A

Introduction à Octave (version libre de Matlab™)

CONTENTS

A.1 Introduction

A.1.1 Pourquoi apprendre un deuxième langage ?

Il existe plusieurs centaines de langages de programmation... Comme nous l'avons rappelé au premier chapitre, il est important pour un programmeur d'en connaître plusieurs afin d'avoir une approche « polyglotte » de l'informatique, et ceci pour plusieurs raisons :

1. il n'existe pas de langage universel,
2. les langages — et parfois les modes — évoluent avec le temps,
3. certains langages sont plus adaptés (ou optimisés) que d'autres à traiter d'un problème,
4. il est sain de savoir séparer le fond et la forme d'un code.

Ainsi au cours de votre carrière vous serez inévitablement confrontés à plusieurs langages informatiques : soit parce qu'il sera imposé par votre employeur ou un projet collectif, soit parce que vous devrez comprendre et peut-être modifier ou traduire des codes existants développés par d'autres. En complément de *Python* qui est un langage interprété et lent, l'idéal serait d'apprendre un langage compilé et rapide comme le *C*. Mais ici nous allons simplement vous ouvrir un peu l'esprit avec un second langage interprété, sans doute l'un des plus proches de celui que vous avez déjà abordé, en tout cas pour tout ce qui concerne le calcul numérique et le graphisme. Donc rassurez-vous il n'y aura donc pas (trop) de surprises.

A.1.2 *Octave* versus *Matlab*™

GNU Octave — que nous appellerons plus simplement *Octave* — a été conçu vers 1988 (soit 4 ans après la première version commerciale de *Matlab*™), mais son développement n'a réellement commencé qu'en 1992. Cependant, si la partie calcul numérique — et en particulier le calcul matriciel — a été fonctionnel dès le début puisqu'elle est au cœur du langage, il faudra attendre près de 20 ans (en 2013 avec la version 4) pour profiter des fonctions graphiques évoluées et notamment avoir une interface type bureau, fonctionnelle à peu près équivalente à celle de son jumeau commercial. Depuis, le projet *Octave* est très actif, multi-plateforme (*Linux*, *MS Windows* et *MacOS*), et il doit aujourd'hui être considéré comme une alternative sérieuse à la version payante.

Octave est donc un logiciel libre qui se veut un clone de *Matlab*™, en ce sens que toute fonction ou syntaxe *Matlab*™ incompatible est considérée comme une fonctionnalité manquante sous *Octave* (ou un bogue à corriger). Pour simplifier, on parle donc du **même langage** et non d'un langage « similaire » ou « équivalent », et l'on peut considérer que tout code *Matlab*™ est sensé tourner sous *Octave*. En revanche,

l'inverse n'est pas forcément vrai car *Octave* permet quelques libertés syntaxiques qui rendent ses codes potentiellement incompatibles avec *Matlab*™, sauf moyennant quelques adaptations.

Dans ce chapitre d'introduction, nous nous limiterons volontairement à du code strictement compatible entre les deux logiciels afin que vous puissiez passer de l'un à l'autre sans difficulté. Mais vous trouverez ci-dessous le détail de ces différences ; certaines sont importantes car elles rapprochent un peu plus *Octave* et *Python* !

Les ajouts syntaxiques dans *Octave*

Voici pour mémoire les ajouts de syntaxe *Octave* qui n'existent pas dans *Matlab*™ :

1. les lignes commentées peuvent être préfixées avec le caractère `#` (en plus du caractère `%`) ;
2. les opérateurs typiques du langage *C* sont supportés : `++`, `--`, `+=`, `*=`, `/=` ;
3. l'indexation d'une expression est possible sans besoin de créer une nouvelle variable, ex : `[1:10](3)` ou `a'(:)` ;
4. l'opérateur logique de négation peut s'écrire avec le point d'exclamation `!` (comme pour la plupart des langages), aussi bien qu'avec le tilde `~` (spécificité de *Matlab*™/*Octave*) ;
5. les chaînes de caractères peuvent être définies par des guillemets `"` (double-quote) aussi bien que des apostrophes `'` (single-quote) ;
6. les blocs logiques peuvent être terminés par un mot-clé spécifique, pour plus de clarté, comme `endif`, `endfor`, `endwhile`, etc. ;
7. les fonctions peuvent être définies à l'intérieur d'un script ou sur la ligne de commande ;
8. il existe la boucle `do-until` (similaire à la boucle C do-while).

Cependant, il ne faudrait pas opposer les logiciels gratuits et payants — *Octave* et *Matlab*™ — de façon trop manichéenne. Le monde du logiciel libre a d'immenses qualités mais aussi quelques défauts dont il faut avoir conscience. Voici en résumé les avantages et inconvénients à considérer objectivement lorsque vous comparer les deux produits :

	Matlab™	*Octave*
gratuité	✗	✓
code libre et ouvert	✗	✓
Jupyter notebook	✗	✓
exempt de bogue	✓	✗
compilateur	✓	✗
maintenance et hotline	✓	✗
documentation complète	✓	✗
version stables	2/an	≈ 1/an
rapidité d'exécution	3 à 10 ×	1 ×

Ainsi, la version libre du logiciel conviendra sans doute à un laboratoire de recherche ou à un enseignant et ses étudiants. Mais une entreprise commerciale ou avec des obligations opérationnelles se tournera plus volontiers vers une licence et une maintenance payante afin de 1) minimiser les dysfonctionnements et 2) avoir des codes plus rapides. C'est d'ailleurs pour cette même raison que de nombreuses entreprises continuent à utiliser des systèmes d'exploitation payants (type *MS Windows*) et résistent encore à la transition vers les systèmes libres, même si fort heureusement ces derniers tendent à être de plus en plus fiables et bien documentés.

En résumé si votre entreprise, votre université ou votre laboratoire dispose d'une licence *Matlab*™ et que vous souhaitez coder avec ce langage, il ne faut pas hésiter à en profiter ! Vous ferez des programmes

robustes et rapides, tout en permettant de partager un code ouvert avec le monde du libre grâce à *Octave*.

A.1.3 Un mot sur les bisbilles entre "matlabeurs" et "pythoneurs"...

Dans les années 1990 et 2000, *Matlab*™ est devenu extrêmement populaire et était enseigné dans la plupart des écoles et universités, parfois comme simple initiation à la programmation et, trop souvent sans doute, le considérant comme un langage à tout faire et ignorant ses spécificités et performances en calcul matriciel. Vers les années 2010, la communauté scientifique a ainsi connu le début d'une vague importante de "migration" d'utilisateurs visiblement insatisfaits qui se sont tournés vers *Python*. Vous trouverez encore de nombreux sites web expliquant de façon plus ou moins prosélytiste qu'il faut absolument vous convertir ! Les principaux arguments sont essentiellement ceux liés aux avantages du logiciel libre, mais on trouve aussi des commentaires plus subjectifs sur une meilleure lisibilité ou élégance du code, la supériorité des fonctions, le dynamisme de la communauté des utilisateurs... voire des réactions de rejet totalement disproportionnées tenant plus d'une forme de communautarisme. Ces sites n'évoquent que rarement la solution libre *Octave* véritablement dédiée au calcul numérique, et qui, si elle n'avait été délaissée à l'époque, aurait évité à des milliers de programmeurs de procéder à la fastidieuse traduction de leurs lignes de codes, pour un gain d'efficacité souvent insignifiant.

Dans ce cours nous voulons absolument dépasser ces petites guerres stériles et vous faire comprendre qu'il n'y a aucune raison de dénigrer un langage de programmation au profit d'un autre, pas plus qu'on ne le ferait pour une langue. Après les avoir abordés, vous utiliserez le langage qui convient le mieux à votre projet ou à vos affinités, et devrez rester ouvert aux autres.

A.1.4 Autres alternatives libres à *Matlab*™

Enfin pour être quasi exhaustif, il faut signaler qu'aux côtés de *GNU Octave* il existe d'autres alternatives libres et gratuites à *Matlab*™ :

1. *SciLab*, une initiative française des chercheurs de l'Inria : conçu dès 1982 à partir des premiers développements de *Matlab*™, la première version libre date de 1990 grâce à une collaboration avec l'école des Ponts et Chaussées (aujourd'hui Ponts ParisTech). Le langage est très similaire à *Matlab*, mais la compatibilité des codes n'est pas assurée (ni recherchée).
2. *FreeMat*, un logiciel sous licence libre très similaire à *Matlab*™ et *Octave*, développé par un petit groupe de bénévoles en 2004, mais dont le développement n'a pas été poursuivi depuis la dernière version de 2013.
3. et bien entendu *Python* avec ses librairies *Numpy* et *Matplotlib* !

A.1.5 Installation d'*Octave*

Octave peut être installé sur les trois grands types de plateforme que sont *Linux*, *MS Windows* et *Mac OS*. Toutes les informations d'installation sont sur le site octave.org. Une fois installé sur votre système vous aurez accès à l'interface graphique (le bureau) qui est l'équivalent du *Spyder* de *Python* avec une fenêtre de commandes, un éditeur de fonction, un accès visuel aux variables de votre espace de travail, un navigateur de fichiers, un accès à la documentation et un historique de vos commandes. Vous pourrez aussi lancer *Octave* dans un terminal sans cette interface.

Octave est compatible avec *Jupyter Notebook*. Pour l'utiliser dans ce cadre, il faut installer le noyau *Octave* via *Anaconda*, avec la commande suivante :

```
conda install -c conda-forge octave_kernel
```

Lors de la création d'un nouveau Notebook, vous aurez alors le choix entre **Python** et **Octave** !

A.2 *Octave* versus *Python/Numpy*

A.2.1 Les points communs et similarités

Comme nous l'avons évoqué à plusieurs reprises, les similarités entre les deux langages sont nombreuses et nous les verrons tout au long de ce chapitre. Voici les deux principales caractéristiques communes :

1. *Python* et *Octave* sont des langages sous licences libres, gratuits et dont les codes sources sont ouverts au public (*open source*). Bien qu'ayant des historiques très différents, on peut dire qu'ils s'inscrivent tous deux dans le « mouvement du logiciel libre » qui vise à favoriser la solidarité et la coopération entre les personnes qui utilisent des ordinateurs.
2. *Python* et *Octave* sont tous deux des langages très évolués permettant d'écrire des programmes structurés comportant des fonctions, la programmation orientée objet[1], la gestion de flux d'entrée/sortie, l'interaction avec des programmes externes compilés, et une interface graphique très performante (2D/3D[2], traitement d'images, animations) permettant de produire des figures de haute qualité pour des publications scientifiques par exemple.
3. Comme *Python*, *Octave* est un langage semi-interprété : il nécessite un environnement permettant d'interpréter le code puis l'exécuter sur la machine. Cette catégorie de langage offre la plus grande souplesse de programmation pour créer et tester rapidement un code. En revanche, les programmes sont beaucoup plus lents à l'exécution que leur équivalent écrit en langage compilé (comme en *Fortran* ou *C*).

De très nombreuses fonctions ont leur équivalent dans les deux langages. Il arrive qu'elles portent exactement le même nom, mais peuvent aussi différer dans leurs arguments ou leur dénomination. Il existe des dictionnaires permettant de passer d'un langage à l'autre. Le projet *Numpy* propose ainsi une page destinée aux utilisateurs du langage *Matlab*™ : . Dans ce cours, nous proposons un dictionnaire inverse donnant les équivalences entre les fonctions *Python* et *Matlab*™ : .

Avant d'aborder les différences, ci-dessous deux exemples de code strictement équivalents à ceux en *Python* vus aux chapitres 3 et 4. La structure des programmes, la syntaxe et les noms de variables ont été préservés pour insister sur la similarité des deux langages. Notez pour la figure que le rendu graphique est différent en raison des paramètres par défaut (taille des polices de caractère, remplissage des marqueurs, épaisseur des lignes, etc...). En jouant sur les paramètres de l'un ou l'autre des langages, on peut bien évidemment obtenir un résultat strictement similaire.

1. Il faut préciser ici que *Python* est fondamentalement un langage « orienté objet » : une liste, un module, une variable, une fonction, ... tout est un objet. Avec *Octave* la notion d'objet se limite aux « classes » (définir un objet associé à des propriétés, des méthodes, des événements et un dictionnaire) et aux graphismes.

2. Notons qu'en 2022, la librairie *Matplotlib* de *Python* gère en réalité très mal le 3D. Le toolkit *mplot3D* est encore considéré par ses propres développeurs comme immature. Ce n'est pas le cas avec *Octave* qui suit de près les fonctions de représentation 3D très évoluées de *Matlab*™.

GNU Octave

```
Tab1 = [3,16,12;
        8,9,14;
        6,16,13;
       15,14,16;
       13,15,11];

% Calcul vectorisé:
NotSem1 = Tab1(:,1);
disp(NotSem1')

% Calcul des notes du semestre 2 pour tous les étudiants
NotSem2 = Tab1(:,2) * 0.8  + Tab1(:,3) *0.2;
disp(NotSem2')

% Calcul de la moyenne finale
NotFinale = (NotSem1 + NotSem2) / 2;
disp(NotFinale')
```

```
    3    8    6   15   13
   15.200   10.000   15.400   14.400   14.200
    9.1000    9.0000   10.7000   14.7000   13.6000
```

GNU Octave

```
% Données
%------------------------
MoyGen_Claire = [11,13,9,14,16];
MoyGen_Damien = [9,10,9,11,9];

% Vecteur contenant le numéro des années d'étude
Annees = 1:5;

% Bloc graphique (X,Y)
%------------------------
figure

plot(Annees, MoyGen_Claire,'Color','r','LineStyle','-','Marker','o')
hold on
plot(Annees, MoyGen_Damien,'Color','b','LineStyle','-','Marker','o')
hold off
xlabel('Années étude')
ylabel('Note sur 20')
title('Moyennes Générales')
legend({'Claire','Damien'},'Location','Northwest')

grid on
```

Figure A.1

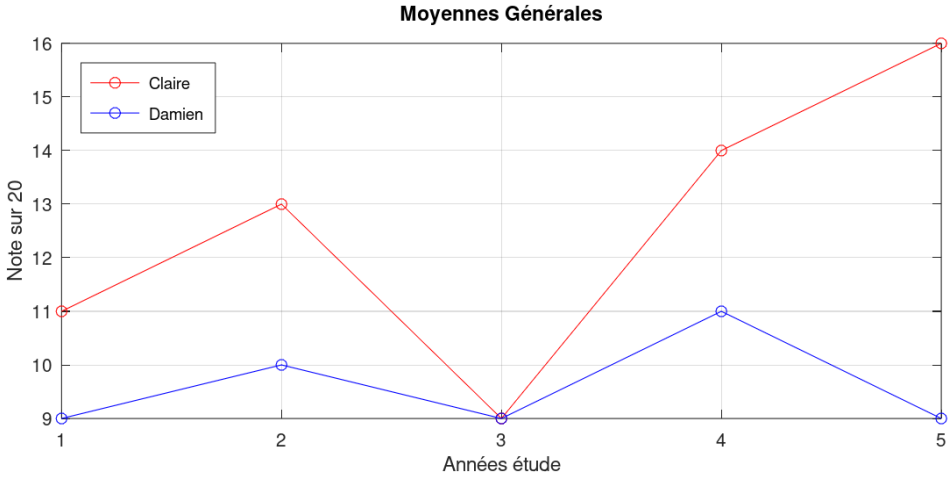

FIGURE. A.1. *Deux courbes sur une figure (version Octave de la figure 4.2)*

A.2.2 Les différences

A.2.2.1 Objectifs initiaux et évolution

Octave est basé sur *Matlab*™ dont le nom est l'acronyme de ***Mat**rix **Lab**oratory*. C'est donc dès l'origine un logiciel centré sur le *calcul matriciel* (et non les mathématiques comme certains le pensent !) et donc particulièrement adapté au traitement de données numériques. Ce sont d'ailleurs les premières applications de traitement de signal qui l'ont fait préférer à la programmation en *Fortran* (puis en *C*) dans les années 80. Ainsi, la syntaxe d'un calcul matriciel et d'algèbre linéaire en *Octave* sera particulièrement compacte et simple (voir ci-dessous).

L'une des grandes forces d'*Octave* est d'avoir préservé cette syntaxe originale malgré les évolutions de fonctionnalités depuis maintenant plus de trois décennies. Les évolutions de langages se font toujours avec une compatibilité descendante, c'est-à-dire préservant les fonctions plus anciennes et bien évidemment la syntaxe. Ainsi, à l'instar de ses grands frères *C* et *Fortran*, un code de calcul matriciel écrit dans les années 80 tournera toujours avec une version d'aujourd'hui. En outre, si l'abandon d'anciennes fonctions existe, il ne concerne jamais des fonctions fondamentales et leur suppression est annoncée plusieurs années à l'avance par un message d'alerte à chaque utilisation. *Octave* est ainsi un langage d'une forte stabilité syntaxique, garante de sa longévité.

Python en revanche est un langage beaucoup plus mobile et évolutif : ses développeurs n'hésitent pas à changer une syntaxe, réformer des fonctions ou renommer des modules d'une version majeure à l'autre, dans le but d'améliorer le langage. Par exemple entre *Python* 2 (de 2000 à 2010) et *Python* 3 (depuis 2008), l'instruction de base `print` a été transformée en fonction (voir exemples ci-dessous), ce qui engendre de multiples erreurs si vous exécutez un code écrit en *Python* 2 sous *Python* 3. *Python* 3 est ainsi une version du langage volontairement réformatrice, dans un but de simplification du code, et dont l'incompatibilité descendante est totalement assumée. Le langage *Python* 2 a de plus été officiellement abandonné en 2020, ce qui lui confère une durée de vie d'une décennie seulement.

Python 2

```
# print est une instruction
>>> print "Bonjour","à","tous"
Bonjour à tous

# cette syntaxe affiche un tuple
>>> print("Bonjour","à","tous")
('Bonjour', '\xc3\xa0', 'tous')

# division entière
>>> 7/2
3
```

Python 3

```
>>> print "Bonjour","à","tous"
  File "<stdin>", line 1
    print "Bonjour","à","tous"
          ^
SyntaxError: Missing parentheses in call ...
>>> print("Bonjour","à","tous")
Bonjour à tous

# division flottante par défaut
>>> 7/2
3.5
```

A.2.2.2 Gestion et calcul sur les matrices

Sous *Octave*, les matrices à N dimensions sont le type par défaut de toute variable numérique, éventuellement complexe, codée en flottant double précision (64 bits)[3]. Ainsi un scalaire, un vecteur ligne ou colonne, une matrice à 2 dimensions ou une grille à N dimensions seront tous un même objet traité de la même façon numériquement. Ce qui les différenciera sera uniquement si l'une ou l'autre des dimensions est égale (ou plus exactement, a une longueur égale) à 1 :

- un nombre **scalaire** sera considéré comme une "matrice" 2D à une seul élément, de taille 1×1 (les dimensions supérieures ayant toutes une longueur de 1),
- un **vecteur colonne** sera considéré comme une matrice dont seule la première dimension a une longueur supérieure à 1 ($m \times 1$),
- un **vecteur ligne** sera une matrice dont la première dimension a une longueur de 1 et la seconde est supérieure à 1 ($1 \times n$),
- une **matrice** au sens mathématique aura ses 2 premières dimensions de longueur supérieure à 1 ($m \times n$),
- une **grille à 3 dimensions** sera une matrice dont les 3 premières dimensions ont une longueur supérieure à 1 ($m \times n \times p$)

Si un vecteur (ligne ou colonne) est en réalité une matrice 2D et non un objet de à une seule dimension comme avec *Numpy*, la plupart des instructions de calcul sur les matrices vont tout de même détecter que c'est un vecteur... nous verrons cela en détail plus loin.

Avec *Python*, ce qui se rapproche le plus du type de variable par défaut d'*Octave* est un `array` de la bibliothèque *Numpy*.

A.2.2.3 Indexation des vecteurs/matrices

La façon d'indexer les matrices et leur manipulation est l'une des différences fondamentales entre *Octave* et *Python*.

L'aspect le plus flagrant est qu'avec *Octave*, le premier indice d'un tableau est `1`, alors que *Python* utilise une indexation basée sur le zéro, de sorte que l'élément initial d'une séquence a l'index `0`. La confusion possible et les petites guerres entre les deux approches (le monde des langages informatiques se divise en deux catégories !) viennent du fait que chaque méthode a des avantages et des inconvénients :

— l'indexation basée sur 1 est conforme à l'usage courant "humain", où il semble logique que le premier élément d'une séquence ait l'index 1 ;

— l'indexation basée sur le zéro simplifie les calculs car elle est plus proche du langage "machine" (par

3. Si les premières versions du langage répondaient strictement et exclusivement à cette définition, il est bien sûr possible de déclarer et manipuler des variables de classes différentes (voir plus loin). Cependant, la variable de base par défaut et les calculs se font encore et toujours par défaut sur une matrice de double précision.

exemple dans le processeur, un nombre entier codé sur 8 bits peut prendre 256 valeurs qui vont de 0 à 255).

Une autre particularité d'*Octave* est l'indexation linéaire des matrices, une méthode n'existant pas en *Python*. Les éléments d'une matrice, quelles que soient ses dimensions, sont adressées par un index unique qui les numérote dans l'ordre des dimensions, soit la première colonne d'abord (direction de la dimension 1), puis les dimensions supérieures. Cette indexation permet d'adresser un groupe d'éléments à partir d'un seul vecteur d'index, ce qui est plus proche de l'indexation physique de la mémoire dans l'ordinateur — et donc plus efficace —, plutôt que de devoir recourir systématiquement à un index par dimension, soit un index pour la colonne et un index pour la ligne. Cette dernière solution reste cependant autorisée et peut donc être utilisée si elle est plus adaptée aux besoins du programmeur.

Enfin, l'adressage des tableaux (vecteurs ou matrices) est également différent sur deux points :

1. **Les intervalles d'indices.** Avec *Python*, la limite supérieure d'un intervalle est exclue, ce qui fait qu'un intervalle de l'indice i à l'indice j s'exprime par $i : (j+1)$. En *Octave*, la limite supérieure est inclue, avec $i : j$.
2. **L'adressage d'une seule dimension.** Avec *Python*, il est possible d'adresser la première dimension d'un tableau en omettant la deuxième. Avec *Octave*, une syntaxe équivalente sera interprétée comme une indexation linéaire ; il faut donc impérativement spécifier que l'on souhaite tous les éléments de la deuxième dimension.

Exemples avec une matrice 2×4 :

Python

```
>>> x = np.array([[1,2,3,4],[5,6,7,8]])
>>> print(x)
[[1 2 3 4]
 [5 6 7 8]]
>>> print(x[1,0:3])
[5 6 7]
>>> print(x[0]) # première ligne
[1 2 3 4]
>>> print(x[:,1]) # deuxième colonne
[2 6]

>>> print(x[4:6])
[]
```

GNU Octave

```
> x = [1,2,3,4;5,6,7,8];
> disp(x)
   1   2   3   4
   5   6   7   8
> disp(x(2,1:3))
   5   6   7
> disp(x(1,:)) % première ligne
   1   2   3   4
> disp(x(:,2)) % deuxième colonne
   2
   6
> disp(x(5:6)) % indexation linéaire
   3   7
```

Notez une autre différence notable sur les deux exemples de ligne et colonne : avec cette syntaxe en *Python* c'est un vecteur qui est renvoyé (un tableau à une seule dimension) et non une matrice. Avec *Octave*, la notion de vecteur n'existe pas en tant que telle : il s'agit toujours d'une matrice à au moins 2 dimensions et dont seule une dimension a une longueur supérieure à 1 ; ainsi, l'extraction d'une ligne renvoie un *vecteur ligne* (une matrice $1 \times n$), et l'extraction d'une colonne renvoie un *vecteur colonne* (une matrice $m \times 1$). Pour obtenir le strict équivalent en *Python* il faudrait écrire `x[0:1,:]` pour le vecteur ligne, et `x[:,1:2]` pour le vecteur colonne.

A.2.2.4 Séquences

Malgré leur syntaxe très ressemblante, la notion de séquence est notablement différente entre *Python* et *Octave*.

En terme de syntaxe pure, *Python* utilise une écriture particulièrement compacte et pratique. Pour mémoire, une séquence de a à b par incrément de c sera notée d'une façon générale `a:b:c` et il faut juste se

rappeler que la limite supérieure *b* est toujours exclue de la séquence. Les raccourcis permettant d'omettre une partie ou la totalité des paramètres sont bien pratiques (lorsque *c* vaut 1, *a* vaut 0, et/ou *b* vaut le dernier élément +1). Ainsi, la séquence de nombres entiers de 1 à 5 s'écrira simplement `1:6` ou `range(1,6)`, mais elle ne peut être utilisée qu'à l'intérieur d'indices d'un tableau (*slice index*) ou dans une boucle `for`. Pour créer un vecteur contenant cette séquence de nombres et utilisable en tant que matrice pour des calculs, il faut utiliser la fonction `np.arange(1,6)` ou `np.r_[1:6]` qui renvoient toutes deux un objet de type `array`.

Avec *Octave*, une séquence de *a* à *b* par incrément de *c* s'écrira `a:c:b` et *b* fera partie de la séquence (si l'incrémentation le permet). Si *c* vaut 1 on peut écrire simplement `a:b`. Lorsque la séquence est utilisée pour pointer les indices d'un tableau, le mot clé `end` correspond au dernier élément de la matrice ou de la dimension correspondante. D'autre part, la syntaxe `1:5` est une contraction strictement équivalente à une matrice contenant les nombres de 1 à 5, sous forme de vecteur ligne. Elle peut être utilisée indifféremment comme indices dans une autre matrice, comme vecteur pour un calcul ou comme séquence d'une boucle `for`.

Python

```
>>> x = 2*np.r_[1:6]
>>> print(np.block([x[0:3],np.r_[8:11]]))
[ 2  4  6  8  9 10]
>>> print(x*2)
[ 4  8 12 16 20]
>>> for y in x: print(y)
2
4
6
8
10
```

GNU Octave

```
> x = 2*(1:5);
> disp([x(1:3),8:10])
    2    4    6    8    9   10
> disp(x*2)
    4    8   12   16   20
> for y = x, disp(y), end
2
4
6
8
10
```

A.2.2.5 Les chaînes de caractères

En *Octave*, les chaînes de caractères sont aussi des tableaux pouvant être manipulés comme des vecteurs numériques ! La chaîne `'octave'` par exemple est en fait un vecteur des codes ASCII correspondants aux 6 lettres le composant. Ainsi :

GNU Octave

```
> double('octave')
ans =
   111    99   116    97   118   101
```

À l'inverse, pour afficher les 10 premières lettres majuscules de l'alphabet, on peut directement créer une chaîne à partir des codes ASCII de 65 à 74 :

GNU Octave

```
> char(65:74)
ans = ABCDEFGHIJ
```

Cette simplicité semble un peu anecdotique mais ouvre de belles perspectives pour écrire des programmes de cryptographie !

En revanche, cela conduit à une sérieuse différence de comportement, puisqu'avec *Octave*, toute

opération arithmétique sur une chaîne de caractères conduit à un calcul sur les valeurs numériques des codes ASCII, ce qui peut conduire à une erreur si les dimensions ne sont pas compatibles, alors qu'avec *Python* une syntaxe équivalente génère soit une erreur, soit une réplique de la chaîne dans le cas particulier de la multiplication.

Python

```
# le '+' concatène les chaines
>>> 'python' + '3.9'
'python3.9'

# le '*' duplique la chaine
>>> 'python'*2
'pythonpython'



>>> 'python'+1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str ...
```

GNU Octave

```
> ['octave', '7.2'] % concaténation
> strcat('octave','7.2') % alternative
ans = octave7.2

> repmat('octave',1,2)
ans = octaveoctave
% le '*' multiplie le code ASCII
> 'octave'*2
ans =
   222   198   232   194   236   202

> 'octave'+1
ans =
   112   100   117    98   119   102
```

A.2.2.6 Le point-virgule

En *Octave*, le point-virgule à la fin d'une instruction a deux utilités :

1. il permet de séparer deux instructions sur la même ligne (comme la virgule) ;
2. il est nécessaire pour ne pas afficher le résultat à l'écran : il est donc recommandé de terminer les instructions par un point-virgule pour éviter que votre code n'affiche à l'écran tout le contenu des variables manipulées. L'omission du point-virgule est en revanche très utile pour afficher le contenu d'une variable en ligne de commande, sans avoir recours à la fonction `disp()`.

Pour rappel, avec *Python* le point-virgule sert également à séparer les instructions sur une même ligne et n'a donc pas d'utilité en fin de ligne. La règle d'affichage ou non d'un résultat est plus "logique" : si le calcul n'est pas stocké dans une variable alors il sera affiché, sinon le stockage reste muet.

Python

```
>>> a = 1 + 3

>>> b = 2/5
>>> a+b
4.4
```

GNU Octave

```
> a = 1 + 3
a = 4
> b = 2/5;
> a+b
ans = 4.4000
```

A.3 *Octave* en action

Dans tout ce qui va suivre, nous vous recommandons d'exécuter la commande `help` *fonction* pour chacune des fonctions énoncées (aide rapide) ou `doc` *fonction* (aide détaillée) ; vous obtiendrez ainsi une description concise ou exhaustive en complément des grandes lignes abordées dans ce cours.

A.3.1 Variables et matrices

A.3.1.1 Les types de variables (classes)

Octave permet de manipuler une quinzaine de types de variables, appelées *classes*, pouvant toutes être attribuées en tableaux. Je vous les présente en six groupes :

1. **double** : variable numérique flottant double précision, permettant d'effectuer tous les calculs numériques possibles avec le maximum de précision (64 bits) ;
2. **logical** : variable booléenne, permettant de stocker le résultat d'un test, et prenant uniquement la valeur 0 (FAUX) ou 1 (VRAI) ; notez que les fonctions `true` et `false` permettant de définir plus explicitement ces valeurs logiques.
3. **char** : variable caractère, permettant de stocker une chaîne de caractères alpha-numérique ;
4. **cell** : variable cellule, permettant de stocker des variables de types différents dans un même tableau, repérées par leur indice ;
5. **struct** : variable structure, permettant de stocker des variables de types différents, repérées par des champs nominatifs ;
6. **single, int8, uint8, int16, uint16, int32, uint32, int64, uint64** : autres types de variables numériques permettant de stocker et manipuler différentes précisions (pour la manipulation de fichiers binaires par exemple) ou d'optimiser la mémoire. Respectivement : nombres réels flottants simple précision, nombres entiers signés ou non et codés sur 8, 16, 32 ou 64 bits. Attention les opérateurs autorisés sur ces types de variables sont limités.

Dans les exercices qui suivent, nous nous limiterons à l'utilisation des 4 premiers types, mais je vous incite à utiliser les structures si vous souhaitez rationaliser vos programmes, notamment pour la gestion de données et métadonnées.

Pour connaître les types et la taille des tableaux de vos variables, utilisez la commande `whos`, qui retournera par exemple :

GNU Octave

```
> whos

   Attr Name        Size                     Bytes  Class
   ==== ====        ====                     =====  =====
        a           1x4                          0  cell
        b         100x100                    80000  double
        c           1x1                          1  logical
        d           1x4                          4  char
```

A.3.1.2 Les tableaux (matrices) et leurs indices

Contrairement aux cellules et aux structures, les variables numériques et les caractères peuvent être regroupées en tableaux seulement si elles sont du même type. Nous allons nous attarder aux tableaux de variables numériques, les matrices.

```
a = 1;
```

est un scalaire (une matrice de dimensions 1×1).

```
b = [1 2 3 4 5]
```

définit un vecteur ligne (matrice de dimensions 1×5). Notez que l'on peut indifféremment utiliser l'espace ou la virgule pour séparer les colonnes. Sans le point-virgule à la fin de l'instruction, *Octave* affiche le résultat à l'écran :

```
b =
     1     2     3     4     5
```

Pour accéder à la 3ème valeur du tableau `b`, on utilisera la syntaxe `b(3)`.

```
c = [1 2 3;4 5 6]
```

est la matrice de dimensions 2×3 :

```
c =
     1     2     3
     4     5     6
```

Le point-virgule (éventuellement suivi d'un retour charriot) sépare les lignes de la matrice. Pour accéder aux valeurs, deux solutions : soit indiquer les deux indices (ligne,colonne) par exemple

```
c(2,1)
ans =
     4
```

soit indiquer l'indice global unique de la variable qui compte par ordre de colonnes (éléments de la 1ère colonne, puis ceux de la 2ème colonne, ...), ainsi `c(5)` vaut 3.

On peut utiliser des vecteurs d'indices pour accéder à plusieurs valeurs d'un tableau. Les indices sont gérés comme n'importe quel tableau numérique, mais doivent être des nombres entiers strictement positifs. Ainsi `c(2,[1 3])` renverra les deux valeurs `[4 6]` sous forme de vecteur ligne.

Pour accéder à toute une ligne ou toute une colonne, il faut utiliser les deux points. Ainsi, `c(:,1)` correspond à la première colonne de la matrice, c'est-à-dire `[1;4]`. Si vous avez compris l'histoire de l'index global, vous savez que `c(:)` renverra toute la matrice "déroulée" sous forme d'un unique vecteur colonne (par convention) `[1;4;2;5;3;6]`.

Enfin, les indices de tableaux acceptent un mot clé `end` qui correspond à la dernière valeur de la ligne, de la colonne ou de la matrice entière pour les indices globaux.

A.3.1.3 Définir des vecteurs et matrices élémentaires

Il existe plusieurs fonctions permettant de créer des tableaux prédéfinis de taille quelconque.

: (deux points)
L'expression `debut:fin` génère un vecteur ligne de valeurs espacées de 1 en partant de `debut` et jusqu'à `fin`. On peut également spécifier un intervalle avec `debut:pas:fin`. Cette méthode est à privilégier lorsque l'on veut imposer l'incrément d'un vecteur plutôt que le nombre total d'éléments. Elle est particulièrement bien adaptée aux indices de tableaux par exemple, et vous aurez compris que les deux points seuls `:` sont en fait la contraction de `1:end`. Notez que suivant les valeurs de `debut`, `pas` et `fin`, on peut par erreur obtenir un vecteur vide (par exemple, `1:-1`).

linspace, logspace
La fonction `linspace(debut,fin,N)` va créer un vecteur de N éléments espacés linéairement entre `debut` et `fin`. Cette méthode est à privilégier lorsque l'on souhaite imposer le nombre d'éléments plutôt que le pas d'incrémentation. Elle a l'avantage de ne jamais renvoyer un vecteur vide. La fonction cousine `logspace(X1,X2,N)` permet de définir des intervalles sur une échelle logarithmique avec N éléments de 10^{X_1} à 10^{X_2}.

nan, zeros, ones, rand
On peut définir un tableau ne contenant que des NaN (*Not a Number* une valeur particulière des nombre flottants) avec `nan(M,N)`, que des 0 avec `zeros(M,N)`, que des 1 avec `ones(M,N)`, ou encore avec des nombres aléatoires uniformément répartis entre 0 et 1 `rand(M,N)`. À chaque fois M est le nombre de lignes et N le nombre de colonnes du tableau résultant. Attention les formes contractées `nan(N)`, `zeros(N)`, `ones(N)` et `rand(N)` renvoient un tableau de taille $N \times N$ (matrice carrée) ; si N est trop grand la fonction entrainera une erreur (saturation de mémoire).

repmat
Cette fonction permet de répliquer dans n'importe quelle dimension un scalaire, un vecteur ou une matrice. Par exemple `repmat(b,2,1)` duplique le vecteur `b` (défini ci-avant) dans sa première dimension pour en faire une matrice 2×5. De même `repmat(1,M,N)` produira le même résultat que `ones(M,N)`.

[ ], cat
La concaténation de tableaux s'opère soit avec la syntaxe des crochets (et espace/virgule et point-virgule) pour les vecteurs et matrices, soit avec la fonction `cat` qui permet d'adresser les dimensions supérieures à 2.

meshgrid
L'instruction `[X,Y]=meshgrid(x,y)` renvoie, à partir des vecteurs `x` et `y`, deux matrices `X` et `Y` de mêmes dimensions pouvant servir de grilles de coordonnées.

A.3.1.4 Regrouper des variables de types différents (cellules)

Octave permet de définir des tableaux contenant des éléments de différentes tailles ou même de différents types. Ce peut être très utile pour structurer des données, mais évidemment il ne sera pas possible d'y appliquer des calculs de façon vectorielle. Les cellules sont définies avec les accolades `{ }` et ont le type de variable *cell*. Reprenons par exemple la matrice `c` de dimension 2×3 définie plus haut :

```
d={c sum(c) prod(c(:))};
```

crée une variable `d` de type *cell* contenant respectivement la matrice `c`, le vecteur des sommes des colonnes, et le produit de tous les éléments sous forme d'un tableau 1×3 :

```
d =
    [2x3 double]    [1x3 double]    [720]
```

Notez que pour de simples raisons d'affichage, *Octave* présente la taille et le type de variable de chaque élément plutôt que son contenu, sauf si celui-ci est un scalaire (cas du 3ème élément dans l'exemple).

Pour accéder au deuxième élément du tableau de cellules, on utilisera la syntaxe `d{2}` qui renverra le vecteur des sommes. Notons que la syntaxe `d(2)` est également autorisée mais renvoie une cellule d'un seul élément contenant `d{2}`, et non directement son contenu (une matrice numérique).

Les cellules peuvent ainsi être utilisées pour stocker des matrices de taille différentes au sein d'une même variable. Par exemple :

```
M = cell(8,1);
for n = 1:8
   M{n} = rand(n);
end
```

commence par définir un tableau de cellule vide puis le remplit avec des tableaux de nombres aléatoires de taille croissante : `M(1)` est un scalaire et `M(8)` est une matrice 8×8.

A.3.2 Les opérateurs de base

Hormis les opérateurs arithmétiques purement matriciels qui doivent respecter les règles du calcul matriciel, les opérateurs arithmétiques et relationnels peuvent s'appliquer soit entre deux scalaires, soit entre deux tableaux de même taille, soit entre un scalaire et un tableau de taille quelconque.

A.3.2.1 Les opérateurs arithmétiques

Par défaut, les opérateurs arithmétiques s'appliquent aux matrices : `+` (addition), $-$ (soustraction), `*` (produit), `/` (division), `\` (division gauche), `^` (puissance), `'` (transposé conjuguée complexe), et `( )` (regroupement d'expressions pour imposer l'ordre d'exécution).

Pour travailler sur les tableaux et que les opérations s'appliquent à tous les éléments de façon vectorielle, il faut ajouter un point avant les opérateurs suivants : `.*` (multiplication), `./` (division), `.\` (division gauche), `.^` (puissance).

Ainsi pour élever au carré tous les éléments de `c`, il ne faut pas écrire :

```
c^2
error: for x^y, only square matrix arguments are permitted and one argument must be scalar.
Use .^ for elementwise power.
```

ce qui a tenté de calculer le produit de la matrice *c* par elle-même ; mais en revanche :

```
c.^2
ans =
     1     4     9
    16    25    36
```

qui aurait pu s'écrire aussi `c.*c`. Autre exemple, calculer l'inverse de chaque élément de `b` :

```
1./b + 1
ans =
    2.0000    1.5000    1.3333    1.2500    1.2000
```

A.3.2.2 Les opérateurs relationnels et logiques

Pour comparer des expressions entre elles, on dispose des opérateurs classiques : > (strictement plus grand que), < (strictement plus petit que), >= (plus grand que ou égal à), <= (plus petit que ou égal à), == (égal à) et ~= (différent de).

Le résultat de la comparaison sera une variable booléenne de la même taille que les éléments comparés. Par exemple, `d=(3<2)` affecte à `d` la valeur 0 (FAUX), alors que

```
c>2
ans =
     0     0     1
     1     1     1
```

renvoie une matrice de même taille que *c* contenant des 1 pour les éléments strictement supérieurs à 2, et 0 pour les autres.

Suivant la même logique, pour tester l'égalité de deux matrices de même dimensions *A* et *B*, l'expression `A == B` retournera une matrice avec des 1 pour les éléments égaux et des 0 pour les autres. Pour intégrer ce résultat dans un test conditionnel, la bonne façon est d'utiliser la fonction `isequal(A,B)` qui renverra un unique 1 en cas d'égalité.

La commande spéciale `find` permet de renvoyer un vecteur d'indices des éléments vrais (de valeur non nulle) d'une expression. Ainsi, `find(c>=4)` sera un vecteur colonne (par convention) `[2;4;6]`. *Octave* autorise un raccourci de syntaxe permettant de se passer la plupart du temps de la commande `find` : lorsque les indices de tableaux sont des variables booléennes (par exemple le résultat d'un test), il n'adresse que les éléments associés à une valeur vraie. Ainsi, `c(c>=4)` est équivalent à `c(find(c>=4))` et renvoie les valeurs `[4;5;6]`.

Notons aussi les deux fonctions particulières suivantes : `all(A)` retourne 1 si tous les éléments du vecteur *A* sont vrais (ou des colonnes de *A* si c'est une matrice), 0 sinon ; et `any(A)` retourne 1 si au moins l'un des éléments de *A* est vrai, 0 s'ils sont tous faux.

Les comparaisons peuvent être inversées par ~ (NÉGATION), ou liées entre elles par les opérateurs logiques `&` (ET), `|` (OU), `XOR` (OU EXCLUSIF).

A.3.3 Les contrôles de flux conditionnels

A.3.3.1 if, else, elseif

L'instruction `if` permet d'évaluer une expression et d'exécuter des commandes particulières si elle est vraie. Les instructions optionnelles `else` et `elseif` permettent de définir d'autres commandes à exécuter alternativement si la condition est fausse ou suivant d'autres expressions. La structure générale s'écrit comme ceci :

```
if CONDITION1
    commandes à effectuer si CONDITION1 est VRAI
elseif CONDITION2
    commandes si CONDITION1 est FAUX mais que CONDITION2 est VRAI
elseif CONDITION3
    commandes si CONDITION1 et CONDITION2 sont FAUX mais CONDITION3 est VRAI
...
else
    commandes si CONDITION1 et CONDITION2 et ... sont tous FAUX
end
```

Notez que l'on peut omettre `elseif` et/ou `else` si l'on ne veut rien exécuter lorsque la première condition est fausse.

A.3.3.2 switch, case, otherwise

C'est une variante permettant d'exécuter une série de commandes suivant la valeur d'une variable ou d'une expression.

```
switch VARIABLE
    case VALEUR1
        commandes à effectuer si VARIABLE égale VALEUR1
    case VALEUR2
        commandes à effectuer si VARIABLE égale VALEUR2
    ...
    otherwise
        commandes à effectuer par défaut
end
```

La partie `otherwise` est facultative. *valeur1* et *valeur2* peuvent être une cellule contenant plusieurs variables, auquel cas le test sera vrai si *variable* est égal à au moins l'une des valeurs de la cellule.

A.3.4 Les boucles

Si l'on veut exécuter plusieurs fois un jeu de commandes, il existe deux instructions.

A.3.4.1 for

```
for VARIABLE=EXPRESSION
   commandes à effectuer pour chaque colonne de EXPRESSION
end
```

La boucle est exécutée en attribuant successivement à *variable* chaque colonne de *expression*. Si *expression* est un simple vecteur ligne de N éléments, la boucle va s'exécuter N fois. Par exemple :

```
for a = 1:10
    disp(a)
end
```

affiche les valeurs successives de a de 1 à 10, et à la fin de la boucle, a vaut 10.

expression peut aussi contenir un vecteur quelconque comme `[1,3,-4,8]`, ou encore n'importe quel résultat de fonctions présentées au paragraphe A.3.1.3. Autre exemple, pour faire une boucle sur 10 nombres aléatoires, on écrira `for a=rand(1,10)`, plutôt qu'une boucle sur un indice de 1 à 10 qui pointerait ensuite sur un vecteur prédéfini de 10 nombres aléatoires (c'est ce qu'on aurait dû faire en langage *C*).

Enfin si *expression* est une matrice, *variable* vaudra successivement chaque colonne de cette matrice ;

ceci est particulièrement utile pour le calcul vectoriel (éviter une seconde boucle par exemple).

Note importante : s'il est fortement déconseillé de modifier la valeur de *variable* au sein d'une boucle (c'est très inélégant et apporte de la confusion à la lisibilité du code), il faut savoir que contrairement au langage *C* ou *Fortran*, *Octave* ignorera d'éventuels changements à chaque nouvelle itération de la boucle, se contentant d'assigner à *variable*, sans broncher, chaque élément successif de *expression*.

A.3.4.2 while

```
while CONDITION
   commandes à effectuer tant que CONDITION est VRAI
end
```

La boucle est exécutée tant que *condition* est vrai. Si rien ne modifie *condition* dans les commandes, la boucle est effectuée à l'infini. A contrario, si *condition* est FAUX dès le départ, les commandes seront ignorées.

A.3.4.3 continue, break

Au sein d'une boucle `for` ou `while`, l'instruction `continue` permet d'ignorer le reste des commandes jusqu'au `end` et de poursuivre la boucle.

L'instruction `break` force la sortie de la boucle (poursuit le programme immédiatement après l'instruction `end`).

A.3.5 Exploitation du calcul vectoriel

Sous *Octave*, le calcul vectoriel n'est pas qu'une simple question d'optimisation, c'est une réelle nécessité. Voici quelques petits exemples pour vous faire prendre conscience de l'importance de cet aspect dans un programme.

A.3.5.1 Opérateurs arithmétiques

Nous souhaitons remplir un tableau de 100 000 nombres aléatoires entre -10 et 10. Ci-dessous deux codes permettant d'accomplir cette tâche. Nous utilisons les instructions `tic` et `toc` qui donnent le temps d'exécution d'un bloc de commandes, ici sur un petit ordinateur portable sans prétention.

GNU Octave

```
n = 100000;
x = zeros(n,1);
tic
for k = 1:n
   x(k) = rand*20 - 10;
end
toc
```

```
Elapsed time is 0.569022 seconds.
```

Dans ce premier exemple, la matrice/vecteur `x` est remplie par un scalaire à chaque itération de la boucle. Mais comme nous l'avons dit en début du document, avec *Octave* on doit autant que possible éviter les boucles sur les indices de tableaux en utilisant la fonctionnalité vectorielle des instructions, ce qui donnerait le programme suivant :

GNU Octave

```
n = 100000;
tic
x = rand(n,1)*20 - 10;
toc
```

```
Elapsed time is 0.00133109 seconds.
```

C'est-à-dire avec un temps d'exécution près de 500 fois plus rapide que la première version ! La raison est que le langage *Octave* a été conçu pour travailler sur des matrices, et que la boucle sur les éléments a cette fois été exécutée de façon optimisée pour le processeur au cœur du langage.

Dans cet exemple simple, il ne s'agit que d'un gain de fractions de secondes. Mais imaginez qu'en traitant de gros volumes de données ou en effectuant une série d'opérations plus complexes, un code bien vectorisé qui durerait par exemple 10 minutes mettrait près de trois jours dans sa version "mal écrite" ! À ce stade il ne s'agit plus vraiment d'*optimisation*, mais de façon plus pragmatique, de simple *faisabilité*.

A.3.5.2 Opérateurs de tests relationnels

Dans ce second exemple on va utiliser notre tableau de 100 000 nombres aléatoires entre 0 et 1 et compter le nombre de valeurs strictement supérieures à 0,5. En programmation classique, on écrirait une boucle et un test à l'intérieur :

GNU Octave

```
n = 100000;
x = rand(n,1); % vecteur de n nombres aléatoires entre 0 et 1

N = 0;
for k = 1:n
   if x(k) > 0.5
      N = N + 1;
   end
end
```

Voici maintenant la version vectorielle :

GNU Octave

```
N = sum(x > 0.5)
```

Vous voyez qu'en une seule instruction, vous avez effectué l'équivalent d'une boucle, d'un test et d'un incrément de variable. Non seulement cette deuxième version est plus rapide, mais elle a surtout gagné en clarté ! Explications : comme on a vu précédemment le test sur le vecteur `x` renvoie un vecteur même taille avec des nombres booléens (1 pour VRAI et 0 pour FAUX). La fonction `sum` fait la somme de ce vecteur et renvoie donc le nombre d'éléments ayant respecté la condition de test. On aurait aussi pu écrire :

```
N = length(find(x > 0.5))
```

Ici on a explicitement demandé de rechercher les valeurs > 0.5 (instruction `find` qui renvoie la liste des indices de la matrice `x` respectant le test) et d'en calculer le nombre d'éléments (instruction `length` qui renvoie la longueur d'un vecteur).

Question : Comment auriez-vous fait pour calculer la somme des valeurs de x supérieures à 0.5 ? Et bien tout simplement comme ceci :

```
N = sum(x(x > 0.5))
```

où le test renvoie les indices booléens qui pointent sur les valeurs du vecteur...

A.3.5.3 Sous quelles conditions un programme est-il vectorisable ?

Il n'y a pas de réponse globale à cette question. Mais si une partie de l'algorithme consiste à appliquer un calcul N fois de façon indépendante (à un ensemble de points par exemple), il y a de fortes chances que l'on puisse le vectoriser en constituant des vecteurs ou des matrices de dimension N. En revanche si les calculs individuels ne sont pas indépendants les uns des autres, par exemple dans le cas d'une boucle itérative (le résultat i dépend du résultat $i-1$), la vectorisation sera plus délicate, voire impossible.

A.3.6 La gestion des entrées/sorties

A.3.6.1 Saisie au clavier (entrée standard)

Pour demander à l'utilisateur d'entrer une information au clavier, on utilise l'instruction `input`, par exemple :

```
nom = input('quel est votre nom?');
```

On note dans cette instruction que c'est une affectation. On affecte à la variable `nom` ce que l'utilisateur a entré au clavier. L'argument de `input` est une variable caractère (donc soit une variable caractère définie par ailleurs, soit une chaîne de caractères entre apostrophes), qui sera affichée à l'écran, permettant à l'utilisateur de savoir ce qu'on lui veut.

Notez enfin qu'il est préférable, pour exploiter les choix d'un utilisateur, d'écrire une fonction avec des arguments d'entrée (voir paragraphe A.3.7) plutôt que d'utiliser cette forme interactive un peu désuète.

A.3.6.2 Écrire à l'écran (sortie standard)

Par défaut, *Octave* écrit à l'écran le résultat de toutes les expressions calculées ou assignation de variables, sauf si la ligne contenant l'expression est suivie d'un point-virgule. L'instruction permettant d'afficher la valeur d'une variable quelconque X est `disp(X)`, équivalant à la fonction *Python* `print`. Une autre instruction légèrement différente, `display(X)`, fait la même chose mais annoncera en plus le nom de la variable. Pour écrire une chaîne de caractères quelconque, on va par exemple utiliser `disp('Ceci est un message')`, ou encore intégrer un contenu de variable numérique par conversion en chaîne de caractères et concaténation : `disp(['La variable a = ',num2str(a)])`.

Mais si l'on souhaite écrire ce que l'on veut avec un format particulier, on utilisera la commande très évoluée et proche du langage *C*, `fprintf` :

```
fprintf(FORMAT,VAR1,VAR2,...)
```

où *format* est une chaîne de caractères contenant le texte à écrire qui peut inclure des formats spécifiques (par exemple `%f` pour un nombre flottant, `%d` pour un entier, `%s` pour une chaîne de caractère, ...) à chaque endroit où l'on veut faire apparaître une variable. Celles-ci sont ensuite spécifiées comme arguments *var1*, *var2*, ... dans l'ordre d'apparition des opérateurs % de la chaîne *format*. L'équivalent de notre exemple précédent avec `disp` serait : `fprintf('La variable a = %f\n',a)`. Notez le `\n` en fin de chaîne correspondant à une nouvelle ligne.

Les formats autorisés peuvent être évolués (spécifier le nombre de chiffres avant et après la virgule par exemple) et concernent tous les types de variables. Ainsi `fprintf` est la fonction la plus générale pour convertir n'importe quoi en chaîne de caractères. Signalons sa fonction jumelle `sprintf` qui a les mêmes arguments d'entrée mais renverra la chaîne de caractère résultante en sortie (donc pouvant être utilisé par une autre fonction, ou attribuée à une variable) plutôt que d'écrire sur la sortie standard.

Par rapport à la fonction homonyme en *C*, `fprintf` de *Octave* est "vectorisée" : si une variable en argument est un tableau, la chaîne *format* sera répétée autant de fois que les éléments du tableau (dans l'ordre des colonnes, comme d'habitude).

A.3.6.3 Écrire dans un fichier

S'il s'agit simplement de sauvegarder toutes les valeurs d'une matrice numérique, par exemple la variable A, on utilisera la fonction :

```
save NomFichier A -ascii
```

ou encore la fonction plus évoluée :

```
dlmwrite(NomFichier,A)
```

qui permet de spécifier plusieurs options de format (séparateur, précision, ...).

Pour écrire dans un fichier d'une façon plus générale, il faut procéder en trois étapes : ouverture du fichier, écriture, et fermeture. On commence par l'ouvrir, c'est-à-dire l'identifier et lui associer un numéro :

```
fid = fopen(NomFichier,'w');
```

où *NomFichier* est une variable caractère contenant le chemin du fichier. Le `'w'` sert à spécifier qu'on va écrire dans le fichier. *fid* est l'identifiant du fichier ; s'il vaut -1, c'est que l'ouverture du fichier a échoué, sinon, on peut commencer à écrire. Cela se fait par l'instruction `fprintf` vue précédemment, mais en lui indiquant *fid* en premier argument :

```
fprintf(fid,FORMAT,VAR1,VAR2,...);
```

Comme vu précédemment, la fonction `fprintf` est vectorielle et permet d'écrire un tableau entier en une seule instruction. En revanche, pour combiner des tableaux de texte et des tableaux numériques (par exemple regroupées dans un tableau de cellules), il faudra utiliser une boucle.

Afin de libérer l'identifiant de fichier (et parce que les informaticiens craignent les courants d'air), il faut fermer ce qu'on a ouvert :

```
fclose(fid);
```

A.3.6.4 Importer des données depuis un fichier

Comme toujours il y a plusieurs façons de faire et cela va dépendre 1) du type de données à importer (numérique, caractères ou les deux) et 2) de comment on compte travailler sur ces données.

Commençons par le plus simple : si le fichier contient un tableau régulier de données uniquement numériques, avec un nombre de colonnes rigoureusement constant du début à la fin du fichier, et un séparateur de colonnes type espace(s), virgule ou tabulation (certes cela fait beaucoup de conditions...), on peut utiliser la fonction `load` :

```
A = load(NomFichier);
```

qui va lire le fichier *NomFichier* et créer la variable `A`, une matrice avec toutes les valeurs du fichier. Cette méthode a l'avantage d'être extrêmement rapide, même pour les fichiers volumineux. Si le fichier comporte des lignes d'en-tête et/ou un séparateur autre, on peut utiliser la fonction `importdata`.

Cependant la plupart des fichiers de données vont avoir un format plus ou moins quelconque. Il faut alors utiliser des fonctions plus évoluées.

C'est presque pareil que pour l'écriture. On commence par ouvrir le fichier :

```
fid = fopen(NomFichier,'r');
```

où *NomFichier* est une variable caractère, et `'r'` sert à spécifier qu'on va lire dans le fichier (facultatif). La lecture du contenu peut alors se faire avec l'instruction `textscan`, qui s'utilise comme suit :

```
data = textscan(fid,FORMAT);
```

où `fid` est toujours le numéro du fichier, *format* la liste des types de données de chaque ligne du fichier (comme pour `fprintf`). La variable *data* résultante est alors un tableau de cellules. Ensuite on ferme le fichier avec `fclose(fid)`.

Voici un exemple concret d'utilisation. Soit le fichier `infos.dat` contenant les données suivantes :

```
Robert   10/12/1963  12.45  35  Oui
Louise   07/09/1976  37.93  20  Non
Arthur   28/03/1989  26.12  84  Oui
```

Nous allons le lire avec les instructions :

GNU Octave

```
fid = fopen('infos.dat');
data = textscan(fid, '%s %s %f %d %s');
fclose(fid);
```

La variable *data* est un tableau de cellules de 5 éléments, contenant :

GNU Octave

```
data{1} = {'Robert'; 'Louise'; 'Arthur'}
data{2} = {'10/12/1963'; '07/09/1976'; '28/03/1989'}
data{3} = [12.4500; 37.9300; 26.1200]
data{4} = [35; 20; 84]
data{5} = {'Oui'; 'Non'; 'Oui'}
```

Signalons enfin l'existence des instructions dites "de bas niveau", très proche du *C* : `fscanf`, `fgetl`, `fgets` et `fread` permettant de lire de façon individuelle des nombres/caractères, des lignes de texte ou des données binaires, respectivement.

A.3.7 Les scripts et fonctions (fichiers `.m`)

A.3.7.1 Scripts

Les langages interprétés permettent d'exécuter des instructions directement en ligne de commande. Ceci est très pratique pour tester rapidement un petit programme ou vérifier la syntaxe d'une fonction. Mais pour des raisons de sauvegarde, traçabilité, répétabilité et partage, il est fortement conseillé d'écrire ses instructions dans un script, c'est-à-dire un fichier texte avec l'extension `.m`. Il suffira de "lancer" le programme simplement comme une fonction *Octave* (à condition que le nom de fichier ne contienne pas d'espace ou de caractères spéciaux). L'éditeur intégré de *Octave* ainsi que beaucoup d'éditeurs libres comme *gedit* ou *vim* permettent la coloration syntaxique ce qui aide à l'écriture du code.

A.3.7.2 Fonctions

Si un script est amené à être exécuté plusieurs fois, avec des arguments différents (variables, nom de fichier, paramètres, options, ...), on peut très simplement en faire une fonction. Pour cela il faut commencer par mettre en en-tête du script l'instruction

GNU Octave

```
function [sortie1,sortie2,...] = mafonction(entree1, entree2,...)
```

et sauver le fichier sous le nom `mafonction.m`. Les instructions utiliseront les variables d'entrée (et seulement celles-là) et devront définir ou calculer toutes les variables de sortie. Notez que toutes les variables utilisées dans la fonction seront locales et ne tiendront pas compte des variables de l'environnement appelant la fonction. Une fonction ne doit donc surtout pas commencer par `clear all` ce qui effacerait toutes les variables d'entrée. La fonction sera ensuite appelée par le programme principal (ou une autre fonction), comme n'importe quelle autre fonction *Octave*.

Vous pouvez aussi très facilement intégrer une aide en ligne qui sera vue par `help`, `doc` et `lookfor`, en ajoutant des lignes de commentaires immédiatement après la première ligne. Prenez exemple sur les fonctions *Octave*, par exemple en tapant

GNU Octave
```
type rank
```

Voici un exemple simple : nous allons créer une fonction qui calcule le module d'un vecteur. Imaginons qu'un programme ait besoin de calculer plusieurs fois un module sur des scalaires et des vecteurs :

GNU Octave
```
x1 = 15;
y1 = -8;
m1 = sqrt(x1^2 + y1^2);
...
x2 = [2.3,-11.7];
x2 = [-5.9,6.1];
m2 = sqrt(x2.^2 + y2.^2);
```

Voici la fonction qui sera un fichier nommé `module.m` :

GNU Octave
```
function y=module(x,y)
%MODULE Module of components.
%   MODULE(X,Y) returns the module of vectors or scalars X and Y.
y = sqrt(x.^2 + y.^2);
```

La fonction utilise `x` et `y` comme variables d'entrée (il n'y a donc pas de `x=` ou `y=` dans le code) et calcule la variable de sortie `y`. Dans le programme principal, nous pourrons utiliser la fonction comme ceci :

GNU Octave
```
...
m1 = module(x1,y1);
...
m2 = module(x2,y2);
```

Il y aurait beaucoup d'autres choses à dire sur les fonctions... je me contenterai d'énumérer les possibilités suivantes :

- définir des sous-fonctions dans le même fichier ;
- utiliser des variables globales (voir `global`) ;
- utiliser un nombre variable d'arguments d'entrée ou de sortir (voir `varargin` et `varargout`) ;
- définir des fonctions privées qui ne seront vues que par certaines fonctions (dans un sous-répertoire `private`) ;
- définir une fonction anonyme, sans fichier `.m` associé (voir `@`).

A.3.8 La gestion des dates et heures

C'est un problème crucial que chaque langage de programmation a abordé de façon particulière. Le plus classique est de considérer le temps absolu comme la combinaison d'une date (année, mois, jour) et d'une heure (heure, minutes, secondes et fractions de secondes), soit 6 ou 7 scalaires dont il faut contrôler les intervalles de validité (en particulier le nombre de jours dans chaque mois et les années bissextiles).

Au cœur des systèmes informatiques, l'horloge des ordinateurs est manipulée par un nombre de secondes depuis la date conventionnelle du 1er janvier 1970 à 00:00:00), initialement codé sur un entier 32 bits. C'est ce qu'on appelle communément l'"heure Unix" ou POSIX.

Le problème de ce système est qu'il doit être complété par un autre compteur pour les fractions de secondes (on perd alors l'intérêt d'un nombre réel unique pour coder le temps), et surtout qu'il est limité : sur une machine 32 bits il ne peut coder que des dates entre le 13 décembre 1901 à 20:45:52 et le 19 janvier 2038 à 03:14:07. Heureusement, les machines modernes à 64 bits ont étendu ces limites à plusieurs centaines de milliards d'années. Mais si en 2038 il reste des ordinateurs 32 bits en fonction, il y a fort à parier qu'ils présenteront le "bogue du 19 janvier 2038" qui remettra leur horloge au 13 décembre 1901 !

Octave a choisi un système différent : ce qu'on appelle communément le "temps Matlab" est le nombre de jours décimaux depuis la date virtuelle du 0 janvier de l'an 0000 (qui en pratique n'existe pas, mais peu importe), codé par un flottant double précision (64 bits, y compris sur les machines 32 bits). Ceci permet de définir toutes les dates et heures sur des millénaires avec une précision d'environ 10 μs ! Par exemple la date du 5 septembre 2022 à 10:58:35 correspond au nombre 738769.4573495371. La fonction `now` renvoie le temps du système.

Pour convertir n'importe quelle date et heure en temps Matlab, il y a la fonction `datenum` qui accepte soit une chaîne de caractères, soit un vecteur numérique avec année, mois, jour, heure, minute, seconde. Cette fonction est très tolérante et souple d'utilisation : par exemple `datenum(2022,0,248)` renverra la date correspondant au jour ordinal 248 de l'année 2022 (5 septembre). Ou encore l'heure Unix peut être convertie très simplement par `datenum(1970,1,1,0,0,unixtime)`.

Pour afficher et convertir le temps Matlab en date et heure lisibles par un humain, il y a la fonction `datestr` qui renvoie une chaîne de caractères, et la fonction `datetick` spécifiquement pour convertir les labels d'axes graphiques, et la fonction`datevec` qui renvoie les 6 valeurs correspondantes année, mois, jour, heure, minute et seconde.

Toutes ces fonctions sont bien entendu vectorielles et peuvent convertir en une fois un vecteur ou une matrice de dates.

A.3.9 Édition de scripts et gestion des erreurs

Les ordinateurs sont stupides : ils ne font que ce qu'on leur demande, mais ce défaut est justement aussi leur grande qualité ! Lorsque vous lancez l'exécution d'une série d'instructions (faire tourner un script), il peut arriver que "ça plante". *Octave* affiche un message d'erreur aussi austère qu'explicite : il vous dira exactement — et uniquement — où il n'est pas d'accord (nom de la fonction et numéro de ligne de code) et pour quelle raison. Il y a deux types de message d'erreur :

A.3.9.1 Les erreurs de syntaxe

Ce sont les plus faciles à corriger car le message vous indiquera exactement où votre code ne respecte pas l'orthographe et la grammaire des fonctions utilisées. En outre, l'éditeur intégré de *Octave* vous dira instantanément, pendant que vous écrivez, s'il y a un problème par un petit indicateur rouge (en haut à droite de la fenêtre) et de petites lignes rouges au niveau des erreurs. Cet indicateur doit être vert lorsque tout va bien.

A.3.9.2 Les erreurs d'exécution

Ce sont les plus difficiles à corriger. Le message est absolument à prendre au premier degré : il n'est pas là pour vous dire où vous avez fait l'erreur, mais *où il a rencontré un problème d'exécution*, ce qui peut être totalement différent. C'est ce qu'on appelle un "bug" et c'est à vous de remonter le fil de votre programme, en vous mettant à la place de l'interpréteur, pour détecter la faute commise.

A.3.10 Quelques conseils pour terminer

Un langage c'est une grammaire, une orthographe et... un style ! Voici enfin quelques conseils de programmation, que vous n'êtes pas obligés de suivre si vous avez vos propres habitudes, bien entendu, mais

j'y donne aussi quelques règles incontournables. Souvenez-vous que plus un code est écrit proprement, plus il est lisible ce qui a un double avantage : d'une part il sera plus facile d'y détecter un bogue, et d'autre part il ne rebutera pas un éventuel utilisateur/développeur autre que vous!

A.3.10.1 Noms de variables

Si le code est long et complexe, utilisez des noms de variable explicites, tout en restant concis : `NbMaxPts` par exemple, semble plus approprié que `n` (un peu court...) ou que l'interminable, quoique formellement correct[4], `le_nombre_maximum_de_points_dans_mon_vecteur`. Réservez les variables à un seul caractère aux variables génériques clairement identifiables, comme des coordonnées `x` et `y`, ou encore `k` pour renvoyer les indices d'une matrice.

Avant d'assigner une nouvelle variable, vérifiez que son nom n'est pas déjà utilisé par une variable ou fonction existantes (éventuellement les vôtres) ou encore un mot clé réservé. Pour cela, utilisez la commande `which -all` *variable*.

Octave comporte quelques variables prédéfinies : `i` et `j` (nombre imaginaire $\sqrt{-1}$), `pi` (π), `eps` (précision relative des nombres flottants $\varepsilon = 2^{-52}$), `inf` ou `Inf` (infini), `nan` ou `NaN` (Not-a-Number). Si vous utilisez malgré tout ces noms de variables, ce n'est pas interdit : ils "écraseront" les valeurs prédéfinies au sein de votre fonction (localement pendant son exécution) ou de votre environnement. Vous pouvez retrouver les valeurs par défaut avec un simple `clear` *variable*.

Octave fait la différence entre majuscules et minuscules... mais les fonctions intégrées sont toutes en minuscules. Une bonne façon de clarifier un code est de mettre, par exemple, une majuscule en début de nom, ou bien tout mettre en majuscules. Pour une fois nous avons un léger avantage sur les anglophones : nous pouvons aisément distinguer nos propres variables en les écrivant en français (mais sans accent...).

Évitez de changer de nom de variable au cours d'un même code... et ne définissez une nouvelle variable que si elle doit être utilisée plus d'une fois par le programme, ou bien par soucis de clarification du code ; n'oubliez pas que définir une nouvelle variable requiert de la mémoire...

A.3.10.2 La bonne syntaxe

N'hésitez jamais à ajouter des commentaires (ligne commençant par le caractère `%`) à chaque partie ou instruction importante de votre code.

Usez de l'indentation : décalez les instructions par une tabulation au sein des contrôles de flux (`for ... end`, `if ... else ... end`, etc). L'éditeur de l'environnement intégré *Octave* peut vous y aider.

Espacez vos instructions : *Octave* tolère des espaces de part et d'autre des opérateurs `=`, `*`, `>`, etc.

Essayez de limiter vos lignes à 80 caractères maximum : sinon elles deviennent difficiles à lire (et à imprimer). Vous pouvez couper la plupart des instructions avec trois points (`...`) puis un retour à la ligne.

Mettez systématiquement un point-virgule à la fin des lignes. La plupart des fonctions *Octave* renvoient des arguments de sortie, et n'importe quel calcul ou attribution de variable renvoie son résultat, ce qui peut envahir rapidement votre écran de commande. Si vous souhaitez afficher explicitement un résultat, utiliser les fonctions `disp` ou `fprintf`, ce sera plus propre et vous permettra d'ajouter un texte explicatif.

A.3.10.3 Aide et documentation

Difficile de connaître toutes les instructions et fonctions *Octave*, et encore moins de retenir l'ordre et la signification de tous les arguments ... mais alors pour trouver votre bonheur, retenez au moins ces deux-là :

- `help` vous rappellera les différentes catégories de fonctions (y compris les *toolboxes* installées), puis `help` *catégorie* vous en donnera la liste détaillée, et enfin `help` *fonction* vous donnera la syntaxe et les arguments d'entrée et sortie ;
- `lookfor` *expression* va rechercher dans les noms et titres de fonctions n'importe quelle expression (mot entier ou partiel).

4. En pratique les noms de variable sont limités à `namelengthmax` = 63 caractères.

A.3.10.4 Partager ses fonctions

Enfin, je signale ici qu'il existe une vaste communauté d'utilisateurs *Octave* qui partagent leurs fonctions via le portail matlabcentral sous licence BSD[5]. Ceci dépasse le cadre de ce cours, mais à l'avenir, je vous incite vivement 1) à vérifier si une fonction existe avant d'entreprendre d'en écrire une, et 2) à partager vos fonctions originales avec le reste du monde.

5. La licence BSD (*Berkeley Software Distribution*) permet de réutiliser tout ou une partie du logiciel sans restriction, qu'il soit intégré dans un logiciel libre ou propriétaire.