QUELQUES ACTIVITÉS AVEC R
LOGICIEL POLYVALENT PROFESSIONNEL DE STATISTIQUE
Illustrations en géométrie, analyse, probabilité et statistique(s) 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**.

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 solutions 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 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 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

[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), 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.

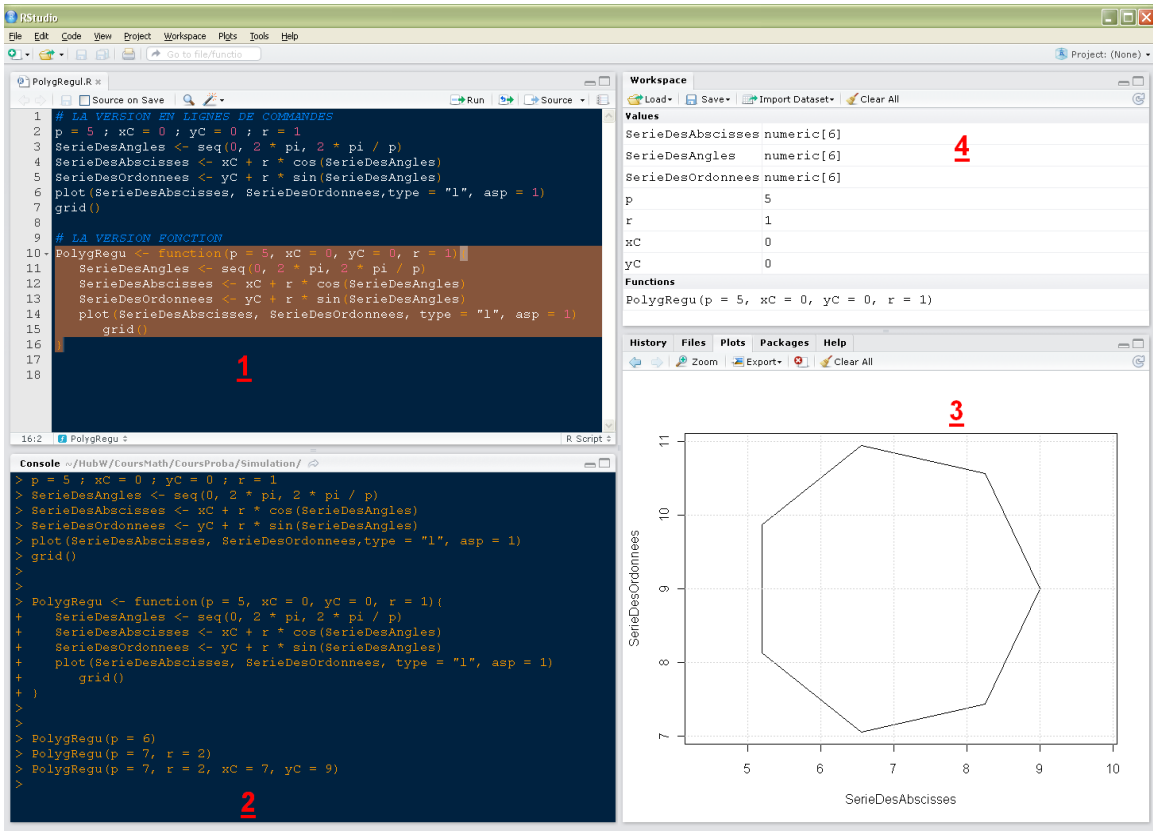
[3]http://pbil.univ-lyon1.fr/R/ ; http://perso.math.univ-toulouse.fr/dejean/2012/09/24/actualites/ ; http://www.biostat.fr/docs/cours1.pdf ;

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 touts 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.

Une fois les lignes de commandes saisies, pour les exécuter, il suffit de les sélectionner et de cliquer sur l'icone **"Run"**. Les lignes sont alors **copiées** dans la console (fenêtre **2**) puis **exécutées, automatiquement.** On peut aussi saisir des commandes directement dans la console, mais ça n'est pas très utile.

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**. Tous les 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

[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.

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 : 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(...) crée 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 ...)

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 !**

Dans les procédures **R,** les lignes de commandes sont en **orange**, 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.

[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.

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 statistiques).

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 type, 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

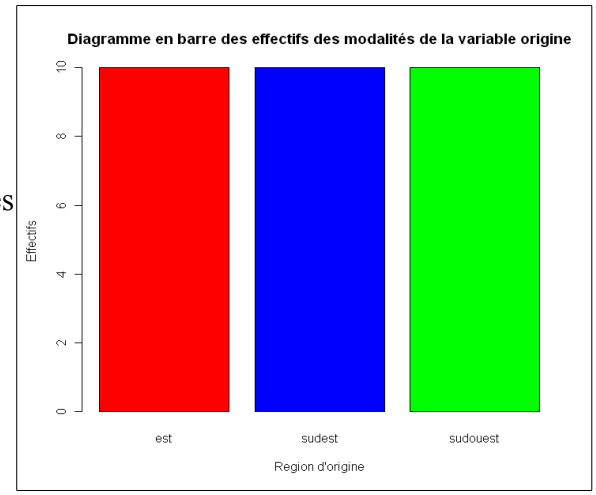
Il s'agit de faire un tableau des effectifs des modalités de la variable "origine". On dit que l'on effectue un tri à plat de la variable "origine".

```
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.

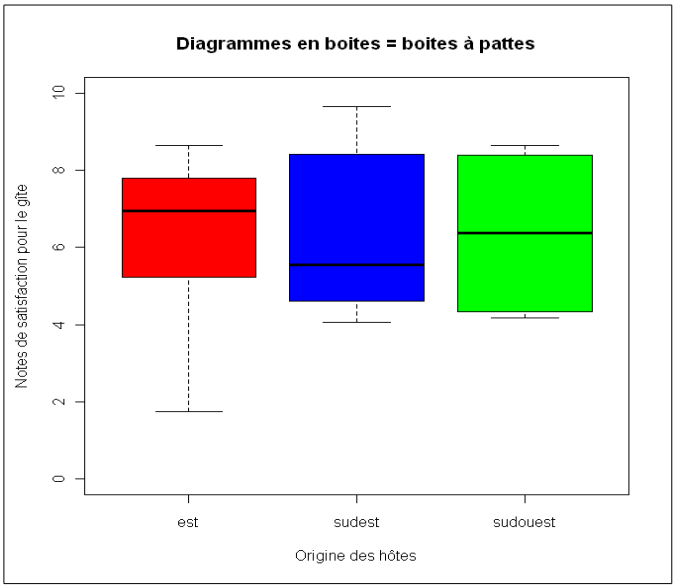
```
attach(satis)
plot(origine, note, horizontal = FALSE,
  col = c("red", "blue", "green"),
  main = "Diagrammes en boites = boites à pattes",
  xlab = "Origine des hôtes",
  ylab = "Notes de satisfaction pour le gîte",
  ylim = c(0, 10),
  varwidth = TRUE)
```

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 7) :

```
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 supérieur 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 superposer 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.

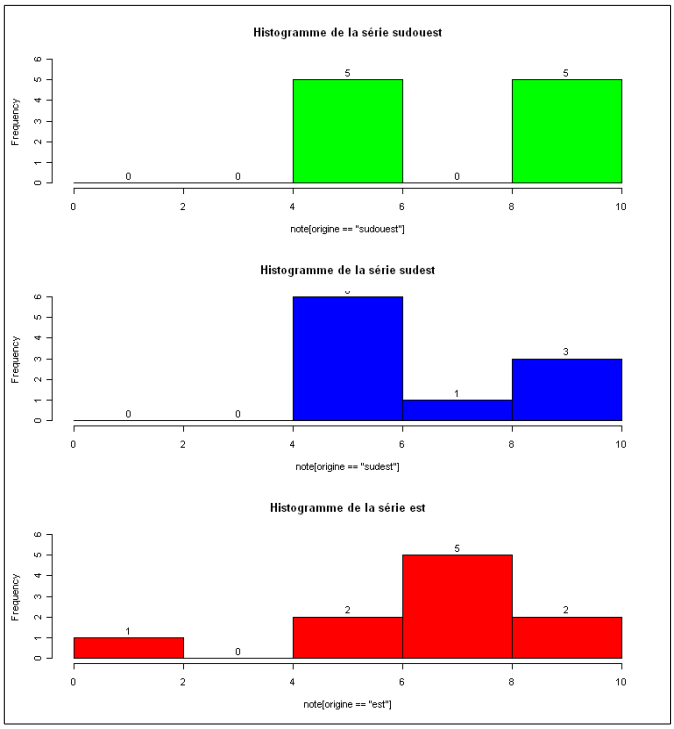
- `breaks(...)` fixe les valeurs des bornes des classes.
- `right = FALSE` fixe des intervalles du type [a ; b[(par défaut ce sont des classes à l'anglo-saxone]a ; b].
- `labels = TRUE` fait figurer les valeurs représentées, au dessus des rectangles.
- par défaut et pour des classes d'égale étendue, 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].

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

[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.

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 estime (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.

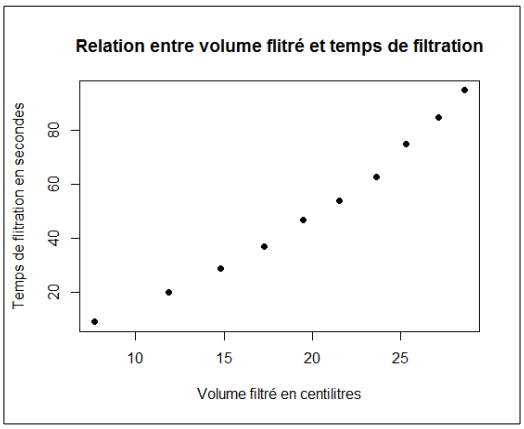
► Saisie des données sous forme de tableau et tracé du nuage de points (volu ; temps) :

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)
```

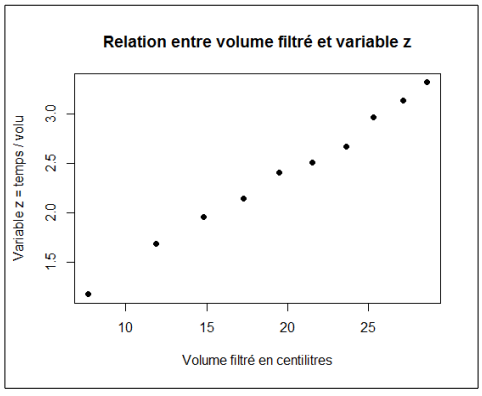
```
Extrait du tableau R nommé filtra
  volu temps
1  7.7     9
2 11.9    20
3 14.8    29
4 17.3    37
5 19.5    47
...

plot(volu, temps, pch = 21, bg = "black",
  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.
```



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

Les commandes et les résultats :

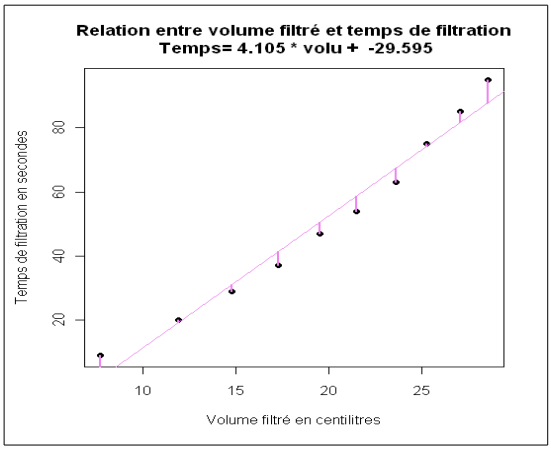
```
(z <- temps / volu)
Les 3 premières valeurs de la variable z
[1] 1.168831 1.680672 1.959459

plot(volu, z, pch = 21, bg = "black",
   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

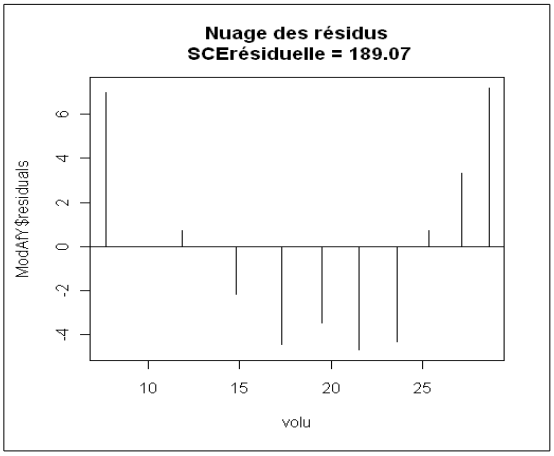
`plot` trace le nuage, `abline` ajoute la droite sur le nuage, `paste` permet d'afficher des messages avec le contenu de variables :

```
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".

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

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

plot(volu, ModAfY$residuals, type = "h",
      main = paste("Nuage des résidus\n SCErésiduelle =",
        round(SCErY, 3)))
   abline(h = 0)
```

● 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
```

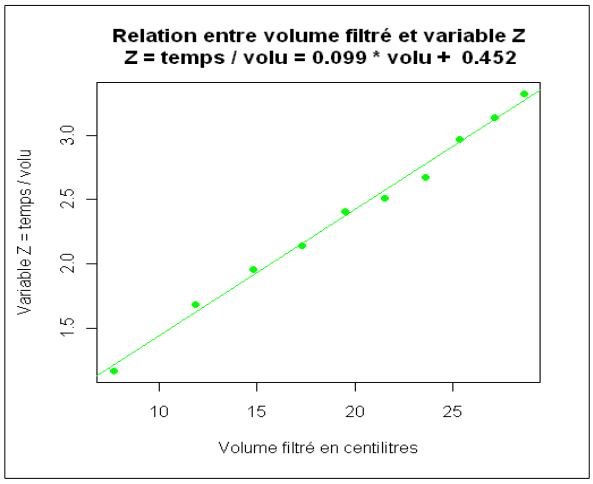
► L'ajustement du nuage (volu ; temps / volu) = (volu ; z) : un meilleur modèle

● calcul des paramètres :

Les commandes et les résultats :

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


ModAfZ$coefficients
  Intercept           X
 0.45176953  0.09855047
```

Le modèle s'écrit :



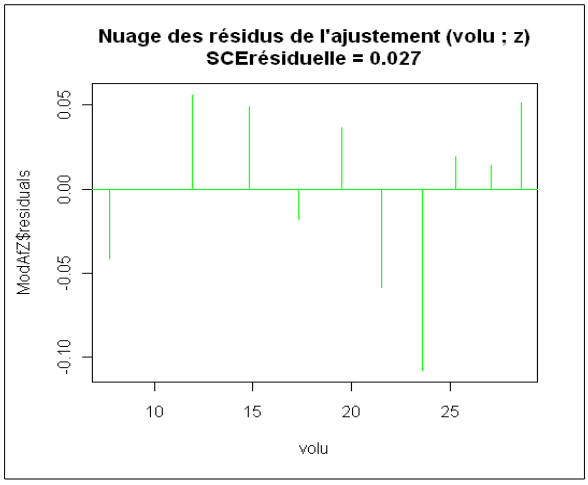
● On trace la droite sur le nuage (volu ; z) :

```
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)

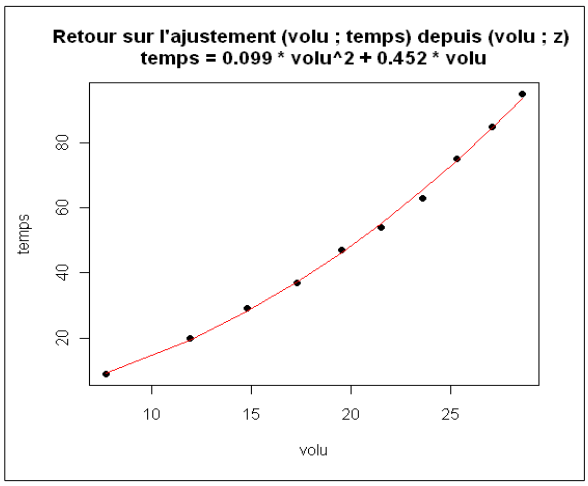
```
TempsZest <- function(x){
    yest <- (ModAfZ$coefficients[2] * x^2 + ModAfZ$coefficients[1] * x)
    return(yest)
}
```

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

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

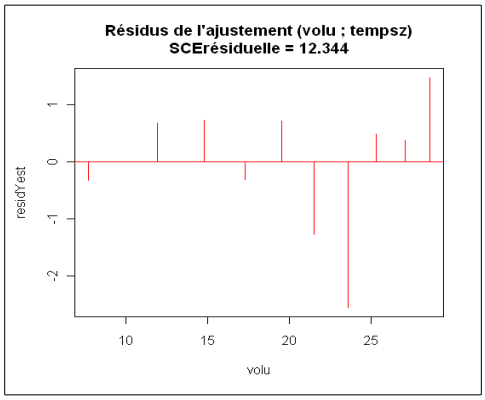
On trace la courbe ajustée sur le nuage observé. `lines(SérieDesX, SérieDesY, ...)` permet de tracer la courbe reliant des points repérés par leurs coordonnées cartésiennes,

```
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")
```

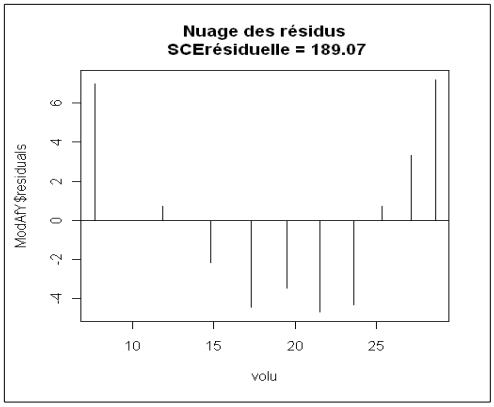


On calcule les résidus et on en fait une représentation graphique

```
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 DICOTOMIE

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
```

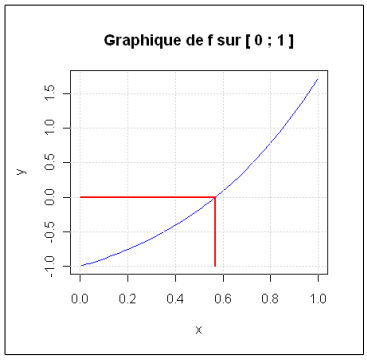
3° Deuxième prolongement dans lequel je vais généraliser l'algorithme précédent à une fonction strictement monotone sur un intervalle [A ; B], avec f(A) × f(B) < 0. À l'utilisation du programme il faut avoir exécuté la fonction concernée. Le programme demande de saisir les valeurs des deux bornes. La 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.

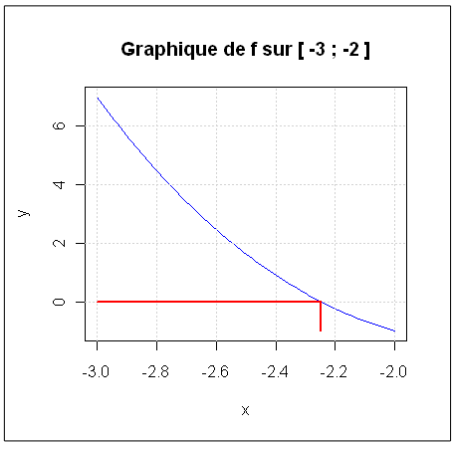
```
Etranger2009_2 <- function(Ainit, Binit, N = 4){
   a <- Ainit ; b <- Binit
      while (b - a > 10^-N){
         m <- (a + b) / 2
         P <- f(a) * f(m)
         if (P > 0) {a <- m} else {b <- m}
      }
   cat("Solution approchée de f(x) = 0 :\n",
      "[", a, ";", b, "]\n\n")
   plot(f, Ainit, 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 ]
```

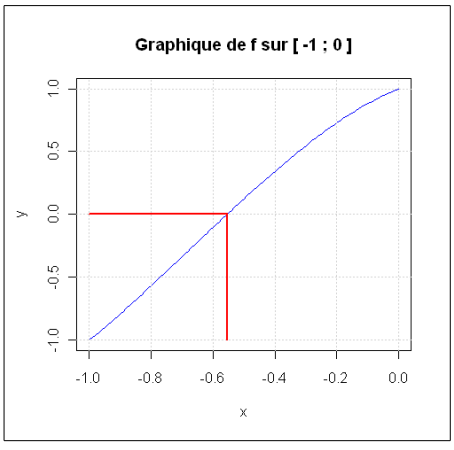


Parmi ces trois exemples, il y a une mauvaise utilisation du programme :

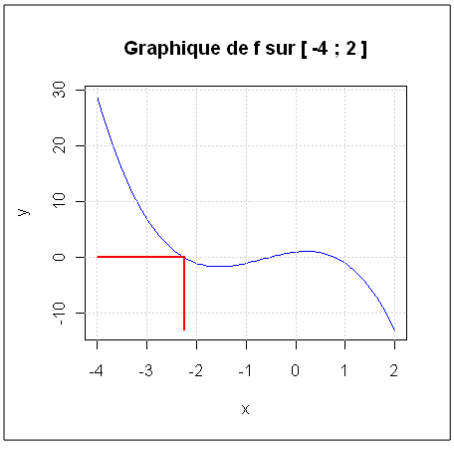
```
Avec :
f <- function(x){- x^3 - 2 * x^2
+ x + 1}

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 ]
```



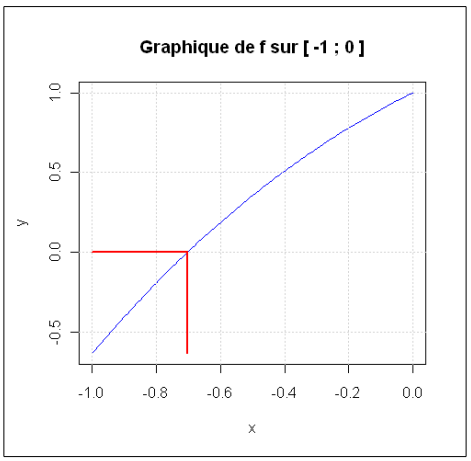
4° Troisième prolongement : une alternative didactique

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 ?

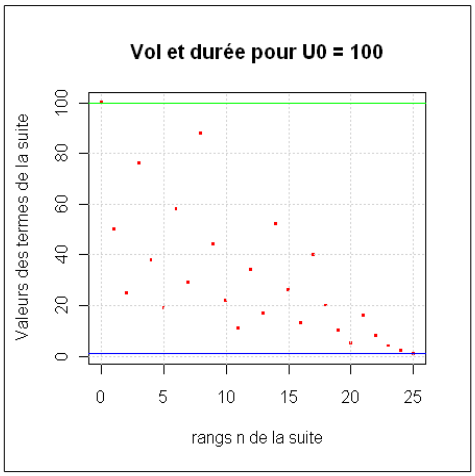
1° L'aspect chaotique du comportement de la suite est bien mis en évidence par la représentation graphique.

| 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.

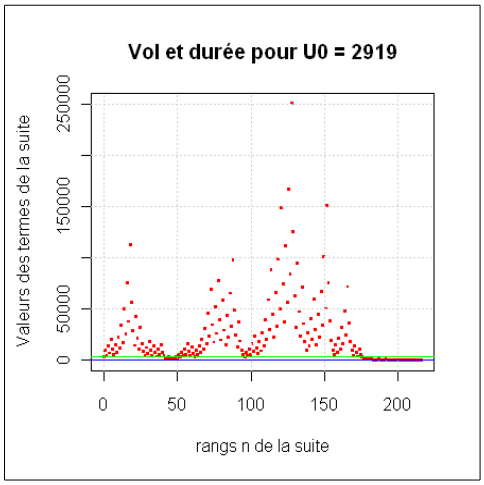
```
# Cette fonction calcule, en fonction de U0, le nombre
# d'itérations nécessaires pour atteindre la valeur 1
# La plus petite valeur d'indice tel que Un+1 < U0
#.ainsi que la valeur maximale de Un. Graphique de Un par n.
# Allez explorer le voisinage de U0 = 2919 **
syracn = function(U0 = 100){
   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.

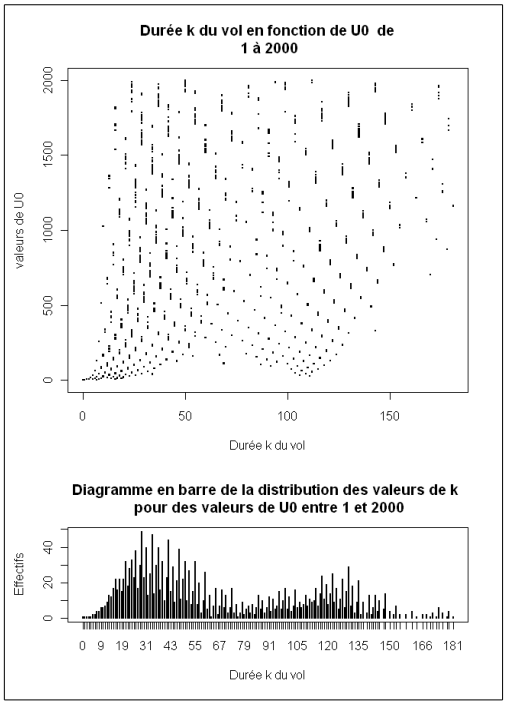
Il est intéressant de noter la "polyvalence" de la fonction **plot** qui, selon le type des données qui lui sont fournies, sait trouver un type de graphique adapté, nuages de points, diagrammes en barres, diagrammes en boites. Lorsqu'on lui fournit un tableau des effectifs, par exemple dans **plot(effec,...)**, **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 superposés 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.

Je profite de cette belle occasion pour 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                    Avec :
f <- function(x){exp(x)}                                n <- 50L
                                                        ...
# VERSION EN LIGNES DE COMMANDES                        ...
# Sur un intervalle [0 ; 1] où f est strictement
# CROISSANTE. On peut choisir la valeur de n            Grand_S 1.701156
n <- 5L                                                 Petit_s 1.735522
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
```

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                    Avec :
f <- function(x){exp(x)}                                n <- 50L
                                                        ...
# VERSION EN LIGNES DE COMMANDES                        ...
# Sur un intervalle [0 ; 1] où f est strictement
# CROISSANTE. On peut choisir la valeur de n            Grand_S 1.701156
n <- 5L                                                 Petit_s 1.735522
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
```

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

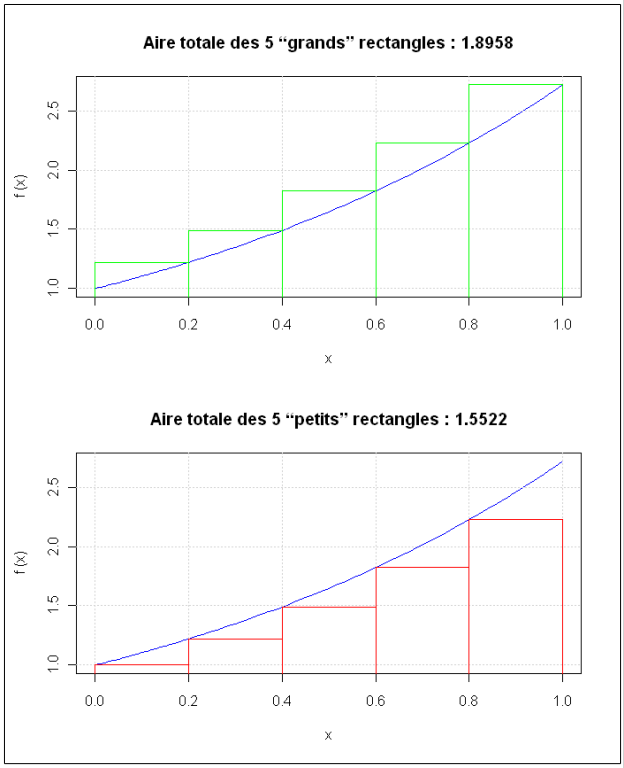
C'est la commande **`lines(...)`** incluse dans des boucles qui trace les segments de droite constituant les rectangles. Les paramètres de la fonction sont les bornes a et b de l'intervalle et le nombre n de subdivisions.

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.

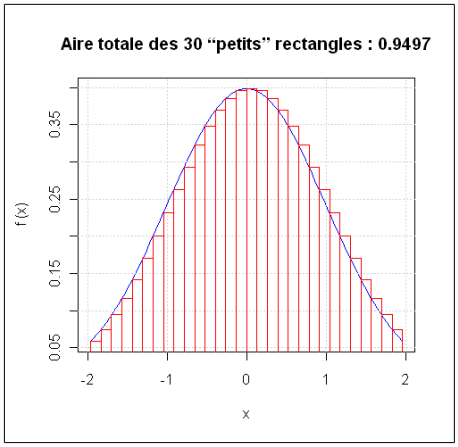
```
# Définition de la fonction utilisée
f <- function(x){exp(x)}

# 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.

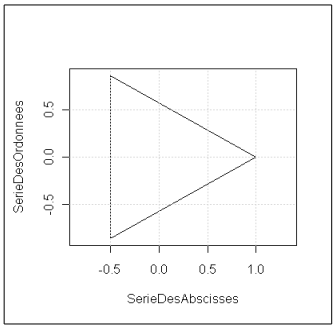
```
1 # LA VERSION EN LIGNES DE COMMANDES
2 p = 5 ; xC = 0 ; yC = 0 ; r = 1
3 SerieDesAngles <- seq(from = 0, to = 2 * pi, by = 2 * pi / p)
4 SerieDesAbscisses <- xC + r * cos(SerieDesAngles)
5 SerieDesOrdonnees <- yC + r * sin(SerieDesAngles)
6 plot(SerieDesAbscisses, SerieDesOrdonnees, type = "l",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

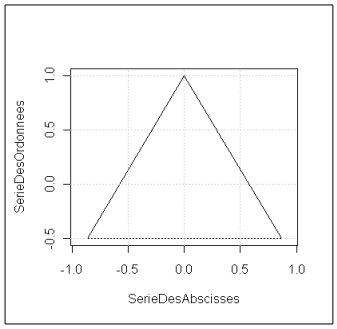
Il est intéressant d'observer ce qui se passe lorsque n devient grand.

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 ...)

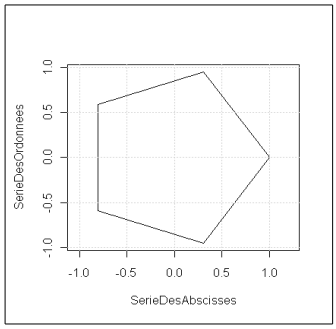
PolygReguR()



PolygReguR(theta = 0)



PolygReguR(p = 5, theta = 0)



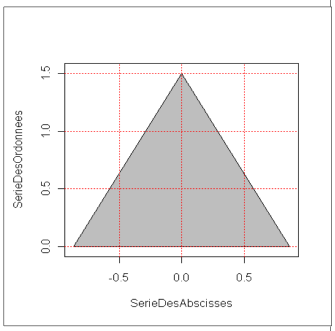
B – DES POLYGONES AUX TRIANGLES DE SIERPINSKI

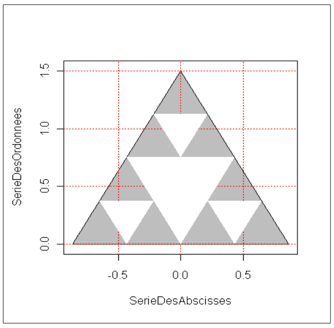
Les triangles équilatéraux sont des cas particuliers de polygones réguliers. Ça m'a fait penser aux triangles de Sierpinski. Je me suis donc demandé si l'on pouvait utiliser les algorithmes précédents pour construire des triangles de Sierpinsky.

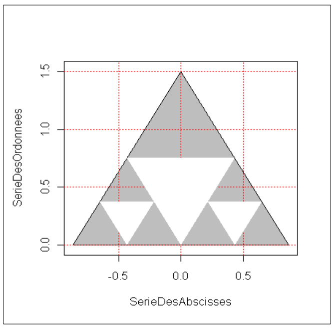
1° J'ai d'abord fait quelques essais en ligne de commande pour repérer les “répétitions”. **R** possède deux types de fonctions graphiques, celles de “haut niveau” (high level) qui génèrent des graphiques complets et celles de “bas niveau” (low level) qui complètent un graphique existant, créé par une 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”.

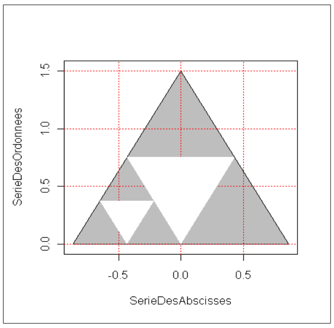
J'ai superposé des triangles blancs sur le triangle initial gris. On peut facilement faire des jeux de couleur.

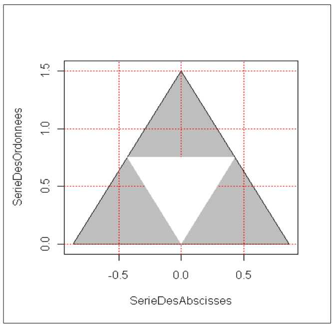
```
# LE PREMIER TRIANGLE
# CAS PARTICULIER DE POLYGONE : LES TRIANGLES DE SIERPINSKI
Sierp1 <- function(xC = 0, yC = 0, r = 1, theta = pi / 2){
   SerieDesAngles <- seq(0, 2 * pi, 2 * pi / 3)
   SerieDesAbscisses <- xC + r * cos(SerieDesAngles + theta)
   SerieDesOrdonnees <- yC + r * sin(SerieDesAngles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

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)

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





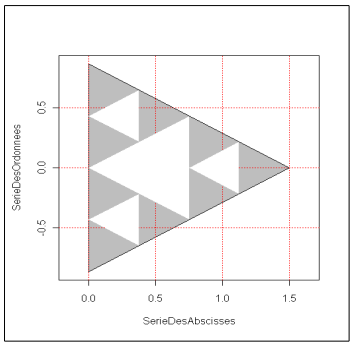







```
#--------------------
```

```
#---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)
```

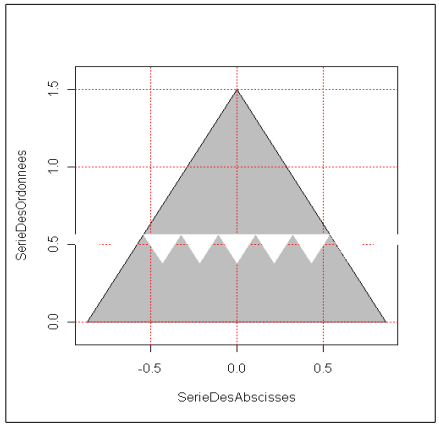


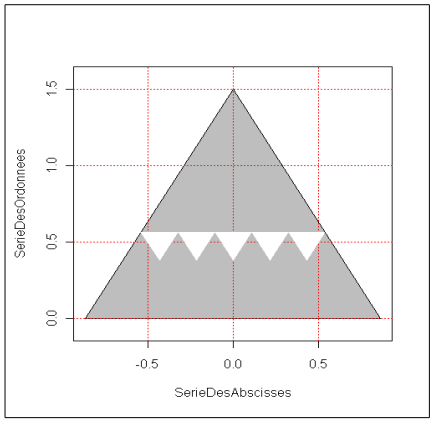
Il ne reste plus qu'à trouver le bon algorithme pour faire tout cela automathiquement !!!

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

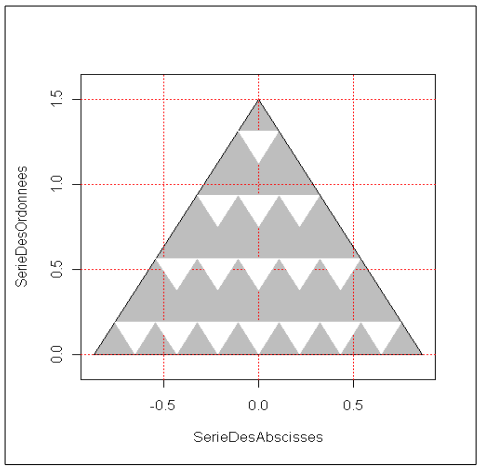
J'ai détaillé quelques étapes de la recherche qui m'a amené à l'algorithme terminal.

```
#  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 = "grey")
      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 = "white", 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 gris ...
# On se limite aux triangles de l'intérieur du grand gris.
#---------------------------------------
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 GRIS
#-------------------------------------------------------------
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)