1.1 Introduction

1.1.1 Qu'est-ce que MATLAB et GNU Octave ?

MATLAB

- MATLAB est un logiciel commercial de **calcul numérique/scientifique**, de **visualisation** et de **programmation** très performant et convivial développé par la société **The MathWorks Inc**. Attention: ce n'est cependant **pas** un logicel de calcul algébrique ou symbolique (pour cela, voir les logiciels commerciaux Mathematica ou Maple, ou le logiciel libre Maxima).

- Le nom de MATLAB vient de *MAT*rix *LAB*oratory, les éléments de données de base manipulés par MATLAB étant des **matrices** (pouvant bien évidemment se réduire à des vecteurs et des scalaires) qui ne nécessitent ni dimensionnement ni déclaration de type. Contrairement aux langages de programmation classiques (scalaires et à compiler), les **opérateurs** et **fonctions** MATLAB permettent de manipuler directement et interactivement ces données matricielles, rendant ainsi MATLAB particulièrement efficace en calcul numérique, analyse et visualisation de données en particulier.

- Mais MATLAB est aussi un **environnement de développement** ("progiciel") à part entière : son **langage** d'assez haut niveau, doté notamment de structures de contrôles, fonctions d'entrée-sortie et de visualisation 2D et 3D, outils de construction d'interface utilisateur graphique (GUI)... permet à l'utilisateur d'élaborer ses propres **fonctions** ainsi que de véritables programmes ("**M-files**") appelés **scripts** vu le caractère interprété de ce langage.

MATLAB est disponible sur tous les systèmes d'exploitation standards (Windows, GNU/Linux, MacOS X...). Le champ d'utilisation de MATLAB peut être étendu aux **systèmes non linéaires** et aux problèmes associés de simulation avec le produit complémentaire **SIMULINK**. Les capacités de MATLAB peuvent en outre être enrichies par des fonctions spécialisées regroupées au sein de dizaines de "**toolboxes**" (boîtes à outils qui sont des collections de "M-files") couvrant des domaines très variés tels que :

- analyse de données
- statistiques
- mathématiques symboliques (accès au noyau Maple V)
- analyse numérique (accès aux routines NAG)
- traitement d'image, cartographie
- traitement de signaux (et du son en particulier)
- acquisition de données et contrôle de processus (gestion ports série/parallèle, cartes d'acquisition, réseau TCP ou UDP), instrumentation
- logique floue
- finance
- etc...

Une interface de programmation applicative (**API**) rend finalement possible l'interaction entre MATLAB et les environnements de développement classiques (exécution de routines C ou Fortran depuis MATLAB, ou accès aux fonctions MATLAB depuis des programmes C ou Fortran). MATLAB permet en outre de déployer de véritables **applications** à l'aide des outils de conversion optionnels suivants :

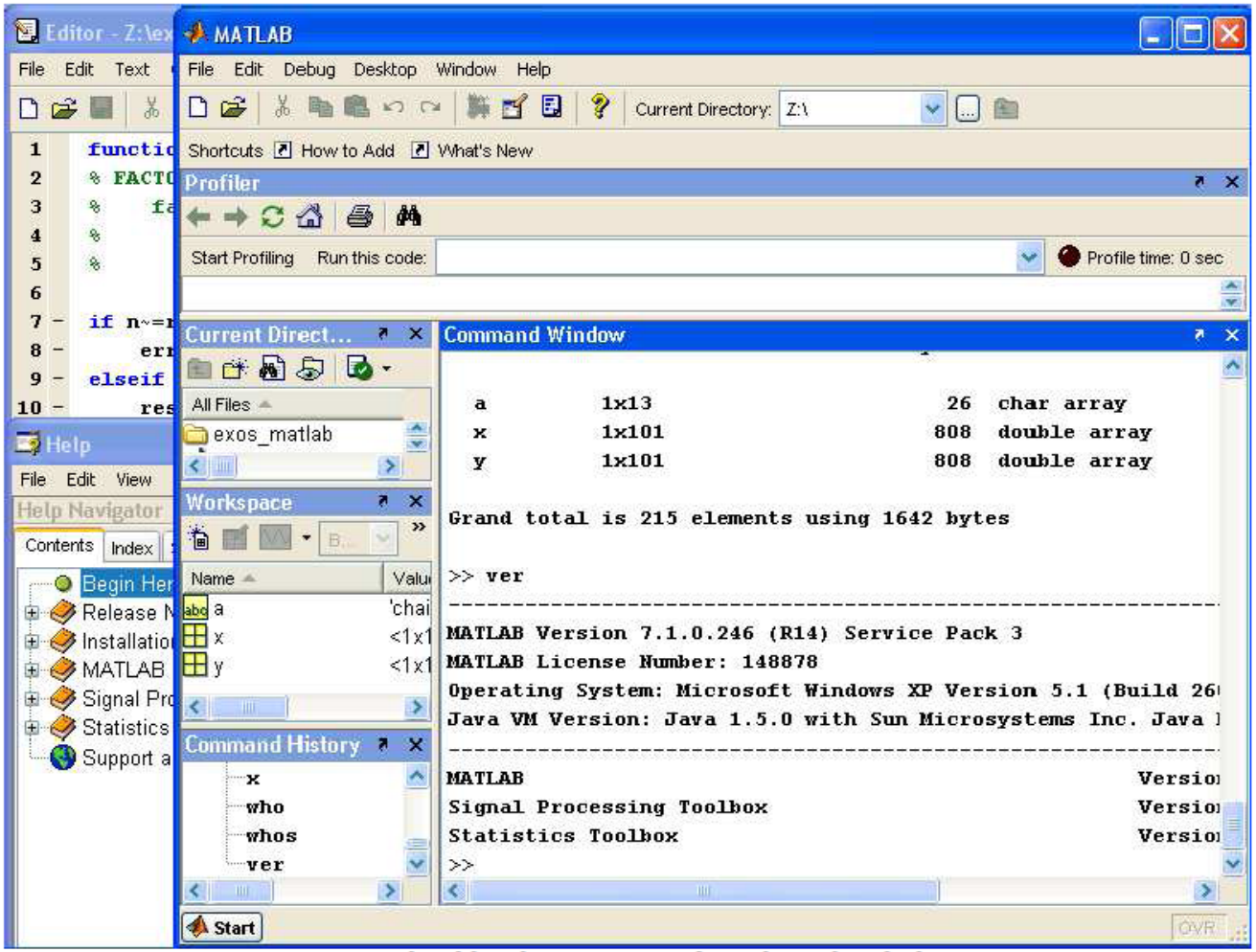
- MATLAB -> code **C/C++**, avec le MATLAB Compiler
- MATLAB -> **Excel add-ins**, avec le MATLAB Excel Builder
- MATLAB -> objets **COM** Windows, avec le MATLAB COM Builder
- etc...

Toutes ces caractéristiques font aujourd'hui de MATLAB un standard incontournable en milieu académique, dans les différents domaines de l'ingénieur et la recherche scientifique.

1.3.1 Interface graphique et environnement de développement (IDE)

MATLAB offre en standard un **IDE** (Integrated Development Environment), c'est-à-dire qu'il présente une interface graphique (GUI) se composant, en plus de la fenêtre de Commande et des fenêtres de Graphiques, de divers autres outils sous forme de fenêtres graphiques :

- MATLAB 5.3 : Workspace Browser, Editor/Debugger, Path Browser, Graphic Property Editor, Guide Control Panel, Help Desk...
- MATLAB 7 : Workspace, Editor, Current Directory, Command History, Profiler, Help... (voir figure ci-dessous)



Environnement de développement (IDE) intégré de MATLAB 7

1.4 Outils d'aide et d'information, références internet utiles

1.4.1 Aide en ligne

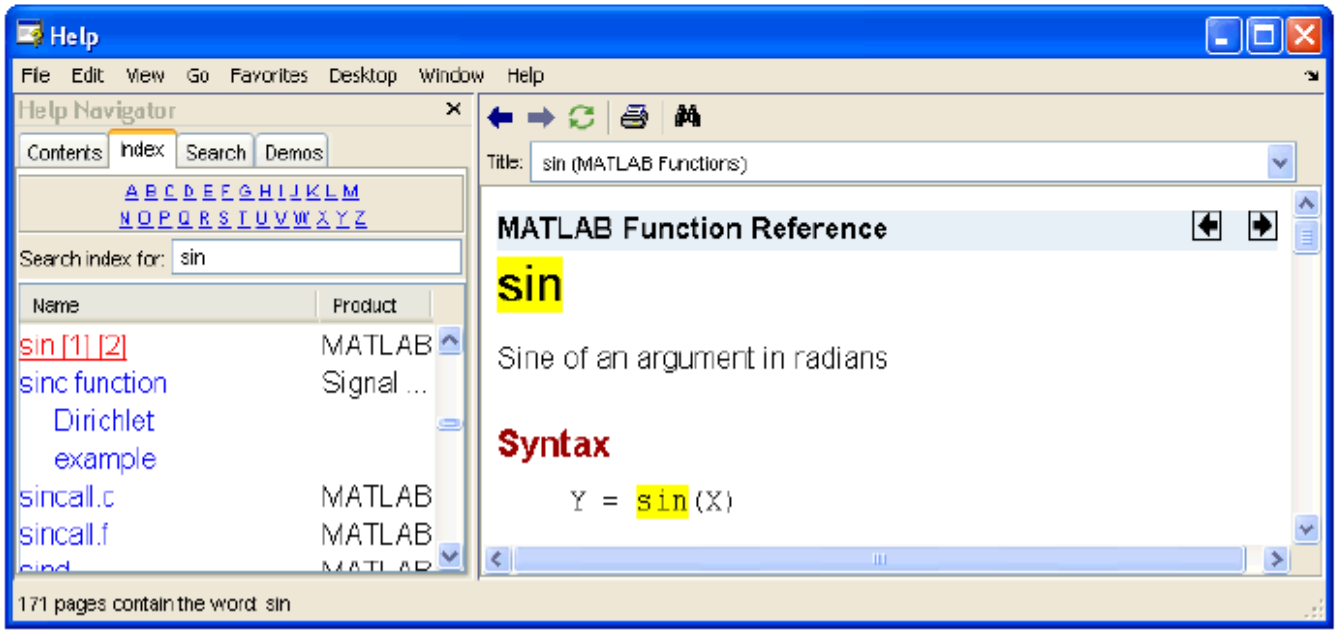
`help` *`fonction`*

Affiche, dans la fenêtre de commande MATLAB/Octave, la **description** et la **syntaxe** de la *fonction* MATLAB/Octave spécifiée. Le mode de défilement, continu (c'est le défaut dans MATLAB) ou "paginé" (défaut dans Octave), peut être modifié avec la commande **`more on|off`** (voir plus bas)

Passée sans paramètres, la commande **`help`** liste les rubriques d'aide principales (correspondant à la structure de répertoires définie par le **`path`**)

`helpwin` *`fonction`*, ou **`doc`** *`fonction`*, ou menu **`Help>MATLAB Help`**, ou icône **`[?]`** de la Toolbar MATLAB

Même effet que la commande **`help`**, sauf que le résultat est affiché dans la fenêtre d'aide spécifique MATLAB "Help" (voir illustration ci-dessous)



Fenêtre d'aide MATLAB 7

doc *fonction* (remplace l'ancien **help -i** *fonction* de Octave 2)
Sous Octave, cette commande recherche et affiche (avec l'outil Info) l'information relative à la *fonction* spécifiée à partir du **manuel** Octave

lookfor {-all} *mot-clé*
Recherche par *mot-clé* dans l'aide MATLAB/Octave. Cette commande retourne la liste de toutes les fonctions dont le *mot-clé* spécifié figure dans la première ligne (H1-line) de l'aide.
Sous Octave, l'affichage paginé peut donner l'impression que rien se se passe si l'on ne patiente pas. Le cas échéant, désactiver l'affichage paginé avant de passer cette commande. A partir de la version 3.2.0, la vitesse d'exécution de cette commande a été améliorée par un mécanisme de caching des textes d'aide
Avec l'option **-all**, la recherche du *mot-clé* spécifié s'effectue dans l'entier des textes d'aide et pas seulement dans leurs 1ères lignes (H1-lines); prend donc passablement plus de temps et retourne davantage de références (pas forcément en relation avec ce que l'on cherche...)

Ex: **help inverse** retourne dans MATLAB l'erreur comme quoi aucune fonction "inverse" n'existe ; par contre **lookfor inverse** présente la liste de toutes les fonctions MATLAB/Octave en relation avec le thème de l'inversion (notamment la fonction **inv** d'inversion de matrices)

Manuel Octave en-ligne complet, ou via
Démarrer > Programmes > GNU Octave x.x > Documentation puis dans les sous-menu **HTML** ou **PDF**
- Accès au **manuel Octave** sous forme hyper-texte (HTML). Voyez en particulier, tout au bas de la table des matières, le "Function Index" qui est un index hyper-texte de toutes les fonctions Octave
- Voyez aussi cet **Octave Quick Reference Card** (aide-mémoire en 3 pages, PDF)

1.4.2 Exemples et démos

intro sous MATLAB 5.3
echodemo intro sous MATLAB 7
Lancement d'un petit didacticiel d'**introduction** à MATLAB

rundemos(*repertoire***)**
Lance les démos définies dans le répertoire spécifié.
Implémenté depuis Octave 3.2.0

demos, ou **helpwin demos**, ou menu **Help>Demos**
Passe dans l'onglet "Demos" de la fenêtre "Help" (voir illustration ci-dessous) où l'on trouve quantité de **démonstrations interactives** illustrant les capacités de MALAB (voir aussi **help demos** qui donne la liste et la description de toutes ces démos). Pour chacune de ces démos le code MATLAB détaillé est présenté.

1.5 Types de nombres (réels/complexes, entiers), variables et fonctions

1.5.1 Types réels, double et simple précision

De façon interne (c'est-à-dire en mémoire=>workspace, et sur disque=>MAT-files), MATLAB/Octave stocke par défaut tous les nombres en virgule flottante "**double précision**" (au format IEEE qui occupe 8 octets par nombre). Les nombres ont donc une **précision finie** de 16 chiffres décimaux significatifs, et une **étendue** allant de 10^{-308} à 10^{+308}. Cela permet donc de manipuler, en particulier, des coordonnées géographiques.

Les **nombres réels** seront saisis par l'utilisateur selon les conventions de notation décimale standard (si nécessaire en notation scientifique avec affichage de la puissance de 10)

Ex de nombres réels valides : `3` , `-99` , `0.000145` , `-1.6341e20` , `4.521e-5`

Il est cependant possible, depuis la version 3.2.0 d'Octave, de définir comme sous MATLAB des réels en virgule flottante "**simple précision**", donc stockés sur des variables occupant 2x moins d'espace en mémoire (4 octets), mais de précision 2x moindre (7 chiffres décimaux significatifs, et une étendue allant de 10^{-38} à 10^{+38}). On utilise pour cela la fonction `single( nombre | variable )` , ou en ajoutant le paramètre `'single'` à certaines fonctions telles que `ones` , `zeros` , `eye` ... De façon inverse, la fonction `double(variable)` retourne, sur la base d'une *variable* simple précision, un résultat double précision. **ATTENTION** cependant : lorsque l'on utilise des opérateurs ou fonctions mélangeant des opérandes/paramètres de types simple et double précision, le résultat retourné sera toujours de type simple précision. Vous pouvez vérifier tout cela avec la commande `whos` .

Ex • l'expression `3 * ones(2,2)` retourne une matrice double précision

• mais les expressions `single(3) * ones(2,2)` ou `3 * ones(2,2,'single')` ou `single(3 * ones(2,2))` retournent toutes une matrice simple précision

1.5.2 Types entiers, 64/32/16/8 bits

On vient de voir que MATLAB/Octave manipule **par défaut** les nombres sous forme **réelle en virgule flottante** (double précision ou, sur demande, simple précision). Ainsi l'expression `nombre = 123` stocke de façon interne le nombre spécifié sous forme de variable réelle double précision, bien que l'on ait saisi un nombre entier.

Il est cependant possible de manipuler des variables de **types entiers**, respectivement :

- 8 bits : nombre stocké sur 1 octet ; si signé, étendue de -128 (-2^7) à 127
- 16 bits : nombre stocké sur 2 octets ; si signé, étendue de -32'768 (-2^15) à 32'767
- 32 bits : nombre stocké sur 4 octets ; si signé, étendue de -2'147'483'648 (-2^31) à 2'147'483'647 (9 chiffres)
- 64 bits : nombre stocké sur 8 octets ; si signé, étendue de -9'223'372'036'854'775'808 (-2^63) à 9'223'372'036'854'775'807 (18 chiffres)

Les opérations arithmétiques sur des entiers sont **plus rapides** que les opérations analogues réelles.

On dispose, pour cela, des possibilités suivantes (int64 complétement supporté sous Octave à partir de la version 3.2.0) :

- les fonctions `int8`, `int16`, `int32` et `int64` : génèrent des variables entières **signées** stockées respectivement sur 8 bits, 16 bits, 32 bits ou 64 bits ; les valeurs réelles (double ou simple précision) sont arrondies au nombre le plus proche (équivalent de `round`)
- les fonctions `uint8`, `uint16`, `uint32` et `uint64` : génèrent des variables entières **non signées** (unsigned) stockées respectivement sur 8 bits, 16 bits, 32 bits ou 64 bits
- en ajoutant l'un des paramètres `'int8'`, `'uint8'`, `'int16'`, `'uint16'`, `'int32'`, `'uint32'`, `'int64'` ou `'uint64'` à certaines fonctions telles que `ones`, `zeros`, `eye` ...
- les valeurs réelles (double ou simple précision) sont **arrondies** au nombre le plus proche (équivalent de `round`)

IMPORTANT : Lorsque l'on utilise des opérateurs ou fonctions mélangeant des opérandes/paramètres de types entier et réels (double ou simple précision), le résultat retourné sera toujours de **type entier** ! Si l'on ne souhaite pas ça, il faut convertir au préalable l'opérande entier en réel double précision (avec `double(entier)`) ou simple précision (avec `single(entier)`) !

M Sous MATLAB, certaines opérations mixant des données de type réel avec des données de type entier 64 bits ne sont pas autorisées. Ainsi l'expression `13.3 * int64(12)` génère une erreur.

1.5.3 Nombres complexes

MATLAB/Octave est aussi capable de manipuler des **nombres complexes** (stockés de façon interne sous forme de **réels double precision**, mais sur 2x 8 octets pour la partie réelle resp. la partie imaginaire)

Ex de nombres complexes valides (avec partie réelle et imaginaire) : `4e-13 - 5.6i` , `-45+5*j`

Le tableau ci-dessous présente quelques fonctions MATLAB/Octave relatives aux nombres complexes.

Fonction	Description
`real(nb_complexe)` `imag(nb_complexe)`	Retourne la **partie réelle** du *nb_complexe* spécifié, respectivement sa **partie imaginaire** Ex: `real(3+4i)` retourne 3, et `imag(3+4i)` retourne 4
`conj(nb_complexe)`	Retourne le **conjugué** du *nb_complexe* spécifié Ex: `conj(3+4i)` retourne 3-4i
`abs(nb_complexe)`	Retourne le **module** du *nb_complexe* spécifié Ex: `abs(3+4i)` retourne 5
O `arg(nb_complexe)`	Retourne l'**argument** du *nb_complexe* spécifié Ex: O `arg(3+4i)` retourne 0.92730
`isreal(var)` , `iscomplex(var)`	Permet de tester si l'argument (sclaire, tableau) contient des nombres réels ou complexes

1.5.4 Généralités sur les variables

Les variables créées au cours d'une session (interactivement depuis la fenêtre de commande MATLAB/Octave ou par des M-files) résident en mémoire dans ce que l'on appelle le "**workspace**" (espace de travail, voir chapitre "**Workspace**"). Le langage MATLAB ne requiert **aucune déclaration** préalable de **type** de variable et de **dimension** de tableau/vecteur. Lorsque MATLAB/Octave rencontre un nouveau nom de variable, il crée automatiquement la variable correspondante et y associe l'espace de stockage approprié dans le workspace. Si la variable existe déjà, MATLAB/Octave change son contenu et, si nécessaire, lui alloue un nouvel espace de stockage en cas de redimensionnement de tableau. Les variables sont définies à l'aide d'**expressions**.

Un **nom de variable** valide consiste en une lettre suivie de lettres, chiffres ou caractères souligné "_". Les lettres doivent être dans l'intervalle a-z et A-Z, donc les caractères accentués ne sont pas autorisés. MATLAB (mais pas Octave) n'autorise cependant pas les noms de variable dépassant **63** caractères (voir la fonction `namelengthmax`).

Ex de noms de variables valides : `x_min` , `COEFF55a` , `tres_long_nom_de_variable`

Ex de noms non valides : `86ab` (commence par un chiffre), `coeff-555` (est considéré comme une expression), `temp_mesurée` (contient un caractère accentué)

Les noms de variable sont **case-sensitive** (distinction des majuscules et minuscules).

Ex: `MAT_A` désigne une matrice différente de `mat_A`

Pour désigner un **ensemble de variables** (principalement avec commandes `who` , `clear` , `save` ...), on peut utiliser les **caractères de substitition** `*` (remplace 0, 1 ou plusieurs caractères quelconques) et `?` (remplace 1 caractère quelconque).

Ex: si l'on a défini les variables `x=14 ; ax=56 ; abx=542 ;` , alors :

`who *x` liste toutes les variables `x` , `ax` et `abx`

`clear ?x` n'efface que la variables `ax`

Une "**expression**" MATLAB/Octave est une construction valide faisant usage de nombres, de variables, d'opérateurs et de fonctions.

Ex: `pi*r^2` et `sqrt((b^2)-(4*a*c))` sont des expressions

Nous décrivons ci-dessous les comandes de base relatives à la gestion des variables. Pour davantage de détails sur la gestion du workspace et les commandes y relatives, voir le chapitre "**Workspace**".

variable = expression

Affecte à *variable* le résultat de l' *expression*, et affiche celui-ci

Ex: **r = 4** , **surface=pi*r^2**

variable = expression ;

Affecte à *variable* le résultat de l'*expression*, mais effectue cela "silencieusement" (en raison du **;**) c'est-à-dire sans affichage du résultat à l'écran

expression

Si l'on n'affecte pas une expression à une variable, le résultat de l'évaluation de l'*expression* est affecté à la variable de nom prédéfini **ans** ("answer")

Ex: **pi*4^2** retourne la valeur 50.2655... sur la variable **ans**

variable

Affiche le contenu de la *variable* spécifiée

who {*variable(s)*}

Liste le nom de toutes les variables couramment définies dans le workspace (ou seulement la(les) *variable(s)* spécifiées)

whos {*variable(s)*}

Affiche une liste plus détaillée que **who** de toutes les variables couramment définies dans le workspace (ou seulement la(les) *variable(s)* spécifiées) : nom de la variable, dimension, espace mémoire, classe.

variable **= who{s}** ...

La sortie des commandes **who** et **whos** peut elle-même être affectée à une *variable* de type tableau cellulaire (utile en programmation !)

clear {*variable(s)*}

Détruit du workspace toutes les variables (ou la/les *variable(s)* spécifiées, séparées par des espaces et non pas des virgules !)

Ex: **clear mat*** détruit toutes les variables dont le nom commence par "mat"

1.5.5 Généralités sur les chaînes de caractères

Il est également possible de manipuler du **texte** (des "**chaînes**" de caractères) dans MATLAB/Octave. De façon interne, MATLAB stocke chacun des caractères sur 2 octets, et Octave sur 1 octet. La chaîne elle-même est vue comme un vecteur-ligne contenant autant d'éléments que de caractères.

Pour toutes les fonctions MATLAB/Octave manipulant des chaînes, on peut leur passer celles-ci soit de façon littérale en les délimitant par des **apostrophes** (par exemple `'Hello world'`), soit via des **variables**.

`string = 'chaîne de caractères'`

Enregistre la *chaîne de caractères* (définie entre apostrophes) sur la variable *string* qui est un vecteur-ligne. Si la chaîne contient un apostrophe, il faut le dédoubler (sinon il serait interprété comme signe de fin de chaîne... et la suite de la chaîne provoquerait une erreur)

Ex: `section = 'Sciences et ingénierie de l''environnement'`

`string(i:j)`

Retourne la partie de la chaîne *string* comprise entre le *i*-ème et le *j*-ème caractère

Ex: suite à l'exemple ci-dessus, `section(13:22)` retourne la chaîne "ingénierie"

Pour davantage de détails, voir plus loin le chapitre dédié aux "**Chaînes de caractères**".

1.5.6 Généralités sur les fonctions

Comme en ce qui concerne les noms de variables, les noms de fonctions sont "case-sensitive" (distinction des majuscules et minuscules). Les noms de toutes les **fonctions** prédéfinies MATLAB/Octave sont en **minuscules**.

Ex: `sin()` est la fonction sinus, tandis que `SIN()` n'est pas définie !

Les fonctions MATLAB/Octave sont implémentées soit au niveau du noyau MATLAB/Octave (fonctions "built-ins") soit au niveau de M-files et packages (dont on pourrait voir et même changer le code).

Ex: `which sin` indique que `sin` est une fonction built-in, alors que `which axis` montre dans quel M-file est implémentée la fonction `axis`.

Attention : les noms de fonction ne sont **pas réservés** et il serait donc possible de les écraser !

Ex: si l'on définissait `sin(1)=444`, l'affectation `val=sin(1)` retournerait alors 444 ! Pour restaurer la fonction originale, il faudra dans ce cas passer la commande `clear sin`, et la fonction `sin(1)` retournera alors à nouveau le sinus de 1 radian (qui est 0.8415).

M `helpwin elfun | specfun | elmat`

Affiche respectivement la liste des fonctions mathématiques *élé*mentaires, avancées (*spé*cialisées), *mat*ricielles

Pour une présentation détaillée des principales fonctions MATLAB/Octave, voir les chapitres dédiés plus loin ("**Fonctions de base**", "**Fonctions matricielles**").

L'utilisateur a la possibilité de créer ses propres fonctions (voir chapitre "**Fonctions**"),

1.6.2 Caractères spéciaux dans les commandes MATLAB et Octave

La commande M `helpwin punct` décrit l'ensemble des caractères spéciaux MATLAB. Parmi ceux-ci, les caractères ci-dessous sont particulièrement importants.

Caractère	Description
`;`	• Suivie de ce caractère, une commande sera normalement exécutée (sitôt le `<enter>` frappé), mais son résultat ne sera pas affiché. Caractère faisant par la même occasion office de séparateur de commandes lorsque l'on saisit plusieurs commandes sur la même ligne • Utilisé aussi comme caractère de séparation des lignes d'une matrice lors de la définition de ses éléments
`,`	• Caractère utilisé comme séparateur de commande lorsque l'on souhaite passer plusieurs commandes sur la même ligne • Utilisé aussi pour délimiter les indices de ligne et de colonne d'une matrice • Utilisé également pour séparer les différents paramètres d'entrée et de sortie d'une fonction Ex: `a=4 , b=5` affecte les variables a et b et affiche le résultat de ces affectations ; tandis que `a=4 ; b=5` affecte aussi ces variable mais n'affiche que le résultat de l'affectation de b. `A(3,4)` désigne l'élément de la matrice A situé à la 3e ligne et 4e colonne
`...` ("ellipsis") `\`	• Utilisé en fin de ligne lorsque l'on veut continuer une instruction sur la ligne suivante (sinon la frappe de `<enter>` exécute l'instruction)
`:` ("colon")	• Opérateur de définition de séries (voir chapitre "**Séries**") et de plage d'indices de vecteurs et matrices Ex: `5:10` définit la série "5 6 7 8 9 10"
`%` ou O `#`	• Ce qui suit est considéré comme un commentaire (non évalué par MATLAB/Octave). Utile pour documenter un script ou une fonction (M-file) • Lorsqu'il est utilisé dans une chaîne, le caractère `%` débute une définition de format (voir chapitre "**entrées-sorties**") Ex: commentaire : `r=5.5 % rayon en [cm]` ; format `sprintf('Rabais %2u%%', 25)`

`%{`
plusieurs
lignes
de code...
`%}`
	• Dans un M-file, les séquences `%{` et `%}` délimitent un commentaire s'étendant sur plusieurs ligne (possible sous Octave depuis la version 3.2). Notez bien qu'il ne doit rien y avoir d'autre dans les 2 lignes contenant ces séquences `%{` et `%}` (ni avant ni après)

`'` (apostrophe)	• Caractère utilisé pour délimiter le début et la fin d'une chaîne de caractère • Également utilisé comme opérateur de transposition de matrice

Les séparateurs **`<espace>`** et **`<tab>`** ne sont en principe pas significatifs dans une expression (MATLAB/Octave travaille en "format libre"). Vous pouvez donc en mettre 0, 1 ou plusieurs, et les utiliser ainsi pour mettre en page ("indenter") le code de vos M-files.

Ex: **`b=5*a`** est équivalent à **`b = 5 * a`**

Pour nous-autres, utilisateurs d'ordinateurs avec clavier "Suisse-Français", rappelons que l'on forme ainsi les caractères suivants qui sont importants sous MATLAB (si vous n'avez pas de touche **`<AltGr>`** vous pouvez utiliser à la place la combinaison **`<ctrl-alt>`**) :

- pour `[` frapper **`<AltGr-è>`**
- pour `]` frapper **`<AltGr-!>`**
- pour `\` frapper **`<AltGr-<>`**
- pour `~` frapper **`<AltGr-^>`** suivi de **`<espace>`**

3.1 Scalaires et constantes

MATLAB/Octave ne différencie fondamentalement pas une matrice d'un vecteur ou d'un scalaire, et ces éléments peuvent être redimensionnés dynamiquement. Une variable **scalaire** n'est donc, en fait, qu'une variable matricielle "dégénérée" de 1x1 élément (vous pouvez le vérifier avec `size(variable_scalaire)`).

Ex de définition de scalaires : `a=12.34e-12` , `w=2^3` , `r=sqrt(a)*5` , `s=pi*r^2` , `z=-5+4i`

MATLAB/Octave offre un certain nombre de **constantes** utiles. Celles-ci sont implémentées par des "built-in functions" . Le tableau ci-dessous énumère les constantes les plus importantes.

Constante	Description
`pi`	3.14159265358979 (la valeur de "pi")
`i` ou `j`	racine de -1 (`sqrt(-1)`) (nombre imaginaire)
`e` ou `exp(1)`	2.71828182845905 (la valeur de "e")
`Inf` ou `inf`	infini (par exemple le résultat du calcul `5/0`)
`NaN` ou `nan`	indéterminé (par exemple le résultat du calcul `0/0`)
`NA`	valeur manquante
`realmin`	env. $2.2e^{-308}$: le plus petit nombre positif utilisable (en virgule flottante double

`realmax`	env. $1.7e^{+308}$: le plus grand nombre positif utilisable (en virgule flottante double précision)
`eps`	env. $2.2e^{-16}$: c'est la précision relative en virgule flottante double précision (ou le plus petit nombre représentable par l'ordinateur qui est tel que, additionné à un nombre, il crée un nombre juste supérieur)
`true`	vrai ou `1` ; mais n'importe quelle valeur différente de `0` est aussi "vrai" Ex: si `nb` vaut `0`, la séquence `if nb, disp('vrai'), else, disp('faux'), end` retourne "faux", mais si `nb` vaut n'importe quelle autre valeur, elle retourne "vrai"
`false`	faux ou `0`

MATLAB/Octave manipule en outre des **variables spéciales** de nom prédéfini. Les plus utiles sont décrites dans le tableau ci-dessous.

Variable	Description
`ans`	variable sur laquelle MATLAB retourne la valeur d'une expression qui n'a pas été affectée à une variable (ou nom de variable par défaut pour les résultats)
`nargin`	nombre d'arguments passés à une fonction
`nargout`	nombre d'arguments retournés par une fonction

3.2 Opérateurs de base

La commande `helpwin ops` décrit l'ensemble des opérateurs et caractères spéciaux sous MATLAB

3.2.1 Opérateurs arithmétiques de base

Les **opérateurs arithmétiques** de base sous MATLAB/Octave sont les suivants (voir le chapitre "**Opérateurs matriciels**" pour leur usage dans un contexte matriciel) :

Opérateur ou fonction	Description	Précédence
`+` ou fonction `plus`	Addition	4
{var2=} `++` *var1* {var2=}*var1* `++`	Pré- ou post-incrémentation (seulement sous Octave) : • pré-incrémentation: incrémente d'abord *var1* de 1, **puis** affecte le résultat à *var2* (ou l'affiche, si *var2* n'est pas spécifiée et que l'instruction n'est pas suivie de `;`) ; donc équivalent à `var1=var1+1; var2=var1;` • post-incrémentation: affecte d'abord *var1* à *var2* (ou affiche la valeur de *var1* si *var2* n'est pas spécifiée et que l'instruction n'est pas suivie de `;`), **puis** incrémente *var1* de 1 ; donc équivalent à `var2=var1; var1=var1+1;`	

`-` ou fonction `minus`	Soustraction	4
O {var2=} `--` *var1* O {var2=}*var1* `--`	Pré- ou post-décrémentation (seulement sous Octave) : • pré-décrémentation: décrémente d'abord *var1* de 1, **puis** affecte le résultat à *var2* (ou l'affiche, si *var2* n'est pas spécifiée et que l'instruction n'est pas suivie de `;`) ; donc équivalent à `var1=var1-1; var2=var1;` • post-décrémentation: affecte d'abord *var1* à *var2* (ou affiche la valeur de *var1* si *var2* n'est pas spécifiée et que l'instruction n'est pas suivie de `;`), **puis** décrémente *var1* de 1 ; donc équivalent à `var2=var1; var1=var1-1;` Si *var1* est un vecteur ou une matrice, agit sur tous ses éléments.	
`*` ou fonction `mtimes`	Multiplication	3
`/` ou fonction `mrdivide`	Division	3
`\` ou fonction `mldivide`	Division à gauche Ex: `14/7` est équivalent à `7\14`	3
`^` ou fonction `mpower` ou O `**`	Puissance Ex: `4^2` => 16, `64^(1/3)` => 4 (racine cubique)	2
`( )`	Parenthèses (pour changer ordre de précédence)	1

Les expressions sont évaluées de gauche à droite avec l'**ordre de précédence** habituel : puissance, puis multiplication et division, puis addition et soustraction. On peut utiliser des parenthèses `( )` pour modifier cet ordre (auquel cas l'évaluation s'effectue en commençant par les parenthèses intérieures).
Ex: `a-b^2*c` est équivalent à `a-((b^2)*c)` ; mais `8 / 2*4` retourne 16, alors que `8 /(2*4)` retourne 1

L'usage des fonctions plutôt que des opérateurs s'effectue de la façon suivante :
Ex: à la place de `4 - 5^2` on pourrait par exemple écrire `minus(4,mpower(5,2))`

3.2.2 Opérateurs relationnels

Les **opérateurs relationnels** permettent de faire des **tests numériques** en construisant des "*expressions logiques*", c'est-à-dire des expressions retournant les valeurs vrai ou faux. Rappel: dans MATLAB/Octave, la valeur faux est `false` ou `0`, et la valeur vrai est `true` ou `1` voire n'importe quelle valeur différente de `0`.

Les opérateurs de test ci-dessous "pourraîent" être appliqués à des **chaînes de caractères**, mais pour autant que la taille des 2 chaînes (membre de gauche et membre de droite) soit identique ! Cela retourne alors un vecteur logique (avec autant de 0 ou de 1 que de caractères dans ces chaînes). Pour tester l'égalité exacte de chaînes de longueur quelconque, on utilisera plutôt les fonctions `strcmp` ou `isequal` (voir chapitre "**Chaînes de caractères**").

Les opérateurs relationnels MATLAB/Octave sont les suivants (voir `helpwin relop`) :

Opérateur ou fonction	Description
`==` ou fonction `eq`	Test d'égalité
`~=` ou `!=` ou fonction `ne`	Test de différence
`<` ou fonction `lt`	Test d'infériorité
`>` ou fonction `gt`	Test de supériorité
`<=` ou fonction `le`	Test d'infériorité ou égalité
`>=` ou fonction `ge`	Test de supériorité ou égalité

Ex:

- si l'on a `a=3, b=4, c=3`, l'expression `a==b` ou la fonction `eq(a,b)` retournent alors "0" (faux), et `a==c` ou > `eq(a,c)` retournent "1" (vrai)
- si l'on définit le vecteur `A=1:5`, l'expression `A>3` retourne alors le vecteur [0 0 0 1 1]
- le test `'abc'=='axc'` retourne le vecteur [1 0 1] ; mais le test `'abc'=='vwxyz'` retourne une erreur (chaînes de tailles différentes)

3.2.3 Opérateurs logiques

Les **opérateurs logiques** ont pour arguments des *expressions logiques* et retournent les valeurs logiques vrai (`1`) ou faux (`0`). Les opérateurs logiques principaux sont les suivants (voir `helpwin relop`) :

Opérateur ou fonction	Description
`~` *expression* `not` (*expression*) `!` *expression*	**Négation** logique (rappel: NON 0 => 1 ; NON 1 => 0)
expression1 `&` *expression2* `and` (*expression1*, *expression2*)	**ET** logique. Si les *expressions* sont des matrices, retourne une matrice (rappel: 0 ET 0 => 0 ; 0 ET 1 => 0 ; 1 ET 1 => 1)
expression1 `|` *expression2* `or` (*expression1*, *expression2*)	**OU** logique. Si les *expressions* sont des matrices, retourne une matrice (rappel: 0 OU 0 => 0 ; 0 OU 1 => 1 ; 1 OU 1 => 1)
`xor` (*expression1*, *expression2*)	**OU EXCLUSIF** logique (rappel: 0 OU EXCL 0 => 0 ; 0 OU EXCL 1 => 1 ; 1 OU EXCL 1 => 0)
expression1 `&&` *expression2*	**ET** logique, mais retourne dans ce cas un scalaire même si les *expressions* sont des matrices
expression1 `||` *expression2*	**OU** logique, mais retourne dans ce cas un scalaire même si les *expressions* sont des matrices
Pour des opérandes binaires, voir les fonctions `bitand` , `bitcmp` , `bitor` , `bitxor` ...	

Ex : si `A=[0 0 1 1]` et `B=[0 1 0 1]` , alors :

- `A | B` ou `or(A,B)` retourne le vecteur [0 1 1 1] , et `A || B` retourne le scalaire 0
- `A & B` ou `and(A,B)` retourne le vecteur [0 0 0 1] , et `A && B` retourne le scalaire 0

3.3 Fonctions de base

3.3.1 Fonctions mathématiques

Utilisées sur des **vecteurs ou matrices**, les fonctions ci-dessous seront appliquées à tous les éléments et retourneront donc des vecteurs ou matrices.

Pour les fonctions **trigonométriques**, les angles sont exprimés en [radians].
Rappel : on passe des [gons] aux [radians] avec les formules : *radians* = 2*pi**gons*/400, et *gons* = 400**radians*/2/pi.

Les principales **fonctions mathématiques** disponibles sous MATLAB/Octave sont les suivantes :

Fonction	Description
`sqrt(var)`	Racine carrée de *var*. Remarque : pour la racine *n*-ème de *var*, faire `var^(1/n)`
`exp(var)`	Exponentielle de *var*
`log(var)` et `log10(var)`	Logarithme naturel, resp. base 10, de *var*
`cos(var)` et `acos(var)`	Cosinus, resp. arc cosinus, de *var*. Angle exprimé en radian
`sin(var)` et `asin(var)`	Sinus, resp. arc sinus, de *var*. Angle exprimé en radian

sec(*var*) et **csc**(*var*)	Sécante, resp. cosécante, de *var*. Angle exprimé en radian
tan(*var*) et **atan**(*var*)	Tangente, resp. arc tangente, de *var*. Angle exprimé en radian
cot(*var*) et **acot**(*var*)	Cotangente, resp. arc cotangente, de *var*. Angle exprimé en radian
atan2(*dy*,*dx*)	Angle entre -pi et +pi correspondant à *dx* et *dy*
cart2pol(*x*,*y* {,*z*}) et **pol2cart**(*th*,*r* {,*z*})	Passage de coordonnées carthésiennes en coordonnées polaires, et vice-versa
cosh, **acosh**, **sinh**, **asinh**, **sech**, **asch**, **tanh**, **atanh**, **coth**, **acoth**	Fonctions hyperboliques...
factorial(*n*)	Factorielle de *n* (c'est-à-dire : n*(n-1)*(n-2)*...*1). La réponse retournée est exacte jusqu'à la factorielle de 20 (au-delà, elle est calculée en virgule flottante double précision, c'est-à-dire à une précision de 15 chiffres avec un exposant)
rand **rand**(*n*) **rand**(*n*,*m*)	Génère un nombre aléatoire compris entre 0.0 et 1.0 Génère une matrice carrée *n*x*n* de nb. aléatoires compris entre 0.0 et 1.0 Génère une matrice *n*x*m* de nb. aléatoires compris entre 0.0 et 1.0

`fix(var)`
`round(var)`
`floor(var)`
`ceil(var)`
	Troncature à l'entier, dans la direction de zéro (donc 4 pour 4.7, et -4 pour -4.7)
	Arrondi à l'entier le plus proche de *var*
	Le plus grand entier qui est inférieur ou égal à *var*
	Le plus petit entier plus grand ou égal à *var*
	
	Ex:
	`fix(3.7)` et `fix(3.3)` => 3, `fix(-3.7)` et `fix(-3.3)` => -3
	`round(3.7)` => 4, `round(3.3)` => 3, `round(-3.7)` => -4, `round(-3.3)` => -3
	`floor(3.7)` et `floor(3.3)` => 3, `floor(-3.7)` et `floor(-3.3)` => -4
	`ceil(3.7)` et `ceil(3.3)` => 4, `ceil(-3.7)` et `ceil(-3.3)` => -3

`mod(var1,var2)` `rem(var1,var2)`	Fonction *var1* "modulo" *var2* Reste ("remainder") de la division de *var1* par *var2* Remarques: - *var1* et *var2* doivent être des scalaires réels ou des tableaux réels de même dimension - `rem` a le même signe que *var1*, alors que `mod` a le même signe que *var2* - les 2 fonctions retournent le même résultat si *var1* et *var2* ont le même signe Ex: `mod(3.7, 1)` et `rem(3.7, 1)` retournent 0.7, mais `mod(-3.7, 1)` retourne 0.3, et `rem(-3.7, 1)` retourne -0.7
`idivide(var1, var2, 'regle')`	Division entière. Fonction permettant de définir soi-même la *règle* d'arrondi. Implémentée depuis Octave 3.2.0
`abs(var)`	Valeur absolue (positive) de *var* Ex: `abs([3.1 -2.4])` retourne [3.1 2.4]

Voir `helpwin elfun` où sont notamment encore décrites : autres fonctions trigonométriques (sécante, cosécante, cotangente), fonctions hyperboliques, manipulation de nombres complexes...

Voir encore `helpwin specfun` pour une liste des fonctions mathématiques spécialisées (Bessel, Beta, Jacobi, Gamma, Legendre...).

3.3.2 Fonctions de changement de type de nombres

Au chapitre "**Généralités sur les nombres**" on a décrit les différents types relatifs aux nombres : réels virgule flottante (double ou simple précision), et entiers (64, 32, 16 ou 8 bits). On décrit ci-dessous les fonctions permettant de passer d'un type à l'autre :

Fonction	Description
var2 = `single(var1)` var4 = `double(var3)`	Retourne, dans le cas où *var1* est une variable réelle double précision ou entière, une variable *var2* en **simple précision** Retourne, dans le cas où *var3* est une variable réelle simple précision ou entière, une variable *var4* en **double précision** Fonctions implémentées, sous Octave, depuis la version 3.2.0
`int8`, `int16`, `int32` et `int64` ou `uint8`, `uint16`, `uint32` et `uint64`	Fonctions retournant des variables de type **entiers signés**, respectivement stockés sur 8 bits, 16 bits, 32 bits ou 64 bits ou des **entiers non signés** (unsigned) stockés sur 8 bits, 16 bits, 32 bits ou 64 bits Fonctions implémentées, sous Octave, depuis la version 3.2.0

3.3.3 Fonctions logiques

Les **fonctions logiques** servent à réaliser des tests. Comme les opérateurs relationnels et logiques (voir plus haut), elles retournent en général les valeurs vrai (`true` ou `1`) ou faux (`false` ou `0`). Il en existe un très grand nombre dont en voici quelque-unes :

Fonction	Description
`isfloat(var)`	Vrai si la variable *var* est de type réelle (simple ou double précision), faux sinon (entière, chaîne...) Sous Octave, implémenté depuis la version 3.2.0
`isepmty(var)`	Vrai si la variable *var* est vide (de dimension 1x0), faux sinon. Notez bien qu'il ne faut ici **pas** entourer *var* d'apostrophes, contrairement à la fonction `exist` . Ex: si `vect=5:1` ou `vect=[]` , alors `isempty(vect)` retourne "1"
`ischar(var)`	Vrai si *var* est une chaîne de caractères, faux sinon. Ne plus utiliser `isstr` qui va disparaître.
`exist ('objet' {,'var|builtin|file|dir'})`	Vérifie si l'*objet* spécifié existe. Retourne "1" si c'est une *variable*, "2" si c'est un M-*file*, "3" si c'est un MEX-*file*, "4" si c'est un MDL-*file*, "5" si c'est une fonction *builtin*, "6" si c'est un P-*file*, "7" si c'est un *dir*ectoire. Retourne "0" si aucun objet de l'un de ces types n'existe. Notez bien qu'il **faut** ici entourer *objet* d'apostrophes, contrairement à la fonction `isempty` ! Ex: `exist('sqrt')` retourne "5", `exist('axis')` retourne "2", `exist('variable_inexistante')` retourne "0"

4. Objets : séries/vecteurs, matrices, chaînes, tableaux multidimensionnels et cellulaires, structures

4.1 Séries (ranges)

L'opérateur MATLAB/Octave `:` (deux points, en anglais "colon") est très important. Outre l'adressage des éléments d'un tableau, il permet de construire des **séries linéaires** sous la forme de vecteurs ligne. On peut utiliser à cet effet soit l'opérateur `:` soit la fonction équivalente `colon` :

`début:fin` ou `colon(début,fin)`

Crée une série numérique **linéaire** débutant par la valeur *début*, autoincrémentée de "1" et se terminant par la valeur *fin*. Il s'agit donc d'un **vecteur ligne** de dimension 1xM où M=*fin*-*début*+1. Si *fin*<*début*, crée une série vide (vecteur de dimension 1x0)

Ex :

- `1:5` crée le vecteur ans=[1 2 3 4 5]
- `x=1.7:4.6` crée le vecteur x=[1.7 2.7 3.7]

`début:pas:fin` ou `colon(début,pas,fin)`

Crée une série numérique **linéaire** (vecteur ligne) débutant par la valeur *début*, incrémentée ou décrémentée du *pas* spécifié et se terminant par la valeur *fin*. Crée une série vide (vecteur de dimension 1x0) si *fin*<*début* et que le *pas* est positif, ou si *fin*>*début* et que le *pas* est négatif

Ex :

- `-4:-2:-11.7` retourne le vecteur ans=[-4 -6 -8 -10]
- `x=0:0.5:2*pi` crée x=[0.0 0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0 5.5 6.0]

Lorsqu'on connait la valeur de *début*, la valeur de *fin* et que l'on souhaite générer des séries **linéaires** ou **logarithmique** de *nbval* valeurs, on peut utiliser les fonctions suivantes :

série = `linspace(début,fin {,nbval})`

Crée une *série* (vecteur ligne) de *nbval* éléments **linéairement** espacés de la valeur *début* jusqu'à la valeur *fin*. Si l'on omet le paramètre *nbval*, c'est une série de **100** éléments qui est créée

Ex : `v=linspace(0,-5,11)` crée v=[0.0 -0.5 -1.0 -1.5 -2.0 -2.5 -3.0 -3.5 -4.0 -4.5 -5.0]

série = `logspace(début,fin {,nbval})`

Crée une *série* **logarithmique** (vecteur ligne) de *nbval* éléments, débutant par la valeur $10^{début}$ et se terminant par la valeur 10^{fin}. Si l'on omet le paramètre *nbval*, c'est une série de **50** éléments qui est créée
Ex: `x=logspace(2,6,5)` crée x=[100 1000 10000 100000 1000000]

Sous Octave depuis la version 3, définir une série avec la syntaxe `début:pas:fin` (plutôt qu'avec `linspace`) est particulièrement **intéressant au niveau utilisation mémoire** ! En effet, quelle que soit la taille de la série qui en découle, celle-ci n'occupera en mémoire que 24 octets (c'est-à-dire l'espace de stockage nécessaire pour stocker en double précision les 3 valeurs définissant la série) !

Ex: `s1=0:10/99:10;` et `s1=linspace(0,10,100);` sont fonctionellement identiques, mais :
- sous MATLAB 7.x : les variables `s1` et `s2` consomment toutes deux 800 octets (100 réels double précision)
- alors que sous Octave 3.x : `s2` consomme aussi 800 octets, mais `s1` ne consomme que 24 octets !!!
  noter cependant que, selon son usage, cette série est susceptible d'occuper aussi 800 octets (p.ex. `s1'` ou `s1*3`)

Pour construire des séries d'un autre type (géométrique, etc...), il faudra réaliser des boucles `for` ou `while` ... (voir chapitre "**Structures de contrôle**").

4.2 Vecteurs (ligne ou colonne)

Comme on l'a dit en ce qui concerne les variables scalaires, MATLAB/Octave ne fait pas vraiment de différence entre une matrice, un vecteur et un scalaire, étant donné que ces éléments peuvent être redimensionnés dynamiquement. Une variable de type **vecteur** n'est donc, en quelque sorte, qu'une matrice NxM dégénérée d'une seule ligne (1xM) ou une seule colonne (Nx1).

ATTENTION: les **éléments** du vecteurs sont numérotés par des entiers **débutant** par **la valeur 1** (et non pas 0, comme dans d'autres langages de programmation).

Syntaxe	Description
vec= `[val1 val2 val3 ...]` = `[val var expr ...]`	Création d'un **vecteur ligne** *vec* contenant les valeurs *val*, variables *var*, ou expressions *expr* spécifiées. Celles-ci doivent être délimitées par des `<espace>`, `<tab>` ou `,` (virgules). Ex: `v1=[1 -4 5]`, `v2=[-3,sqrt(4)]` et `v3=[v2 v1 -3]` retournent v3=[-3 2 1 -4 5 -3]

vec= `[val ; var ; expr ...]`
= `[val1`
`val2`
`... ]`
= `[var val var val ...]'`
	Création d'un **vecteur colonne** *vec* contenant les valeurs *val* (ou variables *var*, ou expressions *expr*) spécifiées.
	Celles-ci **doivent** être délimitées par des `;` (point-virgules) (1ère forme ci-contre) et/ou par la touche `<enter>` (2e forme).
	La 3ème forme ci-contre consiste à définir un vecteur ligne et à le transposer avant de l'affecter à *vec*.
	
	Ex:
	• `v4=[-3;5;2*pi]`, `v5=[11 ; v4]`, `v6=[3 4 5 6]'` sont des vecteurs colonne valides
	• mais `v7=[v4 ; v1]` provoque une erreur car on combine ici un vecteur colonne avec un vecteur ligne

`vec'`	**Transposée** du vecteur *vec*. Si *vec* était un vecteur ligne, il devient un vecteur colonne (ou vice-versa)
vec= `début{:pas}:fin` ou =M `colon(début{,pas},fin)`	Initialisation d'un vecteur ligne *vec* à une **série linéaire** (voir chapitre sur les "Séries" ci-dessus)
vec= `linspace(début,fin{,n})` *var*= `logspace(début,fin{,n})`	Initialisation d'un vecteur ligne *vec* à une série **linéaire**, respectivement **logarithmique** (voir chapitre sur les "Séries" ci-dessus)
`vec(i)`	Désigne le *i*-ème élément du vecteur ligne ou colonne *vec* Ex: `v3(2)` retourne la valeur "2", et `v4(2)` retourne "5"
`vec(i{:p}:j)`	Adressage des éléments d'indice *i* à *j* du vecteur ligne ou colonne *vec* avec un pas de "1" ou de "*p*" si spécifié (construction faisant usage des "séries" présentées ci-dessus !). Si *vec* est un vecteur ligne, le résultat retourné sera un vecteur ligne ; respectivement si *vec* est un vecteur colonne, le résultat retourné sera un vecteur colonne. Pour l'indice *j* on peut utiliser la valeur `end` qui désigne le **dernier élément** du vecteur. Ex: • `v4(2:end)` retourne un vecteur colonne contenant la 2e jusqu'à la dernière valeur de v4, c'est-à-dire dans le cas présent [5;6.28] • `v3(2:2:6)` retourne un vecteur ligne contenant la 2e, 4e et 6e valeur de v3, c'est-à-dire [2 -4 -3]

`vec([i j k:l])`	La notation d'indices entre crochets `[]` permet de désigner un ensemble continu ou discontinu d'éléments. Dans le cas ci-contre, on désigne les éléments *i*, *j* et *k* à *l*.
`vec(i { {:p} :j} )=val`	Soit le vecteur *vec* (ligne ou colonne) de *n* éléments : l'instruction ci-contre, si *i*>*n* et *j*>*n*, étend la **taille** du vecteur à *i* ou *j* éléments en affectant aux *i*-ème à *j*-ème éléments la valeur *val* spécifiée, et aux autres nouveaux éléments créés la valeur "0" Ex: si `v8=[1 2 3]`, alors `v8(6:2:10)=44` étend v8 qui devient [1 2 3 0 0 44 0 44 0 44]

`for k=i{:p}:j`
`  vec(k)=expression`
`end`
	Initialise les éléments (spécifiés par la série *i*{:*p*}:*j*) du vecteur ligne *vec* par l'*expression* spécifiée
	
	Ex: `for i=2:2:6, v9(i)=i^2, end` crée le vecteur v9=[0 4 0 16 0 36] (les éléments d'indice 1, 3 et 5 n'étant pas définis, ils sont automatiquement initialisés à 0)

`vec(i:j)=[]` ou `vec([k l m])=[]` ou combinaison de ces 2 notations...	**Destruction** des éléments *i* à *j* du vecteur *vec* (qui est redimensionné en conséquence), respectivement destruction des *k*-ème *l*-ème et *m*-ème éléments Ex: soit `v10=(11:20)` • l'instruction `v10(4:end)=[]` (ou `v10(4:length(v10))=[]`) redéfini v10 à [11 12 13] • alors que `v10([1 3:7 10])=[]` redéfini v10 à [12 18 19]
`length(vec)`	Retourne la **taille** (nombre d'éléments) du vecteur ligne ou colonne *vec*

4.3 Matrices

Pour MATLAB/Octave, une **matrice** est un tableau rectangulaire à 2 dimensions de NxM éléments (N lignes et M colonnes) de types nombres réels ou complexes ou de caractères. La présentation ci-dessous des techniques d'**affectation** de matrices (usage des crochets `[ ]`) et d'**adressage** de ses éléments (usage des parenthèses `( )`) est donc simplement une généralisation à 2 dimensions de ce qui a été vu pour les vecteurs à 1 dimension (chapitre précédent). Il faut simplement savoir en outre que, pour adresser un élément d'une matrice, il faut spécifier son **numéro de ligne et de colonne** séparés par une "`,`" (virgule).

ATTENTION: comme pour les vecteurs, les indices de ligne et de colonne sont des valeurs entières débutant par 1 (et non pas 0 comme dans d'autres langages).

On dispose en outre de fonctions d'initialisation spéciales liées aux matrices.

Syntaxe	Description
mat= `[`*v11 v12 ... v1m ; v21 v22 ... v2m ; ... ... ; vn1 vn2 ... vnm* `]`	Définit une **matrice** *mat* de *n* lignes x *m* colonnes dont les éléments sont initialisés aux valeurs *vij*. Notez bien que les éléments d'une ligne sont séparés par des `<espace>`, `<tab>` ou `,` (virgules), et que les différentes lignes sont délimitées par des `;` (point-virgules) et/ou par la touche `<Enter>`. Il faut qu'il y aie exactement le même nombre de valeurs dans chaque ligne, sinon l'affectation échoue. Ex: `m1=[-2:0 ; 4 sqrt(9) 3]` définit la matrice de 2 lignes x 3 colonnes avant pour valeurs [-2 -1 0 ; 4 3 3]

mat= `[vco1 vco2 ...]` ou *mat*= `[vli1 ; vli2 ; ...]`	Construit la matrice *mat* par **concaténation de vecteurs** colonne *vcoi* ou de vecteurs ligne *vlii* spécifiés. Notez bien que les séparateurs entre les vecteurs colonne est l'`<espace>`, et celui entre les vecteurs ligne est le `;` ! L'affectation échoue si tous les vecteurs spécifiés n'ont pas la même dimension. **Ex**: si `v1=1:3:7` et `v2=9:-1:7`, alors `m2=[v2;v1]` retourne la matrice [9 8 7 ; 1 4 7]

`[mat1 mat2 {mat3...}]` ou `horzcat(mat1, mat2 {,mat3...})`
 	
respectivement:
`[mat4; mat5 {; mat6...}]` ou `vertcat(mat1, mat2 {,mat3...})`
	Concaténation de matrices (ou vecteurs). Dans le premier cas, on concatène côte à côte (**horizontalement**) les matrices *mat1*, *mat2*, *mat3*... Dans le second, on concatène **verticalement** les matrices *mat4*, *mat5*, *mat6*... Attention aux dimensions qui doivent être cohérentes : dans le premier cas toutes les matrices doivent avoir le même nombre de lignes, et dans le second cas le même nombre de colonnes.
	
	Ex: ajout devant la matrice m2 ci-dessus de la colonne `v3=[44;55]` : avec `m2=[v3 m2]` ou avec `m2=horzcat(v3,m2)`, ce qui donne m2=[44 9 8 7 ; 55 1 4 7]

`ones(n{,m})`	Renvoie une matrice de *n* lignes x *m* colonnes dont **tous les éléments** sont égaux à "1". Si *m* est omis, crée une matrice carrée de dimension *n* **Ex**: `c * ones(n,m)` renvoie une matrice *n* x *m* dont tous les éléments sont égaux à *c*
`zeros(n{,m})`	Renvoie une matrice de *n* lignes x *m* colonnes dont **tous les éléments** sont égaux à "0". Si *m* est omis, crée une matrice carrée de dimension *n*

`eye(n{,m})`	Renvoie une matrice **identité** de *n* lignes x *m* colonnes dont les éléments de la diagonale principale sont égaux à "1" et les autres éléments sont égaux à "0". Si *m* est omis, crée une matrice carrée de dimension *n*

`diag(vec)`
 	
`diag(mat)`
	Appliquée à un vecteur *vec* ligne ou colonne, cette fonction retourne une **matrice carrée** dont la **diagonale** principale porte les éléments du **vecteur** *vec* et les autres éléments sont égaux à "0"
	
	Appliquée à une matrice *mat* (qui peut ne pas être carrée), cette fonction retourne un **vecteur-colonne** formé à partir des éléments de la diagonale de cette matrice

`mat(:)=val` `mat([i j k:l],:)=val`	Si la matrice *mat* existe : - réinitialise tous les éléments de *mat* à la valeur *val* - réinitialise tous les éléments de la *i*-ème, *j*-ème et *k*-ème à *l*-ème lignes à la valeur *val*
mat2= `repmat(mat1,M,N)`	Renvoie une matrice *mat2* formée à partir de la matrice *mat1* **dupliquée en "tuile"** *M* fois verticalement et *N* fois horizontalement Ex: `repmat(eye(2),1,2)` retourne [1 0 1 0 ; 0 1 0 1]
`mat=[]`	Crée une matrice **vide** *mat* de dimension 0x0
[*n m*]= `size(var)` *taille*= `size(var,dimension)` *n*= `rows(mat_2d)` *m*= `columns(mat_2d)`	La première forme renvoie, sur un vecteur ligne, la **taille** (nombre *n* de **lignes** et nombre *m* de **colonnes**) de la matrice ou du vecteur *var*. La seconde forme renvoie la *taille* de *var* correspondant à la *dimension* spécifiée (*dimension*= `1` => nombre de lignes, `2` => nombre de colonnes).

	Les fonctions `rows` et `columns` retournent respectivelement le nombre *n* de **lignes** et nombre *m* de **colonnes**. Ex: `mat2=eye(size(mat1))` définit une matrice identité "mat2" de même dimension que la matrice "mat1"
`length(mat)`	Appliquée à une matrice, cette fonction analyse le nombre de **lignes** et le nombre de **colonnes** puis retourne **le plus grand** de ces 2 nombres (donc identique à `max(size(mat))`). Cette fonction est par conséquent assez **dangereuse** à utiliser sur une matrice !
`numel(mat)` *(NUMber of ELements)*	Retourne le nombre d'éléments du tableau *mat* (donc identique à `prod(size(mat))` ou `length(mat(:))` , mais un peu plus "lisible")
`mat(i,j)`	Désigne **l'élément** (*i*,*j*) de *mat*, donc retourne un scalaire
`mat(i:j,k:m)`	Désigne la **partie de la matrice** *mat* dont les éléments se trouvent dans les lignes *i* à *j* et dans les colonnes *k* à *m* Notez bien les **formes simplifiées** très courantes de cette notation pour désigner des **lignes** ou **colonnes entières** d'une matrice : • `mat(i,:)` : la ligne *i* • `mat(i:j,:)` : les lignes *i* à *j* • `mat(:,k)` : la colonne *k* • `mat(:,k:m)` : les colonnes *k* à *m*

`mat([lignes],[cols])`	La notation d'indices entre crochets `[]` permet de désigner un **ensemble continu ou discontinu** de lignes et/ou de colonnes **Ex 1**: si l'on a la matrice `m3=[1:4; 5:8; 9:12; 13:16]` - `m3([2 4],1:3)` retourne [5 6 7 ; 13 14 15] - `m3([1 4],[1 4])` retourne [1 4 ; 13 16] **Ex 2**: si l'on a une matrice A 10x10, une matrice B 5x10 et un vecteur-ligne y 1x20 - l'affectation `A([1:3 9],:) = [B(1:3,:) ; y(1:10)]` remplace les lignes 1 à 3 de la matrice A par les 3 premières lignes de B, et la 9ème ligne de A par les 10 premiers éléments de y
`mat(i)` et `mat(i:j)`	Lorsque l'on adresse une matrice à la façon d'un vecteur (en ne précisant qu'un indice *i* ou une série *i*:*j* pour cet indice), la recherche s'effectue en numérotant les éléments de la matrice **colonne après colonne**. La première forme retourne, sur un scalaire, le *i*-ème élément ; la seconde forme retourne, sur un vecteur ligne, les *i*-ème au *j*-ème éléments. **Ex**: `m3(3)` retourne "9", et `m3(7:9)` retourne [10 14 3]
`mat(:)`	Retourne un **vecteur colonne** constitué des colonnes de la matrice (colonne après colonne). **Ex**: si `m4=[1 2;3 4]`, alors `m4(:)` retourne [1 ; 3 ; 2 ; 4]
`mat(i:j,:)=[]` et `mat(:,k:m)=[]`	**Destruction** de **lignes** ou de **colonnes** d'une matrice (et redimensionnement de la matrice en conséquence). La première expression supprime les lignes *i* à *j*, et la seconde supprime les colonnes *k* à *m*. Ce type d'opération ne permet de supprimer que des lignes entières ou des colonnes **entières** **Ex**: en reprenant la matrice m3 ci-dessus, l'instruction `m3([1 3:4],:)=[]` réduit cette matrice à la seconde ligne [5 6 7 8]

4.4 Opérateurs matriciels

4.4.1 Opérateurs arithmétiques sur vecteurs et matrices

La facilité d'utilisation et la puissance de MATLAB/Octave proviennent en particulier de ce qu'il est possible d'exprimer des opérations matricielles de façon très naturelle en utilisant directement les opérateurs arithmétiques de base (déjà présentés au niveau scalaire au chapitre "**Opérateurs de base**"). Nous décrivons ci-dessous l'usage de ces opérateurs dans un contexte matriciel (voir aussi `helpwin arith` et `helpwin slash`, ainsi qu'une petite démonstration interactive des opérations matricielles élémentaires avec `matmanip`).

Opérateur ou fonction	Description
`+` ou fonction `plus(m1,m2,...)` `-` ou fonction `minus`	**Addition** et **soustraction**. Les 2 arguments doivent être des vecteurs ou matrices de même dimension, à moins que l'un des deux ne soit un scalaire auquel cas l'addition/soustraction applique le scalaire sur tous les éléments du vecteur ou de la matrice. Ex: `[2 3 4]-[-1 2 3]` retourne [3 1 1], et `[2 3 4]-1` retourne [1 2 3]
`*` ou fonction `mtimes(m1,m2,...)`	**Produit matriciel**. Le nombre de colonnes de l'argument de gauche doit être égal au nombre de lignes de l'argument de droite, à moins que l'un des deux arguments ne soit un scalaire auquel cas le produit applique le scalaire sur tous les éléments du vecteur ou de la matrice. Ex: • `[1 2]*[3;4]` ou `[1 2]*[3 4]'` produit le scalaire "11" (mais `[1 2]*[3 4]` retourne une erreur!) • `2*[3 4]` ou `[3 4]*2` retournent [6 8]

`.*` ou fonction `times(m1,m2,...)`	**Produit éléments par éléments**. Les 2 arguments doivent être des vecteurs ou matrices de même dimension, à moins que l'un des deux ne soit un scalaire (auquel cas c'est identique à l'opérateur `*`). Ex: si `m1=[1 2;4 6]` et `m2=[3 -1;5 3]` • `m1.*m2` retourne [3 -2 ; 20 18] • `m1*m2` retourne [13 5 ; 42 14] • `m1*2` ou `m1.*2` retournent [2 4 ; 8 12]
`kron`	Produit **tensoriel** de Kronecker
`\` ou fonction `mldivide`	**Division matricielle à gauche** `A\B` est la solution "X" du système linaire "A*X=B". On peut distinguer 2 cas : • Si "A" est une matrice carrée NxN et "B" est un vecteur colonne Nx1, `A\B` est équivalent à `inv(A)*B` et il en résulte un vecteur "X" de dimension Nx1 • S'il y a surdétermination, c'est-à-dire que "A" est une matrice MxN où M>N et B est un vecteur colonne de Mx1, l'opération `A\B` s'effectue alors selon les moindres carrés et il en résulte un vecteur "X" de dimension Nx1
`/` ou fonction `mrdivide`	**Division matricielle** (à droite) `B/A` est la solution "X" du système "X*A=B" (où X et B sont des vecteur ligne et A une matrice). Cette solution est équivalente à `B*inv(A)` ou à `(A'\B')'`

`./` ou fonction `rdivide`	**Division éléments par éléments**. Les 2 arguments doivent être des vecteurs ou matrices de même dimension, à moins que l'un des deux ne soit un scalaire auquel cas la division applique le scalaire sur tous les éléments du vecteur ou de la matrice. Les éléments de l'objet de gauche sont divisés par les éléments de même indice de l'objet de droite
`.\` ou fonction `ldivide`	**Division à gauche éléments par éléments**. Les 2 arguments doivent être des vecteurs ou matrices de même dimension, à moins que l'un des deux ne soit un scalaire. Les éléments de l'objet de droite sont divisés par les éléments de même indice de l'objet de gauche. Ex: `12./(1:3)` et `(1:3).\12` retournent tous les deux le vecteur [12 6 4]
`^` ou fonction `mpower`	Elévation à la **puissance matricielle**. Il faut distinguer les 2 cas suivants (dans lesquels "*M*" doit être une matrice carrée et "*scal*" un scalaire) : • `M^scal` : si *scal* est un entier>1, produit matriciel de *M* par elle-même *scal* fois ; si *scal* est un réel, mise en oeuvre valeurs propres et vecteurs propres • `scal^M` : mise en oeuvre valeurs propres et vecteurs propres
`.^` ou fonction `power` ou `.**`	Elévation à la **puissance éléments par éléments**. Les 2 arguments doivent être des vecteurs ou matrices de même dimension, à moins que l'un des deux ne soit un scalaire. Les éléments de l'objet de gauche sont élevés à la puissance des éléments de même indice de l'objet de droite
`[,]` ou `horzcat`, `[;]` ou `vertcat`, `cat`	**Concaténation** horizontale, respectivement verticale (voir chapitre "Matrices" ci-dessus)
`( )`	Permet de spécifier l'ordre d'évaluation des expressions

4.4.2 Opérateurs relationnels et logiques sur vecteurs et matrices

Les opérateurs relationnels et logiques, qui ont été présentées au chapitre "**Opérateurs de base**", peuvent aussi être utilisées sur des vecteurs et matrices. Elles s'appliquent alors à tous les éléments et retournent donc également des vecteurs ou des matrices.

Ex: si l'on a `a=[1 3 4 5]` et `b=[2 3 1 5]`, alors `c = a==b` ou `c=eq(a,b)` retourment le vecteur c=[0 1 0 1]

4.5 Fonctions matricielles

4.5.1 Fonctions de réorganisation de matrices

Fonction	Description
Opérateur `'` ou fonction `ctranspose`	**Transposition** normale de matrices réelles et transposition **conjuguée** de matrices **complexes**. Si la matrice ne contient pas de valeurs complexes, `'` a le même effet que `.'` Ex: `v=(3:5)'` crée directement le vecteur colonne [3 ; 4 ; 5]
Opérateur `.'` ou fonction `transpose`	**Transposition** non conjuguée de matrices complexes Ex: si l'on a la matrice complexe `m=[1+5i 2+6i ; 3+7i 4+8i]`, la transposition non conjuguée `m.'` fournit [1+5i 3+7i ; 2+6i 4+8i], alors que la transposition conjuguée `m'` fournit [1-5i 3-7i ; 2-6i 4-8i]
`reshape(var,M,N)`	Cette fonction de **redimensionnement** retourne une matrice de *M* lignes x *N* colonnes contenant les éléments de *var* (qui peut être une matrice ou un vecteur). Les éléments de *var* sont lus colonne après colonne, et la matrice retournée est également remplie colonne après colonne. Le nombre d'éléments de *var* doit être égal à *M*x*N*, sinon la fonction retourne une erreur. Ex: `reshape([1 2 3 4 5 6 7 8],2,4)` et `reshape([1 5 ; 2 6 ; 3 7 ; 4 8],2,4)` retournent [1 3 5 7 ; 2 4 6 8]

vec = `mat(:)`	Déverse la **matrice** *mat* colonne après colonne sur le **vecteur-colonne** *vec* **Ex**: si `m=[1 2 ; 3 4]` , alors `m(:)` retourne le vecteur-colonne [1 ; 3 ; 2 ; 4]

`sort(var {,mode})`
 	
`sort(var, d {,mode})`
	Fonction de **tri** par **éléments** (voir aussi la fonction `unique` décrite plus bas). Le *mode* de tri par défaut est `'ascend'` (tri ascendant), à moins que l'on spécifie `'descend'` pour un tri descendant
	
	• appliqué à un **vecteur** (ligne ou colonne), trie dans l'ordre de valeurs croissantes les éléments du vecteur
	• appliqué à une **matrice** *var*, trie les éléments à l'intérieur des **colonnes** (indépendemment les unes des autres)
	• si l'on passe le paramètre *d*=2, trie les éléments à l'intérieur des **lignes** (indépendemment les unes des autres)
	
	Ex: si `m=[7 4 6;5 6 3]` , alors `sort(m)` retourne [5 4 3 ; 7 6 6]
	`sort(m,'descend')` retourne [7 6 6 ; 5 4 3]
	et `sort(m,2)` retourne [4 6 7 ; 3 5 6]

`sortrows(mat {,no_col})`	**Trie les lignes** de la **matrice** *mat* dans l'ordre croissant des valeurs de la première colonne, ou dans l'ordre croissant des valeurs de la colonne *no_col* **Ex**: en reprenant la matrice m de l'exemple précédent : `sortrows(m)` (identique à `sortrows(m,1)`) et `sortrows(m,3)` retournent [5 6 3 ; 7 4 6], alors que `sortrows(m,2)` retourne [7 4 6 ; 5 6 3]

`fliplr(mat)` `flipud(mat)`	Retournement de la matrice *mat* par **symétrie horizontale** (**l**eft/**r**ight), respectivement **verticale** (**u**p/**d**own) Ex: `fliplr([1 2 3 ; 4 5 6])` retourne [3 2 1 ; 6 5 4], et `flipud([1 2 3 ; 4 5 6])` retourne [4 5 6 ; 1 2 3]
`rot90(mat {,K})`	Effectue une rotation de la matrice *mat* de *K* fois 90 degrés dans le sens inverse des aiguilles d'une montre. Si *K* est omis, cela équivaut à K=1 Ex: `rot90([1 2 3 ; 4 5 6])` retourne [3 6 ; 2 5 ; 1 4], et `rot90([1 2 3 ; 4 5 6],-2)` retourne [6 5 4 ; 3 2 1]
`flipdim`, `permute`, `ipermute`, `tril`, `triu`	Autres fonctions de réorganisation de matrices...

4.5.2 Fonctions mathématiques sur vecteurs et matrices

Les fonctions mathématiques présentées au chapitre "**Fonctions de base**" peuvent aussi être utilisées sur des vecteurs et matrices. Elles s'appliquent alors à tous les éléments et retournent donc également des vecteurs ou des matrices.

Ex: si l'on définit la série (vecteur ligne) `x=0:0.1:2*pi`, alors `y=sin(x)` ou directement `y=sin(0:0.1:2*pi)` retournent

un vecteur ligne contenant les valeurs du sinus de "0" à "2*pi" avec un incrément de "0.1"

4.5.3 Fonctions de calcul matriciel et statistiques

On obtient la liste des fonctions matricielles avec `helpwin elmat` et `helpwin matfun`.

Fonction	Description
`norm(vec)`	Calcule la **norme** (longueur) du vecteur *vec*. On peut aussi passer à cette fonction une matrice (voir help)
`dot(vec1,vec2)`	Calcule la **produit scalaire** des 2 vecteurs *vec1* et *vec2* (ligne ou colonne). Equivalent à `vec1 * vec2'` s'il s'agit de vecteurs-ligne, ou à `vec1' * vec2` s'il s'agit de vecteurs-colonne On peut aussi passer à cette fonction des matrices (voir help)
`cross(vec1,vec2)`	Calcule la **produit vectoriel** (en 3D) des 2 vecteurs *vec1* et *vec2* (ligne ou colonne, mais qui doivent avoir 3 éléments !).
`inv(mat)`	**Inversion** de la matrice carrée *mat*. Une erreur est produite si la matrice est singulière (ce qui peut être testé avec la fonction `cond` qui est plus approprié que le test du déterminant)
`det(mat)`	Retourne le **déterminant** de la matrice carrée *mat*
`trace(mat)`	Retourne la **trace** de la matrice *mat*, c'est-à-dire la somme des éléments de sa diagonale principale
`rank(mat)`	Retourne le **rang** de la matrice *mat*, c'est-à-dire le nombre de lignes ou de colonnes linéairement indépendants

`min(var{,d})` et `max(var{,d})`	Appliquées à un vecteur ligne ou colonne, ces fonctions retournent le **plus petit**, resp. le **plus grand** élément du vecteur. Appliquées à une matrice *var*, ces fonctions retournent : • si le paramètre *d* est **omis** ou qu'il vaut **1** : un vecteur **ligne** contenant le plus petit, resp. le plus grand élément de chaque **colonne** de *var* • si le paramètre *d* vaut **2** : un vecteur **colonne** contenant le plus petit, resp. le plus grand élément de chaque **ligne** de *var* • ce paramètre *d* peut être supérieur à 2 dans le cas de "tableaux multidimensionnels" (voir plus bas)
`sum(var{,d})` et `prod(var{,d})`	Appliquée à un vecteur ligne ou colonne, retourne la **somme** ou le **produit** des éléments du vecteur. Appliquée à une matrice *var*, retourne un vecteur ligne (ou colonne suivant la valeur de *d*, voir plus haut sous `min` / `max`) contenant la somme ou le produit des éléments de chaque colonne (resp. lignes) de *var* Ex : `prod([2 3;4 3] {,1})` retourne le vecteur ligne [8 9], `prod([2 3;4 3],2)` retourne le vecteur colonne [6 ; 12] et `prod(prod([2 3;4 3]))` retourne le scalaire 72
`cumsum(var{,d})` et `cumprod(var{,d})`	Réalise la **somme** partielle (**cumulée**) ou le **produit** partiel (**cumulé**) des éléments de *var*. Retourne une variable de même dimension que celle passée en argument (vecteur -> vecteur, matrice -> matrice) Ex : `cumprod(1:10)` retourne les factorielles de 1 à 10, c-à-d. [1 2 6 24 120 720 5040 40320 362880 3628800]
`mean(var{,d})`	Appliquée à un vecteur ligne ou colonne, retourne la **moyenne arithmétique** des éléments du vecteur. Appliquée à une matrice *var*, retourne un vecteur ligne (ou colonne suivant la valeur de *d*, voir plus haut sous `min` / `max`) contenant la moyenne arithmétique des éléments de chaque colonne (resp. lignes) de *var*. Un troisième paramètre, spécifique à Octave, permet de demander le calcul de la moyenne **géométrique** (`'g'`) ou de la moyenne **harmonique** (`'h'`).

std(*var*{,*f*{,*d*}})	Appliquée à un vecteur ligne ou colonne, retourne l'**écart-type** des éléments du vecteur. Appliquée à une matrice *var*, retourne un vecteur ligne (ou colonne suivant la valeur de *d*, voir plus haut sous **min** / **max**) contenant l'écart-type des éléments de chaque colonne (resp. lignes) de *var*. **Attention** : si le flag "*f*" est omis ou qu'il vaut "0", l'écart-type est calculé en normalisant par rapport à "*n*-1" (où *n* est le nombre de valeurs) ; s'il vaut "1" on normalise par rapport à "*n*"
median(*var*{,*d*})	Calcule la **médiane**
cov	Retourne vecteur ou matrice de **covariance**
eig , **eigs** , **svd** , **svds** , **cond** , **condeig** ...	Fonctions en relation avec **vecteurs propres** et **valeurs propres** (voir help)
lu , **chol** , **qr** , **qzhess** , **schur** , **svd** , **housh** , **krylov** ...	Fonctions en relation avec les méthodes de décomposition/factorisation de type : - **LU**, **Cholesky**, **QR**, **Hessenberg**, - **Schur**, valeurs singulières, householder, **Krylov**...

4.5.4 Fonctions matricielles de recherche

Fonction	Description
vec = `find(mat)` [*v1*, *v2* {, *v3* }] = `find(mat)`	Recherche des **indices** des **éléments non-nuls** de la matrice *mat* • Dans la 1ère forme, MATLAB/Octave retourne un vecteur-colonne *vec* d'indices à une dimension en considérant les éléments de la matrice *mat* colonne après colonne • Dans la seconde forme, les vecteurs-colonne *v1* et *v2* contiennent respectivement les numéros de ligne et de colonne des éléments non nuls ; les éléments eux-mêmes sont éventuellement déposés sur le vecteur-colonne *v3* **Remarques importantes** : • À la place de *mat* vous pouvez définir une **expression logique** (voir aussi le chapitre "**Indexation logique**" ci-dessous) ! Ainsi par exemple `find(isnan(mat))` retournera un vecteur-colonne contenant les indices de tous les éléments de *mat* qui sont indéterminés (égaux à NaN). • Le vecteur *vec* résultant permet ensuite d'adresser les éléments concernés de la matrice, pour les récupérer ou les modifier. Ainsi par exemple `mat(find(mat<0))=NaN` remplace tous les éléments de *mat* qui sont inférieurs à 0 par la valeur NaN. Ex: soit la matrice `m=[1 2 ; 0 3]` • `find(m)` retourne [1 ; 3 ; 4] (indices des éléments non-nuls) • `find(m<2)` retourne [1 ; 2] (indices des éléments inférieurs à 2) • `m(find(m<2))=-999` retourne [-999 2 ; -999 3] (remplacement des valeurs inférieures à 2 par -999) • `[v1,v2,v3]=find(m)` retourne indices v1=[1 ; 1 ; 2] v2=[1 ; 2 ; 2], et valeurs v3=[1 ; 2 ; 3]

`unique(mat)`	Retourne un vecteur contenant les éléments de *mat* **triés** dans un ordre croissant et **sans répétitions**. Si *mat* est une matrice ou un vecteur-colonne, retourne un vecteur-colonne ; sinon (si *mat* est un vecteur-ligne), retourne un vecteur-ligne. (Voir aussi les fonctions `sort` et `sortrows` décrites plus haut). *mat* peut aussi être un tableau cellulaire (contenant par exemple des chaînes) Ex: • si `m=[5 3 8 ; 2 9 3 ; 8 9 1]`, la fonction `unique(m)` retourne alors [1 ; 2 ; 3 ; 5 ; 8 ; 9] • si `a={'pomme','poire';'fraise','poire';'pomme','fraise'}`, alors `unique(a)` retourne {'fraise';'poire';'pomme'}

`intersect(var1,var2)`
 	
`setdiff(var1,var2)`
 	
`union(var1,var2)`
	Retourne un vecteur contenant, de façon **triée** et **sans répétitions**, les éléments qui :
	
	• `intersect` : sont **communs** à *var1* et *var2*
	• `setdiff` : existent dans *var1* mais **n'existent pas** dans *var2*
	• `union` : existent dans *var1* **et/ou** dans *var2*
	
	Le vecteur résultant sera de type ligne, à moins que *var1* **et** *var2* soient tous deux de type colonne.
	
	var1 et *var2* peuvent être des tableaux cellulaires (contenant par exemple des chaînes)
	
	O Sous Octave, *var1* et *var2* peuvent être des matrices numériques, alors que MATLAB est limité à des vecteurs numériques
	
	Ex:
	• si `a={'pomme','poire';'fraise','cerise'}` et `b={'fraise','abricot'}`, alors
	- `setdiff(a,b)` retourne {'cerise';'poire';'pomme'}
	- `union(m1,m2)` retourne {'abricot';'cerise';'fraise';'poire';'pomme'}
	• O si `m1=[3 4 ; -1 6 ; 6 3]` et `m2=[6 -1 9]`, alors `intersect(m1,m2)` retourne [-1 6]

`ismember(mat1,mat2)`	Voir plus bas (fonctions matricielles logiques)

4.5.5 Fonctions matricielles logiques

Outre les fonctions logiques de base (qui, pour la plupart, s'appliquent aux matrices : voir chapitre "**Fonctions de base**"), il existe des fonctions logiques spécifiques aux matrices décrites ici.

Fonction	Description
`isequal(mat1,mat2)`	Retourne le scalaire vrai ("1") si tous les éléments de *mat1* sont **égaux** aux éléments de *mat2*, faux ("0") sinon
`isscalar(var)` `isvector(var)`	Retourne le scalaire vrai si *var* est un **scalaire**, faux si c'est un vecteur ou tableau ≥ 2-dim Retourne le scalaire vrai si *var* est un **vecteur** ou **scalaire**, faux si tableau ≥ 2-dim
`iscolumn(var)` `isrow(var)`	Retourne le scalaire vrai si *var* est un vecteur **colonne** ou **scalaire**, faux si tableau ≥ 2-dim Retourne le scalaire vrai si *var* est un vecteur **ligne** ou **scalaire**, faux si tableau ≥ 2-dim
mat3 = `ismember(mat1,mat2)`	Cherche si les valeurs de *mat1* sont présentes dans *mat2* : retourne une matrice *mat3* de la même dimension que *mat1* où *mat3(i,j)*=1 si la valeur *mat1(i,j)* a été trouvée quelque-part dans dans *mat3*, sinon *mat3(i,j)*=0. Les matrices (ou vecteurs) *mat1* et *mat2* peuvent avoir des dimensions différentes. *mat1* et *mat2* peuvent être des tableaux cellulaires (contenant par exemple des chaînes) Ex : Si `a=[0 1 2 ; 3 4 5]` et `b=[2 4;6 8;10 12;14 16;18 20]`, la fonction `ismember(a,b)` retourne alors [0 0 1 ; 0 1 0] Si `a={'pomme','poire';'fraise','cerise'}` et `b={'fraise','abricot'}`, alors `ismember(a,b)` retourne [0 0 ; 1 0]

`any(vec)` et `all(vec)`
 	
`any(mat)` et `all(mat)`
	Retourne le scalaire vrai si l'un au moins des éléments du vecteur *vec* n'est **pas nul**, respectivement si tous les éléments ne sont pas nuls
	
	Comme ci-dessus, mais analyse les colonnes de *mat* et retourne ses résultats sur un vecteur ligne

4.5.6 Indexation logique

Introduction

Sous le terme d' "**indexation logique**" (logical indexing, logical subscripting) on entend la technique d'indexation par une **matrice logique**, c'est-à-dire une matrice booléenne composée de 0 ou de 1. Ces "matrices logiques d'indexation" résultent le plus souvent :

- d'opérations basées sur les "opérateurs relationnels et logiques" (p.ex. `==`, `>`, `~`, etc...) (voir le chapitre "**opérateurs de base**")
- de "fonctions logiques de base" (les fonctions `is*`, p.ex. `isnan`) (voir le chapitre "**opérateurs de base**")
- ainsi que des "fonctions matricielles logiques" (voir ci-dessus)
- si la matrice logique est construite "à la main" (avec des valeurs 0 et 1), on devra lui appliquer la fonction `logical` pour en faire une vraie matrice logique booléenne (voir exemple ci-dessous).

Il faudrait en principe que les **dimensions** de la matrice logique soient **identiques** à celles de la matrice que l'on indexe (cela engendrant, dans le cas contraire, des différences de comportement entre MATLAB et Octave...).

L'avantage de l'indexation logique réside dans le fait qu'il s'agit d'un **mécanisme vectorisé** (donc bien plus efficaces qu'un traitement basé sur des boucles `for` ou `while`).

Dans ce qui vient d'être dit, le terme "matrice" désigne bien entendu également des tableaux **multidimensionnels** ou de simples vecteurs (ligne ou colonne). Et encore mieux : l'indexation logique peut aussi être appliquée à des **structures** et des **tableaux cellulaires** !!! (voir les exemples spécifiques dans les chapitres traitant de ces deux types de données).

Utilisation de l'indexation logique

vec = `mat(mat_log)`

Examine la matrice *mat* à travers le "masque" de la matrice logique *mat_log* (de mêmes dimensions que *mat*), et retourne un **vecteur-colonne** *vec* comportant les éléments de *mat*(*i*,*j*) où *mat_log*(*i* ,*j*)=1. Les éléments sont déversés dans *vec* en examinant la matrice *mat* colonne après colonne.

Remarques importantes :

• *mat_log* peut être (et est souvent !) une expression logique basée sur la matrice *mat* elle-même. Ainsi, par exemple, `mat(mat>val)` (indexation de la matrice *mat* par la **matrice logique** produite par `mat>val`) retournera un vecteur-colonne contenant tous les éléments de *mat* qui sont supérieurs à *val*.

• On peut rapprocher cette fonctionnalité de la fonction `find` décrite plus haut. Pour reprendre l'exemple ci-dessus, `mat(find(mat>val))` (indexation de la matrice *mat* par le **vecteur d'indices à une dimension** produit par `find(mat>val)`) retournerait également les éléments de *mat* qui sont supérieurs à *val*.

Ex:

• Soit la matrice `m=[5 3 8 ; 2 9 3 ; 8 9 1]` ; `m(m>3)` retourne le vecteur-colonne [5 ; 8 ; 9 ; 9 ; 8] (contenant donc les éléments supérieurs à 3)

• Si l'on construit manuellement une matrice logique `m_log1=[1 0 1;0 1 0;1 1 0]` , on ne peut pas faire `m(m_log1)` , car `m_log1` n'est alors pas considéré par MATLAB/Octave comme une matrice logique (booléenne) mais comme une matrice de nombres... et MATLAB/Octave essaie alors de faire de l'indexation standard avec des indices nuls, d'où l'erreur qui est générée ! Il faut plutôt faire `m_log2=logical(m_log1)` (ou `m_log2=(m_log1~=0)`), puis `m(m_log2)` . On peut bien entendu aussi faire directement `m(logical(m_log1))` ou `m(logical([1 0 1;0 1 0;1 1 0]))` . En effet, regardez avec la commande `whos` , les types respectifs de `m_log1` et de `m_log2` !

• Pour remplacer les valeurs indéterminées (NaN) d'une série de mesures `s=[-4 NaN -2.2 -0.9 0.3 NaN 1.5`

`2.6]` en vue de faire un graphique, on fera `s=s(~isnan(s))` ou `s=s(isfinite(s))` qui retournent toutes deux s=[-4 -2.2 -0.9 0.3 1.5 2.6]

`mat(mat_log) = valeur`

Utilisée sous cette forme-là, l'indexation logique ne retourne pas un vecteur d'éléments de *mat*, mais **modifie certains éléments** de la matrice *mat* : tous les éléments de *mat*(*i*,*j*) où *mat_log*(*i* ,*j*)=1 seront remplacés par la *valeur* spécifiée. Comme cela a été vu plus haut, la matrice logique *mat_log* devrait avoir les mêmes dimensions que *mat*, et *mat_log* peut être (et est souvent !) une expression logique basée sur la matrice *mat* elle-même.

Ex:

• En reprenant la matrice `m=[5 3 8 ; 2 9 3 ; 8 9 1]` de l'exemple ci-dessus, l'instruction `m(m<=3)=-999` modifie la matrice `m` en remplaçant tous les éléments inférieurs où égaux à 3 par -999 ; celle-ci devient donc [5 -999 8 ; -999 9 -999 ; 8 9 -999]

• L'indexation logique peut aussi être appliquée à des chaînes de caractères pour identifier ou remplacer des caractères. Soit la chaîne `str='Bonjour tout le monde'`. L'affectation `str(isspace(str))='_'` remplace dans `str` tous les caractères <espace> par le caractère '_' et retourne donc str='Bonjour_tout_le_monde'

4.6 Chaînes de caractères

4.6.1 Généralités

Dans les usages courants, MATLAB/Octave stocke les **chaînes de caractères** ("string") sous forme de **vecteurs-ligne** (type "char array" sous MATLAB et "char" sous Octave) dans lesquels chaque caractère est un **élément** du vecteur (occupant physiquement 2 octets). Mais il est aussi possible de manipuler des **matrices de chaînes**, comme nous l'illustrons ci-dessous (ainsi que des "**tableaux cellulaires**" de chaînes : voir plus loin).

`string = 'chaîne de caractères'`

Enregistre la *chaîne de caractères* (définie entre apostrophes) sur la variable *string* qui est ici un **vecteur-ligne**. Si la chaîne contient un apostrophe, il faut le dédoubler (sinon il est interprété comme signe de fin de chaîne... et la suite de la chaîne provoque une erreur)

Ex : `section = 'Sciences et ingénierie de l''environnement'`

`string = "chaîne de caractères"`

Propre à Octave, cet usage de guillemets est intéressante car elle permet de définir, dans la chaîne, des caractères spéciaux :

- `\t` pour le caractère `<tab>`
- `\n` pour un saut à la ligne (`<newline>`) ; mais la chaîne reste cependant un vecteur ligne et non une matrice
- `\"` pour le caractère `"`
- `\'` pour le caractère `'`
- `\\` pour le caractère `\`

Ex : `disp("Texte\ttabulé\net sur\t2 lignes")`

`string(i:j)`

Retourne la **partie de la chaîne** *string* comprise entre le *i*-ème et le *j*-ème caractère

Ex : suite à l'exemple ci-dessus, `section(13:22)` retourne la chaîne "ingénierie"

`string(i:end)` , équivalent à `string(i:length(string))`

Retourne la **fin de la chaîne** *string* à partir du *i*-ème caractère

Ex: suite à l'exemple ci-dessus, `section(29:end)` retourne la chaîne "environnement"

`[s1 s2 s3...]`

Concatène horizontalement les chaînes *s1*, *s2*, *s3*

Ex: soit `s1=' AAA ', s2='CCC ', s3='EEE '`

alors `[s1 s2 s3]` retourne " AAA CCC EEE "

`strcat(s1,s2,s3...)`

Concatène horizontalement les chaînes *s1*, *s2*, *s3* en supprimant les caractères <espace> **terminant** les chaînes *s1*, *s2*... ("trailing blanks") (mais pas les <espace> commençant celles-ci). Noter que, sous Octave, cette suppression des espaces n'est implémentée qu'à partir de la version 3.2.0

Ex: soit `s1=' AAA ', s2='CCC ', s3='EEE '`

alors `strcat(s1,s2,s3)` retourne " AAACCCEEE"

`strvcat(s1,s2,s3...)`

Concatène **verticalement** les chaînes *s1*, *s2*, *s3*

Sous Octave, implémenté depuis la version 3.2.0

`mat_string = str2mat(s1,s2,s3...)`
`mat_string = char(s1,s2,s3...)` (depuis version 5 de MATLAB)
`mat_string = [s1 ; s2 ; s3 ...]`

Produit une **matrice de chaînes de caractères** *mat_string* contenant la chaîne *s1* en 1ère ligne, *s2* en seconde ligne, *s3* en 3ème ligne, etc... La première et la seconde forme fonctionnent à la fois sous MATLAB et Octave (la seconde depuis MATLAB 5). Quand à la 3ème forme, elle ne fonctionne que sous Octave (MATLAB générant une erreur si les chaînes *s1*, *s2*, *s3*... n'ont pas toutes la même longueur).

⚠ **Remarque importante**: pour produire cette matrice *mat_string*, MATLAB et Octave complètent automatiquement chaque ligne par le nombre nécessaire de caractères <espace> ("trailing blanks") afin que toutes les lignes soient de la même longueur (même nombre d'éléments, ce qui est important dans le cas où les chaînes *s1*, *s2*, *s3*... n'ont pas le même nombre de caractères). Cet inconvénient n'existe pas si l'on recourt à des **tableaux cellulaires** plutôt qu'à des matrices de chaînes.
On peut **convertir** une matrice de chaînes en un "tableau cellulaire de chaînes" avec la fonction `cellstr` (voir chapitre "**Tableaux cellulaires**").

Ex:
• en utilisant les variables "s1", "s2", "s3" de l'exemple ci-dessus, `mat=str2mat(s1,s2,s3)` retourne la matrice de chaînes de dimension 3x16 caractères :

```
Jules Dupond
Albertine Durand
Robert Muller
```

puis `mat=str2mat(mat,'xxxx')` permettrait ensuite d'ajouter une ligne supplémentaire à cette matrice
• pour stocker ces chaînes dans un tableau cellulaire, on utiliserait `tabl_cel={s1;s2;s3}` ou `tabl_cel={'Jules Dupond';'Albertine Durand';'Robert Muller'}`
• ou pour convertir la matrice de chaîne ci-dessus en un tableau cellulaire, on utilise `tabl_cel=cellstr(mat)`

`mat_string(i,:)`
`mat_string(i,j:k)`

Retourne la *i*-ème ligne de la matrice de chaînes *mat_string*,
respectivement la sous-chaîne de cette ligne allant du *j*-ème au *k*-ème caractère
Ex : en reprenant la matrice "mat" de l'exemple ci-dessus, `mat(2,:)` retourne "Albertine Durand", et `mat(3,8:13)` retourne "Muller"

Remarque importante concernant l'usage de **caractères accentués** dans des **scripts ou fonctions** (M-files) :

- Que ce soit sous MATLAB ou Octave, si un M-file définit des chaînes contenant des caractères acccentués (caractères non ascii-7bits) et les écrit sur un fichier, l'**encodage des caractères** dans ce fichier dépend de l'encodage du M-file qui l'a généré. Si le M-file est encodé ISO-latin-1, le fichier produit sera encodé ISO-latin1 ; si le M-file est encodé UTF-8, le fichier produit sera encodé UTF-8...
- Sous **Windows**, si le M-file est encodé UTF-8 et qu'il affiche des chaînes dans la console MATLAB/Octave (avec `disp`, `fprintf` ...) :
  - sous MATLAB (testé sous 7.8), les caractères accentués ne s'affichent pas proprement
  - sous Octave 3.2.x, ils s'afficheront proprement pour autant que la police de caractères utilisée dans la fenêtre de commande soit de type TrueType (par exemple Lucida Console) et que l'on ait activé le code-page Windows UTF-8 avec la commande `dos('chcp 65001')`
- Sous **Linux**, le mode par défaut est UTF-8 et il n'y a pas de problème particulier

4.6.2 Fonctions générales relatives aux chaînes

Sous MATLAB, M `helpwin strfun` donne la listes des fonctions relatives aux chaînes de caractères.

Fonction	Description
`length(string)`	Retourne le **nombre de caractères** de la chaîne *string*

`deblank(string)`
`strtrim(string)`
 	
`blanks(n)`
	Supprime les car. <espace> terminant *string* (**trailing blanks**)
	Supprime les car. <espace> débutant et terminant *string* (**leading & trailing blanks**)
	
	Retourne une chaîne de *n* caractères <espace>

`findstr(string,s1 {,`O `overlap})` ou `strfind(cell_string,s1)`	Retourne, sur un vecteur ligne, la **position** dans *string* de toutes les chaînes *s1* qui ont été trouvées. Noter que `strfind` est capable d'analyser un tableau cellulaire de chaînes, alors que `findstr` ne peut qu'analyser des chaînes simples. Si ces 2 fonctions ne trouvent pas de sous-chaîne *s1* dans *string*, elles retournent un tableau vide (`[]`) O Si le paramètre optionnel *overlap* est présent est vaut `0`, `findstr` ne tient pas compte des occurences superposées (voir exemple ci-dessous) Ex: si l'on a `str='Bla bla bla *xyz* bla etc...'`, alors • `star=findstr(str,'*')` ou `star=strfind(str,'*')` retournent le vecteur [13 17] indiquant la position des "*" dans la variable "str" • `str(star(1)+1:star(2)-1)` retourne la sous-chaîne de "str" se trouvant entre "*", soit "xyz" • `length(findstr(str,'bla'))` retourne le nombre d'occurences de "bla" dans "str", soit 3 • `isempty(findstr(str,'ZZZ'))` retourne "vrai" (valeur `1`), car la sous-chaîne "ZZZ" n'existe pas dans "str" • `findstr('abababa','aba')` retourne [1 3 5], alors que O `findstr('abababa','aba',0)` retourne [1 5]

strmatch(*mat_string*,*s1* {,'exact'})	Retourne un vecteur-colonne contenant les numéros des lignes de la **matrice de chaîne** *mat_string* qui 'commencent' par la chaîne *s1*. En ajoutant le paramètre 'exact', ne retourne que les numéros des lignes qui sont 'exactement identiques' à *s1*. Ex: `strmatch('abc', str2mat('def','abc','yyy','abc xxx'))` retourne [2 ; 4] En ajoutant le paramètre `'exact'`, ne retourne que [2]
regexp(*mat_string*, *pattern*) **regexpi**(*mat_string*, *pattern*)	Effectue une **recherche** dans *mat_string* à l'aide du motif défini par l'**expression régulière** *pattern* (extrêmement puissant... lorsque l'on maîtrise les expression régulières Unix). La seconde forme effecte une recherche "case insensitive" (ne différenciant pas majuscules/minuscules).
strrep(*string*,*s1*,*s2*)	Retourne une copie de la chaîne *string* dans laquelle toutes les occurences de *s1* sont **remplacées** par *s2* Ex: `strrep('abc//def//ghi/jkl','//','|')` retourne "abc|def|ghi/jkl"
regexprep(*s1*, *pattern*, *s2*)	Effectue un **remplacement**, dans *s1*, par *s2* là où l'**expression régulière** *pattern* est satisfaite
strsplit(*string*,*car_sep*)	**Découpe** la chaîne *string* en utilisant les différents **caractères** de *car_sep*, et retourne les sous-chaînes résultantes sur un vecteur cellulaire de chaînes. Ex: `strsplit('abc/def/ghi*jkl','/*')` retourne le vecteur cellulaire `{'abc','def','ghi','jkl'}`

split(*string*,*str_sep*)	**Découpe** la chaîne *string* selon la **chaîne**-séparateur *str_sep*, et retourne les sous-chaînes résultantes sur une matrice de chaînes. Note: cette fonction est dépréciée (disparaîtra sous Octave 3.6) en faveur de `strsplit` (qui fonctionne cependant différemment)
	
	Ex: `split('abc//def//ghi/jkl','//')` retourne la matrice
	`abc`
	`def`
	`ghi/jkl`

[*debut fin*]= **`strtok(`***`string,delim`***`)`**	**Découpe** la chaîne *string* en 2 parties selon le(s) caractère(s) de délimitation énuméré(s) dans la chaîne *delim* ("tokens") : sur *debut* est retournée la première partie de *string* (caractère de délimitation non compris), sur *fin* est retournée la seconde partie de *string* (commençant par le caractère de délimitation). Si le caractère de délimitation est <tab>, il faudra entrer ce caractère tel quel dans *delim* (et non pas '\t' qui serait interprété comme les 2 délimiteurs \ et t). Si ce que l'on découpe ainsi ce sont des nombres, il faudra encore convertir les chaînes résultantes en nombres avec la fonction **`str2num`** (voir plus bas). **Ex**: **`[debut fin]=strtok('Abc def, ghi.', ',:;.')`** découpera la chaîne en utilisant les délimiteurs de phrase habituels et retournera, dans le cas présent, debut='Abc def' et fin=', ghi.'
`strjust(`*`var`***`,'left|center|right')`**	**Justifie** la chaîne ou la matrice de chaîne *var* à gauche, au centre ou à droite. Si l'on ne passe à cette fonction que la chaîne, la justification s'effectue à droite
`sortrows(`*`mat_string`***`)`**	**Trie** par ordre alphabétique croissant les lignes de la matrice de chaînes *mat_string*
vect_log = ***`string1==string2`***	**Comparaison caractères après caractères** de 2 chaînes *string1* et *string2* de longueurs identiques (retourne sinon une erreur !). Retourne un vecteur logique (composé de 0 et de 1) avec autant d'élément que de caractères dans chaque chaîne. Pour tester l'égalité exacte de chaînes de longueur indéfinie, utiliser plutôt **`strcmp`** ou **`isequal`** (voir ci-dessous).

strcmp(*string1*,*string2*) ou **isequal**(*string1*,*string2*) **strcmpi**(*string1*,*string2*) **strncmp**(*string1*,*string2*,*n*) **strncmpi**(*string1*,*string2*,*n*)	**Compare** les 2 chaînes *string1* et *string2*: retourne 1 si elles sont identiques, 0 sinon. La fonction **strcmpi** ignore les différences entre majuscule et minuscule ("casse") Ne compare que les ***n* premiers** caractères des 2 chaînes La fonction **strncmpi** ignore les différences entre majuscule et minuscule ("casse")
ischar(*var*) **isletter**(*string*) **isspace**(*string*)	Retourne 1 si *var* est une **chaîne** de caractères, 0 sinon. Ne plus utiliser **isstr** qui va disparaître. Retourne un vecteur de la taille de *string* avec des 1 là où *string* contient des caractères de l'**alphabet**, et des 0 sinon. Retourne un vecteur de la taille de *string* avec des 1 là où *string* contient des **caractères de séparation** (<espace> <tab> <newline> <formfeed>), et des 0 sinon.
isstrprop(*var*, *propriete*)	Test les *propriétés* de la chaîne *var* (alphanumérique, majuscule, minuscule, espaces, ponctuation, chiffres décimaux/hexadécimaux, caractères de contrôle...) Sous Octave, implémenté depuis la version 3.2.0

4.6.3 Fonctions de conversion relatives aux chaînes

Fonction	Description
`lower(string)` `upper(string)`	Convertit la chaîne *string* en **minuscules**, respectivement en **majuscules**
`abs(string)` ou `double(string)`	Convertit les caractères de la chaîne *string* en leurs **codes** décimaux selon la table **ASCII** ISO-Latin-1 Ex: `abs('àéèçâêô')` retourne le vecteur [224 233 232 231 226 234 244] (code ASCII de ces caractères accentués)
`char(var)`	Convertit les nombres de la variable *var* en **caractères** (selon encodage 8-bits ISO-Latin-1) Remarque: avec Octave-Forge 2.1.42 sous Windows, la fenêtre de commande refuse que l'on entre au clavier des caractères accentués ; cette fonction permet donc de contourner cette difficulté Ex: `char(224)` retourne le caractère "à", `char([233 232])` retourne la chaîne "éè"
`sprintf(format,variable(s)...)`	Permet de convertir un(des) **nombre(s)** en une **chaîne** (voir chapitre "**Entrées-sorties**") Voir aussi les fonctions `int2str` et `num2str` (qui sont cependant moins flexibles)
`mat2str(mat {,n})`	Convertit la **matrice** *mat* en une **chaîne** de caractère incluant les crochets [ ] et qui serait dont "évaluable" avec la fonction `eval` (voir ci-dessous). L'argument *n* permet de définir la précision (nombre de chiffres). Cette fonction peut être intéressante pour sauvegarder une matrice sur un fichier (en combinaison avec `fprintf`, voir chapitre "**Entrées-sorties**"). Ex: `mat2str(eye(3,3))` produit la chaîne "[1 0 0;0 1 0;0 0 1]"

4.7 Tableaux multidimensionnels

4.7.1 Généralités

Sous la dénomination de "**tableaux multidimensionnels**" (multidimensional arrays, ND-Arrays), il faut simplement imaginer des matrices ayant **plus de 2 indices** (ex: `B(2,3,3)`). S'il est facile de se représenter la 3e dimension (voir Figure ci-contre), c'est un peu plus difficile au-delà :

- 4 dimensions pourrait être vu comme un vecteur de tableaux 3D
- 5 dimensions comme une matrice 2D de tableaux 3D
- 6 dimensions comme un tableau 3D de tableaux 3D...

Un tableau tridimensionnel permettra, par exemple, de stocker une séquence de matrices 2D de tailles identiques (pour des matrices de tailles différentes, on devra faire appel aux "tableaux cellulaires" décrits plus loin) relatives à des données physiques de valeurs spatiales (échantillonées sur une grille) évoluant en fonction d'un 3e paramètre (altitude, temps...).

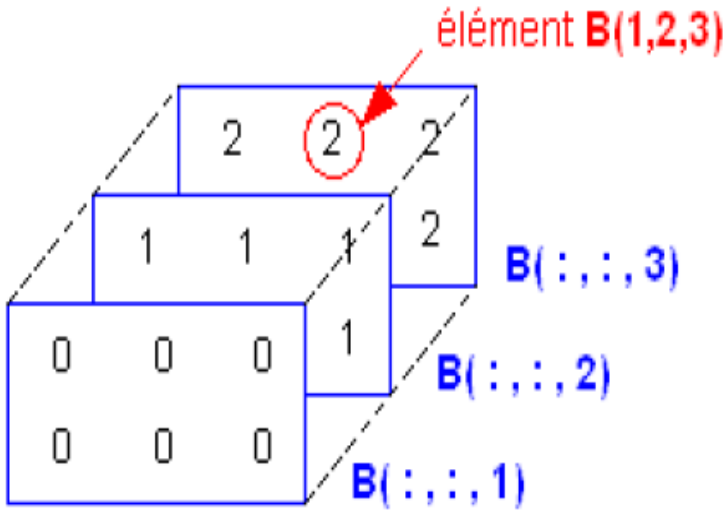


Figure: exemple d'un tableau tridimensionnel B de dimension 2x3x3

Les tableaux multidimensionnels sont supportés depuis longtemps sous MATLAB, et depuis la version 2.1.51 d'Octave.

Ce chapitre illustre la façon de définir et utiliser des tableaux multidimensionnels. Les exemples, essentiellement 3D, peuvent sans autre être extrapolés à des dimensions plus élevées.

4.7.2 Tableaux multidimensionnels

La **génération** de tableaux multidimensionnels peut s'effectuer simplement par indexation, c'est-à-dire **en utilisant un 3ème, 4ème... indice** de matrice.

Ex :

- si le tableau `B` ne pré-existe pas, la simple affectation `B(2,3,3)=2` va générer un tableau tridimensionnel (de dimension 2x3x3 analogue à celui de la Figure ci-dessus) dont le dernier élément, d'indice (2,3,3), sera mis à la valeur 2 et tous les autres éléments initialisés à la valeur 0
- puis `B(:,:,2)=[1 1 1 ; 1 1 1]` ou `B(:,:,2)=ones(2,3)` ou encore plus simplement `B(:,:,2)=1` permettrait d'initialiser tous les éléments de la seconde "couche" de ce tableau 3D à la valeur 1
- et `B(1:2,2,3)=[2;2]` permettrait de modifier la seconde colonne de la troisième "couche" de ce tableau 3D
- on pourrait de même accéder individuellement à tous les éléments `B(k,l,m)` de ce tableau par un ensemble de boucles `for` tel que (bien que ce ne soit pas efficace ni élégant pour un langage "vectorisé" tel que MATLAB/Octave) :

```
for k=1:2          % indice de ligne
  for l=1:3        % indice de colonne
    for m=1:3      % indice de "couche"
      B(k,l,m)=...
    end
  end
end
```

Certaines fonctions MATLAB/Octave déjà présentées plus haut permettent de générer directement des tableaux multidimensionnels lorsqu'on leur passe plus de 2 arguments : `ones`, `zeros`, `rand`, `randn`.

Ex:

- `C=ones(2,3,3)` génère un tableau 3D de dimension 2x3x3 dont tous les éléments sont mis à la valeur 1
- `D=zeros(2,3,3)` génère un tableau 3D de dimension 2x3x3 dont tous les éléments sont mis à la valeur 0
- `E=rand(2,3,3)` génère un tableau 3D de dimension 2x3x3 dont les éléments auront une valeur aléatoire comprise entre 0 et 1

Voir aussi les fonctions de génération et réorganisation de matrices, telles que `repmat(tableau,[M N P ...])` et `reshape(tableau,M,N,P...)`, qui s'appliquent également aux tableaux multidimensionnels.

Les opérations dont l'un des deux opérandes est un **scalaire**, les **opérateurs de base** (arithmétiques, logiques, relationnels...) ainsi que les fonctions opérant "**élément par élément**" sur des matrices 2D (fonctions trigonométriques...) travaillent de façon identique sur des tableaux multidimensionnels, c'est-à-dire s'appliquent à tous les éléments du tableau. Par contre les fonctions qui opèrent spécifiquement sur des matrices 2D et vecteurs (algèbre linéaire, fonctions "matricielles" telles que inversion, produit matriciel, etc...) ne pourront être appliquées qu'à des sous-ensembles 1D (vecteurs) ou 2D ("tranches") des tableaux multidimensionnels, donc moyennement un usage correct des indices de ces tableaux !

Ex:

- en reprenant le tableau C de l'exemple précédent, `F=3*C` retourne un tableau dont tous les éléments auront la valeur 3
- en faisant `G=E+F` on obtient un tableau dont les éléments ont une valeur aléatoire comprise entre 3 et 4
- `sin(E)` calcule le sinus de tous les éléments du tableau E

Certaines fonctions présentées plus haut (notamment les fonctions **statistiques** `min`, `max`, `sum`, `prod`, `mean`, `std`...) permettent de spécifier un "**paramètre de dimension**" `d` qui est très utile dans le cas de tableaux

multidimensionnels. Illustrons l'usage de ce paramètre avec la fonction **sum** :

sum(*tableau, d*)

Calcule la **somme** des éléments en faisant **varier le *d*-ème indice** du *tableau*

Ex: dans le cas d'un *tableau* de dimension 3x4x5 (nombre de: lignes x colonnes x profondeur)

- **sum(*tableau*,1)** retourne un tableau 1x4x5 contenant la somme des éléments par ligne
- **sum(*tableau*,2)** retourne un tableau 3x1x5 contenant la somme des éléments par colonne
- **sum(*tableau*,3)** retourne une matrice 3x4x1 contenant la somme des éléments calculés selon la profondeur

La génération de tableaux multidimensionnels peut également s'effectuer par la fonction de **concaténation** de matrices (voire de tableaux !) de dimensions inférieures avec la fonction **cat**

cat(*d, mat1, mat2*)

Concatène les 2 matrices *mat1* et *mat2* selon la *d*-ème dimension. Si *d*=1 (indice de ligne) => concaténation verticale. Si *d*=2 (indice de colonne) => concaténation horizontale. Si *d*=3 (indice de "profondeur") => création de "couches" suppémentaires ! Etc...

Ex:

- **A=cat(1,zeros(2,3),ones(2,3))** ou **A=[zeros(2,3);ones(2,3)]** retoument la matrice 4x2 A=[0 0 0 ; 0 0 0 ; 1 1 1 ; 1 1 1] (on reste en **2D**)
- **A=cat(2,zeros(2,3),ones(2,3))** ou **A=[zeros(2,3),ones(2,3)]** retoument la matrice 2x4 A=[0 0 0 1 1 1 ; 0 0 0 1 1 1] (on reste en **2D**)
- et **B=cat(3,zeros(2,3),ones(2,3))** retourne le tableau à **3 dimensions** 2x3x2 composé de B(:,:,1)=[0 0 0 ; 0 0 0] et B(:,:,1)=[1 1 1 ; 1 1 1]
- puis **B=cat(3,B,2*ones(2,3))** ou **B(:,:,3)=2*ones(2,3)** permettent de rajouter une nouvelle "couche" à ce tableau (dont la dimension passe alors à 2x3x3) composé de B(:,:,3)=[2 2 2; 2 2 2], ce qui donne exactement le tableau de la Figure ci-dessus

Les fonctions ci-dessous permettent de connaître la **dimension** d'un tableau (2D, 3D, 4D...) et la "**taille de chaque dimension**" :

vect= `size(tableau)`
taille= `size(tableau, dimension)`
Retourne un vecteur-ligne *vect* dont le i-ème élément indique la taille de la i-ème dimension du *tableau*
Retourne la *taille* du *tableau* correspondant à la *dimension* spécifiée

Ex:

- pour le tableau `B` ci-dessus, `size(B)` retourne le vecteur [2 3 3], c'est-à-dire respectivement le nombre de lignes, de colonnes et de "couches"
- et `size(B,1)` retourne ici 2, c'est-à-dire le nombre de lignes (1ère dimension)
- pour un scalaire (vu comme une matrice dégénérée) cette fonction retourne toujours [1 1]

`numel(tableau)` *(NUMber of ELements)*
Retourne le nombre d'éléments *tableau*. Identique à `prod(size(tableau))` ou `length(mat(:))`, mais un peu plus "lisible"
Ex: pour le tableau `B` ci-dessus, `numel(B)` retourne donc 18

`ndims(tableau)`
Retourne la dimension *tableau* : 2 pour une matrice 2D et un vecteur ou un scalaire (vus comme des matrices dégénérées !), 3 pour un tableau 3D, 4 pour un tableau quadri-dimensionnel, etc... Identique à `length(size(tableau))`
Ex: pour le tableau `B` ci-dessus, `ndims(B)` retourne donc 3

Il est finalement intéressant de savoir, en matière d'échanges, qu'Octave permet de sauvegarder des tableaux multidimensionnels sous forme texte (utiliser `save -text` ...), ce que ne sait pas faire MATLAB.

4.8 Structures (enregistrements)

4.8.1 Généralités

Une "**structure**" (enregistrement, record) est un type d'objet MATLAB/Octave (que l'on retrouve dans d'autres langages) se composant de plusieurs "**champs**" nommés (fields) qui peuvent être **de types différents** (chaînes, matrices, tableaux cellulaires...), champs qui peuvent eux-mêmes se composer de sous-champs... MATLAB/Octave permet logiquement de créer des "**tableaux de structures**" (structures array) multidimensionels.

On accède aux champs d'une structure avec la **syntaxe** `structure.champ.sous_champ` ... (usage du caractère "`.`" comme séparateur). Pour illustrer les concepts de base relatifs aux structures, prenons l'exemple d'une structure permettant de stocker les différents attributs d'une personne (nom, prénom, age, adresse, etc...).

Exemple:

A) Création d'une **structure** *personne* par définition des attributs du 1er individu :

- avec `personne.nom='Dupond'` la structure est mise en place et contient le nom de la 1ère personne ! (vérifiez avec `whos personne`)
- avec `personne.prenom='Jules'` on ajoute un champ *prenom* à cette structure et l'on définit le prénom de la 1ère personne
- et ainsi de suite : `personne.age=25 ; personne.code_postal=1010 ; personne.localite='Lausanne'`
- on peut, à ce stade, vérifier le contenu de la structure en frappant `personne`

Tableau de structures **personne**

nom: Dupond **prenom**: Jules **age**: 25 **code_postal**: 1010 **localite**: Lausanne **enfants**: - **tel.prive**: 021 123 45 67 **tel.prof**: 021 987 65 43
nom: Durand **prenom**: Albertine **age**: 30 **code_postal**: 1205 **localite**: Geneve **enfants**: Arnaud Camille **tel.prive**: - **tel.prof**: -
nom: Muller **prenom**: Robert **age**: 28 **code_postal**: 2000 **localite**: Neuchatel **enfants**: - **tel.prive**: - **tel.prof**: -

B) Définition d'autres individus => la structure devient un **tableau de structures** :

- ajout d'une 2e personne avec `personne(2).nom='Durand' ; personne(2).prenom='Albertine' ; personne(2).age=30 ; personne(2).code_postal=1205 ; personne(2).localite='Geneve'`
- ajout d'une 3e personne via une notation plus compacte :

```
personne(3)=struct('nom','Muller','prenom','Robert','age',28,'code_postal',2000,'local
0
personne(3)=setfield('nom','Muller','prenom','Robert','age',28,'code_postal',2000,'loca
```

C) Ajout de **nouveaux champs** à un tableau de structures existant :

- ajout d'un champ *enfants* de type "tableau cellulaire" (voir chapitre suivant) en définissant les 2 enfants de la 2e personne avec :
  `personne(2).enfants={'Arnaud','Camille'}`
- comme illustration de la notion de **sous-champs**, définissions les numéros de téléphone privé et prof. ainsi :
  `personne(1).tel.prive='021 123 45 67' ; personne(1).tel.prof='021 987 65 43'`
  Attention : le fait de donner une valeur au champ principal *personne.tel* (avec `personne.tel='Xxx'`) ferait disparaître les sous-champs *tel.prive* et *tel.prof* !

D) Accès aux **structures** et aux **champs** d'un tableau de structures :
- la notation `structure(i)` retourne la *i*-ème structure du tableau de structures *structure*
- par extension, `structure([i j:k])` retournerait un tableau de structures contenant la *i*-ème structure et les structures *j* à *k* du tableau *structure*
- avec `structure(i).champ` on accède au contenu du *champ* spécifié du *i*-ème individu du tableau *structure*

- Avec `personne(1)` on récupère donc la structure correspondant à notre 1ère personne (Dupond),
  et `personne([1 3])` retourne un tableau de structures contenant la 1ère et la 3e personne
- `personne(1).tel.prive` retourne le No tel privé de la 1ère personne (021 123 45 67)
  Attention : comportements bizarres dans le cas de sous-champs : `personne(2).tel` retourne `[]` (ce qui est correct vu que la 2e personne n'a pas de No tél), mais `personne(2).tel.prive` provoque une erreur !
- `personne(2).enfants` retourne un tableau cellulaire contenant les noms des enfants de la 2e personne
  et `personne(2).enfants{1}` retourne le nom du 1er enfant de la 2e personne (Arnaud)

- Pour obtenir la **liste** de **toutes les valeurs d'un champ** spécifié, on utilise :
  sous **MATLAB** ou **Octave** (ici pour des chaînes : liste des noms de tous les individus) :
  - soit `tabl_cel_noms = { personne.nom }` qui retourne un objet *tab_cel_noms* de type "tableau cellulaire"
  - ou `[tab_cel_noms{1:length(personne)}] = deal(personne.nom)` (idem)
  - ou encore la boucle `for k=1:length(personne), tab_cel_noms{k}=personne(k).nom ; end` (idem)

  sous **MATLAB** ou **Octave** (ici pour des nombres : liste des ages de tous les individus) :
  - `vec_ages = [ personne.age ]` retourne un vecteur-ligne *vec_ages*

  sous **Octave** seulement :
  - `liste_noms = personne.nom` retourne un objet *liste_noms* de type "liste"

- Et l'on peut même utiliser l'**indexation logique** pour extraire des parties de structure !
  Voici un exemple très parlant : l'instruction `prenoms_c = { personne( [personne.age] > 26 ).prenom }` retourne le vecteur cellulaire *prenoms_c* contenant les prénoms des personnes âgées de plus de 26 ans ; on a pour ce faire "indexé logiquement" la structure `personne` par le vecteur logique `[personne.age] > 26`

E) Suppression de **structures** ou de **champs** :
- pour supprimer des structures, on utilise la notation habituelle `structure(...)=[]`
- pour supprimer des champs, on utilise la fonction `structure = rmfield(structure,'champ')`

- `personne(:).age=[]` supprime l'âge des 2 personnes, mais conserve le champ âge de ces structures
- `personne(2)=[]` détruit la 2e structure (personne Durand)
- `personne = rmfield(personne,'tel')` supprime le champ *tel* (et ses sous-champs *prive* et *prof*) dans toutes les structures du tableau *personne*

F) Champs de type **matrices** ou **tableau cellulaire** :
- habituellement les champs sont de type scalaire on chaîne, mais ce peut aussi être des matrices ou des tableaux cellulaires !
- avec `personne(1).naissance_mort=[1920 2001]` on définit un champ `naissance_mort` de type vecteur ligne
- puis on accède à l'année de mort du premier individu avec `personne(1).naissance(2)` ;
- ci-dessus, `enfants` illustre un champ de type tableau cellulaire

G) Matrices de structures :
- ci-dessus, `personne` est en quelque-sorte un vecteur-ligne de structures
- on pourrait aussi définir (même si c'est un peu "tordu") un tableau bi-dimensionnel (matrice) de structures en utilisant 2 indices (numéro de ligne et de colonne) lorsque l'on définit/accède à la structure, par exemple `personne(2,1)` ...

4.8.2 Fonctions spécifiques relatives aux structures

Fonction	Description

`struct`
`setfield`
`rmfield`
	Ces fonctions ont été vues dans l'exemple ci-dessus...

`nfields(struct)`	Retourne le nombre de champs de la structure *struct*

`fieldnames(struct)`
 	
`struct_elements(struct)`
	Retourne la **liste des champs** de la structure (ou du tableau de structures) *struct*. Cette liste est de type "tableau cellulaire" (à 1 colonne) sous MATLAB, et de type "liste" dans Octave.
	La fonction `struct_elements` fait de même, mais retourne cette liste sous forme d'une matrice de chaînes.

`getfield(struct,'champ')`	Est identique à `struct.champ`, donc retourne le contenu du champ *champ* de la structure *struct*

`isstruct(var)`
 	
`isfield(struct,'champ')`
	Test si var est un objet de type **structure** (ou tableau de structures) : retourne 1 si c'est le cas, 0 sinon.
	
	Test si *champ* est un **champ** de la structure (ou du tableau de structures) *struct* : retourne 1 si c'est le cas, 0 sinon.

[*n m*]= `size(tab_struct)` `length(tab_struct)`	Retourne le nombre *n* de **lignes** et *m* de **colonnes** du tableau de structures *tab_struct*, respectivement le **nombre total** de structures

4.8.2 Fonctions spécifiques relatives aux structures

Fonction	Description

`struct`
`setfield`
`rmfield`
	Ces fonctions ont été vues dans l'exemple ci-dessus...

`nfields(struct)`	Retourne le nombre de champs de la structure *struct*

`fieldnames(struct)`
 	
`struct_elements(struct)`
	Retourne la **liste des champs** de la structure (ou du tableau de structures) *struct*. Cette liste est de type "tableau cellulaire" (à 1 colonne) sous MATLAB, et de type "liste" dans Octave.
	La fonction `struct_elements` fait de même, mais retourne cette liste sous forme d'une matrice de chaînes.

`getfield(struct,'champ')`	Est identique à `struct.champ`, donc retourne le contenu du champ *champ* de la structure *struct*

`isstruct(var)`
 	
`isfield(struct,'champ')`
	Test si var est un objet de type **structure** (ou tableau de structures) : retourne 1 si c'est le cas, 0 sinon.
	
	Test si *champ* est un **champ** de la structure (ou du tableau de structures) *struct* : retourne 1 si c'est le cas, 0 sinon.

[*n m*]= `size(tab_struct)` `length(tab_struct)`	Retourne le nombre *n* de **lignes** et *m* de **colonnes** du tableau de structures *tab_struct*, respectivement le **nombre total** de structures

`for k=1:length(tab_struct)`
`  % on peut accéder à tab_struct(k).champ`
`end`
	On **boucle** ainsi **sur tous les éléments** du tableau de structures *tab_struct* pour accéder aux valeurs correspondant au *champ* spécifié.
	Ex : `for k=1:length(personne), tab_cel_noms{k}=personne(k).nom ; end` (voir plus haut)

5.1.1 Généralités

De façon interne, MATLAB/Octave gère les dates et le temps sous forme de **nombres** (comme la plupart des autres langages de programmation, tableurs...). L' "**origine du temps**", pour MATLAB/Octave, a été définie au **1er janvier de l'an 0** à minuit, et elle est mise en correspondance avec le nombre 1 (vous pouvez vérifier cela avec `datestr(1.0001)`). **Chaque jour** qui passe, ce nombre est **incrémenté de 1**, et les heures, minutes et secondes dans la journée correspondent donc à des fractions de jour (partie décimale du nombre exprimant le temps).

On obtient la liste des fonctions relatives à la gestion du temps avec `helpwin timefun` ou au chapitre "Timing Utilities" du manuel Octave.

5.1.2 Fonctions retournant la date et heure courante

Fonction	Description
`now`	Retourne le nombre exprimant la **date et heure locale courante** (donc le nombre de jours -et fractions de jours- écoulés depuis le 1er janvier 0000). Passez au préalable la commande `format long` si vous voulez afficher ce nombre en pleine précision. Ex: • `rem(now,1)` (partie décimale) retourne donc l'heure locale courante sous forme de fraction de jour, et `datestr(rem(now,1),'HH:MM:SS')` retourne l'heure courante sous forme de chaîne ! • `floor(now)` (partie entière) retourne donc le numéro de jour relatif à la date courante, et `datestr(floor(now))` la même chose que la fonction `date`

`date`	Retourne la **date courante** sous forme de **chaîne** de caractère au format 'dd-mmm-yyyy' (où mmm est le nom du mois en anglais abrégé aux 3 premiers caractères) Ex: `date` retourne 08-Apr-2005
`clock`	Retourne la **date et heure courante** sous forme d'un vecteur-ligne de 6 **valeurs numériques** [*annee mois jour heure minute seconde*]. Est identique à `datevec(now)`. Pour avoir des valeurs entières, faire `fix(clock)`. Ex: `clock` retourne le vecteur [2005 4 8 20 45 3] qui signifie 8 Avril 2005 à 20h 45' 03"

5.1.3 Fonctions de conversion

Fonction	Description

`datenum(annee, mois, jour {, heure, minute, seconde })`
 	
`datenum(date_string)`
	Retourne le **nombre** exprimant la date spécifiée par la chaîne *date_string* (voir `help datenum` pour les formats possibles), ou par les nombres *annee*, *mois*, *jour*, { *heure*, *minute* et *seconde* }
	
	Ex : `datenum(2005,4,8,20,45,0)` et `datenum('08-Apr-2005 20:45:00')` retournent le nombre 732410.8645833334

`datestr(date_num, 'format')`
 	
`datestr(date_num, code)`
	Convertit en **chaîne de caractères** la date/heure *date_num* spécifiée de façon numérique. Le formatage peut être spécifié par un *format* ou un *code* (voir `help datestr` pour plus de détails). Parmi les symboles qui peuvent être utilisés et combinés dans un *formats*, mentionnons :
	- `dd` , `dddd` , `ddd` (numéro du **jour**, nom du jour, nom abrégé)
	- `mm` , `mmmm` , `mmm` (numéro du **mois**, nom complet, nom abrégé)
	- `yyyy` , `yy` (**année** à 4 ou 2 chiffre)
	- `HH` (heure), `MM` (minutes), `SS` (secondes)
	
	Si le paramètre *date_num* est compris entre 0 et 1, cette fonction retourne des **heures/minutes/secondes**.

	Ex: • `datestr(now,'dd-mmm-yyyy HH:MM:SS')` retourne '08-Apr-2005 20:45:00' • `datestr(now,'ddd')` retourne 'Fri' (abréviation de Friday) (voir aussi la fonction `weekday` ci-dessous) 0 X Cette fonction était buguée sous Octave 3.0.5 à 3.2.x lorsque *date_num* correspondait à une date antérieure au 1er janvier 1970 sous Windows, et antérieure au 13 décembre 1901 sous Linux. C'est corrigé depuis Octave 3.4.
[*annee mois jour heure minute seconde*] = `datevec(date)`	A partir de la *date* spécifiée sous forme de chaîne de caractères (voir `help datevec` pour les formats possibles) ou sous forme numérique, retourne un vecteur ligne de 6 **valeurs numériques** définissant l'*annee*, *mois*, *jour*, *heure*, *minute* et *seconde* (comme `clock`). Pour avoir des valeurs entières, faire `fix(datevec(date))`. Ex: `datevec(732410.8646180555)` et `datevec('08-Apr-2005 20:45:03')` retournent le vecteur [2005 4 8 20 45 3]

5.1.4 Fonctions utilitaires

Fonction	Description

`calendar`
`calendar(annee,mois)`
`calendar(date)`
	Retourne une matrice 6x7 contenant le **calendrier du mois** courant, ou du mois contenant la *date* spécifiée (sous forme de chaîne de caractère ou de nombre), ou du *mois*/*annee* spécifiée (sous forme de nombres)
	
	Ex: `calendar(2005,4)` ou `calendar('8-Apr-2005')` retournent :
	
	Apr 2005
	S M Tu W Th F S
	0 0 0 0 0 1 2
	3 4 5 6 7 8 9
	10 11 12 13 14 15 16
	17 18 19 20 21 22 23
	24 25 26 27 28 29 30
	0 0 0 0 0 0 0
	
	(les 2 premières lignes, ici en gras, ne se trouvent pas dans la matrice 6x7 si vous appelez la fonction `calendar` en l'affectant à une variable)

[*numero_jour nom_jour*] = `weekday(date)`	Retourne le *numero_jour* (nombre) et *nom_jour* (chaîne) (respectivement: `1` et `Sun`, `2` et `Mon`, `3` et `Tue`, `4` et `Wed`, `5` et `Thu`, `6` et `Fri`, `7` et `Sat`) correspondant à la *date* spécifiée (passée sous forme de chaîne de caractère **ou** de nombre). Si cette fonction est affectée à une variable scalaire, retourne le *numero_jour*. Voir aussi, plus haut, la fonction `datestr` avec le format `'ddd'`. Ex: `[no nom]=weekday(732410.8646180555)` et `[no nom]=weekday('08-Apr-2005 20:45:03')` retournent les variables No=6 et Nom='Fri'

`eomday(annee,mois)`	Retourne le **nombre de jours** du *mois/annee* (spécifié par des nombres) Ex: `eomday(2005,4)` retourne 30 (i.e. il y a 30 jours dans le mois d'avril 2005)
`datetick('x|y|z',format)`	Sur l'axe spécifié (x, y ou z) d'un **graphique**, remplace au niveau des **labels** correspondants aux lignes de quadrillage (tick lines), les valeurs numériques par des **dates** au *format* indiqué Sous Octave, implémenté depuis la version 3.2.0

5.1.5 Fonctions de timing et de pause

Fonction	Description
`cputime`	• Sous MATLAB : retourne le nombre de secondes qui se sont écoulées depuis le début de la session MATLAB ("ellapse time") • Sous Octave : retourne le nombre de secondes de processeur consommées par Octave depuis le début de la session ("CPU time") ; sous Octave cette fonction a encore d'autres paramètres de sortie (voir `help cputime`) Ex: `t1 = cputime; A=rand(1000,1000); B=inv(A); dt = cputime-t1` : génération d'une matrice aléatoire A de dimension 1000 x 1000, inversion de celle-ci sur B, puis affichage du temps comsommé au niveau CPU pour faire tout cela (env. 8 secondes sur un Pentium 4 à 2.0

`tic`
`  instructions MATLAB/Octave...`
`ellapse_time = toc`
	La fonction `tic` démarre un compteur de temps, et la fonction `toc` l'arrête en retournant (sur la variable *ellapse_time* spécifiée) le **temps écoulé** en secondes
	
	Ex: l'exemple ci-dessus pourrait être aussi implémenté ainsi : `tic; A=rand(1000,1000); B=inv(A); dt = toc`

`etime(t2,t1)`	Retourne le **temps**, en secondes, séparant l'instant *t1* de l'instant *t2*. Ces 2 paramètres sont au format `clock` (donc vecteur-ligne [*annee mois jour heure minute seconde*]) Ex: l'exemple ci-dessus pourrait être aussi implémenté ainsi : `t1 = clock; A=rand(1000,1000); B=inv(A); dt = etime(clock,t1)`

`pause(secondes)`
ou `sleep(secondes)`
 	
`pause`
	Se met en **attente** durant le nombre de *secondes* spécifié.
	
	Passée sans paramètre, la fonction `pause` attend que l'utilisateur frappe n'importe quelle touche au clavier.
	
	Ex: dans un script, les lignes suivantes permettent de faire une pause bien explicite :
	`disp('Frapper n''importe quelle touche pour continuer... ') ; pause ;`