

SEANCES 1 & 2

Introduction et découverte de Rstudio and Co.

Thibaud Deguilhem, Département d'économie - LADYSS 7533, Université de Paris

2020-11-02

Introduction générale au cours d'ASA-R

Le pourquoi de ce cours

Bien que ce cours ait été initialement construit spécifiquement pour les étudiant-e-s de Master 1 MECI et APE, quiconque voulant apprendre à manipuler R pour faire de la statistique appliquée peut en bénéficier et, pour cette raison, toutes les fiches sont disponibles en ligne et accessibles librement sur la page du cours. Si vous n'avez pas suivi de cours d'introduction aux statistiques en licence (statistiques descriptives en L1, inférentielles et économétrique en L2-L3), certains des concepts que nous allons utilisés peuvent être difficiles (voir très difficiles...), mais je ferai de mon mieux pour les expliquer clairement dans les différentes fiches et, si nécessaire, de brefs rappels seront ajoutés en ligne durant le semestre.

De même, si R est votre premier langage de programmation, vous trouverez probablement les premières séances assez difficiles, tout logiciel a un certain coût d'entrée (plus ou moins élevé) *a fortiori* sans connaître les bases de la programmation informatique. Mais *don't worry* ! Si R est votre premier langage de programmation, c'est une très bonne nouvelle, car ce que vous allez apprendre ici vous aidera assurément lorsque vous passerez sur d'autres langages Python, SQL, C++ (n'est-ce pas les étudiant-e-s PISE ?).

Enfin, bien que les techniques quantitatives de ce cours s'appliquent à la plupart des questions de *data analysis* dans la recherche en socio-économie ou dans le champ des études et du conseil, le cours proposera toujours une entrée “appliquée” aux sciences sociales à travers des exemples et des applications adaptées avec R afin d'aborder des problèmes couramment rencontrés dans la recherche et les études contemporaines (particulièrement intéressants pour les étudiant-e-s d'APE !).

- *En quoi consiste ce cours (en quelques mots) ?*

Ce cours est destiné à présenter les usages possibles de R (Rstudio plus précisément) pour réaliser une “bonne” étude de statistique appliquée. A travers la connaissance et la maîtrise des “bonnes pratiques” du codage/langage “R”, les étudiant-e-s qui suivent ce cours pourront alors développer différentes compétences importantes en techniques

quantitatives pour leur parcours en recherche ou leur future alternance allant à la **gestion des données**, aux **représentations graphiques**, en passant par l'**inférence statistique**, l'**économétrie** et l'introduction à l'**analyse de la causalité**¹.

- *Qu'est-ce que ce cours n'est pas (en quelques mots) ?*

Bien que les bases de mathématique, statistiques et probabilités acquises en licence soient nécessaires, ce cours ne couvre qu'une petite partie de ces différents domaines de la statistique appliquée et **jamais de façon théorique** !

Pourquoi ? Parce que je crois, comme Balboaca en introduction de son ouvrage paru en 2020² qu'en sciences sociales “*une excellente théorie statistique n'a que peu de valeur en l'absence d'applicabilité sur des données réelles. En outre, toute excellente théorie mathématique trouvera sa fin tôt ou tard sans une mise en œuvre appropriée*”.

Si vous souhaitez : (i.) entrer un peu plus dans la théorie qui supporte les outils d'analyse que nous utiliserons et appliquerons (et leur formalisation), (ii.) effectuer des analyses plus complexes ou (iii.) créer des représentations graphiques que nous ne traiterons pas dans ce cours, notamment pour votre mémoire (peut-être !), je suis certain que vous trouverez la réponse en effectuant une recherche rapide sur Google ou en regardant toutes les ressources conseillées sur la page du cours. Sinon, un forum est toujours à votre disposition en ligne, utilisez tout ce qui est à votre disposition !

¹ Seulement une première introduction, pour les PISE et les CCESE nous approfondirons cette question de la causalité au second semestre avec le cours d'*Analyse multivariée*.

² *Applied and Computational Statistics*, Basel : MDPI Press.

L'informatique et l'analyse statistique appliquée (computational statistics) : un domaine récent et en expansion

Au début du XXème siècle, le calcul des procédures statistiques simples telles que l'ANOVA (analyse de la variance à un ou plusieurs facteurs) ou la régression linéaire multiple étaient assez laborieuses et des procédures plus avancées telles que l'analyse factorielle ou celles plus complexes d'estimation nécessitaient une quantité importante de calculs matriciels alors très difficile à réaliser.

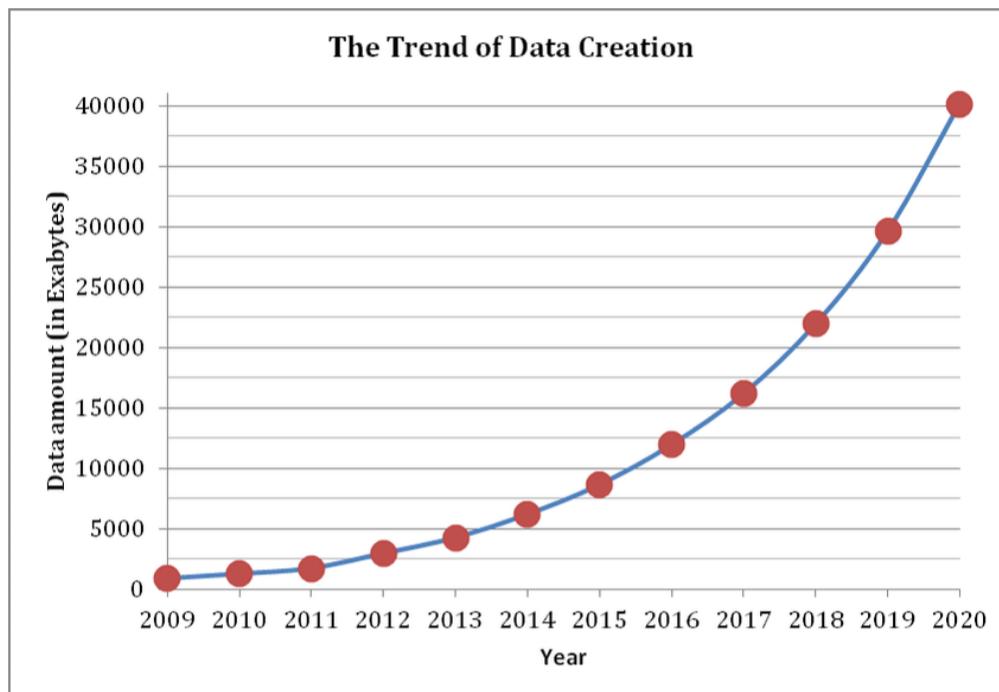
On peut s'en douter en regardant le travail au sein d'un laboratoire de statistiques en 1930 (à Chicago)³ :



The Iowa State University Statistics Lab in 1930

³ Pour plus de précision sur l'histoire des “computational statistics”, voir <https://www2.amstat.org/asa175/statcomputing.cfm>

Aujourd’hui, en raison de l’avancée dans le domaine informatique au cours du dernier demi-siècle, le champ de ce qui s’appelle *computational statistics* a radicalement transformé la pratique de la statistique appliquée. Petit à petit ces deux domaines, *computational sciences* et les *statistics*, n’en n’ont fait plus qu’un, au point qu’aujourd’hui, parler de statistique appliquée, ne peut désormais plus se faire sans la considérer comme la fusion des statistiques théoriques d’une part et de l’informatique d’autre part, toutes les deux mises au service du traitement et de l’analyse de très nombreux jeux de données dont la collecte ne cesse d’augmenter (exponentiellement).



Désormais, il est très facile de trouver un jeu de données⁴, de soumettre des données à l’un des innombrables programmes statistiques disponibles et obtenir une sortie (un résultat ou “*output*”) donnant l’impression, fausse et dangereuse, que le logiciel permettant de produire des résultats statistiques prenait le pas sur les questions (théoriques et analytiques) auxquelles l’analyse se proposait de répondre. Cette croissance exponentielle de ce qui concerne les “data”, de la collecte à l’analyse, peut être considérée comme une aubaine pour utiliser les données disponibles, toujours plus nombreuses, obtenir des informations très riches sur des contextes très divers et variés et, *in fine*, produire une analyse quantitative pertinente du monde social (*c.f.* séances 5 à 8) et/ou se positionner dans un débat à propos d’orientations de politiques publiques (*c.f.* séance 9). Mais, cela incite parfois à ne pas se poser suffisamment les “bonnes” questions en amont du traitement et de l’analyse⁵ avant générer des analyses et produire des résultats à ce propos. Or, comme le note Guttag en ouverture de son ouvrage de 2013⁶, “*un ordinateur fait deux choses, et deux choses uniquement : il effectue des calculs et il se souvient des résultats de ces calculs. Mais il fait ces deux les choses extrêmement bien*” (p. 1).

⁴ Nous reviendrons sur cette question lors de la séance 3 mais sachez dès maintenant qu’il existe de nombreux catalogues contributifs en ligne, Kaggle par exemple : <https://www.kaggle.com/>.

⁵ Quelle est la question précise que l’on se pose ? Que nous disent les travaux antérieurs sur la question ? Quels sont les arguments et les enjeux du débat académique et/ou institutionnel sur la question ?

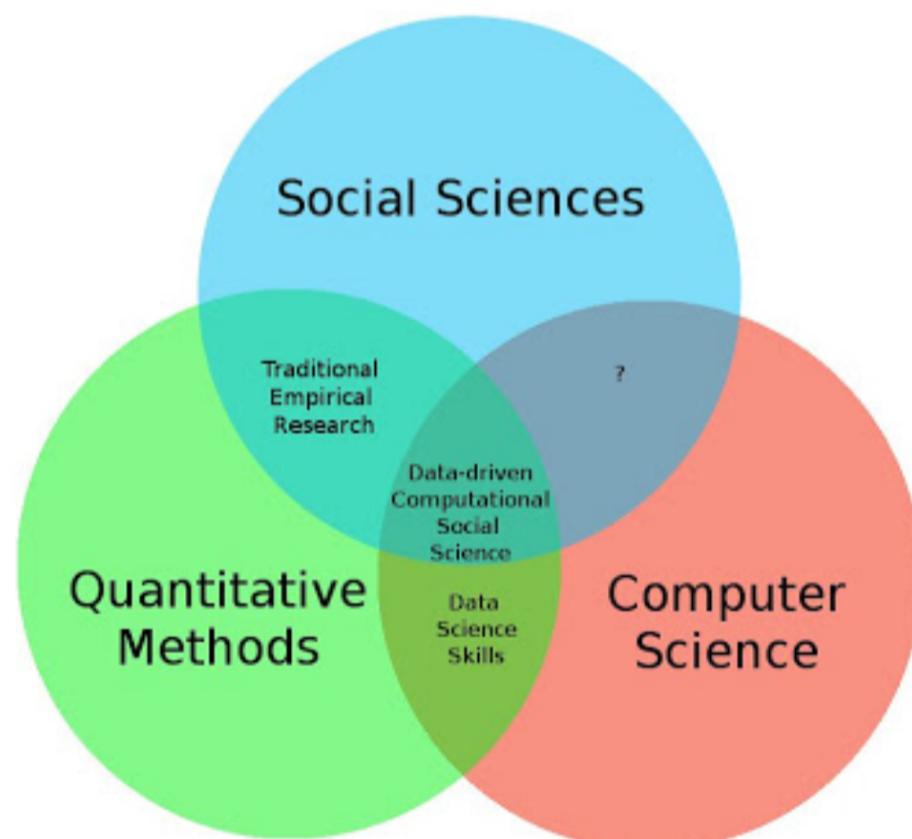
⁶ *Introduction to Computation and Programming Using Python*. Cambridge: MIT Press.

Concernant de réelle capacités réflexives en amont de ces calculs, c'est ailleurs qu'il faudra chercher...⁷.

Il est évident que l'émergence des *Data sciences* au cœur de la recherche et des études quantitatives en sciences sociales sur les populations statistiques, les échantillons ou les modèles d'estimation/prévision ont permis de grandes avancées et d'appliquer les outils de la statistique théorique et mathématique à de nombreux domaines (géographie, économie, sociologie, psychologie, sciences politiques...) permettant d'étendre nos questionnements à de nombreux phénomènes du monde social. L'augmentation de la puissance de calcul informatique combinée à cette articulation pertinente entre *data sciences* et *statistics* ont permis aux statistiques appliquées de devenir un outil très important (pas unique !) des sciences sociales à travers par exemple la construction d'algorithmes permettant de traiter des très grands jeux de données clients⁸. Cette alliance entre statistiques appliquées sur informatique et sciences sociales est devenu presque incontournable dans nombre de sous-champs disciplinaires (la socio-économie, la sociologie et la géographie notamment !), apportant de nouveaux outils d'analyse quantitative, des résultats originaux (produits sur de très grands échantillons) et des questions supplémentaires à des problématiques anciennes comme nouvelles, multidisciplinaires, interdisciplinaires et transdisciplinaires.

⁷ Précisons que nous ne nions pas qu'une vive discussion, plus journalistique qu'académique d'ailleurs, existe sur la question de savoir si un ordinateur peut réellement "penser" ou non. Ce débat, qui rejoint plus largement celui de l'"intelligence artificielle" dont le terme "intelligence" correspond plus à de l'algorithme d'apprentissage supervisé ou non (procédures de classifications notamment, *c.f.* semestre 2 du Master : cours d'"analyse multivariée appliquée") qu'à de véritables capacités réflexives, est bien au-delà de la portée de ce cours.

⁸ Ce que l'on appelle le "Big Data".



Ce qui comptera pour nous dans ce cours d'ASA-R, c'est la capacité à obtenir un ordinateur (normalement c'est bon pour tout le monde...), et plus précisément les programmes R et Rstudio (on verra comment plus loin), pour produire ce que nous voulons en analyse statistique appliquée par rapport à différentes questions que nous nous poserons.

Pour résumer en quelques points cette introduction : *pas de bonne analyse statistique appliquée en sciences sociales sans :*

1. une/plusieurs “bonne-s” question-s de recherche établies en amont de l’analyse et par rapport aux travaux antérieurs sur le sujet
2. de “bonnes” données correspondantes (collectées pour la question de recherche ou trouvées pour)
3. une bonne maîtrise des outils statistiques de collecte : “techniques d’échantillonnage”
4. une connaissance approfondie des techniques d’analyse statistique (descriptive, inférentielle, économétrique)
5. une “bonne” maîtrise des outils informatiques permettant d’appliquer ces techniques d’analyse sur les données à disposition
6. Produire de “bons” résultats, adaptés et pertinents pour apporter des réponses à la question de recherche
7. une “bonne” maîtrise des résultats afin d’interpréter “correctement” et “précisément” les output produits par rapport à la question que l’on se pose dans le contexte dans lequel ont été collectées les données



Objectif général du cours

Rappelons que l’objectif général de ce cours consistera à utiliser efficacement Rstudio en appliquant divers outils de l’analyse statistique afin de produire *in fine* des

résultats/output (sorties graphiques ou non) nécessaires,
avant de les interpréter et de le commenter.

Lectures obligatoires

Le premier chapitre (pp. 32–48) de l’ouvrage de Daniel J. Denis (2016)

[https://drive.google.com/drive/folders/10pmxgDqRUkpGMPpdRxfBKYSR3cxb6fqp?](https://drive.google.com/drive/folders/10pmxgDqRUkpGMPpdRxfBKYSR3cxb6fqp?usp=sharing)
usp=sharing est incontournable en complément de cette introduction.

Séances 1 et 2 : R, Rstudio and friends

Lecture obligatoire

Le premier chapitre (pp. 31–48) de l’ouvrage de Daniel J. Denis (2020)

[https://drive.google.com/drive/folders/10pmxgDqRUkpGMPpdRxfBKYSR3cxb6fqp?](https://drive.google.com/drive/folders/10pmxgDqRUkpGMPpdRxfBKYSR3cxb6fqp?usp=sharing)
usp=sharing est incontournable pour ces deux séances.

Les questions auxquelles vous devrez savoir répondre à la fin de la séance

- Comment installer R et Rstudio sur mon ordinateur personnel ?
- Comment utiliser l’interface de Rstudio et les différentes fenêtres ?
- Comment installer un package ?
- Qu’est-ce qu’un vecteur et une matrice ?
Comment manipuler ces objets avec R et à quoi servent-ils ?
- Comment faire des maths avec Rstudio ?

Objectif des séances 1 et 2

Ces deux premières séances d’introduction consistera d’abord en une présentation du logiciel Rstudio et à l’environnement R-Markdown. Par la suite, en revenant sur des notions élémentaires de mathématiques et/ou de

statistiques, les étudiant-e-s comprendront rapidement la logique générale du code et le fonctionnement de Rstudio comme R-Markdown.

Quelques rappels (très rapides !) d'informatique élémentaire avant de débuter sur R

Fondamentalement, tous les langages informatiques contiennent des éléments constitutifs d'une langue propre et reconnue (les 0 et 1 qui composent le code binaire). À partir de cette “couche primaire”, chaque logiciel produit une couche supérieure qui lui est propre, permettant de réaliser ce pour quoi le langage spécifique a été conçu. Pour un programme statistique tels que R et Rstudio, la couche supérieure comprendra, sans surprise, de nombreuses fonctions statistiques dont les commandes seront spécifiques (différentes sur STATA, sur Python etc.) !

Prenons désormais un exemple : le calcul d'une moyenne arithmétique. Il serait laborieux si à chaque fois que nous le souhaitons, nous devions entrer dans R le code suivant :

```
x <- c(0, 2, 5, 8, 9)  
  
x_mean <- sum(x)/length(x)
```

Résultat de cette commande :

```
x_mean  
## [1] 4.8
```

Nous avons bien calculé la moyenne arithmétique en additionnant les observations dans le “`vector(sum(x))`”, puis en divisant par le nombre d’observations, appelé aussi la “longueur du vecteur” `length(x)` (quelques souvenirs de L1/L2 : $\bar{x} = \frac{1}{n} \sum_{i=1}^n nx_i$). Bien entendu, tout bon logiciel statistique ne nous obligera pas à faire ce calcul chaque fois que nous souhaitons calculer une moyenne. Au lieu de cela, les logiciels R et Rstudio ont des fonctions intégrées qui effectuent ces opérations automatiquement et beaucoup plus rapidement. Sur Rstudio, tout ce que nous avons à faire pour obtenir la moyenne est d’écrire :

```
mean(x)  
## [1] 4.8
```

Cependant, il convient également de noter qu’au fond, les langages informatiques peuvent être réduits à ce que nous avons évoqué précédemment, ce que nous avons appelé “*code*”. Ce “*code*” fait référence à ce qui se passe “dans les coulisses”. En effet, si tous les langages informatiques étaient limités au travail en “*code*”, l’informatique dans son

ensemble, même pour des choses très simples, prendrait une éternité ! Pour gagner du temps, les développeurs de logiciels “collent” des morceaux de “code” ensemble afin d’automatiser certaines commandes communes ou régulières. Ainsi, ces “choses” faites automatiquement, comme calculer la moyenne par “`mean(x)`”, constitue un petit bout de programme spécifique pour justement calculer la moyenne (ça tombe bien, c’est ce que nous voulions) !

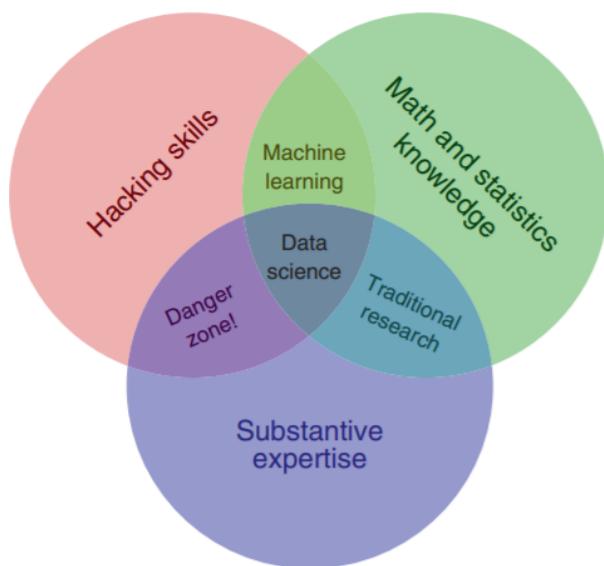
Dès lors que nous entrons “`mean(x)`” sur un vecteur nommé “`x`” dans Rstudio (il faudra bien évident le définir préalablement...), le logiciel a déjà le code pré-organisé pour effectuer cette opération. Autrement dit, l’ordinateur a déjà l’algorithme/programme (le calcul d’une moyenne simple est un exemple d’algorithme et un programme est un algorithme !) mis en place pour que nous puissions simplement faire un appel à cette commande au lieu de déplacer des informations binaires pour le programmer à nouveau et perdre *in fine* beaucoup de temps.

Alors, pourquoi cette discussion est-elle intéressante ? Il est tout simplement essentiel de comprendre le cheminement dans lequel nous nous apprêtons à nous engager, qui est celui de l’utilisation d’un langage informatique spécifique (R et Markdown) pour générer une sortie statistique ou autre que nous souhaitons ensuite interpréter. Connaître et maîtriser Rstudio signifie donc : **apprendre à interagir avec lui et lui faire faire ce que vous avez besoin de faire**. Cette maîtrise nécessite énormément d’essais et d’erreurs (c’est le *coût d’entrée*) et les débutant-e-s dans la programmation ont parfois l’impression que les “expert-e-s” du domaine ne font jamais d’erreur et programment facilement sans recevoir de messages d’erreur : **FAKENEWS** !

Les meilleurs programmeurs reçoivent continuellement des messages d’erreur et passent beaucoup de temps à sur le “débogage” de scripts et de programmes pour les faire fonctionner. Obtenir des messages d’erreur jusqu’à ce que quelque chose fonctionne est l’**art de la programmation**. La courbe d’apprentissage pour les langages de script tels que R peut paraître un peu raide au début, et souvent, vous penserez alors que l’ordinateur ne fera jamais ce dont vous avez besoin, mais avec **patience, persévérance, application et rigueur**, vous apprendrez les leçons de chaque erreur ou réussite et apprendrez maîtriser une nouvelle langue, la programmation en R, qui vous permettra d’échanger avec beaucoup de monde !

Un schéma qui illustrera la combinaison de compétences nécessaires pour réussir efficacement dans ce cours et qui

vous donnera un peu les pré-requis simplement en matière de motivation :



Pour la plupart des étudiant-e-s, la bonne combinaison se situe quelque part le milieu où se trouvent la “science des données” (*data sciences*) et la “recherche traditionnelle”.

Mais les choses ne sont pas aussi simples. Apprendre et utiliser R et Rstudio, c'est aussi parfois apprendre à faire un peu de “code hors de sentiers battus” en cours de route.

Notez la **zone de danger**, cependant, où avoir une certaine expertise de fond mais des connaissances très faibles en mathématiques et des statistiques vous fera échouer très probablement. Rappelez-vous bien que **R et Rstudio sont uniquement des outils au service d'autres outils** que sont les statistiques elles mêmes au service de questions théoriques/analytiques en sciences sociales... Le premier ne se substitue absolument pas au second qui ne se substitue absolument pas au troisième !

Avertissement pour ce cours : ne jamais supposer que les connaissances informatiques peuvent remplacer la maîtrise des mathématiques et des statistiques ou votre culture en sciences sociales⁹. Obtenir une sortie dans n'importe quel programme logiciel ne signifie pas comprendre la sortie, ou comprendre si ce que vous avez produit est correct, adapté et pertinent.

Qu'est-ce que R et Rstudio ?

R et Rstudio sont différents des autres logiciels de statistiques utilisées tels que SPSS et STATA par exemple, où traditionnellement l'utilisateur soumet une collection de lignes de code ou de mini-programmes, tous à la fois, à quel point le l'ordinateur génère un tas de sorties, certaines sont bienvenues et utiles, mais le reste est potentiellement superflus et des codes supplémentaires doivent venir désactiver un jeu d'options pour configurer la sortie statistique adaptée. A l'inverse, pour la plupart des fonctions de R ou Rstudio, c'est allant chercher dans des “bibliothèques” (packages) et/ou des environnements

⁹ Les applications de ce cours seront basés sur des travaux menés en économie et en sociologie. Il sera nécessaire de connaître les enjeux du débat académique sur les différentes thématiques !

spécifiques que R et Rstudio reconnaissent des lignes de code et produisent les sorties statistiques adaptées, ici chaque option est intégrée directement dans le code (très rarement par défaut). Cette manière de fonctionner ressemble un peu plus à un travail de développeur informatique, il faut donc connaître à la fois les bases du code mais aussi les packages que l'on utilise afin de converser avec le programme à la manière d'un informaticien. De cette façon, R est plus similaire aux langages informatiques (Python, Java, C++...).

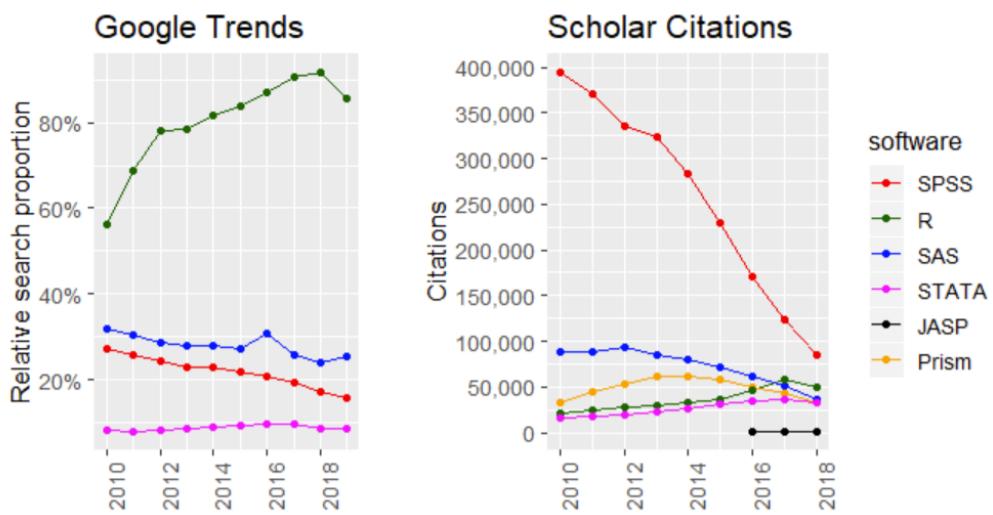
Pourquoi R est si génial ?

Comme indiqué précédemment, vous avez probablement déjà une idée de pourquoi R est si génial. Cependant, afin de vous éviter d'abandonner la première fois que vous vous heurtez à un mur de programmation (fameux messages d'erreur), voici six bonnes raisons de vous accrocher et de continuer :

1. R et Rstudio sont 100% gratuit¹⁰ et, par conséquent, s'appuie sur une très grande communauté d'utilisateurs (forum, blog, cours online...). Contrairement à SPSS, STATA, Matlab, Excel et JMP, R est, et sera toujours entièrement gratuit, c'est une certitude ! Cela n'aide pas seulement votre portefeuille, cela signifie qu'une immense communauté de programmeurs/utilisateurs/analystes/étudiants développera constamment de nouvelles fonctionnalités et des packages à une vitesse qui dépasse ses concurrents ! Contrairement au Fight Club (pour ceux qui ont vu ce "vieux" film...), la première règle de R est : "parlez de R !". Si jamais vous avez une question sur la façon de mettre en œuvre quelque chose avec R, une recherche rapide Google vous mènera assurément à une réponse.

¹⁰ Pas tout à fait pour Rstudio, mais les packs qui nous intéressent le sont.

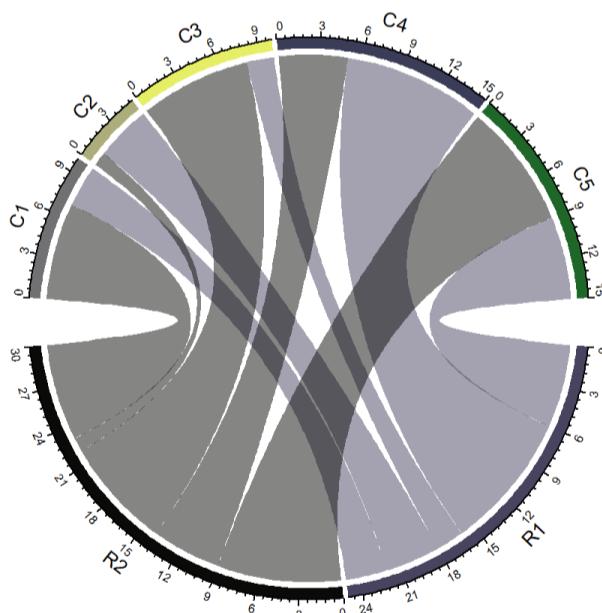
2. R domine de la tête et des épaules les autres logiciels ! Pour illustrer cela, regardez simplement les deux figures suivantes :



3. R fait parti désormais du présent et l'avenir de la programmation statistique car, de plus en plus, des interfaces entre R et les autres logiciels statisitques tels que Python sont développées¹¹.

4. R est incroyablement polyvalent ! Vous pouvez utiliser R pour tout faire, du calcul de simples statistiques récapitulatives à la réalisation de simulations complexes en passant par la cartographie, l'analyse de réseaux, la création de représentations graphiques magnifiques comme le diagramme ci-dessous (avec le package “circlize”). Si vous pouvez imaginer une tâche analytique, vous pouvez presque certainement l'implémenter dans R.

¹¹ Par exemple Jupyter permettant de coder en R et en Python dans le même notebook. Voir : <https://www.dataquest.io/blog/jupyter-notebook-tutorial/>, pour une introduction à Jupyter avec Python, et pour un exemple de notebook commun R et Python (et GAMs) : <https://towardsdatascience.com/guide-to-r-and-python-in-a-single-jupyter-notebook-ff12532eb3ba>.



5. En utilisant RStudio (ce que nous allons faire), un programme pour vous aider à écrire du code R, vous pouvez facilement et de manière transparente combiner du code R, des analyses, des représentations graphiques et du texte écrit dans des documents élégants en un seul endroit à l'aide de Sweave (R et Latex) ou RMarkdown (ce que nous allons faire aussi). En fait, j'ai traduit tout ce cours (le texte, la mise en forme, les graphiques, le code... oui, tout) dans RStudio

en utilisant Sweave (Markdown + Knitr). Avec RStudio et Sweave, au lieu d'essayer de gérer deux ou trois programmes, disons Excel, Word et (pfff...) SPSS, où vous passez la moitié de votre temps à copier, coller et formater des données, des images et des tests, vous pouvez tout faire en un place pour que rien ne soit mal lu, mal saisi ou oublié.

6. Les analyses effectuées dans R sont transparentes, facilement partageables et reproductibles ! Si vous demandez à un utilisateur SPSS/STATA comment il a effectué une analyse spécifique, il va soit A) ne pas s'en souvenir, B) essayer (nervusement) de construire sur place une procédure d'analyse qui a du sens – qui peut ou non correspondre à ce qu'il a réellement fait il y a des mois ou des années, ou C) Vous demandez ce que vous faites dans sa maison ! J'utilise moi-même principalement STATA et R, donc je parle d'expérience à ce sujet. Si vous demandez à un utilisateur R (qui utilise de bonnes techniques de programmation, celles que nous allons voir ensemble !) comment il a effectué une analyse, il devrait toujours être en mesure de vous montrer le code exact qu'il a utilisé. Bien sûr, cela ne signifie pas qu'il a utilisé l'analyse appropriée ou l'a interprétée correctement, mais avec tout le code d'origine, tout problème d'ordre technique doit être complètement transparent !



Installation de R et Rstudio

R est un logiciel de statistique distribué gratuitement par le *Comprehensive R Archive Network (CRAN)* à l'adresse suivante pour Windows : <http://cran.r-project.org/>, ou pour Mac OS : <http://cran.r-project.org/>.

Attention, l'installation varie d'un système d'exploitation à l'autre mais les fonctionnalités, le code associé restent les mêmes et la plupart des programmes sont portables d'un système à l'autre.

Contrairement à une idée reçue largement véhiculée, R et Rstudio sont deux applications/logiciels différents.

Pourtant, pour utiliser Rstudio il faut d'abord installer R.RStudio est un logiciel distinct qui fonctionne avec R pour rendre R beaucoup plus convivial avec des fonctionnalités utiles qui rendent votre programmation R plus facile et plus efficace. RStudio fonctionne sous Windows Mac et Linux et même sur le Web en utilisant RStudio Server/cloud.

Ainsi, Rstudio repose sur R qui est un langage de script conçu initialement spécifiquement pour l'analyse statistique et la gestion des données. C'est un dérivé du langage de programmation "S", mais il est gratuit et est continuellement mis à jour par les contributeurs grâce à ses nombreux packages.

L'installation de R et Rstudio est très facile (voir la fiche "tuto" en ligne ici ou la page ici), il suffit de suivre les instructions !

- **Installation de R**

Accédez au site Web de R Project et téléchargez R pour votre système d'exploitation.

- **Installation de Rstudio**

Une fois R installé, allez sur le site Web de Rstudio et cliquez sur Télécharger Rstudio et suivez les instructions de votre système d'exploitation.

Dans le cadre de ce cours nous n'utiliserons que Rstudio mais sachez que R et Rstudio sont deux logiciels différents (et oui !) bien que le second utilise le code du premier (mais pas que...) et rend surtout R beaucoup plus convivial avec des fonctionnalités utiles qui rendent la programmation en R plus facile et plus efficace.

Présentation de l'interface de Rstudio

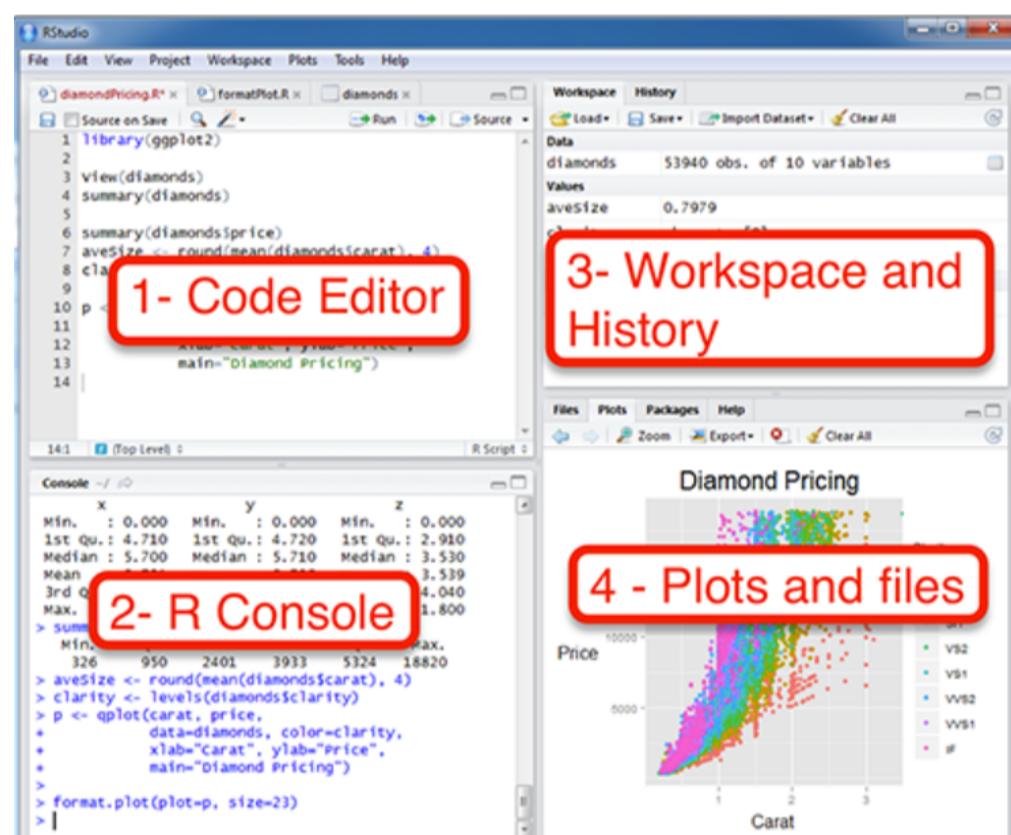
Maintenant que Rstudio est installé, ouvrez le en cliquant sur l'icône du bureau, et c'est parti !



Rstudio est un environnement de développement intégré (IDE) pour R. Il comprend une console, un éditeur de coloration syntaxique qui prend en charge l'exécution directe de code, ainsi que des outils de traçage, d'historique, de débogage et de gestion de l'espace de travail.

Lorsque vous lancez Rstudio, vous verrez les quatre fenêtres ou volets suivants.

- Le “code editor”
- La console
- Environnement de travail (“workspace”)/Historique
- Graphiques et fichiers



Le “code editor”

Le volet “code editor” (dans le quadrant en haut à gauche par défaut) est l'endroit où vous créez et modifiez des scripts écrit en R ou autres langages connus par Rstudio, comme nous le verrons avec RMarkdown. Dit autrement, il s'agit de vos collections de codes (à enregistrer bien évidemment) !

Don't worry, les scripts R ne sont que des fichiers texte avec l'extension “.R”. Lorsque vous ouvrez RStudio, il démarre automatiquement un nouveau script sans titre. Avant de commencer à taper un script R sans titre, vous devez toujours enregistrer le fichier sous un nouveau nom de fichier (comme “*Pirates_analysis.R*”). De cette façon, si quelque chose se bloque sur votre ordinateur pendant que vous travaillez, Rstudio conservera votre code et l'ouvrira automatiquement à la réouverture de Rstudio.

```
1 # Date: 22 September, 2016
2 # Title: Exploring the pirates dataset
3 # Name: Nathaniel Phillips
4
5 # Load the yarrr library
6 library(yarr)
7
8 # Basic info on the pirates dataframe
9 str(pirates)
10 head(pirates)
11
12 # Number of pirate sexes
13 table(pirates$sex)
14
15 # Correlation between height and weight
16 cor(pirates$height, pirates$weight)
17 |
```

Vous remarquerez que lorsque vous saisissez du code dans un script dans le panneau d'édition (dit aussi “source”), R n'évalue pas réellement le code pendant que vous tapez. Pour que R évalue réellement votre code, vous devez d'abord “envoyer” le code à la console (nous en parlerons dans la section suivante).

Il existe de nombreuses façons d'envoyer votre code depuis la source vers la console. Le moyen le plus lent est de copier et coller. Un moyen plus rapide est de mettre en évidence le code que vous souhaitez évaluer et de cliquer sur le bouton “Exécuter” en haut à droite de l'éditeur (en cliquant sur la commande en jaune “run”). Vous pouvez également utiliser la touche de raccourci Commande + Retour sur Mac ou Ctrl + Entrée sur PC pour envoyer tout le code en surbrillance à la console.

```
183 conf_USD <- ggplot(COVID19_USCOMPARE_R,
184   aes(x=date,
185     y=Rate_d,
186     color=state)) +
187   geom_line()+
188   geom_point(size=1)+
189   theme(legend.position="top")
190 conf_USD + labs(x = "Date", y = "Death rate 08/13/2020", color = "U
191 |
192
193
194
195
196 gather(COVID19_USCOMPARE_R,
197   value = "LOG_confirmed",
198   key = "state")
199 |
```

La console : le cœur de Rstudio !

La console est le cœur de Rstudio. C'est là que Rstudio évalue réellement le code. Au début de la console, vous verrez le texte comme ci-dessous avec la fin, une flèche bleue. C'est une invitation qui vous indique que Rstudio (même chose sur R !) est prêt pour un nouveau code.

```

Console ~/Desktop/asdf/ ↵

R version 3.3.1 (2016-06-21) -- "Bug in Your Hair"
Copyright (C) 2016 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin13.4.0 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> |

```

Vous pouvez taper du code directement dans la console après l’invitation (“>”) et obtenir une réponse immédiate. Par exemple, si vous tapez $1 + 1$ dans la console et appuyez sur Entrée [pratique], vous verrez que Rstudio donne immédiatement une sortie de 2^{12} , comme par magie, Rstudio sais compter !

¹² Ne vous inquiétez pas pour le [1] devant le résultat pour le moment, nous y reviendrons plus tard.

```

1 + 1
## [1] 2

```

Chaque instruction (par exemple “ $1 + 1$ ”) écrite dans la console doit être validée en appuyant sur Entrée pour être exécutée. Si l’instruction est “correcte” (reconnu par Rstudio en fonction des packages chargés, *c.f.* onglet “packages”), Rstudio redonne la main après l’exécution et indique “>”, vous invitant alors à recommencer ! Au contraire, si l’inscription est incorrecte ([pratique] essayez par exemple : “ $1 +$ ”), alors Rstudio retourne le signe “+”, vous invitant à compléter l’inscription (il manque quelque chose). On sort alors du problème en complétant simplement à la suite du plus ([pratique] ajoutez un “1” à la suite du “+”). On peut aussi récupérer la main en tapant “Ctrl + c” ou “Echap”. Si l’inscription est erronée, alors un message d’erreur apparaît ([pratique] essayez par exemple : “`mea(x)`”)

Ensuite, tapez le même code dans l’éditeur de code (source), puis envoyez le code à la console en mettant en surbrillance le code et en cliquant sur le bouton Exécuter "dans le coin supérieur droit de la Fenêtre source Vous pouvez également utiliser le raccourci clavier sur Mac ou sous Windows [pratique].

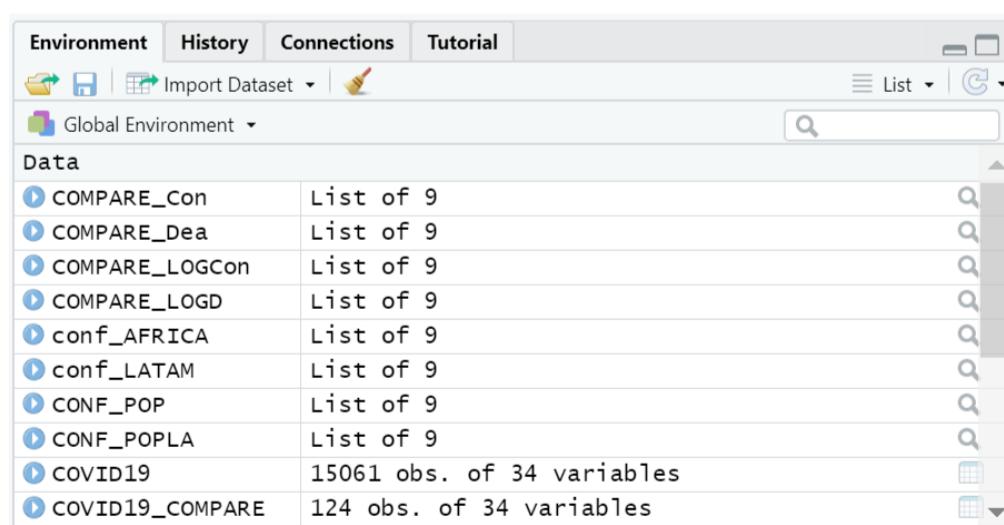
Comme vous pouvez le voir, vous pouvez exécuter du code soit en l’exécutant à partir de l’éditeur, soit en le tapant directement dans la console. Cependant, 99% du temps,

vous devriez utiliser l'éditeur plutôt que la console. La raison en est simple : si vous tapez du code dans la console, il ne sera pas enregistré (bien que vous puissiez consulter l'historique de vos commandes, on y reviendra). Et si vous faites une erreur en tapant du code dans la console, vous devrez tout retaper à nouveau. Au lieu de cela, il vaut mieux écrire tout votre code dans l'éditeur en enregistrant régulièrement votre travail. Lorsque vous êtes prêt à exécuter du code, vous pouvez ensuite l'envoyer, “Exécuter”/“Run”, à la console.

A retenir : essayez toujours d'écrire la majeure partie de votre code dans l'éditeur de code et non directement dans la console.
Réservez la console pour déboguer ou faire des analyses rapides.

L'environnement de travail et l'historique

L'onglet environnement (“workspace”) situé dans le cadran en haut à droite par défaut affiche les noms de tous les objets de données (tels que les vecteurs, les matrices et les dataframes) que vous avez définis dans votre session actuelle. Par exemple, nous pouvons définir un vecteur “addition” avec l'opération précédente [pratique]. Vous pouvez également afficher des informations telles que le nombre d'observations et de lignes dans les objets de données. L'onglet a également quelques actions cliquables comme Importer un jeu de données qui ouvrira une interface utilisateur graphique (GUI) pour les données importantes dans Rstudio. Cependant, ce menu s'oublie rapidement et on finit par ne plus le regarder.



The screenshot shows the RStudio interface with the "Environment" tab selected. The top navigation bar includes tabs for "Environment", "History", "Connections", and "Tutorial". Below the tabs, there are buttons for "Import Dataset" and "New File". On the right side, there are dropdown menus for "List" and "C". A search bar is also present. The main area is titled "Global Environment" and lists various objects:

Object	Type
COMPARE_Con	List of 9
COMPARE_Dea	List of 9
COMPARE_LOGCon	List of 9
COMPARE_LOGD	List of 9
conf_AFRICA	List of 9
conf_LATAM	List of 9
CONF_POP	List of 9
CONF_POPLA	List of 9
COVID19	15061 obs. of 34 variables
COVID19_COMPARE	124 obs. of 34 variables

L'onglet historique de ce panneau vous montre simplement un historique de tout le code que vous avez précédemment évalué dans la console. Pour être honnête, je ne regarde jamais ça. En fait, je n'avais même pas réalisé qu'il était là avant de commencer à écrire ce cours !

```

COVID19$death_pop = COVID19$death_pop_SPAIN + COVID19$death_pop_IT...
COVID19<-select(COVID19, -death_pop_SPAIN, -death_pop_ITALY, -deat...
COVID19$LOG_Death = log10(COVID19$Deaths)
COVID19$LOG_Death[is.na(COVID19$LOG_Death)] = 0
COVID19$LOG_Death[is.infinite(COVID19$LOG_Death)] = 0
COVID19_LATAM = subset(COVID19, COVID19$Country == "Colombia" | co...
COVID19_LATAM_E = subset(COVID19, COVID19$Country == "Colombia" | ...
COVID19_LATAM_WK = subset(COVID19_LATAM_E, COVID19_LATAM_E>Date_ok...
COVID19_COMPARE = subset(COVID19, COVID19$Country == "Italy" | cov...
COVID19_COMPARE = subset(COVID19_COMPARE, COVID19_COMPARE>Date_ok ...
COVID19_COMPARE_2 = subset(COVID19, COVID19$Country == "Canada" | ...
COVID19_COMPARE_2 = subset(COVID19_COMPARE_2, COVID19_COMPARE_2$Da...
gather(COVID19_LATAM_WK,
value = "Confirmed",

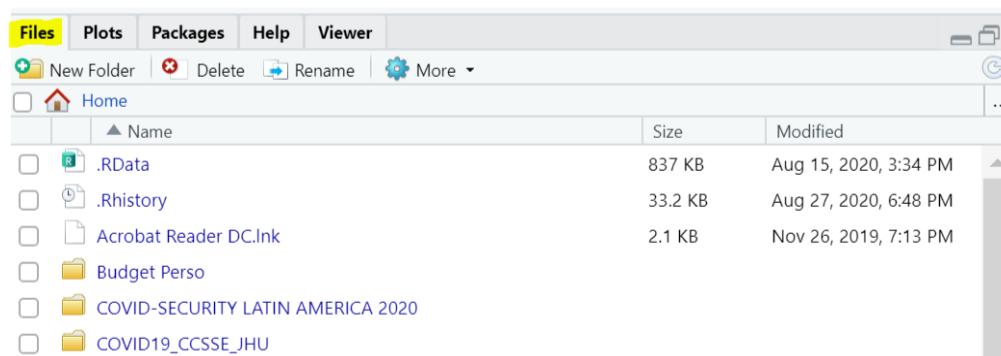
```

À mesure que vous vous familiariserez avec Rstudio, vous trouverez peut-être le panneau Environnement/Historique utile. Mais pour l'instant, vous pouvez simplement l'ignorer. Si vous souhaitez désencombrer votre écran, vous pouvez même simplement minimiser la fenêtre en cliquant sur le bouton Réduire en haut à droite du panneau [pratique].

Les sorties graphiques et les fichiers (packages, help etc...)

Le panneau Fichiers/Graphiques/Packages/Aide vous montre de nombreuses informations utiles. Passons en revue chaque onglet en détail:

- *Fichiers* : le panneau des fichiers vous donne accès au répertoire de fichiers sur votre disque dur. Une fonctionnalité intéressante du panneau “Fichiers” est que vous pouvez l'utiliser pour définir votre répertoire de travail – une fois que vous accédez à un dossier dans lequel vous souhaitez lire et enregistrer des fichiers, cliquez sur “Plus”, puis sur “Définir comme répertoire de travail”.



Par défaut, les données de travail et les commandes utilisées seront enregistrées à l'emplacement où Rstudio a été ouvert, en l'occurrence, là où il a été installé... Mais il est souhaitable de modifier le répertoire de travail avec la fonction “`setwd()`” ou en allant dans “Session” puis dans “Set Working Directory”.

Pour quitter une session sur Rstudio, vous pouvez choisir “sortir” et répondre positivement à la question suivante :

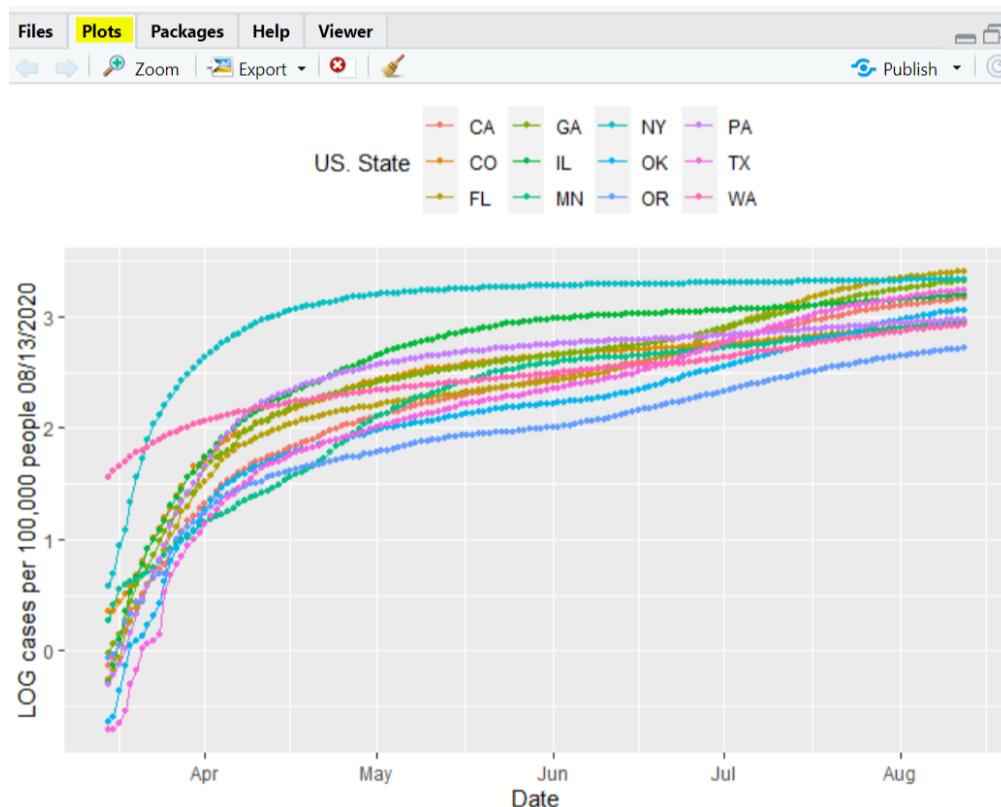
```

Environment History Connections Tutorial
To Console To Source
COVID19$death_pop = COVID19$death_pop_SPAIN + COVID19$death_pop_IT...
COVID19<-select(COVID19, -death_pop_SPAIN, -death_pop_ITALY, -deat...
COVID19$LOG_Death = log10(COVID19$Deaths)
COVID19$LOG_Death[is.na(COVID19$LOG_Death)] = 0
COVID19$LOG_Death[is.infinite(COVID19$LOG_Death)] = 0
COVID19_LATAM = subset(COVID19, COVID19$Country == "Colombia" | co...
COVID19_LATAM_E = subset(COVID19, COVID19$Country == "Colombia" | ...
COVID19_LATAM_WK = subset(COVID19_LATAM_E, COVID19_LATAM_E$Date_ok...
COVID19_COMPARE = subset(COVID19, COVID19$country == "Italy" | cov...
COVID19_COMPARE = subset(COVID19_COMPARE, COVID19_COMPARE>Date_ok ...
COVID19_COMPARE_2 = subset(COVID19, COVID19$Country == "Canada" | ...
COVID19_COMPARE_2 = subset(COVID19_COMPARE_2, COVID19_COMPARE_2$Da...
gather(COVID19_LATAM_WK,
value = "Confirmed",

```

Une icône R sera alors créée dans le répertoire de travail (“working directory” ou “wd”). En redémarrant Rstudio, la session s’ouvrira automatiquement avec son répertoire concerné et les objets créés !

- *Graphiques* : le panneau “Plots” (pas de grande surprise), montre toutes vos sorties graphiques. Il existe des boutons pour ouvrir le tracé dans une fenêtre séparée et exporter la sortie au format pdf ou jpeg (bien que vous puissiez également le faire avec du code en utilisant les fonctions “pdf()” ou “jpeg()”).



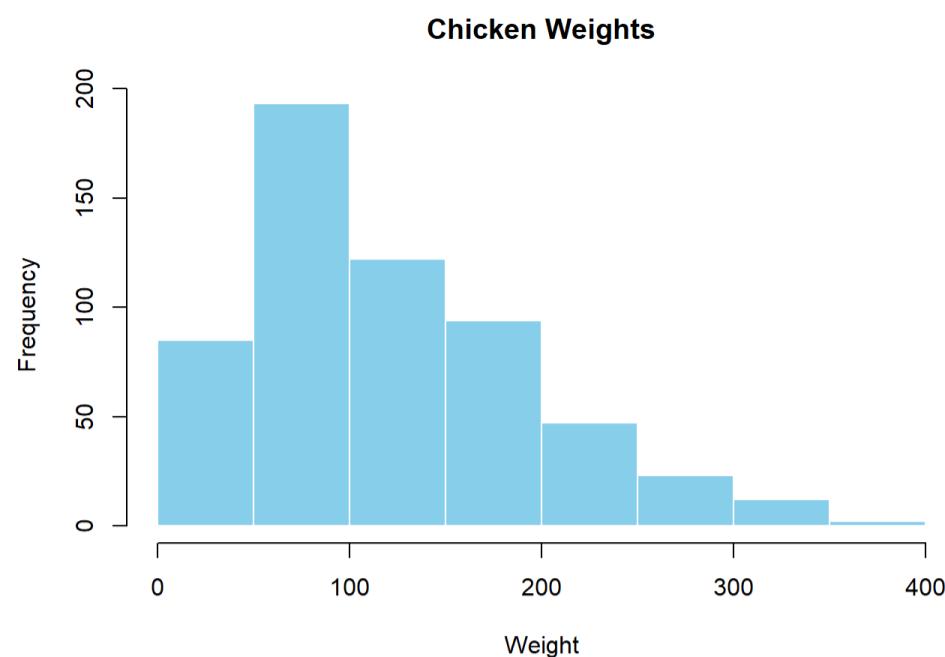
Voyons comment les graphiques sont affichés dans le panneau “Graphiques”. Exécutez le code à droite pour afficher un histogramme des poids des poulets stockés dans l’ensemble de données “ChickWeight”. Lorsque vous le faites, vous devriez voir un histogramme similaire à celui de la figure ci-dessous apparaître dans le panneau graphique.

```

data("ChickWeight")

hist(x = ChickWeight$weight,
      main = "Chicken Weights",
      xlab = "Weight",
      col = "skyblue",
      border = "white")

```



- *Packages* : affiche une liste de tous les packages Rstudio installés sur votre disque dur et indique s'ils sont actuellement chargés ou non. Les packages chargés dans la session en cours sont vérifiés tandis que ceux qui sont installés mais pas encore chargés ne le sont pas. Nous aborderons les packages plus en détail dans la section suivante.

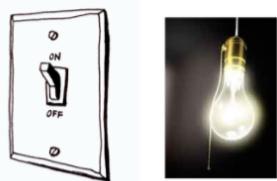
Name	Description	Version	Status
abind	Combine Multidimensional Arrays	1.4-5	Green (dot)
acepack	ACE and AVAS for Selecting Multiple Regression Transformations	1.4.1	Green (dot)
AER	Applied Econometrics with R	1.2-9	Green (dot)
askpass	Safe Password Entry for R, Git, and SSH	1.1	Green (dot)
assertthat	Easy Pre and Post Assertions	0.2.1	Green (dot)
backports	Reimplementations of Functions Introduced Since R-3.0.0	1.1.7	Green (dot)
base64enc	Tools for base64 encoding	0.1-3	Green (dot)
BayesFactor	Computation of Bayes Factors for Common Designs	0.9.12-4.2	Green (dot)
BH	Boost C++ Header Files	1.72.0-3	Green (dot)
bitops	Bitwise Operations	1.0-6	Green (dot)
blob	A Simple S3 Class for Representing Vectors of Binary Data ('BLOBS')	1.2.1	Green (dot)
blockmodeling	Generalized and Classical Blockmodeling of Valued Networks	0.3.6	Green (dot)
BMA	Bayesian Model Averaging	3.18.12	Green (dot)
bookdown	Authoring Books and Technical Documents with R Markdown	0.20	Green (dot)
brew	Templating Framework for Report Generation	1.0-6	Green (dot)
broom	Convert Statistical Analysis Objects into Tidy Tibbles	0.5.6	Green (dot)
callr	Call R from R	3.4.3	Green (dot)
car	Companion to Applied Regression	3.0-8	Green (dot)
carData	Companion to Applied Regression Data Sets	3.0-4	Green (dot)

Lorsque vous téléchargez et installez Rstudio pour la première fois, vous installez le logiciel de base Rstudio qui contiendra la plupart des fonctions que vous utiliserez quotidiennement comme “`mean()`” et “`hist()`”. Cependant, seules les fonctions écrites par les auteurs originaux du langage Rstudio apparaîtront ici. Si vous souhaitez accéder aux données et aux programmes écrits par d'autres personnes, vous devrez les installer sous forme de “packages”. Un package R est simplement un ensemble de données, des fonctions, aux menus d'aide, aux vignettes (exemples), stockées dans un pack que l'on peut ajouter à notre version de base de Rstudio.

Installing a package
`install.packages('my.package')`



Loading a package
`library('mypackage')`



Un “package” est comme une ampoule. Pour l'utiliser, vous devez d'abord l'avoir à la maison, c'est évident ! Sur Rstudio, cela signifiera l'avoir sur notre version de Rstudio en l'installant. Une fois que vous avez installé un package, vous n'avez plus jamais besoin de l'installer à nouveau (attention au restarting de session !). Cependant, chaque fois que vous souhaitez utiliser le package, vous devez l'activer en le chargeant. Voici comment procéder.

- *Installer un nouveau package*

Installer un package signifie simplement télécharger le code du package sur votre version de Rstudio. Il existe deux manières principales d'installer de nouveaux packages. La première méthode, et la plus courante, consiste à les télécharger à partir du CRAN. Le CRAN est LE référentiel central des packages R utilisable avec Rstudio. Pour installer un nouveau package R à partir du CRAN, vous pouvez simplement exécuter le code “`install.packages("nom")`”, dans lequel le “nom” est le nom du package. Par exemple, pour télécharger le package dénommé “yarr”, qui contient plusieurs ensembles de données et fonctions, vous devez exécuter ce qui suit [pratique] :

```
install.packages("yarr")
```

Lorsque vous exécutez la commande “`install.packages("nom")`”, Rstudio téléchargera le package depuis CRAN (on le voit dans la console !). Si tout fonctionne, vous devriez voir des informations sur l'endroit où le package est téléchargé, en plus d'une barre de progression.

Comme pour commander une ampoule, une fois que vous avez installé un package sur votre ordinateur, vous n'avez plus jamais besoin de l'installer à nouveau (à moins que vous ne souhaitiez essayer d'installer une nouvelle version du package). Cependant, chaque fois que vous souhaitez l'utiliser, vous devez l'activer en le chargeant.

- *Chargement d'un package*

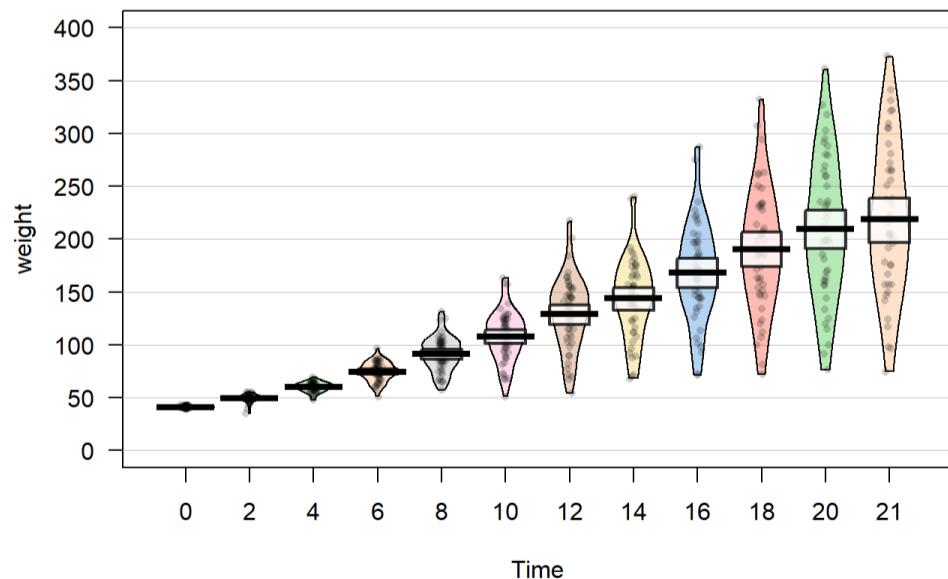
Une fois que vous avez installé un package, il est sur votre ordinateur (votre version de Rstudio). Cependant, ce n'est pas parce qu'il se trouve sur votre ordinateur que Rstudio est prêt à l'utiliser. Si vous souhaitez utiliser quelque chose, comme une fonction ou un ensemble de données, à partir d'un package, vous devez toujours charger le package dans votre session Rstudio. Tout comme une ampoule, vous devez l'allumer pour l'utiliser !

Pour charger un package, vous utilisez la fonction “library()”. Par exemple, maintenant que nous avons installé le package “yarr”, nous pouvons le charger avec la “library("yarr")” :

```
library("yarr")
```

Maintenant que vous avez chargé le package “yarr”, vous pouvez utiliser l'une de ses fonctions ! L'une des fonctions les plus intéressantes de ce paquet s'appelle “pirateplot()”. Plutôt que de vous dire ce qu'est un graphique de pirates, faisons-en un. Exécutez le morceau de code suivant pour créer votre propre pirate plot. Ne vous inquiétez pas des détails du code ci-dessous, vous en apprendrez plus sur le fonctionnement de tout cela plus tard. Pour l'instant, exécutez simplement le code et émerveillez-vous devant votre graphique !

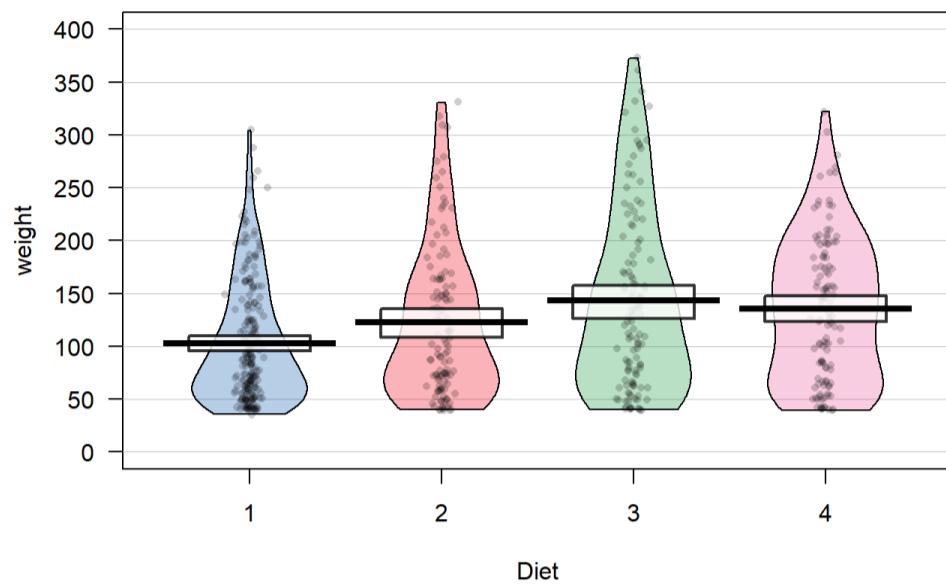
```
pirateplot(formula = weight ~ Time,
           data = ChickWeight,
           pal = "xmen")
```



Il existe une manière dans Rstudio de charger temporairement un package sans utiliser la fonction “library()”. Pour ce faire, vous pouvez simplement utiliser la notation “package::function”. Cette notation indique simplement à Rstudio de charger le package uniquement

pour ce morceau de code. Par exemple, je pourrais utiliser la fonction `pirateplot` du package “`yarr`” comme suit :

```
yarrr::pirateplot(formula = weight ~ Diet,
                   data = ChickWeight)
```



- *Aide* : comme tout bon logiciel, Rstudio comprend un menu d'aide pour les fonctions R. Vous pouvez soit taper le nom d'une fonction dans la fenêtre de recherche, soit utiliser le code pour rechercher une fonction avec le nom.

```
?hist # Comment fonctionne La commande histogramme ?
```

```
?t.test # Celle du T-test ?
```

ou encore :

```
help(hist) # Comment fonctionne La commande histogramme  
help(t.test) # Celle du T-test ?
```

Basic mathematics

Maintenant que Rstudio a été installé avec succès sur votre ordinateur, il est temps de se familiariser avec le logiciel.

Nous l'avons déjà vu, R peut-être considérer comme une calculatrice, extrêmement puissante, et commencez à faire des choses avec lesquelles vous utiliseriez autrement votre calculatrice Casio FX-92 Collège ou TI 83 ! Nous l'avons vu également, un “#” peut apparaître dans un code informatique. Avec ce symbole nous indiquons à Rstudio un commentaire (comme sur Python), ce symbole “#” est destiné à dire le logiciel de ne pas l'incorporer. Dans le cas contraire, nous verrions continuellement des messages d'erreur lorsque nous essayions d'étiqueter quelque chose avec un commentaire.

Examinons désormais quelques petits exemples des opérations mathématiques de base dans Rstudio [pratique].

- **Additionner** : 1755 et 2421
- **Soustraire** : 9567 à 54902
- **Multiplier** : 44 par 126
- **Diviser** : 8904 par 6

```
1755 + 2421
```

```
## [1] 4176
```

```
54902 - 9567
```

```
## [1] 45335
```

```
44 * 126
```

```
## [1] 5544
```

```
8904 / 6
```

```
## [1] 1484
```

Sachez que Rstudio ne reconnaît pas toujours la façon dont nous écrivons communément les mathématiques. Par exemple, si nous voulions multiplier 2 et 3 ensemble, sur papier nous pourrions écrire $(2)(3) = 6$. Cependant, dans Rstudio, voici ce qui se passe lorsque nous essayons ceci:

```
(2)(3)
```

En bref, le programme n'a pas pu exécuter ce que nous pensions être manière d'écrire la multiplication. Tel est le processus d'apprentissage d'un langage informatique, pour apprendre ce que le programme reconnaîtra ou non il faut essayer ou se renseigner¹³ !

¹³ RTFM !

Mais Rstudio est beaucoup mieux que votre calculatrice préférée...car il respectera automatiquement l'ordre des

opérations :

```
2 * 3 + 5
```

```
## [1] 11
```

Notez que Rstudio a d'abord multiplié 2 et 3, puis ajouté 5 (étonnant !). C'est la même chose que nous devrions faire manuellement, donc nous pouvons voir que Rstudio connaît très bien le fameux "CEDMAS" (crochets, puis exposants, puis division / multiplication, puis addition / soustraction). Nous pouvons voir rapidement comment Rstudio peut gérer des séries d'opérations arithmétiques plus compliquées :

```
((2 * 3) + 6 - (2 / 3 - 11))/6^5
```

```
## [1] 0.002872085
```

La dernière partie de l'expression ci-dessus, 6^5 , est bien évidemment l'exponentiel, c'est-à-dire "6 élevé à l'exposant 5" ou 6^5 (confirmez vous-même sur votre smartphone qu'il vaut 7776 !).

Rstudio peut bien sûr gérer des mathématiques beaucoup plus compliquées que les précédentes. Par exemple, il peut calculer des logarithmes, des exponentiels, des dérivés et des intégrales, pour n'en nommer que quelques-uns. Par exemple, vous vous souvenez peut-être que le logarithme et les fonctions exponentielles sont inverses l'une de l'autre. On peut démontrer ça facilement avec Rstudio :

```
log(10)
```

```
## [1] 2.302585
```

```
exp(2.302585)
```

```
## [1] 9.999999
```

A l'erreur d'arrondi prêt, nous voyons $e^{2.302585}$ est égal à $\log(10)$. nous pouvons arrondir le nombre à l'entier le plus proche en utilisant "round()" :

```
round(exp(2.302585))
```

```
## [1] 10
```

Autrement dit, le logarithme (naturel) d'un nombre en base e est l'exposant auquel nous élevons e pour obtenir ce nombre (ouf !) !

Nous utiliserons cette relation inverse entre les logs et les exposants lorsque nous étudierons la régression logistique plus loin dans ce cours (*c.f. Séance 7*). Si vous êtes familier avec les dérivés (si vous ne l'êtes pas, c'est le moment de les revoir !), alors vous savez que la dérivée de la fonction

suivante, $f(x) = 5x + 3$ est égale à 5. Autrement dit, $f'(x) = 5$. Mais Rstudio peut calculer cela pour nous :

```
D(expression(5*x + 3), "x")  
## [1] 5
```

Vous pouvez également contrôler les décimales grâce à Rstudio. Par exemple :

```
print(pi, digits = 3)  
## [1] 3.14  
print(pi, digits = 20)  
## [1] 3.1415926535897931
```

Attention : la fonction “*digits*” correspond au nombre de chiffres renvoyés par Rstudio dans le résultat et non le nombre de décimales !

Rstudio permet également de vérifier la relation entre les membres d'une équation :

```
1/2 * 2 == 1  
## [1] TRUE
```

Ce qui précède est évidemment vrai comme l'indique la réponse booléenne reçue, et Rstudio le confirme.

Cependant, $1/2$ n'a qu'une seule décimale et, par conséquent, 0,5 multiplié par 2 est facilement être égal à 1.

Cependant, si l'expression avait était ce qui suit, nous aurions obtenu une réponse différente :

```
sqrt(2)^2 == 2  
## [1] FALSE
```

Bien sûr, nous nous serions attendus à ce que l'énoncé soit vrai, car le carré de la racine carrée (“`sqrt()`”) de 2 devrait éliminer cette racine carrée et renvoyer simplement le nombre 2 !

Cependant, la raison pour laquelle l'instruction est fausse est que le nombre “`sqrt(2)`” est une approximation et n'a pas de représentation décimale finie. Par conséquent, la déclaration d'égalité mathématique ne peut pas être vraie (mathématiquement) !

Nous pourrions résoudre ce problème à l'aide du package “`dplyr`” (très utile !) et sa fonction “`near()`” ou “`approximation`” :

```
install.packages("dplyr")  
  
library(dplyr)  
  
near(sqrt(2)^2, 2)
```

Ne vous laissez pas berner non plus lorsque Rstudio rapporte une virgule décimale finie à $\sqrt{2}$. Le numéro suivant n'a pas que 6 positions à droite du virgule, Rstudio arrondit simplement le nombre à six décimales.

Notons que des valeurs spéciales sont parfois renvoyées par Rstudio : “Inf” pour l'infini et “NaN” pour “Not a Number”, valeurs résultant de problèmes de calcul. Par exemple :

```
exp(1e10)
```

```
## [1] Inf
```

Ou :

```
log(-2)
```

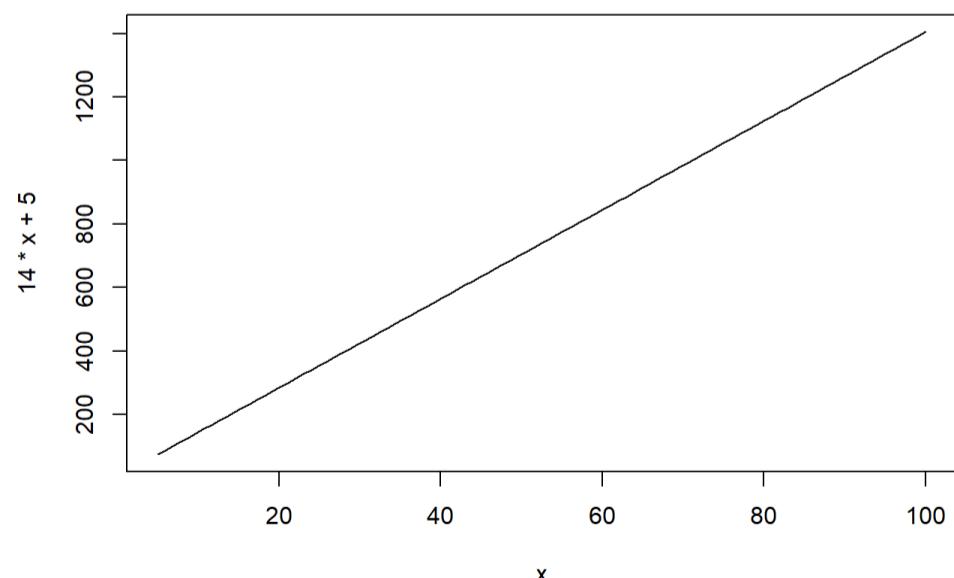
```
## [1] NaN
```

Fonctions et courbes

En parlant de mathématiques, il nous incombe de passer brièvement en revue les types de fonctions que R peut représenter. La fonction “curve()” est particulièrement utile pour cela.

On entre d'abord une fonction linéaire définie par la variable “x”, puis on spécifie la plage de valeurs du domaine sur laquelle la fonction sera représentée. Pour la fonction $f(x) = 14x + 5$ sur l'intervalle allant de $x = 5$ à $x = 100$ (autrement dit, $\forall x \in [5, 100]$) :

```
curve(14*x + 5, 5, 100)
```

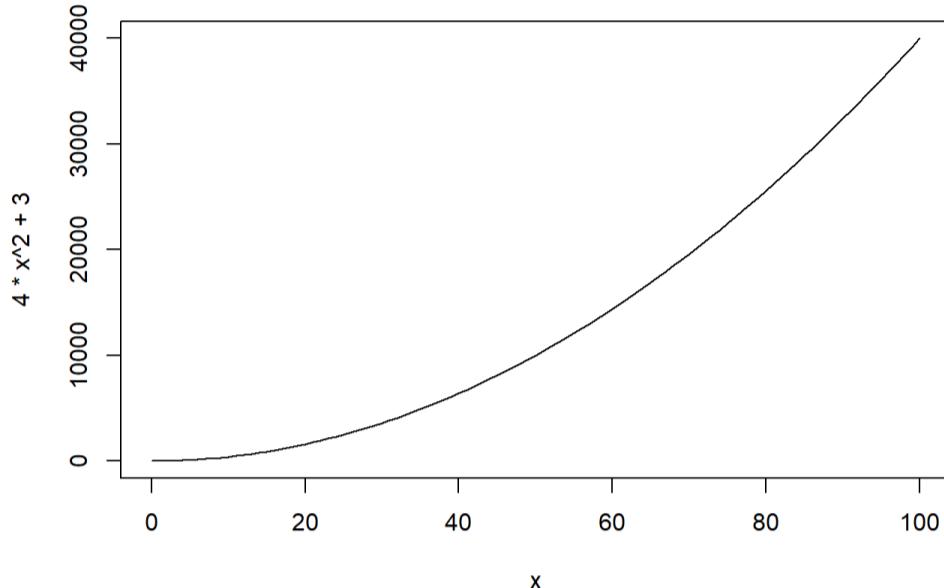


Pour un polynôme défini par la variable “x”, on spécifie également la plage de valeurs pour le domaine. Pour rappel : le domaine de la fonction est l'ensemble des valeurs possibles qui peuvent être saisies dans la fonction. Par exemple, considérons le polynôme $f(x) = 4x^2 + 3$.

C'est une fonction quadratique, que nous pouvons facilement écrire dans R : $4 * x^2 + 3$, rien de plus simple !

Maintenant, spécifions la plage de représentation des valeurs pour le domaine de définition afin que la fonction ne prenne que des valeurs positives de 0 à 100 (pour rappel : $\forall x \in [0; 100]$). Ainsi :

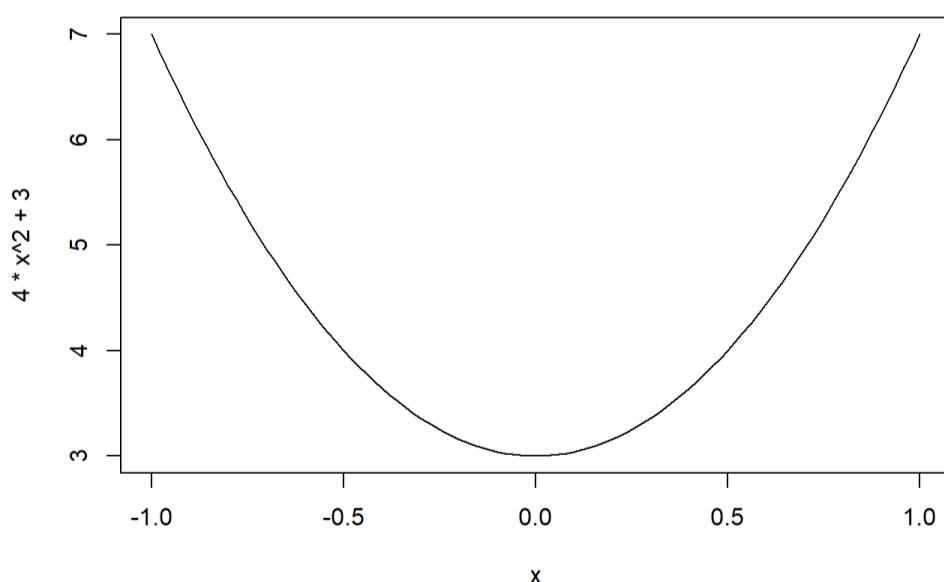
```
curve(4*x^2+3, 0, 100)
```



À première vue, la courbe ne ressemble pas beaucoup à une parabole... Pourquoi ?

Rappel : cela est dû à la façon dont nous avons spécifié la plage de représentation sur le domaine de définition : uniquement $x \geq 0$! Si nous réduisons la plage de -1 à 1 , nous obtenons quelque chose de plus juste :

```
curve(4*x^2+3, -1, 1)
```



La courbe ressemble maintenant à la parabole avec laquelle nous sommes plus familiers. Les polynômes, comme nous le verrons plus loin dans ce cours, peuvent être utilisés pour ajuster les données dans la régression (*c.f.* Séance 6)¹⁴.

¹⁴ Nous apprendrons aussi à faire des choix. En effet, Si une droite (fonction linéaire) constitue le meilleur ajustement pour un nuage de points, alors la méthode est celle de la

Si un polynôme différent est théorisé pour s'adapter aux données, alors la régression polynomiale devient une option. Comme dans l'exemple présenté ci-dessous dans lequel on voit un jeu de données distribué sur deux variables "x" et "y". Dans ce cas, un polynôme de degré 3 ("polynôme cubique") est un ajustement raisonnable à la forme prise par la relation entre "x" et "y" :

```
x <- runif(300, min=-10, max=10)
y <- 0.1*x^3 - 0.5 * x^2 - x + 10 + rnorm(length(x),0,8)

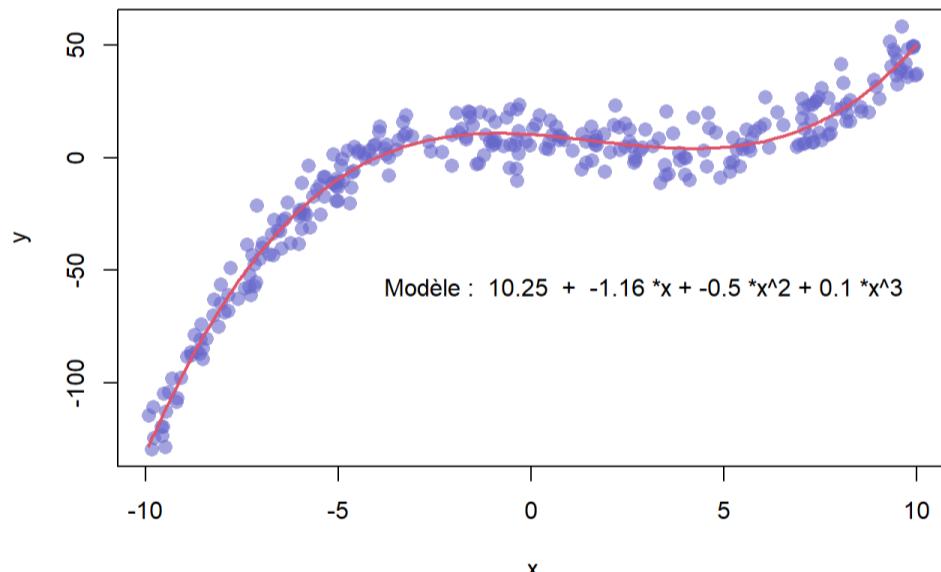
# graphique de y en x :
plot(x,y,col=rgb(0.4,0.4,0.8,0.6),pch=16 , cex=1.3)

# Trouver La fonction polynomiale ?

model <- lm(y ~ x + I(x^2) + I(x^3))

# Pour chaque valeur de x, nous pouvons avoir la valeur es:
myPredict <- predict( model )
ix <- sort(x,index.return=T)$ix
lines(x[ix], myPredict[ix], col=2, lwd=2 )

# On peut enfin "customiser" un peu le graphique en question
coeff <- round(model$coefficients , 2)
text(3, -70 , paste("Modèle : ",coeff[1] , " + " , coeff[2]
```



C'est ce que nous apprendrons à faire dans ce cours : choisir le meilleur modèle à appliquer sur nos données !

régression linéaire : rappelez-vous qu'une droite est simplement un polynôme de degré 1 ! Mais nous verrons aussi que ce n'est pas toujours le cas...

Vecteurs et scalaires

Une compréhension de base des vecteurs et des matrices est essentielle pour comprendre les statistiques appliquées et l'analyse des données. Heureusement, la plupart des concepts sont très intuitifs et peut être expliqué à l'aide de démonstrations dans Rstudio.

Un vecteur peut être défini soit comme une liste de nombres, définition informatique du terme, soit comme un segment de ligne de magnitude et de direction déterminées, définition physique du terme. Lorsque nous définissons généralement un vecteur en statistiques appliquées ou en analyse des données, nous **agrégeons simplement une série d'observations sur une variable**. Techniquement, un vecteur peut également être constitué uniquement d'un **nombre unique**, auquel cas il est généralement appelé **scalaire**. Il y a différentes manières de créer un vecteur dans Rstudio.

Couramment, cela consistera simplement à lister les nom que vous souhaitez donner au vecteur, suivi de la suite d'éléments attribués à ce vecteur. Par exemple, ce qui suit attribue le nom “x” à la liste “1, 3, 5, 7, 9” :

```
x <- c(1, 3, 5, 7, 9)
```

Maintenant que le vecteur est enregistrer sous son nom “x” (regardez dans votre environnement de travail !), nous pouvons l'utiliser en l'appelant par son nom :

```
x  
## [1] 1 3 5 7 9
```

On peut également vérifier qu'il s'agit bien d'un vecteur (Wickham, 2014)¹⁵ :

```
is.atomic(x) || is.list(x)  
## [1] TRUE
```

On aurait aussi pu aussi, si on l'avait voulu, nommer le vecteur à la fin plutôt qu'au début :

```
c(1, 3, 5, 7, 9) -> x  
  
x  
## [1] 1 3 5 7 9
```

Ou encore, nous aurions pu rendre plus explicite le fait que nous “attribuons” le nom “x” à la liste d'éléments :

```
assign("x", c(1, 3, 5, 7, 9))  
  
x  
## [1] 1 3 5 7 9
```

Ainsi, plusieurs manières permettent de construire des vecteurs mais, précision importante, nous n'utiliserons désormais que la première qui est la plus commune sur Rstudio, par convention. Quelle que soit la manière dont nous choisissons, notons que le vecteur “x” comporte 5 éléments. On peut utiliser la fonction “length()” dans Rstudio pour nous le dire :

¹⁵ Wickham, H. (2014). *Advanced R*. New York: Chapman & Hall/CRC The R Series.

```
length(x)
```

```
## [1] 5
```

Nous pouvons ensuite **sélectionner des éléments** au sein d'un vecteur. Le vecteur de sélection “`x[index]`” peut être un vecteur d'entiers positifs, d'entiers négatifs ou logiques. La sélection la plus naturelle est celle des vecteurs entiers positifs. On parle alors d'extraire des éléments d'un vecteur par **indexation**. Pour exemple, pour le vecteur “`x`”, supposons que l'on souhaite extraire les éléments “2” et “4”, qui correspondent aux nombres “3” et “7” dans le vecteur “`x`” :

```
x[c(2, 4)]
```

```
## [1] 3 7
```

Les entiers sont des éléments à sélectionner et doivent être compris entre 1 et la longueur du vecteur “`x`” :

```
x[3] # sélectionne Le 3ème élément du vecteur x
```

```
## [1] 5
```

```
x[2:4] # Du 2ème au 4ème élément de x
```

```
## [1] 3 5 7
```

```
x[c(4, 4, 1:2)] # Les éléments 4, 4, 1 et 2
```

```
## [1] 7 7 1 3
```

```
x[5:1] # Les éléments 5, 4, ..., 1
```

```
## [1] 9 7 5 3 1
```

Supposons désormais que nous voulions **ajouter un nouvel élément** à notre vecteur dans une position particulière dans la liste, comme ajouter un 8 entre les nombres 7 et 9. Nous pourrions accomplir ceci en utilisant ce qui suit :

```
x <- c(x[1:4], 8, x[5])
```

```
x
```

```
## [1] 1 3 5 7 8 9
```

Le vecteur reste inchangé pour les éléments allant de 1 jusqu'à 4, c'est ce qu'indique le premier terme dans la parenthèse (“`x[1:4]`”). Cependant, nous avons bien ajouté le nombre “8” en cinquième position dans le vecteur, comme indiqué par “`x[5]`”.

Nous pouvons également **exclure des éléments** du vecteur. Supposons que nous souhaitons exclure le dernier élément de “`x`”, qui correspond au nombre “9”. Notez que “9” est en sixième position, nous souhaitons donc supprimer le sixième élément :

```
x[-6]
```

```
## [1] 1 3 5 7 8
```

Si nous voulions désormais supprimer les éléments “2” à “4” inclus, nous coderions :

```
x[-2:-4]  
## [1] 1 8 9
```

Les objets : les éléments qui composent les vecteurs

Rstudio admet plusieurs types d'objets :

- Null (empty object): NULL
- Logical (Boolean): TRUE, FALSE or T, F
- Numeric (real number): 1, 2.3222, pi, 1e-10
- Complex (complex number): 2+0i, 2i
- Character (chain of characters): 'hi', "K"

Ainsi, nous pouvons définir un vecteur “x” comprenant ces différents types d’éléments¹⁶ :

```
x1 <- c(NULL, NULL, NULL)  
  
x2 <- c(TRUE, FALSE, TRUE)  
  
x3 <- c(0, 2.78, pi)  
  
x4 <- c ("A", "B", "C")  
  
x1  
## NULL  
  
x2  
## [1] TRUE FALSE TRUE  
  
x3  
## [1] 0.000000 2.780000 3.141593  
  
x4  
## [1] "A" "B" "C"
```

Nous pouvons également savoir directement le type d’objet du vecteur avec la commande “mode()” :

```
mode(x1)  
## [1] "NULL"  
  
mode(x2)  
## [1] "logical"  
  
mode(x3)  
## [1] "numeric"  
  
mode(x4)
```

¹⁶ Nous laisserons de côté les nombres complexes.

```
## [1] "character"
```

Il est également possible de tester à quel type d'objet appartiennent les éléments du vecteur “x” avec les commandes “`is.null()`”, “`is.logical()`”, “`is.numeric()`”, “`is.character()`”. Les résultats sont alors booléens avec des valeurs “`TRUE`” ou “`FALSE`” :

```
is.null(x1)
```

```
## [1] TRUE
```

```
is.null(x2)
```

```
## [1] FALSE
```

Il est enfin possible de convertir le type d'objet de “x” dans un autre type donné explicitement en utilisant les commandes “`as.null()`”, “`as.logical()`”, “`as.numeric()`”, “`as.character()`”.

```
as.character(x1)
```

```
## character(0)
```

```
as.numeric(x2)
```

```
## [1] 1 0 1
```

```
as.logical(x3)
```

```
## [1] FALSE TRUE TRUE
```

Remarque importante : *il faut être prudent quant à la signification de ces conversions. Rstudio donne toujours un résultat pour chaque instruction de conversion, même si c'est ce dernier n'a pas grand sens. La table de conversion (pierre de Rosette !) vous donnera les possibilités (Cornillon et al. 2012) :*

From	To	Function	Conversions
Logical	Numeric	<code>as.numeric</code>	<code>FALSE</code> → 0 <code>TRUE</code> → 1
Logical	Character	<code>as.character</code>	<code>FALSE</code> → "FALSE" <code>TRUE</code> → "TRUE"
Character	Numeric	<code>as.numeric</code>	"1", "2", ... → 1, 2, ... "A", ... → NA
Character	Logical	<code>as.logical</code>	"FALSE", "F" → FALSE "TRUE", "T" → TRUE Other characters → NA
Numeric	Logical	<code>as.logical</code>	0 → FALSE Other numerics → TRUE
Numeric	Character	<code>as.character</code>	1, 2, ... → "1", "2", ...

“Missing value”

Pour un certain nombre de raisons, certains éléments d'un vecteur donné peuvent être manquants (*c.f.* Séance 2, par exemple problème de collecte d'information pendant une enquête). Ces éléments sont appelés données manquantes (“*missing values*”). Ils ne sont donc pas disponibles pour l'utilisateur, et Rstudio les désigne “NA” pour “Not

Available”. Ce n'est pas un vrai mode et il a ses propres règles de calcul. Prenons par exemple le vecteur “z” :

```
z <- c(1, NA, 2)
```

L'ajout d'un scalaire 1 à ce vecteur renverra le résultat suivant :

```
z + 1  
## [1] 2 NA 3
```

Comme précédemment, afin de savoir où trouver les valeurs manquantes pour un vecteur x, il faut poser la question suivante :

```
is.na(z)  
## [1] FALSE TRUE FALSE
```

Cela donne une réponse booléenne de la même longueur que le vecteur “z” (la question est donc posée élément par élément).

Dans le cas d'un vecteur “z”, cela donne un vecteur logique identique avec “TRUE” si l'élément correspondant dans “z” est “NA”, et “FALSE” sinon.

Type de vecteur : vecteurs numériques

Maintenant que nous connaissons les types d'objet admis par les vecteurs, nous pouvons également utiliser certaines fonctions pour construire plus rapidement des vecteurs numériques. Un vecteur peut tout d'abord être défini comme une “séquence” :

```
x <- c(1:6)  
  
x  
## [1] 1 2 3 4 5 6
```

Le vecteur numérique comprend bien la séquence d'éléments indiqués entre les deux bornes, ce sont ici des nombres entiers (par défaut). Nous pouvons également utiliser la commande “seq()” pour définir un vecteur plus finement. De cette façon, le vecteur numérique peut contenir autant d'éléments nécessaires sur l'intervalle fixé en fixant les écarts entre deux éléments successifs :

```
x<- seq(1, 6, by=0.5)  
  
x  
## [1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0 5.5 6.0
```

Ou encore, en fixant le nombre d'éléments sans définir les écarts, égaux par défaut sur l'intervalle fixé :

```
x <- seq(1, 6, length=5)

x
## [1] 1.00 2.25 3.50 4.75 6.00
```

Un vecteur numérique peut enfin être une simple répétition d'éléments, on utilisera alors la commande “`rep()`” :

```
x <- rep(1, 6)

x
## [1] 1 1 1 1 1 1

x <- rep(c(1, 2), each=3)

x
## [1] 1 1 1 2 2 2
```

Type de vecteur : vecteurs “character”

Les vecteurs de type “`character`” comprennent non pas des éléments numériques mais d'ordre qualitatif (du texte, *c.f.* “type de vecteur” ci-dessous). Il est possible de les créer des vecteurs de caractères de la même manière que ceux numériques avec les fonctions “`c`” ou “`rep()`” :

```
x <- c("A", "BB", "C1")

x
## [1] "A"   "BB"  "C1"

x <- rep('A', 5)

x
## [1] "A"   "A"   "A"   "A"   "A"
```

Même si Rstudio interprète “`""`” et “`''`” de la même manière, nous utiliserons “`""`” à partir de maintenant, le plus commun dans la communauté.

Une autre façon de faire est de manipuler différents objets et de les **concaténer** ou d'en extraire une partie. Pour concaténer les objets, utilisez la fonction “`paste()`” :

```
x <- paste("X", 1:5, sep="-")

x
## [1] "X-1" "X-2" "X-3" "X-4" "X-5"

x <- paste(c("X", "Y"), 1:5, sep=". ", collapse="+")

x
```

```
## [1] "X.1+Y.2+X.3+Y.4+X.5"
```

L'argument “collapse” regroupe tous les éléments d'un vecteur avec une longueur de 1. Pour extraire, nous pourrons utiliser la commande “`substr()`” :

```
x<- substr("freerider", 5, 9)  
  
x  
  
## [1] "rider"
```

Ceci extrait les éléments, qui sont ici des lettres, classés de “5” à “9” du mot “freerider” ce qui donne “rider”.

Type de vecteur : vecteurs booléens ou logiques

Les vecteurs booléens sont généralement générés avec des opérateurs logiques: “>”, “>=”, “<”, “<=”, “==”, “!=”, etc. Ils peuvent également être générés à l'aide des fonctions “`seq()`”, “`rep()`” et bien évidemment “`c()`”. Ils peuvent être utilisés pour effectuer des sélections complexes ou conditionner les opérations (voir *Cornillon et al. 2012, p 87*).. Examinons le exemple suivant:

```
x<- 1>0  
  
x  
  
## [1] TRUE
```

Cette commande donne un vecteur logique de longueur 1 dont la valeur est donc “TRUE”, car 1 est évidemment supérieur à 0 ! Dans le même sens :

```
x <- c(1, 14, 18, 7, 8)  
  
x>13  
  
## [1] FALSE TRUE TRUE FALSE FALSE
```

Donne un vecteur logique de même longueur que le vecteur numérique “`x`”. Les éléments du vecteur logique prennent de la même façon la valeur “TRUE” lorsque l'élément correspondant remplit la condition donnée (ici strictement supérieur à 13) et la valeur “FALSE” si, à l'inverse, ce n'est pas le cas.

Lors des calculs, le vecteur logique prend des valeurs numériques, dans ce cas, “FALSE” devient 0 et le “TRUE” devient 1. Un exemple, une test d'objet est créé, qui est le vecteur des logiques (“FALSE, FALSE, TRUE”). Nous calculons ensuite le produit suivant :

```
x <- c(-1, 0, 2)
```

```

test <- x > 1

result <- (1+x^2)*(x>1)

result

## [1] 0 0 5

```

Les fonctions “`all()`” et “`any()`” peuvent également être utilisées. La fonction “`all()`” donne “`TRUE`” si tous les éléments remplissent la condition, “`FALSE`” si au moins un élément ne respecte pas le critère donné. La fonction “`any()`” renvoie “`TRUE`” si l’un des éléments remplit la condition, et “`FALSE`” dans le cas inverse :

```

x <- c(-1, 0, 2, 4)

testA <- all(x>1)

testA

## [1] FALSE

testB <- any(x>1)

testB

## [1] TRUE

```

Opérations entre deux vecteurs numériques

Définissons désormais un second vecteur nommé “`y`” :

```

x <- c(1, 3, 5, 7, 9)

y <- c(2, 4, 6, 8, 10)

```

Ce vecteur “`y`” est de longueur “5” comme notre premier vecteur “`x`” :

```

length(y)

## [1] 5

```

L’addition de vecteurs signifie ajouter le premier élément d’un vecteur au premier élément de l’autre vecteur, le deuxième élément du premier vecteur au deuxième élément de l’autre vecteur et ainsi de suite. De cette manière, en réutilisant notre premier vecteur “`x`” :

```

x <- c(1, 3, 5, 7, 9)

x + y

## [1] 3 7 11 15 19

```

Petit rappel d’algèbre linéaire : seuls les vecteurs avec le même nombre d’éléments peuvent être ajoutés.

Si nous essayons d'ajouter des vecteurs de différentes longueurs dans Rstudio, il commencera à **recycler les éléments du vecteur plus court**. Donc, si le vecteur “y” avait les éléments “2, 4, 6, 8, 10, 12” et nous essayons d'ajouter “y” à “x”, nous obtiendrions :

```
y <- c(2, 4, 6, 8, 10, 12)
```

```
x + y
```

```
## [1] 3 7 11 15 19 13
```

Comme on peut le voir, Rstudio a correctement ajouté les éléments des “5” premières positions dans chaque vecteur (c'est-à-dire : “ $1 + 2 = 3$ ”, “ $3 + 4 = 7$ ”, etc.). Mais lorsqu'il est arrivé au sixième élément sur le vecteur “y”, respectivement égal à “12”, il n'avait pas de sixième élément correspondant sur le vecteur “x” auquel l'ajouter. Alors Rstudio a recyclé le premier élément de “x”. Autrement dit, il a ajouté “12” au premier élément de “x” (“1”), pour obtenir “13”.

Union et intersection

Rstudio peut également gérer assez facilement les opérations sur les ensembles. Par exemple, nous pouvons définir deux ensembles d'éléments (des vecteurs donc...) comme suit :

```
A <- c(1, 2, 3, 4, 5)
```

```
B <- c(2, 3, 4, 5, 6)
```

- *L'union*

L'union de deux ensembles se définit comme l'ensemble des éléments appartenant à l'ensemble A ou à l'ensemble B ou aux deux (rappel de probabilités : $A \cup B : \forall a \text{ et } \forall b \exists B \forall x(x \in B \iff x \in a \vee x \in b)$.



A \cup B

Avec Rstudio, cette union entre les deux ensembles “A” et “B” peut être calculée par :

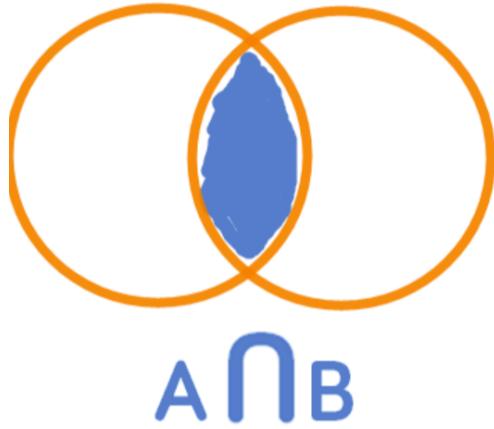
```
union(A, B)
## [1] 1 2 3 4 5 6
```

L'union de ces deux ensembles A et B est bien :

$$A \cup B = \{1, 2, 3, 4, 5\} \cup \{2, 3, 4, 5, 6\} = \{1, 2, 3, 4, 5, 6\}$$

- *L'intersection*

L'intersection de deux ensembles se définit comme l'ensemble des éléments appartenant conjointement à l'ensemble A et à l'ensemble B (rappel de probabilités : $A \cap B : \forall A \exists B \forall x (x \in B \iff x \in a \vee \sigma(x))$.



Avec Rstudio, cette intersection entre les deux ensembles “A” et “B” peut être calculée par :

```
intersect(A, B)
## [1] 2 3 4 5
```

L'intersection de ces deux ensembles est bien :

$$A \cap B = \{1, 2, 3, 4, 5\} \cap \{2, 3, 4, 5, 6\} = \{2, 3, 4, 5\}$$

Nous pouvons être intéressés de savoir si les éléments d'un ensemble “A” sont contenus dans l'autre ensemble “B” :

```
A %in% B
## [1] FALSE TRUE TRUE TRUE TRUE
```

Le premier élément est “FALSE” car le nombre “1” de l'ensemble “A” n'apparaît effectivement pas dans l'ensemble “B”. A contrario, toutes les autres valeurs sont “TRUE” car les nombres suivants de l'ensemble “A” se trouvent bien dans l'ensemble “B”.

Pour rappel : deux ensembles sont égaux s'ils ont les mêmes éléments. Nous pouvons tester l'égalité d'ensemble en utilisant la fonction “setequal()”

```
setequal(A, B)
## [1] FALSE
```

Observons que les deux ensembles ne sont pas égaux (surprise !).

Nous pouvons également calculer une **somme cumulative** d'un vecteur en utilisant “`cumsum()`”. Par exemple, la somme cumulée du vecteur `x` est calculée comme suit :

```
cumsum(x)  
## [1] 1 4 9 16 25
```

Notez que la somme des deux premiers éléments est “4” (“3 + 1”), puis les trois premiers éléments “9” (“1 + 3 + 5”), etc.

Le produit cumulatif peut également être calculé en utilisant “`cumprod()`”. Quand on parle du **produit scalaire** entre deux vecteurs, on se réfère à la **somme de chaque élément multiplié par paire**. Par exemple, pour nos vecteurs “`x`” et “`y`” nous définissons le produit scalaire comme :

```
y <- c(2, 4, 6, 8, 10)  
  
x %*% y  
##      [,1]  
## [1,] 190
```

Pour montrer comment cela a été calculé :

```
cum.prod.function <- 1*2 + 3*4 + 5*6 + 7*8 + 9*10  
  
cum.prod.function  
## [1] 190
```

Notez que pour obtenir la somme, nous avons ajouté les produits respectifs de chaque élément du premier vecteur avec les éléments correspondants du deuxième vecteur.

On aurait pu tout aussi bien calculer le **produit scalaire** en utilisant “`crossprod(x, y)`” :

```
crossprod(x,y)  
##      [,1]  
## [1,] 190
```

Parfois, nous voulons redimensionner un vecteur par un scalaire ! Autrement dit, nous pourrions chercher à multiplier chaque élément du vecteur par un nombre particulier. Par exemple, le vecteur “`x`” de longueur “5” par un facteur de 2 :

```
2 * x
```

```
## [1] 2 6 10 14 18
```

Notez alors que chaque élément du vecteur a bien été multiplié par le scalaire “2”. Cela a eu pour effet d'allonger

le vecteur, puisque chaque élément de celui-ci a été multiplié par 2 !

Matrices

Les matrices sont des objets dit “atomics”, ce qui signifie que tous les éléments qui la composent sont du même mode ou du même type (numéric, logical...). Chaque élément a de la matrice A peut être localisée par sa ligne i et sa colonne j . La matrice A contient m_i lignes et n_j colonnes, elle est donc de dimension mn :

$$A = [a_{ij}] = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & a_{m3} & \cdots & a_{mn} \end{bmatrix}_{m \times n} \in M_{m \times n}$$

(i, j)-th entry: a_{ij}

row: m column: n size: $m \times n$

Sur Rstudio, quand on parle d'une matrice, on se réfère tout simplement à un tableau de vecteurs contenus dans le même objet, la matrice. La compréhension du fonctionnement et des opérations sur les matrices est essentiel pour donner un sens à l'analyse statistique, aux modèles relativement avancés dans des contextes multivariés. Mais don't worry !

Nous revisiterons les matrices au fur et à mesure que nous en aurons besoin. Pour l'instant, nous souhaitons simplement démontrer comment nous pouvons construire une matrice, et calculez quelques opérations matricielles élémentaires.

Pour rappel : les deux attributs intrinsèques d'un objet Rstudio sont (i.) sa longueur, qui correspond ici au nombre total d'éléments de la matrice, et (ii.) le mode ou type d'objet, qui correspond ici au mode des éléments contenus dans cette matrice.

Toutefois, les matrices incluent également un troisième attribut : la dimension “`dim()`”, qui donne le nombre de lignes et le nombre de colonnes. Ils peuvent également posséder un attribut facultatif “`dimnames`”. Voici les principales façons de créer une matrice.

- fonction “`matrice`”

La plus utilisée est la fonction `matrice` qui prend comme arguments le vecteur des éléments et le nombre de lignes ou colonnes dans la matrice :

```
A <- matrix(1:6, 3, 2)

A

##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

où “A” est le nom de notre matrice, “1:6” désigne la matrice à avoir les entrées 1 à 6, c'est-à-dire avoir des nombres dans la matrice allant de 1 à 6, et “3, 2” demande à la matrice d'avoir 3 lignes et 2 colonnes. Mais on peut aussi, utiliser la fonction “c” en spécifiant le nombre de colonnes “ncol” ou de lignes "nrow" :

```
A <- matrix(c(1, 2, 3, 4, 5, 6), ncol=2)
```

```
A

##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

```
A <- matrix(c(1, 2, 3, 4, 5, 6), nrow=3)
```

```
A

##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

Remarque 1 : si nous spécifions trop d'éléments, autrement dit le vecteur est trop long, pour notre matrice mais trop peu de colonnes et/ou de lignes, R tronquera simplement la matrice. Par exemple, une matrice avec des valeurs de 20 à 29, mais ne spécifie qu'une matrice “ 2×2 ”. Par conséquent, Rstudio entre simplement les 4 premiers éléments de la matrice et ignore le reste des éléments :

```
B <- matrix(20:29, 2, 2)
```

```
B

##      [,1] [,2]
## [1,]    20    22
## [2,]    21    23
```

Remarque 2 : si la longueur du vecteur est trop petite par rapport Au nombre d'éléments envisagés dans la matrice, Rstudio remplira tout de même la matrice entière avec les éléments disponibles. Lorsque le vecteur est trop petit, Rstudio le répète :

```
B <- matrix(1:6, nrow=3, ncol=3)
```

```
B
```

```

##      [,1] [,2] [,3]
## [1,]    1    4    1
## [2,]    2    5    2
## [3,]    3    6    3

```

Il est possible de remplir une matrice avec un seul élément (un scalaire) sans avoir à créer le vecteur d'éléments de la matrice :

```

C <- matrix(1, nrow=2, ncol=4)

C

##      [,1] [,2] [,3] [,4]
## [1,]    1    1    1    1
## [2,]    1    1    1    1

```

Remarque 3 : les vecteurs, en dehors de ceux d'éléments de matrice, ne sont pas considérés par Rstudio comme des matrices. Cependant, il est possible de transformer un vecteur en une matrice à une colonne en utilisant la fonction “`as.matrix()`” :

```

x <- seq(1, 6, by=1)

x

## [1] 1 2 3 4 5 6

C <- as.matrix(x)

C

##      [,1]
## [1,]    1
## [2,]    2
## [3,]    3
## [4,]    4
## [5,]    5
## [6,]    6

```

Rappel : la position d'un élément dans une matrice est généralement indiquée par sa ligne i et sa colonne j .

Ainsi, pour sélectionner l'élément “ (i, j) ” de la matrice ‘ A ’, on pourra écrire :

`A[i,j]`

Cependant, il est très rare de n'avoir à sélectionner qu'un seul élément dans une matrice. Généralement, plusieurs des lignes et / ou des colonnes sont sélectionnées. Examinons les différents cas possibles sur la matrice A :

```

A <- matrix(c(1:6), nrow=3)

A

##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6

```

```
## [2,]    2    5
## [3,]    3    6
```

- *Sélection par entier positif*

```
# A[i,] pour i = 2
```

```
A[2, ]
```

```
## [1] 2 5
```

Cela donne bien la ligne $i = 2$ de la matrice A sous la forme d'un vecteur.

```
# A[i,, drop = F] pour i = 3
```

```
A[3,, drop = F]
```

```
## [,1] [,2]
## [1,]    3    6
```

Donne cette fois la ligne $i = 3$ de la matrice A sous la forme d'une matrice à une seule ligne et non un vecteur (donc le nom de la ligne peut être conservé).

```
# m[, c(j)] pour j = 2, 1 et 2
```

```
A[, c(2, 1, 2)]
```

```
## [,1] [,2] [,3]
## [1,]    4    1    4
## [2,]    5    2    5
## [3,]    6    3    6
```

Donne la deuxième, la première et encore la deuxième colonnes de la matrice A sous une forme matricielle.

- *Sélection par entier négatif*

```
# A[-i, ] pour i = 3
```

```
A[-3, ]
```

```
## [,1] [,2]
## [1,]    1    4
## [2,]    2    5
```

Donne la matrice A sans la troisième ligne.

```
# A[, -j] pour j = 1
```

```
A[1:2, -1]
```

```
## [1] 4 5
```

Donne les deux premières lignes de A sans sa première colonne.

- Sélection booléenne (*logique*)

L'instruction suivante ne donne que les colonnes de A pour lesquelles la valeur sur la première ligne est strictement supérieur à 2 :

```
A[, A[1,>2]
```

```
## [1] 4 5 6
```

Il s'agit bien sûr d'une matrice, alors que l'instruction suivante donne un vecteur contenant les valeurs de A strictement supérieures à 2 :

```
A[A > 2]
```

```
## [1] 3 4 5 6
```

- Sélection avec remplacement

L'instruction suivante remplace les valeurs de A qui sont supérieures à 2 avec “NA” :

```
A[A > 2] <- NA
```

```
A
```

```
## [,1] [,2]
## [1,]    1   NA
## [2,]    2   NA
## [3,]   NA   NA
```

Rappels de calcul matriciel

Quelques opérations sur les matrices :

```
A <- matrix(c(1:4), nrow= 2)
```

```
A
```

```
## [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

```
B <- matrix(c(3:6), ncol = 2)
```

```
B
```

```
## [,1] [,2]
## [1,]    3    5
## [2,]    4    6
```

```
C <- A + B
```

```
C
```

```

##      [,1] [,2]
## [1,]    4    8
## [2,]    6   10

D <- A * B

D

##      [,1] [,2]
## [1,]    3   15
## [2,]    8   24

E <- exp(A)

E

##      [,1]      [,2]
## [1,] 2.718282 20.08554
## [2,] 7.389056 54.59815

F <- A^B

F

##      [,1] [,2]
## [1,]    1  243
## [2,]   16 4096

```

Le tableau suivant donne les fonctions les plus utiles pour de l’algèbre linéaire.

Function	Description
X%*%Y	Product of matrices
t(X)	Transposition of a matrix
diag(5)	Identity matrix of order 5
diag(vec)	Diagonal matrix with the values of vector <code>vec</code> in the diagonal
crossprod(X, Y)	Cross product (<code>t(X)%*%Y</code>)
det(X)	Determinant of matrix X
svd(X)	Singular value decomposition
eigen(X)	Matrix diagonalisation
solve(X)	Matrix inversion

Imaginons ce système :

$$26x + 30y = 1$$

$$38x + 44y = 2$$

Par exemple la fonction “`solve()`” pourrait nous permettre de résoudre ce système d’équations très facilement :

```

D <- matrix(c(26, 38, 30, 44), ncol = 2)

D

##      [,1] [,2]
## [1,]    26   30
## [2,]    38   44

# Notons que :

A <- matrix(1:4, ncol=2)

A

##      [,1] [,2]
## [1,]    1    3

```

```

## [2,]    2    4
B <- matrix(c(5, 6, 7, 8), ncol=2)

B
##      [,1] [,2]
## [1,]    5    7
## [2,]    6    8

D <- A %*%t(B) # On multiplie A par la matrice B transposée

D
##      [,1] [,2]
## [1,]   26   30
## [2,]   38   44

# Intéressant !

V <- c(1,2) # On identifie ici les deux inconnues "x" et "y"

solve(D, V)
## [1] -4.0  3.5

```

Ainsi, $x = -4$ et $y = 3.5$!

Rappel : on appelle **transposée** une matrice A de type (m, n) et de terme général a_{ij} , la matrice A^t obtenue en échangeant les lignes i et les colonnes j de même indice de A :

$$A = a_{ij} \iff A^t = a_{ij}^t = a_{ji}$$

A

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$$

Par conséquent : $(A^t)^t = A$

Les facteurs

Les facteurs sont des vecteurs qui permettent d'utiliser et de manipuler des données qualitatives.

La longueur est déterminée par la fonction de “`length()`”, le mode par la fonction “`mode()`” et les catégories du facteur par “`levels()`”. Ils forment une classe d’objets et sont traités différemment selon la fonction, comme la fonction “`plot()`” pour les représentations graphiques (comme nous le verrons lors de la séance 2). Les facteurs peuvent être non

ordinale (homme, femme) ou **ordinale** (niveau d'aptitude en statistiques). Trois fonctions peuvent être utilisées pour créer des facteurs. Tout d'abord la fonction “`factor()`” :

```
gender <- factor(c("M", "M", "F", "M", "F", "M", "M", "M"))

gender
## [1] M M F M F M M M
## Levels: F M
```

Nous pouvons également nommer chaque catégorie “`level()`” au fur et à mesure que le facteur est construit :

```
gender <- factor(c(2, 2, 1, 2, 1, 2, 1), labels=c("woman", "man"))

gender
## [1] man     man     woman man     woman man     woman
## Levels: woman man
```

La fonction “`ordered()`” peut aussi être très utile :

```
ability <- ordered(c("beginner", "beginner", "expert", "expert",
                      "intermediate", "intermediate"))

ability
## [1] beginner      beginner      expert       expert
## [6] intermediate intermediate expert
## Levels: beginner < intermediate < expert
```

La fonction “`as.factor()`” s'utilise de la même manière que précédemment :

```
v <- c(1:5, 5:1)

v
## [1] 1 2 3 4 5 5 4 3 2 1

f <- as.factor(v)

f
## [1] 1 2 3 4 5 5 4 3 2 1
## Levels: 1 2 3 4 5
```

Afin de connaître les catégories, le nombre de catégories et le nombre d'éléments par catégories, on utilise pour le facteur “`f`” :

```
levels(f)
## [1] "1" "2" "3" "4" "5"

nlevels(f)
## [1] 5

table(f)
## f
## 1 2 3 4 5
```

```
## 2 2 2 2 2  
f  
## [1] 1 2 3 4 5 5 4 3 2 1  
## Levels: 1 2 3 4 5
```

Nous le verrons lors de la séance 3, la fonction `table()` peut également être utilisée pour construire des tableaux croisés.

Introduction à R-Markdown

Tout d'abord, une première courte vidéo pour comprendre qu'est-ce que R-Markdown :

“<https://vimeo.com/178485416>”

Et maintenant, une seconde vidéo pour commencer :

“https://www.youtube.com/watch?v=FO6Jo3Y9knk&ab_channel=MichelCucherat”

Maintenant on refait l'exemple ensemble !

Pour aller plus loin : R-Markdown for Begginers.

A partir de maintenant je veux recevoir que des fichiers “.html” de votre part, le tout en R-Markdown !

Contact : thibaud.degUILhem@u-paris.fr