Notes sur Matlab/Octave

1 Codage des nombres en machine

En Matlab, les nombres sont par défaut des nombres à virgule flottante en double- précision, définis par la norme IEEE 754. Ces nombres sont codés sur 64 bits, de la façon suivante :

- Le premier bit est appelé *bit de signe*. Il est à 0 pour les nombres positifs, et à 1 pour les nombres négatifs.
- Les 11 bits suivants codent, en base 2, l'exposant, qui peut être positif ou négatif. Pour obtenir l'exposant, on calcule le nombre positif correspondant et on soustrait $1023 = 2^{10}-1$. Ainsi, les exposants possibles *théoriquement* sont les nombres entiers compris entre -1023 (pour tous les bits à 0) et 1024 pour tous les bits à 1. Attention ! En pratique, ces deux exposants servent à coder des valeurs spéciales :
  - Si tous les bits sont à 0, le nombre à double-précision est un 0 signé (c'est-à-dire $+0$ ou -0) ou un nombre « sous-normal » qui code un très petit nombre, mais avec une précision réduite.
  - Si tous les bits sont à 1, le nombre à double-précision est un infini, `Inf` ou `-Inf`, ou bien un `NaN` (Not a Number), suivant les valeurs des bits de la mantisse.
- Les 52 bits restant servent à coder la *mantisse*, c'est-à-dire les chiffres significatifs dans l'écriture du nombre. Comme nous travaillons en base 2, le premier chiffre significatif est nécessairement 1 donc est omis.

Ainsi, dans un cas « normal » où le nombre positif e codé par les 11 bits d'exposant n'est ni 0 (codage `00000000000`) ni 2047 (codage `11111111111`) où on note s le bit de signe et b_1, b_2, ..., b_{52}, notre nombre s'écrit en base 2 :

$$(-1)^s \times 2^{e-1023} \times (1, b_1 b_2 \cdots b_{52}) .$$

Par exemple, le codage de 1 en double précision est `0` pour le bit de signe, `01111111111` pour l'exposant et la mantisse est composée de 52 bits égaux à `0`.

Le codage du plus petit nombre supérieur à 1 représentable en double-précision est le même sauf que le dernier bit de la mantisse est un `1`.

Pour obtenir ce nombre en Matlab, on écrit `1 + eps`. En effet, la différence entre ce dernier nombre et 1 s'appelle le *epsilon machine*. On voit que pour la double-précision, il vaut exactement 2^{-52}, ce qu'on peut facilement vérifier dans Matlab ou Octave.

Nous avons vu que, étant donné la précision limitée de ces nombres, des comparaisons comme `sin(pi)== 0` ou `(sqrt(2))^2 == 0` seront évaluées à *faux*. On fera donc toujours très attention lorsqu'on fait des tests d'égalité entre des nombres à double-précision.

2 Fin d'une instruction

- Fin de ligne (entrée) : il y a alors affichage de la valeur de l'instruction ;
- point-virgule (`;`) puis fin de ligne (entrée) : pas d'affichage de la valeur de l'instruction.
- virgule (`,`) : affichage de la valeur de l'instruction, possibilité de mettre plusieurs instructions sur une même ligne.
- point-virgule (`;`) : idem sans affichage.

On préfèrera dans la plupart des cas mettre une instruction par ligne sans affichage avec point-virgule puis entrée. Pour un affichage, on préfèrera souvent les fonctions `disp` et `printf` (en Octave) et/ou `fprintf` (en Octave et Matlab). Le format dans `fprintf` suit les mêmes règles qu'en C. Exemples :

```
>> x = 3.78 ; n = 5 ;
>> disp("x = "); disp(x);
x =
  3.7800
>> fprintf("n vaut %d et x vaut %10.4f\n", n, x);
n vaut 5 et x vaut    3.78000.
```

C'est l'instruction `format` qui indique comment sont fait les affichages des valeurs des instructions et ceux de `disp`. Par exemple `format compact` saute moins de ligne et `format long` affiche plus de décimales. Pour revenir au format par défaut, entre `format` (tout court).

3 Obtenir de la documentation

— `help abs` ou `help("abs")` pour de la documentation (ici sur la fonction `abs`) dans la fenêtre de commande ;

— `doc abs` ou `doc("abs")` pour de la documentation dans la fenêtre d'aide.

4 Priorité des opérateurs

Les principaux opérateurs sont, par ordre décroissant de priorité :

1. `()` : appel de fonction, indiçage d'un tableau.
2. `'`, `^` et `.^` : transposition et exponentiation (puissance) ;
3. `+` *unaire*, `-` *unaire* et `~` : plus et moins unaires (comme dans -3 ou $+13,7$) et négation logique ;
4. `*`, `.*`, `/` et `./`, `\`, `.\` : multiplications et divisions ;
5. `+` et `-` : addition et soustraction ;
6. `:` : opérateur de création de « lignes en progressions arithmétiques ».
7. `<`, `<=`, `==`, `~=`, `>=`, `>` : comparaisons ;
8. `&` : « et » sans court-circuit et élément par élément ;
9. `|` : « ou » sans court-circuit et élément par élément ;
10. `&&` : « et » scalaire avec court-circuit.
11. `||` : « ou » scalaire avec court-circuit.
12. `=`, `+=`, `-=`, ... : affectation.

Toutes les associativités se font de la gauche vers la droite, sauf pour les opérateurs d'affectation. Les opérateurs `&&` et `||` n'évaluent l'opérande de droite que si cela est nécessaire. Ainsi, on peut écrire `if a > 0 && 1 / a < 7` sans crainte de recevoir un avertissement si $a = 0$. En Octave, on a également les opérateurs de pré/post - incrémentation/décrémentation (`++a`, `a++`, `--a`, `a--`) mais cela ne fait pas partie du langage Matlab.

5 Fonctions et fichiers de fonctions

```
function [ret1, ret2, ...] = nom_fonc (par1, par2, ...)
  expr1;
  expr2;
  %...
  expr3;
end
```

— `nom_fonc` est le nom de la fonction ;
— `ret1`, `ret2`, ... sont ses valeurs de retour : il peut y en avoir une, plusieurs, ou aucune et s'il n'y en a qu'une les crochets sont optionnels ;
— `par1`, `par2`, ... sont ses paramètres : il peut y en avoir un, plusieurs ou aucun ;
— `expr1`, `expr2`, ... sont les expressions (plus ou moins complexes) qui forment le *corps* de la fonction. Attention ! Les valeurs de retours, lorsqu'il y en a, doivent impérativement être valorisées dans le corps de la fonction.

On peut sauvegarder une fonction dans un *fichier de fonction*.

— Le fichier de fonction a une extension `.m` ;
— il commence impérativement par le mot-clé `function` (sauf commentaires) ;
— il a le même nom que la (première) fonction ;
— seule la *première* fonction définie dans le fichier est appelable depuis la fenêtre de commande ou un autre fichier (de script ou de fonction), les autres, appelées « sous-fonctions » ne sont appelables qu'à l'intérieur du fichier où elles sont définies.

Un fichier de fonction qui est dans le `path` (et celui-ci contient le répertoire courant) est automatiquement rechargé si besoin à chaque appel de la fonction.

Attention ! Pour obtenir toutes les valeurs de retour d'une fonction qui en a plusieurs, il faut les demander explicitement, sinon seule la première est retournée. Par exemple, pour la fonction `max`, si on veut à la fois le maximum `M` d'un vecteur ligne `L` et son indice `idM` dans la ligne, on entre l'instruction `[M, idM] = max(L);`.

6 Fichier de script

— Un fichier de script est un fichier texte dont l'extension est `.m` *qui ne commence pas par le mot-clé* `function`.
— On peut être sûr de ne pas faire d'étourderie en commençant systématiquement les fichiers de script par l'instruction `1;`.
— On appelle un fichier de script d'une des deux façons suivantes :
  — En tapant son nom *sans l'extension* `.m` si il se trouve dans un répertoire du `path` ;
  — En utilisant la fonction `run("chemin_vers_script")`. Le chemin peut-être relatif ou absolu et l'extension `.m` du fichier de script peut être présente ou non.
— Un fichier de script peut contenir une plusieurs fonctions (dans Octave ou Matlab supérieur à R2016b) qui peuvent toutes être appelées après exécution du script. Attention ! En revanche, il faudra toujours recharger le fichier de script après modification des fonctions.

7 Typage faible et dynamique

On ne déclare pas les variables et on ne donne pas leur type. Lorsqu'on exécute par exemple l'instruction `x = 5;`, la variable `x` est créée si elle n'existait pas déjà, son type devient `double` qui est le type par défaut pour les nombre et sa dimension devient `1x1`.

On peut ensuite changer le type de la variable `x`, ainsi que sa dimension. Par exemple exécuter à la suite `x = [true false];` ne provoquera pas d'erreur et fera de `x` une variable de type `logical` et de dimensions `1x2`.

Le typage est dit faible car les conversions de type peuvent être implicites. Les booléens (de type `logical`) sont convertis en `double` valant `1` pour `true` ou `0` pour `false` lorsqu'ils sont utilisés comme opérandes d'opérations numériques. Par exemple, le résultat de `12.7 + true` est `13.7`. Inversement, un `double` non nul est converti implicitement en `true` et un `double` nul est converti en `false` lorsqu'il est opérande d'une opération entre booléens. Par exemple `2.3 & true` donne `true`.

8 Portée des variables, passage par valeur

Les variables initialisées et modifiées à l'intérieur des fonctions sont *locales* à ces fonctions, sauf en utilisant des variables déclarées explicitement comme globale (mot-clé `global`) ou statiques (mot-clé `persistent`), ce que nous ne ferons pas. Ainsi, les fonctions n'intéragissent avec l'extérieur que par le biais des valeurs de leurs arguments et de leurs valeurs de retour.

Les arguments des fonctions sont passés *par valeur*, ce qui veut dire que les fonctions travaillent sur des copies de leurs arguments et ne peuvent donc pas les modifier.

9 Structures de contrôle

Pour un branchement conditionnel :

```
if cond1
  expr11;
  expr12;
  %...
elseif cond2
  expr21;
  %...
elseif cond3
  expr31;
  %...
%...
else
  exprN;
end
```

Les conditions `cond1`, `cond2`, ... sont des expressions scalaires de type `logical`. Les blocs `elseif` et `else` sont facultatifs.

Pour une boucle « pour » :

```
for compteur = debut:pas:fin
  expr1;
  expr2;
  %...
end
```

Le compteur est de type `double` et sa valeur lors du premier passage dans la boucle sera `debut` (qui peut être non entier). Lors du deuxième passage il sera incrémenté de la valeur (éventuellement non entière et/ou négative) du `pas`, etc, jusqu'à la dernière valeur possible dans l'intervalle fermé [`debut`;`fin`] (si le pas est positif) ou [`fin`;`debut`] (si le pas est négatif).

Lorsque le pas vaut 1, il n'est pas nécessaire de le préciser, et la première ligne devient

```
for compteur = debut:fin
```

On utilise une boucle « pour » lorsque le nombre d'itérations est connu à l'avance. On ne peut pas (et c'est une excellente chose) modifier la valeur de `compteur` dans le corps de la boucle.

Pour une boucle conditionnelle « tant que » :

```
while cond
  expr1;
  expr2;
  %...
end
```

Le corps de la boucle est exécuté tant que l'expression scalaire de type logical `cond` est évaluée à `true`.

L'instruction `break` permet de sortir d'une boucle `for` ou `while`. L'instruction `continue` permet de passer directement à l'itération suivante, en ignorant le reste du corps de la boucle. Bien entendu, toutes ces structures de contrôle peuvent être imbriquées. On indentera systématiquement le corps d'une structure de contrôle pour rendre le code plus clair.

En Octave, on peut, à la place de `end` utiliser `endfunction`, `endif`, `endfor`, `endwhile`. C'est plus bavard mais plus explicite. Par compatibilité avec Matlab, on préfère `end` tout court. Si on en resent le besoin pour rendre le code plus clair (multitude de structures imbriquées), on préfèrera utiliser un commentaire (par exemple `end %while`).

10 Tableaux

10.1 Scalaires, vecteurs, matrices

En Matlab, toutes les variables sont des tableaux à au moins 2 dimensions. Soient m et n deux entiers naturels non nuls. On appelle

— *scalaire* un tableau de dimensions 1×1 ;

— *vecteur ligne* un tableau de dimensions $1 \times n$;

— *vecteur colonne* un tableau de dimensions $m \times 1$;

— *matrice* de m lignes et n colonnes un tableau de dimensions $m \times n$.

Dans tous les cas, le couple (m, n) s'appelle *le profil* du tableau, m s'appelle emphl'étendue du tableau dans la première dimension et n son étendue dans la deuxième dimension. On dit que deux tableaux sont *conformants* lorsqu'ils ont le même profil. Un vecteur (tout court) est un vecteur ligne (profil $(1, n)$) ou un vecteur colonne (profil $(m, 1)$).

Pour entrer à la main la matrice

$$A = \begin{bmatrix} -3 & 2 & 0 \\ 5 & -0,7 & 3 \end{bmatrix}$$

on entre `A = [-3 2 0 ; 5 -.7 3 ]` : les éléments d'une même ligne sont séparés par des espaces (ou des virgules) et une ligne est séparée de la suivante par un point-virgule. Attention, chaque ligne doit avoir le même nombre d'éléménts. Les tableaux sont dynamiques : les dimensions peuvent changer au court d'un même programme.

— `x = A(2,1)` : lecture de l'élément de `A` situé à la deuxième ligne et la première colonne. Attention ! En Matlab, les indices des tableaux commencent à 1. Si au moins l'un des indices est plus grand que l'étendue correspondante, Matlab lance une erreur.

— `size(A)` : retourne le profil de la matrice A sous la forme d'un vecteur ligne (ici, `[2 3]` ;

— `size(A, 1)` : retourne l'étendue de la `A` dans la dimension 1 (ici, `2`), autrement dit, son nombre de lignes.

— `size(A, 2)` : retourne l'étendue de la `A` dans la dimension 2 (ici, `2`), autrement dit, son nombre de colonnes.

— `class(A)` : retourne la classe (nécessairement commune) des éléments de la matrice (ici `double`) ;

— `numel(A)` : retourne le nombre d'éléments de `A` ;

— `A(1, end)` : l'indice automatique `end` est toujours le dernier de la dimension correspondante ;

— `A(1,2) = 36` : écriture de l'élément situé à la première ligne et la deuxième colonne. Attention ! En cas de dépassement d'une ou plusieurs dimensions, par exemple `A(3,2) = 12`, Matlab rajoute autant d'éléments à `A` que nécessaire en complétant par des 0.

10.2 Opérateurs sur les tableaux

Opérateurs généraux

— `A'` : matrice transposée (i.e. obtenue en échangeant les lignes et les colonnes) de `A`. Si `A` est à coefficients complexes, `A'` est sa matrice conjuguée (i.e. en plus de la transposition, on conjugue chacun des éléments de `A`).

— `[A B]`, ou `[A, B]` : concaténation horizontale de `A` et `B`. Les deux tableaux doivent avoir le même nombre de lignes.

— `[A ; B]` : concaténation verticale de `A` et `B`. Les deux tableaux doivent avoir le même nombre de colonnes.

Opérateurs sur les matrices de classes `double`. Si les opérandes ne sont pas de type `double`, une conversion implicite en `double` a lieu, si elle est possible.

— `A+B` : retourne la somme élément par élément de `A` et `B`. Il faut que `A` et `B` soient conformants ou bien que l'un des deux tableaux soit un scalaire.

— `-`, `.*`, `./`, `.^` : même chose que ci-dessus avec la soustraction, la multiplication, la division, la puissance, élément par élément.

— `A * B` : produit matriciel de `A` et `B`. Il faut que le nombre de colonnes de `A` soit égal au nombre de lignes de `B`.

— `A ^ n` : puissance matricielle de `A`. Il faut que `A` soit carrée, c'est-à-dire ait autant de lignes que de colonnes et que `n` soit entier.

— `A == B`, `A <= B`, etc. Comparaison, élément par élément. Retourne une matrice de classe `logical`. Les matrices doivent être conformantes ou bien l'un des opérandes doit être scalaire.

— `sin(A)`, `exp(A)`, `...` : les fonctions sont appliquées à chaque élément de la matrice `A`.

— `sum(A)` : si `A` est un vecteur, retourne la somme des éléments de `A` ; si `A` est une matrice qui n'est pas un vecteur, applique la fonction `sum` à chaque colonne et retourne un vecteur ligne composé des résultats.

— `cumsum(A)` : si `A` est un vecteur `[a1 a2 ... an`, retourne un vecteur de même profil contenant les sommes cumulatives `[a1 (a1 + a2)... (a1 + a2 + ... + an)]`. Si `A` n'est pas un vecteur, applique cette opération à chaque colonne et concatène horizontalement les colonnes obtenues.

Opérateurs sur les matrices de classe `logical`. Les matrices sont de classe `logical` ou sont implicitement converties.

— `~A` : retourne la négation, élément par élément de la matrice `A`.

— `A & B`, `A | B` : et logique, ou logique, élément par élément. `A` et `B` doivent être conformants.

— `all(A)` : si `A` est un vecteur de classe logical, retourne `True` si tous les éléments de `A` sont `True`. Si `A` est une matrice de type `logical`, retourne une ligne dont chaque coefficient est le résultat de `all` pour la colonne correspondante dans `A`.

— `any(A)` : même chose que `all` mais renvoit `True` si l'un des éléments du vecteur (resp. de la colonne) est à `True` et `False` sinon.

10.3 Création de tableaux

Toutes les fonctions de cette liste fonctionne de la même manière que `zeros` en ce qui concerne le profil du résultat retourné.

- `zeros(m,n)` ou `zeros([m n])` : matrice de profil (m, n) dont tous les coefficients sont des zéros ;
- `zeros(n)` : matrice carrée de profil (n, n).
- `ones( ... )` : tous les coefficients sont égaux à 1.
- `eye( ... )` : les coefficients de la matrice sont égaux 0 en dehors de la diagonale où ils valent 1.
- `rand( ... )` : les coefficients de la matrices sont des nombres aléatoires uniformes sur $[0\,;1]$.
- `randn( ... )` : les coefficients sont des nombres aléatoires tirés suivant la loi normale centrée réduite.

10.4 Suites arithmétiques

- `debut:pas:fin` : matrice ligne dont le premier terme est `debut` et les termes suivants en progression arithmétique de raison `pas`, tant que l'on ne dépasse pas strictement `fin`.
- `debut:fin` : s'il n'est pas précisé, le pas vaut `1`.
- `linspace(debut, fin, taille)` : matrice ligne de taille `taille`, dont le premier coefficient est `debut`, le dernier est `fin` et les coefficients sont en progression arithmétique.

10.5 Indiçage linéaire

Si `A` est une matrice qui n'est pas un vecteur, `A(k)` est le $k^{\text{ème}}$ élément de `A` lorsque ceux-ci sont énumérés colonne par colonne. Par exemple, si

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

`A(4)` retourne la valeur `4`.

10.6 Extraction de sous-matrices par liste d'indices

La valeur de `A( [i1 i2 ... ip] ; [j1 j2 ... jp] )` est la matrice de dimensions $p \times q$ dont le coefficient d'indice (m, n) vaut `A(im, jn)`, pour tout m entre 1 et p et tout n entre 1 et q. Les crochets sont optionnels si la liste n'est composée que d'un élément. Par exemple, `A ([1 3] , [2 3 6])` est la matrice `A` dans laquelle on aurait supprimé toutes les lignes sauf les lignes `1` et `3` et toutes les colonnes sauf les colonnes `2`, `3` et `6`. Le résultat est donc une matrice de profil $(2, 3)$. On peut également utiliser les suites arithmétiques (l'opérateur `:`) et le mot clé `end`. Dans ce contexte, on peut raccourcir `1:end` en `:`. Par exemple, `A( 1:2:end, :)` correspond à la matrice `A` dans laquelle on aurait enlevé les lignes d'indice pair.

On peut utiliser l'extraction pour modifier les termes de la matrice avec l'affectation. L'opérande de droite de `=` doit alors être soit un tableau de même profil que la matrice extraite, soit un scalaire.

Par exemple,

```
1 >> A = [1 2 3 ; 4 5 6];
2 >> A( 1, [2 3]) = [20 30]
3 A =
```

```
    1  20  30
    4   5   6
>> A(2, :) = 42
A =
     1  20  30
    42  42  42
```

La valeur de `A( [i1, i2, ..., ip] )` est la matrice ligne composée des valeurs `A(i1)`, `A(i2)`, ..., `A(ip)` lorsque `A` est indicée linéairement. Par exemple, `A(1:end)` est la matrice ligne formée de tous les éléments de `A` dans l'ordre des colonnes (d'abord les éléments de la première colonne, puis ceux de la deuxième, etc). Cas particulier : la matrice `A(:)` est toujours un vecteur colonne (avec les mêmes valeurs que `A(1:end)`).

On peut utiliser cette extraction en lecture comme en écriture. Par exemple :

```
>> A = [1 2 3 ; 4 5 6 ; 7 8 9];
>> B = A( [1 5 8] )
B =
    1 5 8
>> A (2:2:end) = 37 % termes d'indices pairs dans l'indicage ↩
   lineaire
A =
     1 37  3
    37  5 37
     7 37  9
```

10.7 Indiçage logique

— Si `B` est une matrice de classe `logical`, `find(B)` retourne la liste des indices linéaire, *dans l'ordre des colonnes*, des éléments de `B` égaux à `True`. Par exemple, `find([true false↩ false ; true true false]` retourne la colonne `[1 ; 2 ; 4]`.

— Si `A` et `B` sont des matrices conformantes, et que `B` est une matrice de classe `logical`, `A (B)` produit le même résultat que `A (find(B))`.

Ceci peut être utilisé en lecture comme en écriture et est très pratique. Par exemple,

```
>> A = [-1 7 3 ; -4 5 7];
>> numel( A( A == 7 ) ) % nombre d'éléments de A egaux a 7
  ans = 2
>> A ( A < 0 ) = 0 % on annule les termes négatifs
  A =
      0 7 3
      0 5 7
```

11 Tracer des graphiques

11.1 Ouvrir et fermer les figures

— `figure(2)` : Fait de la figure 2 la *figure courante*. Ouvre une nouvelle figure numérotée 2 si celle-ci n'existait pas.

— La commande `clf` (pour *clear figure*) efface le contenu de la figure courante.

— La commande `close(2)` ferme (et efface) la figure numéro 2.

— `hold on` : indique que dans la figure courante, les tracés successifs se superposent.

— `hold off` : indique que dans la figure courante chaque tracé sera remplacé par le prochain tracé (comportement par défaut).

11.2 Tracer des courbes

— `plot(X,Y)` : dans la figure courante, relie les points dont les abscisses sont les éléments de `X` et les ordonnées les éléments de `Y`. Beaucoup d'options et d'appels différents existent, utiliser `help("plot")` ou `doc("plot")`.

— `axis( [xmin, xmax, ymin, ymax] )` : change l'échelle de la figure courante.

11.3 Histogrammes

La commande à utiliser est `hist` :

— `hist(X)` : range les valeurs du vecteur `X` en 10 classes, régulièrement réparties et trace l'histogramme des effectifs correspondant dans la figure courante.

— `hist(X, n)`, où `n` est un scalaire entier : même chose, en `n` classe.

— `hist(X, C)`, où `C` est la liste des centres des classes dans lesquelles ranger les valeurs de `X`

— `hist(X,n, 1)`, ... : lorsque `1` est présent en troisième argument, c'est l'histogramme des fréquences qui est tracé.

12 Quelques conseils de programmation

— Donner des noms explicites aux variables et aux fonctions. Par exemple `nblignes` au lieu de `n` si la variable sert effectivement à stocker un nombre de lignes ; `tirer_binomiale` (ou `tirerBinomiale`) plutôt que `X` si la fonction simule la loi binomiale, ...

— Indenter le code dans les fonctions et les structures de contrôle.

— Séparer le code en fonctions courtes. Diviser un gros problème difficile et plusieurs petits problèmes faciles.

— Autant que possible, faire les opérations directement sur les tableaux au lieu d'utiliser les boucles (temps de calcul divisé par environ 100). Par exemple l'exécution du script suivant :

```
tic
somme = 0;
for i = 1:1000000
  somme += i * i;
end
disp(somme);
toc
```

donne sur ma machine :

```
   3.3333e+17
Elapsed time is 3.42495 seconds.
```

Tandis que ce script :

```
tic
somme = sum( (1:1000000).^2 );
disp(somme);
toc
```

qui sert à calculer la même somme, donne, sur la même machine,

```
   3.3333e+17
Elapsed time is 0.0211289 seconds.
```

— Autant que possible préallouer les tableaux (c'est-à-dire les initialiser directement à la taille voulue) plutôt que d'utiliser les concaténations qui sont très coûteuses en temps de calcul. Par exemple, voici un script qui calcule sous forme de tableau les valeurs successives de la récurrence croisée

$$\begin{aligned} u_{n+1} &= \sqrt{u_n v_n}, \\ v_{n+1} &= (u_n + v_n)/2, \end{aligned}$$

avec comme premiers termes $u_1 = 1$ et $v_1 = 10$.

```
tic
u = 1;
v = 10;
termes = [u ; v];
fin = 100000;
for i = 2:fin
  tmp = u;
  u = sqrt(u * v);
  v = (tmp + v)/2 ;
  termes = [termes [u ; v]]; % on change la taille
end
toc
```

Le temps d'éxécution est :

```
Elapsed time is 16.8376 seconds.
```

Avec juste une préallocation la différence est importante :

```
tic
u = 1;
v = 10;
fin = 100000;
termes = zeros(2, fin); % preallocation
termes(:,1) = [u ; v];
for i = 2:fin
  tmp = u;
  u = sqrt(u * v);
  v = (tmp + v)/2 ;
  termes(:, i) = [u ; v]; % la taille ne change pas
end
```

```
13 toc
```

```
1 Elapsed time is 3.22134 seconds.
```

— Expérimenter dans la fenêtre de commande. Vérifier que la fonction, prédéfinie ou non, fait bien ce que vous attendez.

— Aller voir la documentation, directement dans le logiciel ou sur internet.

13 Pour aller plus loin

— Manipulation du path (`path`, `addpath`, ...) et fichiers de configuration (`.octaverc`, `.matlabrc`).

— Lecture et l'écriture de fichiers : `fopen`, `fclose`, `fwrite`, `fprintf`, ...

— Structures de données : tableaux inhomogènes (`cell`), structures (`struct`).

— Classes d'entiers signés et non signés (`int32`, `uint32`, ...). Représentation des entiers positif et des entiers signés en machine (complément à 2).

— Algèbre linéaire : opérateur `\` pour résoudre les systèmes linéaires, très nombreuses fonctions d'algèbre linéaire (décompositions LU, QR, de Cholesky, calcul de valeurs et vecteurs propres, ...). Problèmes de conditionnement des matrices, de stabilité numérique, ...

— Interface avec C, C++ ou Fortran.

— Programmation orientée objet.