 8 points. Si au bout de 2 lancers, il a un total supérieur ou égal à 8 points, il ne lance pas la troisième fléchette.
   On note G_2 l'évènement : « le joueur gagne la partie en 2 lancers ».
   On note G_3 l'évènement : « le joueur gagne la partie en 3 lancers ».
   On note P l'évènement : « le joueur perd la partie ».
   On note $p(A)$ la probabilité d'un évènement A.
   a. Montrer, en utilisant un arbre pondéré, que $p(G_2) = \frac{5}{36}$.
      On admettra dans la suite que $p(G_3) = \frac{7}{36}$
   b. En déduire $p(P)$.
3. Un joueur joue six parties avec les règles données à la question 2.
   Quelle est la probabilité qu'il gagne au moins une partie ?
4. Pour une partie, la mise est fixée à 2 €.
   Si le joueur gagne en deux lancers, il reçoit 5 €. S'il gagne en trois lancers, il reçoit 3 €. S'il perd, il ne reçoit rien.
   On note X la variable aléatoire correspondant au gain algébrique du joueur pour une partie. Les valeurs possibles pour X sont donc : -2, 1 et 3.
   a. Donner la loi de probabilité de X.
   b. Déterminer l'espérance mathématique de X. Le jeu est-il favorable au joueur ?

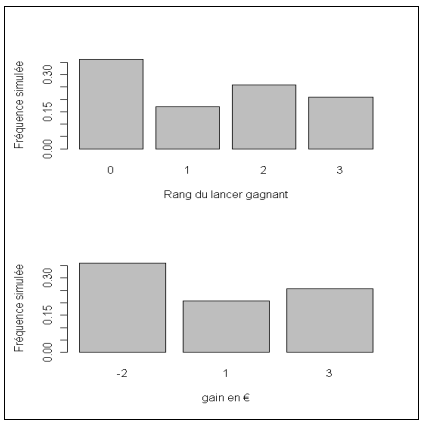
La stratégie de simulation suit le protocole expérimental. Les procédures passent en revue des fonctions **R** classiques. Il est important de signaler la différence entre l'espérance demandée et la moyenne calculée.

```
# On arrête les lancers dès que la partie est gagnée.
# Distribution simulée de variables
# et estimation de probabilités.
flechesA <- function(cible = c(0, 3, 5), proba = c(3/6, 2/6, 1/6),
                     n = 3, gagne = 8, nbsim = 2000){
   vectgagne <- vector(length = nbsim)
   tablogagne <- rep(0, n + 1) ; names(tablogagne) <- 0:n
   for (i in 1:nbsim){
      nblancers <- 0
      CumParties <- 0
      while (CumParties < gagne & nblancers < n) {
         partie <- sample(cible, 1, proba, replace = TRUE)
         CumParties <- CumParties + partie
         nblancers <- nblancers + 1
      }
      if (nblancers == n & CumParties < gagne) {
         vectgagne[i] <- 0
      } else {
         vectgagne[i] <- nblancers
      }
   }
 tablogagne[as.numeric(names(table(vectgagne))) + 1] <- table(vectgagne)
   AuMoins1sur6 <- 1 - (tablogagne[1] / nbsim)^6
   distribX <- c(tablogagne[1], tablogagne[4], tablogagne[3]) / nbsim
   names(distribX) <- c(-2, 1, 3)
   MoyenneX <- sum(distribX * as.numeric(names(distribX)))
   #****************** Affichage des résultats **********************
   cat("\nDistribution simulée du rang du lancers gagnant sur 3 lancers
\n")
   print(tablogagne / nbsim)
   cat("\nEstimation de la proba de perdre après 3 lancers =",
       tablogagne[1] / nbsim, "\n")
   cat("Estimation de la proba de gagner en 1 lancer =",
       tablogagne[2] / nbsim, "\n")
   cat("Estimation de la proba de gagner en 2 lancers =",
       tablogagne[3]/nbsim, "\n")
   cat("Estimation de la proba de gagner en 3 lancers =",
       tablogagne[4] / nbsim, "\n")
   cat("Estimation de la proba de gagner =",
   sum(tablogagne[2:4]) / nbsim, "\n")
   cat("Estimation de la proba de gagner au moins une partie sur 6 =",
       AuMoins1sur6, "\n")
   cat("\n Distribution simulée de la variable aléatoires X :\n")
   print(distribX)
   cat("Moyenne simulée de X =", MoyenneX, "€ \n")
   par(mfrow = c(2, 1))
   barplot(tablogagne / nbsim, xlab = "rang du lancer gagnant",
           ylab = "fréquence simulée")
   barplot(distribX, xlab = "gain en €", ylab = "fréquence simulée")
}
```

```
flechesA()
 Distribution simulée du rang du
lancers gagnant sur 3 lancers
     0      1      2      3
0.6610 0.0000 0.1535 0.1855

Estimation de la proba de perdre
après 3 lancers = 0.661
Estimation de la proba de gagner en
1 lancer = 0
Estimation de la proba de gagner en
2 lancers = 0.1535
Estimation de la proba de gagner en
3 lancers = 0.1855
Estimation de la proba de gagner =
0.339
Estimation de la proba de gagner au
moins une partie sur 6 = 0.9165918

Distribution simulée de la variable
aléatoires X :
    -2      1      3
0.6610 0.1855 0.1535
Moyenne de X = -0.676 €
```




```
flechesA(gagne = 6)
Distribution simulée du rang du lancer gagnant sur 3 lancers
     0      1      2      3
0.3615 0.1715 0.2580 0.2090

Estimation de la proba de perdre après 3 lancers = 0.3615
Estimation de la proba de gagner en 1 lancer = 0.1715
Estimation de la proba de gagner en 2 lancers = 0.258
Estimation de la proba de gagner en 3 lancers = 0.209
Estimation de la proba de gagner = 0.6385
Estimation de la proba de gagner au moins une partie sur 6 = 0.9977682

Distribution simulée de la variable aléatoires X :
    -2      1      3
0.3615 0.2090 0.2580
Moyenne de X = 0.26 €
```



F – SIMULATION D'UN PROBLÈME DONT LA SOLUTION MATHÉMATIQUE EST COMPLEXE – UTILISATION D'UNE FONCTION PARTICULIÈRE `rle(...)`

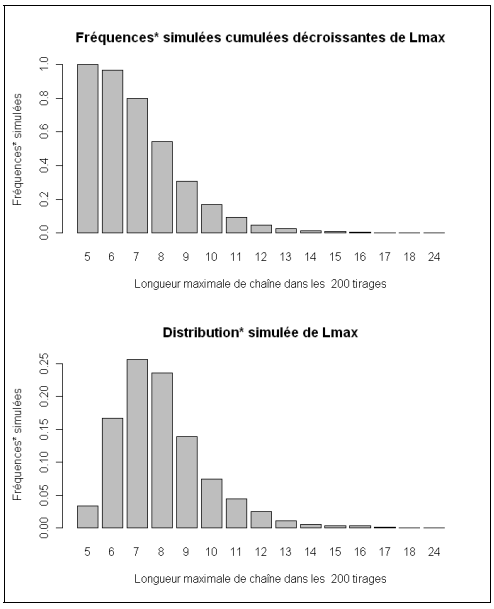
- On lance 200 fois une pièce équilibrée, quelle est la probabilité d'obtenir une suite d'au moins 6 piles ou 6 faces consécutifs (on dit une "chaine" de longueur 6 ou plus).
- Nous allons établir la distribution de la variable **LMax** longueur de la chaine de longueur maximale. Il faut utiliser la fonction interne **`rle()`** qui fournit un tableau des effectifs des longueurs de toutes les chaines rencontrées.
- Pour comprendre comment marche la fonction **`rle()`** de **R** qui détecte et comptabilise les "chaines" voici un exemple détaillé en lignes de commandes :

```
# QueFait rle() ?
> L = 6 ; piece = c("P", "F") ; tirages = 20
> serieLMax <- NULL ; i <- 1
> #
> (experience <- sample(piece, tirages, replace = T))
 [1] "F" "F" "F" "P" "F" "F" "P" "P" "F" "P" "P" "P"
[13] "F" "P" "P" "P" "F" "P" "F" "F"
> (chaines <- rle(experience))
Run Length Encoding
  lengths: int [1:11] 3 1 2 2 1 3 1 3 1 1 ...
  values : chr [1:11] "F" "P" "F" "P" "F" "P" ...

chaines est un objet de R contenant 2 listes :
  lengths contenant les longueurs des 11 chaines de la série experience et
  values contenant les valeurs de ces 11 chaines.
Ainsi il 3 F puis 1 P puis 2 F puis 2 P etc...

> chaines$length
 [1] 3 1 2 2 1 3 1 3 1 1 2
> serieLMax[i] <- max(chaines$length)

> serieLMax
[1] 3

Après la première simulation de 20 tirages, la longueur de la chaine de longueur maximale est donc 3
```

```
On fait une deuxième simulation de 20 tirages :
> i <- i + 1
> (experience <- sample(piece, tirages, replace = T))
 [1] "P" "P" "F" "P" "F" "P" "P" "P" "F" "P" "P" "F"
[13] "P" "P" "F" "F" "P" "F" "F" "P"
> (chaines <- rle(experience))
Run Length Encoding
  lengths: int [1:13] 2 1 1 1 3 1 2 1 2 2 ...
  values : chr [1:13] "P" "F" "P" "F" "P" "F" ...
> chaines$length
 [1] 2 1 1 1 3 1 2 1 2 2 1 2 1
> serieLMax[i] <- max(chaines$length)

> serieLMax
[1] 3 3

> i <- i + 1
●●● Après 48 simulations de 20 jets, on obtient :

> serieLMax
 [1] 3 3 3 6 3 5 5 4 3 5 4 5 3 4 3 6 4 6 3 5 5 6 3 4
[25] 8 5 4 4 5 3 5 6 3 5 8 3 7 3 5 7 2 4 5 4 4 5 5 5

La fréquence de longueurs supérieures ou égales à 6 est obtenus en divisant son effectif par le nombres de simulations :

> (FreqLongSupL <- sum(serieLMax >= L) / i)
[1] 0.187
```

- La fonction suivante permet d'obtenir une série de valeurs simulées de **LMax**, dans la liste **`SerieLMax`** et d'en faire la description sous forme de tableau des fréquences et de diagrammes en barres.

```
# ALGO B1*SIMULATION*DU PROBLÈME DES "CHAÎNES" DE LONGUEUR 6
chainesL = function(L = 6, tirages = 200, nbsim = 2000){
   SerieLmax <- NULL ; piece = c("Pile", "Face")
   for (i in 1:nbsim) {
      expe <- sample(piece, tirages, replace = TRUE)
      chaines <- rle(expe)
      Lmax <- max(chaines$length)
      SerieLmax <- c(SerieLmax, Lmax)
   }
   tableLmax <- table(SerieLmax) / nbsim
   ltable <- length(tableLmax)
   cumLmax <- cumsum(tableLmax[ltable:1])[ltable:1]
   FreqLmax <- sum(tableLmax[(as.numeric(names(tableLmax)) >= L)])
   # Affichage des résultats et des graphiques
   cat("Une estimation de la probabilité d'au moins 1 chaîne\n",
       "de longueur", L, "ou plus dans les", tirages,
       "tirages vaut :", FreqLmax, "\n\n")
   cat("Distribution* simulée de la longueur de la chaîne\n",
       "de longueur maximale\n")
   print(tableLmax)
   par(mfrow = c(2, 1))
   barplot(cumLmax,
      xlab = paste("Longueur maximale de chaîne dans les ",
                   tirages, "tirages"),
      ylab = "Fréquences* simulées",
      main = "Fréquences* simulées cumulées décroissantes de Lmax")
   barplot(tableLmax,
      xlab = paste("Longueur maximale de chaîne dans les ",
                   tirages, "tirages"),
```



```
      ylab = "Fréquences* simulées",
      main = "Distribution* simulée de Lmax")
}

> chainesdB1()
Une estimation de la probabilité d'au moins 1 chaîne
 de longueur 6 ou plus dans les 200 tirages vaut : 0.966

Distribution* simulée de la longueur de la chaîne
 de longueur maximale
SerieLmax
     5      6      7      8      9     10     11     12
0.0340 0.1670 0.2560 0.2360 0.1390 0.0745 0.0445 0.0245
    13     14     15     16     17     18     24
0.0105 0.0055 0.0035 0.0030 0.0010 0.0005 0.0005
```

- On remarque que dans ces 2000 simulations de 200 lancers il y $(0{,}003 + 0{,}001 + 0{,}0005 + 0{,}0005) \times 2000 = 10$ simulations dans lesquelles la chaine de longueur maximale est de taille supérieure ou égale à 16. Un bon exemple contre-intuitif !
- Dans cet exemple, la simulation rend sa résolution accessible à des élèves de lycée. L'algorithme reste "simple" si l'on utilise la fonction **`rle()`**.

G – QUELQUES EXEMPLES HISTORIQUES EN PHYSIQUE ET EN BIOLOGIE

J'ai développé quelques exemples de simulation de variables discrètes ou continues, que l'on peut consulter sur Wikipédia :

Simulation du modèle des urnes d'Ehrenfest (thermodynamique des gaz en physique) :
http://fr.wikipedia.org/wiki/Mod%C3%A8le_des_urnes_d%27Ehrenfest#Exemple_de_programme_en_langage_R_permettant_de_simuler_l.27.C3.A9volution_du_nombre_de_boules_dans_l.27urne_A_au_cours_de_n_tirages

Les lois de Dirichlet et simulations du cas particulier du modèle des urnes de Pólya (dynamique des populations en biologie) :
http://fr.wikipedia.org/wiki/Loi_de_Dirichlet#Mod.C3.A8les_d.27urne_et_simulations_du_cas_particulier_des_urnes_de_P.C3.B3lya

Simulation de l'expérience de l'aiguille (baguette) de Buffon (approximation historique de la valeur de pi) :
http://fr.wikipedia.org/wiki/Aiguille_de_buffon#Programme_en_langage_R_de_simulation_num.C3.A9rique_et_graphique

Simulation de l'expérience du "croix-pile" de D'Alembert (cas particulier historique d'une loi géométrique tronquée) :
http://fr.wikipedia.org/wiki/Pile_ou_face#Programme_de_simulation_num.C3.A9rique_et_graphique_en_langage_R

H – ILLUSTRATION DU THÉORÈME DE MOIVRE-LAPLACE OU COMMENT CONFRONTER R À DES OBJETS ÉTRANGES

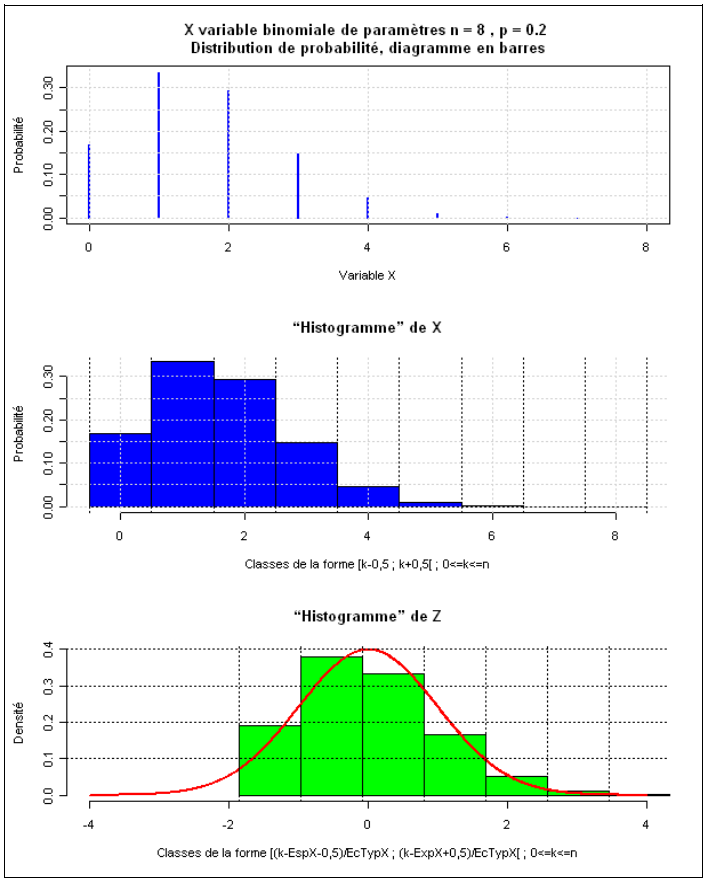
- C'est un exemple historique du passage d'une loi discrète, une loi binomiale, à une loi continue, la loi "normale" (de Gauss) centrée réduite. L'activité présentée consiste à illustrer graphiquement la distribution de probabilité d'une variable aléatoire X suivant une loi binomiale, sous forme d'"histogramme", puis à représenter de la même façon la distribution de la variable X centrée réduite pour ensuite montrer comment la courbe représentant la fonction de densité de la loi normale centré réduite s'ajuste sur cet "histogramme" de façon de plus en plus précise au fur et mesure que n grandit.
- "Histogramme" est entre guillemets car c'est un outil de la statistique descriptive dans lequel la surface des rectangles représente les effectifs ou les fréquences des classes d'une série statistique observée, relativement à une variable continue. Lorsque les classes n'ont pas toutes la même étendue, les ordonnées représentent la densité de classe c'est à dire la fréquence de classe divisée par l'étendue de classe, de façon à ce que la surface totale fasse 1. L'histogramme n'est donc pas l'outil adapté aux probabilités discrètes.
- Ce que confirme bien **R** dont la fonction `hist(...)` ne sait pas faire de graphique à partir d'une loi de probabilité discrète. Il va donc falloir imposer des valeurs à des paramètres auxquels on n'a pas accès directement, mais par l'intermédiaire de la fonction `class(...)`.
- Soit X une variable aléatoire suivant une loi binomiale de paramètres n et p, d'espérance E(X) = n×p et de variance V(X) = n×p×(1 − p). La variable Z = (X − n×p) / racine(n×p×(1 − p)) est la variable centrée réduite correspondante. Pour X, on construit des classes "artificielles" de la forme [k − 0,5 ; k + 0,5], à la base d'un rectangle dont la hauteur représentera la probabilité de k, 0 ≤ k ≤ n. On passe ensuite à Z avec des classes de la forme [((k − 0,5) − E(X)) / √V(X) ; ((k + 0,5) − E(X)) / √V(X)], à la base d'un rectangle dont la hauteur représentera cette fois la probabilité de k divisée par l'étendue de la classe 1 / √V(X) autrement dit une densité. On est alors prêt à superposer la courbe de la densité gaussienne.

`LhistX <- list(breaks = bornesXCR, counts = DistribX * n, mids = XCR, density = DistribX * EcTypX, equidist = TRUE, xname = NULL)` crée un liste contenant tous les éléments nécessaires à la construction d'un histogramme : bornes de classes, effectifs, centres de classes, densité. Tous ces éléments sont habituellement générés par la fonction `hist(...)` à partir de la série statistique représentée. On force ensuite **R** à prendre ces valeurs comme paramètres d'un histogramme en attribuant à `LhistX` la classe (`"class"`) histogramme. La fonction `plot(LhistX)` pourra alors construire un histogramme classique à partir de ces données. `dnorm(...)` est la densité gaussienne.

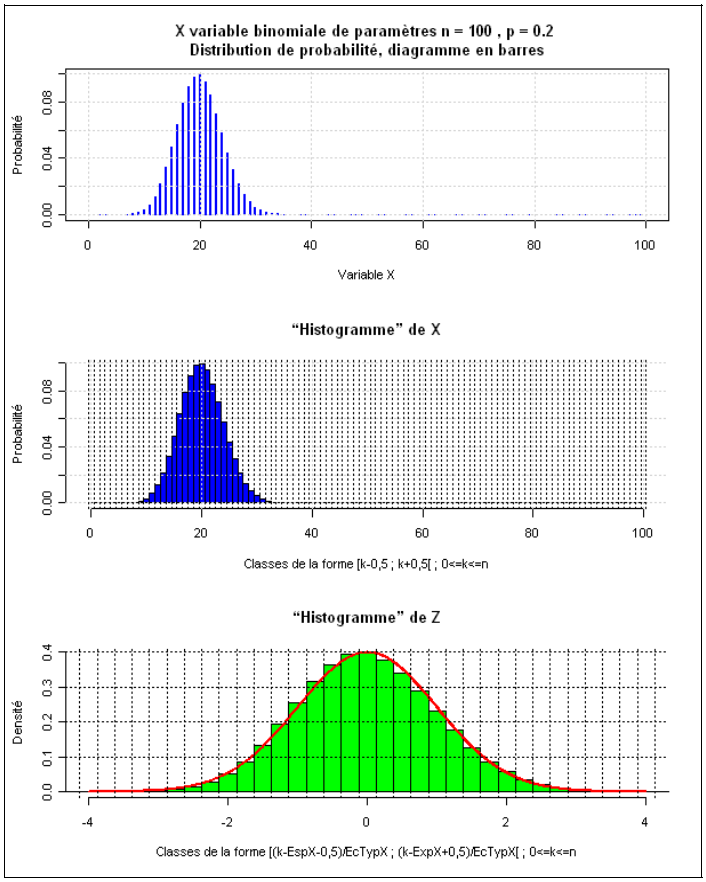
```
MoivreLap <- function(n = 100, p = .2){
   X <- 0:n
   EspX <- n * p ; XC <- X - EspX
   EcTypX <- sqrt(n * p * (1 - p)) ; XCR <- XC / EcTypX
   bornesX <- seq(-.5, n + .5, 1)
   bornesXCR <- c((XC - 1 / 2) / EcTypX, (n - EspX + 1 / 2) / EcTypX)
   DistribX <- dbinom(X, n, p)
   LhistX <- list(breaks = bornesX, counts = DistribX * n, mids = X,
                  density = DistribX, equidist = TRUE, xname = NULL)
      class(LhistX) <- "histogram"
   LhistXCR <- list(breaks = bornesXCR, counts = DistribX * n, mids = XCR,
                    density = DistribX * EcTypX, equidist = TRUE, xname = NULL)
      class(LhistXCR) <- "histogram"
   XGaussCR <- seq(-4, 4, length = 1000) ; YGaussCR <- dnorm(XGaussCR)
   maxY <- max(YGaussCR, DistribX)
   # Affichage des graphiques
   # Partage de la fenêtre graphique
   par(mfrow = c(3, 1))
   # Le diagramme en bâtons
   plot(X, DistribX, type = "h", col = "blue", lwd = 2,
         main = paste("X variable binomiale de paramètres n =", n, ", p =", p,
                      "\nDistribution de probabilité, diagramme en barres"),
         ylab = "Probabilité",
         xlab = "Variable X")
```

```
      grid()
   # L'"histogramme" de X
   plot(LhistX, freq = FALSE, col = "blue",
         main = ""Histogramme" de X",
         xlab = "Classes de la forme [k-0,5 ; k+0,5[ ; 0<=k<=n",
         ylab = "Probabilité",
         ylim = c(0, max(DistribX)))
      grid()
      abline(v = seq(-.5, n + .5, 1), lty = 3)
   # L'"histogramme" de la variable centrée réduite Z
   # avec densité gaussienne superposée
   plot(LhistXCR, freq = FALSE, col = "green", xlim = c(-4, 4),
         main = ""Histogramme" de Z",
         xlab = paste("Classes de la forme [(k-EspX-0,5) / EcTypX ;",
               "(k-ExpX+0,5) / EcTypX[ ; 0<=k<=n"),
         ylim = c(0, maxY),
         ylab = "Densité")
      points(XGaussCR, YGaussCR, type = "l", col = "red", lwd = 2)
      abline(h = c(.1, .2, .3, .4), v = bornesXCR, lty = 3)
}
```

MoivreLap(n = 8, p = .2)



MoivreLap(n = 100, p = .2)



La recherche des valeurs pertinentes pour la construction de l'histogramme permet de donner du sens à la notion délicate de densité abordée pour la première fois en terminale.

Cette fonction permet de comprendre le fonctionnement de **R**, de voir un exemple dans lequel on donne une propriété à un objet, ce qui déterminera la façon dont il sera traité par la suite, par la fonction **`plot(...)`**.

VII – CONCLUSION

Il est assez étonnant de remarquer comment **R**, logiciel professionnel de statistique, se laisse aisément détourner de son objet initial pour s'adapter à des domaines aussi variés que ceux que l'on vient de passer en revue. La très grande majorité des références que l'on peut trouver concernent le domaine de la statistique (essentiellement inférentielle) et de l'analyse de données que je n'ai pas abordées dans ce document.

Dans ces exemples nous avons passé en revue quelques fonctionnalités du langage **R** en mettant l'accent sur ses capacités à mettre en œuvre des algorithmes dans les domaines aussi variés que l'analyse, la géométrie, les probabilité et la statistique. Les programmes obtenus sont facilement lisibles et de difficulté accessible aux lycéens à condition de s'y investir un minimum.

Nous avons vu comment les capacités graphiques de **R** permettent non seulement d'illustrer facilement bon nombre de notions mathématiques des programmes, mais aussi de mettre en œuvre des stratégies permettant de surmonter certains obstacles didactiques du cours de probabilité.

La simulation en probabilité et statistique peut devenir un véritable outil de résolution (approchée) de problèmes. Il ne semble pas utopique de penser que, dans les futurs sujets d'examen, on puisse accepter cette méthode à côté des solutions mathématiques classiques.

Alors que la multiplication des outils TICE que l'on voit en œuvre dans les documents rend difficile le choix des enseignants et augmentent d'autant les temps d'apprentissages chez les élèves, **R** offre une alternative digne d'intérêt. Il m'a personnellement permis d'étendre significativement mon fond d'exercices et d'illustrations de cours et m'a permis d'explorer les situations que je n'avais pas envisagées auparavant.

VIII – BIBLIOGRAPHIE SUCCINCTE

Je joins à ce document quelques "cartes de références", aides mémoires glanées sur internet, qui contiennent les commandes usuelles de **R**.

Pour terminer je citerai quelques ouvrages généralistes permettant d'approfondir la connaissance de **R** :

Lafaye De Micheaux Pierre, Drouilhet Remy, Liquet Benoit ; 2010 ; **Le logiciel R, maîtriser le langage, effectuer des analyses statistiques** ; Springer.

Millot Gaël ; 2009 ; **Comprendre et réaliser des tests statistiques à l'aide de R ; manuel pour les débutants** ; De Boeck.

François Husson ; Sébastien Lê ; Jérôme Pagès ; 2009 ; **Analyse de données avec R** ; P U De Rennes.

Pierre-André Cornillon ; 2010 ; **Régression avec R** ; Springer.

Pierre-André Cornillon, Arnaud Guyader, François Husson et al. ; 2010 ; **Statistique avec R** (2e édition) ; P U De Rennes.

Deepayan Sarkar ; 2008 ; **Lattice, Multivariate Data Visualization with R** ; Springer.

Frédéric Bertrand ; 2010 ; **Initiation aux statistiques avec R ; cours, exemples, exercices et problèmes corrigés ; Licence 3, Master 1, écoles d'ingénieur** ; Dunod.

Yadolah Dodge ; 2008 ; **Premiers pas en simulation** ; Springer Verlag.

Christian P. Robert, George Casella ; 2011 ; **Méthodes de Monte-Carlo avec R** ; Springer Verlag.