Julia

le langage de programmation pour le calcul scientifique

(En cours de rédaction – Version en date du jour)

Xavier Gandibleux

7 juin 2019

Ce document est extrait d'un livre en français consacré au langage de programmation julia. Il s'adresse

- aux personnes débutantes en programmation qui viennent chercher un premier langage convivial, moderne, efficace et présentant une syntaxe proche du formalisme des algorithmes ;
- aux scientifiques qui cherchent à découvrir ou migrer vers un langage de programmation open source et gratuit, alternatif à Python, C/C++, Fortran, Mathlab ou encore Octave ;
- aux spécialistes de l'optimisation qui souhaitent utiliser ou implémenter des algorithmes de résolution avancés avec un langage spécialement adapté aux besoins du calcul scientifique.

Le livre est en préparation, son premier chapitre est disponible. Il vise à apporter au lecteur les clef pour démarrer avec julia. Les chapitres en rédaction développent les sujets suivants :

Première partie (tout public) :

- calculer en julia
- programmer en julia
- fonctions en julia

Seconde partie (public ciblé) :

- modéliser avec JuMP
- optimiser avec un objectif
- optimiser avec plusieurs objectifs

Bonne lecture et n'hésitez pas à me faire connaître par email à `Xavier.Gandibleux@univ-nantes.fr` votre avis sur ce premier écrit, vos remarques, vos attentes, ...

Xavier Gandibleux
Nantes 2019

Table des matières

1 Démarrer avec Julia **7**
1.1 Préliminaires . . . 7
1.1.1 Utiliser Julia installé en local sur son ordinateur . . . 7
1.1.2 Utiliser Julia à distance à l'aide d'un navigateur web . . . 9
1.1.3 Utiliser un éditeur de texte ou un IDE pour Julia . . . 12
1.2 Environnement de travail Julia . . . 14
1.2.1 "REPL" ou mode "Julia" . . . 14
1.2.2 Mode "help" . . . 14
1.2.3 Mode "shell" . . . 14
1.2.4 Mode "package" . . . 15
1.3 Ecrire du code Julia . . . 15
1.3.1 Eléments d'un code Julia . . . 15
1.3.2 Identifiant en Julia . . . 16
1.3.3 Séparateur d'instructions . . . 16
1.3.4 Indentation du programme . . . 17
1.4 Modules et packetages . . . 17
1.4.1 Déclaration d'un module . . . 17
1.4.2 Installation d'un package . . . 17
1.5 Exécuter un code Julia . . . 19
1.5.1 Exécuter dans le REPL du code à la volée . . . 19
1.5.2 Exécuter dans le REPL un code enregistré dans un fichier source . . . 19
1.5.3 Comprendre comment s'exécute un code Julia . . . 20

Table des figures

1.1 Page sur le site Julialang qui rassemble les installations prêtes pour être déployées sous différents systèmes d'exploitation. 8
1.2 Archive téléchargée prête pour l'installation en local. 8
1.3 Application Julia prête pour l'utilisation en local. 8
1.4 Le REPL tel qu'il se présente à l'utilisateur lorsque l'application Julia est invoquée. 8
1.5 Page d'accueil de la JuliaBox. 9
1.6 Gestionnaire de fichiers sous JuliaBox et menu d'activation des applications. 10
1.7 Editeur de texte de JuliaBox. 10
1.8 Terminal sous JuliaBox dans lequel l'application Julia a été invoquée, REPL prêt. 11
1.9 Notebook Jupiter sous JuliaBox. 11
1.10 Vue principale de l'éditeur de texte Atom, présentant l'espace' "Settings" (espace au centre-droit de la fenêtre) et l'espace "Install Packages" permettant l'ajout de plug-ins (espace à droite de la fenêtre). 12
1.11 Vue principale de l'éditeur de texte Atom. 12
1.12 Vue principale de l'environnement de développement intégré Juno sous Atom : icônes de raccourcis d'actions (gauche), gestionnaire de fichiers (espace "Project"), éditeur de texte (haut-centre), tracé graphique (haut-droite), REPL (bas). 13

Chapitre 1

Démarrer avec Julia

Ce chapitre aborde trois volets préliminaires au développement de programmes en Julia. Le premier volet concerne le choix et la mise en place de l'environnement de travail. En particulier, les étapes d'installation d'applications sont décrites. La compréhension des interactions entre l'utilisateur et l'environnement Julia fait l'objet du second volet. Enfin, le troisième volet amène l'utilisateur à se familiariser avec les mécanismes offerts par l'environnement pour exécuter du code Julia.

Au terme de ce chapitre, le lecteur sera capable d'installer un environnement de travail Julia en local pour une utilisation sur un ordinateur personnel, mais aussi de manipuler l'environnement JuliaBox pour une utilisation sur le cloud. Il saura travailler avec les quatre modes de fonctionnement du REPL qui est l'environnement interactif de Julia. Il sera capable d'exécuter du code Julia via l'écriture à la volée et via l'écriture dans un fichier source.

1.1 Préliminaires

1.1.1 Utiliser Julia installé en local sur son ordinateur

Utiliser Julia en local sur son ordinateur permet d'obtenir les performances optimales du langage. Dans ce cas, l'exécution d'un algorithme proprement codé en Julia offrira des performances proches du langage C. Ce mode d'utilisation s'impose pour un usage régulier de Julia.

Julia est disponible sur la plupart des systèmes d'exploitation courants dont macOS, linux et windows. Pour disposer d'une distribution complète de Julia en local sur son ordinateur, il est nécessaire de procéder à son installation, opération à réaliser une seule fois. Les archives prêtes pour une installation aisée pour les différents systèmes d'exploitation sont disponibles sur le site web Julia, à l'adresse `http://julialang.org/downloads/` (figure 1.1). La procédure d'installation et de lancement de Julia est sensiblement différente d'un système d'exploitation à un autre. Dans tous les cas, les manipulations à réaliser restent simples.

Etapes d'une installation complète de l'application Julia sur macOS :

1. télécharger l'archive disponible pour macOS à l'adresse `http://julialang.org/downloads/` ;
2. décompresser l'archive en cliquant sur celle-ci, une icône Julia apparaît (figure 1.2) ;
3. ranger l'application en déplaçant l'icône à l'endroit de votre choix sur votre ordinateur.

L'installation de l'application Julia est terminée.

Etapes pour lancer l'exécution de l'application Julia sur macOS :

1. invoquer l'application en cliquant sur l'icône Julia (figure 1.3) ;
2. un terminal s'ouvre et présente le REPL à l'utilisateur (figure 1.4 et section 1.2.1).

L'application Julia est prête pour être utilisée.

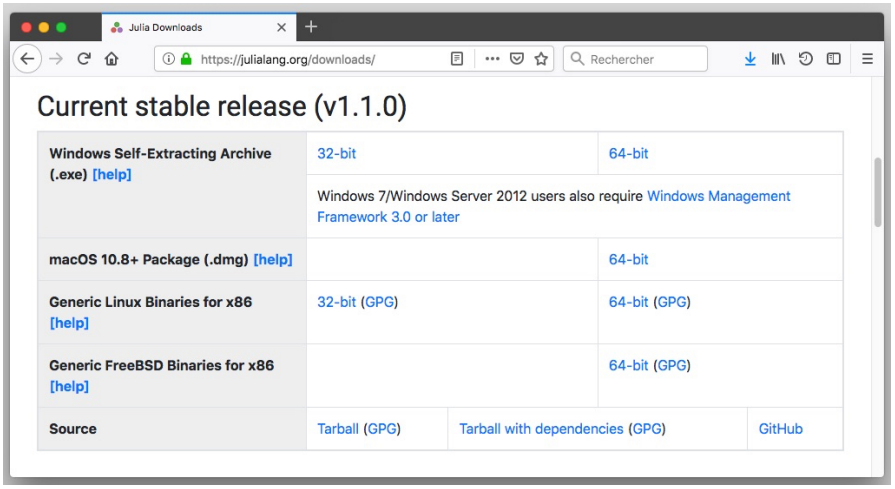

FIGURE 1.1 – Page sur le site Julialang qui rassemble les installations prêtes pour être déployées sous différents systèmes d'exploitation.

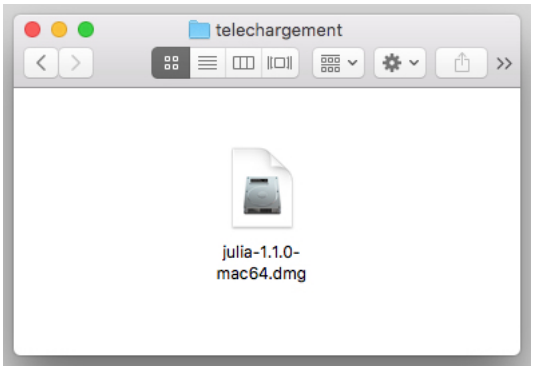

FIGURE 1.2 – Archive téléchargée prête pour l'installation en local.

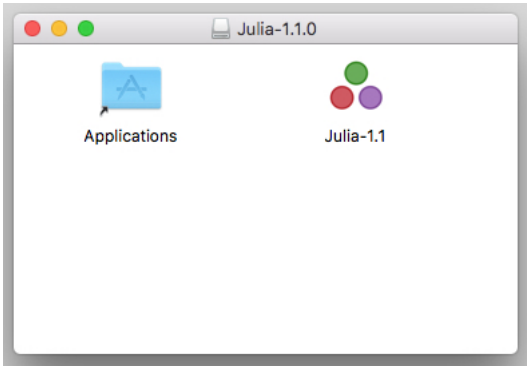

FIGURE 1.3 – Application Julia prête pour l'utilisation en local.

FIGURE 1.4 – Le REPL tel qu'il se présente à l'utilisateur lorsque l'application Julia est invoquée.

A noter qu'une installation peut être aisément personalisée, permettant par exemple d'invoquer l'application Julia en ligne depuis un terminal. De même, il est possible d'exécuter un code Julia depuis un terminal en ligne de commande sans lancer préalablement l'application Julia. Ces manipulations nécessitent une certaine aisance avec l'environnement système de l'ordinateur et présentent des spécificités d'un système d'exploitation à l'autre.

1.1.2 Utiliser Julia à distance à l'aide d'un navigateur web

Utiliser Julia à distance se fait sous JuliaBox à l'aide d'un navigateur web. Cet environnement permet de démarrer immédiatement sans aucune installation. Dans ce cas, les programmes sont exécutés sur le cloud par un serveur distant. Ce mode d'utilisation est idéal pour faire ses premiers pas de programmation en Julia, ainsi que pouvoir accéder à tout moment à l'environnement de production de Julia. En contrepartie, l'exécution d'un programme peut être ralentie du fait de (1) la charge instantanée du serveur et (2) la communication entre le serveur et l'ordinateur utilisé.

Etapes pour accèder à la JuliaBox :

1. se rendre à l'adresse `https://juliabox.com/` depuis un navigateur ;
2. se connecter avec une des méthodes proposées (figure 1.5).

La JuliaBox est active. Elle met à disposition :

1. un gestionnaire de fichiers (figure 1.6),
2. un éditeur de texte (figure 1.7),
3. un terminal (figure 1.8),
4. un notebook Jupyter (figure 1.9).

Etapes pour lancer l'exécution de l'application Julia sur JuliaBox :

1. lancer le terminal en cliquant sur le mot `Terminal` dans le menu `New` (voir figure 1.6),
2. entrer la commande `julia`, le REPL (voir 1.2.1) se présente à l'utilisateur (voir figure 1.8).

L'environnement JuliaBox est prêt pour être utilisé.

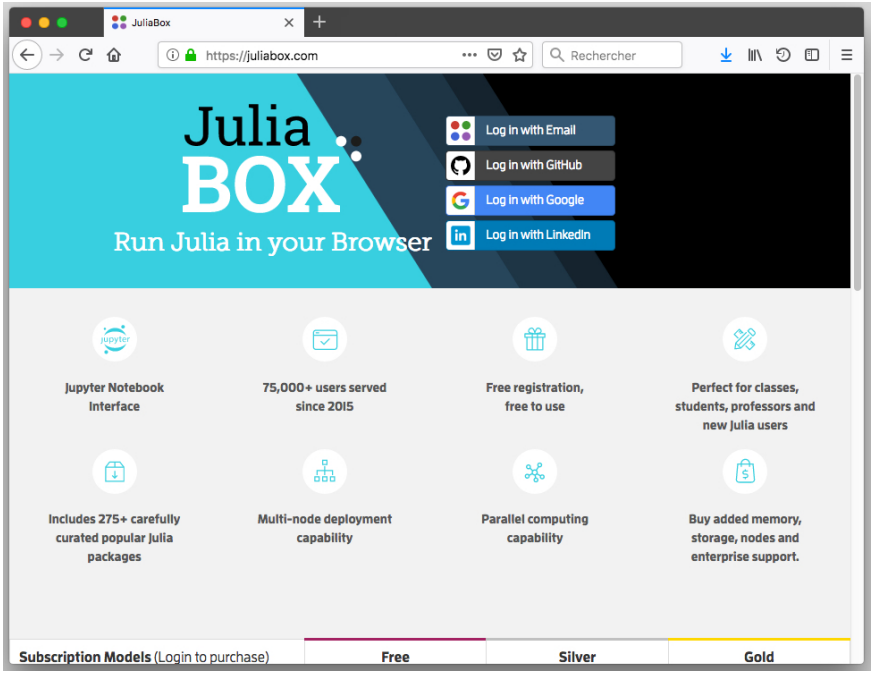

FIGURE 1.5 – Page d'accueil de la JuliaBox.

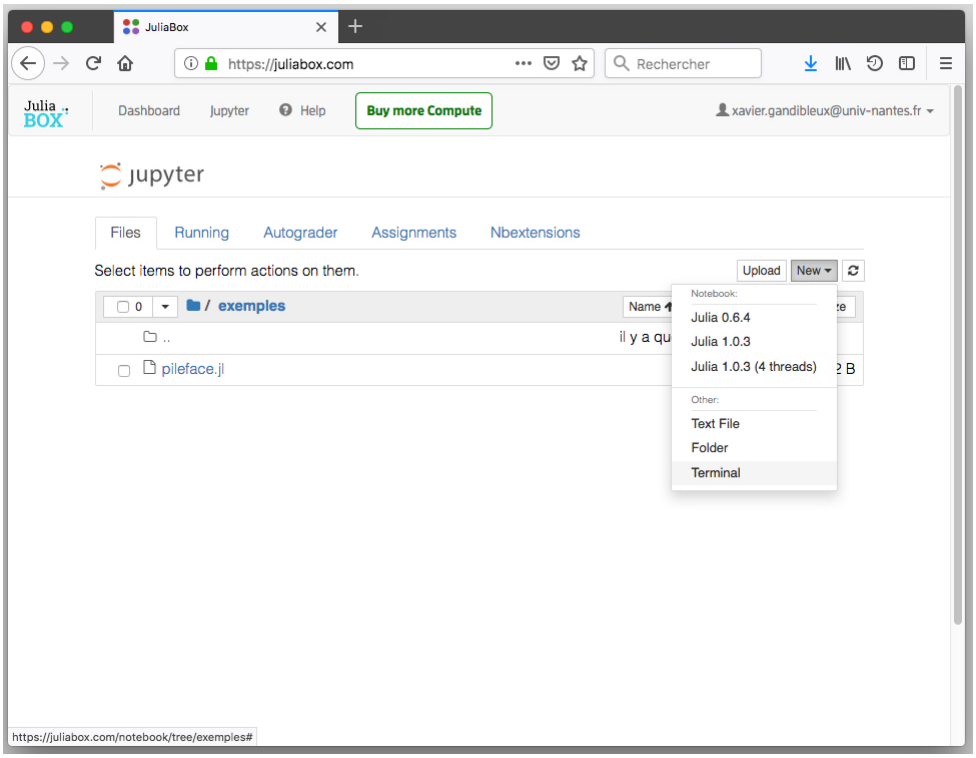

FIGURE 1.6 – Gestionnaire de fichiers sous JuliaBox et menu d'activation des applications.

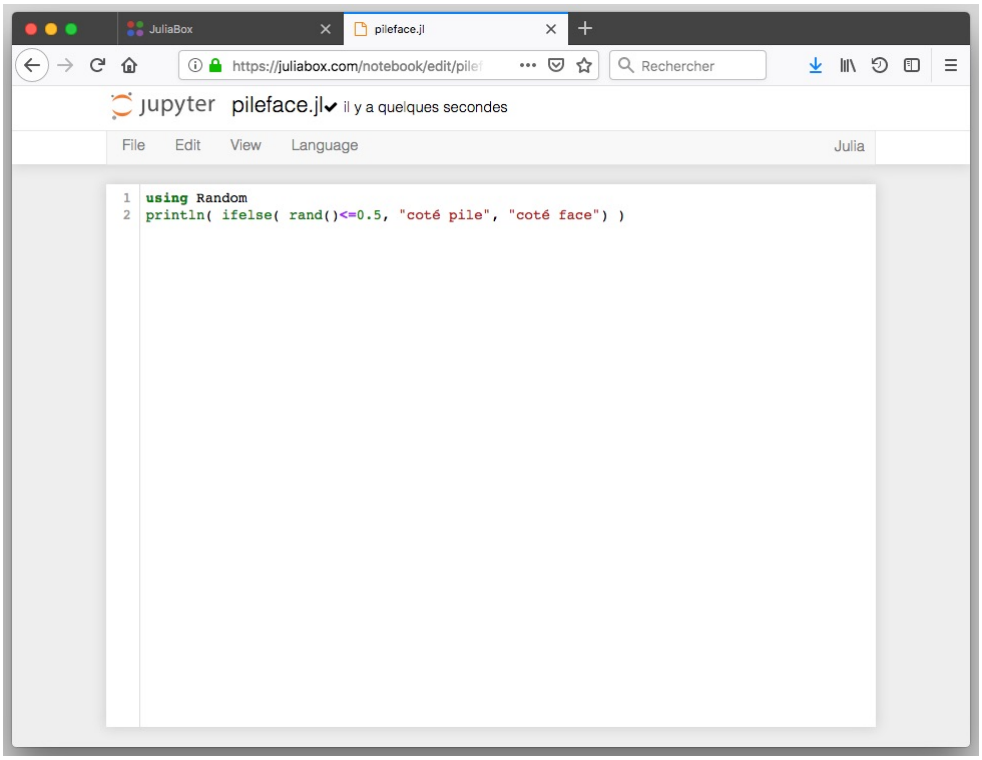

FIGURE 1.7 – Editeur de texte de JuliaBox.

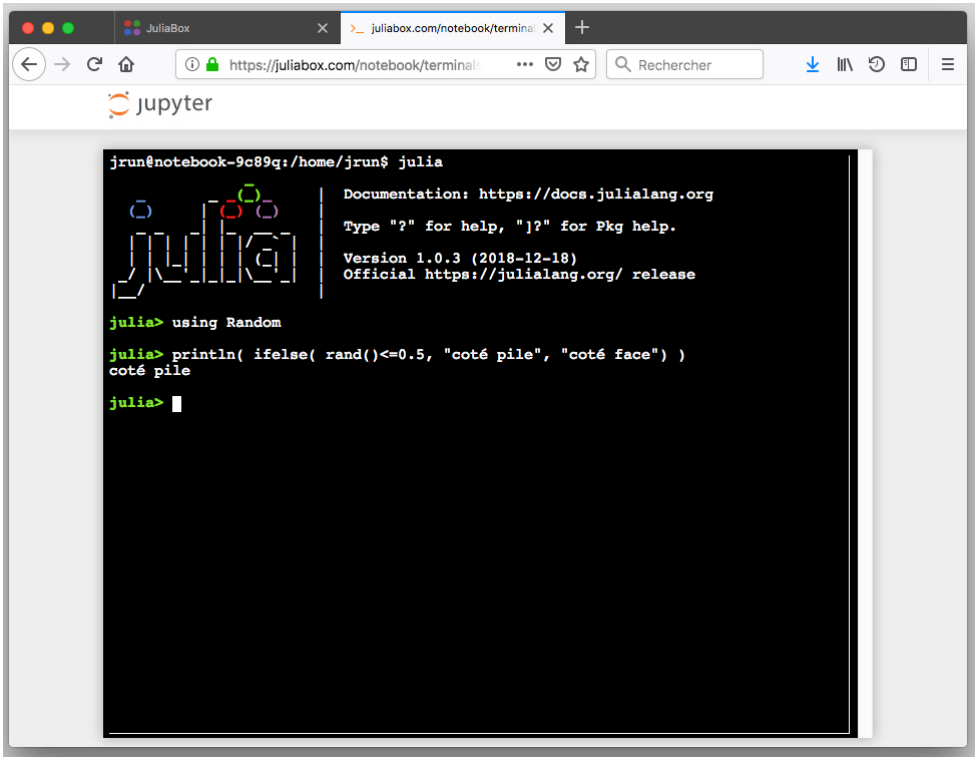

FIGURE 1.8 – Terminal sous JuliaBox dans lequel l'application Julia a été invoquée, REPL prêt.

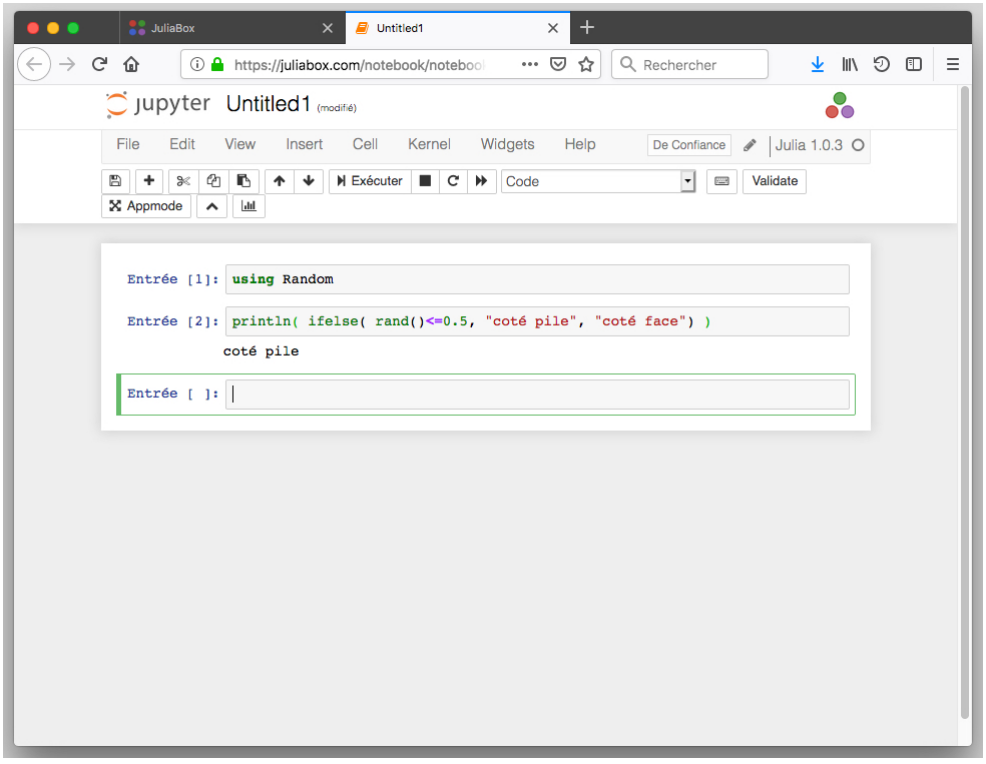

FIGURE 1.9 – Notebook Jupiter sous JuliaBox.

1.1.3 Utiliser un éditeur de texte ou un IDE pour Julia

Un éditeur de texte est un logiciel utilitaire destiné à la création et l'édition de fichiers textes. Il est indispensable pour coder un programme dans un langage de programmation donné. N'importe quel éditeur de texte peut être utilisé pour coder en Julia. Parmi les plus classiques, `Vim`, `Nano` et `emacs` permettent de reconnaitre et colorer les instructions Julia. Cependant, l'éditeur de texte open-source `Atom` (voir `http://atom.io`), étendu par l'ajout du plugin `language-julia` (voir 1.10), apporte un confort d'utilisation remarquable (figure 1.11).

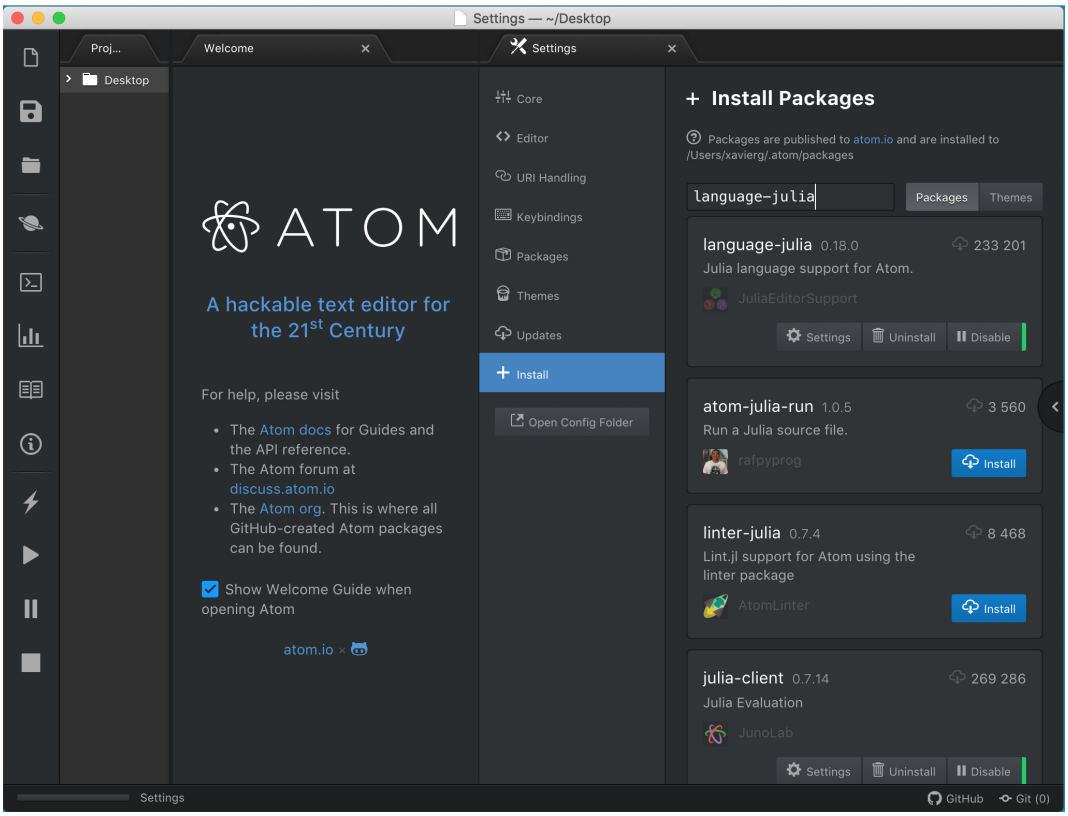

FIGURE 1.10 – Vue principale de l'éditeur de texte Atom, présentant l'espace' "Settings" (espace au centre-droit de la fenêtre) et l'espace "Install Packages" permettant l'ajout de plug-ins (espace à droite de la fenêtre).

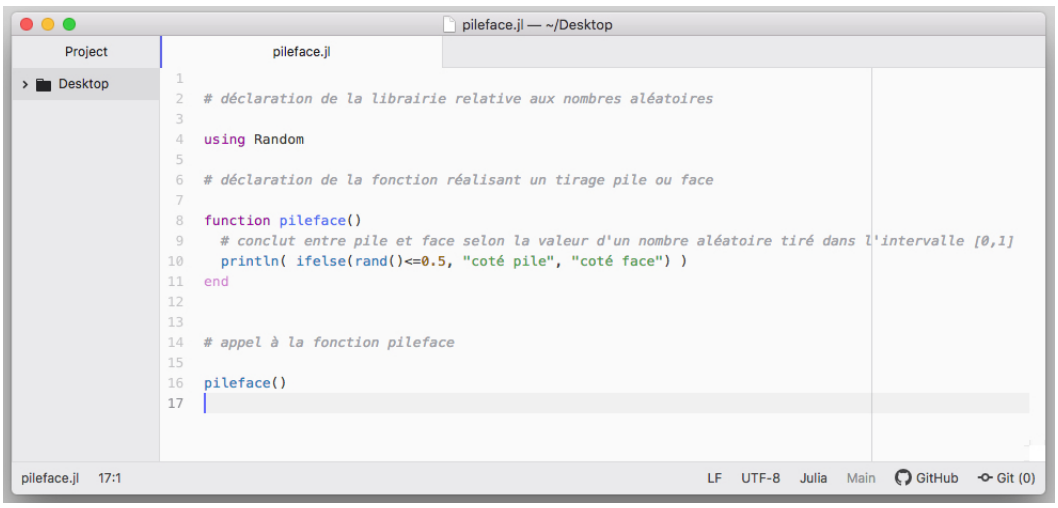

FIGURE 1.11 – Vue principale de l'éditeur de texte Atom.

Etapes de l'installation de `Atom` et du plugin `language-julia` :

1. télécharger l'application `Atom` disponible à l'adresse `https://atom.io/` et suivre les indications d'installation ;
2. étendre les fonctionnalités de `Atom` en ajoutant le plugin `language-julia` comme suit :
   - dans `Atom`, aller dans les préférences et dans l'espace "Settings" ; sélectionner le menu "Install" ;
   - dans la boîte de recherche des extensions, rechercher et installer `language-julia`. `Atom` enchainera automatiquement les étapes jusqu'à l'achèvement de l'installation.

L'installation de `Atom` est terminée.

Un IDE (pour integrated development environment) est un environnement de développement intégré qui comporte, en plus de l'éditeur de texte, différents outils logiciel facilitant le développement et le test de code. `Juno` (voir `http://junolab.org/`) est un IDE destinés au développement sous Julia. Il est bâti sur `Atom` étendu par l'ajout du plugin `uber-juno`. L'environnement de IDE permet de coder en Julia, exécuter dans le REPL ou encore d'afficher graphiquement des résultats (voir figure 1.12).

Etapes de l'installation de `Atom` et du plugin `Juno` :

1. télécharger l'application `Atom` disponible à l'adresse `https://atom.io/` et suivre les indications d'installation ;
2. étendre les fonctionnalités de `Atom` en ajoutant le plugin `uber-juno` comme suit :
   - dans `Atom`, aller dans les préférences et dans l'espace "Settings" ; sélectionner le menu "Install" ;
   - dans la boîte de recherche des extensions, rechercher et installer `uber-juno`. `Atom` enchainera automatiquement les étapes jusqu'à l'achèvement de l'installation.

L'installation de `Juno` est terminée.

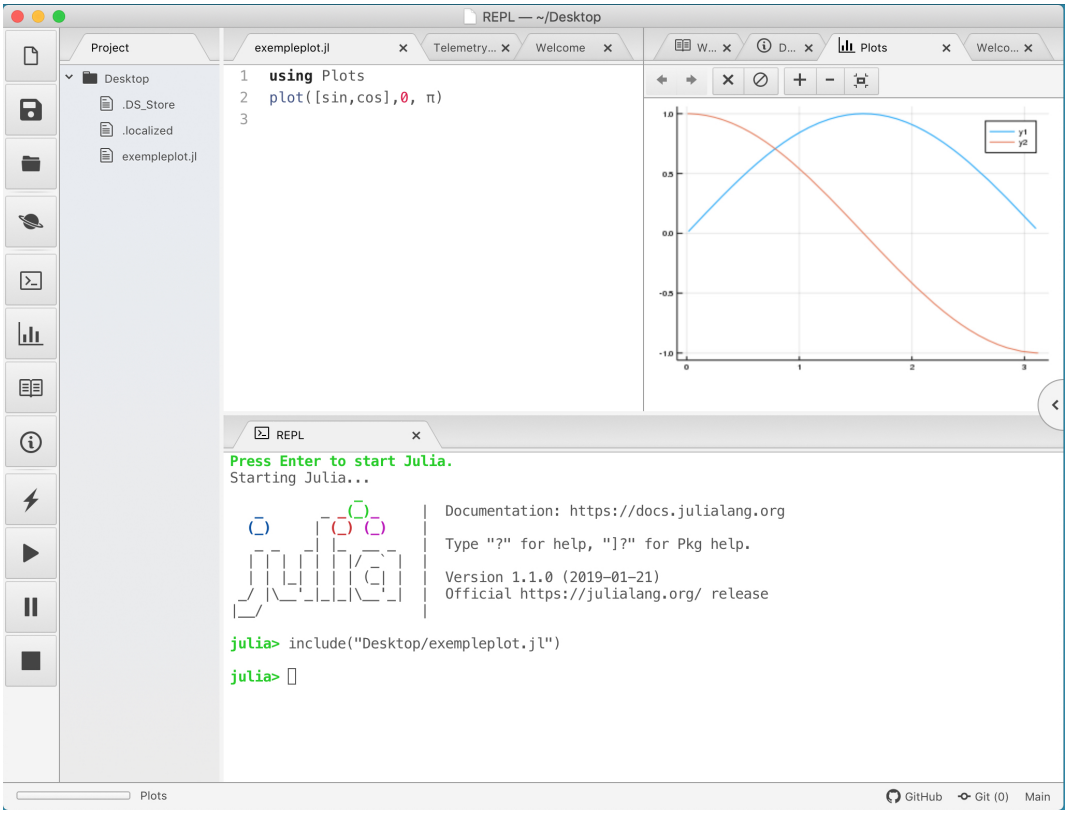

FIGURE 1.12 – Vue principale de l'environnement de développement intégré Juno sous Atom : icônes de raccourcis d'actions (gauche), gestionnaire de fichiers (espace "Project"), éditeur de texte (haut-centre), tracé graphique (haut-droite), REPL (bas).

1.2 Environnement de travail Julia

1.2.1 “REPL” ou mode “Julia”

Le REPL (voir figure 1.4) est l'environnement interactif présenté à l'utilisateur lorsque l'application Julia est invoquée. L'acronyme REPL est le résultat de la concaténation des quatre opérations continument enchainées : *Read* (lire ce qui est entré au clavier), *Evaluate* (évaluer ce qui a été entré), *Print* (présenter le résultat à l'écran), *Loop* (recommencer le cycle). Le REPL permet donc de saisir une expression Julia et d'obtenir directement le résultat de l'évaluation de l'expression. C'est l'environnement de fonctionnement nominal.

A distance ou en local, les interactions avec le langage Julia passent par le REPL. Il permet d'interagir avec le compilateur JIT pour exécuter les codes Julia.

1.2.2 Mode “help”

L'environnement d'aide en ligne (ou mode help) s'active en entrant le caractère point-d'interrogation (`?`) sous le REPL. Le basculement de mode provoque un changement du prompt, lequel prend l'allure suivante :

```
help?>
```

L'utilisateur est invité à présenter l'élément pour lequel il recherche de l'aide. Si il exprime qu'une partie d'une instruction connue par exemple, ou une instruction inconnue, l'environnement proposera des suggestions proches de ce qu'il a été exprimé. Le retour au mode “Julia” est systématique après avoir obtenu la réponse de l'environnement d'aide.

Exemple : consultation de l'aide en ligne pour l'instruction `ifelse` :

```
help?> ifelse
search: ifelse

ifelse(condition::Bool, x, y)

Return x if condition is true, otherwise return y. This differs from ?
or if in that it is an ordinary function, so all the arguments are evaluated
first. In some cases, using ifelse instead of an if statement can eliminate
the branch in generated code and provide higher performance in tight loops.

Examples
=========

julia> ifelse(1 > 2, 1, 2)
2
```

1.2.3 Mode “shell”

L'environnement permettant de passer des commandes du système d'exploitation (ou mode shell) s'active en entrant le caractère point-virgule (`;`) sous le REPL. Le basculement de mode provoque un changement du prompt, lequel prend l'allure suivante :

```
shell>
```

L'utilisateur est invité à présenter une commande du système d'exploitation. Le retour au mode "Julia" est systématique après l'exécution de la commande invoquée.

Exemple : invoque la commande `ls -l` dans le mode shell :

```
shell> ls -l
total 0
-rw-r--r-- 1 xavierg staff 0 7 mar 09:06 code.jl
-rw-r--r-- 1 xavierg staff 0 7 mar 09:07 donnees.dat
-rw-r--r-- 1 xavierg staff 0 7 mar 09:07 rapport.tex
-rw-r--r-- 1 xavierg staff 0 7 mar 09:07 resultats.txt
```

1.2.4 Mode "package"

L'environnement de gestion des packetages (ou mode package) s'active en entrant le caractère crochet-fermant (`]`) sous le REPL. Le basculement de mode provoque un changement du prompt, lequel prend l'allure suivante :

```
(v1.1) pkg>
```

Plusieurs commandes dédiées à la gestion des packetages sont à disposition de l'utilisateur ; parmi celles-ci :

`help`	: aide en ligne présentant les commandes disponibles
`status`	: liste les packetages installés
`add`	: ajout de packetages
`remove`	: retrait de packetages
`update`	: mise à jour des packetages

Le retour au mode "Julia" est obtenu en invoquant un `CTRL-C` ou aussi un `backspace` sur une ligne vide.

Exemple : invoque la commande `status` dans le mode package qui donne la liste les packetages installés :

```
julia>
(v1.1) pkg> status
    Status `~/.julia/environments/v1.1/Project.toml`
  [60bf3e95] GLPK v0.9.1
  [4076af6c] JuMP v0.19.0
(v1.1) pkg> ^C
julia>
```

1.3 Ecrire du code Julia

1.3.1 Eléments d'un code Julia

Le langage de programmation Julia est composé de commentaires (section **??**), de types (section **??**), de variables (section **??**), d'instructions simples (section **??**) et composées (section **??**). Ces éléments permettent d'écrire de nouveaux traitements correspondant à des algorithmes spécifiés et systématisés sous la forme de fonctions (section **??**). Julia dispose aussi d'un très grand nombre de fonctions usuelles (section **??**) prédéfinies. Elles sont regroupées en différents modules (section 1.4) selon leur fonctionnalité, eux même regroupés en packetages.

Un code Julia est généralement constitué :

— de *commentaires*,
— de déclarations d'un ensemble de *modules* à utiliser,
— de définition de *types*,
— de définition de *variables globales* et *constantes*,
— de définition de *fonctions*,
— d'*instructions*.

Le comportement d'une fonction (donné lors de sa définition) est décrit par des définitions de types, de constantes et variables locales, et une séquence d'instructions. Les valeurs des variables peuvent être modifiées par les fonctions et instructions du programme.

1.3.2 Identifiant en Julia

Un identifiant (de variable, de type, de fonction, de packetage, de module) doit toujours avoir été déclaré avant de pouvoir être utilisé. Les règles de Julia pour élaborer un identifiant valide sont les suivantes :

— le premier caractère :
il doit être une lettre (`A-Z` ou `a-z`)

— les caractères suivants :
ils peuvent être une lettre (`A-Z` ou `a-z`), un chiffre (`0-9`) ou un symbole (`! _`)

Remarques :

— Julia est sensible à la casse, c'est à dire qu'il y a une distinction entre la version majuscule et minuscule d'une lettre.

Par exemple, les identifiants `Volt` et `volt` font référence à deux entités distinctes.

— Julia admet l'usage de caractères spécifiques, dont les lettres grecques, comme identifiant de nom de variables par exemple. Ces caractères s'obtiennent en écrivant une commande LaTeX. Dans le REPL comme dans l'éditeur de texte d'Atom, la lettre souhaitée s'obtient en écrivant la commande LaTeX correspondante, immédiatement suivie de la touche de tabulation.

Par exemple, pour obtenir la lettre π il suffit d'écrire la commande `\pi`, suivie de tabulation.

1.3.3 Séparateur d'instructions

Soit par exemple un programme composé des trois instructions simples symbolisées par `<Inst1>` `<Inst2>` `<Inst3>`. Pour que leur écriture soit valide au sein d'un programme, les instructions doivent être distinguées, soit par un retour à la ligne, soit en utilisant le séparateur d'instructions.

Retour à la ligne. Le retour à la ligne agit comme séparateur d'instructions. Deux instructions consécutives sont donc implicitement distinguées lorsqu'elles sont écrites sur des lignes différentes dans le programme :

```
<Inst1>
<Inst2>
<Inst3>
```

En règle générale, il est recommandé d'écrire une seule instruction par ligne de programme. Cette règle reflète une pratique qui a pour effet de favoriser la lisibilité du programme dans son ensemble.

Utilisation du séparateur d'instruction. Quand plusieurs instructions sont écrites sur une même ligne dans le programme, elles doivent être distinguées en utilisant le caractère point-virgule (`;`), qui est le séparateur d'instructions.

```
<Inst1>
<Inst2> ; <Inst3>
```

L'écriture de plus d'une instruction par ligne de programme est tolérée quand la lisibilité du programme n'est pas mise en cause.

1.3.4 Indentation du programme

On dit qu'il y a une indentation entre deux lignes de code consécutives d'un programme lorsque il y a présence d'un décalage positif ou négatif, obtenu par l'ajout ou le retrait de tabulations ou d'espaces, entre le début de la seconde instruction par rapport à la première.

Exemple 1 : l'instruction `<Inst2>` est indentée. L'indentation est de quatre espaces :

```
println("pas indenté")
    println("indenté de 4 espaces")
println("pas indenté")
```

Contrairement notamment à Python, la syntaxe des instructions de Julia n'impose aucune règle d'indentation lors de l'écriture d'un programme. Comme dans tout langage de programmation, l'indentation sera d'usage au niveau de l'écriture d'instructions composées toujours pour une meilleure lecture et compréhension du code.

1.4 Modules et packetages

Julia dispose d'une vaste collection de modules. Un module regroupe un ensemble de fonctions qui implémentent différentes opérations. Les fonctions les plus courantes sont dans le module `base` et nécessitent aucune installation ni déclaration préliminaire avant d'être utilisées. D'autres, comme par exemple les fonctions relatives aux nombres aléatoires ou à l'algèbre linéaire, se trouvent dans des modules fournis a priori avec Julia (en l'occurence, les modules `Random` et `LinearAlgebra`). Ces modules doivent être préalablement déclarés avant d'utiliser les fonctions qu'ils apportent. Enfin, quand il s'agit de modules qui ne sont pas fournis a priori avec Julia, mais apportés par la communauté des contributeurs à Julia, le packetage contenant ces modules doit être préalablement installé avant de procéder à la déclaration et à l'utilisation des dit modules.

1.4.1 Déclaration d'un module

L'opération de déclaration d'un module est à réaliser à chaque nouvelle session de travail, avant d'utiliser une fonction du module en question. L'opération se fait avec l'instruction `using` suivi du nom du module. Plusieurs modules peuvent être déclarés simultanément, à condition de les séparer avec le caractère virgule (`,`).

Exemple 1 : déclaration des modules `LinearAlgebra`, `Random` et `Pkg` :

```
using LinearAlgebra
using Random, Pkg
```

1.4.2 Installation d'un package

Techniquement parlant, un code Julia est organisé en fichiers, modules et packetages, ce dernier pouvant contenir un ou plusieurs modules. Afin d'être facilement distribué dans la communauté scientifique et aisément installable dans une configuration Julia locale, un module est conditionné dans un packetage, lequel est déposé sur la plateforme d'hébergement des packetages pour Julia. La liste des packetages officiels est consultable aux URL suivantes : `https://pkg.julialang.org/` et `https://juliaobserver.com/` .

L'opération d'ajout d'un packetage doit être réalisée une seule fois à l'environnement Julia qui en fera usage, avant la première déclaration d'un module du dit package. Deux manières cohabitent pour ajouter un package : soit passer par l'environnement de gestion des packetages, soit utiliser les fonctionnalités du module `Pkg` qui permettent de manipuler des packetages.

Méthode 1 : ajout du packetage nommé `nomPacketage` via l'environnement de gestion des packetages.

1. sous le mode Julia du REPL, entrer le caractère crochet-fermant (`]`) qui a pour effet de basculer dans le mode "package" ;
2. invoquer la commande d'ajout avec la syntaxe suivante : `add nomPacketage` ;
3. sortir du mode "package" avec la commande `CTRL-C`, qui a pour effet de revenir au mode "Julia" du REPL.

Exemple : les actions suivantes illustrent cette première méthode d'ajout pour le packetage `GLPK` :

```
julia>
(v1.1) pkg> add GLPK
 Resolving package versions...
 Installed LinQuadOptInterface --- v0.6.0
 Installed GLPK ----------------- v0.9.1
  Updating ‘~/.julia/environments/v1.1/Project.toml‘
  [60bf3e95] + GLPK v0.9.1
  Updating ‘~/.julia/environments/v1.1/Manifest.toml‘
  [60bf3e95] + GLPK v0.9.1
  [f8899e07] + LinQuadOptInterface v0.6.0
  Building GLPK -> ‘~/.julia/packages/GLPK/rw631/deps/build.log‘
(v1.1) pkg> ^C
julia>
```

Méthode 2 : ajout du packetage nommé `nomPacketage` les fonctionnalités du module `Pkg`

1. déclarer l'utilisation à venir du module `Pkg` en invoquant `using Pkg`
2. invoquer la commande d'ajout avec la syntaxe suivante, `Pkg.add("nomPacketage")`

Exemple : les actions suivantes illustrent cette seconde méthode d'ajout pour le packetage `GLPK` :

```
julia> using Pkg
julia> Pkg.add("GLPK")
 Resolving package versions...
  Updating ‘~/.julia/environments/v1.1/Project.toml‘
  [60bf3e95] + GLPK v0.9.1
  Updating ‘~/.julia/environments/v1.1/Manifest.toml‘
  [60bf3e95] + GLPK v0.9.1
  [f8899e07] + LinQuadOptInterface v0.6.0
julia>
```

Remarque :

> L'installation d'un packetage peut conduire à devoir installer d'autres packetages du fait d'éventuelles relations de dépendances présentes entre eux. L'installation des packetages dépendants est automatique et est assurée par le gestionnaire de packetage. Par conséquent, il n'est pas rare que l'ajout d'un packetage prenne plusieurs minutes (temps qui est dépendant de différents facteurs tels que la vitesse du réseau auquel est connecté l'ordinateur, du nombre de packetages dépendants à installer, du volume des packetages installés devant être compilés en local).

1.5 Exécuter un code Julia

1.5.1 Exécuter dans le REPL du code à la volée

Dans cette première manière d'exécuter du code, les instructions et données du programmes sont directement écrites à la volée dans le REPL. L'ensemble des éléments déclarés (modules) et définis (variables globales et constantes, fonctions) sont posés et les instructions sont immédiatement exécutées. Pour exemple, les lignes de code suivantes illustrent un traitement qui réalise un tirage pile ou face suivant une variable aléatoire :

```
julia> using Random
julia> println( ifelse( rand()<=0.5, "coté pile", "coté face") )
coté pile
```

A l'issue de l'écriture de chaque ligne, aussitôt qu'une instruction est identifiée complète et valide par le REPL, elle est traitée. L'effet de l'instruction apparaît immédiatement dans le terminal.

Cette manière de procéder s'avère confortable pour exécuter un traitement très court, pour tester une commande, mettre au point interactivement un traitement simple. Cependant, il est attendu dans la pratique de Julia que les traitements fassent systématiquement l'objet de fonctions. Par conséquent, il est recommandé d'éviter d'écrire une séquence d'instructions en dehors de fonctions.

1.5.2 Exécuter dans le REPL un code enregistré dans un fichier source

Dans cette seconde manière d'exécuter du code, les instructions du programmes sont enregistrées dans un fichier qui sera à charger dans le REPL. Par convention, un fichier contenant du code source Julia utilise `.jl` comme extension de fichier. Le code source suivant est une version du programme réalisant un tirage pile ou face dans laquelle une fonction `pileface` est définie.

Code 1: code source du programme `pileface.jl` présentant une fonction.

```
1 # déclaration du module relatif aux nombres aléatoires
2 using Random
3
4 # déclaration de la fonction pileface
5 function pileface()
6   # conclut entre pile et face selon la valeur d'un nombre aléatoire tiré dans l'intervalle [0,1]
7   println( ifelse( rand()<=0.5, "coté pile", "coté face") )
8 end
```

Le chargement d'un programme dans le REPL s'effectue en invoquant la commande `include` avec le nom du fichier à charger. En supposant que le fichier précédent soit sauvegardé sous le nom `pileface.jl`, la commande s'écrit :

```
julia> include("pileface.jl")
pileface (generic function with 1 method)
```

A l'issue du chargement du programme, l'ensemble des éléments déclarés (modules) et définis (variables globales et constantes, fonctions) sont posés et les instructions sont exécutées. Celles-ci présentent généralement des appels vers les fonctions définies du programme. La fonction `pileface` peut maintenant être invoquée dans le REPL et donne le résultat suivant :

```
julia> pileface()
coté pile
```

1.5.3 Comprendre comment s'exécute un code Julia

Un programme écrit en Julia sera « compilé juste à temps (JIT) », c'est à dire qu'une instruction est compilée quand elle est demandée pour être exécutée et son code machine est produit à ce moment. Ce principe combine l'avantage de l'efficacité d'un langage compilé classique comme C ou C++ (ou le code machine est produit a priori en résultat de la phase de compilation) avec l'avantage de la flexibilité et l'interactivité d'un langage interprété comme Python (ou le code machine est produit à la volée lors de l'exécution d'une instruction).

Corollaire à ce principe, la première exécution d'un programme apparaîtra « lente » du fait qu'elle intègre le temps inhérent à la création du code machine, contrairement aux exécutions suivantes du même programme.

Aussi, quand le compilateur est suffisamment renseigné sur le code machine à produire, il génèrera un code machine efficace. Le typage joue un rôle important à ce niveau. Typer correctement un code contribue à renseigner le compilateur, lui dispensant de devoir inférer les types non précisés. Cela lui évitera de produire un code machine surdimensionné pour couvrir les informations non précisées qui lui manque lors de l'exécution.