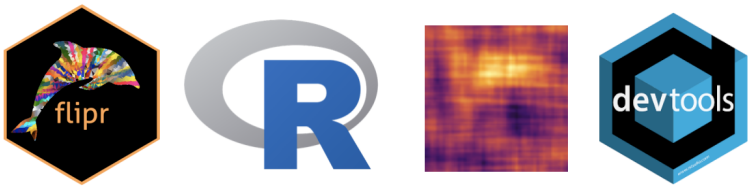


Rapport de stage - M1 Ingénierie Statistique

Bonnes pratiques pour le développement collaboratif de logiciel

Application au développement de packages R autour de l'inférence statistique par permutation avec implémentation de tests unitaires

Auteur : **Chiapello Juliette**
M1 Ingénierie Statistique

Encadrant de stage : **Stamm Aymeric**
Ingénieur de recherche

Laboratoire de Mathématiques Jean Leray
Université de Nantes
Juin - Juillet 2021

Sommaire

Introduction

Le stage a été initié à ma demande car je souhaitais apprendre différentes pratiques relatives à la programmation, appliquées dans un contexte mathématique. Ces pratiques concernaient :

- la création de classes dans R et Python,
- l'utilisation de tests unitaires dans le développement de fonctions/packages destiné(e)s à des utilisateurs,
- une meilleure connaissance de ggplot2,
- la création de packages,
- la création de belles visualisations,
- l'utilisation du débugger dans RStudio,
- le clean code.

Mon encadrant de stage, Aymeric Stamm, m'a proposé de travailler sur un des packages qu'il développe en R et Python : le package *flipr* en R (et *flippy* en Python). Participer au développement de *flipr* devait en effet me permettre d'effectuer une phase de recherche documentaire puis de mettre en pratique les compétences précédemment citées. J'avais de plus demandé à pouvoir davantage coder en Python qu'en R par simple préférence.

Nous avions donc initialement intitulé le stage de la façon suivante :

« Bonnes pratiques pour le développement collaboratif de logiciel : application au développement d'un module Python autour de l'inférence statistique par permutation avec implémentation de tests unitaires. »

Et nous avions défini les activités prévues de la manière suivante :

- Réalisation des tests unitaires du package R *flipr*,
- Portage du package R *flipr* en Python via la réalisation d'un module Python dédié.

Le stage a commencé quelques mois plus tard. Entre temps, avec deux de mes camarades de promotion, Céline et Léonie, nous avons rendu un projet en R portant sur une problématique de géostatistique [1]. En plus du fait qu'il m'ait permis de beaucoup mieux manipuler le code R, ce projet contenait un certain nombre de fonctions que nous avions codées et nous avons convenu avec mon tuteur, Aymeric Stamm, qu'un bon entraînement pour que je découvre la manière de créer des packages en R serait de rassembler l'ensemble de ces fonctions dans un package que je créerai.

Le stage a donc débuté ainsi, par la création d'un package R à partir d'un ensemble de fonctions déjà implémentées. Cela correspondra à la première partie du rapport. La mise en ligne de documents et plus particulièrement de code et de packages sur Github est aussi expliquée à cet endroit.

Ensuite, nous avons discuté du fait de concevoir une *Shiny App* (une application) avec des fonctionnalités faisant appel à ce code. La conception de cette *Shiny App* est expliquée dans la deuxième partie du rapport.

En parallèle de ces premières considérations, nous avons envisagé de tester le code avec des tests unitaires. Cette étape a été un des points centraux du stage. Tout le développement concernant les tests utilisés en informatique en général, et les tests unitaires en particulier, constitue la troisième partie du rapport.

Ayant rodé la méthode de test avec le package simple issu du projet en géostatistique réalisé dans l'année, nous avons alors convenu du fait que je pouvais commencer à travailler sur le package développé par Aymeric Stamm, à savoir *flipr*. Le fonctionnement de *flipr* est exposé dans la quatrième partie du rapport.

Le langage Python n'a finalement pas été utilisé pendant le stage.

[1] Il s'agissait du projet SSPM (Supervised Study Project in Mathematics), réalisé avec l'encadrement de Monsieur Nicolas Bez et intitulé « créer des champs aléatoires structurés spatialement par méthodes à noyaux ».

1 Création d'un package R

Le projet de géostatistique que nous avons rendu, Céline, Léonie et moi, nous avait permis de mettre en place un code R regroupant plusieurs fonctions. Ce sont ces fonctions qui ont été reprises pour la création d'un package R nommé *randomfields*. Nous allons tout d'abord rappeler ces fonctionnalités. Puis, cette partie présentera les étapes de création d'un package R. Enfin, nous verrons comment mettre le package en ligne sur Github et comment installer les *github actions* et les badges servant à vérifier que le package fonctionne et que le code est couvert par des tests unitaires.

1.1 Le package : méthode à noyau - champs aléatoires structurés

Le package R, qui reprend les fonctions codées dans le projet rendu en cours d'année, a pour but de permettre les opérations suivantes :

- représenter des matrices (grilles) Z par des grilles de couleurs,
- créer de nouvelles grilles Y par moyennes glissantes sur des grilles Z existantes (méthode à noyau où le noyau est une fonction moyenne). Une moyenne glissante a pour support une fenêtre (glissante).

On se place ensuite dans le cadre spécifique de grilles Z générées aléatoirement à partir de lois statistiques connues. Les matrices Z n'ont donc pas de structures spatiales. En revanche, les grilles Y possèdent des structures spatiales. C'est une conséquence de la moyenne glissante : certains couples de points (ceux issus de fenêtres qui se recouvrent) ont une covariance non-nulle. On peut montrer que cette covariance ne dépend que de la distance entre les points (stationnarité d'ordre 2). Il est alors possible de :

- construire de façon empirique le graphique de la covariance en fonction de la distance entre deux points d'une grille structurée Y,
- établir ce graphique de façon théorique : il suffit de connaître l'écart-type de la loi ayant servi à générer la grille initiale Z.

Enfin, le package assure aussi la fonctionnalité de :

- réaliser des moyennes par blocs,
- calculer la réduction de variance suite à ces moyennes par blocs.

1.2 Etapes de création et de développement de package R

Cette section résume les étapes de la création de package en R. Elle s'appuie sur la documentation suivante : https://r-pkgs.org. L'ensemble de la procédure est récapitulée sur la Figure 1 : Cheat sheet R packages.

1.2.1 Initialisation du package (étapes à ne faire qu'une seule fois)

Librairies utiles

Avant toute chose, la librairie *devtools* est à installer et charger pour pouvoir créer et développer un package. Il est possible la charger systématiquement par défaut en ajoutant au fichier **.Rprofil** (fichier caché, installé par défaut avec R) les lignes :

```
if (interactive()) {
suppressMessages(require(devtools))
}
```

On peut également rajouter dans le `if` la ligne :

```
suppressMessages(require(testthat))
```

si l'on souhaite que la librairie permettant de faire des tests unitaires (voir troisième partie du rapport), *testthat*, soit également chargée par défaut.

Création du package

La commande `create_package("chemin/Nomdupackage")` permet de créer un nouveau package à l'endroit voulu. A cette occasion sont créés plusieurs éléments :

- un dossier **R** qui contiendra les fonctions,
- un fichier **NAMESPACE**, qui déclare les fonctions à exporter (celles du package) et celles à importer (celles provenant d'autres packages),
- un fichier **DESCRIPTION**, déclarant, entre autres, le nom de l'auteur, la license et les éventuels autres packages utilisés,
- une session R **Nomdupackage.Rproj**, qui s'ouvre automatiquement, et qui sera à systématiquement utiliser pour ce projet,
- des fichiers cachés : **.Rbuildignore**, qui déclare les fichiers à ignorer lorsque le package est compilé, **.gitignore**, qui anticipe l'usage de Git et spécifie certains fichiers à ignorer par Git, **.Rproj.user** qui est un dossier interne à RStudio.

Fichier DESCRIPTION

Le fichier **DESCRIPTION** est à compléter avec le nom du ou des auteur(s), et avec un titre détaillé (champ *Title*) et une description (champ *Description*).

Licence

Vient ensuite le choix de la licence. Le site officiel conseille la licence MIT : `use_mit_license("Prénom Nom")`, mais différents choix sont possibles. Voir : https://r-pkgs.org/license.html. Les fichiers **LICENSE** et **LICENSE.md** sont alors créés.

README : une aide pour les utilisateurs

Enfin, les fichiers **README.md** et **README.rmd**, donnant une description du package aux utilisateurs, peuvent être générés avec les commandes `use_readme_rmd()` et `build_readme()`. Ils sont à compléter.

La phase d'écriture des fonctions ou des classes peut alors commencer.

1.2.2 Développement du package (étapes à faire de façon cyclique)

On se place toujours dans la session R **Nomdupackage.Rproj**.

Création de fonctions

Pour créer une fonction, on utilise la commande `use_r("Nom_de_la_fonction")`. Cela crée un fichier vierge **Nom_de_la_fonction.R** dans le dossier **R** (on peut aussi créer ce fichier à la main). On peut alors écrire la fonction dans ce nouveau fichier. Si la fonction a besoin d'appeller des fonctions d'autres packages, il est possible d'utiliser la commande `use_package()`. Cela ajoutera aux imports du fichier **DESCRIPTION** le nom du package appelé. Par exemple, si l'on souhaite utiliser ggplot2, on écrira `use_package("ggplot2")`. Ensuite, on peut utiliser les fonctions du package en utilisant la syntaxe "`::`" pour spécifier l'appartenance des fonctions à ce package. Par exemple : `ggplot2::aes`. Une autre possibilité, si la syntaxe avec les deux points alourdie vraiment trop le code (comme dans le cas de ggplo2), est de rajouter une ligne :

```
#' @import ggplot2
```

dans la documentation de la fonction (voir le paragraphe sur la documentation). On peut alors appeler directement les fonctions voulues sans avoir recours à la syntaxe avec les deux points.

ATTENTION 1

On notera de penser à limiter le nombre de dépendances avec d'autres packages. Cela évite aux utilisateurs de notre package d'installer d'autres packages non-voulus sur leur ordinateur. Il est possible d'éventuellement recoder des fonctions pour limiter les dépendances.

ATTENTION 2

La première option, c'est-à-dire l'emploi de la fonction use_package("nomdelalibrairie") ainsi que la syntaxe avec les deux points est à privilégier. Elle permet notamment de bien visualiser les fonctions qui dépendent d'autres packages dans le code, grâce au nom du package et aux deux points qui les précèdent. L'emploi de la ligne `#' @import ggplot2` pour ggplot2 reste très spécifique : il y a tellement d'appels à ggplot2 dans les fonctions des graphiques que l'on préfèrera cette option dans ce cas précis.

La commande `load_all()` permet de charger le package et de rendre la fonction disponible. Les raccourcis clavier sont : CRTL + SHIFT + L pour Windows et Linux et CMD + SHIFT + L pour Mac OS.

Tests unitaires

La mise en place des tests unitaires pour vérifier que la fonction se comporte bien comme attendue (voir la troisième partie du rapport pour la procédure détaillée des tests) est une bonne pratique de dévelopement. La meilleure pratique, le Test Driven Development, est même d'écrire chaque test en amont, avant d'écrire les lignes de code correspondantes dans la fonction. Quoiqu'il en soit, l'existence ou non de tests n'influe pas à proprement parler sur le fonctionnement du package (cette étape peut être faite plus tard). Nous conseillons tout de même d'écrire des tests, avant ou juste après l'écriture de la fonction. Pour cela, on utilise les deux commandes successives :

```
use_testthat()
use_test("Nom_de_la_fonction")
```

Un dossier **test** apparaît lorsque ces commandes sont exécutées pour la première fois. A l'intérieur se trouvent les fichiers de tests des fonctions du package. Un modèle d'écriture de tests est présenté par défaut dans le fichier de tests **test-Nom_de_la_fonction.R** créé. Pour exécuter les tests du fichier, on peut cliquer sur l'icône "Run tests" en haut à droite dans RStudio. Mais on peut aussi faire tourner l'ensemble des tests du package (de toutes les fonctions que l'on aura créées) avec la commande `test()`. Les raccourcis clavier

correspondant à cette commande sont les suivants : CRTL + SHIFT + T pour Windows et Linux et CMD + SHIFT + T pour Mac OS.

Documentation

La fonction a également besoin d'une documentation ! Elle est essentielle pour que le package compile correctement. Pour cela, on place le curseur quelque part sur la fonction et on ouvre le menu *Code* puis on clique sur *Insert Roxygen Skeleton*. Il suffit ensuite de compléter les champs. C'est dans ce squelette que l'on peut ajouter une entrée (`#' @import ggplot2` par exemple) si besoin est, comme expliqué dans le paragraphe sur l'importation de package. Attention cependant, nous rappelons que mieux vaut utiliser `use_package()` et la syntaxe avec les deux points en général.

La commande `document()` est alors à exécuter. Les raccourcis clavier sont : CRTL + SHIFT + D pour Windows et Linux et CMD + SHIFT + D pour Mac OS. Cela écrit le nom de la fonction dans le fichier **NAMESPACE** (et ajoute un dossier **man** la première fois que la commande est exécutée : on ne s'attardera pas sur ce dossier).

Si on exécute à nouveau `load_all()` (CRTL + SHIFT + L pour Windows et Linux et CMD + SHIFT + L pour Mac OS), la fonction ET sa documentation deviennent accessibles.

Vérification du bon fonctionnement du package

Il est temps de vérifier que l'ensemble du package fonctionne. Pour cela on utilise la commande `check()` (raccourcis : CRTL + SHIFT + E pour Windows et Linux et CMD + SHIFT + E pour Mac OS). Si d'éventuels erreurs, warnings ou notes sont renvoyés, ils sont à corriger.

Installation

Dernière étape (non-obligatoire) : l'installation du package sur l'ordinateur. Il suffit d'exécuter la commande `install()`. Ensuite on peut appeller le package avec la commande `library(Nomdupackage)` depuis n'importe quelle session R.

Toutes ces étapes sont résumées sur le schéma "Cheat sheet R packages". Elles sont appellées à être répétées de nombreuses fois, à chaque fois que l'on crée une nouvelle fonction (ou une nouvelle classe).

On peut exécuter toutes les lignes de code présentées en vert sur la cheatsheet (et en bleu si on veut inclure de suite les tests) pour créer un package : cette cheatsheet se donne pour but d'être très opérationnelle !

Remarque : il existe bien sûr déjà une cheatsheet au format officiel ; elle est disponible à cette adresse : https://rklopotek.blog.uksw.edu.pl/files/2017/09/package-development.pdf

R Packages - Cheat sheet

For details see : https://r-pkgs.org/

Libraries

```
library(devtools)
library(testthat)
```

OR

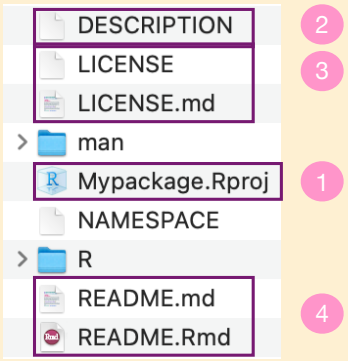
For personal start-up configuration with devtools and testthat, write in your `.Rprofile` startup file

```
if (interactive()) {
  suppressMessages(require(devtools))
}
```

Add also testthat

FIRST SETUP (Do it once)

- Create your package :
  `create_package("path/Mypackage")`
- `Mypackage.Rproj` session is opened (1)
- Edit the `DESCRIPTION` file : (2)
  - Make yourself the author
  - Write some descriptive text in the TITLE and DESCRIPTION fields
- Choose a license (see https://r-pkgs.org/license.html for details) :
  Ex : `use_mit_licence("Jane Doe")` (Write your name and surname) (3)
- Initialize a `README.md` (and fill the fields)
  (Optional if your package is only for you without using GitHub)
  `use_readme_rmd()`
  `build_readme()` (4)



WORKING FLOW (Do it cyclically)

Open `Mypackage.Rproj`

Add a function
`use_r("Afunction")`
The file `Afunction.R` is created

Write the function in the created file `Afunction.R`

If you need to use an fonction from another package to write your actual function, run :
`use_package("package_name")`
➡ This adds the package to the "Imports" section of the DESCRIPTION file.

To call the function, use :
`package_name::thefunctionyouneed()`

⚠ Do not create too many dependencies ! You can also re-write some functions

`load_all()` ➡ Make *Afunction* available in the current session
(Better to avoid running the functions. Save them and load them with *load_all()*)
Shortcuts : CRTL + SHIFT + L / CMD + SHIFT + L

Create a unit tests file
`use_testthat()`
`use_test("Afunction")`
Write unit tests in the created file `test-Afunction.R`
Check all unit tests with `test()`
Or use a shortcut :
CRTL + SHIFT + T / CMD + SHIFT + T

Write the documentation :
Place the cursor somewhere in the function and then click on :
Code > Insert Roxygen Skeleton
Fill the fields. Then run :
`document()`
Shortcuts : CRTL + SHIFT + D / CMD + SHIFT + D
`load_all()`
➡ You can use *Afunction* and its documentation in the current session !

`check()`
➡ It checks that the hole package works !
➡ You must correct errors/warnings/ notes if needed
Shortcuts : CRTL + SHIFT + E / CMD + SHIFT + E

Run `install()` if you want :
➡ it installs your package on your computer and you can run
`library(Mypackage)`
to access your package whenever you want !

Write functions, write classes, ...

Figure 1: Cheat sheet R packages

1.3 Git et Github

1.3.1 Git

Git est un *Version Control System*, autrement dit un outil de gestion des versions d'un projet. Il fonctionne localement sur un ordinateur en permettant de réaliser des sauvegardes au fur et à mesure de l'avancement. Ces sauvegardes permettent à tout moment de revenir à un état antérieur.

Une fois Git installé, les deux lignes suivantes permettent de le configurer dans un terminal :

```
git config --global user.name "John Doe"
```

```
git config --global user.email johndoe@example.com
```

Il s'agit de choisir notre nom d'utilisateur et notre adresse e-mail associée.

Ensuite, depuis le terminal, on peut se rendre dans le dossier où l'on a créé le package et initialiser un dépôt Git:

```
cd path/Monpackage
```

```
git init
```

Remarque : on peut aussi utiliser la commande `use_git()` dans RStudio pour cette étape.

Par la suite, on peut "mettre sur la pile" (staging area) les fichiers que l'on a modifiés et que l'on souhaite inclure dans la prochaine sauvegarde :

```
git add "nomdufichier"
```

Cette ligne peut être exécutée autant de fois que l'on souhaite rajouter des fichiers. Pour inclure tous les fichiers en une seule ligne, on utilise :

```
git add .
```

Après cette étape, la commande :

```
git commit -m "commentaire"
```

récupère tous les fichiers "sur la pile" et enregistre leurs modifications en leur associant un commentaire qui spécifie l'état du projet lors de cet enregistrement.

1.3.2 Github

Github permet de mettre en ligne nos sauvegardes (ce qui évite toute perte s'il y a un problème en local). C'est aussi un outil de partage et de collaboration : Github sert par exemple à faire du "pair-reviewing" : les collaborateurs peuvent consulter le code et faire des propositions de modifications ou des commentaires. De plus, dans le cadre de la création de packages avec R, la mise en place de certaines procédures de tests est possible : il s'agit des *github actions*. Ces *github actions* peuvent échouer ou réussir. L'échec ou le succès est représenté par des badges qui deviennent verts ou rouges (les badges ne sont pas obligatoires, mais ils sont une bonne représentation des résultats des *github actions*). Nous avons placé deux badges sur le package *randomfield* : un qui se nomme "**R-CMD-Check**" et l'autre "**code-coverage**". Le premier correspond à une *github action* qui réalise des tests sur le bon fonctionnement du package, et ce sur différentes machines et différents systèmes. Ce sont des tests un peu plus restrictifs que ceux réalisés par la fonction `check()` de **R** utilisée lors de la création du package en local. Le second badge correspond à une *github action* qui

vérifie que les tests unitaires passent. Il est associé à un troisième badge qui donne le pourcentage de lignes de code testées. La mise en ligne sur github permet, via les *github actions*, de travailler dans un processus d'intégration continue ou CI (Continuous Integration), et de vérifier ainsi régulièrement que les *github actions* renvoient des succès.

Remarque : on peut utiliser Github directement avec R, notamment avec la commande `use_github()`

Les lignes suivantes permettent de mettre en place la *github action* **R-CMD-check()** et de placer le badge correspondant :

```
use_github_action_check_standard()
use_github_actions_badge(name = "R-CMD-check")
```

De même, les lignes suivantes permettent de mettre en place la *github action* du **code-coverage** et de placer deux badges indiquants, pour l'un, si les tests unitaires passent bien tous avec succès, et pour l'autre quel est le pourcentage de code couvert. :

```
use_coverage()
use_github_action("test-coverage")
use_github_actions_badge(name = "test-coverage")
```

Le service en ligne disponible sur codecov.io permet de réaliser le calcul du nombre de lignes de code testées sur le nombre de lignes totales pour réaliser le pourcentage. Il indique aussi visuellement les lignes testées (en vert) et celles non-testées (en rouge) quand on clique sur les fichiers .R. Nous verrons dans la partie sur les tests qu'une haute couverture de code n'est pas forcément un gage du fait que le code est bien testé. Il s'agit plutôt d'un indicateur pour savoir qu'elles sont les parties testées. Ensuite, il faut être sûr que les tests sont pertinents, c'est-à-dire, principalement, qu'ils échouent bien s'il y a des fonctionnalités du code qui ne fonctionnent plus. Un des buts premiers des tests est en effet de pouvoir remanier le code (refactoring) pour l'améliorer, tout en vérifiant continuellement que toutes ses fonctionnalités sont préservées.

Sur codecov.io, l'accès au service se fait par la création d'un compte gratuit, à lier au compte Github. Ensuite, un bouton permet de se rendre sur les repositories (dossiers) Github non synchronisés avec codecov. En se rendant sur le repository qui correspond à celui de notre package, une procédure de mise en place du codecoverage est présentée. Un TOKEN est en particulier donné. Il est à copier et à coller au sein de la ligne de code suivante dans la session R de travail :

```
covr::codecov(token = 'PASTE CLIPBOARD')
```

Depuis le terminal, on peut mettre en ligne le projet : suivre alors la procédure de Github pour lier le dossier Git en local au repository Github en ligne. Pour cela, on peut créer un nouveau repository sur Github et, par exemple, recopier les lignes du deuxième encart proposé par Github dans le terminal. La partie locale et la partie en ligne sont alors liées. Ensuite on utilise la commande `git push` après chaque nouvelle sauvegarde que l'on fait sur Git en local, afin d'actualiser aussi le projet en ligne. A chaque fois que l'on "push" sur Github, l'ensemble de la procédure (R-CMD-check et codecoverage) se lance sur Github, dans l'onglet "github actions". Après quelques minutes, on peut savoir si les tests ont réussi ou échoué et visualiser cela sur les badges !

On note que ces badges sont des indicateurs pour nous-mêmes mais aussi pour des personnes s'intéressant à nos projets.

La mise en ligne du package du Github a été expliquée. Nous allons maintenant utiliser ce package dans une application qui porte sur la représentation de champs aléatoires.

2 Création d'une *Shiny App*

Les *Shiny Apps* sont des applications que l'on peut créer en R, notamment grâce à la librairie *shiny*. L'application que nous avons développée se nomme "*Random Fields*" et a pour but de visualiser spatialement des champs aléatoires. Elle fait appel à la librarie *randomfields* dont nous avons expliqué la création et la mise en ligne dans la partie précédente.

Le lien est le suivant : *http://juliette-chiapello.shinyapps.io/RandomFields*

(Penser à refermer après utilisation car il n'y a "que" 25h d'utilisation par mois tous utilisateurs confondus.)

Le premier onglet contient du texte et présente l'application. Le deuxième permet de visualiser quatre distributions statistiques en représentant leur densité de probabilités. En-dessous, les valeurs d'une réalisation sont affichées dans une grille en fonction d'une échelle de couleur. Les quatre distributions sont la loi normale, la loi de Bernoulli, la loi de Poisson et la loi de Student. Le troisième onglet montre l'effet du passage d'une moyenne glissante à support carré sur une grille. La réduction de variance est illustrée sur des boxplots et des histogrammes. Le quatrième onglet permet de visualiser les covariances des grilles en fonction de la distance entre les points. Enfin le dernier onglet affiche cette covariance de manière théorique.

2.1 Création d'une *Shiny App*

Une *Shiny App* contient deux grandes parties : la partie *UI* (User Interface ou Interface Utilisateur) et une partie *server*. La partie *UI* permet de recueillir les entrées (*input*) données par les utilisateurs, et également d'afficher les sorties (*output*) renvoyées par l'application. C'est la partie dite *"front end"*. La partie *server* correspond quant à elle au *"back end"*, c'est-à-dire qu'elle réalise les calculs pour créer les sorties à partir des entrées.

2.2 La réactivité

Dans les *Shiny Apps*, le fait de bouger un paramètre dans un onglet engendre la réactuallisation de l'ensemble des sorties de l'onglet. Parfois cela n'est pas ce que l'on souhaite. Ce phénomène peut être à éviter pour deux raisons : tout d'abord, cela risque d'engendrer des calculs inutiles et ainsi ralentir l'application. Ensuite, cela n'est parfois pas le comportement voulu. A titre d'exemple, dans l'application *Random Fields*, le quatrième onglet permet de faire varier r, le rayon de la fenêtre glissante, sans régénérer le champ initial: la grille du haut représentant le champ initial reste identique quand r varie, contrairement à la grille du bas qui se réactualise : cela est possible grace à la mise en place de fonctions réactives qui déterminent ce qui est réactualisé ou non.

2.3 Documentation

Nous pensons que les liens suivants sont intéressants pour découvrir le monde des *Shiny Apps* :

- *https://shiny.rstudio.com/tutorial/*. Ce lien donne la base du fonctionnement et permet de créer rapidement des applications.
- *https://mastering-shiny.org/*. Ce livre en ligne permet d'approfondir les notions vues sur le lien précédent et il contient notamment une section qui explique en détail le fonctionnement de la réactivité.
- *https://rstudio.github.io/shinydashboard/index.html*. Le package *shinydashboard* crée des applications dont l'interface utilisateur par défaut diffère de celle du package *shiny*. Ces interfaces

utilisateurs contiennent notamment un menu sur le côté. C'est une option de présentation intéressante (utilisée dans *Random Fields*).

- *https://golemverse.org/*. Un outil pour mieux organiser le code des *Shiny Apps*.
- *https://engineering-shiny.org/index.html*. Un livre pour penser les développement des *Shiny Apps* de manière professionelle.
- *https://www.shinyapps.io/*. Un outil qui permet de mettre en ligne des *Shiny Apps*. Quand elles restent des petits projets, le compte est gratuit (25h de consultation disponibles par mois et jusqu'à 5 applications hébergées).

Et pour aller plus loin :

- *https://rstudio.github.io/shinytest/*. Une librairie d'aide à la mise en place des tests sur les applications *shiny*, disponible sur Github .
- *https://rinterface.github.io/shinyMobile/?fbclid=IwAR3tVueaEg8SFjVC5wyLm0xbB0ZgokKi4CWraC8hMS75lVgmoi5MA6N5ZM4*. Une librairie, *shinyMobile*, pour créer des applications sur téléphone (iOS et Android). Disponible sur le CRAN.
- *https://www.youtube.com/c/ShinyDeveloperSeries/featured*. Une chaine youtube consacrée au développement des *Shiny Apps*.
- *https://www.youtube.com/watch?v=zAqoLCQ83Ns&list=PL9HYL-VRX0oRbLoj3FyL5zeASU5FMDgVe*. La playlist youtube de RStudio consacrée aux *Shiny Apps*.

3 Informatique : test du code et bonnes pratiques

Cette troisième partie aborde le sujet des tests en informatique. Elle expose tout d'abord quelques types de tests (seront présentés succintement les tests unitaires, les tests d'intégration, les tests fonctionnels ou *behavioural tests*, les tests de validation et les golden tests) et détaille les raisons pour lesquelles il est important d'écrire ces tests. Une section sera ensuite dédiée plus particulièrement aux tests unitaires pour préciser les pratiques usuelles à propos de ce qu'il est important de tester, de la manière de tester, et de la manière d'évaluer la qualité des tests.

Cette troisième partie s'appuie sur la synthèse de différentes conférences Devox (the developer community conference) et vidéos suivies sur youtube et de quelques documents supplémentaires en ligne.

Les vidéos/conférences suivies sont les suivantes :

- « Effective Unit Testing by Elliotte Rusty Harold »
  Elliotte Rusty Harold : actuellement Tech Lead de Google Cloud Tools (outils pour les développeurs de Google Cloud) pour Eclipse.
  https://www.youtube.com/watch?v=fr1E9aVnBxw
- « Write awesome tests by Jeroen Mols »
  Développeur chez PHILIPS pour l'application Philips Hue. Et développeur expert chez Android.
  https://www.youtube.com/watch?v=F8Gc8Nwf0yk
- « Victor Rentea - Unit Testing like a Pro: The Circle of Purity»
  Entre autres, lead architect chez IBM.
  https://www.youtube.com/watch?v=1Z_h55jMe-M
- « Unit tests vs. Integration tests - MPJ's Musings - FunFunFunction #55 »
  Chaîne youtube de vulgarisation à propos du code
  https://www.youtube.com/watch?v=vqAaMVoKz1c

3.1 Présentation générale des tests

Développer un programme nécessite de vérifier qu'il fonctionne et ce à différents niveaux : c'est l'objectif des tests en informatique. Il existe deux grandes catégories de tests : les tests fonctionnels et les tests non-fonctionnels. Les premiers vérifient que les fonctionnalités de l'application sont bien mises en place. Les seconds sont, pour la plupart, des indicateurs de performance : évaluation de la vitesse d'exécution du programme, du fait qu'il n'y ait pas de failles de sécurité, de la manière dont réagit le programme "sous stress" : par exemple avec des gros volumes de données en entrée, ou de nombreuses entrées en parallèles ...

Nous nous intéressons ici aux tests fonctionnels en présentant un rapide inventaire de ceux-ci. Puis nous verrons plus en détail un type de tests fonctionnels en particulier, les tests unitaires, qui sont les tests les plus bas niveau et qui sont réalisés par les personnes qui développent (les autres tests pouvant impliquer d'autres acteurs ou davantage d'acteurs).

NB :

- C'est une classification possible qui est présentée, elle n'a pas vocation à être unique ni unanime : les classifications peuvent différer un peu les unes des autres en fonction de l'organisation des projets et de la philosophie de chaque entreprise concernant la manière de tester.

- Il y a deux acceptions au terme “fonctionnel”. Il est parfois utilisé par opposition au terme “non-fonctionnel” (comme expliqué précédemment). Mais il est également utilisé pour désigner un type particulier de tests qui évaluent des scénarios, (c’est-à-dire des séries d’actions qu’un utilisateur est susceptible de réaliser). Les tests qui évaluent des scénarios portent le nom de *functional tests* ou *behavioural tests*, et appartiennent à une catégorie plus large de tests : les tests fonctionnels. Autrement dit, la grande catégorie “tests fonctionnels” (par opposition à “tests non-fonctionnels”) contient plusieurs types de tests : les tests unitaires, les tests d’intégration, les tests de validation, et les tests ... fonctionnels, ou encore les *behavioural tests*. Nous allons présenter différents types de tests fonctionnels (par opposition à non-fonctionnels), du plus bas au plus haut nivau. Ils sont représentés sur l’image suivante sous forme de pyramide, avec les tests de plus bas niveau en bas. Le nombre de tests diminue à chaque fois que l’on passe au niveau supérieur (car ils sont de plus en plus longs à réaliser et de plus en plus complexes à mettre en place) : d’où le fait que la taille des étage diminue et qu’on ait choisi une représentation en forme de pyramide.

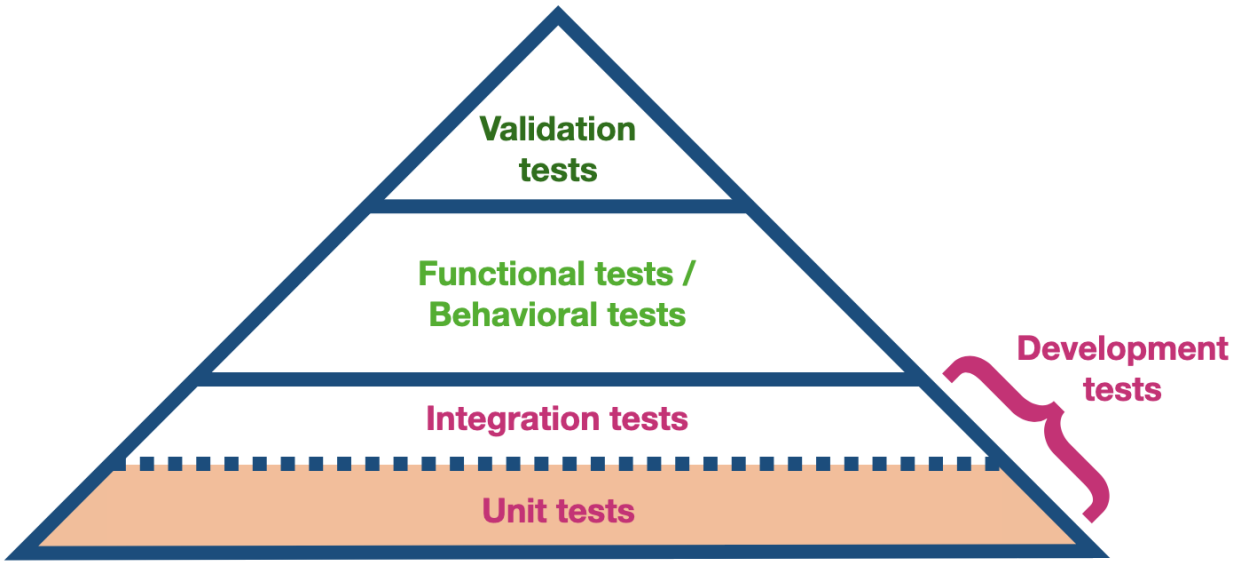


Figure 2: Les tests fonctionnels

3.2 Les tests fonctionnels

3.2.1 Tester le code : phase de développement du code (Development tests)

Tests unitaires (Unit tests)

Les tests vérifiant les opérations les plus élémentaires sont les tests unitaires. Comme leur nom l’indique, ils sont ceux qui testent les opérations unitaires du code, c’est-à-dire les opérations qu’on ne peut pas diviser en sous-opérations. Ils sont écrits tout au long de la phase de développement du code, en parallèle avec celui-ci. Le lancement des tests unitaires peut se faire à la main : il est conseillé d’exécuter le fichier de tests très régulièrement, à chaque modification du code. Mais ces tests sont aussi mis en place de façon à être réalisés de manière automatique à chaque fois que le code est actualisé en ligne sur Github par exemple (via la *github action* relative au *codecoverage* : voir la partie 1 du rapport).

Tests d'intégration

Une fois que plusieurs parties du code (par exemple des modules) sont écrites et que tous leurs tests unitaires passent avec succès, il faut vérifier que les interactions entre les parties du code fonctionnenent : c'est l'objet des tests d'intégration. A titre d'exemple (je reprends ici le modèle du youtubeur MPJ dans sa vidéo *Unit tests vs. Integration tests* (https://www.youtube.com/watch?v=vqAaMVoKz1c&t=2s), on peut imaginer une application qui comprend : une interface utilisateur, une partie servant à réaliser des calculs, un système de paiement et une base de données. Ces quatres parties peuvent être testées indépendamment avec un ensemble de tests unitaires et très bien fonctionner. En plus de cela, les tests d'intégration permettent de vérifier qu'elles fonctionnent correctement *entre elles*, c'est-à-dire lorsqu'elles font appel les unes aux autres. Il suffit que l'une d'elles donne en sortie un certain type d'objets servant d'entrée à une autre partie du code conçue pour recevoir un autre type d'entrées et les tests d'intégration ne fonctionnent plus, même si indépendemment les unes des autres, toutes les parties du code fonctionnent. Les tests d'intégration ne peuvent pas être aussi exhaustifs que les tests unitaires car cela impliquerait de tester de trop nombreuses combinaisons possibles entre les composants : il faut faire des choix.

Figure 3: Tests d'intégration

3.2.2 Tester des scénarios : tester d'un point de vue utilisateurs. Deux écoles.

Ecole 1 : les tests fonctionnels (Functional tests)

Les tests fonctionnels se placent du point de vue de l'utilisateur et consistent à imaginer des séries d'actions (scénarios) qu'il est susceptible de réaliser. Il s'agit de vérifier que ces séries d'actions fonctionnent. Dans l'application précédemment présentée, on peut vouloir vérifier le fait que si l'on souhaite payer, on puisse le faire (cela fait appel à l'interface utilisateur, la partie calcul, et le système de paiement) : l'utilisateur va cliquer sur une série de boutons et éventuellement rentrer des informations. C'est tout le déroulement de la procédure qui est testé. Comme les tests d'intégration, les tests fonctionnels ne peuvent pas être aussi exhaustifs que les tests unitaires.

Ecole 2 : Behavioural tests

Les *behavioural tests* testent également des scénarios mais diffèrent des tests fonctionnels dans leur présentation. Cela relève de la manière dont l'entreprise choisit de s'organiser : les *behavioural tests* se donnent pour missions d'être compréhensibles par tous les acteurs d'une entreprise, depuis ceux les plus éloignés du code (comme les acteurs financiers) jusqu'à ceux les plus proches du code (les personnes qui développent), en passant par tous les intermédiaires (managers, ...). Des langages de tests haut niveau ont ainsi été développés afin que les tests soient lisibles en anglais courant (structure GIVEN - WHEN - THEN), par exemple

avec le langage Cucumber. Cela ressemble à l'image suivante, issue de la vidéo Introduction to Cucumber (*https://www.youtube.com/watch?v=lC0jzd8sGIA*).

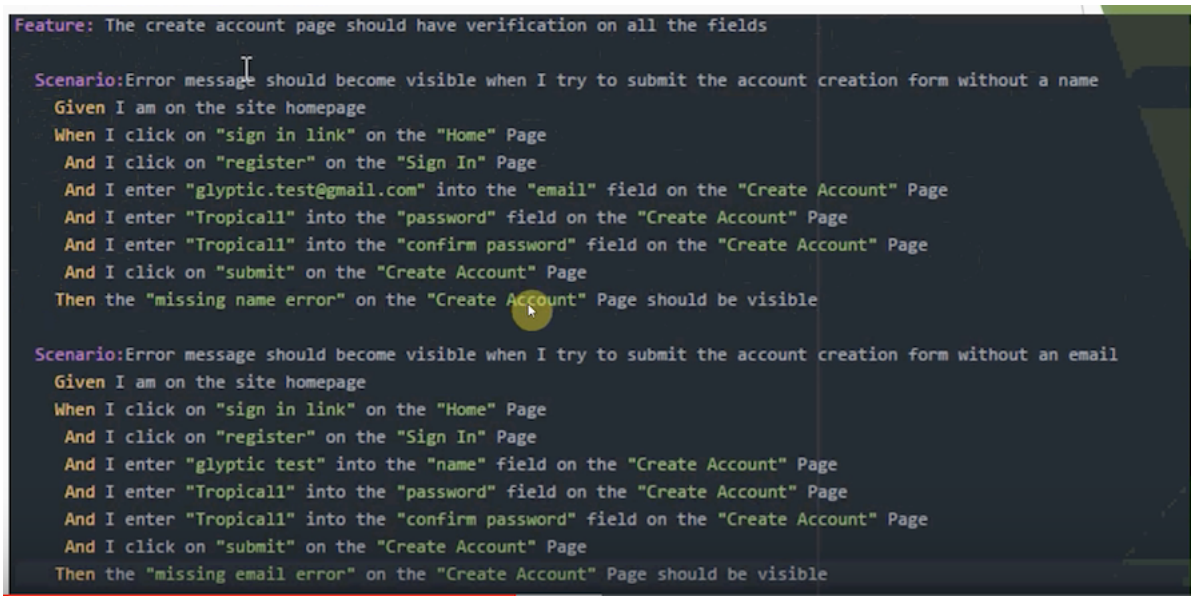


Figure 4: Introduction to Cucumber

Les *behavioural tests* diffèrent des *functional tests* car les *functionals tests* ne sont en général lisibles que par un petit nombre de personnes. Les scénarios, dans le cas des *functionals tests*, sont en effet écrits en langage courant puis traduits en termes informatiques. Dans le cas des *behavioural tests*, les scénarios sont écrits en langage courant et le restent.

3.2.3 Des tests "plus haut niveau"

Tests de validation

Ce ne sont plus à proprement parler des tests informatiques. Les tests de validation servent à vérifier que le produit construit correspond à ce que voulait le client. Si le client ne voulait pas de système de paiement, les tests de validation sont invalidés. C'est avant tout une vérification du cahier des charges.

3.2.4 Les Golden tests : un autre type de tests qui peuvent être plus ou moins haut niveau

Golden tests

Lorsque des développeurs héritent d'un code écrit des années auparavant, le terme *code legacy* est utilisé pour désigner cet ancien code. Ce code peut être testé en le considérant comme une boîte noire : seules les sorties attendues sont testées et le détail des fonctionnalités n'est pas testé (par exemple, il n'y aura pas de tests unitaires). On parle de *golden tests*. C'est une manière d'appréhender des codes volumineux de manière globale sans considérer le fonctionnement de chaque sous-partie.

Snapshots avec R

Le principe des *golden tests* a été repris dans R pour d'autres tests. Par exemple, si l'on souhaite tester une sortie complexe telle qu'un graphique ou un dataframe, les tests reposerons sur ce principe des *golden tests*, notamment à travers une commande de test : *snapshot*. Les snapshots consistent à générer un fichier extérieur au fichier de tests (par exemple un graphique) qui sera enregistré à côté du fichier de tests. Cela se passe lors de la première exécution du test. Ensuite, quand le test est à nouveau exécuté, la sortie est comparée à ce fichier enregistré. Si la sortie n'a pas changée, le test passe avec succès, sinon il retourne un échec. Pour une figure, cela évite d'avoir à utiliser des tests unitaires pour tester que chaque élément du graphique est correctement généré : la figure est directement testée de manière globale, par comparaison avec la figure contenue dans le fichier enregistré. Si un changement dans le code fait que la figure générée n'est plus la même, cela est signalé par le fait que le test ne passe plus.

3.3 Les tests unitaires

3.3.1 Motivation

Cette sections décrit les différentes raisons qui montrent l'importance des tests unitaires pour les développeuses et les développeurs.

Vérification du bon fonctionnement du code

La raison la plus évidente d'écrire des tests unitaires est d'obtenir une preuve du fait que le code fonctionne et qu'il n'y pas d'erreurs dans les sorties lors de la compilation. Cependant, ce n'est pas forcément la raison la plus importante, comme nous allons le voir dans les sections suivantes !

Refactoring

La réécriture continuelle du code fait partie intégrante des processus de développement. Cela peut être pour lui ajouter de nouvelles fonctionnalités. Mais cela peut aussi simplement servir à le rendre plus clair ou plus efficace. Dans les deux cas cela porte le nom de *refactoring*. Les tests unitaires permettent de modifier le code et de savoir facilement si celui-ci continue à fonctionner correctement après modification, autrement dit de vérifier qu'il n'y a pas eu de régression. (Le terme de *régression* est le terme consacré pour parler d'un code modifié qui contient des erreurs ou qui a perdu des fonctionnalités de manière non voulue). La première raison pour laquelle écrire des tests unitaires est donc : arrêter d'avoir peur de remanier le code.

La citation suivante de Robert C. Martin à ce sujet transmet parfaitement cette idée :

« The suite of tests is not there to prove to other people that our code works. The suite of tests is there so that we can refactor ! So that when we bring code up on our screen, we're not afraid of it. How many of you are afraid of your code ? [...] Code comes up on the screen, you look at it and your first thought is : "Uh, someone should clean it !". Your next thought is : "I'm not touching it !"

I want you to think how deeply dysfunctional it is for you to be afraid of the thing that you created.

[...] You need a suite of tests you trust with your life ! »

Les tests unitaires permettent donc de s'assurer non seulement du fait que le code fonctionne, mais surtout du fait qu'il continue de fonctionner après modifications : on peut remanier le code sans craindre les phénomènes de régression.

Point de vue utilisateur, meilleur design

Le tests unitaires, s'ils sont écrits avant de commencer de coder (nous reviendrons sur ce concept par la suite), permettent de définir précisément les fonctionnalités que l'on souhaite obtenir avec le code, en se plaçant d'un point de vue utilisateur. Ils nous font aussi garder à l'esprit les spécifications qui doivent caractériser le code. Cela a en général pour conséquence un meilleur design du code et évite ce que l'on appelle l'*over-engineering* : la quantité de code écrite n'a pas de fonctionnalités superflues.

Les tests correspondent à la documentation

Les tests unitaires peuvent être vus comme le pendant de la documentation : ils sont une série d'exemples qui montrent comment fonctionne le code. Celui-ci devient ainsi facilement utilisable et modifiable par d'autres personnes puisque son fonctionnement est particulièrement explicite grâce au fichier de tests.

Gagner du temps

Sur du moyen à long terme, les tests unitaires représentent un gain de temps considérable puisque le code devient beaucoup plus facile à maintenir, pour toutes les raisons évoquées précedemment.

3.3.2 Les tests unitaires en pratique

Cette partie détaille comment écrire les tests et que tester.

Principe et structure

Le principe fondamental est de vérifier qu'une entrée connue et fixée produit la sortie connue et fixée attendue. Cela implique que les tests aient des sorties claires et assez évidentes.

En terme de structure, le modèle est le suivant :

ARRANGE

Partie setup : la préparation nécessaire au test est faite dans cette partie.

ACT

Appel le code de production

ASSERT

Vérifie que la sortie est celle attendue

Exemple :

Nous reprenons ici l'exemple (réécrit en R) donné sur le site *Automation Panda* (https://automationpanda.com/2020/07/07/arrange-act-assert-a-pattern-for-writing-good-tests/).

test_that("abs function works for a negative number", {

#ARRANGE

negative ⟵ **-5**

#ACT

actual ⟵ **abs**(negative)

#ASSERT

expected ⟵ **5**

expect_equal(actual, expected)

})

Que tester ?

Nous proposons de classer les tests unitaires en deux grands types : les tests de régularité et les tests d'anomalies. Les premiers doivent tester que le code se comporte bien comme attendu quand il est utilisé correctement. Les seconds vérifient l'apparition d'erreurs quand le code n'est pas appelé correctement.

Tests de régularité

Les tests de régularité vérifient que les sorties générées par le code correspondent à celles attendues. Ils sont structurés comme présenté dans la partie **Principe et structure**. Nous prenons en exemple la fonction `variance()` qui retourne la variance non-corrigée d'une série de nombre.

```
variance <- function(M){
 if (!is.vector(M) & !is.matrix(M)){stop("M must be a number/vector/matrix/")}
 if (!is.numeric(M)){stop("M must contain numeric values")}
 return(mean(M**2, na.rm = TRUE) - mean(M, na.rm = TRUE)**2)
}
```

Nous exposons ici une liste des différents cas de figure que nous avons rencontrés en créant ces tests et en nous renseignant à propos des éléments du code à tester. Encore une fois, il ne s'agit pas d'une classification absolue mais d'une proposition.

- Les tests particuliers dont les sorties sont très faciles à prévoir.

Par exemple : on peut tester la variance d'une série de zéros ou d'une série constante et s'attendre à avoir zéro en retour. On peut également tester la variance de la série c(1, 2, 3, 4) et s'attendre à avoir 1,25 en retour. Et cætera.

- Les tests plus généraux dont les sorties restent faciles à prévoir.

Par exemple : on peut tester la variance d'une série statistique générée aléatoirement mais toujours avec le même noyau (il s'agit donc toujours de la même série). Pour cela on utilise la fonction `set.seed()`. On calcule la sortie attendue (partie **expected** des tests) et on vérifie qu'il s'agit bien de la sortie renvoyée par le code (partie **actual** des tests).

- Les tests invoquant des valeurs extrêmes ou des valeurs pouvant poser problème.

Par exemple : on peut tester la variance d'une série statistique qui contient le plus grand ou le petit nombre de type *double* représentable par la machine sur laquelle le code est exécuté.

Avec R

Avec R, on utilise les commandes `.Machine\$double.xmax` *et* `.Machine\$double.xmin` *pour récupérer respectivement le plus grand et le plus nombre de type* double. *Pour le type* integer, *ce sont les commandes* `.Machine\$integer.max` *et* `.Machine\$integer.min`.

Dans le cas d'un nombre à multiplier (par un nombre >1), on peut empêcher l'utilisateur de rentrer un nombre dont la sortie dépasserait la limite (la limite est le plus grand *double* possible atteignable sur la machine), ou simplement savoir et décider que la machine retournera "Inf".

Si le passage par des nombres trop grands ne représente qu'une étape dans les calculs avant de ré-obtenir des plus petits nombres en sortie, on peut travailler en échelle logarithmique et revenir à l'échelle originale en sortie (en prenant l'exponentielle).

Parmi les valeurs pouvant poser problèmes, on notera aussi les valeurs proches de zéros pour les divisions.

Pour les chaînes de caractères, on peut tester des chaînes avec des caractères spéciaux, une chaîne vide ou encore une chaîne très longue.

- Les tests qui vérifient les différentes entrées que l'on peut prendre (ils peuvent se recouper avec les précédents).

Par exemple, est-ce que la fonction calcule la variance si les nombres sont dans une matrice ? Un dataframe ? Si oui, ces types d'entrées sont à tester dans les tests de régularité. Sinon, ils sont à tester dans les tests d'anomalies.

```
library(testthat)
test_that("Regular test - variance of a vector with NA", {
 # arrange
 v <- c(1, 2, 3, 4, 5, NA)
 # act
 actual <- variance(v)
 # assert
 expected <- (1**2)*2/5 + (2**2)*2/5
 expect_identical(actual, expected)
})
```

Un exemple de test de régularité

Tests d'anomalies

Les tests d'anomalies vérifient qu'une mauvaise utilisation du code retourne une erreur, et que celle-ci est appropriée.

On peut tester :

- les entrées qui ont des types non-conformes à ceux attendus,
- les entrées qui ont des valeurs numériques trop grandes (voir section précédente) / trop proches de zéro, etc,
- les problèmes liés à une mauvaise orthographe des paramètres de la part de l'utilisateur (par exemple *inut = 2* ou lieu de *input = 2*),
- les problèmes liés au fait qu'il manque un des paramètres d'entrée,
- les situations qui n'empêchent pas le code de tourner mais qui peuvent renvoyer des warnings : par exemple si certaines valeurs d'une série sont en réalité des valeurs manquantes.

La structure des tests d'anomalies diffère de la structure des tests de régularité car les parties ACT et ASSERT sont réunies. Voici un exemple :

```
test_that("Anomaly test - error when the parametre is a function", {
 # arrange
 f <- function(){return("I am a function")}
```

```
 # act & assert
 expect_error(variance(f), "^M must be a number/vector/matrix/dataframe$")
})
```

Un exemple de test d'anomalies

La présence de l'accent circonflexe au début du message d'erreur et du dollar à la fin permet de faire en sorte que la comparaison entre le message du *stop* (dans la fonction) et de celui dans *except_error* se fasse correctement (problématique à propos des expressions régulières).

Tester par ordre de priorité

Cette section mentionne les éléments du code à tester, des plus importants aux moins importants (éléments issus de la conférence "Unit Testing like a Pro: The Circle of Purity", https://www.youtube.com/watch?v=1Z_h55jMe-M) :

Si le code ne comprend aucun test unitaire, quelles parties tester en priorité ? "Le code qui vous fait peur !", répondront les personnes expérimentées. Autrement dit, le code difficile à comprendre, les parties qui semblent critiques.

Ensuite, le code difficile à tester à la main : celui qui prendrait plusieurs minutes à être testé si on étudiait son comportement via une interface utilisateur.

Puis viennent les bugs. Si des bugs sont trouvés, la meilleure pratique reste d'écrire des tests unitaires d'abord, et ils doivent échouer. Ensuite, il est possible de résoudre le bug. Les tests devraient alors passer avec succès. Cela rejoint la pratique du TDD (Test Driven Development) expliquée par la suite.

Il est aussi conseillé de tester les structures logiques du type For/If/While.

Ensuite, de lever les exceptions utiles, c'est-à-dire, globalement, réaliser les tests d'anomalies qui doivent arrêter le code en renvoyant le bon message d'erreur, ou le laisser continuer à s'exécuter, mais avec un warning.

Puis, de tester les méthodes qui en appellent d'autres.

En avant-dernier lieu, on peut tester le code trivial.

Et enfin, on peut tester le *code legacy*, c'est-à-dire l'éventuel ancien code dont on a pu hériter, qui ne contient *a priori* pas d'erreurs.

Propriété des tests

Nous avons vu quels éléments tester et comment les tester. Nous allons maintenant rendre explicites les propriétés très importantes que doivent remplir les tests unitaires. Il faut qu'ils soient :

- Sensibles, c'est-à-dire qu'ils échouent s'il y a des bugs (au moins certains d'entre eux). Il est problématique d'avoir des tests qui passent toujours avec succès (nommés *evergreen tests*). En effet, ces tests ne testent finalement rien. Pour éviter cela, il est conseillé de faire échouer les tests au moins une fois : soit avant d'écrire le code correspondant, soit après avoir écrit le code mais en modifiant légèrement celui-ci pour vérifier que les tests échouent bien si le code n'est plus fonctionnel. "Don't write evergreen tests !"
- Spécifiques. Un test doit échouer pour une raison précise. Il ne doit pas pouvoir échouer pour plusieurs raisons différentes (c'est le principe du fait qu'il soit unitaire). Son titre mentionne ce qui est testé. De plus, une seule assertion par test est censée être testée. Il est ainsi déconseillé de mettre plusieurs lignes *expect_identical(actual, expected)* (ou autres) au sein d'un même test.
- Isolés en termes de fonctionnement. Les tests ne doivent pas dépendre les uns des autres. Un test n'est pas censé dépendre d'une valeur actualisée dans un test le précédant. Qui plus est, lorsque des procédés

de multithreading sont mis en place, les tests peuvent être exécutés dans n'importe quel ordre : il est nécessaire qu'ils soient indépendants. La non-indépendance peut entraîner des phénomènes dits de "flakiness" (traduisible par friabilité) : parfois les tests passent avec succès et parfois ils échouent, sans raison apparente. Ne rendez pas folles les personnes qui développent.

- Isolés en termes de dépendance temporelle. Autrement dit, s'ils utilisent le temps, les tests ne doivent dépendre que d'une unique base de temps. Il ne faut pas des appels différents à des fonctions temporelles dans un même test : si l'exécution des différentes lignes a lieu de manière suffisamment proche dans le temps, le test peut passer avec succès. Il peut échouer dans le cas contraire. Là encore, on a un problème dit de flakiness. Cette remarque peut être étendue au cas où il y a de l'aléatoire dans les tests : il faut bien évidemment fixer un noyau (via set.seed()), sinon le test risque de passer ou non de manière aléatoire. Un autre cas de flakiness peut être rencontré quand les tests passent ou non selon les supports (système d'exploitation) utilisés. Et enfin lorsque les précisions machines ne sont pas les mêmes d'une machine à l'autre (les tests d'égalité entre des nombres peuvent différer).
- Rapides. Les tests unitaires sont appelés à être exécutés de façon automatique de très nombreuses fois au cours des phases de développement. On peut estimer qu'un test doit tourner au maximum en l'espace d'une seconde et que l'ensemble des tests doit tourner au maximum en une minute. Sans quoi les développeuses et développeurs risquent de ne tout simplement plus lancer les tests. Si des tests unitaires sont plus longs malgré tout, il est conseillé de les lancer en dernier.
- Pas trop nombreux. Pour pouvoir tenir les contraintes de temps qui viennent d'être citées d'une part, et pour pouvoir assurer la maintenance des tests d'autre part. En effet, le fait de remanier le code peut impliquer la modification, l'ajout ou la suppression de tests unitaires. De plus, il existe des écoles qui précisent que dans les classes, les méthodes privées ne sont pas censées être testées, seulement les méthodes publiques.
- Sans condition logique en leur sein.

Remarques supplémentaires :

- Si un test fait appel à une base de données, cela peut compliquer son exécution et également augmenter son temps d'exécution. Voir les techniques de "database mocking" pour ce genre de problématiques.
- Les tests faisant appel à des bases de données peuvent aussi échouer si la base de données a un problème au moment où ils sont exécutés : cela ne vient pas forcément du code.
- Il existe des procédures de "random testing" où des tests sont générés automatiquement et avec des entrées aléatoires.

3.3.3 Évaluation des tests

3.3.3.1 Code coverage

Une manière d'évaluer les tests est de considérer quelle proportion de code est testée. Des outils tels que *codecov* (pour code coverage) analysent le nombre de lignes de code exécutées lors des tests et divisent par le nombre total de lignes de code, renvoyant ainsi un pourcentage. Avoir une bonne couverture du code (voire une couverture de 100%) est donc un prérequis pour s'assurer que le code est bien testé : si une ligne de code n'est pas touchée par les tests, elle ne risque pas d'être testée.

À titre indicatif des ordres de grandeurs que l'on peut vouloir obtenir, voici les commentaires de Elliotte Rusty Harold (présenté au début de cette troisième partie du rapport : actuellement Tech Lead de Google Cloud Tools – outils pour les développeurs de Google Cloud – pour Eclipse). Il indique que Google Cloud Tools possède une couverture de code de 70%.

"*That's a little bit low, [...] it should be higher*"

A l'inverse, la librairie indépendante XOM qu'il a développée possède 99% de code coverage :

"I'm very proud of that."

On recherche donc à ce que la couverture du code soit maximale. Cependant, une couverture de 100% n'assure pas pour autant de la qualité des tests. Les tests peuvent en effet faire appel à l'ensemble du code sans tester certaines fonctionnalités importantes.

"Agnostic about the quality of tests" (Jeroen Mols)

"Unaware of quality coverage" (Jeroen Mols)

Encore une fois, la couverture du code est davantage un prérequis qu'une garantie de la qualité des tests.

On peut ajouter la remarque suivante : les tests unitaires ne sont pas écrits pour prouver au monde que le code fonctionne, par l'intermédiaire du pourcentage élevé de la couverture du code.

"Why the hell do we write unit tests ? To have coverage right ? Noooooo, for god sake, noooo, noooo !" - Victor Rentea

Voir la section **Motivation** pour la réponse à cette question.

De plus, un test avec une couverture de 100% peut-il contenir des bugs ? Bien sûr ! Alors de quels autres outils disposons-nous ? Les tests de mutation !

3.3.3.2 Mutation testing

Pour savoir si les tests couvrent bien le code qu'ils disent qu'ils couvrent ... faites-les échouer ! C'est la philosophie des tests de mutation. Ils consistent à effectuer des changements dans le code de production. On obtient ainsi un code dit mutant. Quand on lance les tests, ceux-ci doivent échouer (au moins un ou certains).

"The tests should squash the mutant by turning red !" - Victor Rentea

Si aucun test n'échoue face à un code de production altéré, c'est que les tests ne couvrent pas vraiment les problèmes qui peuvent apparaitre. Cela représente un meilleur outil de mesure de la qualité des tests que le pourcentage de couverture.

Il est à noter que cette procédure peut être réalisée à la main : on change le code à un endroit, on vérifie qu'au moins un test échoue, puis on remet le code dans son état initial.

Cependant, il existe aussi certaines procédures qui permettent d'altérer le code de façon automatique. La qualité des tests est alors évaluée par un pourcentage qui donne la proportion de code mutant détecté. C'est le cas de certaines procédures en JAVA par exemple (voir par exemple le site Real world mutation testing (https://pitest.org/). En R les packages exécutant ce genre de procédure semblent pour l'instant en développement.

Enfin, pour conclure sur cette partie concernant la qualité des tests, nous précisons qu'il est aussi admis que la qualité des tests repose sur la manière dont ils ont été pensés et codés et que cela n'est pas complètement mesurable quantitativement.

3.3.4 Le Test Driven Development

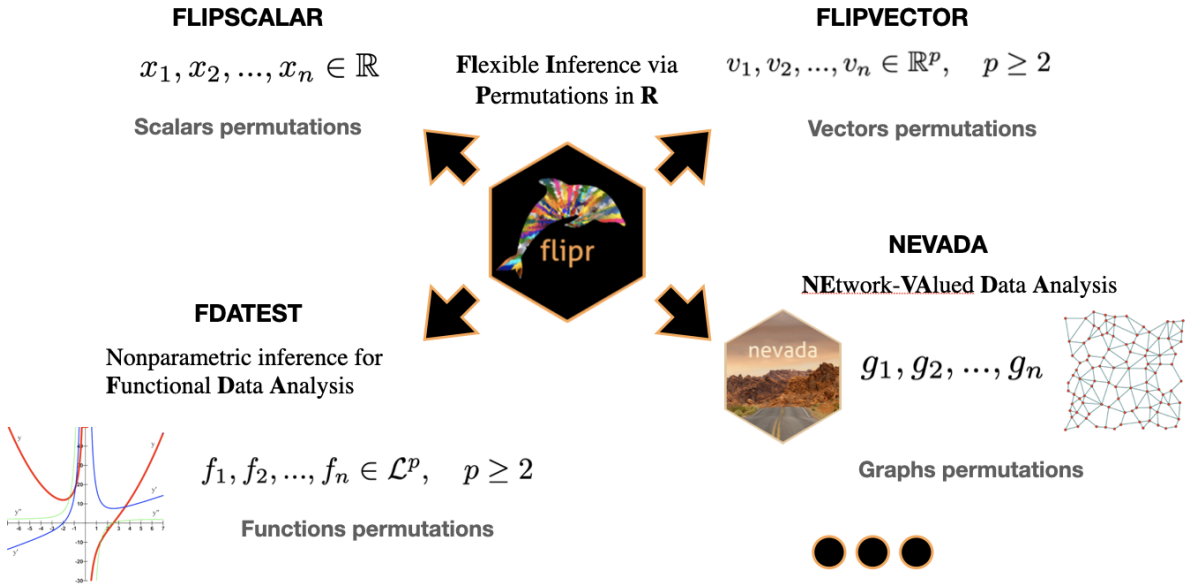
Impossible de conclure une discussion sur les tests unitaires sans parler du TDD : le Test Driven Development. C'est une pratique qui consiste à écrire les tests unitaires avant le code, puis écrire le minimum de code pour faire passer le test, puis remanier le code (refactoring). Cela est considéré comme une très bonne pratique, bien que cela fasse aussi partie du "extreme programming" : c'est une pratique de développement qui peut aussi être vue comme assez extrême ...

4 Application : le package R *flipr*

Un des buts du stage était de mettre en place des tests unitaires sur le package *flipr* qui fait de l'inférence statistique par permutation.

4.1 Contexte du projet : *flipr* et *flippy*

Le package *flipr* permet de faire de l'inférence statistique par permutation sur différents objets mathématiques : des scalaires, des vecteurs, des fonctions, des graphes ... Il a aussi vocation à être exporté en Python.



La documentation de *flipr* est disponible à cette adresse https://astamm.github.io/flipr/index.html.

4.2 Explication sur deux exemples scalaires et leurs tests unitaires

Nous avons commencé à mettre en place des tests unitaires sur la partie qui réalise des statistiques sur des séries de scalaires.

4.2.1 Premier exemple

flipr propose tout d'abord d'entrer deux séries de nombre, ou deux échantillons. Par exemple (l'exemple s'appuie sur des nombres générés à partir d'une loi normale mais cela pourrait être n'importe quelle série de nombres):

```
n <- 15
x1 <- rnorm(n = n, mean = 0, sd = 1)
x2 <- rnorm(n = n, mean = 1, sd = 1)
```

On peut montrer que la variable X_2 peut s'écrire en fonction de X_1 de la façon suivante : $X_2 = \delta + X_1$. Il s'agit ensuite de définir un paramètre à estimer, par exemple la différence des espérances $\delta = \mu_1 - \mu_2$, où μ_1 et μ_2 représentent les espérances des lois à partir desquels les échantillons x_1 et x_2 sont générés. La fonction de plausibilité, qui associe une p-valeur à un δ donné, peut être tracée. Elle est calculée par un procédé de permutations des valeurs entre x_1 et x_2. Si on examine la p-valeur pour l'hypothèse H_0 : $\delta = 0$, on cherche à savoir si les séries ont la même espérance ou non (on peut fixer le seuil à 0.05 et on conclura que les séries n'ont pas la même espérance si la p-valeur est en-dessous de 0.05). Ici, pour un δ de 1 ($\mu_1 - \mu_2 = 1 - 0 = 1$), on s'attend à avoir une p-valeur élevée (proche de 1). En réalité la valeur la plus élevée de la fonction de plausibilité sera obtenue pour $\delta = \overline{x_2} - \overline{x_1}$ (avec $\overline{x_2}$ et $\overline{x_1}$ les moyennes empiriques des séries x_1 et x_2). Elle sera très proche de 1.

La fonction `null_spec()` permet de ramener les deux séries à la même espérance sous H_0. Comme $\mu_1 = \mu_2 - \delta$, on peut prendre l'ensemble des éléments de la série x_2, représentés par y et par le $.x$ dans la fonction `null_spec()` (le $.x$ appelle tous les élément de y), et leur retrancher un paramètre, `parameters`, qui ici correspondra à δ.

```
null_spec <- function(y, parameters) {
 purrr::map(y, ~ .x - parameters)
}
```

On peut alors appeler ou définir la fonction *t.test* pour faire un test de Student d'égalité des moyennes. C'est l'objet des lignes suivantes :

```
 my_t_stat <- function(data, indices) {
   n <- length(data)
   n1 <- length(indices)
   n2 <- n - n1
   indices2 <- seq_len(n)[-indices]
   x1 <- unlist(data[indices])
   x2 <- unlist(data[indices2])
   stats::t.test(x = x1, y = x2, var.equal = TRUE)$statistic
 }

 stat_functions <- list(my_t_stat)
```

Enfin on précise que ce qu'on souhaite tester comme hypothèse sur nos paramètres :

```
stat_assignments <- list(delta = 1)
```

Il est alors possible de définir la fonction de plausibilité (p-valeur en fonction de δ) et de l'appeler pour connaître ce qu'elle vaut en une valeur donnée (ici $\overline{x_2} - \overline{x_1}$). On peut comparer cette valeur à une valeur minimale attendue (dans `expected`, ici 0.99).

Le test unitaire que nous avons mis en place se présente donc ainsi :

```
test_that("Regular test - Documentation example", {
 # Arrange
 set.seed(123)
 n <- 15
 x1 <- rnorm(n = n, mean = 0, sd = 1)
 x2 <- rnorm(n = n, mean = 1, sd = 1)
 d <- mean(x2) - mean(x1)
 null_spec <- function(y, parameters) {
   purrr::map(y, ~ .x - parameters)
 }
 my_t_stat <- function(data, indices) {
   n <- length(data)
   n1 <- length(indices)
   n2 <- n - n1
   indices2 <- seq_len(n)[-indices]
   x1 <- unlist(data[indices])
   x2 <- unlist(data[indices2])
   stats::t.test(x = x1, y = x2, var.equal = TRUE)$statistic
 }

 stat_functions <- list(my_t_stat)
 stat_assignments <- list(delta = 1)

 pf <- PlausibilityFunction$new(
   seed = 1234,
   null_spec = null_spec,
   stat_functions = stat_functions,
   stat_assignments = stat_assignments,
   x1, x2
 )

 # Act
 actual <- pf$get_value(d)

 # Assert
 expected <- 0.99
 expect_gt(actual, expected)
})
```

La fonction `expected_gt()` (**g**reater **t**han) renvoie *TRUE* si son deuxième argument est supérieur au premier. Cela constitue un premier test unitaire.

4.2.2 Deuxième exemple

Repartons de deux séries statistiques, cette fois générées à partir de lois normales ayant des moyennes ET des écarts-types différents (exemple plus général que le précédent) :

```
n <- 15
x1 <- rnorm(n = n, mean = 0, sd = 1)
x2 <- rnorm(n = n, mean = 1, sd = 4)
```

On peut montrer que la variable X_2 peut s'écrire en fonction de X_1 de la façon suivante :

$X_2 = \delta + \rho X_1$

ou encore :

$X_1 = \frac{X_2 - \delta}{\rho}$

C'est cette équation que l'on rentre dans la fonction `null_spec` :

```
null_spec <- function(y, parameters) {
   purrr::map(y, ~ (.x - parameters[1])/parameters[2])
 }
```

On cherchera alors à estimer les paramètres δ et ρ. On dispose pour cela des équations suivantes (déduites en prenant l'espérance puis la variance dans l'équation précédente) :

$\mathbb{E}[X_2] = \delta + \rho \mathbb{E}[X_1]$

$\mathbb{V}[X_2] = \rho^2 \mathbb{V}[X_1]$

ou encore :

$\mu_2 = \delta + \rho \mu_1$

$\sigma_2 = \rho \sigma_1$

On peut finalement écrire :

$\delta = \mu_2 - \frac{\sigma_2}{\sigma_1} \mu_1$

et

$\rho = \frac{\sigma_2}{\sigma_1}$

On pourra alors tracer la fonction de plausibité, c'est-à-dire la p-valeur en fonction de δ et ρ. On utilisera les deux statistiques de test de Student et de Fisher pour l'égalité des moyennes et des variances.

```
stat_functions <- list(stat_t, stat_f)
```

On précise dans la fonction `stat_assigment()` qu'il y a deux paramètres, et que ρ est le deuxième :

```
stat_assignments <- list(delta = c(1,2), rho = 2)
```

Et on a réalise le test unitaire suivant :

```
test_that("Snapshot test - Two normal distributions with different means and variances", {
 # Arrange
 set.seed(123)
 n <- 15
 x1 <- rnorm(n = n, mean = 0, sd = 1)
 x2 <- rnorm(n = n, mean = 1, sd = 4)
 null_spec <- function(y, parameters) {
   purrr::map(y, ~ (.x - parameters[1])/parameters[2])
 }
 stat_functions <- list(stat_t, stat_f)
 stat_assignments <- list(delta = c(1,2), rho = 2)

 pf <- PlausibilityFunction$new(
   seed = 1234,
   null_spec = null_spec,
   stat_functions = stat_functions,
   stat_assignments = stat_assignments,
   x1, x2
 )

 # Act
 actual <- pf$get_value(c(1, 4))

 # Assert
 expect_snapshot(actual)
})
```

Dans la section *#Act*, on évalue la fonction de plausibilité en $\delta = 1$ et $\rho = 4$:

```
 # Act
 actual <- pf$get_value(c(1, 4))
```

On s'attend donc a avoir une p-valeur élevée. Cependant, dans ce cas à deux paramètres, elle n'est plus très proche de 1 (en réalité on obtient une valeur de 0.36). On capture cette valeur grâce à un snapshot qui stocke la valeur obtenue dans un fichier à part la première fois qu'il est exécuté :

```
expect_snapshot(actual)
```

Cette valeur servira simplement de point de comparaison : si le code est plus tard modifié, on vérifiera que l'on obtient toujours cette même sortie (passé la première exécution du test, la valeur obtenue dans *actual* est simplement comparée à celle obtenue lors de la première exécution). Par conséquent, on ne calcul pas dans ce type de test la valeur attendue. Il est donc différent du précédent où l'on entrait un seuil minimum de 0.99 pour la p-valeur et où l'on vérifiait cette condition pour valider le test. Ici la validation se fait par comparaison avec la valeur retournée lors de la première exécution du test. On s'assure simplement de la constance de ce retour.

Remarque : Les p-valeurs les plus élevées sont obtenues lorsque l'on prend les valeurs des estimateurs naturels pour δ et ρ, c'est-à-dire $\overline{x_2} - \frac{s_{n-1}^{(2)}}{s_{n-1}^{(1)}}\overline{x_1}$ pour δ et $\frac{s_{n-1}^{(2)}}{s_{n-1}^{(1)}}$ pour ρ.

Ces premiers tests nous ont permis de commencer à tester *flipr*, ce qui était un des objectif du stage.

Conclusion

J'ai pu au cours de ce stage découvrir la mise en place de packages sous R, la création d'une *Shiny App*, le vaste monde des tests en informatique et les applications possibles avec *flipr* ! Et également quelques bonnes pratiques en programmation telles que l'intégration continue (CI), le pair-reviewing, le fait de remanier continuellement le code pour l'améliorer (refactoring), le fait de réaliser des tests (voire de faire du Test Driven Development !). C'était très intéressant. Je remercie chaleureusement mon encadrant Aymeric Stamm !