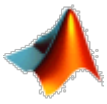
MATLAB & GNU Octave

Table des matières

ENAC-IT & ENAC-SSIE / EPFL / septembre 2015



Introduction à MATLAB et GNU Octave

par **Jean-Daniel BONJOUR**, © 1999-2015 **CC-BY-SA 4.0**
Service Informatique ENAC-IT & Section des Sciences et ingénierie de l'environnement (SSIE)
Faculté ENAC, EPFL, CH-1015 Lausanne

Avant-propos

Mis à jour en été 2015, le présent support de cours se réfère à **MATLAB R2014** et **GNU Octave 4.0.0** (avec extensions **Octave-Forge**). Nous nous efforçons de faire systématiquement le parallèle entre ces deux progiciels - le premier commercial, le second libre/open-source - en démontrant par là le très haut degré de compatibilité de GNU Octave par rapport à MATLAB, et donc le fait que ce logiciel libre peut être utilisé en lieu et place de MATLAB dans la plupart des situations (en environnement académique notamment).

Accessible sous **http://enacit1.epfl.ch/cours_matlab/**, ce support de cours a été conçu comme base à l'introduction à MATLAB et GNU Octave donnée à l'**EPFL** par l'auteur aux étudiants de :
- Sciences et ingénierie de l'environnement, Bachelor 3e semestre (**ENAC-SSIE**), cours "Informatique pour l'ingénieur"
- Génie Civil, Bachelor 3e semestre (**ENAC-SGC**), cours "Programmation MATLAB"

Les **conventions de notations** suivantes sont utilisées dans ce support de cours :

- en police de caractère à espacement fixe ombré bleu : **fonction** ou **commande** MATLAB/Octave à entrer telle quelle ; exemple: `help plot`
- en italique : vous devez substituer vous-même l'information désignée ; il s'agit en général des **paramètres** d'une fonction ; exemples: `save nom_fichier`, `plot(x,y)`
- entre accolades : on désigne ainsi des éléments facultatifs tels que les **options** d'une commande ou les paramètres d'une fonction ; exemples: `save fichier {-append}`, `pause({secondes})` ; ces accolades ne doivent donc pas être saisies ; exception à cette règle: les tableaux cellulaires où les accolades font partie intégrante de la syntaxe MATLAB/Octave
- caractère barre verticale : désigne un **choix** ; exemple: `grid ('on|off')` indiquant les 2 usages possibles `grid('on')` et `grid('off')`
- encadré de bleu : touche de **clavier** (ou combinaison de touches) ou bouton de **souris** ; exemples: enter, ctrl+C, curseur-haut, double-clic, clic-droite
- texte ombré de gris : choix dans un **menu** d'interface graphique ; exemple: File > Save as
- également ombré de gris mais encadré : **bouton** d'interface graphique ; exemple: Save

En règle générale toutes les instructions décrites dans ce support de cours s'appliquent à la fois à MATLAB et à GNU Octave. Dans le cas contraire, ou pour définir certaines spécificités, on utilisera les symboles suivants :

M indique que la fonctionnalité présentée n'est disponible que sous **MATLAB**
O indique que la fonctionnalité est propre à **GNU Octave**,
avec respectivement les backends graphiques basés Qt **Qt**/OpenGL, F **FLTK**/OpenGL ou G **Gnuplot**
signale une fonctionnalité pas encore disponible ou buguée

Par le signe on met en évidence des **fonctions et notions essentielles** MATLAB/Octave que l'étudiant, dans une première approche de cette matière, devrait assimiler en priorité.

Ce support de cours existe aussi sous forme de **fichier PDF** (voir menu ci-contre), mais celui-ci n'est pas mis à jour aussi fréquemment que la version web.

L'auteur reçoit très volontiers **toutes vos remarques** concernant ce support de cours (corrections, propositions de compléments...) qui peuvent lui être directement adressées par email à **jean-daniel.bonjour@epfl.ch**. D'avance un grand merci de votre feedback !

0. Installation et configuration de GNU Octave et des packages Octave-Forge

*Cette page est destinée à toute personne (étudiant, enseignant, chercheur...) souhaitant installer sur sa machine le logiciel libre **GNU Octave** et ses extensions **Octave-Forge** qui constituent le seul environnement de calcul scientifique/numérique et visualisation qui soit entièrement compatible avec **MATLAB** ("clone" MATLAB).*

0.1 Avant-propos

Les étudiants peuvent généralement se procurer sur leur campus une licence personnelle "MATLAB Student Version" (DVD d'installation de **MATLAB**, **Simulink** avec quelques *toolboxes*, vendu au prix de CHF 120.- environ) permettant l'installation de MATLAB sur leur machine privée et l'utilisation dans le cadre de leurs études. À l'EPFL cela est même possible sans bourse délier, l'école finançant la licence couvrant ce type d'usage. Mais pour les adeptes du "logiciel libre" (ou ceux qui ne veulent pas faire les frais d'une licence MATLAB), **GNU Octave** représente actuellement la meilleure alternative libre/open-source à MATLAB (donc utilisable gratuitement et sans restriction).

Il existe encore d'**autres logiciels libres** dans le domaine du calcul scientifique, mais non compatibles avec MATLAB (autre syntaxe, donc code MATLAB non réutilisable tel quel). Ces alternatives sont mentionnées au chapitre "**Qu'est-ce que MATLAB et GNU Octave ?**".

GNU Octave se compose d'un **noyau** de base (Octave Core, **http://www.gnu.org/software/octave/**) ainsi que d'extensions implémentées sous la forme de **packages** (concept analogue aux *toolboxes* MATLAB) distribués via la plateforme **SourceForge** (**http://octave.sourceforge.net/**). Nous décrivons ci-après son installation sur les trois systèmes d'exploitation principaux :

- **GNU/Linux**
- **Windows**
- **Mac OS X**

0.2 Installation de Octave sous GNU/Linux

0.2.0 Généralités sur l'installation de Octave sous GNU/Linux

Sous Linux, l'installation de Octave est très simple et "classique", ce logiciel étant né dans le monde Unix où il est depuis longtemps "packagé" pour la plupart des distributions Linux (paquets *.deb, *.rpm...). Il en est de même que pour Gnuplot (ancien backend graphique de Octave) ainsi que d'autres outils annexes, tous distribués via les "dépôts" (*repositories*) standards de ces distributions.

De façon générale, les étapes de base d'installation de Octave sous Linux consistent donc à installer, au moyen du gestionnaire de paquetages de votre distribution (`apt` sous Debian et dérivés tels que Ubuntu, `yum` sous RedHat et Fedora...), le noyau de base Octave (paquet `octave` et ses dépendances), les *packages* Octave-Forge dont vous avez besoin (généralement packagés sous le nom `octave-`*`package`*), et optionnellement `gnuplot` (qui vient en général automatiquement en dépendance de `octave`). Il reste bien entendu possible de compiler/installer des packages Octave-Forge au sein même de Octave (voir chapitre "**Packages Octave-Forge**").

Pour un aperçu des portages Octave sur les différentes distributions Linux, voyez le **wiki Octave**. La distribution la plus utilisée dans notre faculté étant Ubuntu, c'est sur celle-ci que nous nous concentrons dans le chapitre qui suit.

0.2.1 Installation et configuration de Octave sous Ubuntu

Introduction

Les différentes versions de **Octave** et de ses outils, pour les releases Ubuntu récents, sont :

- Ubuntu **14.04 LTS** (Trusty Tahr) : Octave 3.8.1 | backends FLTK et Gnuplot 4.6.4 | Octave-GUI beta NEW

- Ubuntu **14.10** (Utopic Unicorn) : Octave 3.8.1 | backends FLTK et Gnuplot 4.6.5 | Octave-GUI beta
- Ubuntu **15.04** (Vivid Vervet) : Octave 3.8.2 | backends FLTK et Gnuplot 4.6.6 | Octave-GUI beta
- Ubuntu **15.10** (Wily Werewolf) : Octave 4.0.0 | backends Qt NEW , FLTK et Gnuplot 4.6.6 | Octave-GUI final NEW
- Ubuntu **16.04 LTS** (X... X...) : à venir...

Installation de GNU Octave 4.0.0 sous Ubuntu 14.04 à 15.04 avec le packaging *alternatif* de Mike Miller

Installation

Comme GNU Octave 4.0 ne fait son apparition sous Ubuntu que depuis la version 15.10, sous Ubuntu 15.04 et antérieur nous vous proposons la **procédure** suivante basée sur le packaging backport de **Mike Miller** :

1. définition du dépôt alternatif : `sudo add-apt-repository ppa:octave/stable`
2. mise à jour de la BD de packages : `sudo apt-get update`
3. installation proprement dite de Octave : `sudo apt-get install octave`
4. installation de la documentation Octave : `sudo apt-get install octave-doc octave-htmldoc octave-info`

A ce stade, vous disposerez de : **Octave Core 4.0.0** (mais sans packages Octave-Forge), le **GUI/IDE**, les backends graphiques **Qt**/OpenGL, **FLTK**/OpenGL et **Gnuplot**, ainsi que la **documentation** Octave.

Si vous désirez installer des **packages Octave**, ne faites **surtout pas** cela à partir des dépôts Canonical, car ces packages sont anciens et dépendent de l'ancienne version Octave (offerte par ces dépôts), et la version plus récente d'Octave que vous venez d'installer pourrait être automatiquement désinstallée ! Donc procédez plutôt ainsi :

5. installation des header-files et de l'outil mkoctfile : `sudo apt-get install liboctave-dev`
6. spécifiquement en vue de l'installation de certains package Octave, faites :
   - pour miscellaneous : `sudo apt-get install units`
   - pour octcdf : `sudo apt-get install libnetcdf-dev libnetcdfc7 netcdf-bin`
   - pour strings : `sudo apt-get install libpcre3-dev`
   - en outre, pour les les formats utiles soient disponibles pour les fonctions `saveas` et `print` : `sudo apt-get install epstool`
7. au sein de Octave, vous pourriez maintenant installer individuellement les packages souhaités avec la commande `pkg install -forge package` ;
   mais si vous souhaitez installer "à la volée" une 40aine de packages Octave-Forge parmi les plus utiles :
   - récupérez sur votre machine le programme **instal_octaveforge_packages_400_ubuntu.m** que nous avons élaboré
   - ATTENTION : pour que ces packages soient installés proprement en faveur de tous les comptes/utilisateurs de votre machine, lancez maintenant Octave depuis une fenêtre terminal en mode **super-utilisateur** avec la commande : `sudo octave --no-gui`
   - puis au sein d'Octave exécutez le programme téléchargé en frappant : `instal_octaveforge_packages_400_ubuntu`
   - notez que cela va durer 10 à 15 minutes, et que vous verrez défiler passablement de warnings (rien de grave, donc n'y prenez pas garde)
8. si l'installation de ces packages se déroule normalement, elle devrait s'achever avec le message "*** *Installation terminée, sortie normale du programme !* ***" ;
   si ce n'est pas le cas, la cause la plus probable est la surcharge du site SourceForge (qui ne répond pas dans les délais) ; re-exécutez l'étape 7 ci-dessus autant de fois que nécessaire jusqu'à ce que l'installation se termine avec le message correct
9. si tout s'est bien déroulé, en passant dans Octave la commande `pkg list` vous deviez voir tous les packages installés en mode auto-load (nom du package suivi d'une étoile), à l'exception des packages suivants que notre installation n'a à dessein pas mis en mode auto-load :
   - dataframe : générerait des erreurs quand on utilise fonction 'plot'
   - nan : ce package masquerait beaucoup de fcts de statistiques
   - windows : package non relevant sous Linux
   - tsa : masque 1 fonction de Octave Core
   - specfun : masque plusieurs fonctions de Octave Core
10. en utilisateur normal (non-superutilisateur), si vous recevez le message "*error: permission denied ; ignoring octave_exception while preparing to exit*" lorsque vous quittez Octave, faites ceci :
    - sous Octave passez la commande `history_file` , et notez le *chemin_et_fichier* indiqué (il s'agit du fichier de l'historique des commandes Octave)

- puis quittez Octave, et sous Linux passez la commande `sudo chmod ugo+rw chemin_et_fichier`

Configuration de Octave sous Linux et autres remarques

a. Si vous utilisez le backend graphique **Gnuplot** et que la **barre d'icônes** est absente au haut de la fenêtre, introduisez, dans votre prologue Octave `~/.octaverc`, la commande : `setenv('GNUTERM','wxt')`

b. Ce point n'est utile que si vous utilisez Octave-CLI (Octave en mode commande dans une fenêtre terminal) : sachez que l'**éditeur** de M-files configuré par défaut est emacs ; si vous souhaitez plutôt utiliser l'éditeur **Gedit** (éditeur de texte standard sous GNOME), il vous faut :
   - introduire, dans votre prologue Octave `~/.octaverc`, la commande : `EDITOR('gedit')`
   - ensuite, si l'éditeur Gedit ne fait pas de coloration syntaxique, activez-la simplement avec: View > Highlight Mode > Scientific > Octave
   - nous vous recommandons d'installer les plugins Gedit et configurer proprement cet éditeur (voir nos "**Conseils relatifs à l'éditeur Gedit sous Linux**")

Installation de GNU Octave 4.x sous Ubuntu ≥ 15.10 avec le packaging standard de Canonical

GNU Octave 4.0 fait officiellement son apparition sous Ubuntu depuis la version 15.10 (Wily Werewolf). Il sera donc possible de l'installer directement à partir des dépôts de Canonical. La procédure *devrait* être (nous ne l'avons pas encore testée) sensiblement la même que pour **Octave 3.8.x sous Ubuntu 13.04 à 15.04**. Nous vous en dirons plus le moment venu...

0.2.2 Anciennes versions de Octave-Forge pour Linux

On donne ici pour mémoire les liens vers les descriptions et documentations d'installation d'anciennes versions Octave : packaging **Octave 3.6.1 de Sam Miller** pour Ubuntu 12.04 | packaging **Octave 3.6.4 de Mike Miller** pour Ubuntu 10.04 à 12.10 | packaging **Octave 3.8.x Debian/Canonical** pour Ubuntu 13.04, 13.10, 14.04 |

0.3 Installation de Octave sous Windows

0.3.0 Généralités sur les différentes distributions Octave sous Windows

Le projet Octave a démarré sous Unix, et la plateforme de développement principale est, aujourd'hui encore, principalement Linux (ou systèmes POSIX). Octave a cependant fait l'objet de nombreux "portages" sous Windows : d'abord sous l'environnement libre d'émulation Linux **Cygwin**, puis compilé sous Microsoft **Visual Studio** C++, ensuite avec l'environnement de compilation libre **MinGW**/gcc (Minimalist GNU for Windows) (2009), pour déboucher en 2014 sur un environnement de *build* unifié **Octave MXE**. Cela explique pourquoi on trouve plusieurs distributions et méthodes d'installation Octave.

L'état des différents portages binaires de Octave sous **Windows** est décrit sur le **wiki Octave**. La situation actuelle est la suivante (été 2015) :

A. Le nouvel environnement de *build cross-platform* appelé **Octave MXE** a vu le jour en 2013 et débouché, début 2014, sur la première distribution binaire Octave 3.8 MXE pour Windows. C'est celle-ci que nous vous recommandons et dont nous décrivons l'installation au chapitre suivant.

B. Le portage "traditionnel" basé **Cygwin** que l'on installe ainsi :
- installer **Cygwin** (intégrant le compilateur C++ gcc)
- ensuite soit installer les différents packages binaires Octave pour Cygwin
  soit télécharger les packages sources de Octave et les compiler soi-même (technique nécessitant du temps, des compétences et davantage d'espace-disque)

À moins que vous utilisiez déjà Cygwin, nous ne vous recommandons pas cette méthode.

Avec l'avènement de Octave MXE, le développement des 2 distributions Octave traditionnelles pour Windows, diffusées via la plateforme open-source **SourceForge**, est arrêté depuis 2013.

0.3.1 Caractéristiques, installation et configuration de Octave 4.0.0 MXE pour Windows

Caractéristiques

Cette distribution d'Octave, qui date du 28.5.2015, intègre les composants suivants :

- noyau **GNU Octave** 4.0.0
- **39 packages** Octave-Forge pré-embarqués (voir leur installation plus bas)
- première version officielle d'Octave implémentant l'interface graphique **Octave GUI** NEW basée sur le framework/toolkit **Qt** 4.8.6, comprenant les fenêtres : commande, historique, browser de dossiers/fichiers, workspace, éditeur/debugger, documentation
- trois backends graphiques :
  - **nouveau backend** basé sur la librairie Qt **Qt** et OpenGL, devenant depuis Octave 4.0 le backend par défaut
  - backend basé sur le toolkit F **FLTK** (Fast Light Toolkit) et OpenGL
  - backend traditionnel G **Gnuplot** 4.6.7
- librairies d'algèbre linéaire **OpenBLAS** et **NetLib reference BLAS**
- outils de compilation **MinGW32** GCC 4.9.2 (nécessaires pour l'installation de packages et la compilation de oct/mex-files)
- Ghostscript 9.16 , Pstoedit 3.70 (conversion PDF/Postscript en différents formats), Fig2dev 3.2.5e (conversion de figures en différents formats), GraphicsMagick 1.3.21
- éditeur **Notepad++** 6.2.3 (utilisé par Octave CLI seulement)
- **documentation** GNU Octave aux formats PDF et HTML

Les **nouveautés** de la version 4.0 sont présentées dans **ce fichier** (que l'on peut aussi afficher avec la commande O `news`) ou le menu `News > Release Notes`

Procédure d'installation rapide

Pour vous épargner les différentes étapes d'installation/configuration décrites au chapitre suivant, nous avons réalisé pour vous un "package maison" dont la **procédure d'installation** est passablement simplifiée :

1. Téléchargez l'archive ZIP que nous vous avons préparée en cliquant sur **ce lien** (338 MB)

2. Déballez celle-ci à la **racine** de votre disque `C:\` en faisant souris-droite Extraire tout... dans `C:\` **sans** spécifier de dossier ! Il créera alors automatiquement une arborescence `C:\Octave\Octave-4.0.0`
   Si vous souhaitez déballer cette archive ailleurs (pas conseillé), notez que le chemin d'accès à ce dossier ne doit **en aucun cas** contenir de caractère espace (donc `C:\Program Files (x86)\` ne conviendrait par exemple pas !)
3. Vous trouvez dans `C:\Octave\` un dossier de raccourcis nommé `Octave-Raccourcis`. Copiez-le sur le bureau ou dans le menu Démarrer de Windows (c'est-à-dire dans C:\ProgramData\Microsoft\Windows\Start Menu\Programs)
4. Vous pouvez maintenant lancer Octave à partir des raccourcis de ce dossier !
5. Sous Edit > Preferences > General vous pouvez encore définir le répertoire de travail par défaut de Octave-GUI

Étapes facultatives :

11. Si vous n'avez pas déballé Octave à l'emplacement indiqué ci-dessus, vous devrez :
    - mettre à jour la "cible" des raccourcis
    - passer sous Octave les commandes suivantes (faute de quoi les packages Octave-Forge pré-installés ne seront pas utilisables) :
    `pkg rebuild -auto` puis `pkg rebuild -noauto tsa specfun ltfat`
12. Quittez Octave, relancez-le, et passez la commande `pkg list` pour vous assurer que l'ensemble des packages soient visibles et autoloadés (à l'exception des 3 packages précités)

Vous constaterez que nous avons intégré, dans ce packaging-maison, un **prologue** spécifique (celui qui est en usage dans les salles de PCs ENAC-SSIE). Nous affichons clairement, au démarrage d'Octave, ce qui est implémenté par ce prologue.

Installation et configuration manuelle

Afficher les explications ...

Remarques sur cette version

a. Un nouveau **backend graphique**, basé sur la librairie **Qt** et s'appuyant sur **OpenGL**, fait son apparition sous Octave 4.0 et devient le backend par défaut !
   Si vous désirez utiliser plutôt **FLTK**/OpenGL ou **Gnuplot**, il faudra passer la commande `graphics_toolkit('fltk')` ou `graphics_toolkit('gnuplot')`.
   La barre d'icônes au haut de la fenêtre Gnuplot est désormais toujours présente, donc plus besoin de passer la commande `setenv('GNUTERM','wxt')`

b. Lorsque le répertoire de travail est la **racine d'un lecteur** Windows (par exemple `Z:\`), les M-files créés au cours de la session ne sont pas directement utilisables (considérés comme s'ils étaient invisibles), et il est nécessaire de relancer Octave pour qu'ils soient accessibles. Pour ne pas souffrir de ce bug, travaillez donc toujours dans un sous-répertoire (ou plus profond) de lecteur (p.ex. Z:\exos_octave).

c. Octave 3.x et 4.0 sous Windows ne supporte officiellement pas l'utilisation de **caractères accentués** dans les scripts et dans les noms de fichiers/dossiers (voir **wiki**). On se limitera donc aux caractères ASCII-7bits

d. La fonction `beep` engendrant, sur certaines cartes audio, un son ininterrompu, nous l'avons désactivée dans le prologue SSIE.

0.3.2 Anciennes versions de Octave-Forge pour Windows

On donne pour mémoire ici les liens vers les descriptions et documentations d'installation d'**anciennes versions** Octave : **2.1.42 Cygwin** | **2.1.73 Cygwin** | **3.0.1 MSVC** | **3.0.3 MSVC** | **3.2.0 MinGW** | **3.2.4 MinGW** | **3.4.2 MinGW** | 3.6.2 MinGW (très semblable à 3.6.4) | **3.6.4 MinGW** | **3.8.2-1 MXE** |

0.4 Installation de Octave sous Mac OS X

0.4.0 Généralités sur les différentes distributions Octave sous Mac OS X

Comme sous Windows, Octave a fait l'objet de différents "portages" sous OS X (voir à ce sujet le **wiki Octave**). On distingue les procédures suivantes :

A. Installer une **version pré-compilée** de Octave : cette possibilité était offerte pour Octave 3.4 et 3.8, mais il n'y a actuellement (septembre 2015) rien en ce qui concerne Octave 4.0

B. Recourir à l'un des **packages managers** du monde Mac :
   a. **Fink** (portage sur Mac OS X des outils dpkg et apt de Linux/Debian), en suivant ces **indications**
   b. **Homebrew**, en suivant ces **indications**
   c. **MacPorts** (initialement appelé DarwinPorts), en suivant ces **indications**

   À moins que vous n'utilisiez déjà l'un de ces packages managers, nous vous déconseillons cette méthode, car les systèmes de packaging nécessitent l'installation préalable de l'environnement de développement Apple **Xcode** (très lourd: 2 GB !). Dans le futur, le **Mac OS X gcc Installer** devrait pouvoir se substituer à Xcode.

C. Installer et exécuter Octave dans une **machine virtuelle** sur votre Mac

C'est actuellement (septembre 2015) la méthode C) qui, à défaut d'être simple, est la moins compliquée, et donc celle que nous présentons dans le chapitre qui suit.

0.4.1 Installation de Octave 4.0.0 dans une machine virtuelle sous Mac OS X

Il est en premier lieu nécessaire installer un logiciel de virtualisation. Nous vous recommandons le logiciel libre Oracle VM **VirtualBox**. Nous avons décrit cette procédure d'installation dans une **notice détaillée** (au chapitre 1.3: Installation sous Mac OS X).

Vous avez ensuite 2 possibilités :

A. soit vous créez vous-même, sous VirtualBox, une **machine virtuelle** (VM) et y installez un système d'exploitation complet, par exemple Linux/Ubuntu (ne nécessitant pas de licence) ou Windows (nécessitant une licence) ; puis vous installez GNU Octave à l'intérieur de cette VM en suivant les instructions ci-dessus (selon le système d'exploitation que vous avez choisi)
B. soit vous utilisez la procédure décrite ci-après qui met en place automatiquement une VM Ubuntu dédiée à Octave

Installation d'une VM Ubuntu dédiée à Octave

La procédure d'**installation** ci-après (qui dure ~30 min et nécessite une bonne connexion Internet) est référée dans le **wiki Octave** et décrite dans **cette page** :

1. Commencez donc, si ce n'est déjà fait, par installer **VirtualBox** comme indiqué ci-dessus

2. Installation du serveur X11 **XQuartz** : allez dans `Application > Utilitaires` et exécutez `X11` ; cela vous renvoie vers **http://xquartz.macosforge.org/** ; téléchargez `XQuartz-`*`version`*`.dmg` (~70 MB) et ouvrez cette image-disque ; installez le package `XQuartz.pkg` ; une fois terminé, vous pouvez démonter l'image-disque et jeter le fichier `XQuartz-`*`version`*`.dmg` ; pour vérifier que X11 fonctionne, fermez votre session, ouvrez une nouvelle session, ouvrez une fenêtre terminal et passez-y la commande `xclock &` : cela devrait démarrer le serveur X11 et afficher une horloge

3. Installation de **Vagrant** : allez sur le site **http://vagrantup.com** ; en suivant les liens "Download" puis "Mac OS X Universal 32 and 64-bit", téléchargez `vagrant_`*`version`*`.dmg` (~80 MB) et ouvrez cette image-disque ; installez le package `vagrant.pkg` ; une fois terminé, vous pouvez démonter l'image-disque et jeter le fichier `vagrant_`*`version`*`.dmg`

4. Mise en place des fichiers de **configuration** Vagrant de la VM : à la racine de votre espace utilisateur `/Users/`*`username`* (ou `~`), créez un dossier nommé `Octave400` ; téléchargez les 2 fichiers **Vagrantfile** et **bootstrap.sh** et déposez-les à l'intérieur de ce dossier
   Remarque: dans le fichier `Vagrantfile` vous pourriez adapter l'espace mémoire alloué à la VM Octave en modifiant le nombre `2048` (2 GB)

5. Installation de la **VM** : ouvrez une fenêtre terminal, placez-vous dans le dossier `Octave400` (commande: `cd ~/Octave400` ; remarque: frappez `<alt-N>` pour obtenir le caractère `~`), puis passez la commande:
   `vagrant up && vagrant ssh -c "cd /vagrant && octave --force-gui" && vagrant suspend`
   Cela va automatiquement: télécharger une image de VM Ubuntu 14.04, l'installer dans VirtualBox, démarrer cette VM, installer dans celle-ci GNU Octave, puis démarrer Octave ! En sortant de Octave, la VM sera automatiquement mise en veille. Cette VM pèsera env. 2.5 GB.

Pour **utiliser** cette VM Octave, notez ce qui suit :

a. Le démarrage de Octave (i.e. le réveil de la VM) s'effectue avec la commande décrite à l'étape 5 ; pour vous faciliter la vie, définissez dans votre prologue Mac OS X (c'est-à-dire dans le fichier `~/.bash_profile`) l'alias suivant :
   `alias octave='cd ~/Octave400 && vagrant up && vagrant ssh -c "cd /vagrant && octave --force-gui" && vagrant suspend'`
   De cette façon vous pourrez démarrer tout simplement Octave en passant, dans une fenêtre terminal, la commande: `octave`
b. Notez que sous Octave le dossier `/vagrant` dans lequel vous êtes placé au démarrage correspond, sous Mac OS X, au dossier `~/Octave400`
c. Si vous souhaitez ajouter des packages Octave-Forge, il faudra d'abord passer les commandes suivantes : `vagrant up` puis `vagrant ssh` puis `cd /vagrant` puis `sudo apt-get install liboctave-dev` (outils de développement et header-files) puis `sudo octave` ; à l'intérieur de Octave pour pourrez alors installer des packages de façon classique avec: `pkg install package -forge`

0.4.2 Anciennes versions de Octave pour Mac OS X

On donne pour mémoire ici les liens vers les descriptions et documentations d'installation d'**anciennes versions** Octave : **Octave 3.8.0-6beta** sous Mac OS X 10.9 | **Octave 3.6.4 via Fink** sous Mac OS X 10.8 | **Octave.app 3.4.0** sous Mac OS X ≤ 10.7 |

0.4.3 Installation d'un éditeur de programmation pour Mac OS X

Octave GUI intègre désormais un très bon éditeur. Si vous souhaitez cependant disposer d'un éditeur de programmation indépendant sous Mac OS X, lisez les indications ci-après.

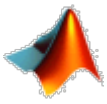
Installation de l'éditeur libre Atom

Atom est un éditeur de programmation et IDE récent qui est libre, multiplateforme et très évolutif, élaboré par la société GitHub Inc. Son installation sous OS X s'effectue comme suit :

1. Téléchargement à partir du site **atom.io** , lien Download For Mac
2. Puis ouvrir l'archive **atom-mac.zip**, et glisser le dossier "Atom" dans le dossier **Applications**
3. Un peu de configuration afin que cet éditeur soit accessible avec la commande `edit` sous Octave-CLI (Octave en mode commande dans une fenêtre terminal) :
   - dans une fenêtre terminal, passer la commande : `sudo mkdir /usr/local/bin` (répertoire qui ne semble bizarrement pas préexister sous OS X 10.9)
   - puis lancer Atom, et faire : Atom > Install Shell Commands (qui va créer un lien symbolique dans le répertoire ci-dessus)
   - dans votre prologue Octave `/Users/votre_username/.octaverc` , insérez la commande : `EDITOR('atom')`
4. Dans Atom, activation de la coloration syntaxique MATLAB/Octave :
   - sous Atom > Preferences > Packages , saisir `Octave` dans le champ "Search packages", puis sélectionner le package "Language Matlab - MATLAB/Octave language support for Atom" et cliquer sur Install
   - pour que ce ne soit pas le langage Objective C qui soit activé avec les M-files MATLAB/Octave, sous Atom > Preferences , saisir `Objective` dans le champ "Filter packages", puis sélectionner le package "Language Objective C" et cliquer sur Disable

Installation de l'éditeur gratuit TextWrangler

Si vous êtes intéressé par cet éditeur : Afficher les explications ...



1. Notions de base MATLAB et GNU Octave

1.1 Introduction

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

MATLAB

MATLAB est un logiciel commercial de **calcul numérique/scientifique**, **visualisation** et **programmation** performant et convivial développé par la société **The MathWorks Inc**. À ne **pas** confondre cependant avec les outils de calcul symbolique ou formel (tels que 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** de dimension quelconque (**tableaux** n-D, pouvant se réduire à des matrices 2D, vecteurs et scalaires) qui ne nécessitent ni déclaration de type ni dimensionnement (*typage dynamique*). Contrairement aux langages de programmation classiques (scalaires), les **opérateurs** et **fonctions** MATLAB permettent de manipuler directement ces tableaux (donc la plupart du temps sans programmer de boucles), 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** de 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 les systèmes d'exploitation standards (Windows, GNU/Linux, Mac OS X...). Le champ d'application 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 plus spécialisées regroupées au sein de dizaines de ***toolboxes*** (boîtes à outils qui sont des collections de *M-files*) couvrant des domaines nombreux et variés tels que :

- analyse de données, analyse numérique
- statistiques
- 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).

Ces caractéristiques (et d'autres encore) font aujourd'hui de MATLAB un standard incontournable en milieu académique, dans la recherche et l'industrie.

GNU Octave, et autres alternatives à MATLAB

MATLAB est cependant un logiciel commercial fermé et qui coûte cher (frais de licence), même aux conditions académiques. Mais la bonne nouvelle, c'est qu'il existe des logiciels libres/open-source analogues voire même totalement **compatibles avec MATLAB**, donc gratuits et multiplateformes :

- **GNU Octave** : logiciel libre offrant la meilleure compatibilité par rapport à MATLAB (qualifiable de "clone MATLAB", surtout depuis la version **Octave 2.9/3.x** et avec les *packages* du dépôt **Octave-Forge**). Pour l'installer sur votre ordinateur personnel (Windows, GNU/Linux, Mac OS X), voyez notre page "**Installation et configuration de GNU Octave et packages Octave-Forge**".

- **FreeMat** : logiciel libre multi-plateforme plus récent mais déjà assez abouti, implémentant un sous-ensemble des fonctionnalités de MATLAB, avec un IDE comprenant: editeur/debuguer, historique, workspace tool, path tool, explorateur de fichiers, graphiques 2D/3D... Pour l'installer, suivez les instructions figurant sous **ce lien**.
- L'environnement **Scientific Python**, basé sur le langage **Python** et constitué d'un très vaste *écosystème* d'outils et bibliothèques libres, notamment : l'interpréteur interactif **IPython** et ses possibilités de "notebook", **NumPy** pour manipuler des tableaux, **SciPy** implémentant des fonctions de calcul scientifique de haut niveau, **MatPlotLib** pour réaliser des graphiques 2D au moyen de fonctions similaires à celles de MATLAB/Octave, **Mayavi** pour la visualisation 3D, l'IDE **Spyder**, etc... Voyez à ce sujet notre **support de cours** !
- **Julia**, projet récent et prometteur de développement d'un langage de programmation de haut niveau, performant et dynamique, pour le calcul numérique et scientifique, la visualisation, mais aussi à usage général.
- **Scilab** : logiciel libre analogue à MATLAB et Octave en terme de possibilités, très abouti, plus jeune que Octave mais non compatible avec MATLAB/Octave (syntaxe et fonctions différentes, nécessitant donc une réécriture des scripts).
- **SageMath** : sous une interface basée Python, il s'agit d'une combinaison de nombreux logiciels libres (NumPy, SciPy, MatPlotLib, SymPy, Maxima, GAP, FLINT, R...) destinés au calcul numérique et symbolique/formel. La syntaxe est cependant différente de celle de MATLAB/Octave.

Dans des **domaines voisins**, on peut mentionner les logiciels libres suivants :

- statistiques et grapheur spécialisé : **R** (clone de S-Plus), ...
- traitement de données et visualisation : **GDL** (clone de IDL), ...
- calcul symbolique/formel : **Maxima**, ...
- autres : voyez notre **annuaire des principaux logiciels libre** !

1.1.2 Quelques caractéristiques fondamentales de MATLAB et GNU Octave

À ce stade de la présentation, nous voulons insister sur quelques caractéristiques de base fondamentales de MATLAB et GNU Octave :

- Le langage MATLAB/Octave est **interprété**, c'est-à-dire que chaque expression est traduite en code machine au moment de son exécution. Un programme MATLAB/Octave, appelé *script* ou *M-file*, n'a donc pas besoin d'être compilé avant d'être exécuté. Si l'on recherche cependant des performances supérieures, il est possible de convertir des fonctions M-files en **P-code**, voire en code C ou C++ (avec le MATLAB Compiler). Depuis la version 6.5, MATLAB intègre en outre un JIT-Accelerator (*just in time*) qui augmente ses performances.

- MATLAB et Octave sont "**case-sensitive**", c'est-à-dire qu'ils distinguent les majuscules des minuscules (dans les noms de variables, fonctions...).
  Ex: les variables `abc` et `Abc` sont 2 variables différentes ; la fonction `sin` (sinus) existe, tandis que la fonction `SIN` n'est pas définie...

- Le **typage** est entièrement **dynamique**, c'est-à-dire que l'on n'a pas à se soucier de déclarer le type et les dimensions des variables avant de les utiliser.

- La numérotation des **indices** des éléments de tableaux débute à `1` (et non pas `0` comme dans la plupart des langages actuels : Python, C/C++, Java...)

1.2 GNU Octave versus MATLAB

GNU Octave, associé aux packages Octave-Forge, se présente donc comme le logiciel libre/open-source le plus compatible avec MATLAB. Outre l'apprentissage de MATLAB/Octave, l'un des objectifs de ce support de cours est de vous montrer les **similitudes** entre Octave-Forge et MATLAB. Il subsiste cependant quelques **différences** que nous énumérons sommairement ci-dessous. Celles-ci s'atténuent avec le temps, étant donné qu'Octave évolue dans le sens d'une toujours plus grande compatibilité avec MATLAB, que ce soit au niveau du noyau de base ou des "**packages**" Octave-Forge (voir chapitre "**Les packages Octave-Forge**") implémentant les fonctionnalités des "**toolboxes**" MATLAB les plus courantes.

Caractéristiques propres à MATLAB :

- logiciel **commercial** (payant) développé par une société (The MathWorks Inc.) et dont le code est fermé
- de nombreuses **toolboxes** commerciales (payantes) élargissent les fonctionnalités de MATLAB dans des domaines très divers
- possibilité d'éditer les **graphiques** par double-clic (éditeur de propriétés)

- fonctions permettant de concevoir des **interfaces-utilisateur graphiques** (GUI)

Caractéristiques propres à Octave-Forge :

- logiciel **libre** et **open-source** (gratuit, sous licence GPL v3) développé de façon communautaire
- logiciel *packagé* (*.deb, *.rpm...) pour la plupart des distributions GNU/Linux, et disponible sous forme de portages binaires (ou sinon compilable) sur les autres systèmes d'exploitation courants (Windows, Mac OS X)
- extensions implémentées sous forme de **packages** également libres (voir chapitre "**Les packages Octave-Forge**")
- le **caractère modulaire** de l'architecture et des outils Unix/Linux se retrouve dans Octave, et Octave **interagit facilement** avec le monde extérieur ; Octave s'appuie donc sur les composants Unix/GNU Linux, plutôt que d'intégrer un maximum de fonctionnalités sous forme d'un environnement monolithique (comme MATLAB)
- fonctionnalités graphiques s'appuyant sur différents "**backends**" (**Qt**/OpenGL, **FLTK**/OpenGL, **Gnuplot**, Octaviz... voir chapitre "**Graphiques, images, animations**") dont il découle quelques différences par rapport à MATLAB
- fonctionnalités poussées d'**historique** (rappel de commandes...)

Jusqu'à récemment, les usagers de MATLAB dédaignaient GNU Octave en raison de l'absence d'interface utilisateur graphique (GUI). Ce reproche n'est plus valable depuis la version 3.8 qui implémente, avec **Octave GUI**, un **IDE** complet (file browser, workspace, history, éditeur/debugger...).

1.3 Démarrer et quitter MATLAB ou Octave, prologues et épilogues, interface graphique

1.3.1 Démarrer et quitter MATLAB ou Octave

Lancement de MATLAB ou Octave sous **Windows** :

Vous trouvez bien entendu les raccourcis de lancement MATLAB et Octave dans le menu `Démarrer > Tous les programmes` ...

Dans les salles d'enseignement EPFL-ENAC-**SSIE** sous Windows, les raccourcis se trouvent sous :

- **MATLAB** : `Démarrer > Tous les programmes > Math & Stat > Matlab x.x > MATLAB` (interface graphique)
- **Octave** : `Démarrer > Tous les programmes > Math & Stat > Octave x.x MXE > Octave GUI` (interface graphique), respectivement `Octave Command Line` (fenêtre terminal)

Dans les salles d'enseignement EPFL-ENAC-**SGC** sous Windows, ils se trouvent sous :

- **MATLAB** et **Octave** : `Démarrer > Tous les programmes > Programmes GC > ...`

Lancement de MATLAB ou Octave sous **Linux** :

Sous **Ubuntu** avec Unity, vous disposez des lanceurs `MATLAB` et `GNU Octave` dans le "Dash" (en frappant la touche super ou touche windows). Ils démarrent ces logiciels en mode interface graphique. Vous pouvez ensuite *ancrer* ces lanceurs dans votre propre barre de lanceurs.

Lancement depuis une **fenêtre terminal** (shell) :

- **MATLAB** : la commande `matlab` démarre MATLAB en mode interface graphique.
  Pour lancer MATLAB en mode commande dans la même fenêtre terminal (p.ex. utile si vous utilisez MATLAB à distance sur un serveur Linux), faites: `matlab -nodesktop -nosplash`
- **Octave** : depuis la version 4.0, la commande `octave` (sans paramètre) lance Octave en mode interface graphique (Octave GUI).
  Pour démarrer Octave en mode commande dans une fenêtre terminal (Octave CLI) tout en bénéficiant des fonctionnalités liées à la librairie Qt (graphiques, fenêtres de dialogue...) il faut désormais frapper `octave --no-gui`. Une autre alternative plus légère consiste à utiliser l'option `--no-gui-libs` qui n'offre pas les fonctionnalités basées sur Qt mais permet d'effectuer des graphiques basés FLTK.

Si l'exécutable `matlab` ou `octave` n'est pas trouvé, complétez le `PATH` de recherche de votre shell par le chemin complet du répertoire où est installé MATLAB/Octave, ou définissez un *alias* de lancement intégrant le chemin du répertoire d'installation.

Sortie de MATLAB ou Octave :

- Dans la fenêtre de console **MATLAB** ou **Octave**, vous pouvez utiliser à choix, dans la fenêtre "Command Window", les commandes `exit` ou `quit`
- Sous **MATLAB**, vous pouvez encore utiliser le raccourcis ctrl-Q
- Sous **Octave**, vous pouvez encore utiliser le raccourci ctrl-D, ou sous Octave GUI faire `File > Exit`

1.3.2 Répertoire de travail de base, prologues et épilogues

Répertoire de travail de base

Le "**répertoire de travail de base**" est celui dans lequel MATLAB ou Octave vous place au début d'une session. Il s'agit par défaut du "répertoire utilisateur de base" (appelé *home*), donc `/home/votre_username` sous Linux, `/Users/votre_username` sous Mac OS X... Si vous désirez le changer :

- sous MATLAB, allez sous `Preferences > MATLAB > General`. S'agissant de Windows, on peut aussi éditer la propriété "Démarrer dans:" du raccourci de lancement MATLAB.
  Notez que c'est par défaut `Z:\` dans le cas des salles d'enseignement ENAC-SSIE
- sous Octave, allez sous `Edit > Preferences > General`

Prologues

Le mécanisme des "**prologues**" permet à l'utilisateur de faire exécuter automatiquement par MATLAB/Octave un

certain nombre de commandes en début de session. Il est implémenté sous la forme de **scripts** (M-files). Le prologue est très utile lorsque l'on souhaite **configurer certaines options**, par exemple :

- sous MATLAB ou Octave :
  - affichage d'un texte de bienvenue (commande `disp('texte')`)
  - ajout, dans le `path` de recherche MATLAB/Octave, des chemins de répertoires dans lesquels l'utilisateur aurait défini ses propres scripts ou fonctions (voir la commande `addpath('path1:path2:path3...')` au chapitre "**Environnement MATLAB/Octave**")
- sous Octave spécifiquement :
  - changement du prompt (invite de commande) (voir la commande `PS1` au chapitre "**Fenêtre de commandes MATLAB/Octave**")
  - changement de backend graphique (voir la commande `graphics_toolkit` au chapitre "Graphiques/**Concepts de base**")
  - choix de l'éditeur pour Octave-CLI (voir la commande `EDITOR` au chapitre "**Éditeur et debugger**")

Lorsque MATLAB démarre, il passe successivement par les échelons de prologues suivants :

- le script de démarrage système `matlabrc.m` (voir `helpwin matlabrc`)
- puis *le premier* script nommé `startup.m` qu'il trouve en parcourant le "répertoire utilisateur de base" et les différents répertoires définis dans le `path` MATLAB (voir chapitre "**Environnement MATLAB/Octave**").

Lorsque Octave démarre, il passe successivement par les échelons de prologues suivants :

- le prologue `OCTAVE_HOME/share/octave/site/m/startup/octaverc`, puis le prologue `OCTAVE_HOME/share/octave/version/m/startup/octaverc` (qui est un lien symbolique vers le fichier `/etc/octave.conf`)
- puis l'éventuel script nommé `.octaverc` se trouvant dans le "répertoire utilisateur de base"
- et enfin, si l'on démarre Octave en mode commande depuis une fenêtre terminal, l'éventuel `.octaverc` se trouvant dans le répertoire courant.
- pour (ré)exécuter ces scripts manuellement en cours de session, vous pouvez faire `source('chemin/.octaverc')`

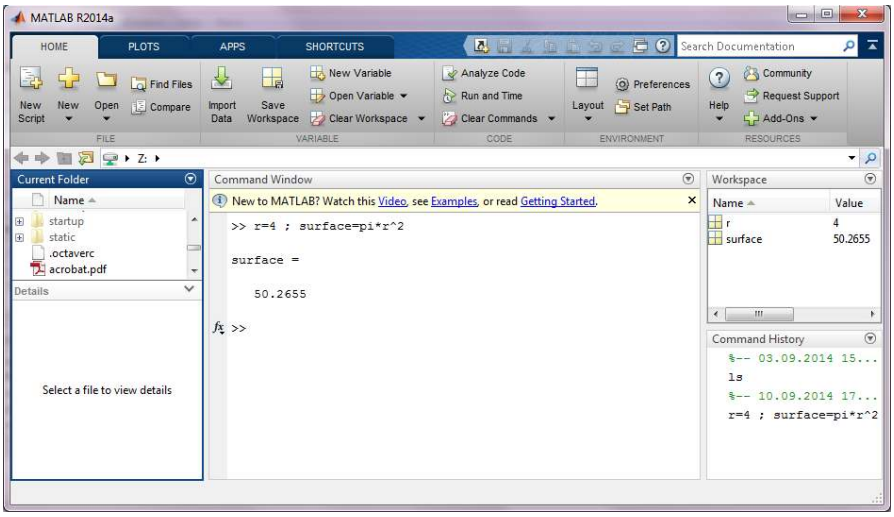
Épilogues

S'agissant des **épilogues** MATLAB et Octave, il s'agit du mécanisme automatiquement invoqué lorsque l'on termine une session :

- MATLAB exécute *le premier* script nommé `finish.m` qu'il trouve en parcourant le "répertoire utilisateur de base" puis les différents répertoires définis dans le `path` MATLAB
- Octave s'appuie sur la fonction `atexit`

1.3.3 Interface graphique et IDE de MATLAB

MATLAB intègre depuis longtemps un **IDE** (Integrated Development Environment), c'est-à-dire une interface graphique (GUI) se composant, en plus de la console de base et des fenêtres de graphiques, de diverses sous-fenêtres, d'un bandeau de menus et d'une barre d'outils (voir illustration ci-dessous).



Interface graphique et IDE de MATLAB R2014 (ici sous Windows)

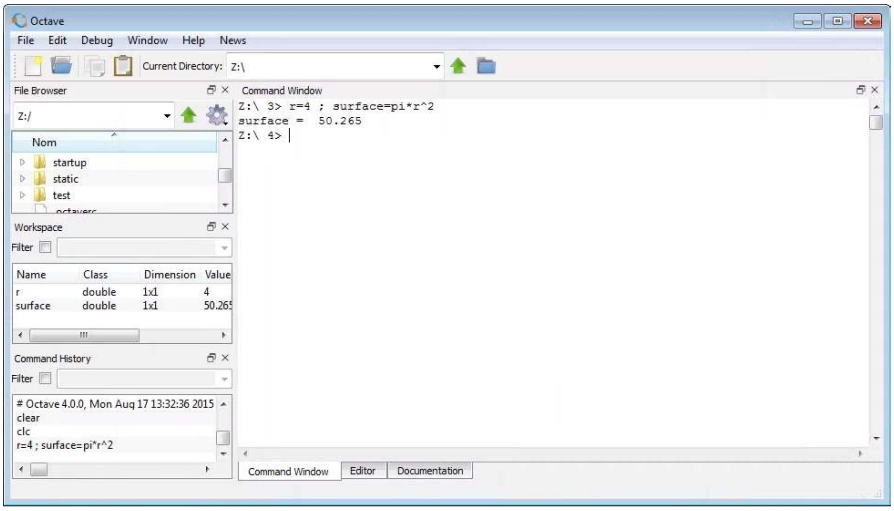
Les différents réglages MATLAB s'effectuent au moyen du bouton Preferences

Cette interface graphique offre les sous-fenêtres suivantes :

- **Command Window** : console MATLAB, c'est-à-dire fenêtre d'entrée/sortie standard
- **Current Folder** : un double-clic sur un dossier change de répertoire courant, et sur un fichier cela l'ouvre dans l'éditeur. Voyez aussi le menu contextuel droite-clic qui permet de exécuter/renommer/détruire un fichier, respectivement renommer/détruire pour un dossier... Dans la ligne d'en-tête, droite-clic permet d'activer l'affichage des attributs taille/type/date de modification.
- **Workspace** : outre l'affichage des variables du workspace (comme le ferait `whos`), il est possible d'éditer celles-ci par double-clic. En outre le menu contextuel droite-clic permet de les renommer, supprimer et grapher
- **Command History** : affiche les dernières commandes passées. Le double-clic sur une commande permet de la ré-exécuter. Pour ré-exécuter plusieurs commandes, les sélectionner (maj-clic pour sélection continue, ctrl-clic pour sélection discontinue), puis faire droite-clic Evaluate Selection . De plus les commandes sélectionnées peuvent copiées dans le presse-papier avec droite-clic Copy ou être injectées dans un nouveau script avec droite-clic Create Script
- **Editor** : voir le chapitre "**Éditeur et debugger**"
- **Help** : voir le chapitre "**Outils d'aide...**"
- **Menu bandeau** : composé de 4 onglets HOME (commandes de base), PLOTS (fonction graphiques), APPS, SHORTCUTS (raccourcis), EDITOR (édition et debugging), PUBLISH, VIEW
- **Barre d'outils** : permet essentiellement de changer de répertoire courant et lancer de recherches par nom de fichier/répertoire

1.3.4 Interface graphique et IDE de GNU Octave

GNU Octave n'a longtemps été utilisable (en usage interactif ou lancement de scripts) que depuis une fenêtre de terminal. Dès 2014 une interface graphique officielle très attendue, nommée **Octave GUI** (Octave Graphical User Interface), voit le jour sous Octave 3.8. Elle se compose, en plus de la console de base et des fenêtres de graphiques, de diverses sous-fenêtres, d'une barre de menus et d'une barre d'outils (voir illustration ci-dessous).



Interface graphique et IDE de GNU Octave 4.0 (ici sous Windows)

Les différents réglages de Octave GUI s'effectuent via le menu `Edit > Preferences`

Cette interface graphique offre les sous-fenêtres suivantes :

- **Command Window** : console Octave, c'est-à-dire fenêtre d'entrée/sortie standard
- **File Browser** :
  - Le champ supérieur indique quel est le répertoire courant. Par un menu déroulant, affichant l'historique des répertoires précédents, il permet aussi de changer de répertoire. L'icône/menu à droite permet de créer des dossiers ou fichiers, de rechercher des dossiers et fichiers par leur nom, et même de rechercher des fichiers selon leur contenu (comme `Edit > Find Files`).
  - Dans la liste en-dessous, un double-clic sur un dossier change de répertoire courant, et sur un fichier cela l'ouvre dans l'éditeur. Voyez aussi le menu contextuel droite-clic qui permet de exécuter/renommer /détruire un fichier, respectivement chercher/renommer/détruire pour un dossier... Dans la ligne d'en-tête, droite-clic permet d'activer l'affichage des attributs taille/type/date de modification.
- **Workspace** : outre l'affichage des variables du workspace (comme le ferait `whos`), les possibilités se limitent pour l'instant (Octave 4.0) à celles offertes par le menu contextuel droite-clic : renommer une variable, l'afficher (`disp`) ou la grapher (`plot` ou `stem`). Un filtre permet de restreindre l'affichage des variables dont le nom contient une chaîne.
- **Command History** : affiche les dernières commandes passées (comme le ferait `history`). Le double-clic sur une commande permet de la ré-exécuter. Pour ré-exécuter plusieurs commandes, les sélectionner (maj-clic pour sélection continue, ctrl-clic pour sélection discontinue), puis faire droite-clic `Evaluate` . De plus les commandes sélectionnées peuvent copiées dans le presse-papier avec droite-clic `Copy` ou être injectées dans un nouveau script avec droite-clic `Create script` . Un filtre permet aussi de restreindre l'affichage des commandes contenant une chaîne.
- **Editor** : voir le chapitre "**Éditeur et debugger**"
- **Documentation** : voir le chapitre "**Outils d'aide...**"
- **Barre de menus** : nous attirons notamment votre attention sur :
  - `File > Recent Editor Files` : éditer un fichier récemment ouvert
  - `Edit > Preferences` : adapter les réglages de Octave GUI
  - `Window > Reset Default Window Layout` : rétablir la disposition par défaut des sous-fenêtres
- **Barre d'outils** : outre les icônes habituelles (nouveau fichier, ouvrir, copier, coller...) on retrouve le champ/menu d'affichage/sélectionnement du répertoire courant

Pour mémoire, d'autres tentatives de développement d'interfaces graphiques à Octave (voire même d'IDE's complets) ont existé par le passé et sont encore parfois utilisées (notamment QtOctave), généralement sous Linux et parfois sous Windows : Afficher les détails ...

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

1.4.1 Aide en ligne

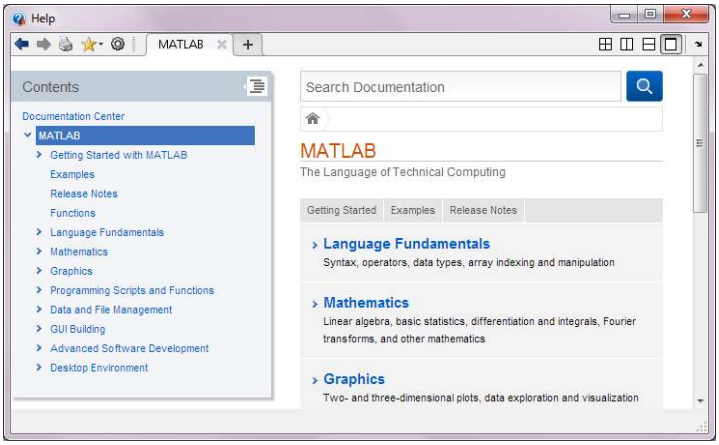
`help` *`fonction`*

Affiche, dans la fenêtre de commande MATLAB/Octave, la **syntaxe** et la **description** 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`** *`sujet`*, ou menu Help ou icône ? du bandeau MATLAB

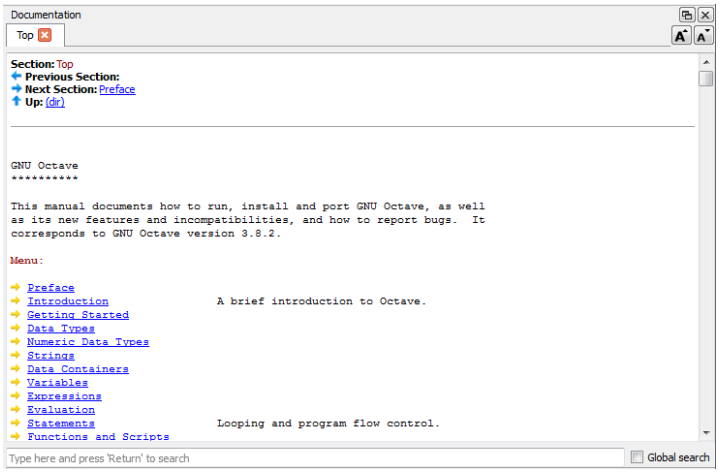
Même effet que la commande **`help`**, sauf que le résultat est affiché dans la fenêtre d'aide spécifique MATLAB "Help" (voir illustration ci-dessous)



Fenêtre d'aide MATLAB R2014 (ici sous Windows)

`doc` *`sujet`*

Sous Octave, cette commande recherche et affiche l'information relative au *sujet* désiré à partir du **manuel** Octave. Avec Octave GUI, le résultat est affiché dans la fenêtre de l'onglet "Documentation" qui offre un mécanisme de navigation par hyper-liens.



Fenêtre de documentation GNU Octave 4.0 (ici sous Windows)

`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.

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)

Accès au Manuel Octave complet (HTML)
Avec Help > Documentation > Online, ou via **ce lien**
Dans les salles ENAC-SSIE sous Windows avec Démarrer > Tous les programmes > Math & Stat > Octave x.x > Documentation, puis sous-menus HTML ou PDF
Voyez en particulier, vers la fin de la table des matières, le "Function Index" qui est un index hyper-texte de toutes les fonctions Octave

Voyez aussi ce **Octave Quick Reference Card** (aide-mémoire en 3 pages, PDF) ainsi que cette **FAQ**

1.4.2 Exemples et démos

`echodemo intro`
Lancement d'un petit didacticiel d'**introduction** à MATLAB

`demos`
Passe dans le chapitre "Examples" de l'aide MATLAB où l'on trouve des démonstrations **interactives** illustrant les capacités de MALAB. Pour chacune de ces démos le code MATLAB détaillé est présenté.

`rundemos(`*package*`)`
Lance les démos définies dans le répertoire du package spécifié (les packages sont sous `OCTAVE_HOME/share/octave/packages/`*package*)

Ex : `rundemos('signal-`*version*`')` : lance les démos du package "signal" (traitement de signaux) dans la *version* spécifiée (faites `pkg list` pour connaître le version de package installée)

1.4.3 Ressources Internet utiles relatives à MATLAB et Octave

Sites Web

- MATLAB
  - site de la société The MathWorks Inc (éditrice de MATLAB) : **http://www.mathworks.com**
  - article sur MATLAB dans Wikipedia : **français**, **anglais**
  - fonctions/scripts libres développées pour MATLAB (fonctionnant aussi souvent sous Octave) : **http://www.mathworks.com/matlabcentral/fileexchange/**
  - Wiki Book "Matlab Programming" : **http://en.wikibooks.org/wiki/MATLAB_Programming**

- GNU Octave
  - site principal consacré à GNU Octave : **http://www.octave.org** (http://www.gnu.org/software/octave/)
  - dépôt des paquets "Octave-Forge" sur SourceForge.net : **http://octave.sourceforge.net**
  - article sur GNU Octave dans Wikipedia : **français**, **anglais**
  - espace de partage de fonctions/scripts Octave : **http://agora.octave.org/** (depuis été 2012)
  - Wiki Book Octave : **http://fr.wikibooks.org/wiki/Programmation_Octave** (FR), **http://en.wikibooks.org/wiki/Octave_Programming_Tutorial** (EN)

- Packages Octave-Forge (analogues aux toolboxes MATLAB)
  - liste des packages disponibles : `pkg list -forge`
  - liste, description et téléchargement de packages : **http://octave.sourceforge.net/packages.php**
  - index des fonctions (Octave core et packages Octave-Forge) : **http://octave.sourceforge.net/function_list.html**

- Gnuplot
  - site principal Gnuplot (back-end graphique traditionnel sous Octave) : **http://gnuplot.sourceforge.net**

Forums de discussion, mailing-lists, wikis, blogs

- MATLAB
  - forum MathWorks : **http://www.mathworks.ch/matlabcentral/answers/**
  - forum Usenet/News consacré à MATLAB : **https://groups.google.com/forum/#!forum/comp.soft-sys.matlab**

- Octave
  - **wiki** Octave : **http://wiki.octave.org**
  - **forum** utilisateurs et développeurs, avec mailing lists associées : **http://octave.1599824.n4.nabble.com/** (avec sections: General, Maintainers, Dev)
  - soumission de **bugs** en relation avec **Octave core** (depuis mars 2010) : **http://bugs.octave.org** (http://savannah.gnu.org/bugs/?group=octave)
  - en relation avec **packages** Octave-Forge :
    - soumission de **bugs** : **http://sourceforge.net/p/octave/bugs/**
    - demande de **fonctionnalités** : **http://sourceforge.net/p/octave/feature-requests/**
  - **blogs** en relation avec le développement de Octave :
    - planet.octave : **http://planet.octave.org/**
    - maintainers@octave.org : **http://blog.gmane.org/gmane.comp.gnu.octave.maintainers/**

La commande `info` affiche sous Octave différentes sources de contact utiles : mailing list, wiki, packages, bugs report...

1.5 Types de nombres (réels/complexes, entiers), variables, expressions, 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, donc **64 bits**). 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 de définir des réels en virgule flottante "**simple précision**", donc stockés sur des variables occupant 2 fois moins d'espace en mémoire (4 octets c-à-d. 32 bits), donc de précision deux fois moindre (7 chiffres décimaux significatifs, et une étendue allant de 10^{-38} à 10^{+38}). On utilise pour cela la fonction de conversion `single(` *nombre | variable* `)` , ou en ajoutant le paramètre `'single'` à certaines fonctions telles que `ones` , `zeros` , `eye` ... De façon inverse, la fonction de conversion `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 cela en testant vos variables 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

Si vous lisez des données numériques réelles **à partir d'un fichier texte** et désirez les stocker en **simple précision**, utilisez la fonction `textscan` avec le format `%f32` (32 bits, soit 4 octets). Le format `%f64` est synonyme de `%f` et génère des variables de double précision (64 bits, soit 8 octets).

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 :

- les fonctions de conversion `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 de conversion `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.

Ex :
- `int8(-200)` retourne -128 (valeure minimale signée pour int8), `int8(-4.7)` retourne -5, `int8(75.6)` retourne 76, `int8(135)` retourne 128 (valeure maximale signée pour int8)
- `uint8(-7)` retourne 0 (valeure minimale non signée pour int8), `uint8(135.2)` retourne 135, `uint8(270)` retourne 255 (valeure maximale non signée pour int8)
- si `a=uint8(240)` , `a/320` retourne 0, alors que `single(a)/320` retourne 0.75000
- la série `indices1=int8(1:100)` occupe 8x moins de place en mémoire (100 octets) que la série `indices2=1:100` (800 octets)
- `4.6 * ones(2,2,'int16')` retourne une matrice de dimension 2x2 remplie de chiffres 5 stockés chacun sur 2 octets (entiers 16 bits)

Si vous lisez des données numériques entières **à partir d'un fichier texte** et désirez les stocker sur des **entiers** et non pas sur des réels double précision, utilisez la fonction `textscan` avec l'un des formats suivants :

- entiers signés : `%d8` (correspondant à `int8`), `%d16` (correspondant à `int16`), `%d32` ou `%d` (correspondant à `int32`), `%d64` (correspondant à `int64`)
- entiers non signés (positifs) : `%u8` (correspondant à `uint8`), `%u16` (correspondant à `uint16`), `%u32` ou `%u` (correspondant à `uint32`), `%u64` (correspondant à `uint64`)

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, respectivement pour la partie réelle et 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 Conversion de nombres de la base 10 dans d'autres bases

Notez que les nombres, dans d'autres bases que la base 10, sont ici considérés comme des chaînes (*str_binaire*, *str_hexa*, *str_baseB*) !

- décimal en binaire, et vice-versa : *str_binaire*= `dec2bin(nb_base10)` , *nb_base10*= `bin2dec(str_binaire)`
- décimal en binaire, et vice-versa : *str_hexa*= `dec2hex(nb_base10)` , *nb_base10*= `hex2dec(str_hexa)`

- décimal dans base B, et vice-versa : *str_baseB*= `dec2base(nb_base10, B)` , *nb_base10*= `base2dec(str_baseB, B)`

1.5.5 Généralités sur les variables et expressions

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 MATLAB/Octave**"). Comme déjà dit, le langage MATLAB ne requiert **aucune déclaration** préalable de **type** de variable et de **dimension** de tableau/vecteur (typage dynamique). Lorsque MATLAB/Octave rencontre un nouveau nom de variable, il crée automatiquement la variable correspondante et y associe l'espace mémoire nécessaire dans le workspace. Si la variable existe déjà, MATLAB/Octave change son contenu et, si nécessaire, lui alloue un nouvel espace mémoire 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: noms de variables valides : `x_min` , `COEFF55a` , `tres_long_nom_de_variable`

Ex: 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 donc **case-sensitive** (distinction des majuscules et minuscules).

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

Pour se référer à un **ensemble de variables** (principalement avec commandes `who` , `clear` , `save` ...), on peut utiliser les **caractères de substitution** `*` (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 MATLAB/Octave**".

`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" (effet du caractère `;` terminant l'instruction) 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`

a) `[var1, var2, ... varn] = deal(v1, v2, ... vn)`
b) `[var1, var2, ... varn] = deal(v)`

La fonction `deal` permet d'affecter plusieurs variables en une seule instruction. Il faut que l'on ait exactement le même nombre d'éléments à gauche et à droite de l'expression (forme a)), ou que l'on ait un seul élément à droite (forme b)), sinon `deal` retourne une erreur.

Ex: `[a, b, c] = deal(11, 12, 13)` est identique à `a=11, b=12, c=13`
`[a, b, c] = deal(10)` est identique à `a=b=c=10`
`[coord(1,:), coord(2,:)] = deal([100,200], [110,210])` donne `coord=[100, 200 ; 110, 210]`

`variable`

Affiche le contenu de la *variable* spécifiée

`who {variable(s)}`

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

`whos {variable(s)}`

Affiche une liste plus détaillée que `who` de toutes les variables couramment définies dans le workspace (ou de la (des) *variable(s)* spécifiée(s)) : 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 !)

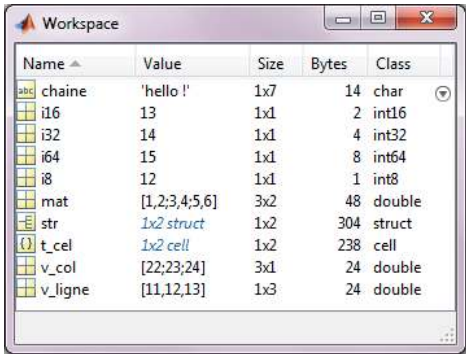
`clear {variable(s)}`

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

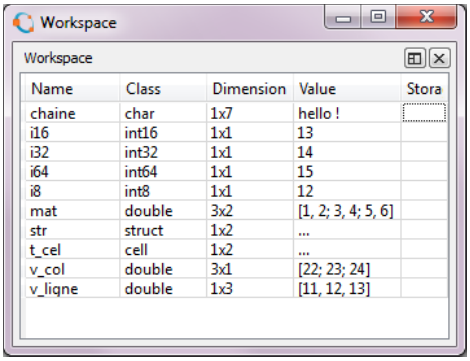
Ex: `clear mat*` détruit toutes les variables dont le nom commence par "mat"

M Layout > Show > Workspace
O Window > Show Workspace

Affichage de la fenêtre "Workspace" MATLAB ou Octave GUI (illustrations ci-dessous) qui présente, à la façon de la commande `whos`, toutes les variables courantes du workspace et qu'il est possible de manipuler avec le menu contextuel droite-clic (renommer, détruire... voire même éditer sous MATLAB par double-clic)



Workspace browser de MATLAB (ici sous Windows)



Workspace browser de Octave GUI (ici sous Windows)

1.5.6 Généralités sur les chaînes de caractères

Il est bien entendu possible de manipuler du **texte** (des "**chaînes**" de caractères) dans MATLAB/Octave. Lorsque l'on utilise une chaîne, celle-ci doit être délimite par deux **apostrophes**, voire des guillemets sous Octave.

`string = 'chaîne de caractères'`
`string = "chaîne de caractères"`

Stocke la *chaîne de caractères* sur la variable *string* qui sera un **vecteur-ligne** de type **char**.
Si la chaîne contient des apostrophes, il faut les dédoubler (sinon ils seraient interprétés 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 dans l'utilisation des chaînes, voir plus loin le chapitre dédié aux "**Chaînes de caractères**" ainsi que l'introduction aux "**Tableaux cellulaires**".

1.5.7 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 par des "toolboxes" MATLAB ou des "packages" Octave, ou encore par vos propres M-files et packages.
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 est 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).

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

L'utilisateur a donc la possibilité de créer ses propres fonctions au moyen de M-files (voir chapitre "**Fonctions**").

1.6 Fenêtre de commandes MATLAB/Octave

1.6.1 Généralités

La fenêtre "Command Window" apparaît donc automatiquement dès que MATLAB ou Octave est démarré (voir illustrations plus haut). Nous présentons ci-dessous quelques commandes permettant d'agir sur cette fenêtre.

`more on|off`
Activation ou désactivation du mode de défilement "paginé" (contrôlé) dans la fenêtre "Command Window". Par défaut le défilement n'est pas paginé dans MATLAB (`off`). Sous Octave, cela dépend des versions. Dans Octave, cette commande positionne la valeur retournée par la fonction built-in `page_screen_output` (respectivement à `0` pour `off` et `1` pour `on`).

En mode paginé, on agit sur le défilement avec les touches suivantes :
• **MATLAB**: enter pour avancer d'une ligne, espace pour avancer d'une page, q pour sortir (interrompre l'affichage)
• **Octave**: mêmes touche que pour MATLAB, avec en outre: curseur-bas et curseur-haut pour avancer/reculer d'une ligne ; PageDown ou f, resp. PageUp ou b pour avancer/reculer d'une page ; 1 G pour revenir au début ; *nombre* G pour aller à la *nombre*-ième ligne ; G pour aller à la fin ; `/chaîne` pour rechercher *chaîne* ; n ou N pour recherche respectivement l'occurrence suivante ou précédente de cette *chaîne* ; h pour afficher l'aide du pagineur

`PS1('specification')`
Changement du prompt primaire de Octave ("invite" dans la fenêtre de commande Octave)
La *specification* est une chaîne pouvant notamment comporter les séquences spéciales suivantes :
`\w` : chemin complet (path) du répertoire courant
`\#` : numéro de commande (numéro incrémental)
`\u` : nom de l'utilisateur courant
`\H` : nom de la machine courante
Ex: la commande `PS1('\w \#> ')` modifie le prompt de façon qu'il affiche le répertoire courant suivi d'un espace puis du numéro de commande suivi de ">" et d'un espace

`clc` ou `home`
M Clear Commands > Command Window
Edit > Clear Command Window
`clc` efface le contenu de la fenêtre de commande (**cl**ear **c**ommand window), et positionne le curseur en haut à gauche
`home` positionne le curseur en haut à gauche, sans effacer la fenêtre de commande sous MATLAB

`format loose|compact`

Activation ou suppression de l'affichage de lignes vides supplémentaires dans la fenêtre de commande (pour une mise en plage plus ou moins aérée). MATLAB et Octave sont par défaut en mode `loose`, donc affichage de lignes vides activé

1.6.2 Caractères spéciaux dans les commandes MATLAB et Octave

La commande `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 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 `#`	• 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 . 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 caractères 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 ne trouvez pas la 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

1.6.3 Rappel et édition des commandes, copier/coller

L'usage des **touches de clavier** suivantes permet de rappeler, éditer et exécuter des commandes MATLAB/Octave passées précédemment :

Touche	Description
curseur-haut et curseur-bas	rappelle la ligne précédente / suivante
curseur-gauche et curseur-droite	déplace le curseur d'un caractère à gauche / à droite
ctrl-curseur-gauche et ctrl-curseur-droite	déplace le curseur d'un mot à gauche / à droite
home et end	déplace le curseur au début / à la fin de la ligne
backspace et delete	détruit le caractère à gauche / à droite du curseur
ctrl-k	détruit les caractères depuis le curseur jusqu'à la fin de la ligne
escape	efface entièrement la ligne
enter	exécute la commande courante

Voir en outre, en ce qui concerne **Octave**, le mécanisme de l'**historique** au chapitre "**Workspace**".

Pour **copier/coller** du texte (commandes, données...) dans la fenêtre de commandes, MATLAB et Octave offrent les mêmes possibilités mais avec une interface différente.

Fonction	**MATLAB** (Win/Lin/Mac)	**Octave GUI** (Win/Lin/Mac)	**Octave CLI Windows**	**Octave CLI Linux** (X-Window)	**Octave CLI MacOS**
Copier la sélection dans le "presse-papier"	Edit > Copy ou ctrl-C (sur Mac cmd-C)	Edit > Copy ou ctrl-C (sur Mac cmd-C)	Sélectionner, puis enter. Ou sélectionner puis bouton clic-droite	La **sélection** courante est automatiquement copiée dans le "presse-papier"	Edit > Copy ou cmd-C Si on a une souris à 3 boutons, on peut aussi utiliser la technique Linux
Coller le contenu du "presse-papier" à la position courante du curseur d'insertion	Edit > Paste ou ctrl-V (sur Mac cmd-V)	Edit > Paste ou ctrl-V (sur Mac cmd-V)	Bouton clic-droite Pour que cela fonctionne, le raccourci de lancement Octave doit être correctement configuré (voir chapitre "**Installation de Octave sous Windows**")	Bouton clic-milieu	Edit > Paste ou cmd-V Si on a une souris à 3 boutons, on peut aussi utiliser la technique Linux

1.6.4 Extension automatique ("completion") de noms de variables/fonctions /fichiers...

La fenêtre "Command Window" MATLAB/Octave offre en outre (comme dans les shell Unix) un mécanisme dit de

"**commands, variables & files completion**" : lorsque l'on entre un nom de fonction/commande, de variable ou de fichier, il est possible de ne frapper au clavier que les premiers caractères de celui-ci, puis utiliser la touche tab pour demander à MATLAB/Octave de compléter automatiquement le nom :

- M s'il y a une ambiguité avec une autre commande/fonction/variable (commençant par les mêmes caractères), MATLAB affiche alors directement un "menu déroulant" contenant les différentes possibilités ; on sélectionne celle que l'on souhaite avec curseur-haut ou curseur-bas, puis on valide avec enter
- O si Octave ne complète rien, c'est qu'il y a une ambiguité avec une autre commande/fonction/variable (commençant par les mêmes caractères) : on peut alors compléter le nom au clavier, ou frapper une seconde fois tab pour qu'Octave affiche les différentes possibilités (et partiellement compléter puis represser tab ...)

1.6.5 Formatage des nombres dans la fenêtre de commandes

Dans tous les calculs numériques, MATLAB/Octave travaille toujours de façon interne en précision maximum, c'est-à-dire en **double précision** (voir plus haut).

On peut choisir le format d'**affichage des nombres** dans la fenêtre "Command Window" à l'aide de la commande `format` :

Commande	Type d'affichage	Exemple
`format {short {e}}`	Affichage par défaut : notation décimale fixe à 5 chiffres significatifs Avec option `e` => notation décimale flottante avec exposant	72.346 7.2346e+001
`format long {e}`	Affichage précision max : 15 chiffres significatifs Avec option `e` => avec exposant	72.3456789012345 7.23456789012345e+001
`format bank`	Format monétaire (2 chiffres après virgule)	72.35
`format hex`	En base hexadécimale	4052161f9a65ee0f
`format rat`	Approximation par des expressions rationnelles (quotient de nombres entiers)	3.333... s'affichera 10/3

O Sous Octave seulement, on peut activer/désactiver le mécanisme d'affichage de vecteurs/matrices précédé ou non par un "facteur d'échelle". Toujours activé sous MATLAB, ce mécanisme n'est pas activé par défaut sous Octave.

Ex : la fonction `logspace(1,7,5)` affichera par défaut, sous Octave :

```
  1.0000e+01   3.1623e+02   1.0000e+04   3.1623e+05   1.0000e+0
```

mais si on se met dans le mode `fixed_point_format(1)`, elle affichera (comme sous MATLAB) :

```
  1.0e+07  *
  0.00000  0.00003  0.00100  0.03162  1.00000
```

Remarquez le "facteur d'échelle" (de multiplication) `1.0e+07` de la première ligne.

Pour un contrôle plus pointu au niveau du formatage à l'affichage, voir les fonctions `sprintf` (string print formated) et `fprintf` (file print formated) (par exemple au chapitre "**Entrées-sorties**").

1.7 Les packages Octave-Forge

Les "**packages**" sont à Octave ce que les "**toolboxes**" sont à MATLAB. C'est à partir de la version 2.9.12 que l'architecture d'Octave implémente complètement les packages (Octave étant auparavant beaucoup plus monolithique).

Tous les packages Octave-Forge sont recensés et disponible en téléchargement via le *dépôt* (*repository*) officiel **http://octave.sourceforge.net/packages.php**. L'installation et l'utilisation d'un package consiste à :

1. le télécharger (depuis le site ci-dessus) => fichier de nom `package-version.tar.gz`
2. l'installer (une fois pour toutes) ; au cours de cette opération, les fichiers constituant le package seront "compilés" puis mis en place
3. le charger (s'il n'est pas en mode "autoload") dans le cadre de chaque session Octave où l'on veut l'utiliser

Les étapes 1. et 2. peuvent être combinées avec la nouvelle option `-forge` (commande `pkg install -forge package`).

Si vous désirez **savoir dans quel package** est implémentée une **fonction** de nom donné (en vue d'installer ce package), vous pouvez consulter la liste des fonctions **http://octave.sourceforge.net/function_list.html** (catégorie "alphabetical"). Le nom du package est spécifié entre crochets à coté du nom de la fonction.

S'agissant des packages **installés**, la commande `which fonction` vous indiquera dans quel package ou quel *oct-file* la fonction spécifiée est implémentée, ou s'il s'agit d'une fonction *built-in*.

Nous décrivons ci-dessous les **commandes de base** relatives à l'installation et l'usage de packages Octave (voir `help pkg` pour davantage de détails).

`pkg list`
Cette commande affiche la liste des packages **installés**. Outre le nom de chaque package, on voit en outre si ceux-ci sont chargés (signalé par `*`) ou non, leur numéro de version et leur emplacement.
Avec [*USER_PACKAGES*, *SYSTEM_PACKAGES*]= `pkg('list')` on stocke sur 2 tableaux cellulaires la liste et description des packages installés respectivement de façon locale (utilisateur courant) et globale (tous les utilisateurs de la machine)

`pkg list -forge`
Affiche la liste des packages Octave disponibles **sur SourceForge** (nécessite connexion Internet)

`pkg describe {-verbose} package | all`
Affiche une description du *package* spécifié (resp. de tous les packages installés).
Avec l'option `-verbose`, la liste des fonctions du package est en outre affichée.

`news package`
Affiche le document *release notes* du *package* spécifié (i.e. les nouveautés apportées par chaque version)

`pkg load|unload package | all`
Cette commande charge (resp. décharge) le *package* spécifié (resp. tous les packages installés). De façon interne le chargement, qui rend "visibles" les fonctions du *package*, consiste simplement à ajouter au path Octave l'emplacement des fichiers du *package*.
Pour ne charger que les packages installés en mode "autoload", on peut faire `pkg load auto`

a) `pkg install {-local|-global} {-auto} {-verbose} package-version.tar.gz`
b) `pkg install {...} -forge package`
a) Installe le *package* spécifié à partir du fichier `package-version.tar.gz` préalablement téléchargé
b) Installe le *package* spécifié en le téléchargeant directement depuis SourceForge (nécessite connexion Internet)

• Si Octave a été démarré en mode super utilisateur (avec `sudo octave` sous Linux ou OS X), le *package* est installé par défaut de façon globale (i.e. pour tous les utilisateurs de la machine), à moins de spécifier l'option `-local`. Si Octave a été démarré depuis un compte non privilégié, l'installation s'effectue par défaut pour l'utilisateur courant (i.e. déposé dans le dossier `octave` se trouvant dans le "profile" de l'utilisateur), à moins de spécifier l'option `-global`. Dans le "profile" également, un fichier `.octave_packages` répertorie les packages locaux de l'utilisateur.

• Avec l'option `-auto`, le package est installé en mode "autoload", c'est-à-dire qu'il sera disponible, dans les sessions Octaves ultérieures, sans devoir le "charger".
• L'option `-verbose` est utile pour mieux comprendre ce qui se passe quand l'installation d'un package pose problème.

Notez que la plupart des packages étant écrits en C/C++, leur installation par cette procédure requiert la présence d'**outils de compilation** (p.ex. MinGW sous Windows, Apple XCode sous OS X...)

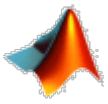
`pkg uninstall package`
Désinstallation du *package* spécifié

`pkg update`
Tente de mettre à jour l'ensemble des packages à partir de SourceForge (nécessite connexion Internet)

a) `pkg rebuild`
b) `pkg rebuild -noauto package(s)`
a) Cette commande reconstruit la base de donnée des packages à partir des répertoires de packages trouvés dans l'arborescence d'installation. Si l'on déplace le dossier d'installation Octave, il est ensuite nécessaire le lancer cette commande.
b) En plus de la reconstruction de la base de donnée des packages, on désactive ici l'autoload de certains *package(s)* à partir des prochaines sessions

Quelques remarques finales concernant le packaging Octave sous **GNU/Linux** :

- le package communément appelé "`octave`" proposé sur les dépôts (repositories) des différentes distributions Linux (Debian, Ubuntu, RedHat, Fedora...) ne contient plus que le Octave core (noyau Octave), donc sans les extensions/packages Octave-Forge (voir chapitre "**Installation de Octave-Forge sous GNU/Linux**")
- ⚠ dans la mesure où vous n'avez pas installé Octave vis un dépôt alternatif (PPA), avant de tenter d'installer des "packages Octave" selon la technique décrite ci-dessus, commencez par voir si le dépôt de votre distribution GNU/Linux ne propose pas le(s) *package(s)* que vous cherchez, sous le nom "`octave-package`" (c'est le cas de l'architecture Octave sous Ubuntu depuis Ubuntu 9.04)



2. Workspace, environnement, commandes générales

2.1 Workspace MATLAB/Octave

2.1.1 Sauvegarde et restauration du workspace et de variables

Les **variables** créées au cours d'une session MATLAB/Octave (interactivement depuis la fenêtre de commande MATLAB/Octave ou en exécutant des M-files...) résident en **mémoire** dans ce que l'on appelle le "**workspace**" (espace de travail). A moins d'être sauvegardées sur disque dans un "**MAT-file**", les variables sont perdues lorsque l'on termine la session.

Les MAT-files sont des fichiers **binaires** de *variables* qui sont identifiables sous MATLAB par leur extension `*.mat` (à ne pas confondre avec les "M-files" qui sont des fichiers-texte de *scripts* ou de *fonctions* et qui ont l'extension `*.m`), alors qu'avec Octave ils n'ont par défaut pas d'extension.

Sous Octave, le **format et version** des MAT-files dépend de la valeur affectée à la fonction built-in Octave `save_default_options` :

- lorsque Octave est démarré avec l'option `--traditional`, la valeur de "save_default_options" est `-mat-binary` qui désigne le format binaire de workspaces MATLAB V6 et supérieur
- mais lorsque Octave est démarré sans option particulière, la valeur de "save_default_options" est `-text`, ce qui veut dire que les fichiers de variables sont sauvegardés dans un format texte propre à Octave et non lisible par MATLAB ; dans ce cas il peut être fort utile, si l'on jongle souvent entre Octave et MATLAB, de changer ce réglage en définissant la commande suivante dans son prologue Octave : `save_default_options('-mat-binary')`
- une autre possibilité consiste à spécifier explicitement le format lorsque l'on passe la commande `save` (voir ci-dessous), par exemple: `save -mat-binary MAT-file.mat`

`save {format et option(s)} MAT-file {variable(s)}`, ou Save Workspace ou File > Save Workspace As

Sauvegarde, dans le *MAT-file* spécifié, toutes les variables définies et présentes en mémoire, ou seulement de la(les) *variable(s)* spécifiées.

- **MATLAB** : Si l'on ne spécifie pas de nom de *MAT-file*, cette commande crée un fichier de nom `matlab.mat` dans le répertoire courant. Si l'on spécifie un nom sans extension, le fichier aura l'extension `.mat`. Si le *MAT-file* spécifié existe déjà, il est écrasé, à moins que l'on utilise l'option `-append` qui permet d'ajouter des variables dans un fichier de workspace existant. Sans spécifier d'option particulière, MATLAB V7 utilise un nouveau format binaire `-V7` spécifique à cette version.
- **Octave** : Il est nécessaire de spécifier un nom de *MAT-file*. Si l'on ne spécifie pas d'extension, le fichier n'en aura pas (donc pas d'extension `.mat`, contrairement à MATLAB => nous vous conseillons de prendre l'habitude de spécifier l'extension `.mat`). Sans spécifier d'option particulière, Octave 3 utilise le format défini par la fonction built-in `save_default_options` (voir plus haut)
- Le paramètre `format` peut notamment prendre l'une des valeurs suivantes :
  - `-v6` ou `-mat-binary` : format binaire MATLAB V6 (double précision)
  - (pas d'option) ou `-mat7-binary` ou `-v7` : format binaire MATLAB V7 (double précision)
  - `-ascii` : format texte brute (voir plus bas)
  - `-binary` : format binaire propre à Octave (double précision)
  - `-text` : format texte propre à Octave

`load MAT-file {variable(s)}`, ou Import Data ou File > Load Workspace

Charge en mémoire, à partir du *MAT-file* spécifié, toutes les variables présentes dans ce fichier, ou seulement

celles spécifiées.

- **MATLAB** : Il n'est pas besoin de donner l'extension `.mat` lorsque l'on spécifie un *MAT-file*. Si l'on ne spécifie pas de *MAT-file*, cette commande charge le fichier de nom `matlab.mat` se trouvant dans le répertoire courant.

`who{s} {variable(s)} -file MAT-file`
Permet, sous MATLAB, de lister les variables du *MAT-file* spécifié plutôt que celles du workspace courant
Sous Octave, on ne peut pas spécifier de variables

Au cours d'une longue session MATLAB (particulièrement lorsque l'on créée/détruit de gros vecteurs/matrices), l'espace mémoire (workspace) peut devenir très **fragmenté** et empêcher la définition de nouvelles variables. Utiliser dans ce cas la commande ci-dessous.

`pack`
Défragmente/consolide l'espace mémoire du workspace (*garbage collector* manuel). MATLAB réalise cela en sauvegardant toutes les variables sur disque, en effaçant la mémoire, puis rechargeant les variables en mémoire. Cette fonction existe aussi sous Octave pour des raisons de compatibilité avec MATLAB mais ne fait rien de particulier.

2.1.2 Sauvegarde et chargement de données numériques via des fichiers-texte

Lorsqu'il s'agit d'**échanger** des **données numériques** entre MATLAB/Octave et d'autres logiciels (tableur/grapheur, logiciel de statistique, SGBD...), les MAT-files standards ne conviennent pas, car sont des fichiers binaires. Outre les fonctions que l'on verra au chapitre "**Entrées-sorties formatées**", une solution consiste à utiliser la commande `save` avec l'option `-ascii` qui permet de sauvegarder des variables numériques MATLAB/Octave sur des **fichiers-texte** (ASCII). Notez que cette technique ne convient donc pas pour manipuler des chaînes, tableaux > 2D, tableaux cellulaires, structures.

`save -ascii {-double} fichier_texte variable(s)`
Sauvegarde, sur le *fichier_texte* spécifié (qui sera écrasé s'il existe déjà), la (les) *variable(s)* spécifiée(s). Les nombres sont écrits en notation scientifique avec **8 chiffres** significatifs, à moins d'utiliser l'option `-double` qui écrit alors en double précision (**16 chiffres** significatifs). Les vecteurs-ligne occupent 1 ligne dans le fichier, les vecteurs colonnes et les matrices plusieurs lignes. Lorsqu'une ligne comporte plusieurs nombres, ceux-ci sont délimités par des espace, à moins d'utiliser l'option `-tabs` qui insère alors des caractères tab. Les chaînes de caractères sont écrites sous forme de nombres (succession de codes ASCII pour chaque caractère).
Il est fortement **déconseillé** de sauvegarder simultanément **plusieurs variables**, car ce format de stockage ASCII ne permet pas de les différencier facilement les unes des autres (l'exemple ci-dessous est parlant !).

Ex : Définissons les variables suivantes :
`nb=123.45678901234 ; vec=[1 2 3] ; mat=[4 5 6;7 8 9] ; str='Hi !' ;`
La commande `save -ascii fichier.txt` génèrera alors grosso modo (différences, entre MATLAB et Octave, dans l'ordre des variables !) le fichier ci-dessous. N'ayant ici pas spécifié de noms de variable dans la commande, toutes les variables du workspace sont écrites (`mat` , `nb` , `str` , `vec`) :

```
 4.0000000e+000  5.0000000e+000  6.0000000e+000
 7.0000000e+000  8.0000000e+000  9.0000000e+000
 1.2345679e+002
 7.2000000e+001  1.0500000e+002  3.2000000e+001  3.3000000e+001
 1.0000000e+000  2.0000000e+000  3.0000000e+000
```

a) `load {-ascii} fichier_texte`
b) `variable = load('fichier_texte') ;`
Charge sur un vecteur ou un tableau 2D les données provenant du *fichier_texte* spécifié. Celles-ci ne peuvent être que **numériques** (pas de chaînes) et **délimitées** par un ou plusieurs espace ou tab. Chaque ligne de données du fichier donnera naissance à une ligne du tableau. Il doit donc y avoir exactement le **même nombre de données** dans toutes les lignes du fichier !
a) Le chargement s'effectue sur une variable de nom identique au nom du fichier (mais sans son extension). L'option `-ascii` est facultative (le mode de lecture ASCII étant automatiquement activé si le fichier spécifié est de type texte).
b) Les données sont chargées sur la *variable* de nom spécifié.

Important : notez bien que si `save -ascii` permet de sauvegarder plusieurs variables d'un coup, la fonction `load` quant à elle ne permet de charger ces données que sur une seule variable.

Voici une **technique alternative** offrant un petit peu plus de finesses (délimiteur...) :

`dlmwrite(`*`fichier_texte, variable, {délimiteur {, nb_row {, nb_col } } } )`*

Sauvegarde, sur le *fichier_texte* spécifié (qui est écrasé s'il existe déjà), la *variable* spécifiée (en général une matrice). Utilise par défaut, entre chaque colonne, le séparateur `,` (virgule), à moins que l'on spécifie un autre *délimiteur* (p.ex. `';'`, ou `'\t'` pour le caractère tab). Ajoute éventuellement (si spécifié dans la commande) *nb_col* caractères de séparation au début de chaque ligne, et *nb_row* lignes vides au début du fichier.
Voir aussi la fonction analogue `csvwrite` (sous Octave dans le package "io").

variable `= dlmread(`*`fichier_texte, {délimiteur {, nb_row {, nb_col } } } )`*

Charge, sur la *variable* spécifiée, les données numériques provenant du *fichier_texte* indiqué. S'attend à trouver dans le fichier, entre chaque colonne, le séparateur `,` (virgule), à moins que l'on spécifie un autre *délimiteur* (p.ex. `';'`, ou `'\t'` pour le caractère tab). Avec les paramètres *nb_row* et *nb_col*, on peut définir le cas échéant à partir de quelle ligne et colonne (numérotés dans ce cas à partir de zéro et non pas 1 !) il faut récupérer les données.
Voir aussi la fonction analogue `csvread` (sous Octave dans le package "io").

Un dernier truc simple pour **récupérer des données numériques** (depuis un fichier texte) sur des variables MATLAB/Octave consiste à enrober 'manuellement' ces données dans un **M-file** (script) et l'exécuter.

Ex : **A)** Soit le fichier de données `fich_data.txt` ci-dessous contenant les données d'une matrice, que l'on veut charger sur `M`, et d'un vecteur, que l'on veut charger sur `V` :

```
1   2   3
4   5   6
9   8   7

22  33  44
```

B) Il suffit de renommer ce fichier en un nom de script `fich_data.m`, y intercaler les lignes (en gras ci-dessous) de définition de début et de fin d'affectation :

```
M = [ ...
      1   2   3
      4   5   6
      9   8   7
] ;

V = [ ...
      22  33  44
] ;
```

C) Puis exécuter ce script sous MATLAB/Octave en frappant la commande `fich_data`

On voit donc, par cet exemple, que le caractère newline a le même effet que le caractère `;` pour délimiter les lignes d'une matrice.

Pour manipuler directement des **feuilles de calcul** binaires (classeurs) **OpenOffice.org Calc** (ODS) ou **MS Office Excel** (XLS), mentionnons encore les fonctions suivantes :

- sous Matlab et Octave : MS Excel : `xlsopen`, `xlsclose`, `xlsfinfo`, `xlsread`, `xlswrite`
- spécifiquement sous Octave : OpenOffice/LibreOffice Calc : `odsopen`, `odsclose`, `odsfinfo`, `odsread`, `odswrite`
- spécifiquement sous Octave : MS Excel: `oct2xls`, `xls2oct` ; OpenOffice/LibreOffice Calc: `oct2ods`, `ods2oct`

Finalement, pour réaliser des **opérations plus sophistiquées** de lecture/écriture de **données externes**, on renvoie le lecteur au chapitre "**Entrées-sorties formatées**" présentant d'autres fonctions MATLAB/Octave plus pointues (telles que `textread`, `fscanf`, `fprintf` ...)

2.1.3 Journal de session MATLAB/Octave

Les commandes présentées plus haut ne permettent de sauvegarder/recharger que des variables. Si l'on veut **sauvegarder les commandes** passées au cours d'une session MATLAB/Octave ainsi que l'output produit par ces commandes, on peut utiliser la commande `diary` qui crée un "journal" de session dans un fichier de type texte. Ce serait une façon simple pour créer un petit script MATLAB/Octave ("M-file"), c'est-à-dire un fichier de commandes MATLAB que l'on pourra exécuter lors de sessions ultérieures (voir chapitre "**Generalités**" sur les M-files). Dans cette éventualité, lui donner directement un nom se terminant par l'extension "`*.m`", et n'enregistrer alors dans ce fichier que les commandes (et pas leurs résultats) en les terminant par le caractère `;`

`diary {fichier_texte} {on}`
: MATLAB/Octave enregistre, dès cet instant, toutes les commandes subséquentes et leurs résultats dans le *fichier_texte* spécifié. Si ce fichier existe déjà, il n'est pas écrasé mais complété (mode append). Si l'on ne spécifie pas de fichier, c'est un fichier de nom "diary" dans le répertoire courant (qui est par défaut "Z:\" en ce qui concerne les salles d'enseignement ENAC-SSIE) qui est utilisé. Si l'on ne spécifie pas `on`, la commande agit comme une bascule (activation-désactivation-activation...)

`diary off`
: Désactive l'enregistrement des commandes subséquentes dans le *fichier_texte* précédement spécifié (ou dans le fichier "diary" si aucun nom de fichier n'avait été spécifié) et ferme ce fichier. Il faut ainsi le fermer pour pouvoir l'utiliser (le visualiser, éditer...)

`diary`
: Passée sans paramètres, cette commande passe de l'état `on` à `off` ou vice-versa ("bascule") et permet donc d'activer/désactiver à volonté l'enregistrement dans le journal.

2.1.4 Historique Octave

Indépendemment du mécanisme standard de "journal", **Octave** gère en outre un **historique** en enregistrant automatiquement, dans le répertoire profile (Windows) ou home (Unix) de l'utilisateur, un fichier `.octave_hist` contenant toutes les commandes (sans leur output) qui ont été passées au cours de la session et des sessions précédentes. Cela permet, à l'aide des commandes habituelles de rappel et édition de commandes (curseur-haut, etc...), de retrouver des commandes passées lors de sessions précédentes. En relation avec cet "historique", on peut utiliser les commandes suivantes :

`history {-q} {n}`
: Affiche la liste des commandes de l'historique Octave. Avec l'option `-q`, les commandes ne sont pas numérotées. En spécifiant un nombre `n`, seules les `n` dernières commandes de l'historique sont listées.

`run_history n1 {n2}`
: Exécute la `n1` -ème commande de l'historique, ou les commandes `n1` à `n2`

ctrl-R
: Permet de faire une recherche dans l'historique

`history_size(0)`
: Effacera tout l'historique lorsqu'on quittera Octave

Clear Commands > Command History
Edit > Clear Command History
: Efface instantanément l'historique de commandes

Différentes fonctions built-in Octave permettent de paramétrer le mécanisme de l'historique (voir l'aide) :

- `history_file` : emplacement et nom du fichier historique (donc par défaut `.octave_hist` dans le profile ou home de l'utilisateur)
- `history_size` : taille de l'historique (nombre de commandes qui sont enregistrées, par défaut 1024)

2.2 Environnement MATLAB/Octave

2.2.1 Généralités

MATLAB et Octave procèdent de la façon suivante lorsqu'ils évaluent les instructions d'un M-File ou ce que saisit l'utilisateur. Si l'utilisateur fait par exemple référence au nom "xxx" :

1. MATLAB/Octave regarde si "xxx" correspond à une variable du **workspace**
2. s'il n'a pas trouvé, il recherche un M-file nommé "xxx.m" (script ou fonction) dans le **répertoire courant** de l'utilisateur
3. s'il n'a pas trouvé, il parcourt, dans l'ordre, les différents répertoires définis dans le "**path de recherche**" MATLAB/Octave (`path`) à la recherche d'un M-file nommé "xxx.m", donc en passant revue les **toolboxes**/**packages**
4. s'il n'a pas trouvé, il regarde s'il y a une fonctions **built-in** ainsi nommée
5. finalement si rien n'est trouvé, MATLAB/Octave affiche une **erreur**

Cet ordre de recherche entraîne que les définitions réalisées par l'utilisateur priment sur les définitions de base de MATLAB/Octave !

Ex: si l'utilisateur définit une variable `sqrt=444` , il ne peut plus faire appel à la fonction MATLAB/Octave `sqrt` (racine carrée) ; pour `sqrt(2)` , MATLAB rechercherait alors le 2e élément du vecteur `sqrt` qui n'existe pas, ce qui provoquerait une erreur ; pour restaurer la fonction `sqrt` , il faut effacer la variable avec `clear sqrt` .

Il ne faut, par conséquent, jamais créer de variables ayant le même nom que des fonctions MATLAB/Octave prédéfinies. Comme MATLAB/Octave est case-sensitive et que pratiquement toutes les fonctions sont définies en minuscules, on évite ce problème en mettant par exemple en majuscule le 1er caractère du nom pour des variables qui pourraient occasionner ce genre de conflit.

2.2.2 Path de recherche

Le "**path de recherche**" MATLAB/Octave définit le "chemin d'accès" aux différents répertoires où sont recherchés les scripts et fonctions invoqués, qu'il s'agisse de M-files de l'utilisateur ou de fonctions MATLAB/Octave built-in ou de toolboxes/packages. Les commandes ci-dessous permettent de visualiser/modifier le path, ce qui est utile pour pouvoir accéder à vos propres fonctions implémentées dans des M-files situés dans un autre répertoire que le répertoire courant.

Pour que vos adaptations du path de recherche MATLAB/Octave soient **prises en compte** dans les **sessions ultérieures**, il est nécessaire de placer ces commandes de changement dans votre **prologue** MATLAB/Octave (voir chapitre "**Démarrer et quitter MATLAB ou Octave**"). Elles seront ainsi automatiquement appliquées au début de chaque session MATLAB/Octave.

Rappelons encore que le path est automatiquement modifié, sous Octave, lors du chargement/déchargement de packages (voir chapitre "**Packages**").

{*variable* =} `path`
: Retourne le "path de recherche" courant.
Voir aussi la fonction `pathdef` qui retourne le path sous forme d'une seule chaîne (concaténation de tous les paths)

`addpath('chemin(s)' {,-end})`
: Cette commande **ajoute**, en tête du path de recherche courant (ou en queue du path si l'on utilise l'option `-end`), le(s) *chemin(s)* spécifié(s), pour autant qu'ils correspondent à des répertoires existants.
Ex (ici sous Windows): `addpath('Z:\fcts','Z:\fcts bis')`

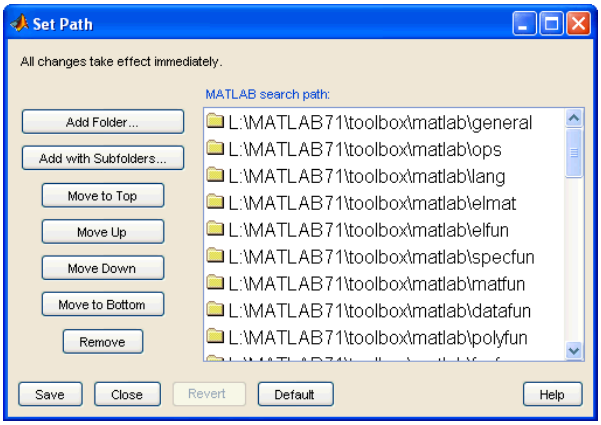
`rmpath('chemin1'{,'chemin2'...})`
: Supprime du path de recherche MATLAB/Octave le(s) *chemin(s)* spécifié(s).
Ex (ici sous Windows): `rmpath('Z:\mes fcts')`

`genpath('chemin')`
: Retourne le path formé du *chemin* spécifié et de tous ses sous-répertoires (récursivement).
Ex (ici sous Unix): `addpath(genpath('/home/dupond/mes_fcts'))` ajoute au path de recherche

courant le dossier `/home/dupond/mes_fcts` **et** tous ses sous-dossiers !

`pathtool` ou Set Path

Affichage de la fenêtre MATLAB "Set Path" (voir illustration ci-dessous) qui permet de voir et de modifier avec une interface utilisateur graphique le path de recherche.



Path Browser MATLAB 7

`path('chemin1'{,'chemin2'})`

Commande dangereuse (utiliser plutôt `addpath`) qui **redéfinirait** (écraserait) entièrement le path de recherche en concaténant les paths *chemin1* et *chemin2*. Retourne une erreur si *chemin1* et/ou *chemin2* ne correspondent pas à des répertoires existants.

Ex (ici sous Unix): `path(path,'/home/dupond/mes_fcts')` : dans ce cas ajoute, en queue du path de recherche courant, le chemin `/home/dupond/mes_fcts`. Si le chemin spécifié était déjà défini dans le path courant, la commande `path` ne l'ajoute pas une nouvelle fois.

a) `which M-file|fonction` ou `which('M-file|fonction')`
b) `which fichier`

a) Affiche le chemin et nom du *M-file* spécifié ou dans lequel est définie la *fonction* spécifiée.
b) Affiche le chemin du *fichier* spécifié.

`type fonction` ou `type fichier`

Affiche le contenu de la fonction *fonction*.m ou du *fichier* spécifié

2.2.3 Répertoire courant

Lorsque l'on manipule sous MATLAB/Octave des fichiers sans préciser leur chemin d'accès, ils sont bien entendu recherchés dans le "répertoire courant" ou "répertoire de travail". Lorsque l'on invoque un script ou fonction, MATLAB/Octave commence également par chercher le M-file éponyme dans le répertoire courant, et ceci parce que le 1er chemin figurant dans le path de recherche est toujours "`.`" (désignant le répertoire courant) ; en cas d'échec il parcourt les autres répertoires indiqués dans le path de recherche.

Les **commandes** en relation avec la gestion du répertoire courant sont les suivantes :

{*variable* =} `pwd`

Retourne le chemin d'accès du **répertoire courant**

`cd {chemin}`

Change de répertoire courant en suivant le *chemin* (absolu ou relatif) spécifié. Si ce *chemin* contient des espaces, ne pas oublier de l'entourer d'apostrophes. Notez que, passée sans spécifier de *chemin*, cette commande affiche sous MATLAB le chemin du répertoire courant, alors que sous Octave elle renvoie l'utilisateur dans son répertoire home.

Ex:
- `cd mes_fonctions` : descend d'un niveau dans le sous-répertoire "mes_fonctions" (chemin relatif)
- `cd ..` : remonte d'un niveau (chemin relatif)
- sous Windows: `cd 'Z:\fcts matlab'` : passe dans le répertoire spécifié (chemin absolu)
- sous Unix: `cd /home/dupond` : passe dans le répertoire spécifié (chemin absolu)

2.3 Commandes MATLAB/Octave en relation avec le système d'exploitation

Remarques préliminaires concernant les commandes ci-dessous :

- lorsque l'on doit spécifier un *fichier*, on peut/doit faire précéder le nom de celui-ci par un *chemin* si le fichier n'est pas dans le répertoire courant
- le séparateur de répertoires est `\` sous Windows (bien que la plupart des commandes acceptent le `/`), et `/` sous Linux ou Mac OS X ;
  ci-dessous, on utilisera partout `\` dans un esprit de simplification

`dir {chemin\}{fichier(s)}`
`ls {chemin\}{fichier(s)}`

Affiche la liste des fichiers du répertoire courant, respectivement la liste du(des) *fichier(s)* spécifiés du répertoire courant ou du répertoire défini par le *chemin* spécifié.
• On peut aussi obtenir des informations plus détaillées sur chaque fichiers en passant par une *structure* avec l'affectation `structure = dir`
• Sous Octave, la présentation est différente selon que l'on utilise `dir` ou `ls`. En outre avec Octave sous Linux, on peut faire `ls -l` pour un affichage détaillé à la façon Unix (permissions, propriétaire, date, taille...)

`readdir('chemin')`
`glob('{chemin\}pattern')`

Retourne, sous forme de vecteur-colonne cellulaire de chaînes, la liste de tous les fichiers/dossiers du répertoire courant (ou du répertoire spécifié par *chemin*). Avec `glob`, on peut filtrer sur les fichiers dont le nom correspond à la *pattern* indiquée (dans laquelle on peut utiliser le caractère de substitution `*`)

[*status, msg_struct, msg_id*] = `fileattrib('fichier')`
attr_struct = `stat('fichier')`

Retourne, sous forme de structure *msg_struct* ou *attr_sctuct*, les informations détaillées relatives au *fichier* spécifié (permissions, propriétaire, taille, date...)

`what {chemin}`

Affiche la liste des fichiers MATLAB/Octave (M-files, MAT-files et P-files) du répertoire courant (ou du répertoire défini par le *chemin* spécifié)

`type {chemin\}fichier`

Affiche le contenu du *fichier*-texte spécifié.

`copyfile('fich_source', 'fich_destin')`

Effectue une copie du *fich_source* spécifié sous le nom *fich_destin*

a) `movefile('fichier', 'nouv_nom_fichier')` ou `rename('fichier','nouv_nom_fichier')`
b) `movefile('fichier', 'chemin')`

a) Renomme le *fichier* spécifié en *nouv_nom_fichier*
b) Déplace le *fichier* dans le répertoire spécifié par *chemin*

`delete fichier(s)`
`unlink('fichier')`

Détruit le(s) *fichier(s)* spécifié(s)

`mkdir('sous-répertoire')` ou `mkdir sous-répertoire`

Crée le sous-*répertoire* spécifié

`rmdir('sous-répertoire')` ou `rmdir sous-répertoire`

Détruit le sous-*répertoire* spécifié pour autant qu'il soit vide

a) `open fichier.m` ou `open('fichier.m')`

b) *structure* = `open('fichier.mat')`
c) `open fichier.ext` ou `open('fichier.ext')`
Ouvre le *fichier* spécifié avec l'application appropriée :
a) Dans le cas d'un M-file, celui-ci est ouvert dans l'éditeur MATLAB/Octave
b) Dans le cas d'un fichier de workspace `.mat`, ses variables sont chargées sur les différents champs de la *structure* spécifiée
c) Si l'extension *.ext* est autre, c'est l'application externe propre au fichier qui est invoquée. Par exemple pour un fichier `.html`, celui-ci est ouvert dans la navigateur web par défaut. Sous Windows, s'il s'agit d'un fichier `.exe`, il est exécuté en tant qu'application.

[*status*, *output*] = `system('commande du système d'exploitation')`
M `! commande du système d'exploitation { & }`
La *commande* spécifiée est passée à l'interpréteur de commandes du système d'exploitation, et l'output de celle-ci est affiché dans la fenêtre de commande MATLAB (ou, en ce qui concerne MATLAB, dans une fenêtre de commande Windows si l'on termine la commande par le caractère `&`) ou sur la variable *output*
Ex (ici pour Windows) : M `! rmdir répertoire` : détruit le sous-*répertoire* spécifié

[*status*, *output*] = `dos('commande {&}' {,'-echo'})`
Sous Windows, exécute la *commande* spécifiée du système d'exploitation (ou un programme quelconque) et affecte sa sortie standard à la variable *output* spécifiée. Sans l'option `-echo`, la sortie standard de la commande n'est pas affichée dans la fenêtre de commande MATLAB
Ex :
`dos('copy fich_source fich_destin')` : copie *fich_source* sous le nom *fich_destin*
`[status, output]=dos('script.bat');` affecte à la variable "output" la sortie de "script.bat"

[*status*, *output*] = `unix(...)`
Sous Unix, commande analogue à la commande `dos`...

[*output*, *status*] = `perl(script {, param1, param2 ...})`
[*output*, *status*] = O `python(script {, param1, param2 ...})`
Exécute le *script* Perl ou Python spécifié en lui passant les arguments *param1*, *param2*...

`computer`
Retourne sur une chaîne le *type de machine* sur laquelle on exécute MATLAB/Octave. On y voit notamment apparaître le système d'exploitation.

`version`
Retourne sur une chaîne le *numéro de version* de MATLAB/Octave que l'on exécute

`ver`
Retourne plusieurs lignes d'information : version de MATLAB/Octave, système d'exploitation, liste des toolboxes MATLAB installées, respectivement des packages Octave installés

O `get_home_directory`
Retourne le chemin du répertoire de base de l'utilisateur

`matlabroot`
O `OCTAVE_HOME`
Retourne le chemin de la racine du dossier où est installé MATLAB ou Octave

variable = `getenv('variable_environnement')`
Retourne la valeur de la *variable d'environnement* indiquée. Les noms de ces variables, propres au système d'exploitation, sont généralement en majuscule.
O Il ne faut pas confondre ces variables d'environnement système avec les variables spécifiques Octave produites (depuis Octave 2.9) par des **fonctions** built-ins, p.ex: `OCTAVE_VERSION`, `EDITOR` ...

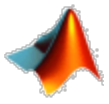
Ex: `getenv('USERNAME')` affiche le nom de l'utilisateur (variable d'environnement système)

`setenv('variable_environnement', 'valeur')`
O `putenv('variable_environnement', 'valeur')`
Définition ou modification d'une *variable d'environnement*.

Ex: sur Mac avec Octave `setenv('GNUTERM','x11')` change l'environnement graphique de Gnuplot de "aqua" à "x11" (X-Window)

status = `unsetenv('variable_environnement')`

Détruit la *variable d'environnement*. Le *status* de retour sera 0 si cela s'est bien passé.



3. Scalaires, constantes, opérateurs et fonctions de base

3.1 Scalaires et constantes

MATLAB/Octave ne différencie fondamentalement pas une matrice à N-dimension (tableau) 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)
`exp(1)` ou `e`	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`	Environ $2.2e^{-308}$: le plus petit nombre positif utilisable (en virgule flottante double précision)
`realmax`	Environ $1.7e^{+308}$: le plus grand nombre positif utilisable (en virgule flottante double précision)
`eps`	Environ $2.2e^{-16}$: 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. Correspond à `1` , mais notez que n'importe quelle valeur numérique différente de `0` ou chaîne non vide est aussi assimilée à "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. Correspond à `0` . Une chaîne vide est aussi assimilée à "faux"

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 le résultat d'une expression qui n'a pas été affectée à une variable, agissant donc comme 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 M `help ops` décrit l'ensemble des opérateurs et caractères spéciaux sous MATLAB/Octave.

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 `plus(v1, v2, v3 ...)`	Addition	4
O {var2=} `++var1` O {var2=} `var1++`	Pré- ou post-incrémentation (sous Octave seulement) : • 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;` Si *var1* est un tableau, agit sur tous ses éléments.	
`-` ou `minus(v1, v2)`	Soustraction	4
O {var2=} `--var1` O {var2=} `var1--`	Pré- ou post-décrémentation (sous Octave seulement) : • 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 tableau, agit sur tous ses éléments.	
`*` ou `mtimes(v1, v2, v3 ...)`	Multiplication	3
`/` ou `mrdivide(v1, v2)`	Division standard	3
`\` ou `mldivide(v1, v2)`	Division à gauche Ex: `14/7` est équivalent à `7\14`	3
`^` ou O `**` ou `mpower(v1, v2)`	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** classique : 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`.

Les opérateurs de test ci-dessous "pourraient" 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 `help 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 `help 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 tableaux, retourne un tableau (rappel: 0 ET 0 => 0 ; 0 ET 1 => 0 ; 1 ET 1 => 1)
expression1 `&&` *expression2*	**ET** logique "short circuit". A la différence de `&` ou `and`, cet opérateur est plus efficace, car il ne prend le temps d'évaluer *expression2* que si *expression1* est vraie. En outre: - sous Octave: retourne un scalaire même si les *expressions* sont des tableaux - ✗ sous MATLAB: n'accepte pas que les *expressions* soient des tableaux
expression1 `|` *expression2* `or` (*expression1*, *expression2*)	**OU** logique. Si les *expressions* sont des tableaux, retourne un tableau (rappel: 0 OU 0 => 0 ; 0 OU 1 => 1 ; 1 OU 1 => 1)
expression1 `||` *expression2*	**OU** logique "short circuit". A la différence de `|` ou `or`, cet opérateur est plus efficace, car il ne prend le temps d'évaluer *expression2* que si *expression1* est fausse. En outre: - sous Octave: retourne un scalaire même si les *expressions*

	sont des tableaux - ✗ sous MATLAB: n'accepte pas que les *expressions* soient des tableaux
`xor` (*expr1*, *expr2* {, *expr3*...})	**OU EXCLUSIF** logique (rappel: 0 OU EXCL 0 => 0 ; 0 OU EXCL 1 => 1 ; 1 OU EXCL 1 => 0)
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]
- `A & B` ou `and(A,B)` retourne le vecteur [0 0 0 1]
- O `A || B` ne fonctionne ici (avec des vecteurs) que sous Octave, et retourne le scalaire 0
- O `A && B` ne fonctionne ici (avec des vecteurs) que sous Octave, et retourne le scalaire 0

3.3 Fonctions de base

3.3.1 Fonctions mathématiques

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

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)` `log10(var)` `log2(var)`	Logarithme naturel de *var* (de base **e**), respectivement de base **10**, et de base **2** Ex: `log(exp(1))` => 1, `log10(1000)` => 3, `log2(8)` => 3
`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)`	Génération de nombres aléatoires **réels** compris entre **0.0 et 1.0** selon une distribution uniforme standard : - génère un nombre aléatoire - génère une matrice carrée *n* x *n* de nombres aléatoires - génère une matrice *n* x *m* de nombres aléatoires Voir encore les fonctions : `randn` (distribution normale standard), `randg` (distribution Gamma), `rande` (distribution exponentielle), `randp` (distribution de Poisson)
`randi(imax {, n {,m} } )` `randi([imin, imax]`	Génération de nombres aléatoires **entiers** selon une distribution uniforme standard : - nombre(s) entier(s) compris entre 1 et *imax* - nombre(s) entier(s) compris entre *imin* et *imax*

`{,n{,m}})`	En l'absence des paramètres *n* et *m*, un seul nombre est généré. Avec le paramètre *n*, une matrice carrée de *n* x *n* nombres est générée. Avec les paramètres *n* et *m*, c'est une matrice *n* x *m* qui est générée.
`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.
`abs(var)`	Valeur absolue (positive) de *var* Ex: `abs([3.1 -2.4])` retourne [3.1 2.4]
`sign(var)`	(signe) Retourne "1" si *var*>0, "0" si *var*=0 et "-1" si *var*<0 Ex: `sign([3.1 -2.4 0])` retourne [1 -1 0]
`real(var)` et `imag(var)`	Partie réelle, resp. imaginaire, de la *var* complexe

Voir `help elfun` où sont notamment encore décrites : autres fonctions trigonométriques (sécante, cosécante, cotangente), fonctions hyperboliques, manipulation de nombres complexes... Et voir encore `help specfun` pour une liste des fonctions mathématiques spécialisées (Bessel, Beta, Jacobi, Gamma, Legendre...).

3.3.2 Fonctions de changement de type numérique

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 :

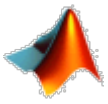
Fonction	**Description**
var2 = `single(var1)` var4 = `double(var3)`	Retourne, dans le cas où *var1* est une variable réelle double précision ou entière, une variable *var2* en **simple précision** Retourne, dans le cas où *var3* est une variable réelle simple précision ou entière, une variable *var4* en **double précision**
`int8` , `int16` , `int32` , `int64` `uint8` , `uint16` , `uint32` , `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

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...)
`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 *var*iable, "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"
`isinf(var)`	Vrai si la variable *var* est infinie positive ou négative (`Inf` ou - `Inf`)
`isnan(var)`	Vrai si la variable *var* est indéterminée (`NaN`)
`isfinite(var)`	Vrai si la variable *var* n'est ni infinie ni indéterminée Ex: `isfinite([0/0 NaN 4/0 pi -Inf])` retourne [0 0 0 1 0]

Les fonctions logiques spécifiques aux tableaux (vecteurs, matrices N-D) sont présentées au chapitre "**Fonctions matricielles**".



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. Il permet de construire des **séries linéaires** sous la forme de **vecteurs ligne**, notamment utilisés pour l'adressage des éléments d'un tableau.

série= `début:fin`
série= `colon(début,fin)`

Crée une *série* numérique **linéaire** débutant par la valeur *début*, auto-incrémentée 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:

- `v=1:9` crée le vecteur v=[1 2 3 4 5 6 7 8 9]
- `v(1:2:end)` retourne le vecteur [1 3 5 7 9]
- `x=1.7:4.6` crée le vecteur x=[1.7 2.7 3.7]

série= `début:pas:fin`
série= `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:

- `x=0:0.2:pi` génère le vecteur x=[0.0 0.2 0.4 0.6 0.8 1.0 1.2 1.4 1.6 1.8 2.0 2.2 2.4 2.6 2.8 3.0]
- `-4:-2:-11.7` retourne le vecteur [-4 -6 -8 -10]

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, la syntaxe `début:pas:fin` (plutôt qu'avec `linspace`) est particulièrement **intéressante 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)

MATLAB/Octave ne fait pas vraiment de différence entre un scalaire, un vecteur, une matrice ou un tableau à N-dimensions, ces objets pouvant être redimensionnés dynamiquement. Un **vecteur** n'est donc qu'une matrice NxM dégénérée d'une seule ligne (1xM) ou une seule colonne (Nx1).

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

On présente ci-dessous les principales techniques d'**affectation** de vecteurs par l'usage des crochets `[ ]`, et **adressage** de ses éléments par l'usage des parenthèses `( )`.

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'`	**Transpose** le vecteur *vec*. Si c'était un vecteur ligne, il devient un vecteur colonne, ou vice-versa
`vec(indices)`	Forme générale de la syntaxe d'**accès aux éléments** d'un vecteur, où *indices* est un vecteur (ligne ou colonne) de valeurs entières positives désignant les indices des éléments concernés de *vec*. Typiquement *indices* peut prendre les formes suivantes : • `ind1:indN` : séquence contiguë (série) d'indices allant de *ind1* jusqu'à *indN* • `ind1:pas:indN` : séquence d'indices de *ind1* à *indN* espacés par un *pas* • `[ind1 ind2 ...]` : indices *ind1*, *ind2* ... spécifiés (séquence discontinue) (notez bien les crochets `[ ]`) S'agissant de l'indice *indN*, on peut utiliser la valeur `end` pour désigner le **dernier élément** du vecteur Ex: Les exemples ci-dessous sont simplement dérivés de cette syntaxe générale : • `v3(2)` retourne la valeur "2", et `v4(2)` retourne "5" • `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] • `v6([2 4])` retourne un vecteur colonne contenant la 2e et la 4e valeur de v6, c'est-à-dire [4 6]' • 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(indices)=[]`	**Destruction** des éléments indicés du vecteur *vec* (qui est redimensionné en conséquence) Ex: soit `v10=(11:20)` c'est-à-dire [11 12 13 14 15 16 17 18 19 20] • l'instruction `v10(4:end)=[]` 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

Une **matrice** MATLAB/Octave 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 (avec les crochets `[ ]`) et d'**adressage** de ses éléments (parenthèses `( )`) est donc une généralisation à 2 dimensions de ce qui a été vu pour les vecteurs à 1 dimension (chapitre précédent), la seule différence étant que, pour se référer à une partie de matrice, il faut spécifier dans l'ordre le(s) numéro(s) de ligne puis de colonne(s) séparés par une virgule "`,`".

Comme pour les vecteurs, les indices de ligne et de colonne sont des valeurs entières débutant par 1 (et non pas 0 comme dans la plupart des 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
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(indices1,indices2)`	Forme générale de la syntaxe d'**accès aux éléments** d'une matrice (tableau à 2 dimensions), où *indices1* et *indices2* sont des vecteurs (ligne ou colonne) de valeurs entières positives désignant les indices des éléments concernés de *mat*. *indices1* se rapporte à la première dimension de *mat* c'est-à-dire les numéros de lignes, et *indices2* à la seconde dimension c'est-à-dire les numéros de colonnes. Dans la forme simplifiée où l'on utilise `:` la place de *indices1*, cela désigne toutes les lignes ; respectivement si l'on utilise `:` la place de *indices2*, cela désigne toutes les colonnes. Ex: Les exemples ci-dessous sont simplement dérivés de cette syntaxe générale : • 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]
`mat(indices)`	Lorsque l'on adresse une matrice à la façon d'un vecteur en ne précisant qu'un vecteur d'*indices*, l'adressage s'effectue en considérant que les éléments de la matrice sont numérotés de façon continue **colonne après colonne**. 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(:)=val` réinitialise tous les éléments de *mat* à la valeur *val*

`mat(indices1,indices2)=[]`	**Destruction** de **lignes** ou de **colonnes** d'une matrice (et redimensionnement de la matrice en conséquence). Ce type d'opération supprime des lignes entières ou des colonnes **entières**, donc on doit obligatoirement avoir `:` pour *indices1* ou *indices2*. **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]

On rapelle ici les fonctions `load {-ascii} fichier_texte` et `save -ascii fichier_texte variable` (décrites au chapitre "**Workspace**") qui permettent d'initialiser une matrice à partir de valeurs numériques provenant d'un *fichier_texte*, et vice-versa.

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 `help arith` et `help slash`).

Opérateur ou fonction	Description
`+` ou fonction `plus(m1,m2,m3,...)` `-` ou fonction `minus(m1,m2)`	**Addition** et **soustraction**. Les 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,m3,...)`	**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,m3,...)`	**Produit éléments par éléments**. Les 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(m1,m2)`	**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(m1,m2)`	**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(m1,m2)`	**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(m1,m2)`	**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)` retournent 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('abc')` retourne 'cba' `fliplr([1 2 3 ; 4 5 6])` => [3 2 1 ; 6 5 4], `flipud([1 2 3 ; 4 5 6])` => [4 5 6 ; 1 2 3]

`flip(tableau, dim)` `flipdim(tableau, dim)`	**Retourne** le *tableau* (qui peut avoir n'importe quelle dimension) selon sa dimension *dim* Ex: `flip([1 2 ; 3 4], 1)` permute les lignes => retourne [3 4 ; 1 2] `flip([1 2 ; 3 4], 2)` permute les colonnes => retourne [2 1 ; 4 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] `rot90([1 2 3 ; 4 5 6],-2)` => retourne [6 5 4 ; 3 2 1]
`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 `help elmat` et `help 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})` `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})` `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})` `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 1**: 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] **Ex 2**: soit le vecteur `v=1:10` • `v(find(and(v>=4, v<=6))) = v(find(and(v>=4, v<=6))) + 30` ajoute 30 à tous les éléments dont la valeur est comprise entre 4 et 6, donc modifie ici `v` et retourne v=[1 2 3 34 35 36 7 8 9 10]
`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 colonne, à moins que *var1* **et** *var2* soient tous deux de type ligne. *var1* et *var2* peuvent être des tableaux cellulaires (contenant par exemple des chaînes) 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'} • si `m1=[3 4 ; -1 6 ; 6 3]` et `m2=[6 -1 9]`, alors `intersect(m1,m2)` retourne [-1 6]

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

4.6 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 (i.e. exclusivement composée de valeurs `true` ou `false`). 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 l'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*)=true. 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 éliminer 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*)=true 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'

4.7 Chaînes de caractères

4.7.1 Généralités

Lorsque l'on définit une **chaîne de caractères** (*string*), celle-ci sera délimitée entre deux **apostrophes**. Si la chaîne contient des apostrophes, le mécanisme d'échappement consiste à les dédoubler.

De façon interne, les chaînes sont stockées par MATLAB/Octave sur des **vecteurs-ligne** de type **char** dans lesquels chaque caractère de la chaîne occupe un **élément** du vecteur. Il est aussi possible de manipuler des **matrices de chaînes**, comme nous l'illustrons ci-dessous, ainsi que des "**tableaux cellulaires**" de chaînes.

Dans le cas où vous manipuleriez des caractères **accentués** (ou tout caractère non-ASCII-7bits), prenez note de la différence suivante qui peut conduire à des problèmes de portage de code :

- MATLAB stocke chacun des caractères sur 2 octets
- Octave sous Linux stocke les caractères non accentués (ASCII 7-bits) sur 1 octet, et les caractères accentués sur 2 octets
- alors que Octave sous Windows (si l'on active le codepage `dos('chcp 437')`) stocke chacun des caractères (non accentués ou accentués) sur 1 octet

`string = 'chaîne de caractères'`

Stocke la *chaîne de caractères* (spécifiée entre apostrophes) sur la variable *string* qui est ici un **vecteur-ligne** contenant autant d'éléments que de caractères. Les apostrophes faisant partie de la chaîne doivent donc être dédoublés (sinon interprétés comme signe de fin de chaîne... la suite de la chaîne provoquant alors une erreur)

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

`string = "chaîne de caractères"`

Propre à Octave, la délimitation de chaîne par 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 : `str = "Texte\ttabulé\net sur\t2 lignes"`

pour obtenir l'équivalent sous MATLAB (ou Octave) : `str = sprintf('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"

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

Concatène **verticalement** les chaînes *s1*, *s2*, *s3*... Produit donc une **matrice de chaînes de caractères** *mat_string* contenant la chaîne *s1* en 1ère ligne, *s2* en seconde ligne, *s3* en 3ème ligne... Les chaînes éventuellement vides sont ignérées, c'est-à-dire ne produisent dans ce cas pas de lignes blanches (contrairement à `char` ou `str2mat`).

Remarque importante: pour produire cette matrice *mat_string*, MATLAB/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=strvcat(s1,s2,s3)` retourne la matrice de chaînes de dimension 3x16 caractères :

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

puis `mat=strvcat(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 = char(s1,s2,s3...)`
M `mat_string = str2mat(s1,s2,s3...)`
O `mat_string = [s1 ; s2 ; s3 ...]`

Concatène **verticalement** les chaînes *s1*, *s2*, *s3*... de la même manière que `strvcat`, à la nuance près que les éventuelles chaînes vides produisent dans ce cas une ligne vide. La 3ème forme ne fonctionne que sous Octave (MATLAB générant une erreur si les chaînes *s1*, *s2*, *s3*... n'ont pas toutes la même longueur)

`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"

Usage de **caractères accentués** dans des **scripts ou fonctions** (M-files) :

- 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 bien entendu de l'encodage du M-file ayant généré ce fichier. 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 **Linux** et **Mac OS X**, l'encodage par défaut est UTF-8 et il n'y a aucun problème particulier, que ce soit avec MATLAB ou Octave !
- Sous **Windows**, si le M-file est encodé UTF-8 et qu'il affiche des chaînes dans la console MATLAB/Octave (avec `disp`, `fprintf`...) :
  - ☒ avec MATLAB (testé sous R2014), les caractères accentués ne s'affichent pas proprement
  - ☒ avec Octave 3.2 à 3.8, 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 Octave 4.0 l'usage de caractères accentués n'est officiellement pas supporté

4.7.2 Fonctions générales relatives aux chaînes

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

Notez encore que, pour la plupart des fonctions ci-dessous, l'argument *string* peut aussi être une **cellule** voir un **tableau cellulaire** de chaînes !

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>
`string(offset:offset+(length-1))` O `substr(string, offset {, length})`	Retourne de la chaîne *string* la **sous-chaîne** débutant à la position *offset* et de longueur *length* Avec la fonction O `substr` : - si *length* n'est pas spécifié, la sous-chaîne s'étend jusqu'à la fin de *string* - si l'on spécifie un *offset* négatif, le décompte s'effectue depuis la fin de *string* Ex: si l'on a `str='abcdefghi'`, alors • `substr(str,3,4)` retourne 'cdef', identique à `str(3:3+(4-1))` • `substr(str,3)` retourne 'cdefghi', identique à `str(3:end)` • `substr(str,-3)` retourne 'ghi', identique à `str(end-3+1:end)`
`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','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,str_sep)`	**Découpe** la chaîne *string* selon la **chaîne**-séparateur *str_sep*, et retourne les sous-chaînes résultantes sur une matrice de chaînes Ex: `strsplit('abc//def//ghi/jkl','//')` retourne la matrice `abc` `def` `ghi/jkl`
`ostrsplit(string,cars_sep)`	Propre à Octave, cette fonction **découpe** la chaîne *string* en utilisant les différents **caractères** de *cars_sep*, et retourne les sous-chaînes résultantes sur un vecteur cellulaire de chaînes. Ex: `ostrsplit('abc/def/ghi*jkl','/*')` retourne le vecteur cellulaire `{'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)`	**Compare** les 2 chaînes *string1* et *string2*: retourne 1 si elles sont identiques, 0 sinon. La fonction `strcmpi` ignore les différences entre majuscule

`strncmp(string1,string2,n)` `strncmpi(string1,string2,n)`	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

4.7.3 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) 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]"
`sscanf(string,format)`	Permet de récupérer le(s) **nombre(s)** se trouvant dans la chaîne *string* (voir chapitre "**Entrées-sorties**")
`str2num(string)`	Convertit en **nombres** le(s) nombre(s) se trouvant dans la **chaîne** *string*. Pour des possibilités plus élaborées, on utilisera la fonction `sscanf` décrite au chapitre "**Entrées-sorties**". Ex: `str2num('12 34 ; 56 78')` retourne la matrice [12 34 ; 56 78]

`eval(expression)`	Évalue (**exécute**) l'*expression* MATLAB/Octave spécifiée Ex : si l'on a une chaîne `str_mat='[1 3 2 ; 5.5 4.3 2.1]'`, l'expression `eval(['x=' str_mat])` permet d'affecter les valeurs de cette chaîne à la matrice x

4.8 Tableaux multidimensionnels

4.8.1 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...).

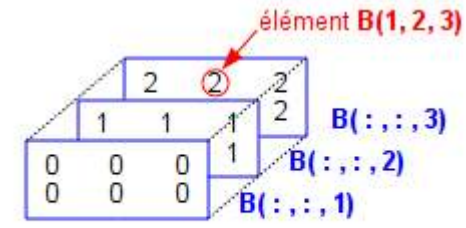


Figure : exemple d'un tableau tridimensionnel B de dimension 2 x 3 x 3

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.

4.8.2 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)]` retournent 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)]` retournent 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.

4.9 Structures (enregistrements)

4.9.1 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'`
- Définition d'une 3e personne via une notation plus compacte en spécifiant **tous les champs d'un coup** :
  `personne(3)=struct('nom','Muller','prenom','Robert','age',28,'code_postal',2000`
  • ATTENTION: on ne peut utiliser cette fonction que si l'on spécifie tous les champs ! Donc `personne(3)=struct('nom','Muller','age',28)` retournerait une erreur (car il manque les champs : prenom, code_postal, localite)
  • ✖ Ⓜ Sous MATLAB seulement, les champs doivent toujours être passés dans le **même ordre** ! Donc si l'on crée la structure `pays(1)=struct('nom','Suisse','capitale','Berne')`, l'instruction `pays(1)=struct('capitale','Paris','nom','France',)` génère ensuite une erreur (les 2 champs nom et capitale étant permutés par rapport à la séquence initiale de création des champs de cette structure) !

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 :
  • pour des champs de type **nombre** (ici liste des âges de tous les individus) :
  - `vec_ages = [ personne.age ]` retourne un vecteur-ligne *vec_ages*
  • pour des champs de type **chaîne** (ici 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)

- Et l'on peut 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 tableaux classiques ou des tableaux cellulaires !

- par exemple 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_mort(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)` ...

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

4.9.2 Fonctions spécifiques relatives aux structures

Fonction	Description
`struct` `setfield` `rmfield`	*Ces fonctions ont été illustrées dans l'exemple ci-dessus...*
`numfields(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)
`for [ valeur , champ ] = tab_struct` `  % on peut utiliser champ` `  %  et valeur` `end`	Propre à Octave, cette forme particulière de la structure de contrôle `for ... end` permet de boucler sur tous les éléments d'un tableau de structures *tab_struct* et accéder aux noms de *champ* et aux *valeurs* respectives

4.10 Tableaux cellulaires (cells arrays)

4.10.1 Généralités

Le "**tableau cellulaire**" ("cells array") est le type de donnée MATLAB/Octave le plus polyvalent. Il se distingue du 'tableau standard' en ce sens qu'il peut se composer d'**objets de types différents** (scalaire, vecteur, chaîne, matrice, structure... et même tableau cellulaire => permettant ainsi même de faire des tableaux cellulaires imbriqués dans des tableaux cellulaires !).

Initialement uniquement bidimensionnels sous Octave, les tableaux cellulaires peuvent désormais être **multidimensionnels** (i.e. à 3 indices ou plus) depuis Octave 3.

Pour définir un tableau cellulaire et accéder à ses éléments, on recourt aux **accolades** `{ }` (notation qui ne désigne ici pas, contrairement au reste de ce support de cours, des éléments optionnels). Ces accolades seront utilisées soit au niveau des **indices** des éléments du tableau, soit dans la définition de la **valeur** qui est introduite dans une cellule. Illustrons ces différentes syntaxes par un exemple.

Exemple:

A) Nous allons **construire le tableau cellulaire** 2D de 2x2 cellules `T` ci-contre par étapes successives. Il contiendra donc les cellules suivantes :
- une chaîne 'hello'
- une matrice 2x2 [22 23 ; 24 25]
- un tableau contenant 2 structures (nom et age de 2 personnes)
- et un tableau cellulaire 1x2 imbriqué { 'quatre' 44 }

Tableau cellulaire **T**

'hello'	22 23 24 25
personne nom: 'Dupond' age: 25 nom: 'Durand' age: 30	{ 'quatre' 44 }

- commençons par définir, indépendemment du tableau celulaire T, le tableau de structures "personne" avec `personne.nom='Dupond'; personne.age=25; personne(2).nom='Durand'; personne(2).age=30;`
- avec `T(1,1)={ 'hello' }` ou `T{1,1}='hello'` on définit la première cellule (examinez bien l'usage des **parenthèses** et des **accolades** !) ;
  comme T ne préexiste pas, on pourrait aussi définir cette première cellule tout simplement avec `T={'hello'}`
- avec `T(1,2)={ [22 23 ; 24 25] }` ou `T{1,2}=[22 23 ; 24 25]` on définit la seconde cellule
- puis `T(2,1)={ personne }` on définit la troisième cellule
- avec `T(2,2)={ { 'quatre' , 44 } }` ou `T{2,2}={ 'quatre' , 44 }` on définit la quatrième cellule
- on aurait aussi pu définir tout le tableau en une seule opération ainsi :
  `T={ 'hello' , [22 23 ; 24 25] ; personne , { 'quatre' , 44 } }`
  Remarque : on aurait pu omettre les virgules dans l'expression ci-dessus

B) Pour **accéder aux éléments** d'un tableau cellulaire, il faut bien comprendre la différence de syntaxe suivante :
- la notation `tableau (i,j)` (usage de **parenthèses**) retourne le "**container**" de la **cellule** d'indice *i,j* du *tableau* (tableau cellulaire à 1 élément)
- par extension, `tableau (i,:)` retournerait par exemple un nouveau tableau cellulaire contenant la *i*-ème ligne de *tableau*
- tandis que `tableau {i,j}` (usage d'**accolades**) retourne le **contenu** (c-à-d. la valeur) de la **cellule** d'indice *i,j*

- ainsi `T(1,2)` retourne le container de la seconde cellule de T (tableau cellulaire à 1 élément)
- et `T(1,:)` retourne un tableau cellulaire contenant la première ligne du tableau T
- alors que `T{1,2}` retourne le contenu de la seconde cellule, soit la matrice [22 23 ; 24 25] proprement dite
  et `T{1,2}(2,2)` retourne la valeur 25 (4e élément de cette matrice)
- avec `T{2,1}(2)` on récupère la seconde structure relative à Durand
  et `T{2,1}(2).nom` retourne la chaîne 'Durand', et `T{2,1}(2).age` retourne la valeur 30

et l'on pourrait p.ex. changer le nom de la second personne avec `T{2,1}(2).nom='Muller'`
- avec `T{2,2}` on récupère le tableau cellulaire de la 4e cellule
  et `T{2,2}{1,1}` retourne la chaîne 'quatre', et `T{2,2}{1,2}` retourne la valeur 44
  et l'on pourrait p.ex. changer la valeur avec `T{2,2}{1,2}=4`

C) Pour **supprimer** une ligne ou une colonne d'un tableau cellulaire, on utilise la syntaxe habituelle :

- ainsi `T(2,:)=[]` supprime la seconde ligne de T

D) Pour **récupérer** sur un **vecteur numérique** tous les nombres d'une colonne ou d'une ligne d'un tableau cellulaire :
- soit le tableau cellulaire suivant: `TC={'aa' 'bb' 123 ; 'cc' 'dd' 120 ; 'ee' 'ff' 130}`

- tandis que `vec_cel=TC(:,3)` nous retournerait un "**vecteur cellulaire**" contenant la 3e colonne de ce tableau,
- on peut directement récupérer (sans faire de boucle `for`), sur un **vecteur de nombres**, tous les éléments de la 3e colonne avec `vec_nb = [ TC{:,3} ]`
- ou par exemple calculer la moyenne de tous les nombres de cette 3e colonne avec `mean( [ TC{:,3} ] )`

E) Et l'on peut même utiliser l'**indexation logique** pour extraire des parties de tableau cellulaire !

Voici un exemple parlant :
- soit le tableau cellulaire de personnes et âges : `personnes={'Dupond' 25; 'Durand' 30; 'Muller' 60}`
- l'instruction `personnes( ([ personnes{:,2} ] > 27)' ,1)` retourne alors, sous forme de tableau cellulaire,
  les noms des personnes âgées de plus de 27 ans (Durand et Muller) ;
- pour ce faire, on a ici "indexé logiquement" la première colonne de `personnes` (contenant les noms) par le vecteur logique `[ personnes{:,2} ] > 27` (que l'on transpose pour qu'il soit en colonne), et on n'extrait de ce tableau `personnes` que la `1` ère colonne (les noms)

Il est intéressant de noter que les tableaux cellulaires peuvent être utilisés comme **paramètres d'entrée** et **de sortie** à toutes les **fonctions** MATLAB/Octave (un tableau cellulaire pouvant, par exemple, remplacer une liste de paramètres d'entrée).

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

4.10.2 Fonctions spécifiques relatives aux tableaux cellulaires

Nous présentons dans le tableau ci-dessous les **fonctions** les plus importantes **spécifiques aux tableaux cellulaires**.

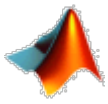
On utilisera en outre avec profit, dans des tableaux cellulaires contenant des chaînes de caractères, les fonctions de tri et de recherche `sort` / `sortrows` , `unique` , `intersect` / `setdiff` / `union` et `ismember` présentées plus haut.

Fonction	Description
`cell(`*n*`)` `cell(`*n*`,`*m*`)` `cell(`*n*`,`*m*`,`*o*`,`*p*`...)`	**Crée** un objet de type **tableau cellulaire** carré de dimension *n* x *n*, respectivement de *n* lignes x *m* colonnes, dont tous les éléments sont vides. Avec plus que 2 paramètres, crée un tableau cellulaire multidomensionnel. Mais, comme l'a démontré l'exemple ci-dessus, un tableau cellulaire peut être créé, sans cette fonction, par une simple affectation de type `tableau={ valeur }` ou `tableau{1,1}=valeur` , puis sa dimension peut être étendue dynamiquement

`iscell(var)` `iscellstr(var)`	**Test** si *var* est un objet de type tableau cellulaire : retourne 1 si c'est le cas, 0 sinon. Test si *var* est un tableau cellulaire de **chaînes**.
[*n m*]= `size(tab_cel)`	Retourne la **taille** (nombre *n* de **lignes** et *m* de **colonnes**) du tableau cellulaire *tab_cel*
mat = `cell2mat(tab_cel)`	Convertit le tableau cellulaire *tab_cel* en une matrice *mat* en concaténant ses éléments Ex: `cell2mat( { 11 22 ; 33 44 } )` retourne [11 22 ; 33 44]
tab_cel_string = `cellstr(mat_string)`	Conversion de la "matrice de chaînes" *mat_string* en un **tableau cellulaire de chaînes** *tab_cel_string*. Chaque ligne de *mat_string* est automatiquement "nettoyée" des <espaces> de remplissage (trailing blanks) avant d'être placée dans une cellule. Le tableau cellulaire résultant aura 1 colonne et autant de lignes que *mat_string*.
mat_string = `char(tab_cel_string)`	Conversion du tableau cellulaire de chaînes *tab_cel_string* en une **matrice de chaînes** *mat_string*. Chaque chaîne de *tab_cel_string* est automatiquement complétée par des <espaces> de remplissage (trailing blanks) de façon que toutes les lignes de *mat_string* aient le même nombre de caractères.
`celldisp(tab_cel)`	Affiche récursivement le **contenu** du tableau cellulaire *tab_cel*. Utile sous MATLAB où, contrairement à Octave, le fait de frapper simplement `tab_cel` n'affiche pas le contenu de *tab_cel* mais le type des objets qu'il contient.
M `cellplot(tab_cel)`	Affiche une figure représentant **graphiquement** le contenu du tableau cellulaire *tab_cel*
`num2cell`	Conversion d'un tableau numérique en tableau cellulaire
`struct2cell`, `cell2struct`	Conversion d'un tableau de structures en tableau cellulaire, et vice-versa
`cellfun(function,tab_cel{,dim})`	Applique la fonction *function* (qui peut être: `'isreal'`, `'isempty'`, `'islogical'`, `'length'`, `'ndims'` ou `'prodofsize'`) à tous les éléments du tableau cellulaire *tab_cell*, et retourne un tableau numérique

4.10.3 Listes Octave

Le type d'objet "liste" était **propre à Octave**. Conceptuellement proches des "tableaux cellulaires", les listes n'ont plus vraiment de sens aujourd'hui et disparaissent de Octave depuis la version 3.4. On trouve sous **ce lien** des explications relatives à cet ancien type d'objet.



5. Diverses autres notions MATLAB/Octave

5.1 Dates et temps

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
date_string = `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
date_num = `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_vec = `clock`	Retourne la **date et heure courante** sous forme d'un vecteur-ligne *date_vec* 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
date_string = `datestr(`*date_num*`{,'format'})`	Convertit en **chaîne de caractères** la date et heure spécifiée par : - la date numérique *date_num* - le vecteur *date_vec*

`datestr(date_num {,code})` `datestr(date_vec {,'format'})` `datestr(date_vec {,code})`	Le formatage peut être défini 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` : **heures** ; `MM` : **minutes** - `SS` : **secondes** ; `FFF` : **milli-secondes** En l'absence de *format* ou de *code*, c'est un format `'dd-mmm-yyyy HH:MM:SS'` qui est utilisé par défaut Si le paramètre *date_num* est compris entre 0 et 1, cette fonction retourne des **heures/minutes/secondes**. Ex: • `datestr(now)` ou `datestr(now,'dd-mmm-yyyy HH:MM:SS')` retournent '08-Apr-2005 20:45:00' • `datestr(now,'ddd')` retourne 'Fri' (abréviation de Friday) (voir aussi la fonction `weekday` ci-dessous)
date_num = `datenum(date_string)` `datenum(date_vec)` `datenum(annee,mois,jour {,heure,min,sec})`	Retourne le **nombre** exprimant la date et heure spécifiée par : - la chaîne *date_string* (voir `help datenum` pour les formats possibles) - le vecteur *date_vec* - les nombres *annee*, *mois*, *jour*, { *heure*, *min* et *sec* } Ex : `datenum('08-Apr-2005 20:45:00')` et `datenum(2005,4,8,20,45,0)` retournent le nombre 732410.8645833334
date_vec ou [*annee mois jour heure minute seconde*] = `datevec(date_string)` `datevec(date_nun)`	Retourne un vecteur ligne de 6 **valeurs numériques** définissant l'*annee*, *mois*, *jour*, *heure*, *minute* et *seconde* à partir de : - la chaîne *date_string* (voir `help datenum` pour les formats possibles) - la date numérique *date_num* Pour avoir des valeurs entières, faire `fix(datevec(date))` . Ex: `datevec('08-Apr-2005 20:45:03')` et `datevec(732410.8646180555)` retournent le vecteur [2005 4 8 20 45 3]

5.1.4 Fonctions utilitaires

Fonction	Description
`calendar` `calendar(annee, mois)` `calendar(date)` *mat* = `calendar(...)`	Affiche 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* spécifié (sous forme de nombres) Affectée à une variable, cette fonction retourne une matrice *mat* 6x7 contenant les numéros de jour du mois correspondant. Ex: `calendar(2005,4)` ou `calendar('8-Apr-2005')` affichent : 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 cette fonction `calendar` en l'affectant à une variable)
[*numero_jour* *nom_jour*] = `weekday(date_string)` `weekday(date_num)`	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 *date_string*, ou de nombre *date_num*). Si cette fonction est affectée à une seule variable, 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)`	Retournent 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

Pour mesurer plus précisément le temps CPU consommé par les différentes instructions et fonctions de votre code MATLAB/Octave, voyez en outre les fonctions de "**profiling**" décrites au chapitre "**Interaction, debugging, profiling...**"), sous-chapitre "Profiling".

Fonction	Description
`pause(secondes)` `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 ;`
`cputime`	Retourne le nombre de secondes de processeur consommées par MATLAB/Octave depuis le début de la session ("CPU time"). Sous Octave cette fonction a encore d'autres paramètres de sortie (voir `help cputime`) Ex: `t0 = cputime; A=rand(1000,1000); B=inv(A); dt = cputime-t0` : 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 GHz, que ce soit sous MATLAB ou Octave)
`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*])

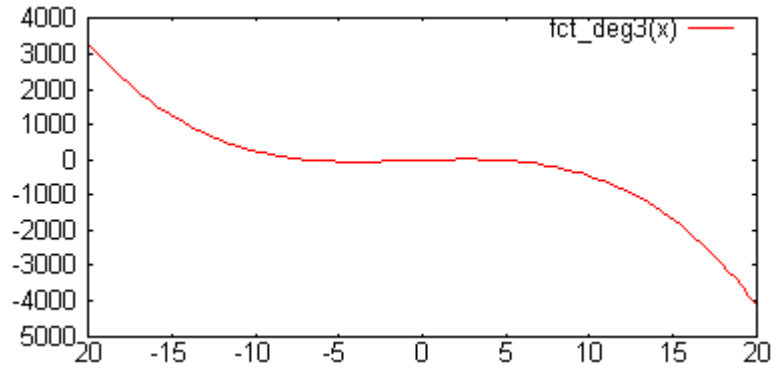
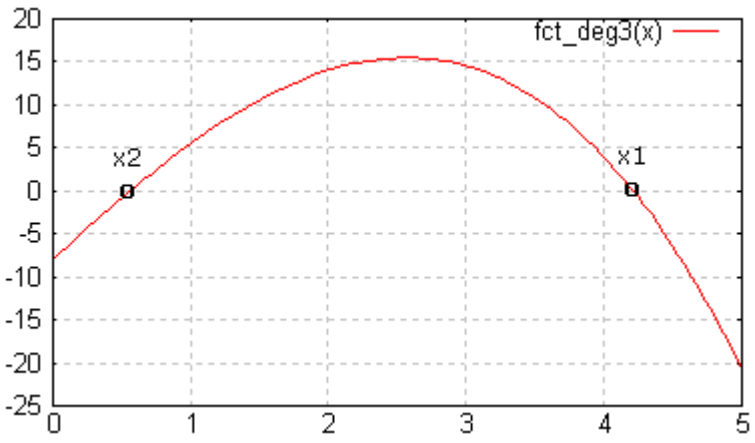
	Ex : l'exemple ci-dessus pourrait être aussi implémenté ainsi : `t1 = clock; A=rand(1000,1000); B=inv(A); dt = etime(clock,t1)`

5.2 Résolution d'équation non linéaire

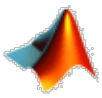
Les fonctions `fzero('fonction',x0)` ou `fsolve('fonction',x0)` permettent de trouver, par approximations successives en partant d'une valeur donnée x= *x0*, la(les) racine(s) d'une *fonction* non linéaire **y=f(x)**, c'est-à-dire les valeurs **x1**, **x2**, **x3**... pour lesquelles f(x)=0 .

Remarque : sous **MATLAB**, la fonction `fzero` est standard, mais la fonction `fsolve` est implémentée dans la toolbox "Optimisation".

Illustrons l'usage de cette fonction par un **exemple** :

Étape	Réalisation
1) Soit la **fonction** de 3e degré : **y = - 0.5*x^3 - x^2 + 15*x - 8**	On doit donc trouver les solutions x1, x2... pour f(x)=0, donc : - 0.5*x^3 - x^2 + 15*x - 8 = 0
2) Commençons par définir cette équation sous forme d'une **fonction** MATLAB/Octave *(voir chapitre "**Fonctions**")*	Réaliser le M-file appelé `fct_deg3.m` contenant par conséquent le code suivant : `function [Y]=fct_deg3(X)` `  Y = - 0.5 * X.^3 - X.^2 + 15*X - 8` `return`
3) Puis **graphons** rapidement cette fonction (autour de -20 ≤ x ≤ 20) pour estimer graphiquement une valeur approximative x0 de départ *(voir l'usage de la fonction `fplot` au chapitre "**Graphiques 2D**")*	Le code `fplot('fct_deg3(x)',[-20 20])` produit le graphique ci-dessous : 
4) Zoomons autour de la solution (qui a l'air de se trouver vers 1 et 4) en rétrécissant l'intevalle à 0 ≤ x ≤ 5	Le code `fplot('fct_deg3(x)',[0 5]) ; grid('on')` produit le graphique ci-dessous (sauf les étiquettes x1 et x2 que nous avons ajoutées manuellement) : 
5) Choisissons la valeur de départ x0 = 5, et recherchons la **première solution**	Entrons `x1=fzero('fct_deg3',5)` ou `x1=fsolve('fct_deg3',5)` Après une série d'itérations, Octave retourne : x1 = 4.2158 On peut vérifier que c'est bien une solution en entrant `fct_deg3(x1)`

	qui retourne bien 0 (ou une valeur infiniment petite)
6) Choisissons la valeur de départ x0 = 0, et recherchons la **seconde solution**	Entrons `x2=fzero('fct_deg3',0)` ou `x2=fsolve('fct_deg3',0)` Après une série d'itérations, Octave retourne : x2 = 0.56011 On peut vérifier que c'est bien une solution en entrant `fct_deg3(x2)` qui retourne bien 0 (ou une valeur infiniment petite)



6. Graphiques, images, animations

6.1 Concepts de base

Les fonctionnalités graphiques sous Octave ont fortement évolué ces dernières années (implémentation des Handle Graphics...) pour rejoindre le niveau de MATLAB. Nous nous basons ici sur les versions suivantes :

- **MATLAB R2014**, avec son moteur de graphiques intégré
- **GNU Octave-Forge 4.0.0**, avec les backends basés **Qt**/OpenGL, **FLTK**/OpenGL et **Gnuplot**.

L'**aide en ligne** relative aux fonctions de réalisation de graphiques s'obtient, de façon classique, en frappant `help` *`fonction_graphique`* (Ex: `help plot`). En outre :

- sous **MATLAB**: les commandes `help graph2d` , `help graph3d` et `help specgraph` affichent la liste des fonctions graphiques disponibles
- sous **Octave**: on se réfèrera au Manuel Octave (HTML ou PDF) au chapitre "Plotting", ou via la commande `doc` *`fonction_graphique`*

Pour une **comparaison** des possibilités graphiques entre Octave/FLTK, Octave/Gnuplot et MATLAB, voyez cette intéressante page : **http://octave.sourceforge.net/compare_plots/**

6.1.1 La notion de "backends graphiques" sous Octave

MATLAB, de par sa nature commerciale monolithique, intègre son propre moteur d'affichage de graphiques.

GNU Octave est conçu, dans la philosophie Unix, de façon plus modulaire en s'appuyant sur des outils externes. En matière de visualisation, on parle de "**backends graphiques**" :

- C'est ainsi que le logiciel libre de visualisation **Gnuplot** a longtemps été utilisé par Octave comme "moteur graphique" standard. À l'origine essentiellement orienté tracé de courbes 2D et de surfaces 3D en mode "filaire", Gnuplot est devenu capable, depuis la version 4.2, de remplir des surfaces colorées, ce qui a permis (depuis Octave 3) l'implémentation de fonctions graphiques 2D/3D classiques MATLAB (fill, pie, bar, surf...). Les "handles graphics" ont commencé à être implémentés avec Gnuplot depuis Octave 2.9.
- Depuis la version 3.4 (en 2011), Octave embarque son propre moteur graphique basé **FLTK** (Fast Light Toolkit) et OpenGL. Celui-ci est plus rapide et offre davantage d'interactivité que Gnuplot.
- Depuis la version 4.0 (en 2015), qui voit l'arrivée d'une interface utilisateur graphique officielle (Octave GUI) basée sur le toolkit graphique **Qt**, Octave intègre un nouveau backend **QtHandles** s'appuyant logiquement aussi sur **Qt** et OpenGL.

Ces différentes solutions n'empêchent pas l'utilisateur de recourir à d'autres "backends graphiques" s'il le souhaite. Parmi les autres projets de couplage ("bindings") avec des grapheurs existants, ou de développement de backends graphiques propres à Octave, on peut citer notamment :

- **Octaviz** : 2D/3D, assez complet (wrapper donnant accès aux classes **VTK**, Visualization ToolKit) (voir article **FI-EPFL 5/07**)
- **OctPlot** : 2D (ultérieurement 3D ?)
- **epsTK** : fonctions spécifiques pour graphiques 2D très sophistiqués (était intégré à la distribution Octave-Forge 2.1.42 Windows)

Quant aux anciens projets suivants, ils sont (ou semblent) arrêtés : **JHandles** (package Octave-Forge, développement interrompu depuis 2010, voir cette ancienne **page**), **Yapso** (Yet Another Plotting System for Octave, 2D et 3D, basé OpenGL), **PLplot** (2D et 3D), **Oplot++** (2D et 3D, seulement sous Linux et MacOSX), **KMatplot** (2D et 3D, ancien, nécessitant Qt/KDE), **KNewPlot** (2D et 3D, ancien, nécessitant Qt et OpenGL), **Grace** (2D).

6.1.2 Choix du backend graphique sous Octave

Depuis Octave 4.0, que ce soit en mode graphique (GUI) ou commande (CLI), le backend sélectionné par défaut est **Qt**/OpenGL. C'est aussi celui-ci que nous vous recommandons désormais d'utiliser.

`available_graphics_toolkits`
Indique quels sont les backends disponibles dans votre distribution d'Octave

backend_courant= `graphics_toolkits`
Retourne le nom du backend qui est couramment actif

`graphics_toolkits('qt' | 'fltk' | 'gnuplot')`
Commute sur le backend spécifié
Attention : avant de passer cette commande commencez par fermer, avec `close('all')`, les fenêtres de graphiques qui auraient été ouvertes avec un autre backend

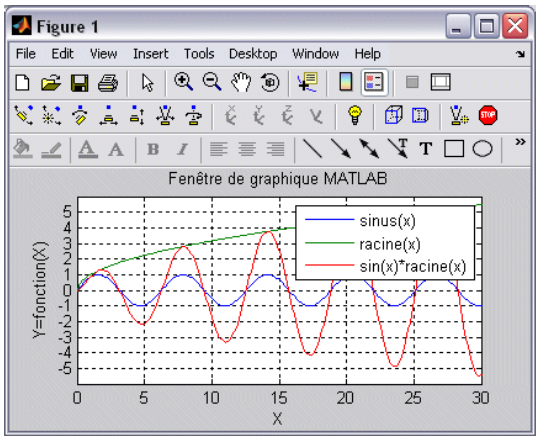
6.1.3 Fenêtres de graphiques

Les graphiques MATLAB/Octave sont affichés dans des **fenêtres de graphiques** spécifiques appelées "**figures**". Celles-ci apparaissent lorsqu'on fait usage des commandes `figure` ou `subplot`, ou automatiquement lors de toute commande produisant un tracé (graphique 2D ou 3D).

Dans les chapitres qui suivent on présente l'aspect et les fonctionnalités des fenêtres graphiques correspondant aux différentes backends graphiques. Le code qui a été utilisé pour produire les illustrations est le suivant :

```
x=0:0.1:10*pi;
y1=sin(x); y2=sqrt(x); y3=sin(x).*sqrt(x);
plot(x,y1,x,y2,x,y3);
grid('on');
axis([0 30 -6 6]);
set(gca,'Xtick',0:5:30);  set(gca,'Ytick',-5:1:5);
title('Fenêtre de graphique MATLAB / FLTK / Gnuplot');
xlabel('X'); ylabel('Y=fonction(X)');
legend('sinus(x)','racine(x)','sin(x)*racine(x)');
```

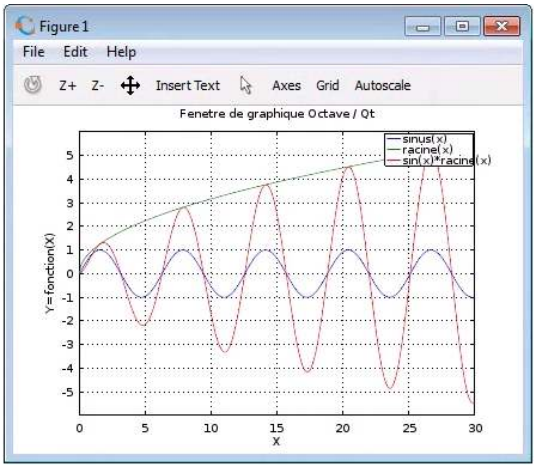
Fenêtre graphique MATLAB v7 à R2014



Les caractéristiques principales des fenêtres de graphiques **MATLAB** sont :

- Une **barre de menus** comportant notamment :
  - Edit > Copy Figure : copie de la figure dans le presse-papier (pour la "coller" ensuite dans un autre document) ; voyez Edit > Copy Options qui permet notamment d'indiquer si vous prenez l'image au format vecteur (défaut => bonne qualité, redimensionnable...) ou raster, background coloré ou transparent...
  - Tools > Edit Plot, ou commande `plotedit`, ou bouton Edit Plot (icône de pointeur) de la barre d'outils : permet de sélectionner les différents objets du graphique (courbes, axes, textes...) et, en double-cliquant dessus ou via les articles du menu Tools, d'éditer leurs propriétés (couleur, épaisseur/type de trait, symbole, graduation/sens des axes...)
  - File > Save as : exportation du graphique sous forme de fichier en différents formats raster (JPEG, TIFF, PNG, BMP...) ou vecteur (EPS...)
  - File > Export Setup, File > Print Preview, File > Print : mise en page, prévisualisation et impression d'un graphique (lorsque vous ne le "collez" pas dans un autre document)
  - Affichage de palettes d'outils supplémentaires avec View > Plot Edit Toolbar et View > Camera Toolbar
  - View > Property Editor, ou dans le menu Edit les articles Figure Properties, Axes Properties, Current Object Properties et Colormap, puis le bouton Inspector (ou la commande `propedit`) : pour modifier de façon très fine les propriétés d'un graphique (via ses handles...)
  - Ajout/dessin d'objets depuis le menu Insert
  - Un menu Camera apparaît lorsque l'on passe la commande `cameramenu`
- La **barre d'outils** principale, comportant notamment :
  - bouton Edit Plot décrit plus haut
  - boutons-loupes + et - (équivalents à Tools > Zoom In|Out) pour zoomer/dézoomer interactivement dans le graphique ; voir aussi les commandes `zoom on` (puis cliquer-glisser, puis `zoom off`), `zoom out` et `zoom(facteur)`
  - bouton Rotate 3D (équivalent à Tools > Rotate 3D) permettant de faire des **rotations 3D**, par un cliquer-glisser avec le bouton souris-gauche, y compris sur des graphiques 2D !
  - boutons Insert Colorbar (équivalent à la commande `colorbar`) et Insert Legend (équivalent à la commande `legend`)
  - boutons Show|Hide Plot Tools (ou voir menu View) affichant/masquant des sous-fenêtres de dialogues supplémentaires (Figure Palette, Plot Browser, Property Editor)

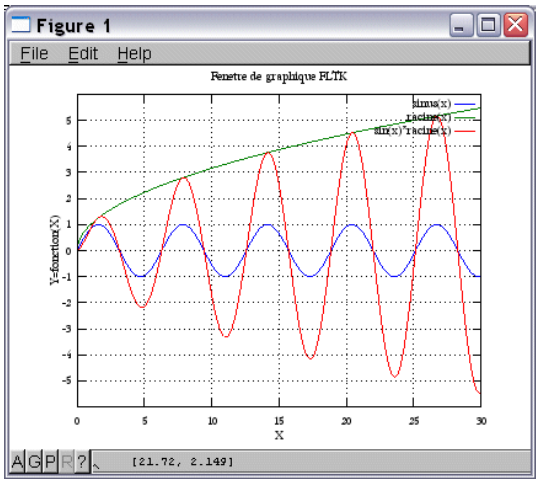
Fenêtre graphique Qt/OpenGL depuis Octave ≥ 4.0



Les caractéristiques principales des fenêtres de graphiques **Qt** sous Octave sont :

- Une **barre d'outils** comportant les boutons suivants (que l'on peut, pour certains, aussi activer via le menu Edit) :
  - Rotate : souris-gauche-glisser-horiz. effectue une **rotation 2D**, souris-gauche-glisser-vertic. effectue une **rotation 3D**, souris-milieu fait un autoscale en annulant la rotation 3D dans le cas d'une figure 2D
  - Z+ (Zoom In) : un clic souris-gauche fait un zoom avant **2x**, souris-gauche-glisser effectue un **rectangle-zoom**
  - Z- (Zoom Out) : un clic souris-gauche fait un zoom arrière **2x**
  - Z+ ou Z- : souris-roulette effectue un zoom avant/arrière (sans affecter la dimension Z dans les graphiques 3D)
  - Pan : souris-gauche-glisser pour **déplacement** horiz./vertical, souris-milieu fait un autoscale
  - Insert Text : ouvre une fenêtre de dialogue permettant de placer dans la figure une **annotation** en définissant police, taille, style et couleur ; comme pour la fonction `annotation` , la position est définie en **unités normalisées** par rapport à la fenêtre de figure, c'est-à-dire système de coordonnées dont l'angle inférieur gauche est (0,0) et l'angle supérieur droite (1,1) (et non dans le système d'axes du graphique comme le texte placé avec `text`) ; l'annotation sera donc fixe par rapport à la fenêtre de figure et ne change pas si l'on fait un pan ou zoom du graphique
  - Select : ☒ n'est pas encore opérationnel
  - Axes : affichage/masquage des **axes** et de la **grille** (bascule)
  - Grid : affichage/masquage de la **grille** (bascule, comme `grid('on|off')`)
  - Autoscale : **autoscaling** des axes (comme `axis('auto')`)
- Une **barre de menus** comportant :
  - File > Save et Save as : **sauvegarder** la figure sur un fichier graphique de type (selon l'extension que vous spécifiez):
    • vectorisé: PDF (défaut), SVG
    • raster: PNG, JPG
  - File > Close Figure ou ctrl-W ou case de fermeture X : referme la fenêtre de figure (comme `close`)
  - Edit > Copy ou ctrl-C : **copie** la figure dans le "presse-papier" en format raster haute définition (plus élevée que ne le ferait une copie d'écran), pour pouvoir la "coller" ensuite dans un autre document
  - Help : informations sur les versions de QtHandles et Qt

Fenêtre graphique FLTK/OpenGL depuis Octave ≥ 3.4



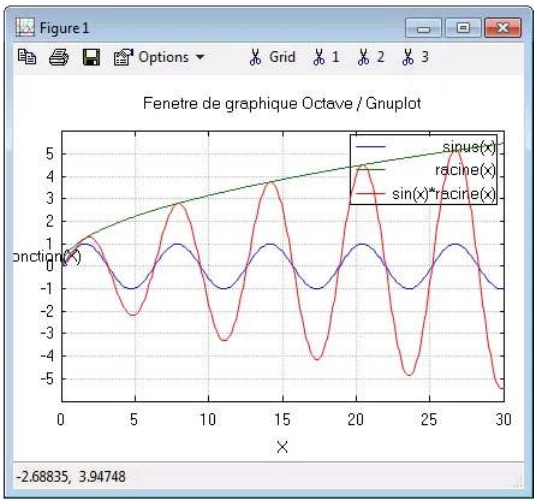
Les caractéristiques principales des fenêtres de graphiques **FLTK** sous Octave sont :

- Une **barre d'outils**, en bas à gauche, utilisée conjointement avec la **souris** :
  - bouton P ou touche p (pan) puis :
    • souris-gauche-glisser : **déplacement** X/Y dans graphiques 2D ou 3D
    • souris-droite-glisser : effectue un **rectangle-zoom**
  - bouton R ou touche r (rotate) puis souris-gauche-glisser : effectuer une **rotation** 3D dans graphiques 2D ou 3D
  - bouton A ou touche a ou souris-gauche-double clic : **autoscaling** des axes (comme `axis('auto')`)
  - souris-roulette : effectue un **zoom** avant/arrière
  - bouton G ou touche g : affichage/masquage de la **grille** (bascule, comme `grid('on|off')`)
  - bouton ? : affichage aide sur les raccourcis clavier et l'usage de la souris
- Une **barre de menus** comportant :
  - File > Save et Save as : **sauvegarder** la figure sur un fichier

graphique de type (selon l'extension que vous spécifiez):
• vectorisé: PDF, SVG, PS (PostScript)
• raster: PNG, JPG

- File > Close ou case de fermeture X : referme la fenêtre de figure (comme `close`)
- Edit : reprend les fonctionnalités P | R | A | G vues plus haut

Fenêtre graphique G Gnuplot ≥ 4.6 depuis Octave ≥ 4.0



Les caractéristiques principales des fenêtres de graphiques **Gnuplot ≥ 4.6** sous Octave sont :

- Une **barre d'outils** comportant :
  - bouton Copy graph to clipboard ou ctrl-C : **copie** la figure dans le "presse-papier" aux formats raster + vecteur GDI (pour pouvoir la "coller" ensuite dans un autre document)
  - bouton Print graph : **imprime** la figure (via dialogue d'impression standard)
  - bouton Save graph as EMF ou ctrl-S : **sauvegarde** la figure sur un fichier graphique raster de type EMF
  - bouton Grid : affichage/masquage de la **grille** (bascule, comme `grid('on|off')`)
  - un(des) bouton(s) 1 | 2 | 3 ... : affichage/masquage des différentes courbes composant le graphique (bascules)
  - un bouton/menu Options permettant de modifier certaines **préférences** de Gnuplot et les sauvegarder dans le fichier "wgnuplot.ini"
- Concernant l'utilisation de la **souris** (mode que l'on peut désactiver/réactiver en frappant m) :
  - fenêtre **2D** ou **3D** :
    • souris-roulette : déplacement du graphique selon l'axe Y
    • souris-maj-roulette : déplacement du graphique selon l'axe X
    • souris-ctrl-roulette : faire un zoom avant/arrière en X/Y (axe Z non affecté dans les graphiques 3D)
  - fenêtre **2D** seulement :
    • souris-droite-glisser souris-droite : effectue un rectangle-zoom
  - fenêtre **3D** seulement :
    • souris-gauche-glisser : **rotation 3D**
    • souris-milieu-mvmt horizontal : **zoom** avant/arrière (utiliser ctrl pour graphiques complexes)
    • souris-milieu-mvmt vertical : changement **échelle en Z** (utiliser ctrl pour graphiques complexes)
    • souris-maj-milieu-mvmt vertical : changement **origine Z** (utiliser ctrl pour graphiques complexes)
  - dans l'angle inférieur gauche s'affichent, en temps réel :
    - fenêtre **2D**: les **coordonnées X/Y** précises du **curseur**, que vous pouvez inscrire dans le graphique en cliquant avec souris-milieu
    - fenêtre **3D**: l'orientation de la vue (angle d'**élévation** par rapport au nadir, et **azimut**) et les facteurs d'**échelle** en X/Y et en Z
- Finalement, s'agissant des **raccourcis clavier** Gnuplot :
  - g : affichage/masquage de la **g**rille (bascule)
  - b : affichage/masquage de la **b**ox du graphique 2D ou 3D (bascule)
  - a : **a**utoscaling des axes (comme `axis('auto')`)
  - p et n : facteur de zoom **p**récédent, respectivement suivant (**n**ext)
  - u : dé-zoomer (**u**nzoom)
  - q : fermeture de la fenêtre graphique (**q**uit)

Pour mémoire, suivre **ce lien** pour accéder aux informations relatives aux anciennes versions de : • Gnuplot 3.x à 4.0 embarqué dans Octave-Forge 2.x Windows, • Gnuplot 4.2.2/4.3 embarqué dans Octave 3.0.1/3.0.3 MSVC

6.1.4 Axes, échelle, zoom/pan/rotation, quadrillage, légende, titre, annotations

Les fonctions décrites dans ce chapitre doivent être **utilisées après** qu'**une fonction de dessin** de graphique ait été passée (et non avant). Elles agissent immédiatement sur le graphique courant.

Fonction et description	
Exemple	**Illustration**

Lorsque l'on trace un graphique, MATLAB/Octave détermine automatiquement les **limites inférieures et supérieures** des axes X, Y {et Z} sur la base des valeurs qui sont graphées, de façon que le tracé occupe toute la fenêtre graphique (en hauteur et largeur). Les **rapports d'échelle** des axes sont donc différents les uns des autres. Les commandes `axis` et `xlim` / `ylim` / `zlim` permettent de modifier ces réglages.

a) `axis([Xmin Xmax Ymin Ymax { Zmin Zmax }])`
b) `axis('auto')`
c) `axis('manual')`
d) *lim_xyz* = `axis`

Modification des valeurs limites (sans que l'un des "aspect ratio" `equal` ou `square`, qui aurait été activé, soit annulé) :
a) recadre le graphique en utilisant les valeurs spécifiées des limites inférieures/supérieures des axes X, Y {et Z}, réalisant ainsi un **zoom** avant/arrière dans le graphique ; sous MATLAB il est possible de définir les valeurs `-inf` et `inf` pour faire déterminer les valeurs min et max (équivalent de `auto`)
b) se remet en mode "autoscaling", c-à-d. définit dynamiquement les limites inférieures/supérieurs des axes X, Y {et Z} pour faire apparaître l'intégralité des données ; sous MATLAB on peut spécifier ' `auto x` ' ou ' `auto y` ' pour n'agir que sur un axe
c) verrouille les limites d'axes courantes de façon que les graphiques subséquents (en mode `hold on`) ne les modifient pas lorsque les plages de valeurs changent
d) passée sans paramètre, la fonction `axis` retourne le vecteur-ligne *lim_xyz* contenant les limites [*Xmin Xmax Ymin Ymax* { *Zmin Zmax* }]

a) `xlim([Xmin Xmax])` , `ylim([Ymin Ymax])` , `zlim([Zmin Zmax])`
b) `xlim('auto')` , `ylim('auto')` , `zlim('auto')`
d) `xlim('manual')` , `ylim('manual')` , `zlim('manual')`
d) *lim_x* = `xlim` , *lim_y* = `ylim` , *lim_z* = `zlim`

Même fonctionnement que la fonction `axis`, sauf que l'on n'agit ici que **sur 1 axe** à la fois

a) `axis('equal')` ou `axis('image')` ou `axis('tight')`
b) `axis('square')`
c) `axis('normal')`
d) `axis('vis3d')`

Modification des rapports d'échelle ("aspect ratio") (sans que les limites inf. et sup. des axes X, Y {et Z} soient affectées) :
a) définit le même rapport d'échelle pour les axes X et Y ; est identique à `daspect([1 1 1])`
b) définit les rapports d'échelle en X et Y de façon la zone graphée soit carrée
c) annule l'effet des "aspect ratio" `equal` ou `square` en redéfinissant automatiquement le rapport d'échelle des axes pour s'adapter à la dimension de la fenêtre graphique ; est identique à `daspect('auto')`
d) sous MATLAB, bloque le rapport d'échelle pour rotation 3D

a) *ratio* = `daspect()`
b) `daspect(ratio)`
c) `daspect('auto')`

Rapport d'échelle entre les axes X-Y{-Z} (data aspect ratio) (voir aussi la commande `pbaspect` relatif au "plot box")
a) récupère, sur le vecteur *ratio* (3 éléments), le rapport d'échelle courant entre les 3 axes du graphique
b) modifie le rapport d'échelle entre les axes selon le vecteur *ratio* spécifié
c) le rapport d'échelle est mis en mode automatique, s'adaptant dès lors à la dimension de la fenêtre de graphique

a) `axis('off | on')`
b) `axis('nolabel | labelx | labely | labelz')`
c) `axis('ticx | ticy | ticz')`

Désactivation/réactivation affichage cadre/axes/graduation, quadrillage et labels :
a) désactive/rétablit l'affichage du cadre/axes/graduation et quadrillage du graphique ; sous MATLAB (mais pas Octave) agit en outre également sur les étiquettes des axes (labels)
b) désactive l'affichage des graduations des axes (ticks), respectivement rétablit cet affichage de façon différenciée en x, y et/ou z
c) active l'affichage des graduations des axes (ticks) et du quadrillage (grid) de façon différenciée en x, y et/ou z

a) `axis('xy')`
b) `axis('ij')`

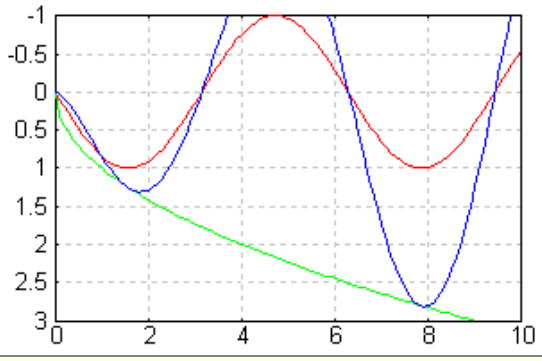
Inversion du sens de l'axe Y :

a) origine en bas à gauche, valeurs Y croissant de bas en haut (par défaut)
b) origine en haut à gauche, valeurs Y croissant de haut en bas.
Remarque : la dénomination `ij` fait référence au système de coordonnées communément utilisé en traitement d'image (origine se situant dans le coin supérieur gauche de l'image, axe horizontal nommé **i** et orienté normalement de gauche à droite, mais axe vertical **j** orienté de haut en bas !)

Ex 1

Ne vous attardez pas sur la syntaxe de la commande plot qui sera décrite plus loin au chapitre "Graphiques 2D"

```
x=0:0.1:10*pi;
y1=sin(x); y2=sqrt(x); y3=sin(x).*sqrt(x);
plot(x,y1,x,y2,x,y3);
legend('off');
 grid('on');
 axis([0 10 -1 3]);  % changement limites (zoom)
 axis('ij');         % inversion de l'axe Y
```



a) `zoom(facteur)`
b) `zoom off|on|xon|yon`
c) `zoom reset|out`

a) Zoome du *facteur* spécifié dans la figure courante. Si `>1` => grossit, si compris entre `0` et `1` => diminue.
Si *facteur* est un scalaire, agit sur tous les axes. Si c'est un vecteur, le 1er élément agit sur l'axe X, le second sur l'axe Y...
b) `off` désactive complètement les possibilités de zoom interactif dans la figure, `on` les réactive. `xon` n'autorise le zoom interactif qu'en X, et `yon` qu'en Y.
c) `reset` mémorise le zoom courant. `out` retourne au facteur de zoom préalablement mémorisé avec `reset`.

`rotate(handle, direction, angle, origine)`

Applique à l'objet graphique *handle* une **rotation** de *angle*, autour de l'axe spécifié par sa *direction* et de l'*origine* indiquée.
Si vous désirez plutôt changer l'orientation de la **vue**, utilisez la fonction `view` présentée plus bas.

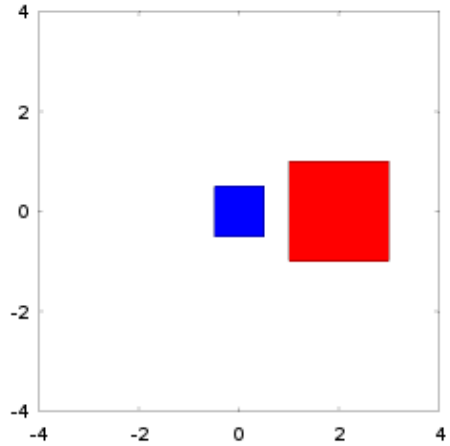
- l'*angle* sera spécifié en degrés
- la direction de rotation est définie par un vecteur (0,0,0)->*direction* (X,Y,Z) ;
  par exemple `[1 0 0]` définit une rotation autour de l'axe X, `[1 1 0]` autour de la bissectrice entre X et Y, etc..
- *origine* définit les coordonnées (X,Y,Z) du point de rotation

S'il s'agit d'un graphique 2D, une rotation dans le plan X/Y s'exprime par une rotation autour de l'axe Z, donc en utilisant la *direction* `[0 0 1]`

Ex 2a

```
xcarre=[-1 1 1 -1]; ycarre=[-1 -1 1 1];

fill(xcarre/2,ycarre/2,'b')  % petit carré bleu
hold('on')
 h =fill(xcarre+2,ycarre,'r'); % grand carré rouge

axis([-4 4 -4 4])
axis('equal')

for k=1:36
   pause(0.1)
   rotate(h, [0 0 1], 10, [0 0 0])
   % on ne fait tourner que le carré rouge,
   % 36 fois de 10 degrés dans le plan horiz.
   % autour de l'origine, donc 360 degrés
   % (tour complet), avec pause 0.1 seconde
   % entre chaque chaque mouvement
end
```

Cliquer sur ce graphique pour voir l'animation !



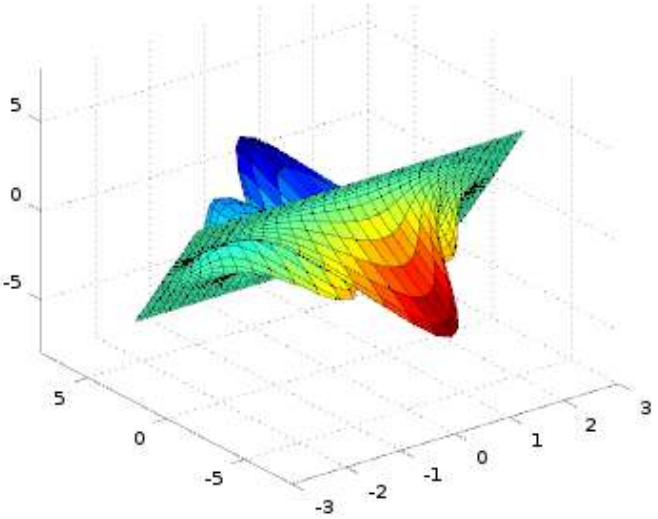
Remarque: cette animation a été capturée avec le logiciel libre recordMyDesktop, puis convertie en gif-animé (pour affichage dans navigateur web) avec l'outil libre ffmpeg

Ex 2b

```
peaks(-3:0.2:3, -4:0.2:4)
 h =get(gca,'children');  % handle de la surface

axis([-3 3 -8 8 -8 8])
```

Cliquer sur ce graphique pour voir l'animation !

```
for k=1:180
    pause(0.02)
    rotate(h, [1 0 0], 1, [0 0 0])
    % chaque 0.02 secondes rotation de 1 degré
    % autour de l'axe X et l'origine [0, 0, 0] ;
    % on fait 180 rotations, donc cela donne
    % un retournement complet haut-bas
end
```



Remarque: cette animation a été capturée avec le logiciel libre recordMyDesktop, puis convertie en gif-animé (pour affichage dans navigateur web) avec l'outil libre ffmpeg

`pan off|on|xon|yon`

`off` désactive complètement les possibilités de déplacememt (*pan*) interactif dans la figure, **`on`** les réactive. **`xon`** n'autorise le déplacement interactif qu'en X, et **`yon`** qu'en Y.

`rotate3d off|on`

`off` désactive la possibilité de rotation 3D interactive dans la figure, **`on`** la réactive.

a) **`set(gca,'Xtick | Ytick | Ztick', [`***`valeurs`***`] )`**
b) **`set(gca,'XTickLabel | YTickLabel | ZTickLabel',`** ***`labels`***`)`

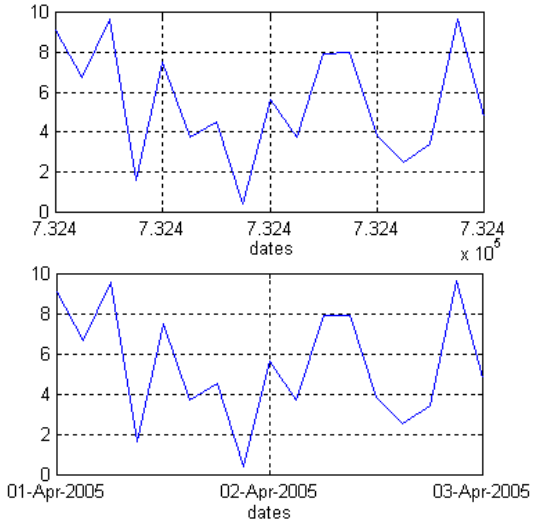
Graduation des axes et lignes de grille :
Commandes basées sur la technique des "Handle Graphics". On utilise ici la fonction **`gca`** (get current axes) qui retourne le "handle" du graphique courant
a) Spécifie les *valeurs* (suite de valeurs en ordre coissant), sur l'*axe* indiqué, auxquelles il faut : dessiner un 'tick' sur l'axe, afficher la valeur, et faire partir une ligne de grille
b) Spécifie le texte à afficher (label) en regard de chaque tick. Le paramètre *labels* peut être un vecteur de nombres, une matrice de chaînes , un tableau cellulaire de chaînes. Commande particulièrement utile si l'on veut graduer l'axe avec des chaînes (p.ex. des dates formatées...). **Important** : le nombre de *valeurs* et de *labels* doit être identique !

Voir aussi la fonction **`datetick`** pour graduer/formater les axes temporels

Ex 3

Ne vous attardez pas sur la syntaxe de la commande plot qui sera décrite plus loin au chapitre "Graphiques 2D"

```
date_debut = datenum('01-Apr-2005');
date_fin   = date_debut + 2;  % 2 jours plus tard

x=date_debut:0.125:date_fin;  % série toutes 3 h.
y=10*rand(1,length(x));

plot(x,y);
grid('on');   % => premier graphique ci-contre

xlabel('dates');

x_tick=date_debut:1:date_fin; % tous 1 jours (24 h)
 set(gca,'Xtick',x_tick)
 set(gca,'XTickLabel', ...
     datestr(x_tick,'dd-mmm-yyyy'))
            % => second graphique ci-contre
```



a) **`grid('on | off')`** ou **`grid on | off`**
b) **`grid`**

a) Activation/désactivation de l'affichage du **quadrillage** (grid). Par défaut le quadrillage d'un nouveau graphique n'est pas affiché.

b) Sans paramètre, cette fonction agit comme une bascule on/off.

Ex: voir l'exemple 1 ci-dessus

`box('on|off')`

Activation/désactivation de l'affichage, autour du graphique, d'un **cadre** (graphiques 2D) ou d'une "**boîte**" (graphiques 3D).
Sans paramètre, cette fonction agit comme une bascule on/off.

`xlabel('label_x' {,'propriété','valeur'...} )`
`ylabel('label_y' {,'propriété','valeur'...} )`
`zlabel('label_z' {,'propriété','valeur'...} )`

Définit et affiche le texte de **légende des axes** X, Y et Z (étiquettes, labels). Par défaut les axes d'un nouveau graphique n'ont pas de labels.
Des attributs, au sens "**Handle Graphics**" (couples *propriété* et *valeur*), permettent de spécifier la police, taille, couleur...

Ex: voir l'exemple 3 ci-dessous

---

a) `legend('legende_t1 ','legende_t2'... {, pos} )`
b) `legend('legende_t1 ','legende_t2'... {, 'Location','loc'} )`
c) `legend('off')`

a) Définit et place une **légende** sur le graphique en utilisant les textes spécifiés pour les tracés *t1*, *t2*...
La position de la légende est définie par le paramètre `pos` : **0**= Automatic (le moins de conflit avec tracés), **1**= angle haut/droite, **2**= haut/gauche, **3**= bas/gauche, **4**=bas/droite, **-1**= en dehors à droite de la zone graphée.
b) Autre manière plus explicite de positionner la légende : le paramètre `loc` peut être :

- pour légende DANS le graphique : **North**= haut, **South**= bas, **East**= droite, **West**= gauche, **NorthEast**= haut/droite, **NorthWest**= haut/gauche, **SouthEast**= bas/droite, **SouthWest**= bas/gauche, **Best**= position générant le moins de conflit avec les données graphées
- pour légende En DEHORS du graphique : **NorthOutside**= haut, **SouthOutside**= bas, **EastOutside**= droite, **WestOutside**= gauche, **NorthEastOutside**= haut/droite, **NorthWestOutside**= haut/gauche, **SouthEastOutside**= bas/droite, **SouthWestOutside**= bas/gauche, **BestOutside**= position générant le moins de conflit avec les données graphées

c) Désactive l'affichage de la légende

Ex: voir l'exemple 3 ci-dessous

`title('titre' {,'propriété','valeur'...} )`

Définit un **titre de graphique** qui est placé au-dessus de la zone graphée. Un nouveau graphique n'a par défaut pas de titre. Pour effacer le titre, définir une chaîne *titre* vide.
Si vous désirez créer un titre composé de **plusieurs lignes**, procédez ainsi : `title( sprintf('1ere ligne\n2eme ligne') )`
Des attributs, au sens "**Handle Graphics**" (couples *propriété* et *valeur*), permettent de spécifier la police, taille, couleur...

Ex: voir l'exemple 3 ci-dessous

---

a) `text(x, y, { z,} 'chaîne' {,'propriété','valeur'...} )`
b) `gtext('chaîne' {,'propriété','valeur'...} )`

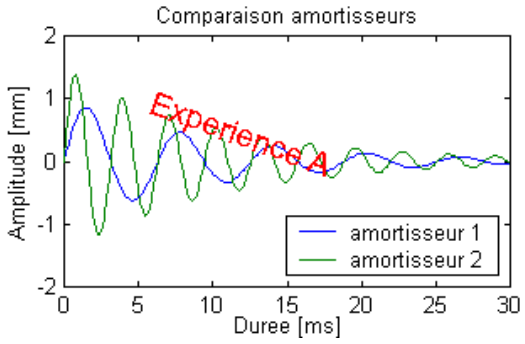
a) Place l'**annotation** *chaîne* dans le graphique aux coordonnées *x*, *y* {,*z*} spécifiées.
Pour afficher un texte composé de **plusieurs lignes**, procédez ainsi : `text(x, y, sprintf('1ere ligne\n2eme ligne') )`
b) L'emplacement du texte dans le graphique est défini interactivement par un clic souris-gauche.

Lorsqu'on utilise plusieurs fois ces fonctions, cela ajoute à chaque fois un nouveau texte au graphique.
Des attributs, au sens "**Handle Graphics**" (couples *propriété* et *valeur*), permettent de spécifier la police, taille, couleur, orientation...
Qt F La propriété `Rotation` n'est implémentée sous Octave 4.0 qu'à l'impression ou sauvegarde de fichier graphique, mais pas à l'affichage !

Ex 4

Ne vous attardez pas sur la syntaxe de la commande plot qui sera décrite plus loin au chapitre "Graphiques 2D"

```
x=linspace(0,30,200);
y1=sin(x)./exp(x/10); y2=1.5*sin(2*x)./exp(x/10);
plot(x,y1,x,y2);
 xlabel('Duree [ms]'); ylabel('Amplitude [mm]');
 title('Comparaison amortisseurs');
 legend('amortisseur 1','amortisseur 2',4);
 text(6,1,'Experience A', ...
         'FontSize',14,'Rotation',-20, ...
         'Color','red');
```



a) `annotation('line', x, y)`
b) `annotation('arrow', x, y)`
c) `annotation('doublearrow', x, y)`
d) `annotation('textarrow', x, y)`
e) `annotation('textbox', pos)`
f) `annotation('rectangle', pos)`
g) `annotation('ellipse', pos)`

Place dans la figure une **annotation**. ATTENTION: notez que la position (*x, y, pos*) est définie en **unités normalisées** par rapport à la fenêtre de figure, c'est-à-dire dans un système de coordonnées dont l'angle inférieur gauche est (0,0) et l'angle supérieur droite (1,1). (Donc pas dans le système d'axes du graphique comme le texte placé avec `text`).
L'annotation sera donc fixe par rapport à la fenêtre de figure et ne bouge pas si l'on fait un pan/zoom/rotation du graphique.

a) Place un texte
b) Place une flèche
c) Place une double flèche
d) Place un texte associé à une flèche
e) Place un texte dans une boîte
f) Place un rectangle
g) Place une ellipse

a) `whitebg()`
b) `whitebg(couleur)`
c) `whitebg('none')`

Change la **couleur de fond** du graphique :
a) Inversion du schéma de couleur, agissant comme une bascule
b) Le fond est mis à la *couleur* spécifiée sous forme de nom (p.ex. `'yellow'`) ou de triplet RGB (p.ex. `[0.95 0.95 0.1]`)
c) Rétablit le schéma de couleur par défaut

Afficher du texte mis en forme (indices, exposants, caractères spéciaux...)

M `texlabel('expression')`

Sous MATLAB, convertit au format TeX l'*expression* spécifiée. Cette fonction est généralement utilisée comme argument dans les commandes `title`, `xlabel`, `ylabel`, `zlabel`, et `text` pour afficher du texte incorporant des indices, exposants, caractères grecs...

Ex: M `text(15,0.8,texlabel('alpha*sin(sqrt(x^2 + y^2))/sqrt(x^2 + y^2)'));` affiche:
$\alpha \sin(\sqrt{x^2 + y^2})/\sqrt{x^2 + y^2}$

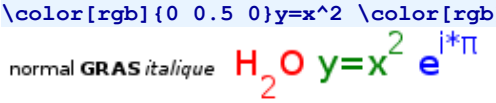
O Qt F Sous Octave, un interpréteur TeX est implémenté dans les backends graphiques Qt et FLTK. Toute fonction affichant du texte peut faire usage des codes de formatage suivants :

- `{ }` : permet de grouper des caractères pour leur appliquer un code de formatage
- `\bf` : ce qui suit sera en gras
- `\it` : ce qui suit sera en italique
- `\rm` : retour à fonte normale (annulation du gras ou de l'italique)
- `^` : le caractère suivant (ou groupe) est mis en exposant
- `_` : le caractère suivant (ou groupe) est mis en indice
- `\car` : permet d'afficher un caractère grec défini par son nom *car*
- `\fontname{fonte}` : utilise la *fonte* spécifiée pour le texte qui suit

`\fontsize{`*`taille`*`}` : affiche le texte qui suit dans la *taille* spécifiée
`\color[rgb]{`*`red green blue`*`}` : affiche le texte qui suit dans la couleur spécifiée par le triplet {*red green blue*}

Notez cependant que ces fonctionnalités ne sont pour l'instant supportées **qu'à l'affichage** et non à l'impression ou sauvegarde d'un fichier graphique !

Ex: `title('normal \bfGRAS\rm {\ititalique} \fontsize{18} \color[rgb]{1 0 0}H_2O \color[rgb]{0 0.5 0}y=x^2 \color[rgb]{0 0 1}e^{i*\pi}')` affiche :



6.1.5 Graphiques superposés, côte-à-côte, ou fenêtres graphiques multiples

Par défaut, MATLAB/Octave envoie tous les ordres graphiques à la **même fenêtre** graphique (appelée "figure"), et chaque fois que l'on dessine un **nouveau graphique** celui-ci **écrase le graphique précédent**. Si l'on désire tracer **plusieurs graphiques**, MATLAB/Octave offrent les possibilités suivantes :

A. **Superposition** de plusieurs tracés de type analogue dans le même graphique en utilisant le même système d'axes (*overlay plots*)
B. Tracer les différents graphiques **côte-à-côte**, dans la même fenêtre mais dans des axes distincts (*multiple plots*)
C. Utiliser des **fenêtres distinctes** pour chacun des graphiques (*multiple windows*)

A) Superposition de graphiques dans le même système d'axes ("overlay plots")

Fonction et description

Exemple | **Illustration**

a) `hold('on')` ou `hold on`
b) `hold('off')` ou `hold off`

a) Cette commande indique à MATLAB/Octave d'accumuler (**superposer**) les ordres de dessin qui suivent dans la même figure (pour empêcher qu'un nouveau tracé efface le précédent). Elle peut être passée avant tout tracé ou après le premier ordre de dessin. Dans les modes "multiple plots" ou "multiple windows" (voir plus bas), l'état on/off de hold est mémorisé indépendemment pour chaque sous-graphique, resp. chaque fenêtre de figure
b) Après cette commande, MATLAB/Octave est remis dans le mode par défaut, c'est-à-dire que tout nouveau graphique effacera le précédent. En outre, les annotations et attributs de graphique précédemment définis (labels x/y/z, titre, légende, état on/off de la grille...) sont bien évidemment effacés.

Remarque: les 2 primitives de base de tracé de lignes `line` et de surfaces remplies `patch`, de même que la fonction `rectangle`, permettent de dessiner par "accumulation" dans un graphique sans que `hold` doive être mis à `on` !

`ishold`

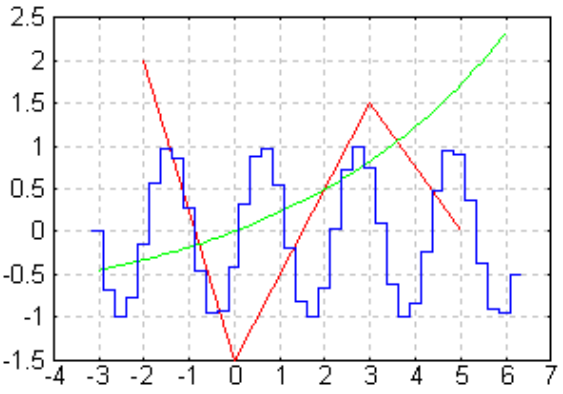
Retourne l'état courant du mode hold pour la figure active ou le sous-graphique actif : `0` (false) si hold est off, `1` (true) si hold est on.

Ex

Ne vous attardez pas sur la syntaxe des commandes plot, fplot et stairs qui seront décrites plus loin au chapitre "Graphiques 2D"

```
x1=[-2 0 3 5]; y1=[2 -1.5 1.5 0];
plot(x1,y1,'r');                % rouge
 hold('on');
fplot('exp(x/5)-1',[-3 6],'g');  % vert
x3=-pi:0.25:2*pi; y3=sin(3*x3);
stairs(x3,y3,'b');              % bleu
grid('on');
```

Vous constaterez que :
• on superpose des graphiques de types différents (plot, fplot, stairs...)
• ces graphiques ont, en X, des plages et des nombres de valeurs différentes



B) Graphiques côte-à-côte dans la même fenêtre ("multiple plots")

Fonction et description

Exemple | **Illustration**

`subplot(L,C,i)`

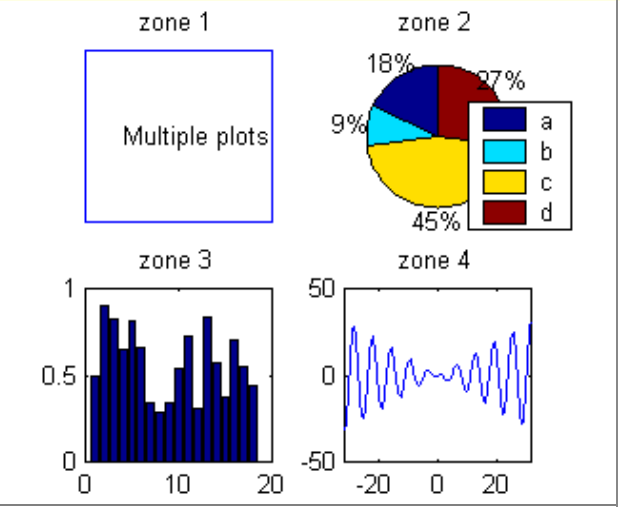
Découpe la fenêtre graphique courante (créée ou sélectionnée par la commande **`figure(numero)`**, dans le cas où l'on fait du "multiple windows") en `L` lignes et `C` colonnes, c'est-à-dire en *L* x *C* espaces qui disposeront chacun leur propre système d'axes (mini graphiques). **Sélectionne** en outre la `i`-ème zone (celles-ci étant numérotées ligne après ligne) comme espace de tracé courant.

- Si aucune fenêtre graphique n'existe, cette fonction en ouvre automatiquement une
- Si l'on a déjà une fenêtre graphique simple (i.e. avec 1 graphique occupant tout l'espace), le graphique sera effacé !
- Dans une fenêtre donnée, une fois le "partitionnement" effectué (par la 1ère commande **`subplot`**), on ne devrait plus changer les valeurs *L* et *C* lors des appels subséquents à **`subplot`**, faute de quoi on risque d'écraser certains sous-graphiques déjà réalisés !

Ex

Ne vous attardez pas sur la syntaxe des commandes plot, pie, bar et fplot qui seront décrites plus loin au chapitre "Graphiques 2D"

```
subplot(2,2,1);
plot([0 1 1 0 0],[0 0 1 1 0]);
text(0.2,0.5,'Multiple plots');
axis('off'); legend('off'); title('zone 1');
subplot(2,2,2);
pie([2 1 5 3]); legend('a','b','c','d');
title('zone 2');
subplot(2,2,3);
bar(rand(18,1)); title('zone 3');
subplot(2,2,4);
fplot('x*cos(x)',[-10*pi 10*pi]);
title('zone 4');
```



`linkaxes(handle_axes, 'x' | 'y' | 'xy' | 'off')`

Lie les limites des systèmes d'axes spécifiés par le vecteur de handles *handle_axes* de façon que les changements effectués dans un graphique (pan, zoom) soient reportés dans les autres. Ne s'applique qu'à des graphiques 2D.
Le second paramètre **`'x'`** lie les axes X seulement, **`'y'`** lie les axes Y seulement, **`'xy'`** lie les axes X et Y, **`'off'`** désactive cette liaison.

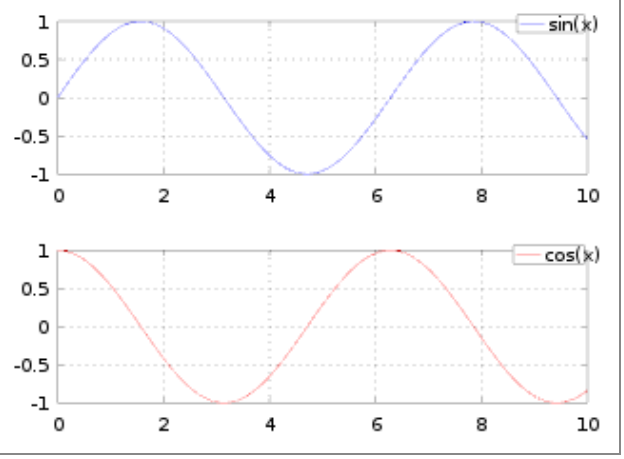
`linkprop(handle_axes, propriété1, propriété2 ...)`

De façon analogue, lie les *propriétés* spécifiées entre les différents graphiques

Ex

```
h1 =subplot(2,1,1);
fplot('sin(x)',[0 10],'b'); grid('on')

h2 =subplot(2,1,2);
fplot('cos(x)',[0 10],'r'); grid('on')

linkaxes ([ h1 h2 ],'x')   % ne lie ici que les axes X
```

puis jouez avec le pan et zoom de l'un des 2 graphiques...



C) Graphiques multiples dans des fenêtres distinctes ("multiple windows")

Fonction et description
a) **`figure`** b) **`figure(numero)`** **a)** Ouvre une **nouvelle fenêtre** de graphique (figure), et en fait la fenêtre de tracé active (dans laquelle on peut ensuite faire du "single plot" ou du "multiple plots"). Ces fenêtres sont automatiquement numérotées 1, 2, 3...

b) Si la fenêtre de *numero* spécifié existe, en fait la **fenêtre de tracé active**. Si elle n'existe pas, ouvre une nouvelle fenêtre de graphique portant ce *numero*.

`gcf` *(**g**et **c**urrent **f**igure)*
Retourne le *numero* de la fenêtre de graphique active (qui correspond, dans ce cas là, au *handle* de la figure)

6.1.6 Autres commandes de manipulation de fenêtres graphiques ("figures")

Fonction et description

`refresh` ou `refresh(numero)`
Raffraîchit (redessine) le(s) graphique(s) dans la fenêtre de figure courante, respectivement la fenêtre de *numéro* spécifié

`clf` ou `clf(numero)` *(**cl**ear **f**igure)*
Efface le(s) graphique(s) dans la fenêtre de figure courante, respectivement la fenêtre de *numéro* spécifié
Remet en outre `hold` à **off** s'il était à **on**, mais conserve la table de couleurs courante.

`cla` *(**cl**ear **a**xis)*
Dans le cas d'une fenêtre de graphique en mode "multiple plots", cette commande n'efface que le sous-graphique courant.

a) `close`
b) `close(numero)`
c) `close all`
a) Referme la fenêtre graphique active (figure courante)
b) Referme la fenêtre graphique de *numero* spécifié
c) Referme toutes les fenêtre graphique !
Met `hold` à **off** s'il n'y a plus de fenêtre graphique

`shg` *(**sh**ow **g**raphic)*
Fait passer la fenêtre de figure MATLAB courante **au premier plan**.
Cette commande est sans effet avec Octave sous Windows.

6.1.7 Traits, symboles et couleurs de base par 'linespec'

Plusieurs types de graphiques présentés plus bas utilisent une syntaxe, initialement définie par MATLAB et maintenant aussi reprise par Octave 3, pour spécifier le **type**, la **couleur** et l'**épaisseur** ou **dimension** de **trait** et de **symbole**. Il s'agit du paramètre `linespec` qui est une combinaison des caractères définis dans le tableau ci-dessous (voir M `help linespec`).

Le symbole M indique que la spécification n'est valable que pour **MATLAB**, le symbole Qt indique qu'elle n'est valable que pour Octave/**Qt**, le symbole F indique qu'elle n'est valable que pour Octave/**FLTK**, le symbole G indique qu'elle n'est valable que pour Octave/**Gnuplot** . Sinon c'est valable pour tous les grapheurs/backends !

Il est possible d'utiliser la fonction `[L,C,M,err]=colstyle('linespec')` pour tester un `linespec` et le décoder sur 3 variables séparées `L` (type de ligne), `C` (couleur) et `M` (marker). Si `linespec` est erroné, une erreur `err` est retournée.

Pour un rappel sur l'ancienne façon de spécifier les propriétés de lignes sous Octave 2.x, suivre **ce lien**.

Couleur ligne et/ou symbole

Caractère	Effet
`y`	jaune (yellow)
`m`	magenta
`c`	cyan
`r`	rouge (red)
`g`	vert clair (green)
`b`	bleu (blue)
`w`	blanc (white)
`k`	noir (black)

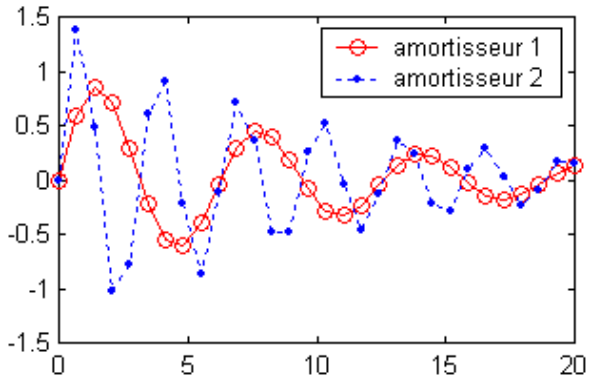
Type de ligne

Caractère	Effet
(rien)	affichage d'une ligne continue, sauf si un symbole est spécifié (auquel cas le symbole est affiché et pas la ligne)
`-`	ligne continue
M Qt F `--`	ligne traitillée
M Qt F `:`	ligne pointillée
M Qt F `-.`	ligne trait-point

Symbole (marker)

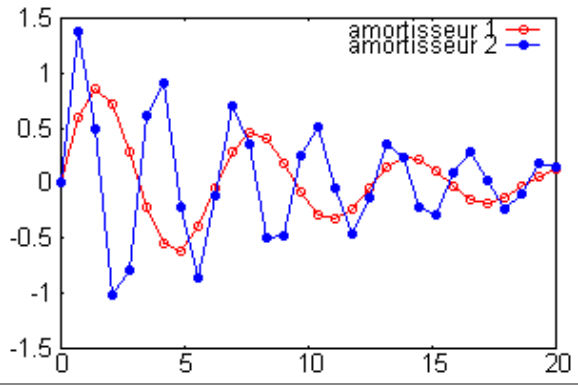
Caractère	Effet
(rien)	pas de symbole
`o`	cercle
`*`	étoile de type astérisque
`+`	signe plus
`x`	croix oblique (signe fois)
`.` (point)	M F petit disque rempli Qt G symbole point
M Qt F `^` `<` `>` `v` G `<` `v` G `>` `^`	M Qt F triangle (orienté selon symbole) G triangle pointé vers le bas vide/rempli G triangle pointé vers le haut vide/rempli
`s`	carré vide (square) (G rempli)
`d`	losange vide (diamond) (G rempli)
`p`	M Qt F étoile à 5 branches (pentagram) G carré vide
`h`	M Qt F étoile à 6 branches (hexagram) G losange vide

Ci-dessous, exemples d'utilisation de ces spécifications `linespec` .

Exemple	Illustration
Ex 1 sous **MATLAB**, **Octave/Qt** et **Octave/FLTK** `x=linspace(0,20,30);` `y1=sin(x)./exp(x/10); y2=1.5*sin(2*x)./exp(x/10);` `plot(x,y1, 'r-o' ,x,y2, 'b:.' );` `legend('amortisseur 1','amortisseur 2');`	

Ex 2 sous **Octave/Gnuplot**

```
% même code que ci-dessus
```



On verra plus loin (chapitre 3D "Vraies couleurs, tables de couleurs et couleurs indexées") qu'il est possible d'utiliser beaucoup plus de couleurs en spécifiant des "**vraies couleurs**" sous forme de triplets RGB (valeurs d'intensités `[red green blue]` de 0.0 à 1.0), ou en travaillant en mode "**couleurs indexées**" via une "**table de couleurs**" (colormap). Les couleurs ainsi spécifiées peuvent être utilisées avec la propriété `'color'` de la commande `set` (voir chapitre qui suit), commande qui permet de définir également plus librement l'épaisseur et le type de **trait**, ainsi que le type de **symbole** et sa dimension.

Pour **définir de façon plus fine** les types de traits, symboles et couleurs, on utilisera la techique des "handles" décrite ci-après dans le chapitre "**Handle Graphics**".

6.1.8 Interaction souris avec une fenêtre graphique

Il est possible d'interagir entre MATLAB/Octave et un graphique à l'aide de la souris.

On a déjà vu plus haut la fonction `gtext('chaîne')` qui permet de **placer interactivement** (à l'aide de la souris) une **chaîne** de caractère dans un graphique.

`[x, y {,bouton}] = ginput(n)`

Attend que l'on clique *n* fois dans le graphique à l'aide de souris-gauche, et retourne les vecteurs-colonne des **coordonnées** *x* et *y* des endroits où l'on a cliqué, et facultativement le numéro de *bouton* de la souris qui a été actionné (1 pour souris-gauche, 2 pour souris-milieu, 3 pour souris-droite).
Si l'on omet le paramètre *n*, cette fonction attend jusqu'à ce que l'on frappe enter dans la figure.

6.2 Graphiques 2D

Sous **MATLAB**, la liste des fonctions relatives aux graphiques 2D est accessible via `help graph2d` et `help specgraph`.
Concernant **Octave/Gnuplot**, on se réfèrera au chapitre "Plotting" du Manuel Octave (HTML ou PDF).

6.2.1 Dessin de graphiques 2D

Fonction et description	
Exemple	**Illustration**

a) `plot(x1, y1 {,linespec} {, x2, y2 {,linespec} ...} )`
`plot(x1, y1 {,'PropertyName',PropertyValue} ... )`
b) `plot(vect)`
c) `plot(mat)`
d) `plot(var1,var2 ...)`

Graphique 2D de lignes et/ou semis de points sur axes linéaires :
a) Dans cette forme (la plus courante), *xi* et *yi* sont des **vecteurs** (ligne ou colonne), et le graphique comportera autant de courbes indépendantes que de paires *xi*/*yi*. Pour une paire donnée, les vecteurs *xi* et *yi* doivent avoir le même nombre d'éléments (qui peut cependant être différent du nombre d'éléments d'une autre paire). Il s'agit d'un 'vrai graphique X/Y' (graduation de l'axe X selon les valeurs fournie par l'utilisateur).

Avec la seconde forme, définition de propriétés du graphiques plus spécifiques (voir l'exemple parlant du chapitre précédent !)

b) Lorsqu'une seule variable *vect* (de type **vecteur**) est définie pour la courbe, les valeurs *vect* sont graphées en Y, et c'est l'indice de chaque valeur qui est utilisé en X (1, 2, 3 ... n). Ce n'est donc plus un 'vrai graphique X/Y' mais un graphique dont les points sont uniformément répartis selon X.
c) Lorsqu'une seule variable *mat* (de type **matrice**) est passée, chaque **colonne** de *mat* fera l'objet d'une courbe, et chacune des courbes s'appuiera en X sur les valeurs 1, 2, 3 ... n (ce ne sera donc pas non plus un 'vrai graphique X/Y')
d) Lorsque l'on passe des paires de valeurs de type **vecteur/matrice**, **matrice/vecteur** ou **matrice/matrice** :

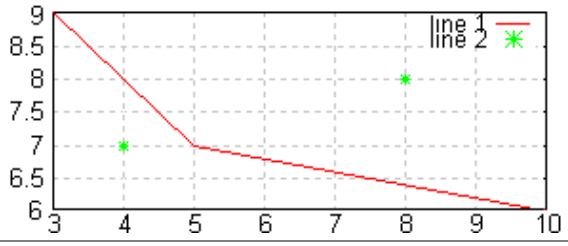
- si *var1* est un vecteur (ligne ou colonne) et *var2* une matrice :
  - si le nombre d'éléments de *var1* correspond au nombre de colonnes de la matrice *var2*, chaque **ligne** de *var2* fera l'objet d'une courbe, et chaque courbe utilisera le vecteur *var1* en X
  - si le nombre d'éléments de *var1* correspond au nombre de lignes de la matrice, chaque **colonne** de *var2* fera l'objet d'une courbe, et chaque courbe utilisera le vecteur *var1* en X
  - sinon, erreur !
- nous ne décrivons pas les autres cas (*var1* est une matrice et *var2* un vecteur, ou tous deux sont une matrice) qui sont très rares

Voir en outre (plus bas dans ce support de cours) :
• pour des graphiques à 2 axes Y : fonction `plotyy`
• pour des graphiques avec axes logarithmiques : les fonctions `semilogx`, `semilogy` et `loglog`
• pour des graphiques en semis de point avec différenciation de symboles sur chaque point : fonction `scatter`
• pour tracer des courbes 2D/3D dans un fichier au format AutoCAD DXF : fonction `dxfwrite` du package "plot" (qui ne semble cependant plus maintenu)

Ex 1 selon forme a) ci-dessus

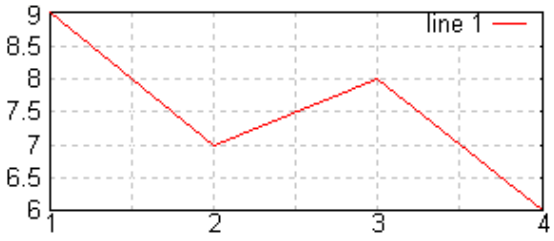
```
plot([3 5 6 10], [9 7 NaN 6], ...
     [4 8], [7 8], 'g*')
```

Remarque importante : lorsque l'on a des valeurs manquantes, on utilise `NaN`



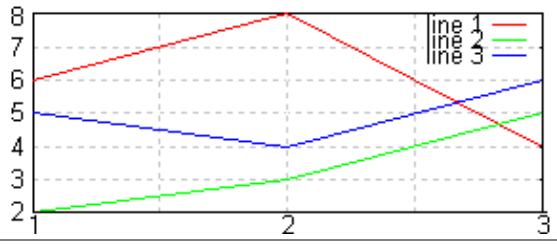
Ex 2 selon forme b) ci-dessus

```
plot([9 ; 7 ; 8 ; 6]);
```



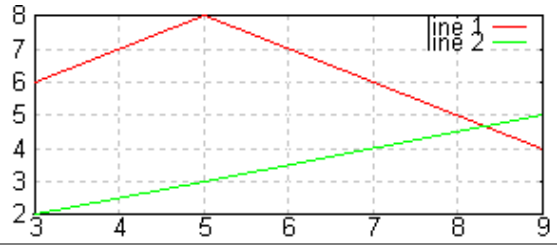
Ex 3 selon forme c) ci-dessus

```
plot([6 2 5 ; 8 3 4 ; 4 5 6]);
```



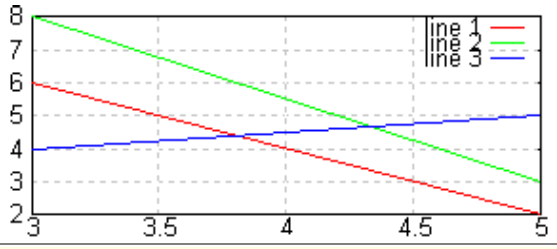
Ex 4.1 selon forme d)1 ci-dessus

```
plot([3 5 9], [6 8 4 ; 2 3 5]);
```



Ex 4.2 selon forme d)2 ci-dessus

```
plot([3 5], [6 8 4 ; 2 3 5]);
```



a) `fplot('fonction', [xmin xmax] {, nb_points } {, linespec } )` (***f**unction **plot**)*

b) `fplot('[fonction1, fonction2, fonction3 ...]', [xmin xmax] ...)`

Graphique 2D de fonctions y=fct(x) :

a) Trace la *fonction* fct(x) spécifiée entre les limites *xmin* et *xmax*.

b) Trace simultanément les différentes *fonctions* spécifiées (remarquez bien la notation entre crochets)

Par rapport à `plot`, il n'y a dans ce cas pas besoin d'échantillonner les valeurs x et y de la fonction (i.e. définition d'un vecteur x puis du vecteur y=fct(x) ...), car `fplot` accepte en argument les 2 méthodes de définition de *fonction* suivantes :

- chaîne de caractère exprimant une **fonction de x** : voir **Ex 1**
- nom d'une **fonction MATLAB/Octave** existante : voir **Ex 2**
- nom d'une **fonction utilisateur** définie sous forme de M-file (voir chapitre **fonctions**) : voir **Ex 3**

Le paramètre optionnel *linespec* permet de spécifier un type particulier de lignes et/ou symboles.

Sous **Octave**, la fonction est échantillonnée (de façon interne) par défaut sur 100 points, ou sur le nombre *nb_points* spécifiés

Sous **MATLAB**, la fonction est échantillonnée (de façon interne) par défaut sur un nombre de points qui varie selon la fonction et l'intervalle ; l'usage de *nb_points*, en-dessous d'une certaine valeur, n'a pas d'effet.

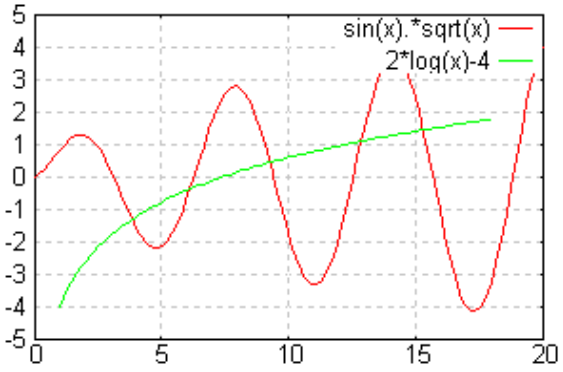
Voir encore la fonction `ezplot` (*easy plot*) qui permet de dessiner une fonction 2D définie sous sa **forme paramétrique**. A titre d'exemple, voyez la fonction `ezplot3` plus bas.

Ex 1

```
fplot('sin(x)*sqrt(x)', [0 20],'r');
hold('on');
fplot('2*log(x)-4',[1 18],'g');
grid('on');
```

ou

```
fplot('[sin(x)*sqrt(x),2*log(x)-4]', ...
     [0 20],'b');
grid('on');
ylim([-5 5])
```



Important : Notez qu'il est ici suffisant de faire le produit sin(x) `*` sqrt(x), donc avec l'opérateur `*` sans devoir utiliser le produit élément par élément `.*`. En effet, l'expression que l'on passe ainsi à `fplot` sera évaluée de façon interne point après point pour différentes valeurs de x, et non pas appliquée à un vecteur x.

Remarque : constatez, dans la 1ère solution, que l'on a

superposé les graphiques des 2 fonctions dans des plages de valeurs en X qui sont différentes !

Ex 2

Grapher des fonctions built-in MATLAB/Octave :

```
fplot('sin',[0 10],'r');
hold('on');
fplot('cos',[0 10],'g');
grid('on');
```

Ex 3

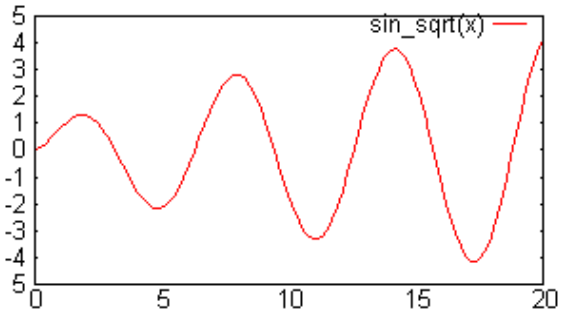
Définir une fonction utilisateur, puis la grapher :

1) Définition, dans un fichier nommé `sin_sqrt.m`, de la fonction suivante (voir chapitre **fonctions**) :

```
function [Y]=sin_sqrt(X)
  Y=sin(X).*sqrt(X);
return
```

> Remarque: sous **Octave**, on pourrait aussi, au lieu de saisir le code de la fonction ci-dessus dans un M-file, l'entrer **interactivement** dans la fenêtre de commande Octave en terminant la saisie par `endfunction` (au lieu de `return`), ce qui donne lieu à une "compiled function".

2) Puis la grapher simplement avec :
`fplot('sin_sqrt(x)',[0 20],'r')`



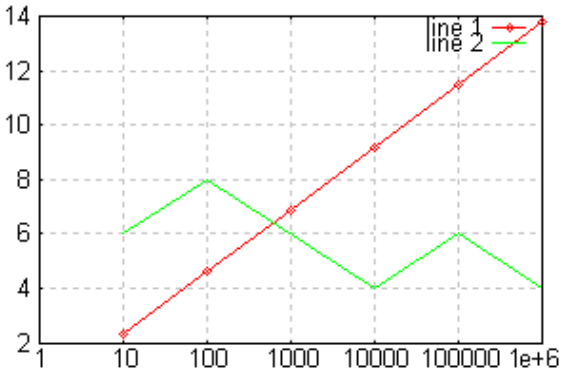
a) `semilogx(...)`
b) `semilogy(...)`
c) `loglog(...)`

Graphique 2D de lignes et/ou semis de points sur axes logarithmiques :
Ces 3 fonctions, qui s'utilisent exactement comme la fonction `plot` (mêmes paramètres...), permettent de grapher dans des systèmes d'axes logarithmiques :
a) axe X logarithmique, axe Y linéaire
b) axe X linéaire, axe Y logarithmique
c) axes X et Y logarithmiques

Voir aussi, plus bas, la fonction `plotyy` pour graphiques 2D à 2 axes Y qui peuvent être logarithmiques

Ex

```
x1=logspace(1,6,6); y1=log(x1);
semilogx(x1,y1,'r-o', ...
         [10 100 1e4 1e5 1e6],[6 8 4 6 4],'g');
grid('on');
```



`plotyy(x1, y1, x2, y2 {,'type1' {,'type2'}} )`

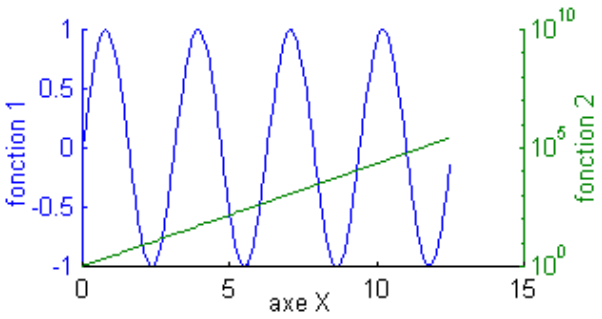
Graphique avec 2 axes Y distincts :
Trace la courbe définie par les vecteurs *x1* et *y1* relativement à l'axe Y de gauche, et la courbe définie par les vecteurs *x2* et *y2* relativement à l'axe Y de droite.
Les paramètres optionnels *type1* et *type2* permettent de définir le type de primitive de tracé 2D utiliser. Ils peuvent

notamment être : **plot** , **semilogx** , **semilogy** , **loglog** , **stem** ...

Ex

```
x=0:0.1:4*pi;
h= plotyy(x, sin(2*x), x, exp(x), ...
          'plot', 'semilogy');
xlabel('axe X');

hy1=get(h(1),'ylabel');
hy2=get(h(2),'ylabel');
set(hy1,'string','fonction 1');
set(hy2,'string','fonction 2');
```

Remarque : nous devons ici utiliser la technique des 'handles' (ici variables h, hy1 et hy2) pour étiqueter les 2 axes Y



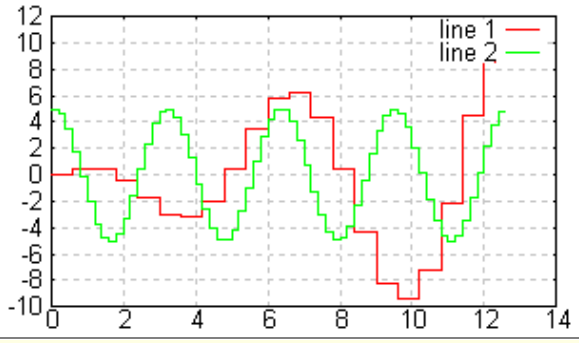
a) **stairs({*x*,} *y*)**
b) **stairs({*x*,} *ymat* {, *linespec* })**

Graphique 2D en escaliers :
a) Dessine une ligne en escaliers pour la courbe définie par les vecteurs (ligne ou colonne) *x* et *y*. Si l'on ne fournit pas de vecteur *x*, la fonction utilise en X les indices de *y* (donc les valeurs 1 à length(*y*)).
b) Traçage de plusieurs courbes sur le même graphique en passant à cette fonction une matrice *ymat* dans laquelle chaque courbe fait l'objet d'une colonne.

Remarque : on peut aussi calculer le tracé de la courbe sans le dessiner avec l'affectation **[*xs*,*ys*]=stairs(...);** , puis le dessiner ultérieurement avec **plot(*xs*,*ys*,*linespec*);**

Ex

```
x1=0:0.6:4*pi; y1=x1.*cos(x1);
 stairs(x1,y1,'r');
hold('on');
x2=0:0.2:4*pi; y2=5*cos(2*x2);
 stairs(x2,y2,'g');
grid('on');
```

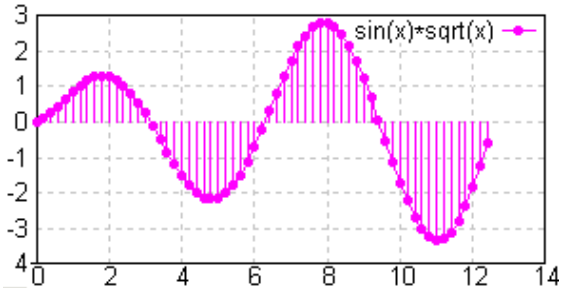


stem({*x*,} *y* {, *linespec* })

Graphique 2D en bâtonnets :
Graphe la courbe définie par les vecteurs (ligne ou colonne) *x* et *y* en affichant une ligne de rappel verticale (bâtonnet, pointe) sur tous les points de la courbe. Si l'on ne fournit pas de vecteur *x*, la fonction utilise en X les indices de *y* (donc les valeurs 1 à length(*y*)).

Ex

```
x=0:0.2:4*pi; y=sin(x).*sqrt(x);
 stem(x,y,'mo-');
grid('on');
```

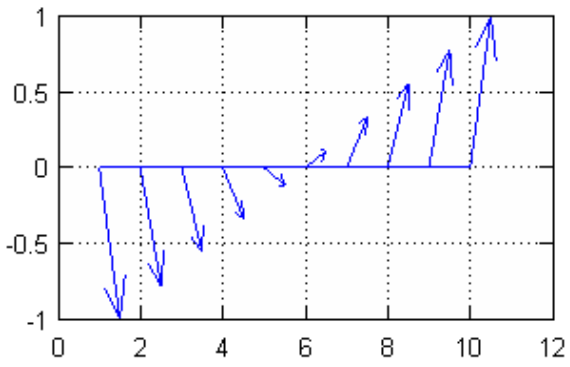


feather({*dx*,} *dy*)

Graphique 2D de champ de vecteurs en "plumes" :
Dessine un champ de vecteurs dont les origines sont uniformément réparties sur l'axe X (en (1,0), (2,0), (3,0), ...) et dont les dimensions/orientations sont définies par les valeurs *dx* et *dy*

Ex

```
dy=linspace(-1,1,10) ;
dx=0.5*ones(1,length(dy)) ;
   % vecteur ne contenant que des val. 0.5
 feather(dx,dy)
grid('on')
axis([0 12 -1 1])
```

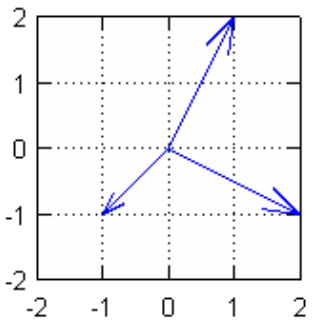


`compass( {dx,} dy)`

Graphique 2D de champ de vecteurs de type "boussole" :
Dessine un champ de vecteurs dont les origines sont toutes en (0,0) et dont les dimensions/orientations sont définies par les valeurs *dx* et *dy*

Ex

```
compass([1 2 -1],[2 -1 -1])
axis([-2 2 -2 2])
grid('on')
```



a) `errorbar(x, y, error {,format} )`
b) `errorbar(x, y, lower, upper {,format} )`

Graphique 2D avec barres d'erreur :
a) Graphe la courbe définie par les vecteurs de coordonnées *x* et *y* (de type ligne ou colonne, mais qui doivent avoir le même nombre d'éléments) et ajoute, à cheval sur cette courbe et en chaque point de celle-ci, des barres d'erreur verticales symétriques dont la longueur totale sera le double de la valeur absolue des valeurs définies par le vecteur *error* (qui doit avoir le même nombre d'éléments que *x* et *y*).
b) Dans ce cas, les barres d'erreur seront **asymétriques**, allant de :

- M *y*-abs(*lower*) à *y*+abs(*upper*)
- O *y*-*lower* à *y*+*upper* (donc Octave utilise le signe des valeurs contenus dans ces vecteurs !)

Attention : le paramètre *format* a une signification différente selon que l'on utilise MATLAB ou Octave :

- M il correspond simplement au paramètre `linespec` (spécification de couleur, type de trait, symbole...) comme dans la fonction `plot`
- O la fonction `errorbar` de Octave offre davantage de possibilités que celle de MATLAB : ce paramètre *format* **doit commencer** par l'un des codes ci-dessous définissant le type de barre(s) ou box d'erreur à dessiner :
  - `~` : barres d'erreur verticales (comme sous MATLAB)
  - `>` : barres d'erreur **horizontales**
  - `~>` : barres d'erreur en X et en Y (il faut alors fournir **4 paramètres** `lowerX, upperX, lowerY, upperY` !)
  - `#~>` : dessine des "**boxes**" d'erreur

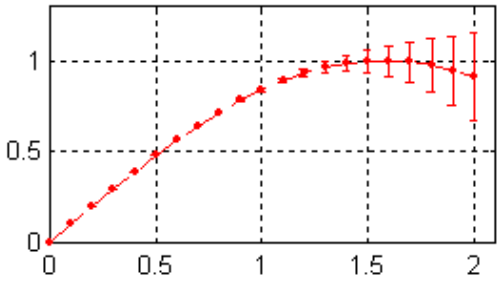
  puis se poursuit par le `linespec` habituel, le tout entre apostrophes

Voir en outre les fonctions suivantes, spécifiques à Octave : O `semilogxerr` , O `semilogyerr` , O `loglogerr`

Ex 1

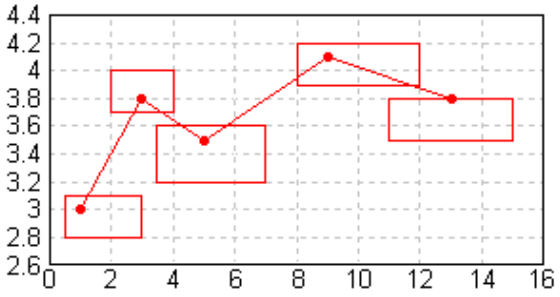
```
x=0:0.1:2; y=sin(x);
y_approx = x - (x.^3/6); % approximation fct sinus
error = y_approx - y;
 errorbar(x,y,error,'r--o');
grid('on');
```

Remarque : on illustre ci-dessus la différence entre la fonction sinus et son approximation par un polynôme



Ex 2 (graphique ci-contre réalisé avec Octave)

```
x=[1 3 5 9 13];
y=[3 3.8 3.5 4.1 3.8];
lowerX=[0.5 1 1.5 1 2];
upperX=[2 1 2 3 2];
lowerY=[0.2 0.1 0.3 0.2 0.3];
upperY=[0.1 0.2 0.1 0.1 0];

 errorbar(x,y, ...
   lowerX,upperX,lowerY,upperY,'#~>r');
hold('on');
plot(x,y,'r-o');
legend('off');
grid('on');
```



`scatter(` *x, y* **`{,`** *size* **`{,`** *color* **`} } {,`** *symbol* **`} {,'filled'} )`**

Graphique 2D de symboles :
Dessin du semis de points défini par les vecteurs de coordonnées *x* et *y* (de type ligne ou colonne, mais qui doivent avoir le même nombre d'éléments)

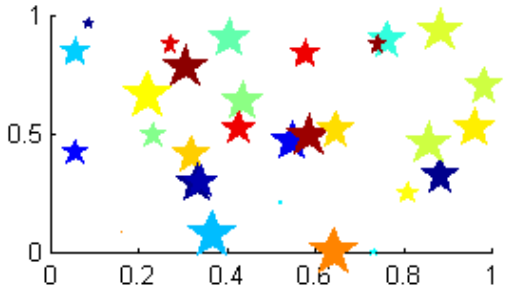
- *size* permet de spécifier la **surface** des symboles : ce peut être soit une valeur scalaire (=> tous les symboles auront la surface spécifiée), soit un vecteur de même dimension que *x* et *y* (=> indique alors taille de chaque symbole) ; concernant l'unité de ce paramètre :
  M surface du "carré englobant" du symbole en [pixels^2] : ex: 100 => symbole de surface 100 pixels^2 donc de coté 10 x 10 [pixels]
  O largeur et hauteur du "carré englobant" du symbole en [pixels]
- *color* permet de spécifier la couleur des symboles : ce peut être :
  • soit **une** couleur, appliquée uniformément à tous les symboles, exprimée sous forme de chaîne selon la syntaxe décrite plus haut (p.ex. `'r'` pour rouge)
  • soit un vecteur (de la même taille que *x* et *y*) qui s'appliquera linéairement à la colormap
  • ou une matrice n x 3 de couleurs exprimées en composantes RGB
- *symbol* permet de spécifier le type de symbole (par défaut: cercle) selon les possibilités décrites plus haut, c'est-à-dire `'o'` , `'*'` , `'+'` , `'x'` , `'^'` , `'<'` , `'>'` , `'v'` , `'s'` , `'d'` , `'p'` , `'h'`
- le paramètre-chaîne `'filled'` provoquera le remplissage des symboles

Remarque : en jouant avec l'attribut *color* et en choisissant une table de couleur appropriée, cette fonction permet de grapher des données 3D x/y/color

Ex: (graphique ci-contre réalisé avec MATLAB ou Octave/FLTK)

```
if ~ exist('OCTAVE_VERSION')
  facteur=50*50 ; % MATLAB
else
  facteur=50 ;    % Octave
end

scatter(rand(30,1),rand(30,1), ...
        facteur*rand(30,1),rand(30,1),'p','filled');
```

Remarque : nous graphons donc ici 30 paires de nombres x/y aléatoires compris entre 0 et 1 ; de même, nous définissons la couleur et la taille des symboles de façon aléatoire



`area( {` *x,* **`}`** *ymat* **`)`**

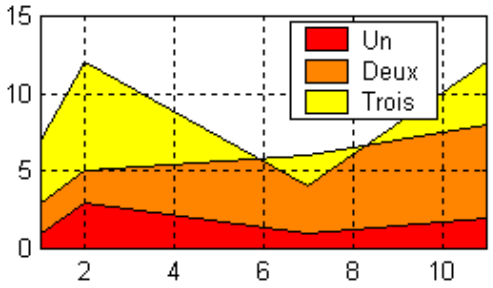
Graphique 2D de type surface :
Graphe de façon empilée (cumulée) les différentes courbes définies par les colonnes de la matrice *ymat*, et colorie les surfaces entre ces courbes. Le nombre d'éléments du vecteur *x* (ligne ou colonne) doit être identique au nombre de

lignes de *ymat*. Si l'on ne spécifie pas *x*, les valeurs sont graphées en X selon les indices de ligne de *ymat*.
Remarque : si on ne veut pas "empiler" les surfaces, on utilisera plutôt la fonction `fill`

sous Octave Qt et FLTK 3.4 à 4.0, l'usage de la fonction `colormap` est sans effet

Ex

```
x=[1 2 7 11];
ymat=[1 2 4 ; 3 2 7 ; 1 5 -2 ; 2 6 4];
 area(x,ymat);
colormap(autumn); % changement palette couleurs
grid('on');
set(gca,'Layer','top'); % quadrillage 1er plan
legend('Un','Deux','Trois')
```



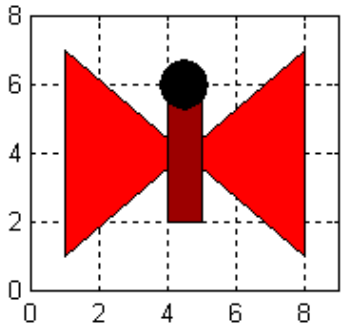
a) `fill(x, y, couleur)`
b) `fill(xA, yA, couleurA {, xB, yB, couleurB ... } )`
c) `patch(x, y, couleur)`

Dessin 2D de surface(s) polygonale(s) remplie(s) :
a) Dessine et rempli de la *couleur* spécifiée le polygone défini par les vecteurs de coordonnées *x* et *y*. Le polygone bouclera automatiquement sur le premier point, donc il n'y a pas besoin de définir un dernier couple de coordonnées xn/yn identique à x1/y1.
b) Il est possible de dessiner plusieurs polygones (A, B...) d'un coup en une seule instruction en passant en paramètre à cette fonction plusieurs triplets *x*,*y*,*couleur*.
c) Primitive de bas niveau de tracé de surfaces remplies, cette fonction est analogue à `fill` sauf qu'elle accumule (tout comme la primitive de dessin de ligne `line`) son tracé dans la figure courante sans qu'il soit nécessaire de faire au préalable un `hold('on')`

On spécifie la `couleur` par l'un des codes de couleur définis plus haut (p.ex. pour rouge: `'r'` ou `[1.0 0 0]`)

Ex

```
a=linspace(0,2*pi,20);
  x= 4.5 + 0.7*cos(a); % contour disque noir de
  y= 6.0 + 0.7*sin(a); % rayon 0.7, centre 4.5/6.0
 fill([1 8 8 1],[1 7 1 7],'r', ...
      [4 5 5 4],[2 2 6 6],[0.6 0 0], ...
      x,y,'k');
axis('equal');
axis([0 9 0 8]);
grid('on');
```



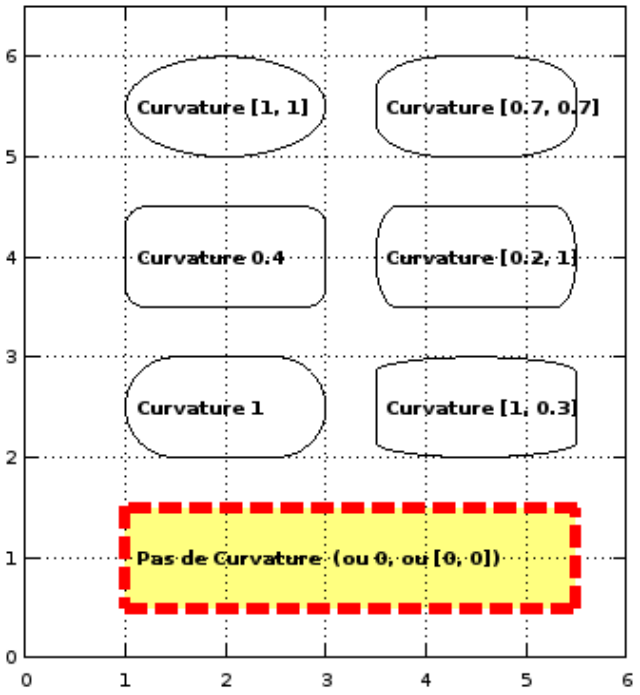
`rectangle('Position',[xmin, ymin, L, H] {, 'Curvature', c | [c_horiz, c_vert] }`
`{,'LineWidth',epaisseur, 'LineStyle',type, 'EdgeColor',coul_trait,`
`'FaceColor',coul_remplissage } )`

Dessin 2D de formes allant du rectangle à l'ellipse :
Dessine une forme de largeur *L*, hauteur *H* et dont l'angle inférieur gauche est placé aux coordonnées (*xmin, ymin*).
Le paramètre `Curvature` permet de définir la courbure des cotés de la forme, à l'aide d'une valeur *c* ou par 2 valeurs [*c_horiz, c_vert*] qui doivent être comprises entre 0 et 1. La valeur `0` signifie aucune courbure des cotés (donc coins carrés) ; `1` signifie qu'il faut courber le coté sur toute sa longueur ; entre deux, la courbure n'est appliqué qu'aux extrémités des cotés. On a ainsi, par exemple :
• avec c= `0` ou c= `[0,0]` (ou en l'absence du paramètre Curvature) : un **rectangle**
• avec c= `1` : les 2 petits cotés de la forme sont des **demi-cercles**
• avec c= `[1,1]` : une **ellipse** parfaite
Il est encore possible d'agir sur l'*epaisseur* du bord de la forme, le *type* de trait, sa couleur *coul_trait*, et la couleur de remplissage *coul_remplissage*.

Notez finalement que cette fonction, qui s'appuie sur la fonction `patch` , accumule son tracé dans la figure courante sans qu'il soit nécessaire de faire au préalable un `hold('on')`

Ex

```
rectangle('position',[1 0.5 4.5 1], ...
   'linewidth',6, 'linestyle','--', ...
   'edgecolor','r', 'facecolor',[1 1 0.5])
text(1.1,1,'Pas de Curvature  (ou 0, ou [0, 0])')

rectangle('position',[1 2 2 1],'curvature',1)
text(1.1,2.5,'Curvature 1')

rectangle('position',[1 3.5 2 1],'curvature',0.4)
text(1.1,4,'Curvature 0.4')

rectangle('position',[1 5 2 1],'curvature',[1 1])
text(1.1,5.5,'Curvature [1, 1]')

rectangle('position',[3.5 2 2 1], ...
   'curvature',[1 0.3])
text(3.6,2.5,'Curvature [1, 0.3]')

rectangle('position',[3.5 3.5 2 1], ...
   'curvature',[0.2 1])
text(3.6,4,'Curvature [0.2, 1]')

rectangle('position',[3.5 5 2 1], ...
   'curvature',[0.7 0.7])
text(3.6,5.5,'Curvature [0.7, 0.7]')

axis([0 6 0 6.5])
axis('equal')
grid('on')
```

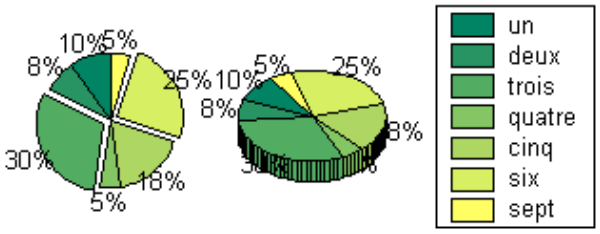


a) `pie(val {,explode} {,labels} )`
b) `pie3(val {,explode} {,labels} )`

Graphique de type camembert :
a) Dessine un camembert 2D sur la base du vecteur `val`, chaque valeur se rapportant à une tranche de gâteau.
Le vecteur logique `explode` (de même taille que *val* et composé de 0 ou de 1) permet de spécifier (avec 1) quelles tranches de gâteau doivent être "détachées"
Le vecteur cellulaire `labels` (de même taille que *val* et composé de chaînes de caractères) permet de spécifier le texte à afficher à coté de chaque tranche en lieu et place des pourcentages
b) Réalise un camembert en épaisseur (3D)

Ex

```
val=[20 15 60 10 35 50 10];

subplot(1,2,1);
 pie(val, [0 0 1 0 0 1 0]);
colormap(summer);  % changement palette couleur

subplot(1,2,2);
 pie3(val);
legend('un','deux','trois','quatre', ...
       'cinq','six','sept', ...
       'location','east');
```



a) `bar( {x,} y )`
b) `bar( {x,} mat {,larg} {,'style'} )`
c) `barh( {y,} mat {,larg} {,'style'} )`

Graphique 2D en barres :
a) Dessine les barres verticales définies par les vecteurs *x* (position de la barre sur l'axe horizontal) et *y* (hauteur de la barre). Si le vecteur *x* n'est pas fourni, les barres sont uniformément réparties en X selon les indices du vecteur *y* (donc positionnées de 1 à n).
b) Sous cette forme, on peut fournir une matrice *mat* dans laquelle chaque ligne définira un groupe de barres qui seront dessinées :

- côte-à-côte si le paramètre *style* n'est pas spécifié ou que sa valeur est `'grouped'`
- de façon empilée si la valeur de ce paramètre est `'stacked'`

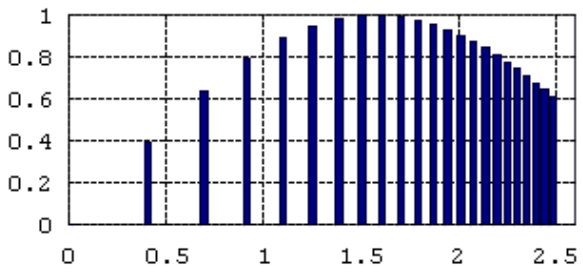
Avec le paramètre *larg*, on spécifie le rapport "largeur des barres / distance entre barres" dans le cadre du groupe ; la valeur par défaut est 0.8 ; si celle-ci dépasse 1, les barres se chevaucheront. Le nombre d'éléments du vecteur *x* doit être égal au nombre de lignes de la matrice *mat*.

c) Identique à la forme b), sauf que les barres sont dessinées horizontalement et positionnées sur l'axe vertical selon les valeurs du vecteur *y*

Remarque : on peut aussi calculer les tracés sans les dessiner avec l'affectation `[xb,yb]=bar(...);`, puis les dessiner ultérieurement avec `plot(xb,yb,linespec);`

Ex 1

```
x=log(1:0.5:12);
y=sin(x);
bar(x,y);
axis([0 2.6 0 1]);
grid('on');
```



Ex 2

Premier graphique ci-contre :

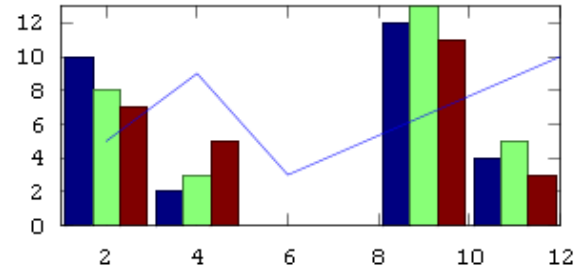
```
x=[2 4 9 11];
mat=[10 8 7 ; 2 3 5 ; 12 13 11 ; 4 5 3];
bar(x,mat,0.9,'grouped');
hold('on');
plot([2 4 6 12],[5 9 3 10]);
```

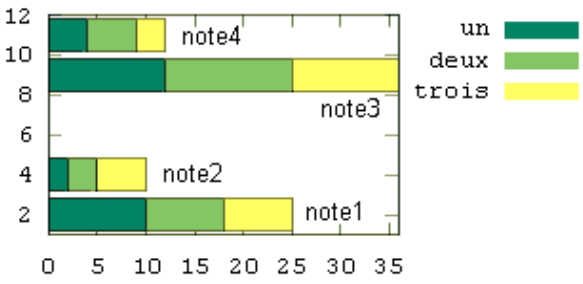
Second graphique ci-contre :

```
barh(x,mat,0.8,'stacked');
legend('un','deux','trois',-1)
colormap(summer)
```

On a ensuite **annoté** le second graphique, en placement interactivement des chaînes de texte définies dans un tableau avec le code ci-dessous :

```
annot={'note1','note2','note3','note4'};
for n=1:length(annot)
   gtext(annot{n});
end
```





a) [*nval* {*xout*}] = `hist(y {,n} )`
b) [*nval* {*xout*}] = `hist(y, x )`

Histogramme 2D de distribution de valeurs, ou calcul de cette distribution :
a) Détermine la **répartition** des valeurs contenues dans le vecteur *y* (ligne ou colonne) selon *n* catégories (par défaut 10) de même 'largeur' (catégories appelées boîtes, bins, ou containers), puis dessine cette répartition sous forme de graphique 2D en barres où l'axe X reflète la plage des valeurs de *y*, et l'axe Y le nombre d'éléments de *y* dans chacune des catégories.
IMPORTANT: Si l'on affecte cette fonction à `[nval {xout}]`, le graphique n'est **pas** effectué, mais la fonction retourne le vecteur-ligne *nval* contenant nombre de valeurs trouvées dans chaque boîte, et le vecteur-ligne *xout* contenant les valeurs médianes de chaque boîtes. On pourrait ensuite effectuer le graphique à l'aide de ces valeurs tout simplement avec la fonction `bar(xout,nval)`.
b) Dans ce cas, le vecteur *x* spécifie les valeurs du 'centre' des boîtes (qui n'auront ainsi plus nécessairement la même largeur !) dans lesquelles les valeurs de *y* seront distribuées, et l'on aura autant de boîtes qu'il y a d'éléments dans le vecteur *x*.

Voir aussi la fonction `[nval {vindex}]=histc(y,limits)` (qui ne dessine pas) permettant de déterminer la distribution des valeurs de *y* dans des catégories dont les 'bordures' (et non pas le centre) sont précisément définies par le vecteur *limits*.

Remarque : sous MATLAB, *y* peut aussi être une **matrice** de valeurs ! Si cette matrice comporte k colonnes, la fonction `hist` effectue k fois le travail en examinant les valeurs de la matrice *y* colonne après colonne. Le graphique contiendra alors *n* groupes de *k* barres. De même, la variable *nval* retournée sera alors une matrice de *n* lignes et k colonnes, mais *xout* restera un vecteur de *n* valeurs (mais, dans ce cas, en colonne).
Remarque : il existe sous Octave une variante de cette fonction nommée `hist2d` (dans le package "plot", qui ne semble cependant plus maintenu)

Voir (plus bas) la fonction `rose` qui réalise aussi des histogrammes de distribution mais dans un système de coordonnées polaire.

Ex

```
y=[4 8 5 2 6 8 0 6 13 14 10 7 4 3 12 13 6 3 5 1];
```

1) Si l'on ne spécifie pas n => n=10 catégories, et comme les valeur *y* vont de 0 à 14, les catégories auront une largeur de (14-0)/10 = 1.4, et leurs 'centres' *xout* seront respectivement : 0.7, 2.1, 3.5, 4.9, etc... jusqu'à 13.3

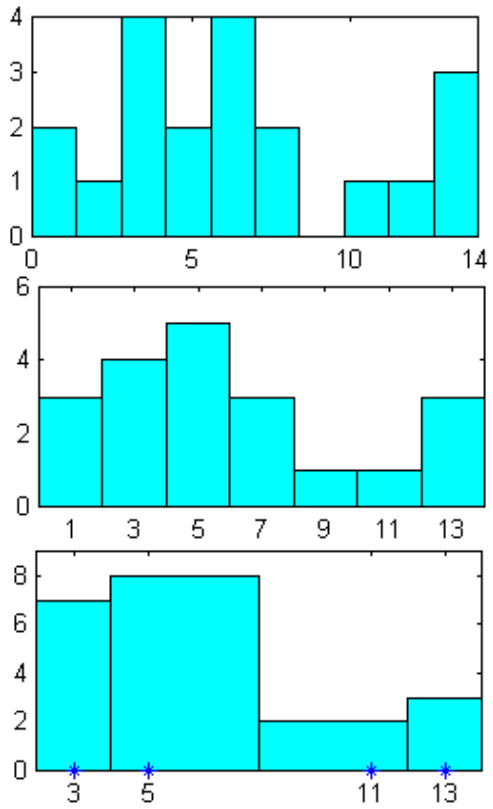
```
 [nval xout]=hist(y)
% => nval=[2 1 4 2 4 2 0 1 1 3]
%    xout=[0.7 2.1 3.5 4.9 6.3 7.7 9.1
%          10.5 11.9 13.3]
 hist(y);   % => 1er graphique ci-contre
set(gca,'XTick',xout) % annote axe X sous barres
```

2) Spécifions n=7 catégories => elles auront une largeur de (14-0)/7 = 2, et leurs 'centres' *xout* seront respectivement : 1, 3, 5, 7, 9, 11 et 13

```
 [nval xout]=hist(y,7)
% => nval=[3 4 5 3 1 1 3]
%    xout=[1 3 5 7 9 11 13]
 hist(y,7);   % => 2e graphique ci-contre
set(gca,'XTick',xout) % annote axe X sous barres
```

3) Spécifions un vecteur centres=[3 5 11 13] définissant les centres de 4 boîtes

```
 [nval xout]=hist(y,centres)
% => nval=[7 8 2 3]
%    xout=[3 5 11 13] % identique à centres
 hist(y,centres)  % => 3e graphique ci-contre
axis([2 14 0 9]);
set(gca,'XTick',centres) % annote axe X sous barres
```



a) **`plotmatrix(`***`m1, m2 {,linespec}`***`)`**

b) **`plotmatrix(`***`m {,linespec}`***`)`**

Matrice de graphiques en semis de points :

a) En comparant les **colonnes** de la matrice *m1* (de dimension P lignes x M colonnes) avec celles de *m2* (de dimension P lignes x N colonnes), affiche une matrice de N (verticalement) x M (horizontalement) graphiques en semis de points

b) Cette forme est équivalente à **`plotmatrix(`***`m, m {,linespec}`***`)`**, c'est à dire que l'on effectue toutes les comparaisons possibles, deux à deux, des colonnes de la matrice *m* et qu'on affiche une matrice de comportant autant de lignes et colonnes qu'ily a a de colonnes dans *m*. En outre dans ce cas les graphiques se trouvant sur la diagonale (qui représenteraient des semis de points pas très intéressants, car distribués selon une ligne diagonale) sont remplacés par des graphiques en histogrammes 2D (fréquence de distribution) correspondant à la fonction **`hist(m(:,`***`i`***`))`**

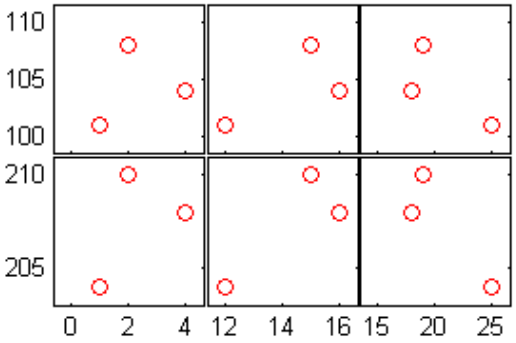
Ex

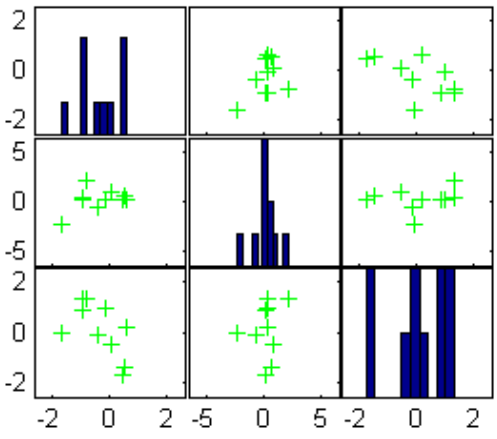
1) La fonction ci-dessous produit le premier graphique ci-contre

```
plotmatrix([1 12 25; 2 15 19; 4 16 18], ...
           [101 204; 108 210; 104 208], 'ro')
```

2) La fonction ci-dessous produit le second graphique ci-contre

```
plotmatrix( randn(10,3), 'g+')
```





`line(x, y {,z} {,'property', value } )`

Primitive de tracé de lignes 2D/3D :
Cette fonction est une primitive de tracé de **lignes** 2D/3D de bas niveau proche de `plot` et `plot3`. Elle s'en distingue cependant par le fait qu'elle permet d'accumuler, dans un graphique, des tracés sans qu'il soit nécessaire de mettre `hold` à `on` !

Remarque : la primitive de tracé de **surfaces remplies** de bas niveau est `patch`

Ex

```
hold('off'); clf;
for k=0:32
  angle=k*2*pi/32;
  x=cos(angle); y=sin(angle);
  if mod(k,2)==0
    coul='red'; epais=2;
  else
    coul='yellow'; epais=4;
  end
   line([0 x], [0 y], ...
        'Color', coul, 'LineWidth', epais);
end
axis('off'); axis('square');
```

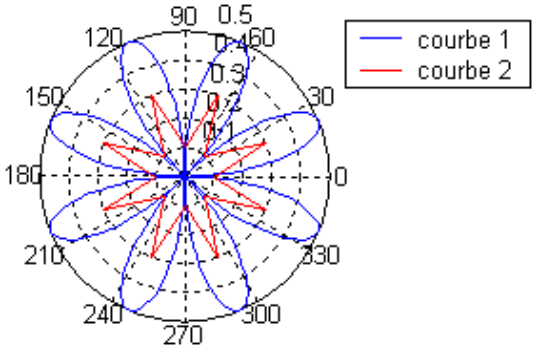
`polar(angle, rayon {,linespec} )`

Graphique 2D de lignes et/ou semis de points en coordonnées polaires :
Reçoit en paramètre les coordonnées polaires d'une courbe (ou d'un semis de points) sous forme de 2 vecteurs *angle* (en radian) et *rayon* (vecteurs ligne ou colonne, mais de même taille), dessine cette courbe sur une grille polaire.
On peut tracer plusieurs courbes en utilisant `hold('on')`, ou en passant à cette fonction des matrices *angle* et *rayon* (qui doivent être de même dimension), la i-ème courbe étant construite sur la base des valeurs de la i-ème colonne de *angle* et de *rayon*.

Voir aussi la fonction `ezpolar` qui permet de tracer, dans un système polaire, une fonction définie par une expression.
Voir en outre les fonctions `cart2pol` et `pol2cart` de conversion de coordonnées carthésiennes en coordonnées polaires et vice-versa.

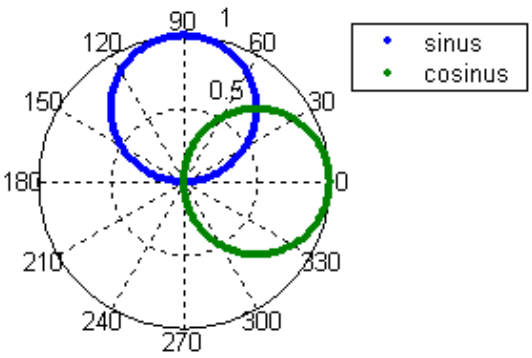
Ex 1

```
angle1=0:0.02:2*pi;
rayon1=sin(2*angle1).*cos(2*angle1);
 polar(angle1, rayon1, 'b');

angle2=0:pi/8:2*pi;
rayon2=[repmat([0.1 0.3],1,8), 0.1];
hold('on');
 polar(angle2, rayon2, 'r');

legend('courbe 1','courbe 2');
```



Ex 2

```
angle=0:0.02:2*pi;
polar([angle' angle'], ...
      [sin(angle') cos(angle')],'.');
legend('sinus','cosinus');
```



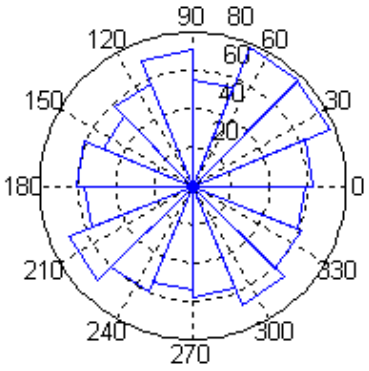
`rose(`*`val`* `{,`*`n`*`} )`

Histogramme polaire de distribution de valeurs (ou histogramme angulaire) :
Cette fonction est analogue à la fonction **`hist`** vue plus haut, sauf qu'elle travaille dans un système polaire angle/rayon. Les valeurs définies dans le vecteur *val*, qui doivent ici être **comprises entre 0 et 2*pi**, sont réparties dans *n* catégories (par défaut 20 si *n* n'est pas spécifié) et dessinées sous forme de tranche de gâteau dans un diagramme polaire où l'angle désigne la plage des valeurs, et le rayon indique le nombre de valeurs se trouvant dans chaque catégorie.

Ex

```
rose(2*pi*rand(1,1000),16);
```

Explications : on établit ici un vecteur de 1000 nombres aléatoires compris entre 0 et 2*pi, puis on calcule et dessine leur répartition en 16 catégories (1ère catégorie pour les valeurs allant de 0 à 2*pi/16, etc...).



6.3 Graphiques 2D½ et 3D

MATLAB/Octave offre un grand nombre de **fonctions de visualisation** permettant de représenter des **données 3D** sous forme de **graphiques 2D½** (vue plane avec représentation de la 3e dimension sous forme de courbes de niveau, champ de vecteurs, dégradés de couleurs...) ou de **graphiques 3D** (vue perspective). Ces données 3D peuvent être des points/**symboles**, des vecteurs, des **lignes**, des **surfaces** (par exemple fonction **z = fct(x,y)**) et des **tranches** de volumes (dans le cas de jeux de données 4D).

S'agissant des représentations perspectives 3D, et comme dans tout logiciel de CAO/modélisation 3D, différents types de **"rendu" des surfaces** sont possibles : "fil de fer" (mesh, wireframe), coloriées, ombrées (shaded surface). L'écran d'affichage ou la feuille de papier étant 2D, la **vue** finale d'un graphique 3D est obtenue par projection 3D->2D au travers d'une "**caméra**" dont l'utilisateur définit l'**orientation** et la **focale**, ce qui donne un effet de perspective.

La liste des fonctions relatives aux graphiques 3D est accessible sous **MATLAB** via `help graph3d` ainsi que `help specgraph`. Concernant **GNU Octave**, on se réfèrera au chapitre "Plotting" du Manuel Octave (HTML ou PDF), et à l'aide en-ligne pour les fonctions additionnelles apportées par Octave-Forge.

6.3.1 Fonctions auxiliaires de préparation/manipulation de données 3D

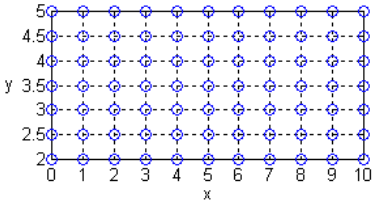
La fonction "meshgrid" de préparation de grilles de valeurs Xm et Ym

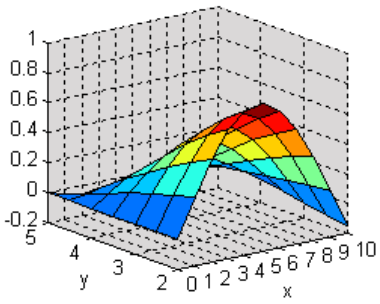
La fonction `meshgrid` est très souvent utilisée lorsqu'il s'agit de calculer le maillage d'une surface 3D. Pour démontrer son utilité et fonctionnement, prenons un exemple concret.

Donnée du problème :

Détermination et visualisation, par un graphique 3D, de la surface **z = fct(x,y) = sin(x/3)*cos(y/3)** en "échantillonnant" cette fonction selon une grille X/Y de dimension de **maille** 1 en X et 0.5 en Y, dans les plages de valeurs **0 ≤ `x` ≤ 10** (=> 11 valeurs) et **2 ≤ `y` ≤ 5** (=> 7 valeurs). Pour représenter graphiquement cette surface, il s'agit au préalable de calculer une matrice `z` dont les éléments sont les "altitudes" `z` correspondant aux points de la grille définie par les vecteurs `x` et `y`. Cette matrice aura donc (dans le cas du présent exemple) la dimension **7 x 11** (respectivement `length(y)` lignes, et `length(x)` colonnes).

```
    x  0 1 2 3 4 5 6 7 8 9 10
y     ------------------------
2     |                      |
2.5   |                      |
3     |       matrice  z     |
3.5   |        (7 x 11)      |
4     |                      |
4.5   |                      |
5     |                      |
      ------------------------
```





Solution 1 : méthode classique ne faisant pas intervenir les capacités vectorisées de MATLAB/Octave :

Cette solution s'appuie sur 2 boucles `for` imbriquées utilisées pour parcourir tous les points de la grille et de calculer individuellement chacun des éléments de la matrice `z`. C'est la technique classique utilisée dans les langages de programmation "non vectorisés", et son implémentation MATLAB/Octave correspond au code suivant :

```
x=0:1:10;  y=2:0.5:5;  % domaine des valeurs de la grille en X et Y

for k=1:length(x)      % parcours de la grille, colonne après colonne
  for l=1:length(y)    % parcours de la grille, ligne après ligne
    z1(l,k)= sin(x(k)/3)*cos(y(l)/3);  % calcul de la matrice z, élément après élément
  end
end

surf(x,y,z1);          % visualisation de la surface
```

Solution 2 : solution MATLAB/Octave vectorisée faisant intervenir la fonction `meshgrid` :

```
x=0:1:10;  y=2:0.5:5;     % domaine des valeurs de la grille en X et Y

 [Xm,Ym]=meshgrid(x,y);
 z2=sin(Xm/3).*cos(Ym/3);  % calcul de la matrice z en une seule instruction vectorisée
    % notez bien que l'on fait produit .* (élém. par élém.) et non pas * (vectoriel)
surf(x,y,z2);              % visualisation de la surface
```

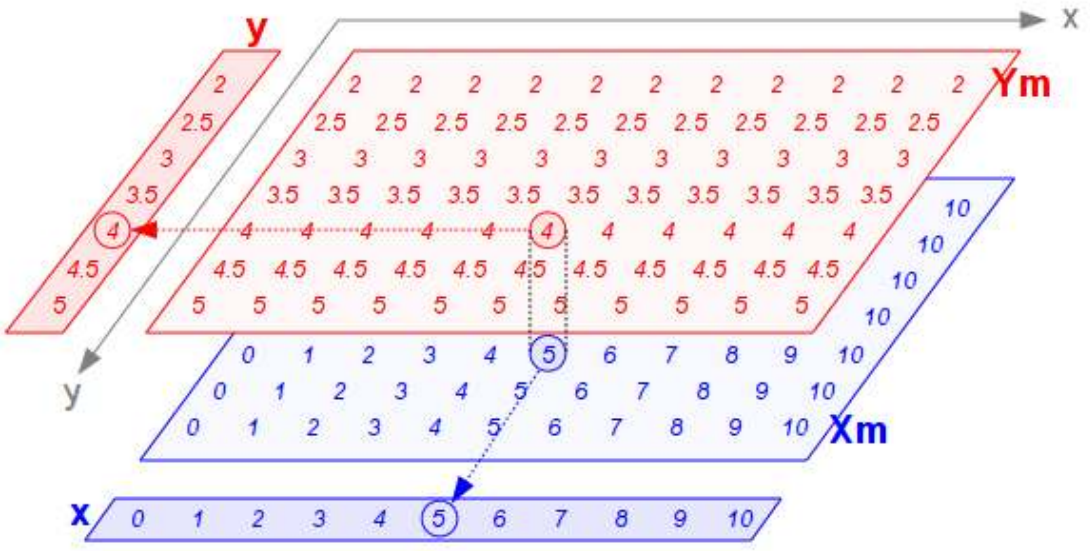
Remarque: dans le cas tout à fait particulier de cette fonction, on aurait aussi pu faire tout simplement

`z=cos(y'/3)*sin(x/3)` *(en transposant* `y` *et en utilisant le produit vectoriel). Nous vous laissons réfléchir pourquoi cela fonctionne ;-)*

Explications relatives au code de la solution 2 :

- sur la base des 2 vecteurs d'échantillonnage `x` et `y` (en ligne ou en colonne, peu importe!) décrivant le domaine des valeurs de la grille en X et Y, la fonction `meshgrid` génère **2 matrices** `Xm` et `Ym` **d'échantillonnage** (voir **figure** ci-dessous) qui ont les propriétés suivantes :
  - `Xm` est constituée par recopie, en `length(y)` lignes, du vecteur `x`
  - `Ym` est constituée par recopie, en `length(x)` colonnes, du vecteur `y`
  - elles ont donc toutes deux pour dimension `length(y)` lignes * `length(x)` colonnes (comme la matrice `z` que l'on s'apprête à déterminer)
- on peut par conséquent calculer la matrice z=fct(x,y) par une seule instruction MATLAB/Octave vectorisée (donc sans boucle `for`) en utilisant les 2 matrices `Xm` et `Ym` et faisant usage des opérateurs "terme à terme" tels que `+` , `-` , `.*` , `./` ... ; en effet, l'élément `z(ligne,colonne)` peut être exprimé en fonction de `Xm(ligne,colonne)` (qui est identique à `x(colonne)`) et de `Ym(ligne,colonne)` (qui est identique à `y(ligne)`)
- vous pouvez vérifier vous-même que les 2 solutions ci-dessus donnent le même résultat avec `isequal(z1,z2)` (qui retournera 1, indiquant que les matrices `z1` et `z2` sont rigoureusement identiques)
- pour grapher la surface avec les fonctions `mesh` , `meshc` , `surf` , `surfc` , `surfl` ..., il est important de noter que :
  - si l'on passe à ces fonctions le seul argument `z` (matrice d'altitudes), les axes du graphiques ne seront **pas** gradués en fonction des valeurs en X et Y, mais selon les indices des éléments de la matrice, c'est-à-dire de 1 à length(x) en X, et de 0 à length(y) en Y
  - pour avoir une **graduation correcte des axes X et Y** (i.e. selon les valeurs en X et Y), il est absolument nécessaire de passer à ces fonctions 3 arguments, à choix : `(x, y, z)` (vecteur, vecteur, matrice), ou `(Xm, Ym, z)` (matrice, matrice, matrice)

```
x = 0  1  2  3  4  5  6  7  8  9  10   (11 él.)   y = 2  2.5  3  3.5  4  4.5  5   (7 élém.)

    -------------(7x11)---------------                 -------------------------(7x11)-------------------------
   | 0  1  2  3  4  5  6  7  8  9  10 |               | 2    2    2    2    2    2    2    2    2    2    2   |
   | 0  1  2  3  4  5  6  7  8  9  10 |               | 2.5  2.5  2.5  2.5  2.5  2.5  2.5  2.5  2.5  2.5  2.5 |
   | 0  1  2  3  4  5  6  7  8  9  10 |               | 3    3    3    3    3    3    3    3    3    3    3   |
Xm=| 0  1  2  3  4  5  6  7  8  9  10 |          Ym=| 3.5  3.5  3.5  3.5  3.5  3.5  3.5  3.5  3.5  3.5  3.5 |
   | 0  1  2  3  4  5  6  7  8  9  10 |               | 4    4    4    4    4    4    4    4    4    4    4   |
   | 0  1  2  3  4  5  6  7  8  9  10 |               | 4.5  4.5  4.5  4.5  4.5  4.5  4.5  4.5  4.5  4.5  4.5 |
   | 0  1  2  3  4  5  6  7  8  9  10 |               | 5    5    5    5    5    5    5    5    5    5    5   |
    ---------------------------------                 -----------------------------------------------------
```



Description générale de la fonction `meshgrid` :

a) `[Xm,Ym] = meshgrid(x {,y} )`
b) `[Xm,Ym,Zm] = meshgrid(x, y, z)`

a) A partir des vecteurs *x* et *y* (de type ligne ou colonne) définissant le domaine de valeurs d'une grille en X et Y, génération

des matrices *Xm* et *Ym* (de dimension `length(y)` lignes * `length(x)` colonnes) qui permettront d'évaluer une fonction z=fct(x,y) (matrice) par une simple instruction vectorisée (i.e. sans devoir implémenter des boucles `for`) comme illustré dans la solution 2 de l'exemple ci-dessus. Il est important de noter que la grille peut avoir un nombre de points différent en X et Y et que les valeurs définies par les vecteurs *x* et *y* ne doivent pas nécessairement être espacées linéairement, ce qui permet donc de définir un **maillage absolument quelconque**.
Si le paramètre *y* est omis, cela est équivalent `meshgrid(x,x)` qui défini un maillage avec la même plage de valeurs en X et Y
La fonction `meshgrid` remplace la fonction `meshdom` qui est obsolète.
b) Sous cette forme, la fonction génère les tableaux tri-dimensionnels *Xm*,*Ym* et *Zm* qui sont nécessaires pour évaluer une fonction v=fct(x,y,z) et générer des graphiques 3D volumétriques (par exemple avec `slice` : voir exemple au chapitre "Graphiques 3D volumétriques").

`ndgrid(...)`
C'est l'extension à **n-dimension** de la fonction `meshgrid`

La fonction "griddata" d'interpolation de grille dans un semis irrégulier

Sous MATLAB/Octave, les fonctions classiques de visualisation de données 3D nécessitent qu'on leur fournisse une matrice de valeurs Z (à l'exception, en particulier, de `fill3` , `tricontour` , `trimesh` , `trisurf`). Or il arrive souvent, dans la pratique, que l'on dispose d'un **semis de points (x,y,z) irrégulier** (c-à-d. dont les coordonnées X et Y ne correspondent pas à une grille, ou que celle-ci n'est pas parallèle au système d'axes X/Y) provenant par exemple de mesures, et que l'on souhaite interpoler une surface passant par ces points et la grapher. Il est alors nécessaire de déterminer au préalable une **grille X/Y régulière**, puis d'**interpoler les valeurs Z** sur les points de cette grille à l'aide de la fonction d'interpolation 2D `griddata` , comme cela va être illustré dans l'exemple qui suit.
La fonction `interp2` se rapproche de `griddata` , mais elle interpole à partir d'une grille (matrice de valeurs Z), et non pas à partir d'un semis de points irrégulier.

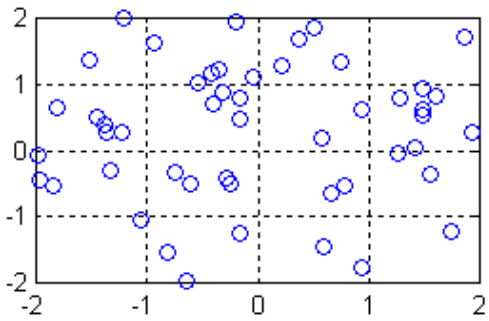
Donnée du problème :

Plutôt que d'entrer manuellement les coordonnées x/y/z d'une série de points irrégulièrement distribués, nous allons générer un **semis de point x/y irrégulier**, puis calculer la valeur z en chacun de ces points en utilisant une fonction z=fct(x,y) donnée, en l'occurence **z= x * exp(-x^2 -y^2)**. Nous visualiserons alors ce semis de points, puis effecuerons une **triangulation de Delaunay** pour afficher cette surface sous forme brute par des triangles. Puis nous utiliserons `griddata` pour interpoler une grille régulière dans ce semis de points, et nous visualiserons la surface correspondante.

Solution :

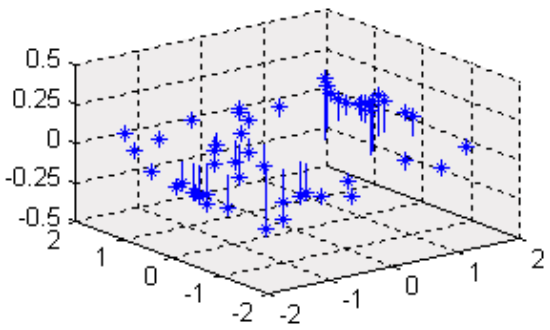
1) Génération aléatoire d'un **semis de points X/Y** irrégulier :

```
x= 4*rand(1,50) -2;  % vecteur de 50 val. entre -2 et +2
y= 4*rand(1,50) -2;  % vecteur de 50 val. entre -2 et +2

plot(x,y,'o');
grid('on');
axis([-2 2 -2 2]);
```



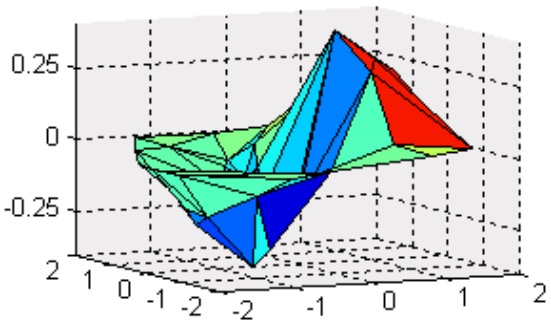
2) Calcul de la **valeur Z** (selon la fonction donnée) en chacun des points de ce semis :

```
z= x.*exp(-x.^2 - y.^2);  % calcul de Z en ces points

stem3(x,y,z,'-*');
grid('on');
```



3) Triangulation de Delaunay pour **afficher la surface brute** :

```
tri_indices= delaunay(x, y);     % formation triangles
                                 % => matrice d'indices
trisurf(tri_indices, x, y, z);   % affichage triangles
set(gca,'xtick',[-2 -1 0 1 2]);
set(gca,'ytick',[-2 -1 0 1 2]);
set(gca,'ztick',[-0.5 -0.25 0 0.25 0.5]);
```

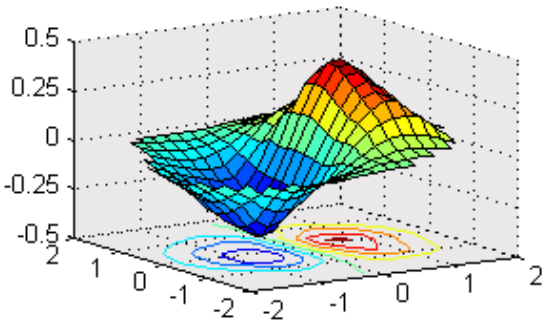


Sous Octave seulement, on aurait aussi pu dessiner dans un graphique 2D les **courbes de niveau** avec cette fonction du package "plot" (qui n'est cependant plus maintenu) :

```
tricontour(tri_indices, x, y, z, [-0.5:0.1:0.5], 'r-o')
```

4) Définition d'une **grille régulière X/Y**, et **interpolation** de la surface en ces points :

```
xi= -2:0.2:2;
yi= xi';   % ce doit être un vecteur colonne !!!
[XI,YI,ZI]= griddata(x,y,z,xi,yi,'linear');
                % interpolation sur grille
surfc(XI,YI,ZI); % affich. surf. interpolée et contours

% pour superposer sur ce graphique le semis de point :
hold('on');
plot3(x,y,z,'.');
```



Description générale des fonctions `griddata` et `interp2` :

`[XI,YI,ZI] = griddata(x,y,z, xi,yi {,methode} )`

Sur la base d'un **semis de points irrégulier** défini par les vecteurs *x*, *y*, *z*, interpole la surface *XI*,*YI*,*ZI* aux points de la grille spécifiée par le domaine de valeurs *xi* et *yi* (vecteurs). On a le choix entre 4 *methodes* d'interpolation différentes :

- `'linear'` : interpolation linéaire basée triangle (méthode par défaut), disontinuités de 0ème et 1ère dérivée
- `'nearest'` : interpolation basée sur le voisin le plus proche, disontinuités de 0ème et 1ère dérivée
- `'cubic'` : interpolation cubique basée triangle, surface lissée

`ZI = interp2(X,Y,Z, xi,yi {,methode} )`

Par opposition à `griddata`, cette fonction d'interpolation 2D s'appuie sur une **grille de valeurs** définies par les **matrices** *X*, *Y* et *Z* (*X* et *Y* devant être passées au format produit par `meshgrid`). Voir l'aide en-ligne pour davantage de détails.

Le cas échéant, voir la fonction d'interpolation 3D `interp3`, et la fonction d'interpolation multidimensionnelle `interpn`.

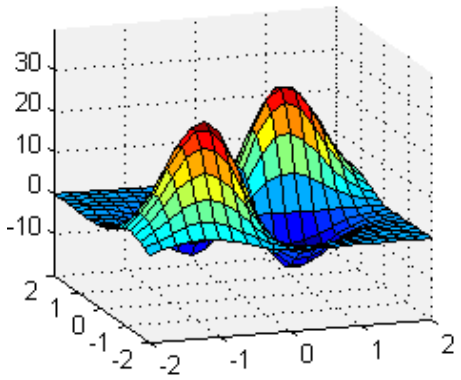
6.3.2 Graphiques 2D½

On appelle les types de graphiques présentés ici des **graphiques 2D½** ("2D et demi") car, représentant des données 3D sur un graphique à 2 axes (2D), ils sont à mi-chemin entre le 2D et le 3D.

Dans la "galerie" de présentation des fonctions graphiques de ce chapitre, nous visualiserons toujours la même fonction 2D **z=fct(x,y)** dont les matrices `X`, `Y` et `Z` sont produites par le code MATLAB/Octave ci-dessous :

```
x=-2:0.2:2;
y=x;
[X,Y]=meshgrid(x,y);
Z=100*sin(X).*sin(Y) .* exp(-X.^2 + X.*Y - Y.^2);
```

Figure ci-contre ensuite produite avec : `surf(X,Y,Z)`



MATLAB et Octave 3

Fonction et description	
Exemple	**Illustration**

{ [*C*, *h*] = } `contour({X, Y,} Z {, n | v } {, linespec } )`

Courbes de niveau :
Dessine les courbes de niveau (isolignes) interpolées sur la base de la matrice `Z`
- les vecteurs ou matrices `X` et `Y` ne servent "qu'à" graduer les axes X et Y
- le scalaire `n` permet de définir le nombre de courbes à tracer
- le vecteur `v` permet de spécifier pour quelles valeurs précises de Z il faut interpoler des courbes
- la couleur des courbes est contrôlée par les fonctions `colormap` et `caxis` (présentées plus loin)
- on peut ausssi spécifier le paramètre `linespec` pour définir l'apparence des courbes
- on peut affecter cette fonction aux paramètres de sortie *C* (matrice de contour) et *h* (handles) si l'on veut utiliser ensuite la fonction `clabel` ci-dessous

De façon interne, `contour` fait appel à la fonction MATLAB/Octave `contourc` d'interpolation de courbes (calcul de la matrice *C*).
Voir aussi la fonction `ezcontour` (easy contour) de visualisation, par courbes de niveau, d'une **fonction** à 2 variables définie sous forme d'une expression fct(x,y).

{handle = } `clabel(C, h {,'manual'} )` (***c**ontour label)*

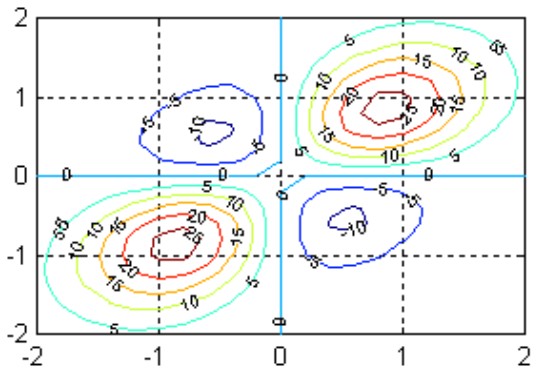
Étiquetage des courbes de niveau :
Place des labels (valeur des cotes Z) sur les courbes de niveau. Les paramètres *C* (matrice de contour) et *h* (vecteur de handles) sont récupérés lors de l'exécution préalable des fonctions de dessin `contour` ou `contourf` (ou de la fonction d'interpolation `contourc`). Sans le paramètre *h*, les labels ne sont pas placés parallèlement aux courbes. Avec le paramètre `'manual'`, on peut désigner manuellement (par un clic souris-gauche) l'emplacement de ces labels.
Octave Qt et FLTK 3.4 à 4.0 n'oriente pas encore les labels selon les courbes

Ex

```
[C,h]= contour(X,Y,Z,[-10:5:30]);  % dessin courbes
   % courbes de niveaux de -10 à 30 tous les 5
h_cl= clabel(C,h);  % dessin des labels ; on
% récupère h_cl, vect. handles pointant v/chaque label
grid('on')
```

On pourrait formater les labels des courbes de niveau avec : (faire `disp(get(h_cl(1)))` pour comprendre structure/champs)

```
set(h_cl,'fontsize',7) % chang.  taille  tous labels

% code pour arrondir à  1 décimale  le texte des labels
for k=1:length(h_cl)   % parcours de chaque label
  h_cl_s = get(h_cl(k));          % struct. k-ème label
  lab_s = h_cl_s.string;          % texte du label...
  lab_n = str2num(lab_s);         %  converti en nb
  lab_rs=sprintf('%5.1f',lab_n); %  converti chaine
```



```
  set(h_cl(k),'string',lab_rs);  %  réappliqué d/graph.
end
```

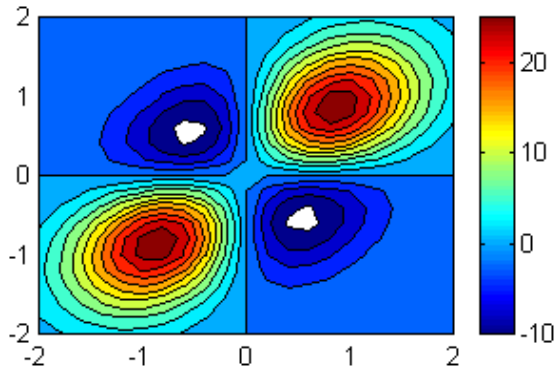
{ [*C*, *h*, *CF*] = } **`contourf({`***X, Y,***`}` *Z* **`{,` *n* `|` *v* `} )`** *(contour **f**illed)*

Courbes de niveau avec remplissage :
Le principe d'utilisation de cette fonction est le même que pour **`contour`** (voir plus haut), mais l'espace entre les courbes est ici rempli de couleurs. Celles-ci sont également contrôlées par les fonctions **`colormap`** et **`caxis`** (présentées plus loin). On pourrait bien évidemment ajouter à un tel graphique des labels (avec **`clabel`**), mais l'affichage d'une légende de type "barre de couleurs" (avec la fonction **`colorbar`** décrite plus loin) est bien plus parlant.

Voir aussi la fonction **`ezcontourf`** (easy contour filled) de visualisation, par courbes de niveau avec remplissage, d'une **fonction** à 2 variables définie sous forme d'une expression fct(x,y).

Ex

```
contourf(X,Y,Z,[-10:2.5:30])
  % courbes tous les 2.5

colormap('default') % essayez winter, summer...
colorbar            % affich. barre couleurs
  % essayez p.ex. : caxis([-40 60])
  %                 caxis([-5 20])
  %                 caxis('auto')
```

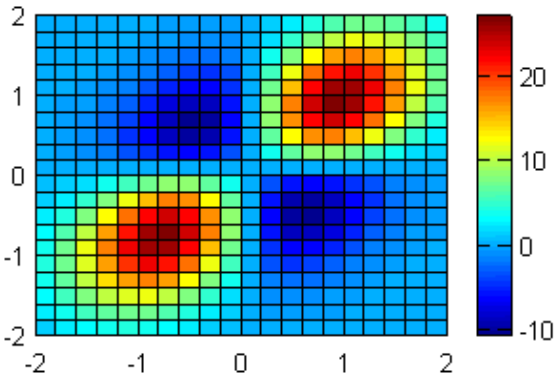


`surface({`*X, Y,***`}` *Z* **`)`** ou
`pcolor({`*X, Y,***`}` *Z* **`)`** *(**p**seudo color)*

Affichage en "facettes de couleur" ou avec lissage interpolé :
La matrice de valeurs *Z* est affichée en 2D sous forme de facettes colorées (mode par défaut, sans interpolation, correspondant à **`interp('faceted')`**).
Les couleurs indexées sont contrôlées par les fonctions **`colormap`** et **`caxis`** (décrites plus loin).
On peut en outre réaliser une interpolation de couleurs (lissage) avec la commande **`shading`** (également décrite plus loin).

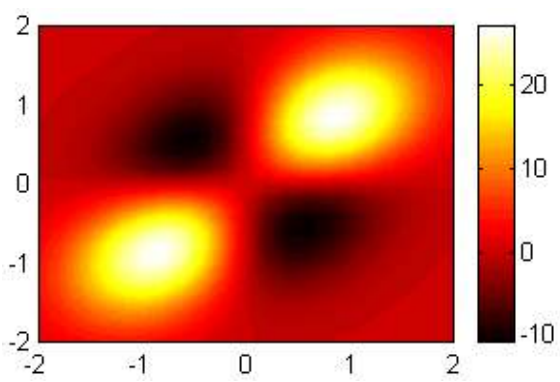
Ex 1

```
surface(X,Y,Z)  % ou:  pcolor(X,Y,Z)
colorbar

shading('flat') % ferait disparaître le
                % bord noir des facettes
```



Ex 2

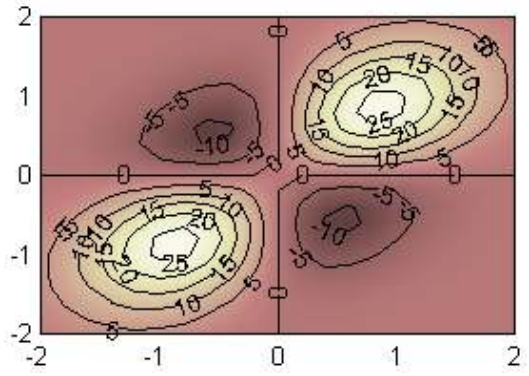
```
pcolor(X,Y,Z)    % ou:  surface(X,Y,Z)
shading('interp') % interpolation de couleur
colormap(hot)     % changement table couleurs
colorbar
```



Ex 3

Illustration de la combinaison de 2 graphiques avec `hold('on')`

```
[C,h]= contour (X,Y,Z,[-10:5:30],'k');
            % courbes noires
 clabel (C,h);
hold('on')
 pcolor (X,Y,Z)      % ou:  surface(X,Y,Z)
colormap(pink(512))
caxis([-15 30])     % atténuer tons foncés
shading('interp')
```



`quiver({X, Y,} dx ,dy {, scale } {, linespec {, 'filled' } } )`

Affichage d'un champ de vecteurs :
Dessine le champ de vecteurs défini par les matrices `dx` et `dy` (p.ex. calculées avec la fonction `gradient` décrite ci-dessous) :

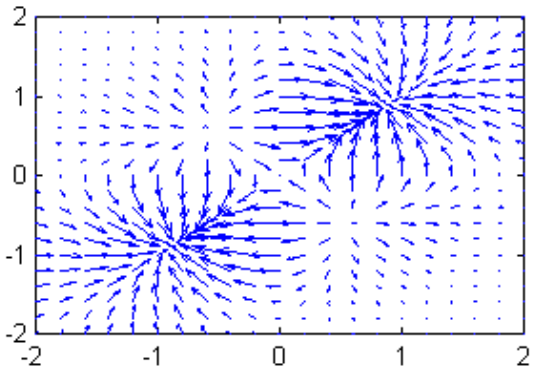
- les vecteurs ou matrices `X` et `Y` ne servent qu'à graduer les axes (si nécessaire)
- le paramètre `scale` permet de définir l'échelle des vecteurs
- avec le paramètre `linespec`, on peut encore définir un type de trait et couleur, ainsi que changer la flèche par un symbole, voire remplir les symboles avec `'filled'`

`[dx, dy] = gradient(Z {, espac_x, espac_y } )`

Calcul d'un champ de vecteurs (vitesses) :
Détermine un champ de vecteurs en calculant le gradient 2D de la matrice *Z*.
Fonction communément utilisée pour générer les vecteurs affichés par la fonction `quiver` ci-dessus.
Définie ici en 2 dimension, cette fonction est aussi utilisable sous MATLAB en N-dimension.

Ex

```
[dx,dy]= gradient(Z,0.25,0.25);
     % calcul champ vecteurs
quiver(X,Y,dx,dy,1.5)
     % affichage champ vecteurs
```



6.3.3 Graphiques 3D

Nous décrivons ci-dessous les fonctions MATLAB/Octave les plus importantes permettant de représenter en 3D des **points**, **lignes**, **barres** et **surfaces**.

Fonction et description

Exemple | **Illustration**

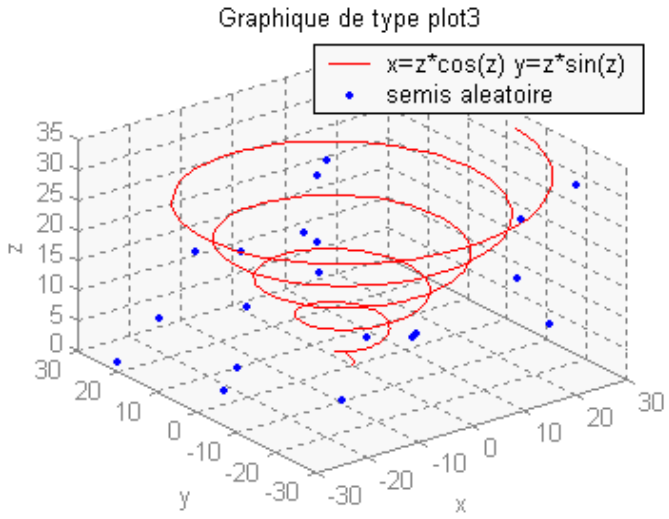
`plot3(x1,y1,z1 {,linespec} {, x2,y2,z2 {,linespec} ...} )`
`plot3(x1,y1,z1 {,'PropertyName',PropertyValue} ... )`

Graphique 3D de lignes et/ou semis de points sur axes linéaires :
Analogue à la fonction 2D `plot` (à laquelle on se réfèrera pour davantage de détails), celle-ci réalise un graphique 3D (et nécessite donc, en plus des vecteurs *x* et *y*, un vecteur *z*).
Comme pour les graphiques 2D, on peut bien évidemment aussi superposer plusieurs graphiques 3D avec `hold('on')`.

Voir en outre (plus bas dans ce support de cours) :
• pour tracer des courbes 2D/3D dans un fichier au format AutoCAD DXF : fonction `dxfwrite` du package "plot" (qui ne semble cependant plus maintenu)

Ex

```
z1=0:0.1:10*pi;
x1=z1.*cos(z1);
y1=z1.*sin(z1);

x2=60*rand(1,20)-30;  % 20 points de coord.
y2=60*rand(1,20)-30;  % -30 < X,Y < 30
z2=35*rand(1,20);     %   0 <  Z  < 35

plot3(x1,y1,z1,'r',x2,y2,z2,'o')

axis([-30 30 -30 30 0 35])
grid('on')
xlabel('x'); ylabel('y'); zlabel('z');
title('Graphique de type plot3')
legend('x=z*cos(z) y=z*sin(z)', ...
       'semis aleatoire',1)

set(gca,'xtick',[-30:10:30])
set(gca,'ytick',[-30:10:30])
set(gca,'ztick',[0:5:35])
set(gca,'Xcolor',[0.5 0.5 0.5], ...
        'Ycolor',[0.5 0.5 0.5], ...
        'Zcolor',[0.5 0.5 0.5])
```



`ezplot3('expr_x','expr_y','expr_z', [tmin tmax] {,'animate'} )` *(easy plot3)*

Dessin d'une courbe paramétrique 3D :
Dessine la courbe paramétrique définie par les expressions x=fct(t), y=fct(t), z=fct(t) spécifiées, pour les valeurs de t allant de *tmin* à *tmax*. Avec le paramètre 'animate', réalise un tracé animé de type `comet3`.

Ex Le code ci-dessous réalise la même courbe rouge que celle de l'exemple `plot3` précédent :

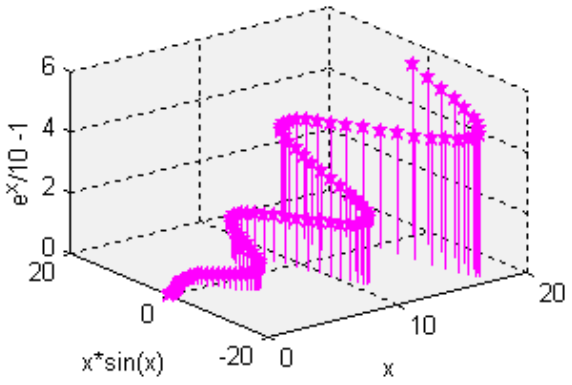
```
ezplot3('t*sin(t)','t*cos(t)','t',[0,10*pi])
```

`stem3(x,y,z {,linespec} {,'filled'} )`

Graphique 3D en bâtonnets :
Analogue à la fonction 2D `stem` (à laquelle on se réfèrera pour davantage de détails), celle-ci réalise un graphique 3D et nécessite donc, en plus des vecteurs *x* et *y*, un vecteur *z*.
Cette fonction rend mieux la 3ème dimension que les fonctions `plot3` et `scatter3` lorsqu'il s'agit de représenter un semis de points !

Ex

```
x=linspace(0,6*pi,100);
y=x.*sin(x);
z=exp(x/10)-1;

stem3(x,y,z,'mp')

xlabel('x')
ylabel('x*sin(x)')
zlabel('e^x/10 -1')
grid('on')
```



a) `mesh({X, Y,} Z {,COUL} )`
b) `meshc({X, Y,} Z {,COUL} )` *(mesh and **c**ontours)*
c) `meshz({X, Y,} Z {,COUL} )`

Dessin de surface 3D sous forme grille (wireframe) :
Dessine, sous forme de grille (wireframe, mesh), la surface définie par la matrice de hauteurs *Z*. Il s'agit d'une représentation en "fil de fer" avec traitement des lignes cachées (hidden lines). Comme vu plus haut, les vecteurs ou matrices *X* et *Y* servent à uniquement à graduer les axes ; s'ils ne sont pas spécifiés, la graduation s'effectue de 1 à size(*Z*).
a) Affichage de la surface uniquement
b) Affichage combiné de la surface et des courbes de niveau (sur un plan sous la surface). En fait `meshc` appelle `mesh`, puis fait un `hold('on')`, puis appelle `contour`. Pour mieux contrôler l'apparence des courbes de niveau, l'utilisateur peut donc exécuter lui-même cette séquence de commandes manuellement au lieu d'utiliser `meshc`.
c) Dessine, autour de la surface, des plans verticaux ("rideaux")

S'agissant du paramètre *COUL* :

- ce paramètre permet de spécifier la couleur de chaque maille (et visualiser ainsi des **données 4D** en définissant la couleur indépendemment de l'altitude)
  - si *COUL* est une matrice 2D de même dimension que *Z*, elle définit des "**couleurs indexées**", et une transformation linéaire est automatiquement appliquée (via les paramètres [*cmin cmax*] décrits plus loin) pour faire correspondre ces valeurs avec les indices de la table de couleurs courante
  - mais *COUL* peut aussi spécifier des "**vraies couleurs**" en composantes RGB ; ce sera alors une matrice 3D dont *COUL(:,:,1)* définira la composante rouge (de 0.0 à 1.0), *COUL(:,:,2)* la composante verte, et *COUL(:,:,3)* la composante bleue
- si *COUL* n'est pas spécifié, la couleur sera "proportionelle" à la hauteur Z, et le choix et l'usage de la palette sera effectué par les fonctions `colormap` et `caxis` (décrites plus loin)

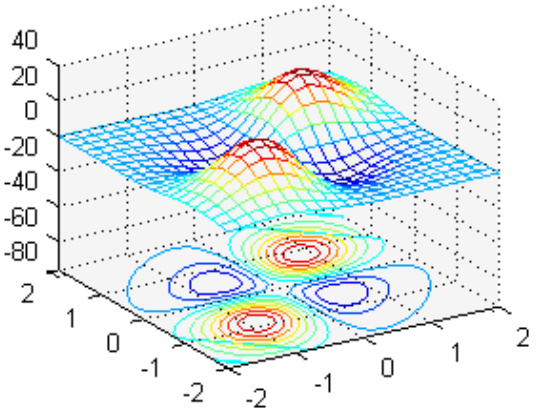
`hidden('on | off')`

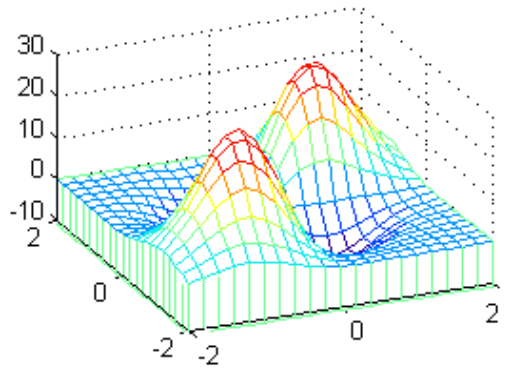
Affichage/masquage des lignes cachées :
Les fonctions `mesh`, `meshc`, `meshz`, `waterfall` effectuant un traitement des lignes cachées (hidden lines), cette commande permet de le désactiver si nécessaire.

Ex

On reprend ici, et dans les exemples suivants, la fonction utilisée dans les exemples du chapitre "Graphiques 2D de représentation de données 3D"

```
x=-2:0.2:2;  y=x;
[X,Y]=meshgrid(x,y);
Z=100*sin(X).*sin(Y).* ...
  exp(-X.^2 + X.*Y - Y.^2);

meshc(X,Y,Z)  % => graphique supérieur

meshz(X,Y,Z)  % => graphique inférieur
```



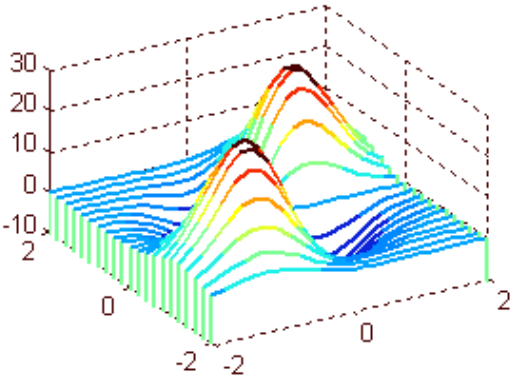


`waterfall({X, Y,} Z {,COUL} )`

Dessin de surface 3D en "chute d'eau" :
Le graphique produit est similaire à celui de **`meshz`**, sauf que les lignes dans la direction de l'axe Y ne sont pas dessinées.
La fonction **`hidden`** peut aussi être utilisée pour faire apparaître les lignes cachées.

Ex

```
h= waterfall(X,Y,Z);
  % on récupère le handle du graphique pour
  % changer ci-dessous épaisseur des lignes

set(h,'linewidth',2)
```



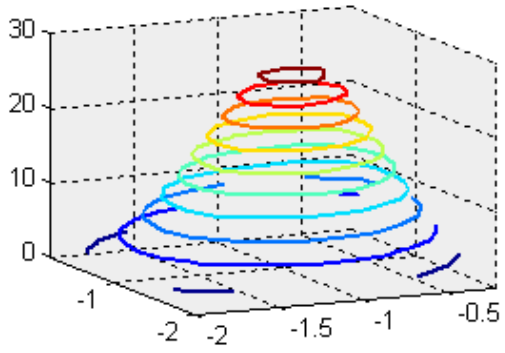
{ [*C*, *h*] = } **`contour3({X, Y,} Z {, n | v } )`**

Dessin de courbes de niveau en 3D :
Cette fonction est analogue à **`contour`** (à laquelle on se réfèrera pour davantage de détails, notamment l'usage des paramètres *n* ou *v*), sauf que les courbes ne sont pas projetées dans un plan horizontal mais dessinées en 3D dans les différents plans XY correspondant à leur hauteur Z.

Voir aussi la fonction M **`contourslice`** permettant de dessiner des courbes de niveau selon les plans XZ et YZ

Ex

```
[C,h]= contour3(x(1:10),y(1:10), ...
                Z(1:10,1:10),[0:2.5:30]);
colormap('default')
set(h,'linewidth',2)
```



`ribbon({x,} Y {,width} )`

Dessin 3D de lignes 2D en rubans :
Dessine chaque colonne de la matrice *Y* comme un ruban. Un vecteur *x* peut être spécifié pour graduer l'axe. Le scalaire *width* défini la largeur des rubans, par défaut 0.75 ; s'il vaut 1, les bandes se touchent.

Sous Octave 3.4 à 4.0 :
- *x* doit être une matrice de même dimension que *Y* (donc ne peut pas être un vecteur)
- *width* ne peut pas être spécifier si l'on omet *x*

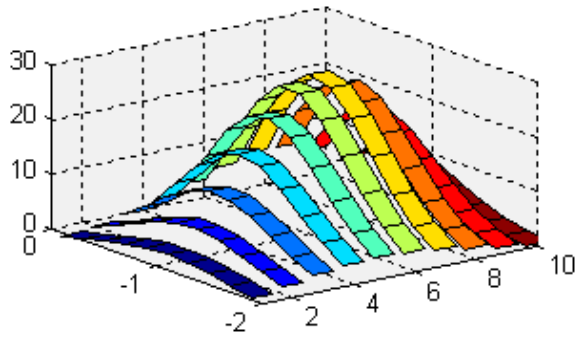
Ex

Graphique ci-contre réalisé avec MATLAB :

```
ribbon(x(1:10),Z(1:10,1:10),0.7)
axis([1 10 -2 0 0 30])
```

Sous Octave, vous pouvez essayer :

```
ribbon(Z(1:10,1:10))
```



a) **`ezmesh{c}('`***`fonction_xy`***`', [`***`xmin xmax ymin ymax`***`], `***`n`*** **`{,'circ'} )`** *(easy mesh/meshc)*
b) **`ezmesh{c}('`***`expr_x`***`,'`***`expr_y`***`,'`***`expr_z`***`', [`***`tmin tmax`***`], `***`n`*** **`{,'circ'} )`**

Dessin d'une fonction z=fct(x,y) ou (x,y,z)=fct(t) sous forme grille (wireframe) :
Dessine, sous forme de grille avec *n* mailles, la fonction spécifiée.
a) fonction définie par l'expressions fct(x,y), pour les valeurs comprises entre *xmin* et *xmax*, et *ymin* et *ymax*
b) fonction définie par les expressions x=fct(t), y=fct(t), z=fct(t), pour les valeurs de t allant de *tmin* à *tmax*

Ex Le code ci-dessous réalise la même surface que celle de l'exemple **`mesh`** précédent :

```
fct='100*sin(x)*sin(y)*exp(-x^2 + x*y - y^2)';
ezmeshc(fct, [-2 2 -2 2], 40)
```

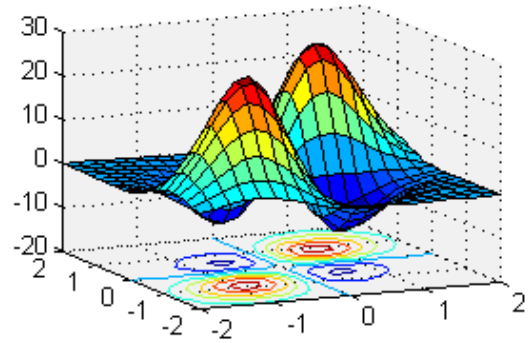
a) **`surf({`***`X,Y,`***`} `***`Z`*** **`{,`***`COUL`***`} {,'`***`PropertyName`***`', `***`PropertyValue...`***`} )`**
b) **`surfc({`***`X,Y,`***`} `***`Z`*** **`{,`***`COUL`***`} {,'`***`PropertyName`***`', `***`PropertyValue...`***`} )`** *(surface and **c**ontours)*

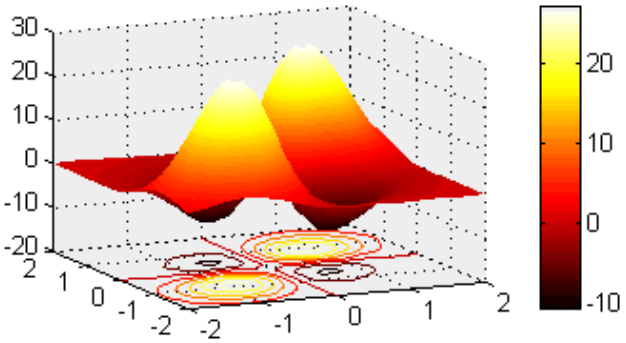
Dessin de surface 3D colorée (shaded) :
Fonctions analogues à **`mesh`** / **`meshc`** (à laquelle on se réfèrera pour davantage de détails), sauf que les facettes sont ici colorées. On peut en outre paramétrer finement l'affichage en modifiant les diverses *propriétés*. De plus :
• les dégradés de couleurs sont fonction de *Z* ou de *COUL* et dépendent de la palette de couleurs (fonctions **`colormap`** et **`caxis`** décrites plus loin), à moins que *COUL* soit une matrice 3D définissant des "vraies couleurs" ;
• le mode de coloriage/remplissage (shading) par défaut est **`'faceted'`** (i.e. pas d'interpolation de couleurs) :
- avec la commande **`shading('flat')`**, on peut faire disparaître le trait noir bordant les facettes
- avec **`shading('interp')`** on peut faire un lissage de couleur par interpolation.

Voir aussi les fonctions **`ezsurf`** et **`ezsurfc`** qui sont le pendant de **`ezmesh`** et **`ezmeshc`**.

Ex

```
surfc(X,Y,Z)  % shading 'faceted' par défaut

axis([-2 2 -2 2 -20 30])
set(gca,'xtick',[-2:1:2])
set(gca,'ytick',[-2:1:2])
set(gca,'ztick',[-20:10:30])
% => graphique supérieur

shading('interp') % voyez aussi shading('flat')
colormap(hot)
colorbar
% => graphique inférieur
```



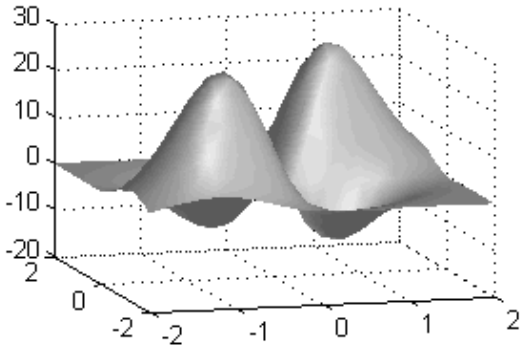


`surfl({`*X*`,`*Y*`,} `*Z*` {,`*s*` {,`*k*` } } )` *(surface lighted)*

Dessin de surface 3D avec coloriage dépendant de l'éclairage :
Fonction analogue à **`surf`**, sauf que le coloriage des facettes dépend ici d'une **source de lumière** et non plus de la hauteur *Z* !
• Le paramètre *`s`* définit la **direction** de la source de lumière, dans le sens surface->lumière, sous forme d'un vecteur [azimut elevation] (en degrés), ou coordonnées [sx sy sz]. Le défaut est 45 degrés depuis la direction de vue courante.
• Le paramètre *`k`* spécifie la **réflectance** sous forme d'un vecteur à 4 éléments définissant les contributions relatives de [lumière ambiante, réflexion diffuse, réflexion spéculaire, specular shine coefficient]. Le défaut est [0.55, 0.6, 0.4, 10]
• On appliquera la fonction **`shading('interp')`** (décrite plus loin) pour obtenir un effet de lissage de surface (interpolation de teinte).
• On choisira la table de couleurs adéquate avec la fonction **`colormap`** (voir plus loin). Pour de bonnes transitions de couleurs, il est important d'utiliser une table qui offre une variation d'intensité linéaire, telle que **`gray`**, **`copper`**, **`bone`**, **`pink`**.

Ex

```
surfl(X,Y,Z);

axis([-2 2 -2 2 -20 30]);
shading('interp');
colormap(gray);
```



`trimesh` (grille, wireframe)
`trisurf` (surface colorée, shaded)

Dessin de surface 3D basé sur une triangulation :
Pour ce type de graphique, permettant de tracer une surface passant par un **semis de points irrégulier**, on se réfèrera à l'exemple du chapitre "La fonction "griddata" d'interpolation de grille dans un semis irrégulier"

`scatter3(`*x, y, z* `{,`*size* `{,`*color* `} } {,`*symbol*`} {,'filled'} )`

Visualisation d'un semis de points 3D par des symboles :
Cette fonction s'utilise de façon analogue à la fonction 2D **`scatter`** (à laquelle on se réfèrera pour davantage de détails).
Voir aussi **`plot3`**, et surtout **`stem3`** qui rend mieux la 3ème dimension.

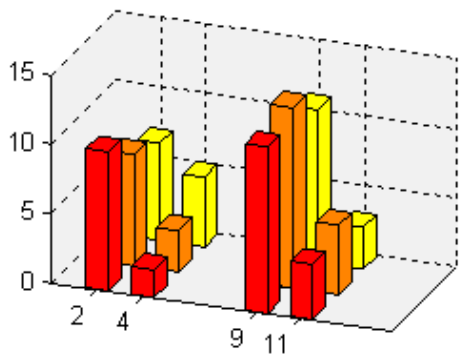
a) **`bar3( {`*x*`,} `*mat* `{,`*larg*`} {,'`*style*`'} )`**
b) **`bar3h( {`*z*`,} `*mat* `{,`*larg*`} {,'`*style*`'} )`**

Graphique de barres 3D :
Représente les valeurs de la matrice *mat* sous forme de barres 3D, verticalement avec la forme **a)**, horizontalement avec la forme **b)**.
S'il est fourni, le vecteur *x* ou *z* est utilisé pour graduer l'axe au pied des barres et espacer celles-ci.
Le scalaire *larg* spécifie le rapport "épaisseur des barres / espacement entre barres en profondeur" dans le cadre d'un groupe ; la valeur par défaut est 0.8 ; si celle-ci atteint 1, les barres se touchent.
Le paramètre *style* peut prendre les valeurs **`'detached'`** (défaut), **`'grouped'`**, ou **`'stacked'`**

Ces 2 fonctions, qui existaient sous Octave 3.0.1/JHandles, ont disparu sous Octave 3.2 à 4.0 !?

Ex (graphique ci-contre réalisé avec MATLAB)

```
x=[2 4 9 11];
mat=[10 8 7 ; 2 3 5 ; 12 13 11 ; 4 5 3];
 bar3(x,mat,0.5,'detached')
colormap(autumn)
```



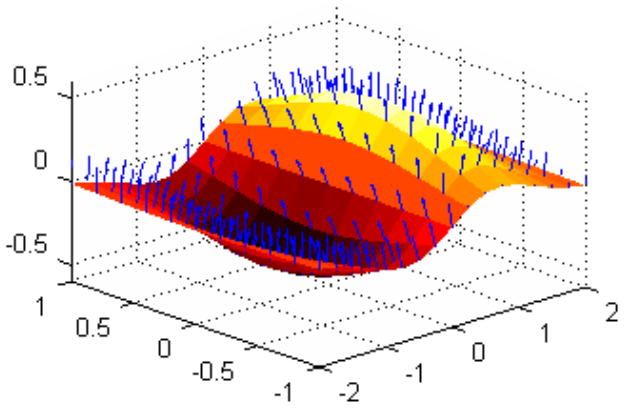
quiver3({*X*, *Y*,} *Z*, *dx*, *dy*, *dz* {, *scale* } {, *linespec* {, 'filled' } })

Dessin d'un champ de vecteurs 3D :
Extension à la 3e dimension de la fonction **quiver** vue plus haut. Dessine, sur la surface *Z*, le champ de vecteurs défini par les matrices *dx*, *dy*, *dz*. Voyez l'aide en-ligne pour davantage de détails.

Voyez aussi la fonction M **coneplot** de tracé de vecteurs dans l'espace par des cônes.

Ex

```
[X,Y]=meshgrid(-2:0.25:2, -1:0.2:1);
Z = X .* exp(-X.^2 - Y.^2);
[U,V,W]= surfnorm (X,Y,Z);
       % normales à la surface Z=fct(X,Y)
surf(X,Y,Z)
hold('on')
 quiver3(X,Y,Z, U,V,W, 0.5, 'b')

colormap(hot)
shading('flat')
axis([-2 2 -1 1 -.6 .6])
```



Et citons encore quelques **autres fonctions** graphiques 3D qui pourront vous être utiles, non décrites dans ce support de cours :

- **sphere** : dessin de **sphère**
- **ellipsoid** : dessin d'**ellipsoïde**
- **cylinder** : dessin de **cylindre** et surface de révolution
- M **fill3** : dessin de polygones 3D (flat-shaded ou Gouraud-shaded) (extension à la 3ème dimension de la fonction **fill**)
- **patch** : fonctions de bas niveau pour la création d'objets graphiques surfaciques

6.3.4 Graphiques 3D volumétriques (représentation de données 4D)

Des "**données 4D**" peuvent être vues comme des données 3D auxquelles sont associées un 4e paramètre qui est fonction de la position (x,y,z), défini par exemple par une fonction **v = fct(x,y,z)**.

Pour **visualiser des données 4D**, MATLAB/Octave propose différents types de graphiques 3D dits "**volumétriques**", le plus couramment utilisé étant `slice` (décrit ci-dessous) où le **4ème paramètre** est **représenté par une couleur** !

De façon analogue aux graphiques 3D (pour lesquels il s'agissait d'élaborer préalablement une matrice 2D définissant la surface Z à grapher), ce qu'il faut ici fournir aux fonctions de graphiques volumétriques c'est une **matrice tri-dimensionnelle** définissant un **cube de valeurs V**. Si ces données 4D résultent d'une fonction v = fct(x,y,z), on déterminera cette matrice 3D V par échantillonnement de la fonction en s'aidant de tableaux auxiliaires **Xm**,**Ym** et **Zm** (ici également tri-dimensionnels) préalablement déterminées avec la fonction `meshgrid`.

Fonction et description

Exemple | **Illustration**

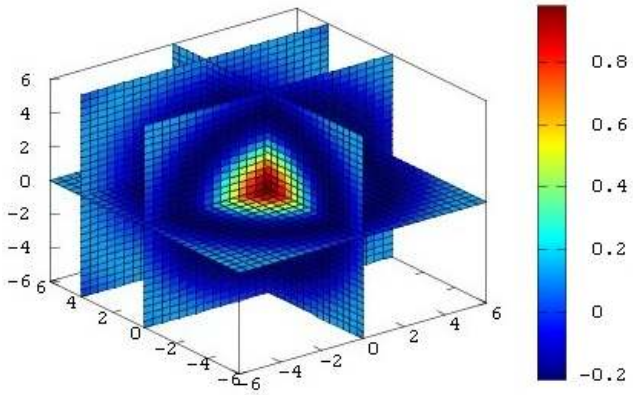
a) `slice({X,Y,Z,} V, sx,sy,sz)`
b) `slice({X,Y,Z,} V, xi,yi,zi {,méthode})`

Affichage 3D de données volumétriques sous forme de "tranche(s)" :
a) Les données volumétriques `V` (tableau tri-dimensionnel) sont représentées selon des **plans orthogonaux** aux axes du graphique (horizontaux ou verticaux) et placés aux cotes définies par les scalaires ou vecteurs `sx`, `sy` et `sz`. Si l'on ne souhaite pas de tranche selon l'une ou l'autre direction, on mettra `[]` en lieu et place du paramètre `si` correspondant à cette direction.
S'agissant des paramètres `X`, `Y` et `Z`, qui ne servent qu'à **graduer** les axes du graphique, on fournira soit les tableaux 3D auxiliaires Xm,Ym,Zm (préalablement déterminés avec `meshgrid` pour calculer `V`), soit des vecteurs (définissant les valeurs d'échantillonage en x/y/z, et de longueur correspondant aux dimensions de V). Si ces paramètres sont omis, la graduation de fera de 1 à n.

b) Sous cette forme, la "tranche" de visualisation peut être une **surface quelconque** (donc pas obligatoirement plane !) définie par une grille régulière `xi` / `yi` associée à des altitudes `zi`. La fonction `slice` se charge elle-même d'interpoler (avec `interp3`) les valeurs de `V` sur cette surface en utilisant l'une des `méthode` suivante : `'linear'` (défaut), `'nearest'` ou `'cubic'`. Cette surface/grille d'interpolation/visualisation sera donc définie par 3 matrices 2D `xi`, `yi` et `zi`

Ex 1

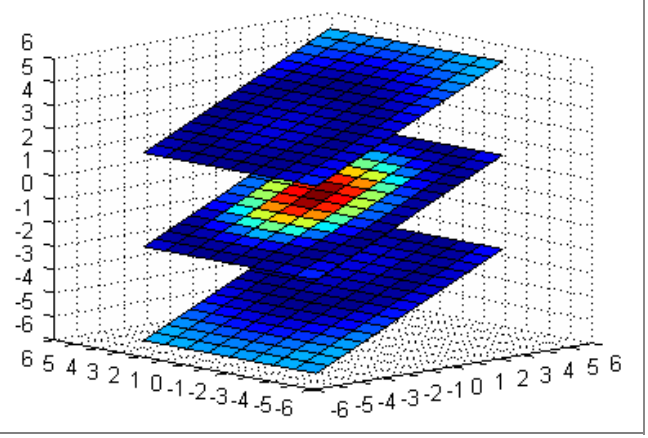
```
min=-6; max=6; n_grid=30;
v_xyz = linspace(min,max,n_grid);
[Xm,Ym,Zm] = meshgrid(v_xyz, v_xyz, v_xyz);
   % ou simplement: [Xm,Ym,Zm]=meshgrid(v_xyz);
V = sin( sqrt(Xm.^2 + Ym.^2 + Zm.^2) ) ./ ...
    sqrt(Xm.^2 + Ym.^2 + Zm.^2);
 slice(Xm,Ym,Zm, V,[],[],0)
   % ci-dessus: tranche horiz.en z=0
hold('on')
 slice(v_xyz,v_xyz,v_xyz,V,0,[0,4],[])
   % tranches verticales en x=0 et en y=[0,4]
   % pour les 3 permiers paramètres, on peut
   %  utiliser Xm,Ym,Zm ou v_xyz,v_xyz,v_xyz
axis([min max min max min max])
colorbar
```



Ex 2

Poursuite avec les données `V` de l'exemple précédent :

```
clf
[xi,yi]=meshgrid([-4:0.5:4],[-4:1:4]);
   % ci-dessus définition grille X/Y
zi = 0.5 * xi; % plan incliné sur cette grille
 slice (v_xyz,v_xyz,v_xyz, V, xi,yi,zi)
zi = zi + 4;  % 2e plan au dessus du précéd.
hold('on')
 slice (v_xyz,v_xyz,v_xyz, V, xi,yi,zi)
zi = zi - 8;  % 3e plan au dessous du précéd.
 slice (v_xyz,v_xyz,v_xyz, V, xi,yi,zi)
grid('on')
axis([min max min max min max])
```

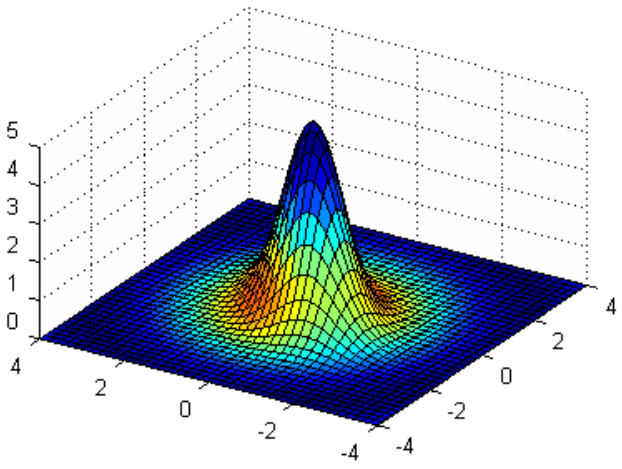


```
set(gca,'Xtick',[min:max],'Ytick',[min:max], ...
        'Ztick',[min:max])
```

Ex 3

Poursuite avec les données `V` de l'exemple précédent :

```
clf
 [xi,yi] =meshgrid([-4:0.2:4]);
 zi =5*exp(-xi.^2 + xi.*yi - yi.^2);
 slice (v_xyz,v_xyz,v_xyz, V, xi,yi,zi)
```



On peut encore mentionner les fonctions de représentation de données 4D suivantes, en renvoyant l'utilisateur à l'aide en-ligne et aux manuels pour davantage de détails et des exemples :

- `streamline`, `stream2`, `stream3` : dessin de lignes de flux en 2D et 3D
- `contourslice` : dessin de courbes de niveau dans différents plans (parallèles à XY, XZ, YZ...)
- `isocolors`, `isosurface`, `isonormals`, `isocaps` : calcul et affichage d'isosurfaces...
- `subvolume`, `reducevolume` : extrait un sous-ensemble d'un jeu de données volumétriques

6.3.5 Paramétres de visualisation de graphiques 3D

Nous présentons brièvement, dans ce chapitre, les fonctions permettant de **modifier l'aspect** des graphiques 3D (et de certains graphiques 2D) : orientation de la vue, couleurs, lissage, éclairage, propriétés de réflexion... La technique des "Handle Graphics" permet d'aller encore beaucoup plus loin en modifiant toutes les propriétés des objets graphiques.

Vraies couleurs, tables de couleurs (colormaps), et couleurs indexées

On a vu plus haut que certaines couleurs de base peuvent être choisies via la spécification `linespec` utilisée par plusieurs fonctions graphiques (p.ex. `'r'` pour rouge, `'b'` pour bleu...), mais le choix de couleurs est ainsi très limité (seulement 8 couleurs possibles).

1) Couleurs vraies

En informatique, chaque couleur est habituellement définie de façon additive par ses composantes RGB (red, green, blue). MATLAB et Octave ont pris le parti de spécifier les **intensités** de chacune de ces 3 couleurs de base par un nombre réel compris **entre 0.0 et 1.0**. Il est ainsi possible de définir des couleurs absolues, appelées "**couleurs vraies**" (**true colors**), par un **triplet RGB** `[red green blue]` sous la forme d'un vecteur de 3 nombres compris entre 0.0 et 1.0.

> **Ex**: `[1 0 0]` définit le rouge-pur, `[1 1 0]` le jaune, `[0 0 0]` le noir, `[1 1 1]` le blanc, `[0.5 0.5 0.5]` un gris intermédiaire entre le blanc et le noir, etc...

2) Table de couleurs

À chaque figure MATLAB/Octave est associée une "**table de couleurs**" (palette de couleurs, **colormap**). Il s'agit d'une matrice, que nous désignerons par `cmap`, dont chaque **ligne** définit **une couleur** par un triplet RGB. Le nombre de lignes de cette matrice est donc égal au nombre de couleurs définies, et le nombre de colonnes est toujours 3 (valeurs, comprises entre 0.0 et 1.0, des 3 composantes RGB de la couleur définie).

3) Couleurs indexées

Bon nombre de fonctions graphiques 2D, 2D½, 3D (`contour`, `surface` / `pcolor`, `mesh` et `surf` pour ne citer que les plus utilisées...) travaillent en mode "**couleurs indexées**" (pseudo-couleurs) : chaque facette d'une surface, par exemple, ne se voit pas attribuer une "vraie couleur" (triplet RGB) mais pointe, par un index, vers une couleur de la table de couleurs. Cette technique présente l'avantage de pouvoir changer l'ensemble des couleurs d'un graphique sans modifier le graphique lui-même mais en modifiant simplement de table de couleurs (palette). En fonction de la taille de la table choisie, on peut aussi augmenter ou diminuer le **nombre de nuances** de couleurs du graphique.

Lorsque MATLAB/Octave crée une nouvelle figure, il met en place une **table de couleur par défaut** en utilisant la fonction `jet(64)`. La fonction `jet(n)` crée une table de *n* couleurs en dégradés allant du bleu foncé au brun en passant par le cyan, vert, jaune, orange et rouge. La table de couleur par défaut d'une nouvelle figure a par conséquent 64 couleurs.

> **Ex**: `jet_cmap=jet(64);` calcule et stocke la table de couleurs par défaut sur la matrice `jet_cmap` ; `jet_cmap(57,:)` retourne par exemple la 57ème couleur de cette table qui, comme vous pouvez le vérifier, est égale (MATLAB) ou très proche (Octave) de [1 0 0], donc le rouge pur.

L'utilisateur peut créer lui-même ses propres tables de couleur et les appliquer à un graphique avec la fonction `colormap` présentée plus bas. Il existe cependant des tables de couleur prédéfinies ainsi que des fonctions de création de tables de couleurs.

> **Ex**: `ma_cmap=[0 0 0;2 2 2;4 4 4;6 6 6;8 8 8;10 10 10]/10;` génère une table de couleurs composées de 6 niveaux de gris allant du noir-pur au blanc-pur ; on peut ensuite l'appliquer à un graphique existant avec `colormap(ma_cmap)`

4) Scaled mapping et direct mapping

La mise en correspondance (mapping) des 'données de couleur' d'un graphique (p.ex. les valeurs de la matrice *Z* d'un graphique `surf(..., Z)`, ou les valeurs de la matrice 2D *COUL* d'un graphique `surf(..., Z, COUL)`) avec les indices de sa table de couleurs peut s'effectuer de façon directe (direct mapping) ou par une mise à l'échelle (scaled mapping).

- **Scaled mapping** : dans ce mode (qui est le défaut), MATLAB fait usage d'un vecteur à 2 éléments `[cmin cmax]` dans lequel *cmin* spécifie la valeur de 'donnée de couleur' du graphique qui doit être mise en correspondance avec la 1ère couleur de la table de couleur ; *cmax* spécifiant respectivement la valeur de 'donnée de couleur' devant être mise en correspondance avec la dernière couleur de la table. Les valeurs de 'donnée de couleur' se trouvant dans cet intervalle sont automatiquement mises en correspondance avec les différentes indices de la table par une **transformation linéaire**. MATLAB définit automatiquement les valeurs *cmin* et *cmax* de façon qu'elles correspondent à la plage de 'données de couleur' de tous les éléments du graphique, mais on peut les modifier avec la commande `caxis([cmin cmax])` présentée plus bas.
- **Direct mapping** : pour activer ce mode, il faut désactiver le scaling en mettant la propriété `'CDataMapping'` à la valeur `'direct'` (en passage de paramètres lors de la création du graphique, ou après coup par manipulation de handle). Dans ce mode, rarement utilisé, les 'données de couleur' sont mise en correspondance directe (sans scaling) avec les indexes de la

matrice de couleur. Une valeur de 1 (ou inférieure à 1) pointera sur la 1ère couleur de la table, une valeur de 2 sur la seconde couleur, etc... Si la table de couleur comporte n couleurs, la valeur n (ou toute valeur supérieure à n) pointera sur la dernière couleur.

Nous présentons ci-dessous les **principales fonctions** en relation avec la manipulation de tables de couleurs.

<table>
<tr><th colspan="2">Fonction et description</th></tr>
<tr><th>Exemple</th><th>Illustration</th></tr>
<tr><td colspan="2">a) colormap(cmap)
b) colormap(named_cmap)
c) cmap = colormap
Appliquer une table de couleurs, ou récupérer la table courante d'une figure :
a) Applique la table de couleur cmap (matrice nx3 de n triplets RGB) à la figure active.
b) Applique une table de couleur nommée à la figure active. On dispose, sous MATLAB/Octave, des 18 tables nommées suivantes (voir leur description plus bas) : autumn , bone , colorcube , cool , copper , flag , gray , hot , hsv , jet , lines , pink , prism , spring , summer , vga , white , winter
c) Récupère, sur la matrice cmap, la table de couleur courante de la figure active

Remarque IMPORTANTE : les objets qui sont basés sur des "vraies couleurs" ou des codes de couleur linespec ne seron pas affectés par un changement de table de couleurs !

colormap('list')
Affiche la liste des fonctions named_cmap de tables de couleurs prédéfinies (autumn , bone , cool ...)</td></tr>
<tr><td colspan="2">a) brighten(beta)
b) cmap = brighten(beta)
Eclaircir ou assombrir une table de couleurs :
Le paramètre beta est un scalaire qui aura pour valeur :
0.0 à 1.0 si l'on veut augmenter la luminosoté de la table de couleur (brighter)
0.0 à -1.0 si l'on veut l'assombrir (darker).
a) Modifie la table de couleur courante de la figure active (avec répecussion immédiate au niveau de l'affichage)
b) Copie la table de couleur courante de la figure active sur cmap, et modifie cette copie sans impact sur la table de couleur courante et sur l'affichage

Remarque IMPORTANTE : les objets qui sont basés sur des "vraies couleurs" ou des codes de couleur linespec ne seron pas affectés par cette fonction !</td></tr>
<tr><td colspan="2">a) cmap = named_cmap { (n) }
b) colormap(named_cmap { (n) })
Calcule d'une table de couleur, ou application d'une table calculée :
MATLAB/Octave offre 17 fonctions named_cmap de calcul de tables de couleurs décrites ci-dessous avec une illustration des dégradés de couleur correspondant (réalisés avec la fonction colorbar).
a) Retourne une table de couleur avec n entrées/couleurs sur la variable cmap (table que l'on peut ensuite appliquer à la figure active avec colormap(cmap) . Si n n'est pas spécifié, la table aura la même taille que la table courante, par défaut 64 couleurs.
b) Calcule et applique directement à la figure active une table de couleur.</td></tr>
<tr><td>Fonctions de création de tables de couleurs :

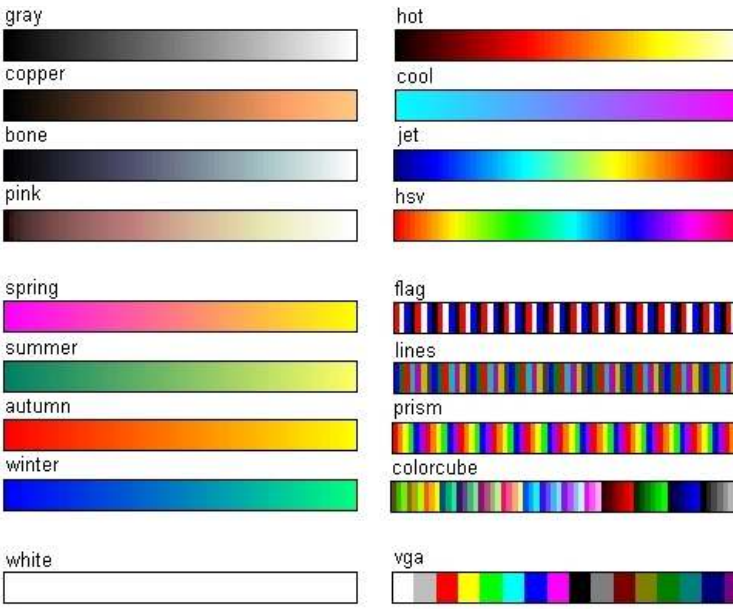
• gray(n) : linear gray-scale
• copper(n) : linear copper-tone
• bone(n) : gray-scale with tinge of blue
• pink(n) : pastel shades of pink

• spring(n) : shades of magenta and yellow
• summer(n) : shades of green and yellow
• autumn(n) : shades of red and yellow
• winter(n) : shades of blue and green

• white(n) : all white

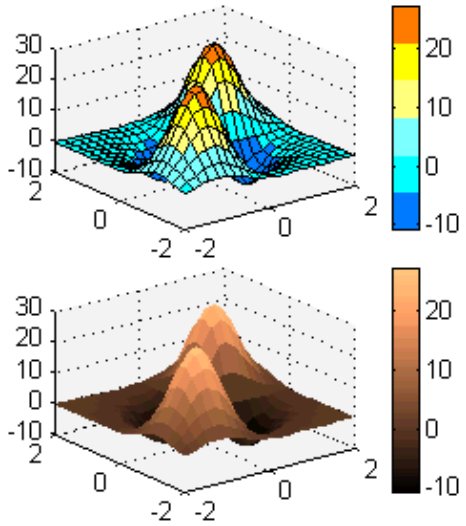
• hot(n) : black-red-yellow-white
• cool(n) : shades of cyan and magenta
• jet(n) : variant of HSV</td><td>Illustration des palettes générées par ces fonctions :</td></tr>
</table>

- **`hsv(n)`** : hue-saturation-value
- **`flag(n)`** : alternating red, white, blue, and black
- **`lines(n)`** : color map with the line colors
- **`prism(n)`** : prism
- **`colorcube(n)`** : enhanced color-cube
- **`vga`** : Windows colormap for 16 colors (fonction retournant palette de 16 couleurs, donc pas de paramètre *n*)



Ex

```
surf(X,Y,Z)  % shading 'faceted' par défaut
axis([-2 2 -2 2 -10 30])
 colormap(jet(6))
colorbar  % => premier graphique ci-contre

 colormap(copper(64))
 shading('flat')
colorbar  % => second graphique ci-contre
```

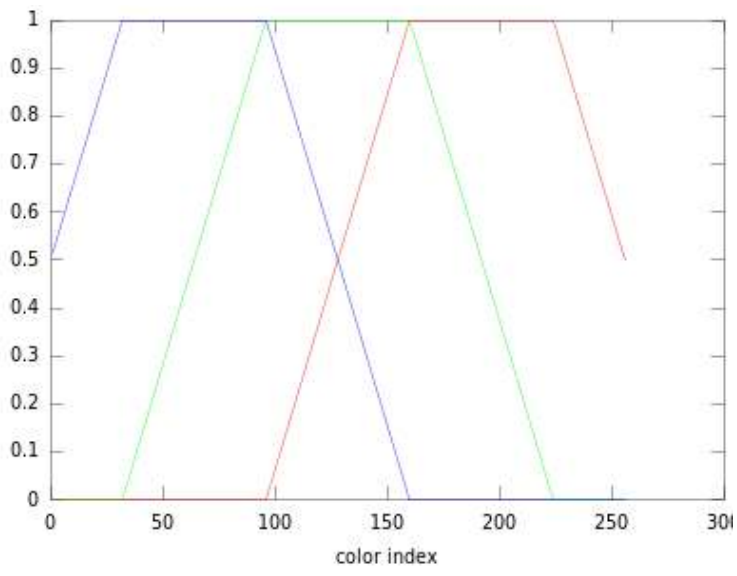


`rgbplot(cmap)`

Graphe les 3 **courbes R/G/B** de la table de couleurs *cmap* spécifiée, avec l'index de couleur en abscisse (i.e. numéro de ligne de `cmap`) et l'intensité en ordonnée (de 0 à 1)

Ex

```
rgbplot (jet(256))
```



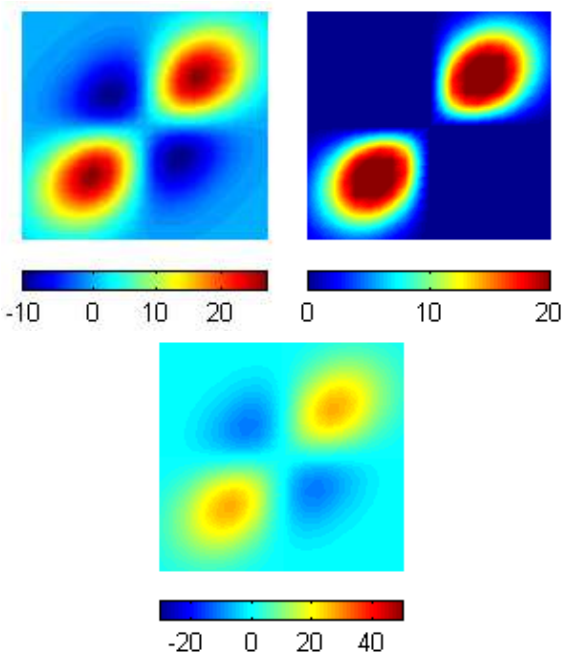
a) `caxis([cmin cmax])` *(**c**olor axis)*
b) `caxis('auto')`
c) [*cmin cmax*] = `caxis`

Changement de l'échelle des couleurs, ou récupérer les valeurs de l'échelle courante :
Agit sur la façon de mettre en correspondance les 'données de couleur' de la figure courante avec les indices de la table de couleurs (voir explications plus haut). Se répercute immédiatement au niveau de l'affichage.
a) Change l'échelle des couleurs en fonction des valeurs *cmin* et *cmax* spécifiées
b) Rétablit un scaling automatique (basé sur les valeurs min et max des données de couleur de la figure)
c) Récupère les valeurs d'échelle *cmin* et *cmax* courantes

Ex

```
pcolor(X,Y,Z)
colormap(jet(64))  % colormap par défaut
shading('interp')
axis('off')
 caxis('auto')     % défaut
colorbar('SouthOutside')   % => premier graphique

 caxis([0 20])
colorbar('SouthOutside')   % => second graphique

 caxis([-30 50])
colorbar('SouthOutside')   % => troisième graphique
```

Examinez bien l'échelle des barres de couleur



`colorbar({'East | EastOutside | West | WestOutside | North | NorthOutside | South | SouthOutside | off'})`

Affichage d'une barre graduée représentant la table de couleurs courante :
Cette barre de couleurs sera placée par défaut (si la fonction est appelée sans paramètre) verticalement à droite du graphique (ce qui correspond à `'EastOutside'`). Selon que l'on spécifie la direction sans/avec `Outside`, la barre sera dessinée dans/en dehors de la plot box.

Autres paramètres de visualisation

Fonction et description	
Exemple	**Illustration**

a) `view(az, el)` ou `view([az el])`
b) `view([xo yo zo])`
c) `view(2 | 3)`
d) `view(T)`
e) `[az,el]=view` ou `T=view`

Orientation de la vue 3D :
L'orientation par défaut de la vue dans une nouvelle figure 3D est : azimut=-37.5 et élévation=30 degrés sous MATLAB, et azimut=30 et élévation=30 degrés sous Octave/Gnuplot. Cette fonction changer cette orientation ou de relever les paramètres courant. Notez bien qu'elle agit sur la vue et non pas sur l'objet (ce que fait quant à elle la fonction `rotate`) !

a) Sous cette forme (voir **figure** ci-dessous), la seule actuellement utilisable sous Octave/Gnuplot, on spécifie l'orientation de la vue 3D par l'**azimut** *az* (compté dans le plan XY depuis l'axe Y négatif, dans le sens inverse des aiguilles) et l'**élévation** verticale *el* (comptée verticalement depuis l'horizon XY, positivement vers le haut et négativement vers le bas), tous deux en **degrés**.
b) L'orientation de la vue 3D est ici spécifiée par un vecteur de coordonnées [*xo yo zo*] pointant vers l'observateur
c) `view(2)` signifie vue 2D, c'est à dire vue de dessus (selon l'axe -Z), donc équivalent à `view(0, 90)`
`view(3)` signifie vue 3D par défaut, donc équivalent à `view(-37.5, 30)`
d) Définit l'orientation à partir de la matrice 4x4 de transformation *T* calculée par la fonction `viewmtx`
e) Récupère les paramètres relatifs à l'orientation courante de la vue

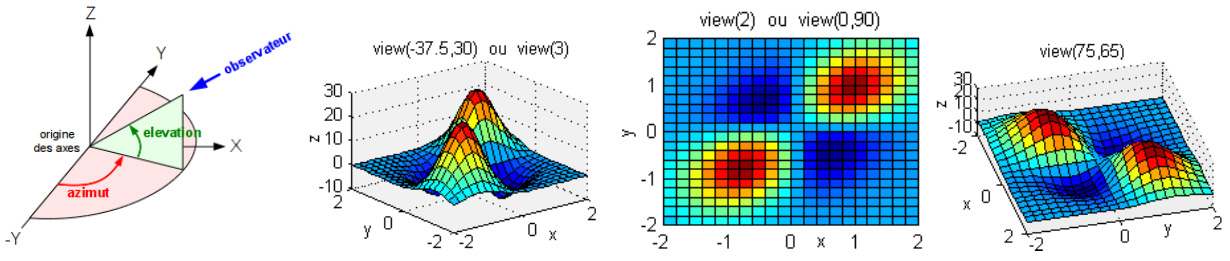
Ex: `view(0,0)` réalise une projection selon le plan XZ, et `view(90,0)` réalise une projection selon le plan YZ

`T = viewmtx(az, el {,phi {,[xc yc zc] } } )` (*view **matrix***)

Matrice de transformation perspective :
Sans changer l'orientation courante de la vue, calcule la matrice 4x4 de transformation perspective *T* sur la base de : l'azimut *az*, l'élévation *el*, l'angle *phi* de l'objectif (0=projection orthographique, 10 degrés=téléobjectif, 25 degrés=objectif normal, 60 degrée=grand angulaire...), et les coordonnées [*xc yc zc*] normalisées (valeurs 0 à 1) de la cible. On peut ensuite appliquer cette matrice *T* à la vue avec `view(T)` .

Sous MATLAB, on peut en outre faire usage de nombreuses autres fonctions de gestion de la "caméra" projetant la vue 3D dans l'espace écran/papier 2D :

- `cameramenu` : ajoute, à la fenêtre graphique, un menu "Camera" doté de nombreuses possibilités interactives : Mouse Mode et Mouse Constraint (effet de la souris), Scene Light (éclairer la scène), Scene Lighting (none, Flat, Gouraud, Phong), Scene Shading (Faceted, Flat, Interp), Scene Clipping, Render Options (Painter, Zbuffer, OpenGL), Remove Menu. Il est alors aussi possible de "lancer la scène en rotation" dans n'importe quelle direction.
- `campos` , `camtarget` , `camva` , `camup` , `camproj` , `camorbit` , `campan` , `camdolly` , `camroll` , `camlookat`



`shading('faceted | flat | interp')`

Méthode de rendu des couleurs :
Par défaut, les fonctions d'affichage de surfaces basées sur les primitives `surface` et `patch` (les fonctions `pcolor` et `surf` notamment) réalisent un rendu de couleurs non interpolé de type `'faceted'` (coloriage uniforme de chaque facette). Il existe en outre :
• le mode `'flat'` , qui est identique à `'faceted'` sauf que le trait de bordure noir des facettes n'est pas affiché
• le mode `'interp'` , qui réalise un lissage de couleurs par interpolation de **Gouraud**

Ex: voir plus haut les exemples `pcolor` , `surfc` , `surfl` et `caxis`

handle = `light( { 'PropertyName', PropertyValue, ... } )`

Activation d'un éclairage :
Active un éclairage de la scène en définissant ses éventuelles propriétés (Position, Color, Style...). On désactive cet éclairage avec `lighting('none')`
Voir aussi la fonction `camlight`.

`lightangle(handle, az, el)`

Direction de l'éclairement :
Définition de l'azimut *az* et de l'élévation *el* de cet éclairage (selon même syntaxe que `view`).
Attention : si on ne passe pas l'argument *handle*, chaque fois que l'on passe cette commande une nouvelle source de lumière est créée !

`lighting('none | flat | gouraud | phong')`

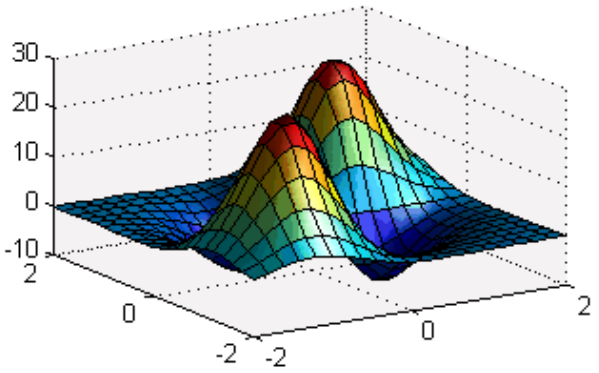
Méthode de rendu relative à l'éclairage :
Choix de l'algorithme de lissage des couleurs résultant de l'éclairage, allant du plus simple (`flat`) au plus élaboré (`gouraud`).

Les propriétés de **réflectance** des surfaces par rapport à la lumière peuvent être définie par la fonction `material('shiny' | 'dull' | 'metal' | [ka kd ks n sc]')`

Ex (graphique ci-contre réalisé avec MATLAB)

Dans ce exemple interactif, on fait tourner une source lumineuse rasante (10 degrés d'élévation) autour de la scène.

```
surf(X,Y,Z)
axis([-2 2 -2 2 -10 30])

hlight= light;                % activ. éclairage
 lighting('gouraud')           % type de rendu
for az=0:5:360
   lightangle(hlight,az,10)   % dir. éclairage
  pause(0.05)
end
```



6.4 Affichage et traitement d'images

MATLAB est également capable, ainsi qu'Octave depuis la version 3 (package octave-forge "image"), de **lire, écrire, afficher, traiter des images** dans les différents formats habituels (JPEG, PNG, TIFF, GIF, BMP...).

La présentation de ce domaine dépassant le cadre de ce cours, nous nous contentons d'énumérer ici les fonctions les plus importantes :

- attributs d'une image : `infos = imfinfo(file_name|URL)` , `imginfo` , `readexif` , `tiff_tag_read` ...
- lecture et écriture de fichiers-image : `[img,colormap,alpha]=imread(file_name)` , `imwrite(img, map, file_name, format...)` , `imformats` ...
- affichage d'image : `image(img)` , `imshow` , `imagesc` , `rgbplot` ...
- modifier le contraste d'une image : `contrast` ...
- transformations : `imresize` , `imrotate` , `imremap` (transformation géométrique), `imperspectivewarp` (perspective spatiale), `imtranslate` , `rotate_scale`
- conversion de modes d'encodage des couleurs : `rgb2ind` , `ind2rgb` , `hsv2rgb` , `rgb2hsv` , `gray2ind` , `ind2gray` ...
- et autres fonctions relatives à : contrôle de couleur, analyse, statistique, filtrage, fonctions spécifiques pour images noir-blanc ...

6.5 Sauvegarder et imprimer des figures

6.5.1 Imprimer une figure ou la sauvegarder sous forme de fichier graphique

La fonction `print` permet d'**imprimer** directement une figure. Elle permet en outre, ainsi que la fonction `saveas` , de **sauvegarder** la figure sous forme de **fichier graphique** dans un format spécifié (vectorisé ou raster) en vue de l'incorporer ensuite dans un document (alternative au copier/coller).

Rappelons encore ici la possibilité de récupérer une figure sous forme raster par copie d'écran, avec les outils du système d'exploitation (p.ex. la touche alt-PrintScreen qui copie l'image de la fenêtre courante dans le presse-papier).

a) `print({handle,} 'fichier', '-ddevice' {,option(s)})`
b) `print({handle,} {-Pimprimante} {,option(s)})`

a. La figure courante (ou spécifiée par *handle*) est **sauvegardée** sur le *fichier* spécifié, au format défini par le paramètre *device*. Sous Octave, il est nécessaire de spécifier l'extension dans le nom de *fichier*, alors que celle-ci est automatiquement ajoutée par MATLAB (sur la base du paramètre *device*).
Une alternative pour sauvegarder la figure consiste à utiliser le menu de fenêtre de figure : File > Export , File > Save

b. La figure courante (ou spécifiée par *handle*) est **imprimée** sur l'imprimante par défaut (voir `printopt`) ou l'*imprimante* spécifiée.
Une alternative pour imprimer la figure consiste à utiliser le menu de fenêtre de figure : File > Print

Paramètre `device` : valeurs possibles :
Remarque: sous Octave, il est possible d'omettre ce paramètre, le format étant alors déduit de l'extension du nom de fichier !

Vectorisé
`svg` : fichier SVG (Scalable Vector Graphics)
`pdf` : fichier Acrobat PDF
`meta` ou `emf` : sous Windows: copie la figure dans le presse-papier en format vectorisé avec preview (Microsoft Enhanced Metafile) ; sous Octave: génère fichier
`ill` : fichier au format Illustrator 88
`hpgl` : fichier au format HP-GL
Sous Octave, il existe encore différents formats liés à TeX/LaTeX (voir `help print`)

PostScript (vectorisé), nécessite de disposer d'une imprimante PostScript
`ps` , `psc` : fichier PostScript Level 1, respectivement noir-blanc ou couleur
`ps2` , `psc2` : fichier PostScript Level 2, respectivement noir-blanc ou couleur
`eps` , `epsc` : fichier PostScript Encapsulé (EPSF) Level 1, respectivement noir-blanc ou couleur
`eps2` , `epsc2` : fichier PostScript Encapsulé (EPSF) Level 2, respectivement noir-blanc ou couleur
Remarque: pour les fichiers `eps*` sous Octave, importés dans MS Office 2003 on voit le preview, alors qu'on ne le voit pas dans LibreOffice 3.x à 4.1

Raster
`png` : fichier PNG (Potable Network Graphics)
`jpeg` ou `jpegnn` ou `jpg` : fichier JPEG, avec niveau de qualité *nn*= 0 (la moins bonne) à 100 (la meilleure)
`tiff` : fichier TIFF
`gif` : fichier GIF

Spécifique à MATLAB sous **Windows**
`bitmap` : copie la figure dans le presse-papier Windows en format raster
`setup` ou `-v` : affiche fenêtre de dialogue d'impression Windows
`win` , `winc` : service d'impression Windows, respectivement noir-blanc ou couleur

Paramètre *options* : valeurs possibles :
`'-rnnn'` : définit la résolution *nnn* en points par pouce ; par défaut 150dpi pour PNG
`'-Sxsize,ysize'` : spécifie sous Octave, pour les formats PNG et SVG, la taille en pixels de l'image générée
`'-portrait'` ou `'-landscape'` : dans le cas de l'impression seulement, utilise l'orientation spécifiée (par défaut portrait)
`'-tiff'` : ajoute preview TIFF au fichier PostScript Encapsulé
`'-append'` : ajoute la figure au fichier PostScript (donc n'écrase pas fichier)

`saveas(no_figure,'fichier' {,'format'} )`
`saveas(handle,'fichier' {,'format'} )`
Sauvegarde la figure *no_figure* (ou l'objet de graphique identifié par *handle*) sur le *fichier* spécifié et dans le *format* indiqué.
• les différents formats possibles correspondent grosso modo aux valeurs indiquées ci-dessus pour le paramètre *device* de la commande `print`
• si l'on omet le *format*, il est déduit de l'extension du nom du *fichier*
• si l'on omet l'extension dans le nom du *fichier*, elle sera automatiquement reprise du *format* spécifié

`orient portrait | landscape | tall`
Spécifie l'orientation du papier à l'impression (par défaut `portrait`) pour la figure courante.
Sans paramètre, cette commande indique l'orientation courante.
Le paramètre `tall` modifie l'aspect-ratio de la figure de façon qu'elle remplisse entièrement la page en orientation `portrait` .

[*print_cmd*, *device*] = `printopt`
Cette commande retourne :
• *print_cmd* : la commande d'impression par défaut (sous Windows: COPY /B %s LPT1:) qui sera utilisée par `print` dans le cas où l'on ne sauvegarde pas l'image sur un fichier
• *device* : le périphérique d'impression (sous Windows: -dwin)
Pour modifier ces paramètres, il est nécessaire d'éditer le script MATLAB `printopt.m`

nb_polylines = `dxfwrite('fichier', polyline1 {,polyline2 ...})`
Spécifique à Octave et implémentée dans le package "plot" (qui ne semble cependant plus maintenu), cette fonction génère un *fichier* graphique au format AutoCAD **DXF** dans lequel sont graphées la(les) **courbe(s)** *polyline1*, *polyline2*... Ces courbes sont exprimées sous forme de tableaux à 2 colonnes (X/Y) ou 3 colonnes (X/Y/Z). On récupère sur la variable *nb_polylines* le nombre de courbes tracées.

6.5.2 Sauvegarder et recharger une figure ou un objet graphique

De façon analogue aux variables avec la commande `save` , MATLAB et Octave permettent de sauvegarder sur fichier avec la fonction `hgsave` une **figure** ou un **objet graphique**, puis le récupérer ultérieurement avec `hgload` pour l'afficher et le compléter. Sous Octave, ces deux fonctions sont implémentées depuis la version 4.0.

`hgsave('fichier')` *(handle graphics save)*
`hgsave(handle, 'fichier' {,'format'})`
Sauvegarde la figure courante (ou l'objet graphique spécifié par *handle*) dans le *fichier* indiqué.
Sans indiquer d'extension à *fichier* , l'extension `.fig` sera ajoutée sous MATLAB, et l'extension `.ofig` sous Octave.
Le paramètre *format* peut prendre les valeurs : `-v6` (correspond à MATLAB < 7), `-v7` (correspond à MATLAB ≥ 7). Si l'on omet ce paramètre, MATLAB utilise `-v7` et Octave utilise un autre format qui lui est propre.

Remarque : avec MATLAB 8.3 et Octave 4.0.0 nous n'avons pas réussi à échanger de tels fichiers d'un logiciel à l'autre, quel que soit le `format` utilisé. Qui est le coupable ?

`handle = hgload('fichier')` *(handle graphics load)*
Récupére le `handle` de l'objet précédemment sauvegardé dans un `fichier`, et affiche cet objet dans une figure.

La technique ci-dessous est analogue, mais n'est implémentée que sous MATLAB :

`saveas(handle,'fichier','fig')` (remarque: c'est l'option `'fig'` qui n'est pas implémentée dans Octave)
Sauvegarde l'objet graphique spécifié par `handle` dans le `fichier` indiqué.
Pour sauvegarder la figure entière, indiquer `gcf` à la place de `handle`. Sans indiquer d'extension au `fichier`, l'extension `.fig` sera ajoutée.

`open('fichier')`
Recharge et affiche dans une figure l'objet précédemment sauvegardé dans un `fichier`.

6.6 Handle Graphics

Les graphiques MATLAB/Octave sont constitués d'**objets** (systèmes d'axes, lignes, surfaces, textes...). Chaque objet est identifié par un **handle** auquel sont associés différents **attributs** (ou propriétés, *properties*). En modifiant ces attributs, on peut agir très finement sur l'apparence du graphique, voire même l'animer !

Ces objets sont organisés selon une **hérarchie** qui s'établit généralement ainsi (voir l'attribut `type` de ces handles) :

- `root` (*handle*= `0`) : sommet de la hiérarchie, correspondant à l'écran de l'ordinateur
  - `figure` (*handle*= `gcf` = *`numero_figure`*) : fenêtre de graphique,
    ou encore fenêtre d'interface utilisateur graphique sous MATLAB
    - `axes` (*handle*= `gca`) : système(s) d'axes dans la figure
      (ex: `plot` possède 1 système d'axes, alors que `plotyy` en possède 2)
      - `line` : ligne (produite par les fonctions telles que plot, plot3...)
      - `surface` : représentation 3D d'une matrice de valeurs Z (produite pas les fonctions telles que mesh, surf...)
      - `patch` : polygone rempli
      - `text` : chaîne de caractère
      - etc...

Cette hiérarchie d'objets transparaît lorsque l'on examine comment les handles sont reliés entre eux :

- l'attribut `parent` d'un handle fournit le handle de l'objet de niveau supérieur (objet parent)
- l'attribut `children` d'un handle fournit le(s) handle(s) du(des) objet(s) enfant(s)

On est donc en présence d'une "arborescence" d'objets et donc de handles (qui pourraient être assimilés aux *branches* d'un arbre) et d'attributs ou propriétés (qui seraient les *feuilles* au bout de ces *branches*). Chaque **attribut** porte un nom explicite qui est une chaîne de caractère **non case-sensitive**.

Notez finalement que sous Octave les handles sont implémentés depuis la version 2.9/3, mais que les propriétés de handles sont vraiment bien prises en charge depuis les backends **FLTK** (Octave 3.4) et **Qt** (Octave 4.0).

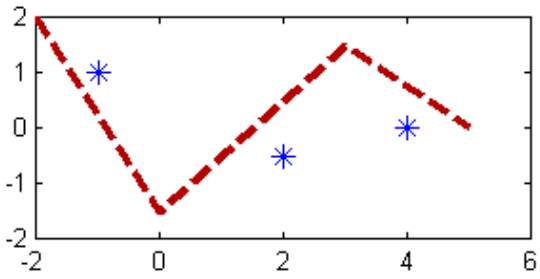
Fonction et description	
Exemple	**Illustration**
A) **Récupérer un handle** : `handle_objet = fonction_graphique(...)` Trace l'objet spécifié par la ***fonction_graphique***, et retourne le *handle_objet* correspondant (qui, selon l'objet, sera un scalaire ou un vecteur de handles). `handle_axes = gca` *(**g**et **c**urrent **a**xis)* Retourne le *handle_axes* du **système d'axes** du graphique courant. Si aucun graphique n'existe, dessine un système d'axes (en ouvrant une fenêtre de figure si aucune figure n'existe). `handle_figure = gcf` *(**g**et **c**urrent **f**igure)* Retourne le *handle_figure* de la **figure** courante. Ce handle est identique au numéro de la figure ! Si aucune fenêtre de figure n'existe, ouvre une nouvelle figure vierge (exactement comme le ferait la fonction `figure`).	
B) **Afficher ou récupérer les attributs (propriétés) relatifs à un handle** : a) `get(handle)` b) *var* = `get(handle, 'PropertyName')` c) *structure* = `get(handle)` **a)** Affiche à l'écran les valeurs courantes des différents attributs (propriétés) de l'objet spécifié par son *handle*. **b)** Récupère sur *var* la valeur (*PropertyValue*) courante correspondant à l'attribut *PropertyName* spécifié. **c)** Récupère sur une *structure* les valeurs courantes des différents attributs (propriétés) de l'objet spécifié par son *handle*.	
C) **Modifier les attributs (propriétés) relatifs à un handle** : `set(handle, 'Property1Name', Property1Value, {'Property2Name', Property2Value, ...} )` Modifie des attributs (propriétés) spécifiées de l'objet désigné par son *handle*. Le nom *PropertyName* des attributs n'est pas case-sensitive. Il est toujours spécifié en tant que chaîne de caractères (entre apostrophes). Les valeurs *PropertyValue* des attributs peuvent être de différents types, selon le type d'attribut (nombre, chaîne,	

tableau...).

Ex 1 Graphique ci-contre réalisé avec MATLAB ou Octave Qt & FLTK.
Sous Octave/Gnuplot, la seule différence est qu'on a une ligne continue

L'exemple ci-dessous illustre un cas simple d'utilisation des handles pour **formater** un graphique.

```
x1=[-2 0 3 5]; y1=[2 -1.5 1.5 0];
x2=[-1 2 4];   y2=[1 -0.5 0];

 h1= plot(x1,y1);     % ligne, récup. handle
 get(h1)   % affiche les attributs de la courbe
hold('on');
 h2= plot(x2,y2,'o'); % symboles, récup. handle
 get(h2)   % affiche attributs du semis points

 get(h1,'parent')
 get(h2,'parent')
 gca     % on voit que h1 et h2 ont même 'parent'
       % qui est le système d'axes !
 get(gca,'parent')
 gcf     % 'parent' du système d'axe est la fig. !
 get(gcf,'parent')  % parent fig. est l'écran !
 get(0)  % affiche les attributs de l'écran

 set(h1,'linewidth',3,'color',[0.7 0 0], ...
       'linestyle','--');  % def. "vraie couleur"
 set(h2,'marker','*','markersize',10);
```

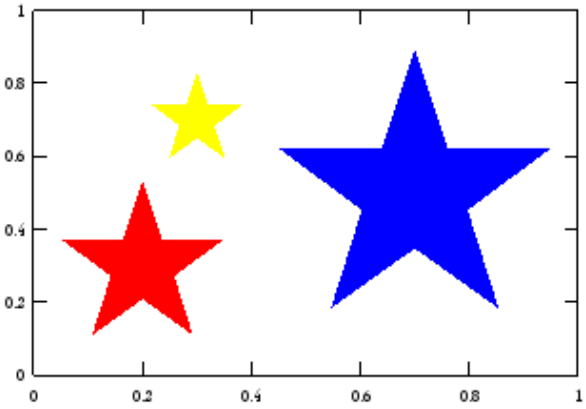


Ex 2 Animation ci-contre réalisée avec MATLAB ou Octave Qt & FLTK.

L'exemple ci-dessous illustre une possibilité d'**animation** basée sur les handles : on joue ici sur la taille des symboles affichés... mais on pourrait jouer sur les données X/Y

```
dmax=[90 50 150];   % dim. max étoiles
 h =scatter([.2 .3 .7], [.3 .7 .5], dmax, ...
  [1 0 0; 1 1 0; 0 0 1],'p','filled'); % aff. étoiles
axis([0 1 0 1])
 get(h)          % affiche les attributs

frames=500;    % nombre d'images de l'animation
duree_max=5;   % durée max. de l'animation [sec]
osc=[8 15 3];  % périodes d'oscillations 3 étoiles

for k=1:frames
  for n=1:3    % numéro de l'étoile
    dims(n)= dmax(n)* (sin(2*pi*k*osc(n)/frames)+1);
  end
   set(h,'sizedata',dims);
  pause(duree_max/frames)  % temporiser animation
end
```

Cliquer sur ce graphique pour voir l'animation !



Remarque: cette animation a été capturée avec le logiciel libre CamStudio,
puis convertie avi->gif-animé (pour affichage dans navigateur web)

Définir les propriétés d'un objet directement lors du dessin (sans passer par son handle) :

fonction_graphique (param. habituels... , 'PropertyName', PropertyValue, 'PropertyName', PropertyValue, ...)

Certaines fonctions de dessin (pas toutes !) permettent de spécifier les propriétés graphiques de l'objet directement lors de l'appel à la fonction, à la suite des paramètres habituels.

Ex 3 Le code ci-dessous produit exactement la même figure que celle de l' **Ex 1** ci-dessus

```
x1=[-2 0 3 5]; y1=[2 -1.5 1.5 0];
x2=[-1 2 4];   y2=[1 -0.5 0];

plot(x1,y1, 'linewidth',3,'color',[0.7 0 0], ...
```

```
            'linestyle','--' );
hold('on');
plot(x2,y2, 'marker','*','markersize',10, ...
            'linestyle','none' );
```

Et Octave serait même capable de réunir ces 2 plots en un seul ! ✗ Bug MATLAB ?

6.7 Animations et movies

Certaines **fonctions de base** MATLAB et Octave permettent de réaliser des **animations interactives**. On a vu, par exemple, la fonction `view` de rotation d'une vue 3D par déplacement de l'observateur, et l'on présente ci-après des fonctions de graphiques animés (`comet` ...). Mais de façon plus générale, l'animation passe par l'exploitation des "**handles graphics**" (changer interactivement les attributs graphiques, la visibilité, voire les données elles-mêmes...).

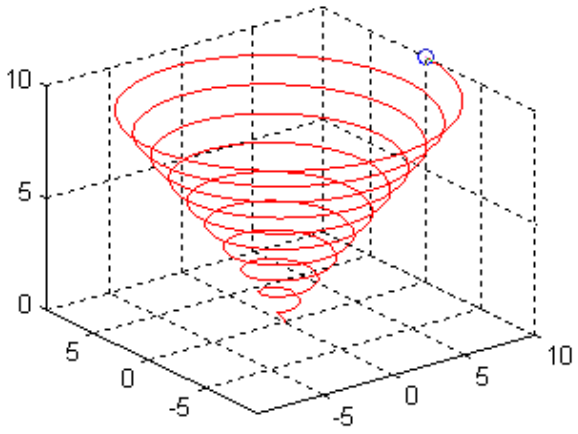
On souhaite parfois aussi sauvegarder une animation sur un fichier indépendant de MATLAB/Octave ("**movie**" stand-alone dans un format vidéo standard), par exemple pour l'intégrer dans une présentation (OpenOffice.org/LibreOffice Impress, MS PowerPoint...).

6.7.1 Graphiques animés, ou fonctions d'animation de graphiques

Graphiques de type "comète"

Propre à MATLAB et simple à mettre en oeuvre, cette technique permet de tracer une courbe 2D ou 3D sous forme d'animation pour "visualiser" une trajectoire. Cela peut être très utile pour bien "comprendre" une courbe dont l'affichage est relativement complexe (enchevêtrement de lignes) en l'affichant de manière progressive, à l'image d'une "**comète**" laissant une trace derrière elle (la courbe).

A titre d'exemple, voyez vous-même la différence en affichant la courbe 3D suivante, successivement avec `plot3` puis `comet3` :
`z= -10*pi:pi/250:10*pi; x= (cos(2*z).^2).*sin(z); y= (sin(2*z).^2).*cos(z);`

Fonction et description	
Exemple	**Illustration**
a) `comet( {x,} y {,p} )` b) `comet3( {x, y,} z {,p} )` Trace la courbe définie par les vecteurs *x*, *y* et *z* à l'aide d'une comète dont la queue a une taille de *p**length(y), *p* étant un scalaire compris entre 0.0 et 1.0 (valeur par défaut 0.1 s'il n'est pas spécifié). **a)** La courbe est tracée dans un **graphique 2D**. Si *x* n'est pas spécifié, cette fonction utilise en *x* les indices du vecteur *y* (c'est-à-dire les valeurs 1 à length(*y*)). **b)** La courbe est tracée dans un **graphique 3D**	
Ex `nb_points=200; % définir en fct vitesse processeur` `nb_tours=10;` `angle=linspace(0,nb_tours*2*pi,nb_points);` `z=linspace(0,10,nb_points);` `x=z.*cos(angle);` `y=z.*sin(angle);` `comet3(x,y,z,0.1)  % affichage progressif courbe !` `grid('on')`	**Cliquer** sur ce graphique pour voir l'animation !  *Remarque: cette animation a été capturée avec le logiciel libre CamStudio, puis convertie avi->gif-animé (pour affichage dans navigateur web)*

6.7.2 Animations de type "movie"

Réaliser une animation sous MATLAB/Octave consiste à assembler une séquence d'images dans un fichier vidéo.

a) Technique basée 'getframe/movie' sous MATLAB

Une animation de type "movie" s'élabore, sous MATLAB, de la façon suivante :

1. le script génère, de façon itérative, autant d'images (graphiques) que nécessaire pour constituer une animation fluide
2. à chaque itération, le graphique courant (la figure) est "capturé" en tant que "frame" (pixmap) via la fonction `getframe` ,

et accumulé sur une immense "matrice-movie"

3. une fois tous les frames capturés, l'animation peut directement être jouée (depuis le script ou en mode commande) en parcourant la "matrice-movie" avec la fonction **movie** ; pour être visualisée indépendemment de MATLAB, l'animation peut être sauvegardée sous forme de fichier-vidéo à l'aide des fonctions **movie2avi** (au format Windows AVI) ou **qtwrite** (au format QuickTime)

Fonction et description	
Exemple	**Illustration**

a) *mov_mat*(*k*) = **getframe**
b) *mov_mat*(*k*) = **getframe**(*handle* {,*rect*})

"Capture" de l'image de la figure courante sous forme de "frame" :
Capture, sous forme de "movie-frame", de l'image de la figure courante, et enregistrement comme *k*-ème élément de la matrice-movie *mov_mat*. Tous les frames saisis doivent avoir la même dimension (largeur x hauteur), raison pour laquelle il ne pas modifier la taille de la fenêtre graphique au cours de l'élaboration de l'animation ! Dans la forme **b)**, on spécifie le *handle* de la figure et l'on peut définir, par le vecteur *rect* de forme [*gauche bas largeur hauteur*], une portion de l'image (unités en pixels).
Il est important de noter que cette capture s'effectue au niveau **raster** ("pixmap"), et que la **taille-mémoire** occupée par le frame (et la matrice-frame) sera proportionelle à la surface (carré des dimensions) de la fenêtre de figure. La dimension originale de la fenêtre graphique dans laquelle s'élabore l'animation a donc beaucoup d'importance sur la quantité de mémoire-vive nécessaire pour l'élaboration du movie, de même que sur la qualité d'affichage ultérieure de l'animation (=> importance de trouver un bon compromis !).

Remarques par rapport à d'anciennes versions de MATLAB :
• depuis MATLAB 5.3, il n'est plus nécessaire de préallouer l'espace-mémoire nécessaire pour la matrice-movie avec la fonction **moviein** (fonction qui devient donc inutile)
• la fonction **getframe** remplace, depuis MATLAB 5.3, l'ancienne fonction **capture** (fonction qui est donc obsolète)

---

movie(*mov_mat* {, *n* {, *fps* } }

Visualisation de movie depuis MATLAB :
Joue l'animation préalablement élaborée dans la matrice-movie *mov_mat*. L'animation sera jouée *n* fois (par défaut 1x), à la cadence de *fps* frames par seconde (par défaut 12 frames/sec). Il est possible de spécifier avec un vecteur *n* le numéro des frames à jouer.

---

a) **movie2avi**(*mov_mat, 'filename'* {, *param, value*})
b) **qtwrite**(*mov_mat, colormap, 'filename'*)

Sauvegarde d'une animation MATLAB sous forme de fichier vidéo indépendant :
a) Sauvegarde l'animation *mov_mat* sur un fichier-vidéo au format Windows AVI (Audio Video Interleaved). Parmi les paramètres possibles (voir l'aide), on peut notamment spécifier une table de couleur, le type de compression (codec), le nombre de frames par secondes, la qualité... La fonction inverse d'importation **aviread** permettrait de lire un fichier AVI sur une matrice-movie.
b) Disponible uniquement sur Macintosh et nécessitant QuickTime, cette fonction sauvegarde l'animation *mov_mat* sur un fichier-vidéo au format QuickTime, en utilisant la table de couleurs *colormap*.

Remarques :
• Comme alternative à ces fonctions d'exportation de movie, on pourrait **sauvegarder sur disque** chaque frame sous forme de **fichier-image** avec les 2 instructions **[***img***]=frame2im(***mov_mat***(***k***));** (voir description de cette fonction ci-dessous) et **imwrite(***img***, ['image-' num2str(n+100) '.jpg'], 'jpg');**, puis assembler les différents fichiers-image **image-***numéro***.jpg** ainsi produits en une animation (MPEG, QuickTime, GIF-animé, AVI...) avec l'un des nombreux utilitaires existant (commerciaux, tel que Animation Shop 3, ou gratuits...).
• Il est bien clair qu'une animation MATLAB peut aussi être sauvegardée, au niveau de sa "matrice-movie" *mov_mat*, sous forme d'un **fichier de workspace** (commande **save** *mat-file mov_mat*), puis être rechargée dans une session MATLAB ultérieure (avec **load** *mat-file*) et directement jouée (avec **movie(***mov_mat***)**) ; mais cette façon de procéder n'est pas efficace car la place mémoire/disque utilisée par la variable *mov_mat* est très importante (pas de compression) ; mieux vaut donc utiliser les formats vidéo classiques.

---

a) *mov_mat*(*k*) = **im2frame(***img* {,*colormap*})
b) [*img* {,*colormap*}] = **frame2im(***mov_mat***(***k***))**

Conversion image <-> frame :
a) Conversion d'une image *img* en un frame de movie *mov_mat*(*k*), en utilisant la table de couleur courante ou la *colormap* spécifiée
b) et vice-versa

b) Technique basée 'avifile' ou fichiers-image, sous MATLAB ou Octave

Avec MATLAB ou Octave, on peut également fabriquer une **vidéo standard** à l'aide des fonctions suivantes (implémentées dans le package "vidéo" sous Octave) :

- ouverture/création du fichier vidéo : `file_id = avifile(file_name, ...)`
- insertion des image (frames) dans le fichier-vidéo : `addframe(file_id, image )`
- les fonctions `aviinfo(file_name)` et `aviread(file_name, frame_no)` peuvent en outre être utiles

On présente ci-dessous 3 implémentations possibles : MATLAB, Octave Windows, Octave Linux.

Implémentation MATLAB (sous Windows ou Linux)

```
% Création/ouverture du fichier vidéo, ouverture fenêtre figure vide
   fich_avi = avifile('animation.avi','compression','Cinepak')  % sous Windows
   % fich_avi = avifile('animation.avi','compression','none')   % sous Linux
   fig=figure;

% Paramètres du graphique
   nb_frames=50;          % nb frames animation
   x=0:0.2:2*pi;  y=x;    % plage de valeurs en X et Y
   [Xm,Ym]=meshgrid(x,y); % matrices grille X/Y

% Boucle de dessin des frames et insertion dans la vidéo
   for n=1:nb_frames;
     z=cos(Xm).*sin(Ym).*sin(2*pi*n/nb_frames);
     surf(x,y,z)                    % affichage n-ième image
     azimut=mod(45+(360*n/nb_frames),360);  % azimut modulo 360 degrés
     view(azimut, 30)               % changement azimut vue
     axis([0 2*pi 0 2*pi -1 1])     % cadrage axes
     axis('off')
     img_frame = getframe(fig);     % récupération frame
     fich_avi = addframe(fich_avi, img_frame); % insertion frame
   end

% Fermeture fichier vidéo et fenêtre figure
   status=close(fich_avi)
   close(fig)
   disp('La video est creee !')
```

Implémentation Octave sous Windows

La fonction `getframe` n'existant pas (encore) sous Octave 4.0, comme artifice on passe par une écriture temporaire des frames sur fichiers-disque. Par rapport à la solution MATLAB ci-dessus, le paramètre `'-rdpi'` (de la commande `print` de sauvegarde des frames) nous permet ici de diminuer la résolution (donc la taille) de la vidéo.

```
% Création/ouverture du fichier vidéo, ouverture fenêtre figure vide
   fich_avi = avifile('animation.avi','compression','msmpeg4v2')  % sous Windows
     % faire   avifile('codecs')   pour liste des codecs utilisables
   fig=figure;     % ouverture fenêtre figure vide

% Création sous-répertoire dans lequel on va enregistrer fichiers frames
   mkdir('frames');

% Paramètres du graphique
   nb_frames=50;          % nb frames animation
   x=0:0.2:2*pi;  y=x;    % plage de valeurs en X et Y
   [Xm,Ym]=meshgrid(x,y); % matrices grille X/Y

% Boucle de dessin des frames et insertion dans la vidéo
   for n=1:nb_frames;
     z=cos(Xm).*sin(Ym).*sin(2*pi*n/nb_frames);
     surf(x,y,z)                    % affichage n-ième image
     azimut=mod(45+(360*n/nb_frames),360);  % azimut modulo 360 degrés
     view(azimut, 30)               % changement azimut vue
     axis([0 2*pi 0 2*pi -1 1])     % cadrage axes
     axis('off')

     fprintf('%d ',n)     % indicateur de progression
     print('-dpng','-r80',sprintf('frames/frame-%04d.png',n))
        % sauvegarde frame sur fichier raster PNG, en résol. 80 dpi
     pause(0.1)   % sinon, risque que le fichier ne soit pas prêt pour lecture
     img_frame = imread(sprintf('frames/frame-%04d.png',n));
        % lecture image à partir fichier
        % => tableau de dim. HxLx3 d'entiers non signés de type uint8 (1 octet)
     img_frame = single(img_frame)/255 ;
```

```
        % normalisation des valeurs: uint8 [0 à 255] -> real-single [0 à 1]
      addframe(fich_avi, img_frame);    % ajout frame à la vidéo
   end

% Fermeture fichier vidéo et fenêtre figure
   clear fich_avi   % ce n'est, bizarrement, pas close qu'il faut utiliser !
   close(fig)
   disp('La video est assemblee, le dossier ''frames'' peut etre detruit !')
```

Implémentation Octave sous Linux

Pour assembler les fichiers-images en une vidéo, on illustre ici l'utilisation efficace sous Linux de l'outil en mode-commande **`ffmpeg`**. On n'a donc, dans ce cas, pas besoin des fonctions **`avifile`** et **`addframe`** présentées plus haut !

```
% Ouverture fenêtre figure vide
   fig=figure;     % ouverture fenêtre figure vide

% Création sous-répertoire dans lequel on va enregistrer fichiers frames
   mkdir('frames');

% Paramètres du graphique
   nb_frames=50;          % nb frames animation
   x=0:0.2:2*pi;  y=x;    % plage de valeurs en X et Y
   [Xm,Ym]=meshgrid(x,y); % matrices grille X/Y

% Boucle de dessin des frames et sauvegarde sur disque
   for n=1:nb_frames;
     z=cos(Xm).*sin(Ym).*sin(2*pi*n/nb_frames);
     surf(x,y,z)                    % affichage n-ième image
     azimut=mod(45+(360*n/nb_frames),360);  % azimut modulo 360 degrés
     view(azimut, 30)               % changement azimut vue
     axis([0 2*pi 0 2*pi -1 1])     % cadrage axes
     axis('off')

     fprintf('%d ',n)     % indicateur de progression
     print('-dpng','-r80',sprintf('frames/frame-%04d.png',n))
       % sauvegarde frame sur fichier raster PNG, en résol. 80 dpi
   end

% Fermeture fenêtre figure
   close(fig)

% Assemblage de la vidéo (par outil Linux et non pas Octave)
   system('ffmpeg -f image2 -i frames/frame-%04d.png -vcodec mpeg4 animation.mp4');
   disp(''); bidon=input('Frames generes ; frapper <enter> pour assembler la video');
     % sous Ubuntu nécessite package "ffmpeg"
     % puis package "gstreamer0.10-ffmpeg" pour visualiser vidéo sous Totem
   disp('La video est assemblee, le dossier ''frames'' peut etre detruit !')
```

6.7.3 Animations basées sur l'utilisation des "handles graphics"

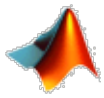
Basée sur les "handles graphics", l'animations de graphiques est plus complexe à maîtriser, mais c'est aussi la plus polyvalente et la plus puissante. Elle consiste, une fois un objet graphique dessiné, à utiliser ses "handles" pour en **modifier les propriétés** (généralement les valeurs **`'xdata'`** et **`'ydata'`** ...) avec la commande MATLAB/Octave **`set(`***`handle, 'PropertyName', PropertyValue, ...`***`)`**. En **redessinant** l'objet après chaque modification de propriétés, on anime ainsi interactivement le graphique.

La fonction **`rotate`** (présentée plus haut), qui permet de faire tourner un objet graphique désigné par son handle, peut également être utile dans le cadre d'animations.

6.7.4 Animations basées sur le changement des données de graphique ("refreshdata")

La technique ci-dessous s'appuie aussi sur les "handles graphics".

Fonction et description	
Exemple	**Illustration**
`refreshdata({`*`handle`***`})`** Cette fonction évalue les propriétés **`'XDataSource'`**, **`'YDataSource'`** et **`'ZDataSource'`** de la figure courante ou de l'objet spécifié par *handle*, et met à jour la figure si les données correspondantes ont changé	



7. Programmation : interaction, debugging, structures de contrôle, scripts, fonctions, entrées-sorties fichier, GUI

7.1 Généralités

Les "**M-files**" sont des fichiers au format texte (donc "lisibles") contenant des instructions MATLAB/Octave et portant l'extension ***.m**. On a vu la commande `diary` (au chapitre "**Workspace**") permettant d'enregistrer un "journal de session" qui, mis à part l'output des commandes, pourrait être considéré comme un M-file. Mais la manière la plus efficace de créer des M-files (c'est-à-dire "programmer" en langage MATLAB/Octave) consiste bien entendu à utiliser un **éditeur** de texte ou de programmation.

On distingue fondamentalement deux types de M-files : les **scripts** (ou programmes) et les **fonctions**. Les scripts travaillent dans le workspace, et toutes les variables créées/modifiées lors de l'exécution d'un script sont donc visibles dans le workspace et accessibles ensuite interactivement ou par d'autres scripts. Les fonctions, quant à elles, n'interagissent avec le workspace ou avec le script appelant essentiellement au moyen des "paramètres" d'entrée/sortie, les autres variables manipulées restant internes (locales) aux fonctions.

MATLAB/Octave est un langage **interprété** (comme les langages Perl, Python, Ruby, PHP, les shell Unix...), c'est-à-dire que les M-files (scripts ou fonctions) sont directement exécutables, donc n'ont pas besoin d'être préalablement compilés avant d'être utilisés (comme c'est le cas des langage classiques C/C++, Java, Fortran...). A l'exécution, des fonctionnalités interactives de **debugging** et de **profiling** permettent d'identifier les bugs et optimiser le code.

MATLAB et Octave étant de véritables progiciels, le **langage** MATLAB/Octave est de "haut niveau" et offre toutes les facilités classiques permettant de développer rapidement des applications interactives évoluées. Nous décrivons dans les chapitres qui suivent les principales possibilités de ce langage dans les domaines suivants :

- interaction avec l'utilisateur (affichage dans la console, saisie au clavier, affichage de warnings/erreurs...)
- structures de contrôle (boucles, tests...)
- élaboration de scripts et fonctions
- commandes d'entrées-sorties (gestion de fichiers, formatage...)
- développement d'interfaces utilisateur graphiques (GUI)

7.2 Éditeurs

Les M-files étant des fichiers-texte, il est possible de les créer et les éditer avec n'importe quel **éditeur de texte/programmation** de votre choix. Idéalement, celui-ci devrait notamment offrir des fonctionnalités d'**indentation** automatique, de **coloration syntaxique**...

7.2.1 Commandes relatives à l'édition

Pour **passer dans l'éditeur**, depuis la fenêtre de commande MATLAB/Octave, afin de **créer ou éditer** un *M-file* :

`edit` *M-file* ou `edit('M-file')` ou `open` *M-file* ou `open('M-file')`
Bascule dans l'éditeur et ouvre le *M-file* spécifié. Si celui-ci n'existe pas, il est proposé de créer un fichier de ce nom (sauf sous MATLAB où `open` retourne une erreur).
Il est obligatoire de passer un nom de fichier à la commande `open` . S'agissant de `edit` , si l'on ne spécifie pas de *M-file* cela ouvre une fenêtre d'édition de fichier vide.

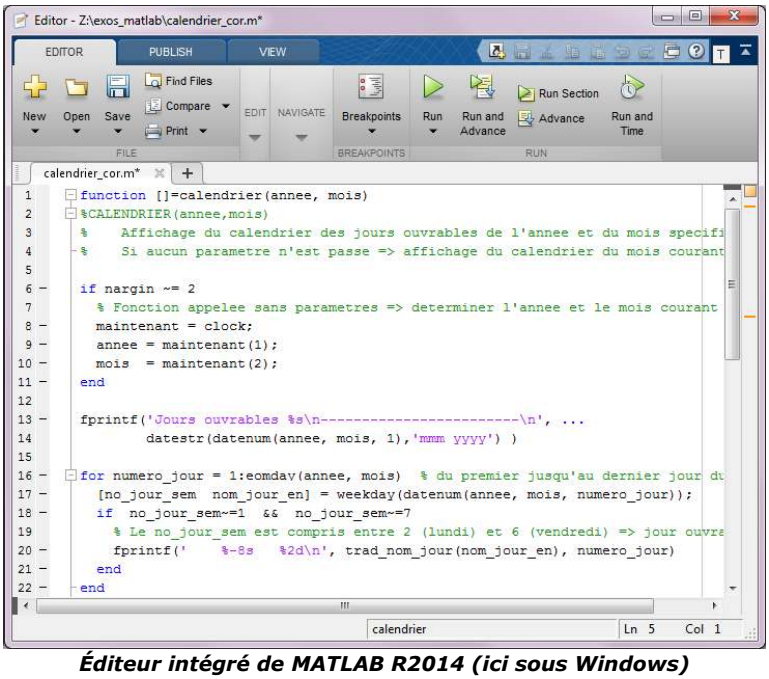
Sous MATLAB et sous Octave GUI, c'est l'éditeur intégré à ces IDE's qui est bien entendu utilisé. Si vous utilisez Octave-CLI (Octave en ligne de commande dans une fenêtre terminal), ce sera l'éditeur défini par la fonction `EDITOR` (voir plus bas).

Depuis les **interfaces graphiques** MATLAB et Octave GUI, on peut bien entendu aussi faire :

- ouvrir un fichier existant : File > Open , bouton Open File , ou double-clic sur l'icône d'un M-file dans l'explorateur de fichiers intégré ("Current folder" ou "File Browser")
- créer un nouveau fichier : New > Script | Function , File > New > New Script | New Function ou bouton New Script

7.2.2 Éditeur/débugger intégré à MATLAB

MATLAB fournit un IDE complet comportant un **éditeur** (illustration ci-dessous) offrant également des possibilités de debugging.



Éditeur intégré de MATLAB R2014 (ici sous Windows)

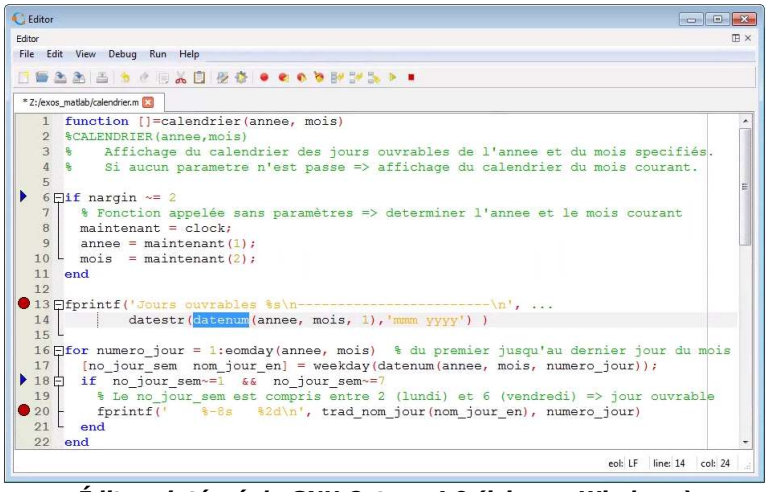
Quelques **fonctionnalités** utiles de l'éditeur intégré de MATLAB :

- Pour **indenter** à droite / désindenter à gauche un ensemble de ligne, sélectionnez-les et faites tab / maj-tab (ou Indent > Increase | Decrease)
- Pour **commenter/décommenter** un ensemble de lignes de code (c'est-à-dire ajouter/enlever devant celles-ci le caractère %), sélectionnez-les et faites ctrl-R / ctrl-T (ou Comment > Comment | Uncomment)
- Concernant l'usage des boutons et raccourcis de **debugging**, voir le chapitre "**Debugging**"

7.2.3 Éditeurs pour GNU Octave

Éditeur/débugger intégré à Octave GUI

Avec l'arrivée de l'interface graphique **Octave GUI** (Octave Graphical User Interface), Octave ≥ 3.8 fournit également un IDE complet comportant un **éditeur** permettant de faire de la coloration syntaxique, autocomplétion, debugging (définition de *breakpoints*, exécution *step-by-step*...).



Éditeur intégré de GNU Octave 4.0 (ici sous Windows)

Quelques **fonctionnalités** utiles de l'éditeur intégré de Octave GUI :

- Sous les 3 systèmes d'exploitation, l'éditeur utilise l'**encodage UTF-8**. Cela ne pose pas de problème sous GNU/Linux et Mac OS X où l'on peut donc utiliser des caractères spéciaux (accentués...).
  Sous Windows cependant, c'est la console Command Window qui ne supporte pas les caractères spéciaux. Donc limitez-vous actuellement sur ce système aux caractères ASCII 7bit (non accentués...).
- Pour **indenter** à droite / désindenter à gauche un ensemble de ligne, sélectionnez-les et faites tab / maj-tab (ou Edit > Format > Indent | Unindent)
- Pour **commenter/décommenter** un ensemble de lignes de code (c'est-à-dire ajouter/enlever devant celles-ci le caractère `%`), sélectionnez-les et faites ctrl-R / ctrl-maj-R (ou Edit > Format > Comment | Uncomment)
- Il est possible de placer des **bookmarks** ▶ dans la marge gauche avec F7 (ou Edit > Navigation > Toggle Bookmark), puis de se déplacer de bookmarks en bookmarks avec F2 (ou Edit > Navigation > Next Bookmark) et maj-F2 (ou Edit > Navigation > Previous Bookmark)
- Concernant l'**autocomplétion** : sous Edit > Preferences onglet "Editor", vous voyez que l'autocomplétion est par défaut activée pour les fonctions, builtins et keywords. Vous pouvez encore activer "Match words in document" pour l'autocomplétion des noms de variables ! Si l'autocomplétion vous dérange, vous pouvez désactiver l'option "Show completion list automatically" et en faire usage à la demande avec ctrl-espace (ou Edit > Commands > Show Completion List). La sélection dans la liste s'effectue avec curseur-haut ou bas, puis la complétion avec enter ou tab
- Lorsque le curseur se trouve juste à gauche ou à droite d'un **crochet/parenthèse/accolade** ouvrant ou fermant, vous pouvez vous déplacer vers le caractère correspondant (fermant ou ouvrant) avec ctrl-M (ou Edit > Navigation > Move to Matching Brace), ou sélectionner tout ce qui se trouve entre deux avec ctrl-maj-M (ou Edit > Navigation > Select to Matching Brace).
- Concernant les **blocs** d'instructions correspondant aux structures de contrôle, vous pouvez cliquer sur les symboles - et + dans la marge gauche pour replier/déplier ces blocs.
- Concernant l'usage des boutons et raccourcis de **debugging**, voir le chapitre "**Debugging**"
- La **personnalisation** de l'éditeur s'effectue avec Edit > Preferences dans les onglets "Editor" et "Editor Styles"

Autres éditeurs pour Octave

Il est cependant possible (et nécessaire lorsqu'on utilise Octave en ligne de commande depuis une fenêtre terminal) de faire appel à d'autres éditeurs de programmation. La situation dépend du système d'exploitation (voir notre chapitre "**Installation/configuration GNU Octave**") :

- sous Windows : les distributions GNU Octave MinGW et MXE intègrent l'éditeur **Notepad++** (auparavant c'était **Scintilla SciTE**), mais d'autres bons éditeurs de programmation font aussi l'affaire, tels que : **Atom**, **ConTEXT**, **cPad**...
- sous Linux : on peut utiliser les éditeurs de base **Gedit** sous GNOME et **Kate** sous KDE, ainsi que **Geany**, **Atom**...
- sous Mac OS X : nous vous recommandons **Atom** (libre), sinon **TextWrangler** (freeware)...

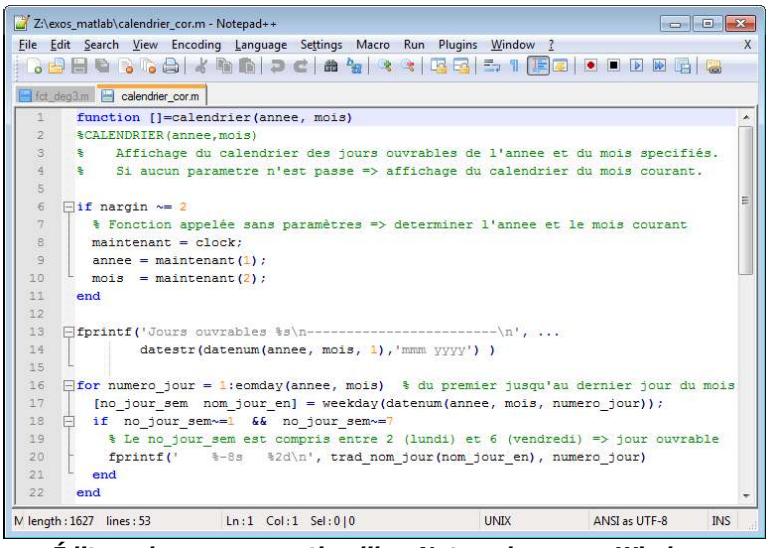
On spécifie quel **éditeur** doit être invoqué lorsque l'on travaille avec **Octave-CLI** (i.e. lorsque l'on passe la commande `edit`) avec la fonction built-in `EDITOR('path/editeur')` que l'on intègre généralement dans son prologue `.octaverc`

Ex : Le morceau de script multi-plateforme ci-dessous teste sur quelle plateforme on se trouve et redéfinit ici "Gedit" comme éditeur par défaut dans le cas où l'on est sous Linux :

```
if ~isempty(findstr(computer,'linux'))
  EDITOR('gedit')        % définition de l'éditeur par défaut
  edit('mode','async')   % passer la commande "edit" de façon détachée
else
  % on n'est pas sous Linux, ne rien faire de particulier
end
```

Système	Éditeur conseillé	Définition de l'éditeur (pour prologue `.octaverc`)	Indenter à droite, désindenter à gauche	Commenter, décommenter
Multiplateforme	**Atom**	`EDITOR('atom')`	Edit>Lines>Indent (ou tab, ou alt-cmd-6) Edit>Lines>Outdent (ou maj-tab, ou alt-cmd-5)	Edit>Toggle Comments (ou maj-cmd-7)
Windows	**Notepad++**	`EDITOR ('path\\notepad++.exe')`	Edit>Indent>Increase (ou tab) Edit>Indent>Decrease	Edit>Comment/Uncom.> Toggle Block Comment (ou ctrl-Q)

			(ou maj-tab)	
Linux	**Gedit** (GNOME)	`EDITOR('gedit')`	tab maj-tab	Edit>Comment Code (ou ctrl-M) Edit>Uncomment Code (ou ctrl-maj-M) (voir cependant ci-dessous)
Mac OS X	**TextWrangler**	`EDITOR('edit')`	Text>Shift Right (ou cmd-]) Text>Shift Left (ou cmd-[)	Il est nécessaire d'élaborer un "script TextWrangler"...



Éditeur de programmation libre Notepad++ sous Windows

Conseils relatifs à l'éditeur Gedit sous Linux

En premier lieu, enrichissez votre éditeur **Gedit** par un jeu de "**plugins**" supplémentaires déjà packagés :

- pour ce faire, sous Linux/Ubuntu, installez le paquet "gedit-plugins" (en passant la commande : `sudo apt-get install gedit-plugins`)
- vous activerez ci-dessous les plugins utiles, depuis Gedit, via Edit > Preferences , puis dans l'onglet " `Plugins` "
- certains de ces plugins peuvent ensuite être configurés via le bouton Configure Plugin

Activation de la **coloration syntaxique** :

- sous Gedit, via View > Highlight Mode > Scientific > Octave (ou via le menu déroulant de langage dans la barre de statut de Gedit)
- activation de la mise en évidence des parenthèses, crochets et acollades : via Edit > Preferences , puis dans l'onglet "View" avtiver "Highlight matching brackets"

Affichage des **numéros de lignes** : via Edit > Preferences , puis dans l'onglet "View" activer "Display line numbers"

Pour pouvoir **mettre en commentaire** un ensemble de lignes sélectionnées :

- d'abord activer le plugin "Code comment"
- on peut dès lors utiliser, dans le menu Edit , les commandes Comment code (raccourci ctrl-M) et Uncomment code (ctrl-maj-M)

Fermeture automatique des **parenthèses, crochets, acollades, apostrophes** ... : en activant simplement le plugin "Bracket Completion"

Affichage des **caractères spéciaux** tab, espace ... : en activant (et configurant) le plugin "Draw Spaces"

Pour **automatiser certaines insertions** (p.ex. structures de contrôles...) :

- activer le plugin "Snippets"
- puis, pour par exemple faire en sorte que si vous frappez `if` tab cela insère automatiquement l'ensemble de lignes

suivantes :

```
if espace
tab
else
tab
end
```

définissez avec `Tools > Manage Snippets`, dans la catégorie "Octave", avec le bouton `+` (Create new snippet), un snippet nommé `if` avec les attributs :

- tab trigger : `if`
- dans le champ Edit : le code à insérer figurant ci-dessus
- shortcut key (facultatif) : associez-y un raccourci clavier

Et réalisez ainsi d'autres snippets, par exemples pour les structures : `for...end`, `while...end`, `do...until`, `switch...case`...

7.3 Interaction écran/clavier, warnings et erreurs

Pour être en mesure de développer des scripts MATLAB/Octave interactifs (affichage de messages, introduction de données au clavier...) et les "débugger", MATLAB et Octave offrent bon nombre de fonctionnalités utiles décrites dans ce chapitre.

7.3.1 Affichage de texte et de variables

`disp(variable)`
`disp('chaîne')`

Affiche la *variable* ou la *chaîne* de caractère spécifiée. Avec cette commande, et par oposition au fait de frapper simplement `variable`, seul le contenu de la variable est affiché et pas son nom. Les nombres sont formatés conformément à ce qui a été défini avec la commande `format` (présentée au chapitre "**Fenêtre de commande**").

Ex: les commandes `M=[1 2;3 5] ; disp('La matrice M vaut :') , disp(M)` produisent l'affichage du texte "La matrice M vaut :" sur une ligne, puis celui des valeurs de la matrice M sur les lignes suivantes

{*count*=} `fprintf('format', variable(s)...)`
{*count*=} `printf('format', variable(s)...)`

Affiche, de façon formatée, la(les) *variable(s)* spécifiées (et retourne facultativement le nombre *count* de caractères affichés). Cette fonction ainsi que la syntaxe du `format`, repris du langage de programmation C, sont décrits en détails au chapitre "**Entrées-sorties**".
L'avantage de cette méthode d'affichage, par rapport à `disp`, est que l'on peut afficher plusieurs variables, agir sur leur formatage (nombre de chiffres après le point décimal, justification...) et entremêler texte et variables sur la même ligne de sortie.

Ex: si l'on a les variables `v=444; t='chaîne de car.';`, l'instruction `fprintf('variable v= %6.1f et variable t= %s \n',v,t)` affiche, sur une seule ligne : "variable v= 444.0 et variable t= chaîne de car."

7.3.2 Affichage et gestion des avertissements et erreurs, beep

Les **erreurs** sont des évènements qui provoquent l'arrêt d'un script ou d'une fonction, avec l'affichage d'un message explicatif.
Les **avertissements** (**warnings**) consistent en l'affichage d'un message sans que le déroulement soit interrompu.

`warning( {'id',} 'message')`
`warning( {'id',} 'format', variable(s)...)`

Affiche le `message` spécifié sous la forme "warning: *message*", puis continue (par défaut) l'exécution du script ou de la fonction.
Le *message* peut être spécifié sous la forme d'un `format` (voir spécification des "Formats d'écriture" au chapitre "**Entrées-sorties**"), ce qui permet alors d'incorporer une(des) `variable(s)` dans le message !
L'identificateur `id` du message prend la forme `composant{:composant}:mnémonique`, où :
- le premier *composant* spécifie p.ex. le nom du package
- le second *composant* spécifie p.ex. le nom de la fonction
- le *mnémonique* est une notation abrégée du message

L'identificateur `id` est utile pour spécifier les conditions de traitement de l'avertissement (voir ci-dessous).
Sous Octave, une description de tous les types de warnings prédéfinis est disponible avec `help warning_ids`

struct = `warning`

Passée sans paramètre, cette fonction **indique** de quelle façon sont traités les différents types de messages d'avertissements (warnings). Les différents états possibles sont :
`on` = affichage du message d'avertissement, puis continuation de l'exécution
`off` = pas d'affichage de message d'avertissement et continuation de l'exécution
`error` = condition traitée comme une **erreur**, donc affichage du message d'avertissement puis interruption !

`warning('on|off|error', 'id' )`

Changement de la **façon de traiter** les avertissements du type `id` spécifié. Voir ci-dessus la signification des conditions `on`, `off` et `error`. On arrive ainsi à désactiver (off) certains types d'avertissements, les réactiver (on), ou même les faire traiter comme des erreurs (error) !

`warning('query', 'id' )`
Récupère le statut courant de traitement des warnings de type *id*

{*string*=} `lastwarn`
Affiche (ou récupère sur la variable *string*) le dernier message d'avertissement (warning)

Ex:
- `X=123; S='abc'; warning('Demo:test','X= %u et chaine S= %s', X, S)` affiche l'avertissement : 'warning: X= 123 et chaîne S= abc'
- puis si l'on fait `warning('off','Demo:test')` et que l'on exécute à nouveau le `warning` ci-dessus, il n'affiche plus rien
- puis si l'on fait `warning('error','Demo:test')` et que l'on exécute à nouveau le `warning` ci-dessus, cela affiche cette fois-ci une erreur : 'error: X vaut: 123 et la chaîne S: abc'

`error('message')`
`error('format', variable(s)...)`
Affiche le *message* indiqué sous la forme "error: *message*", puis interrompt l'exécution du script ou de la fonction dans le(la)quel(le) cette instruction a été placée, ainsi que l'exécution du script ou de la fonction appelante. Comme pour `warning` , le *message* peut être spécifié sous la forme d'un *format* , ce qui permet alors d'incorporer une(des) *variable(s)* dans le message.

Sous **Octave**, si l'on veut éviter qu'à la suite du *message* d'erreur soit affiché un "traceback" de tous les appels de fonction ayant conduit à cette erreur, il suffit de terminer la chaîne *message* par le caractère "newline", c'est-à-dire définir `error("message... \n")` . Mais comme on le voit, la chaîne doit alors être définie entre guillemets et non pas entre apostrophes, ce qui pose problème à MATLAB. Une façon de contourner ce problème pour faire du code portable pour Octave et MATLAB est de définir `error(sprintf('message... \n'))`

Remarque générale : Lorsque l'on programme une fonction, si l'on doit prévoir des cas d'interruption pour cause d'erreur, il est important d'utiliser `error(...)` et non pas `disp('message'); return` , afin que les scripts utilisant cette fonction puissent tester les situations d'erreur (notamment avec la structure de contrôle `try...catch...end`).

{*string*=} `lasterr`
Affiche (ou récupère sur la variable *string*) le dernier message d'erreur

`beep`
Émet un beep sonore

7.3.3 Entrée d'information au clavier

a) *variable*= `input('prompt') ;`
b) *chaîne*= `input('prompt', 's') ;`
MATLAB/Octave affiche le *prompt* ("invite") spécifié, puis attend que l'utilisateur entre quelque-chose au clavier terminé par la touche enter

a) En l'absence du paramètre `'s'` , l'information entrée par l'utilisateur est "interprétée" (évaluée) par MATLAB/Octave, et c'est la valeur résultante qui est affectée à la *variable* spécifiée. L'utilisateur peut donc, dans ce cas, saisir une donnée de n'importe quel type et dimension (nombre, vecteur, matrice...) voire toute expression valide !
On peut, après cela, éventuellement détecter si l'utilisateur n'a rien introduit (au cas où il aurait uniquement frappé enter) avec : `isempty(variable)` , ou `length(variable)==0`
b) Si l'on spécifie le second paramètre `'s'` (signifiant string), le texte entré par l'utilisateur est affecté tel quel (sans évaluation) à la variable *chaîne* indiquée. C'est donc cette forme-là que l'on utilise pour saisir interactivement du texte.

Dans les 2 cas, on place généralement, à la fin de cette commande, un `;` pour que MATLAB/Octave "travaille silencieusement", c'est-à-dire ne quittance pas à l'écran la valeur qu'il a affectée à la variable.

Ex:
- la commande `v1=input('Entrer v1 (scalaire, vecteur, matrice, expression, etc...) : ') ;` affiche "Entrer v1 (scalaire, vecteur, matrice, expression, etc...) : " puis permet de saisir interactivement la

variable numérique "v1" (qui peut être de n'importe quel type/dimension)
• la commande `nom=input('Entrez votre nom : ', 's') ;` permet de saisir interactivement un nom (contenant même des espace...)

a) `pause`
b) `pause(secondes)` ou `sleep(secondes)`
a) Lorsque le script rencontre cette instruction sans paramètre, il effectue une **pause**, c'est-à-dire attend que l'utilisateur frappe **n'importe quelle touche** au clavier pour continuer son exécution.
b) Si une **durée** `secondes` est spécifiée, le script reprend automatiquement son exécution après cette durée.
Sous MATLAB, on peut passer la commande `pause off` pour désactiver les éventuelles pauses qui seraient effectuées par un script (puis `pause on` pour rétablir le mécanisme des pauses).

7.4 Debugging, optimisation, profiling

7.4.1 Debugging

Lorsqu'il s'agit de débuguer un script ou une fonction qui pose problème, la première idée qui vient à l'esprit est de parsemer le code d'instructions d'**affichages intermédiaires**. Plutôt que de faire des `disp`, on peut alors avantageusement utiliser la fonction `warning` présentée plus haut, celle-ci permettant en une seule instruction d'afficher du texte et des variables ainsi que de désactiver/réactiver aisément l'affichage de ces warnings. Mais il existe des fonctionnalités de debugging spécifiques présentées ci-après.

Commandes de debugging simples

`echo on | off`
`echo on all | off all`
• Active (`on`) ou désactive (`off` , c'est le cas par défaut) l'affichage/écho de toutes les commandes exécutées par les **scripts**
• Active (`on all`) ou désactive (`off all` , c'est le cas par défaut) l'affichage/écho de toutes les commandes exécutées par les **fonctions**

Petites différences de comportement entre Octave et MATLAB :
L'affichage est plus agréable dans Octave, chaque commande exécutée étant clairement identifiée par un signe `+` (signe que l'on peut changer avec la fonction `PS4`)

`keyboard`
`keyboard('prompt ')`
Placée **à l'intérieur** d'un M-file, cette commande invoque le mode de debugging "keyboard" de MATLAB/Octave : l'exécution du script est suspendue, et un prompt spécifique s'affiche (`K>>` , respectivement `debug>` ou le *prompt* spécifié). L'utilisateur peut alors travailler normalement en mode interactif dans MATLAB/Octave (visualiser ou changer des variables, passer des commandes...). Puis il a le choix de :
• continuer l'exécution du script en frappant en toutes lettres la commande `return`
• ou avorter la suite du script en frappant la commande `dbquit` sous MATLAB ou Octave
Ce mode "keyboard" permet ainsi d'analyser manuellement certaines variables en cours de déroulement d'un script.

Debugging en mode graphique

Les **éditeurs/débuggers intégrés** de MATLAB et de Octave GUI (depuis Octave 3.8) permettent de placer visuellement des *breakpoints* (points d'arrêt) dans vos scripts/fonctions, puis d'exécuter ceux-ci en mode *step-by-step* (instructions pas à pas). L'intérêt réside dans le fait que lorsque l'exécution est suspendue sur un *breakpoint* ou après un *step*, vous pouvez, depuis la fenêtre de console à la suite du prompt `K>>` ou `debug>` , passer interactivement toute commande MATLAB/Octave (p.ex. vérifier la valeur d'une variable, la modifier...), puis poursuivre l'exécution.

Les boutons décrits ci-dessous se cachent dans l'onglet EDITOR du bandeau MATLAB, et dans palette d'outils de l'éditeur intégré de Octave GUI.

A) Mise en place de **breakpoints** :

- clic dans la marge gauche de l'éditeur, F12, Breakpoint > Set/Clear , Toggle Breakpoint : place/supprime un breakpoint sur la ligne courante, symbolisé par un disque rouge ● (identique à `dbstop` / `dbclear`)

- Next Breakpoint et Previous Breakpoint : déplace le curseur d'édition au beakpoint suivant/précédent du fichier
- Breakpoints > Clear all , Remove All Breakpoints : supprime tous les breakpoints qui ont été définis

B) Puis exécution **step-by-step** :

- Save (File) and Run ou F5 : débute l'exécution du script et suspend l'exécution au premier breakpoint rencontré, la console affichant alors le prompt `K>>` ou `debug>`
- dans la marge gauche une flèche (verte ➡ sous MATLAB, jaune ➡ sous Octave) pointe sur la prochaine instruction qui sera exécutée
- Step ou F10 : exécute la ligne courante et se suspend au début de la ligne suivante (identique à `dbstep`) ; si la ligne courante est un appel de fonction, exécute le reste du code de la fonction d'une traite
- Step In ou F11 : si la ligne courante est un appel de fonction, exécute aussi les instructions de celle-ci en mode step-by-step (identique à `dbstep in`)
- Step Out ou maj-F11 : lorsque l'on est en mode step-by-step dans une fonction, poursuit l'exécution de celle-ci jusqu'à la sortie de la fonction (identique à `dbstep out`)
- Continue ou F5 : poursuit l'exécution jusqu'au breakpoint suivant (identique à `return`)
- Quit Debugging , Exit Debug Mode ou maj-F5 : interrompt définitivement l'exécution (identique à `dbquit`)

Fonctions de debugging

Les fonctions ci-dessous sont implicitement appelées lorsque l'on fait du debugging en mode graphique (chapitre précédent), mais vous pouvez aussi les utiliser depuis la console MATLAB/Octave.

A) Mise en place de **breakpoints** :

`dbstop in script|fonction at no`
`dbstop('script|fonction', no {, no, no...} )` ou `dbstop('script|fonction', vecteur_de_nos)`

Défini (ajoute), lors de l'exécution ultérieure du *script* ou de la *fonction* indiquée, des breakpoints au début des lignes de *no* spécifié
L'avantage, par rapport à l'instruction `keyboard` décrite précédemment, est qu'ici on ne "pollue" pas notre code, les breakpoints étant définis interactivement avant l'exécution

`dbclear in script|fonction at no` respectivement `dbclear in script|fonction`
`dbclear('script|fonction', no {, no, no...} )` respectivement `dbclear('script|fonction')`

Supprime, dans le *script* ou la *fonction* indiquée, les breakpoints précédemment définis aux lignes de *no* spécifié.
Dans sa seconde forme, cette instruction supprime tous les breakpoints relatif au script/fonction spécifié.

struct = `dbstatus {script|fonction}`
struct = `dbstatus {('script|fonction')}`

Affiche (ou retourne sur une *structure*) les vecteurs contenant les nos de ligne sur lesquels sont couramment définis des breakpoints. Si on ne précise pas de script/fonction, retourne les breakpoints de tous les scripts/fonctions

B) Puis exécution en mode **step-by-step** à l'aide des fonctions MATLAB/Octave suivantes :

- l'exécution s'arrête automatiquement au premier beakpoint spécifié, et la console affiche le prompt `K>>` ou `debug>`
- `dbstep {n}` : exécution de la (des *n*) **ligne(s) suivante(s)** du script/fonction
  `dbstep in` : si la ligne courante est un appel de fonction, passe à l'exécution de celle-ci en mode step-by-step
  `dbstep out` : si l'on est en mode step-by-step dans une fonction, poursuit l'exécution de celle-ci jusqu'à la sortie de la fonction
- `return` (commande passée en toutes lettres) : continuation de l'exécution jusqu'au **breakpoint suivant**
- `dbcont` : **achève l'exécution** en ignorant les breakpoints qui suivent
- enter : la commande de debugging précédemment passée est répétée ; sous MATLAB, faire curseur-haut enter
- `dbwhere` : affichage du numéro (et contenu) de la **ligne courante** du script/fonction
- `dbquit` : **avorte** l'exécution du script/fonction

S'agissant d'exécution de scripts/fonctions imbriqués, on peut encore utiliser les commandes de debugging `dbup` , `dbdown` , `dbstack` ...

7.4.2 Optimisation

Il existe de nombreuses techniques pour optimiser un code MATLAB/Octave en terme d'utilisation des ressources processeur et mémoire. Nous donnons ci-après quelques conseils de base. Notez cependant qu'il faut parfois faire la balance entre en optimisation et lisibilité du code.

Optimisation du temps de calcul (CPU)

- Évitez autant que possible les boucles `for`, `while` ... en utilisant massivement les capacités vectorisées de MATLAB/Octave (la plupart des fonctions admettant comme arguments des tableaux). Pensez aussi à l'"**indexation logique**".
- Redimensionner dynamiquement un tableau dans une boucle peut être très coûteux en temps CPU. Il est plus efficace de pré-allouer l'espace du tableau avant d'entrer dans la boucle, quitte à libérer ensuite l'espace non utilisé. Considérons par exemple la boucle ci-dessous :
  ```
  t0=cputime; for k=1:10000000, mat(k)=k; end; fprintf('%6.2f secondes\n', cputime-t0)
  ```
  Le fait de pré-allouer l'espace de `mat`, en exécutant par exemple `mat=zeros(1,10000000);` ou `mat(10000000)=1;` avant ce code, accélérera celui-ci d'un facteur 40x sous MATLAB 8.3 et 5x sous Octave 3.8
- S'agissant de matrices particulières (symétriques, diagonales...), il existe des fonctions MATLAB/Octave spécifiques qui sont plus efficaces que les fonctions standards. Pensez aussi au stockage *sparse* de matrices creuses (voir ci-après).

Optimisation de l'utilisation mémoire (RAM)

- Les nombres sont par défaut stockés en virgule flottante "double précision" (16 chiffres significatifs) occupant en mémoire 8 octets par nombre (64 bits). Si vous gérez des gros tableaux de nombres qui ne nécessitent pas cette précision, un gain de place important peut être obtenu en initialisant ces tableaux en virgule flottante "simple précision" (7 chiffres significatifs) occupant 4 octets par élément. S'il s'agit de nombre entiers, envisagez les types entiers 32 bits (plage de -2'147'483'648 à 2'147'483'647), 16 bits (de -32'768 à 32'767) ou 8 bits (-128 à 127). Voyez pour cela notre chapitre "**Types de nombres**".
- Si vous manipulez des matrices comportant beaucoup d'éléments nuls (vous pouvez visualiser cela avec `spy(matrice)`), pensez à les stocker sous forme *sparse* (conversion avec `sparse(matrice)`, et conversion inverse avec `full(sparse)`). Elles occuperont moins d'espace mémoire, et les opérations de calcul s'en trouveront accélérées (réduction du nombre d'opérations, celles portant sur les zéros n'étant pas effectuées).

7.4.3 Profiling

Sous le terme de "**profiling**" on entend l'**analyse des performances** d'un programme (script, fonctions) afin d'identifier les parties de code qui pourraient être **optimisées** dans le but d'améliorer les performances globales du programme. Les outils de profiling permettent ainsi de comptabiliser de façon fine (au niveau script, fonctions et même instructions) le temps consommé lors de l'exécution du programme, puis de présenter les résultats de cette analyse sous forme de tableaux et d'explorer ces données.

Pour déterminer le temps CPU utilisé dans certaines parties de vos scripts ou fonctions, une alternative aux outils de profiling ci-dessous serait d'ajouter manuellement dans votre code des fonctions de "**timing**" (chronométrage du temps consommé) décrites au chapitre "**Dates et temps**", sous-chapitre "Fonctions de timing et de pause".

Commandes liées au profiling

Enclencher/déclencher le processus de profiling :

`profile on` ou `profile('on')`
`profile resume` ou `profile('resume')`
: Démarre le profiling, c'est-à-dire la comptabilisation du temps consommé :
• avec `on` : efface les données de profiling précédemment collectées
• avec `resume` : reprend le profiling qui a été précédemment suspendu avec `profile off`, sans effacer données collectées

`profile off` ou `profile('off')`
: Interrompt ou suspend la collecte de données de profiling, afin d'en exploiter les données

stats_struct = `profile('status')`
: Retourne la structure *stats_struct* dans laquelle le champ `ProfilerStatus` indique si le profiling est courrament activé (`on`) ou stoppé (`off`)

prof_struct = `profile('info')`

Récupère sur la structure *prof_struct* les données de profiling collectées

`profile clear`
Efface toutes les données de profiling collectées

Explorer sous **MATLAB** les données de profiling :

`profile viewer` ou **`profile('viewer')`** ou **`profview`**
Interrompt le profiling (effectue implicitement un **`profile off`**) et ouvre l'**explorateur de profiling** (le "Profiler"). Il s'agit d'une fenêtre MATLAB spécifique dans laquelle les données de profiling sont hiérarchiquement présentées sous forme HTML à l'aide de liens hyper-textes.

Examiner sous **Octave** les données de profiling :

`profshow(`*`prof_struct {, N }`***`)`**
Affiche sous forme de tableau, dans l'ordre descendant du temps consommé, les données de profiling *prof_struct*. On peut spécifier le nombre *N* de fonctions/instructions affichées. Si *N* n'est pas spécifié, ce sera par défaut 20.

`profexplore(`*`prof_struct`***`)`**
Explore interactivement, dans la fenêtre de commande Octave, les données hiérarchiques de profiling *prof_struct*
- **`help`** : aide en-ligne sur les commandes disponibles
- ***`opt`*** : descend d'un niveau dans l'option *opt* spécifiée
- **`up {`***`nb_niv`***`}`** : remonte d'un niveau, ou de *nb_niv* niveaux ; si l'on est au premier niveau, retourne au prompt Octave
- **`exit`** : interrompt cette exploration interactive

Illustration par un exemple

Soit le script et les 2 fonctions suivants :

Script **`demo_profiling.m`** :

```
%DEMO_PROFILING   Script illustrant, par
%  profiling, l'utilité de vectoriser son code !

t0=cputime;      % compteur temps CPU consommé

% Génération matrice aléatoire
nb_l = 1000;
nb_c =  500;
v_min = -10 ;
v_max =  30 ;
mat = matrice_alea(nb_l, nb_c, v_min, v_max) ;

% Extraction de certains éléments de mat
vmin =  0 ;
vmax = 20 ;
vec = extrait_matrice(mat, vmin, vmax) ;

% Calcul de la moyenne des éléments extraits
if CODE_VECTORISE  % Code vectorisé  :-)
  moyenne_vect = mean(vec) ;
else               % Code non vectorisé  :-(
  somme_elem = 0 ;
  for indice=1:length(vec)
    somme_elem = somme_elem + vec(indice) ;
  end
  moyenne_vect = somme_elem / length(vec) ;
end

% Affichages...
moyenne_vect
duree_calcul=cputime-t0
```

Fonction **`matrice_alea.m`** :

```
function [matrice]=matrice_alea(nb_l,nb_c,v_min,v_max)
% MATRICE_ALEA(NB_L, NB_C, V_MIN, V_MAX)
%   Generation matrice de dimension (NB_L, NB_C)
%   de nb aleatoires compris entre V_MIN et V_MAX

  global CODE_VECTORISE
  v_range = v_max - v_min ;

  if CODE_VECTORISE  % Code vectorisé  :-)
    matrice = (rand(nb_l, nb_c) * v_range) + v_min ;

  else               % Code non vectorisé  :-(
    for lig=1:nb_l
      for col=1:nb_c
        matrice(lig,col) = (rand()*v_range) + v_min ;
      end
    end
  end
return
```

Fonction **`extrait_matrice.m`** :

```
function [vecteur] = extrait_matrice(mat, vmin, vmax)
% EXTRAIT_MATRICE(MAT, VMIN, VMAX)
%   Extrait de la matrice MAT, parcourue col. apres
%   colonne, tous les elements dont la val. est
%   comprise entre VMIN et VMAX, et les retourne
%   sur un vecteur colonne

  global CODE_VECTORISE

  if CODE_VECTORISE  % Code vectorisé (index. logique) :
    vecteur=mat(mat>=vmin & mat<=vmax);

  else               % Code non vectorisé  :-(
    indice = 1;
    for col=1:size(mat,2)
      for lig=1:size(mat,1)
```

```
        if (mat(lig,col) >= vmin) && (mat(lig,col) <= vm
          vecteur(indice) = mat(lig,col) ;
          indice = indice + 1 ;
        end
      end
    end
    vecteur = vecteur' ;
  end
return
```

Vous constatez que, pour les besoins de l'exemple, ces codes comportent du code vectorisé (usage de fonctions vectorisées, indexation logique...) et du code non vectorisé (usage de boucles for/end). Nous avons délimité ces deux catégories de codes par des structures if/else/end s'appuyant sur une variable globale nommée `CODE_VECTORISE`.

Réalisons maintenant le profiling de ce code. En premier lieu, il faut rendre globale au niveau workspace et du script la variable `CODE_VECTORISE` (ceci est déjà fait dans les fonctions), avec l'instruction :

- `global CODE_VECTORISE` % il est important de faire cette déclaration avant d'affecter une valeur à cette variable !

Les durées d'exécution indiquées ci-dessous se rapportent à une machine Intel Pentium Core 2 Duo E6850 @ 3 GHz avec MATLAB R2012 et GNU Octave 3.6.2 MinGW.

Commençons par l'examen du **code non vectorisé** :

1. `CODE_VECTORISE=false;` % on choisit donc d'utiliser ici le code non vectorisé
2. `profile on` % démarrage du profiling
3. `profile status` % contrôle du statut du profiling (il est bien à `on`)
4. `demo_profiling` % exécution de notre script
5. `profile off` % interruption de la collecte de données de profiling
6. `profile status` % contrôle du statut du profiling (il est bien à `off`)

7. Sous **MATLAB** :
   `profile viewer` % ouverture de l'explorateur de profiling
   Constatations :
   - le script s'est exécuté en 9 sec sous Linux/Ubuntu, et étonnamment en 1 sec sous Windows !
   - ✗ le profiler de cette version MATLAB sous Linux/Ubuntu donne des timing erronés, et l'exploitation des hyper-liens est impossible (retourne des erreurs) !

8. Sous **Octave** :
   `prof_struct=profile('info');` % récupération des données de profiling
   `profshow(prof_struct)` % affichage tabulaire des 20 plus importants données (en temps) de profiling
   `profexplore(prof_struct)` % exploration interactive des données de profiling ; "descendre" ensuite avec `1` dans "demo_profiling", puis dans les fonctions "matrice_alea" et "extrait_matrice" ...
   Constatations :
   - le script s'est exécuté en 14 sec sous Linux, et en 23 sec sous Windows
   - sous Windows p.ex. 12 sec sont consommées par la fonction "matrice_alea" (dont 4.5 sec par la fonction "rand"), 9 sec par la fonction "extrait_matrice"
   - on constate notamment que la fonction "rand" est appelée 500'000x
   - on voit donc ce qu'il y a lieu d'optimiser...

Essayez de relancer ce script sans profiling. Vous constaterez qu'il s'exécute un peu plus rapidement, ce qui montre que **le profiling lui-même consomme du temps CPU**, et donc qu'il ne faut l'activer que si l'objectif est bien de collecter des données de performance !

Poursuivons maintenant avec l'examen du **code vectorisé** (remplacement des boucles par des fonctions vectorisées et l'indexation logique) :

1. `CODE_VECTORISE=true;` % on choisit donc d'utiliser ici le code vectorisé
2. `profile on` % démarrage du profiling
3. `demo_profiling` % exécution de notre script
4. `profile off` % interruption de la collecte de données de profiling

5. Sous **MATLAB** :
   `profile viewer` % ouverture de l'explorateur de profiling
   Constatations :
   - le script s'est exécuté en 0.1 sec sous Linux/Ubuntu et sous Windows !
   - on a donc gagné d'un facteur de plus de 100x par rapport à la version non vectorisée !

6. Sous **Octave** :
   ```
   prof_struct=profile('info'); % récupération des données de profiling
   profshow(prof_struct) % affichage tabulaire des 20 plus importants données (en temps) de profiling
   profexplore(prof_struct) % exploration interactive des données de profiling...
   ```
   Constatations :
   - le script s'est exécuté en 0.1 sec sous Windows, et 0.05 sec sous Linux/Ubuntu !
   - on a donc gagné d'un facteur de plus de 200x par rapport à la version non vectorisée !

7.5 Structures de contrôle

Les "**structures de contrôle**" sont, dans un langage de programmation, des constructions permettant d'exécuter des blocs d'instructions de façon itérative (boucle) ou sous condition (test). MATLAB/Octave offre les structures de contrôle de base typiques présentées dans le tableau ci-dessous et qui peuvent bien évidemment être "emboîtées" les unes dans les autres. Notez que la syntaxe est différente des structures analogues dans d'autres langages (C, Java, Python).

Comme MATLAB/Octave permet de travailler en "format libre" (les caractères espace et tab ne sont pas significatifs), on recommande aux programmeurs MATLAB/Octave de bien "**indenter**" leur code, lorsqu'ils utilisent des structures de contrôle, afin de faciliter la lisibilité et la maintenance du programme.

Octave propose aussi des variations aux syntaxes présentées plus bas, notamment : `endfor` , `endwhile` , `endif` , `endswitch` , `end_try_catch` , ainsi que d'autres structures telles que:
`unwind_protect` *body...* `unwind_protect_cleanup` *cleanup...* `end_unwind_protect`
Nous vous recommandons de vous en passer pour que votre code reste portable !

Structure	Description
Boucle **for...end** `for` *var* `=` *tableau* *instructions...* `end`	Considérons d'abord le cas général où *tableau* est une **matrice 2D** (de nombres, de caractères, ou cellulaire... peu importe). Dans ce cas, l'instruction `for` parcourt les différentes colonnes de la matrice (c'est-à-dire *matrice*`(:,`*i*`)`) qu'il affecte à la variable *var* qui sera ici un vecteur colonne. À chaque "itération" de la boucle, la **colonne** suivante de *matrice* est donc affectée à *var*, et le bloc d'*instructions* est exécuté. Si *tableau* est un **vecteur**, ce n'est qu'un cas particulier : • dans le cas où c'est un vecteur ligne (p.ex. une série), la variable *var* sera un scalaire recevant à chaque itération l'élément suivant de ce vecteur • si c'est un vecteur colonne, la variable *var* sera aussi un vecteur colonne qui recevra en une fois toutes les valeurs de ce vecteur, et le bloc d'*instructions* ne sera ainsi exécuté qu'une seule fois puis on sortira directement de la boucle ! Si l'on a *tableau* à **3 dimensions**, `for` examinera d'abord les colonnes de la 1ère "couche" du tableau, puis celles de la seconde "couche", etc... Ex: • `for n=10:-2:0 , n^2, end` affiche successivement les valeurs 100 64 36 16 4 et 0 • `for n=[1 5 2;4 4 4] , n, end` affiche successivement les colonnes [1;4] [5;4] et [2;4]
Boucle **parfor...end** `parfor (`*var*`=`*deb*`:`*fin* `{,` *max_threads*`})` *instructions...* `end`	Variante simplifiée mais **parallélisée** de la boucle `for...end` : • le bloc d'*instructions* est exécuté en parallèle par différents threads (au maximum *max_threads*) et l'ordre d'itération n'est pas garanti • *deb* et *fin* doivent être des entiers, et *fin* > *deb*
Boucle **while...end** ("tant que" la condition n'est pas fausse) `while` *expression_logique* *instructions...* `end`	Si **tous** les éléments de l'objet résultant de l'*expression_logique* (qui n'est donc pas nécessairement scalaire mais peut être un tableau de dimension quelconque) sont Vrais (c'est-à-dire différents de "0"), l'ensemble d'*instructions* spécifiées est exécuté, puis l'on reboucle sur le test. Si **un ou plusieurs** éléments sont Faux (c'est-à-dire égaux à "0"), on saute aux instructions situées après le `end` . On modifie en général, dans la boucle, des variables constituant l'*expression_logique*, sinon la boucle s'exécutera sans fin (et on ne peut dans ce cas en sortir qu'avec un ctrl-C !) Ex: • `n=1 ; while (n^2 < 100) , disp(n^2) , n=n+1 ; end` affiche les valeurs n^2 depuis n=1 et tant que n^2 est inférieur à 100, donc affiche les valeurs 1 4 9 16 25 36 49 64 81 puis s'arrête

Boucle **do...until**
("jusqu'à ce que" une condition se vérifie)

```
do
   instructions...
until expression_logique
```

Spécifique à **Octave**, cette structure de contrôle classique permet d'exécuter des *instructions* en boucle jusqu'à ce qu'une *expression_logique* soit vraie.

La différence essentielle par rapport à la boucle `while` est que l'on vérifie la condition après avoir une fois (au moins) exécuté le bloc d'*instructions*.
Pour atteindre le même but sous **MATLAB**, on pourrait faire une boucle sans fin de type `while true instructions... end` à l'intérieur de laquelle on sortirait par un test et l'instruction `break` (voir plus bas).

Test **if...{{elseif...}else...}end**
("si, sinon si, sinon")

```
  if expression_logique_1
     instructions_1...
{ elseif expression_logique_i
     instructions_i...       }
{ else
     autres_instructions... }
  end
```

Si **tous** les éléments résultant de l'*expression_logique_1* (qui n'est donc pas nécessairement un scalaire mais peut être un tableau) sont Vrais (c'est-à-dire différents de "0"), le bloc *instructions_1* est exécuté, puis on saute directement au `end`.
Sinon (si un ou plusieurs éléments de *expression_logique_1* sont Faux, c'est-à-dire égaux à "0"), MATLAB/Octave examine les éventuelles clauses `elseif` (de d'autres *expression_logique_i*, avec exécution le cas échéant du bloc *instructions_i* correspondant, puis saute à `end`).
Si aucune expression testée n'a été satisfaite, on saute à l'éventuelle instruction `else` pour exécuter le bloc *autres_instructions*.
Noter que les blocs `elseif` ainsi que le bloc `else` sont donc facultatifs.

Ex:

- Soit le bloc d'instructions `if n==1, disp('un'), elseif n==2, disp('deux'), else, disp('autre'), end`. Si l'on affecte `n=1`, l'exécution de ces instructions affiche "un", si l'on affecte `n=2` il affiche "deux", et si "n" est autre que 1 ou 2 il affiche "autre"

Construction **switch...case... {otherwise...}end**

```
switch variable
   case val1
      instructions_a...
   case {val2, val3 ...}
      instructions_b...
   otherwise
      autres_instructions...
end
```

Cette construction réalise ce qui suit : si la *variable* spécifiée est égale à la valeur *val1*, seul le bloc de *instructions_a* spécifié est exécuté ; si elle est égale à la valeur *val2* ou *val3*, seul le bloc de *instructions_b* spécifié est exécuté. Et ainsi de suite... Si elle n'est égale à rien de tout ça, c'est le bloc des *autres_instructions* spécifiées qui est exécuté.
Important : notez bien que si, dans une clause `case`, vous définissez plusieurs valeurs *val* possibles, il faut que celles-ci soient définies sous la forme de **tableau cellulaire**, c'est-à-dire entre caractères `{ }`.
Contrairement au "switch" du langage C, il n'est pas nécessaire de définir des `break` à la fin de chaque bloc
On peut avoir autant de blocs `case` que l'on veut, et la partie `otherwise` est facultative.
En lieu et place de *variable* et de *val1*, *val2*... on peut bien entendu aussi mettre des expressions.

Ex:

- Tester d'une réponse `rep` contenant : Non, non, No, no, N, n, Oui, oui, O, o, Yes, yes, Y, y :

```
switch lower(rep(1))
    case 'n'
        disp('non')
    case { 'o' 'y' }
        disp('oui')
    otherwise
        disp('reponse incorrecte')
end
```

- L'exemple précédent réalisé avec if-elseif-else pourrait être ainsi reprogrammé avec une structure switch-case : `switch n, case 1, disp('un'), case 2, disp('deux'), otherwise, disp('autre'), end`

Construction **try...catch...end**

```
try
   instructions_1...
catch
```

Cette construction sert à implémenter des **traitements d'erreur**.

Les instructions comprises entre `try` et `catch` (bloc *instructions_1*) sont exécutées jusqu'à ce qu'une erreur se produise, auquel cas MATLAB/Octave passe automatiquement à l'exécution des instructions comprises entre

`instructions_2...` `end` `autres_instructions...`	`catch` et `end` (bloc *instructions_2*). Le message d'erreur ne s'affiche pas mais peut être récupéré avec la commande `lasterr`. Si le premier bloc de *instructions_1* s'exécute absolument sans erreur, le second bloc de *instructions_2* n'est pas exécuté, et le déroulement se poursuit avec *autres_instructions*
Sortie anticipée d'une boucle `break`	A l'intérieur d'une boucle `for`, `while` ou `do`, cette instruction permet, par exemple suite à un test, de sortir **prématurément** de la boucle et de poursuivre l'exécution des instructions situées après la boucle. Si 2 boucles sont imbriquées l'une dans l'autre, un `break` placé dans la boucle interne sort de celle-ci et continue l'exécution dans la boucle externe. A ne **pas confondre** avec `return` (voir plus bas) qui sort d'une fonction, respectivement interrompt un script ! Ex: sortie d'une boucle `for` : `for k=1:100` `  k2=k^2;` `  fprintf('carré de %3d = %5d \n',k,k2)` `  if k2 > 200  break , end  % sortir boucle lorsque k^2 > 200` `end` `fprintf('\nsorti de la boucle à k = %d\n',k)` Ex: sortie d'une boucle `while` : `k=1;` `while true    % à priori boucle sans fin !` `  fprintf('carré de %3d = %5d \n', k, k^2)` `  if k >= 10  break , else k=k+1; end` `  % ici on sort lorsque k > 10` `end`
Sauter le reste des instructions d'une boucle et **continuer d'itérer** `continue`	A l'intérieur d'une boucle (`for` ou `while`), cette instruction permet donc, par exemple suite à un test, de sauter le reste des instructions de la boucle et passer à l'**itération suivante** de la **même boucle**. Ex: `start=1; stop=100; fact=8;` `fprintf('Nb entre %u et %u divisibles par %u : ',start,stop,fact)` `for k=start:1:stop` `  if rem(k,fact) ~= 0` `  continue` `  end` `  fprintf('%u, ', k)` `end` `disp('fin')` Le code ci-dessus affiche: "Nb entre 1 et 100 divisibles par 8 : 8, 16, 24, 32, 40, 48, 56, 64, 72, 80, 88, 96, fin"
`double(var)`	Cette instruction permet de convertir en double précision la variable *var* qui, dans certaines constructions `for`, `while`, `if`, peut n'être qu'en simple précision

Les structures de contrôle sont donc des éléments de langage extrêmement utiles. Mais dans MATLAB/Octave, il faut "penser instructions matricielles" (on dit aussi parfois "vectoriser" son algorithme) avant d'utiliser à toutes les sauces ces structures de contrôle qui, du fait que MATLAB est un langage interprété, sont beaucoup moins rapides que les opérateurs et fonctions matriciels de base !

Ex: l'instruction `y=sqrt(1:100000);` est beaucoup plus efficace/rapide que la boucle `for n=1:100000, y(n)=sqrt(n); end` (bien que, dans les 2 cas, ce soit un vecteur de 100'000 éléments qui est créé contenant les valeurs de la racine de 1 jusqu'à la racine de 100'000). Testez vous-même !

7.6 Autres commandes et fonctions utiles en programmation

Nous énumérons encore ici quelques commandes ou fonctions supplémentaires qui peuvent être utiles dans la **programmation** de scripts ou de fonctions.

`return`
Termine l'exécution de la fonction ou du script. Un script ou une fonction peut renfermer plusieurs `return` (sorties contrôlées par des structures de contrôle...). Une autre façon de sortir proprement en cas d'erreur est d'utiliser la fonction `error` (voir plus haut).
On ne sortira jamais avec `exit` ou `quit` qui non seulement terminerait le script ou la fonction mais fermerait aussi la session MATLAB/Octave !

var= `nargin`
A l'intérieur d'une fonction, retourne le nombre d'arguments d'entrée passés lors de l'appel à cette fonction. Permet par exemple de donner des valeurs par défaut aux paramètres d'entrée manquant.
Utile sous **Octave** pour tester si le nombre de paramètres passés par l'utilisateur à la fonction est bien celui attendu par la fonction (ce test n'étant pas nécessaire sous **MATLAB** ou le non respect de cette condition est automatiquement détecté).
Voir aussi la fonction `nargchk` qui permet aussi l'implémentation simple d'un message d'erreur.

Ex: voir ci-après

`varargin`
A l'intérieur d'une fonction, tableau cellulaire permettant de récupérer un nombre d'arguments quelconque passé à la fonction

Ex: soit la fonction `test_vararg.m` suivante :

```
function []=test_vararg(varargin)
fprintf('Nombre d''arguments passes a la fonction : %d \n',nargin)
for no_argin=1:nargin
  fprintf('- argument %d:\n', no_argin)
  disp( varargin{no_argin} )
end
```

si on l'invoque avec `test_vararg(111,[22 33;44 55],'hello !',{'ca va ?'})` elle retourne :

```
Nombre d'arguments passes a la fonction : 4
- argument 1:
   111
- argument 2:
   22   33
   44   55
- argument 3:
   hello !
- argument 4:
   { [1,1] = ca va ? }
```

string= `inputname(k)`
A l'intérieur d'une fonction, retourne le nom de variable du *k*-ème argument passé à la fonction

var= `nargout`
A l'intérieur d'une fonction, retourne le nombre de variables de sortie auxquelles la fonction est affectée lors de l'appel. Permet par exemple d'éviter de calculer les paramètres de sortie manquants....
Voir aussi la fonction `nargoutchk` qui permet aussi l'implémentation simple d'un message d'erreur.

Ex: A l'intérieur d'une fonction-utilisateur `mafonction` :
- lorsqu'on l'appelle avec `mafonction(...)` : `nargout` vaudra 0
- lorsqu'on l'appelle avec `out1 = mafonction(...)` : `nargout` vaudra 1
- lorsqu'on l'appelle avec `[out1 out2] = mafonction(...)` : `nargout` vaudra 2, etc...

string= `mfilename`
A l'intérieur d'une fonction ou d'un script, retourne le nom du M-file de cette fonction ou script, sans son extension `.m`

Ex d'instruction dans une fonction : `warning(['La fonction ' mfilename ' attend au moins un argument'])`

`global` *`variable(s)`*

Définit la(les) *variable(s)* spécifiée(s) comme **globale(s)**. Cela peut être utile lorsque l'on veut partager des données entre le workspace et certaines fonctions sans devoir passer ces données en paramètre lors de l'appel à ces fonctions. Il est alors nécessaire de déclarer ces variables globales, avant de les utiliser, à la fois dans le workspace et à l'intérieur des fonctions.
Une bonne habitude serait d'identifier clairement les variables globales de fonctions, par exemple en leur donnant un nom en caractères majuscules.

Ex : la fonction **`fct1.m`** ci-dessous mémorise (et affiche) le nombre de fois qu'elle a été appelée :

```
function []=fct1()
 global  COMPTEUR
COMPTEUR=COMPTEUR+1;
fprintf('fonction appelee %04u fois \n',COMPTEUR)
return
```

Pour tester cela, il faut passer les instructions suivantes dans la fenêtre de commande MATLAB/Octave :

```
 global  COMPTEUR   % cela déclare le compteur également global dans le workspace
COMPTEUR = 0 ;    % initialisation du compteur
fct1              % => cela affiche "fonction appelee   1 fois"
fct1              % => cela affiche "fonction appelee   2 fois"
```

`persistent` *`variable(s)`*

Utilisable dans les fonctions seulement, cette déclaration définit la(les) *variable(s)* spécifiée(s) comme **statique(s)**, c'est-à-dire conservant de façon interne leurs dernières valeurs entre chaque appel à la fonction. Ces variables ne sont cependant pas visibles en-dehors de la fonction (par opposition aux variables globales).

Ex : la fonction **`fct2.m`** ci-dessous mémorise (et affiche) le nombre de fois qu'elle a été appelée. Contrairement à l'exemple de la fonction **`fct1.m`** ci-dessus, la variable *compteur* n'a **pas** à être déclarée dans la session principale (ou dans le script depuis lequel on appelle cette fonction), et le *compteur* doit ici être initialisé **dans** la fonction.

```
function []=fct2()
 persistent  compteur
% au premier appel, après cette déclaration persistent compteur existe et vaut []
if isempty(compteur)
  compteur=0 ;
end
compteur=compteur+1 ;
fprintf('fonction appelee %04u fois \n',compteur)
return
```

Pour tester cela, il suffit de passer les instructions suivantes dans la fenêtre de commande MATLAB/Octave :

```
fct2              % => cela affiche "fonction appelee   1 fois"
fct2              % => cela affiche "fonction appelee   2 fois"
```

`eval('`*`expression1`***`', {'`***`expression2`***`'})`**

Évalue et **exécute** l'*expression1* MATLAB/Octave spécifiée. En cas d'échec, évalue l'*expression2*.

Ex: le petit script suivant permet de grapher n'importe quelle fonction y=f(x) définie interactivement par l'utilisateur :

```
fonction = input('Quelle fonction y=fct(x) voulez-vous grapher : ','s');
min_max  = input('Indiquez [xmin xmax] : ');
x = linspace(min_max(1),min_max(2),100);
eval(fonction,'error(''fonction incorrecte'')');
plot(x,y)
```

`class(`*`objet`***`)`**

Retourne la "classe" de *objet* (double, struct, cell, char).

`typeinfo(`*`objet`***`)`**

Sous Octave seulement, retourne le "type" de *objet* (scalar, range, matrix, struct, cell, list, bool, sq_string, char matrix, file...).

7.7 Scripts (programmes), mode batch

7.7.1 Principes de base relatifs aux scripts

Un "**script**" ou "**programme**" MATLAB/Octave n'est rien d'autre qu'une suite de commandes MATLAB/Octave valides (par exemple un "algorithme" exprimé en langage MATLAB/Octave) sauvegardées dans un **M-file**, c'est-à-dire un fichier avec l'extension `.m`.

Par opposition aux "fonctions" (voir chapitre suivant), les scripts sont invoqués par l'utilisateur sans passer d'arguments, car ils **opèrent directement dans le workspace** principal. Un script peut donc lire et modifier des variables préalablement définies (que ce soit interactivement ou via un autre script), ainsi que créer de nouvelles variables qui seront accessibles dans le workspace (et à d'autres scripts) une fois le script exécuté.

Il est important de noter qu'il n'est **pas possible** de définir de **fonctions à l'intérieur d'un script** (même si l'on ne souhaite utiliser celles-ci que par le script).

Il est possible (et vivement conseillé) de **documenter** le fonctionnement du script vis-à-vis du système d'**aide en ligne** `help` de MATLAB/Octave. Il suffit, pour cela, de définir, au *tout début* du script, des lignes de commentaire (lignes débutant par le caractère `%`). La commande `help` *M-file* affichera alors automatiquement le 1er bloc de lignes de commentaire contiguës du M-file. On veillera à ce que la toute première ligne de commentaire (appelée "H1-line") indique le nom du script (en majuscules) et précise brièvement ce que fait le script, étant donné que c'est cette ligne qui est affichée lorsque l'on fait une recherche de type `lookfor` *mot-clé*.

Pour **exécuter un script**, on peut utiliser l'une des méthodes suivantes, selon que l'on soit dans l'éditeur intégré, dans la console MATALAB/Octave ou depuis un autre script :

a) Run, Save File and Run ou F5
b) *script* enter
c) `run('{`*chemin*`/}`*script*`{.m}')` ou `run {`*chemin*`/}`*script*`{.m}`
d) `source('{`*chemin*`/}`*script*`.m')` ou `source {`*chemin*`/}`*script*`.m`

a) Lancement de l'exécution depuis l'éditeur intégré MATLAB ou Octave GUI
b) Le *script* doit obligatoirement se trouver dans le répertoire courant ou dans le *path* de recherche MATLAB/Octave (voir chapitre "**Environnement**"). Il ne faut pas spécifier l'extension `.m` du *script*
c) Cette forme permet d'exécuter un *script* situé en dehors du répertoire courant en indiquant le *chemin* d'accès (absolu ou relatif). On peut omettre ou spécifier l'extension `.m` du script
d) Cette forme est propre à Octave. Dans ce cas l'extension `.m` doit obligatoirement être indiquée

En phase de **debugging**, on peut activer l'affichage des commandes exécutées par le script en passant la commande `echo on` avant de lancer le script, puis désactiver ce "traçage" avec `echo off` une fois le script terminé.

Exemple de script: Le petit programme ci-dessous réalise la somme et le produit de 2 nombres, vecteurs ou matrices (de même dimension) demandés interactivement. Notez bien la 1ère ligne de commentaire (H1-line) et les 2 lignes qui suivent fournissant le texte pour l'aide en-ligne. On exécute ce programme en frappant `somprod` (puis répondre aux questions interactives...), ou l'on obtient de l'aide sur ce script en frappant `help somprod`.

```
%SOMPROD Script réalisant la somme et le produit de 2 nombres, vecteurs ou matrices
%
% Ce script est interactif, c'est-à-dire qu'il demande interactivement les 2 nombres,
% vecteurs ou matrices dont il faut faire la somme et le produit (élément par élément)

V1=input('Entrer 1er nombre (ou expression, vecteur ou matrice) : ') ;
V2=input('Entrer 2e  nombre (ou expression, vecteur ou matrice) : ') ;

if ~ isequal(size(V1),size(V2))
  error('les 2 arguments n''ont pas la meme dimension')
end

%{
  1ère façon d'afficher les résultats (la plus propre au niveau affichage,
  mais ne convenant que si V1 et V2 sont des scalaires) :
      fprintf('Somme = %6.1f  Produit = %6.1f \n', V1+V2, V1.*V2)

  2ème façon d'afficher les résultats :
      Somme = V1+V2
      Produit = V1.*V2
%}
```

```
% 3ème façon (basique) d'afficher les résultats
    disp('Somme =') ,   disp(V1+V2)
    disp('Produit =') , disp(V1.*V2)

return  % Sortie du script (instruction ici pas vraiment nécessaire,
        %                   vu qu'on a atteint la fin du script !)
```

7.7.2 Exécuter un script en mode batch

Pour autant qu'il ne soit pas interactif, on peut exécuter un **script** depuis un **shell** (dans fenêtre de commande du système d'exploitation) ou en mode **batch** (p.ex. environnement GRID), c'est-à-dire sans devoir démarrer l'interface-utilisateur MATLAB/Octave, de la façon décrite ici.

Avec **MATLAB** :

- En premier lieu, il est important que le `script.m` s'achève sur une instruction `quit`, sinon la fenêtre MATLAB (minimisée dans la barre de tâches sous Windows) ne se refermera pas
- Passer la commande (sous Windows depuis une fenêtre "invite de commande", sous Linux depuis une fenêtre shell) :
  `matlab -nodesktop -nosplash -nodisplay -r script { -logfile fichier_resultat }`
  où :
  - sous Windows, il faudra faire précéder la commande `matlab` du `path` (chemin d'accès à l'exécutable MATLAB, p.ex. `C:\Program Files (x86)\MATLAB\R2011b\bin\` sous Windows 7)
  - à la place de `-logfile fichier_resultat` on peut aussi utiliser `> fichier_resultat`
  - le fichier de sortie `fichier_resultat` sera créé (en mode écrasement s'il préexiste)

- Sachez finalement qu'il est possible d'utiliser interactivement MATLAB en mode commande dans une fenêtre terminal (shell) et sans interface graphique (intéressant si vous utilisez MATLAB à distance sur un serveur Linux) ; il faut pour cela démarrer MATLAB avec la commande : `matlab -nodesktop -nosplash`

Avec **Octave** :

- Contrairement à MATLAB, il n'est ici pas nécessaire que le `script.m` s'achève par une instruction `quit`
- Vous avez ensuite les possibilités suivantes :

  - Depuis une fenêtre de commande (fenêtre "invite de commande" sous Windows, shell sous Linux), frapper :
    `octave --silent --no-window-system script.m { > fichier_resultat }`

  - En outre, sous **Linux** ou **Mac OS X**, vous pouvez aussi procéder ainsi :
    - faire débuter le script par la ligne: `#!/usr/bin/octave --silent --no-window-system`
    - puis mettre le script en mode execute avec la commande: `chmod u+x script.m`
    - puis lancer le script avec: `./script.m { > fichier_resultat }`

  Notez que, dans ces commandes :

  - avec `> fichier_resultat`, les résultats du script sont redirigés dans le *fichier_resultat* spécifié et non pas affichés dans la fenêtre de commande
  - `--silent` (ou `-q`) : n'affiche pas les messages de démarrage de Octave
  - `--no-window-system` (ou `-W`) : désactive le système de fenêtrage (i.e. X11 sous Linux) ; les graphiques apparaissent alors dans la fenêtre de commande (simulés par des caractères), mais il reste possible d'utiliser la commande `saveas` pour les sauvegarder sous forme de fichiers-image
  - en ajoutant encore l'option `--norc` (ou `-f`) ou `--no-init-file`), on désactiverait l'exécution des prologues (utilisateurs `.octaverc`, et système)
  - vous pourriez donc aussi faire débuter votre script par la ligne `#!/usr/bin/octave -qWf`

  Ex: le script ci-dessous produit un fichier de données "job_results.log" et un graphique "job_graph.png" :

```
% Exemple de script MATLAB/Octave produisant des données et un graphique
% que l'on peut lancer en batch avec :
%   matlab -nosplash -nodisplay -nodesktop -r job_matlab > job_results.log
%   octave --silent --no-window-system --norc job_matlab.m > job_results.log

data = randn(100,3)
fig = figure;
plotmatrix(data, 'g+');
saveas(fig, 'job_graph.png', 'png')
quit
```

- A l'intérieur de votre script, vous pouvez récupérer sur un vecteur cellulaire avec la fonction `argv()` les différents

arguments passés à la commande **`octave`**

En outre avec **Octave**, si vous ne désirez exécuter en "batch" que quelques *commandes* sans faire de script, vous pouvez procéder ainsi :

- Depuis une fenêtre de commande ou un shell, frapper: **`octave -qf --eval "`***`commandes...`***`" { > `***`fichier_resultat`***`.txt }`**

  Ex: la commande **`octave -qf --eval "disp('Hello !'), arguments=argv(), a=12; douze_au_carre=12^2, disp('Bye...')"`** affiche :

```
Hello !
arguments =
{
  [1,1] = -qf
  [2,1] = --eval
  [3,1] = disp('Hello'), arguments=argv(), a=12; douze_au_carre=12^2, disp('Bye...')
}
douze_au_carre =  144
Bye...
```

7.7.3 Tester si un script s'exécute sous MATLAB ou sous Octave

Étant donné les différences qui peuvent exister entre MATLAB et Octave (p.ex. fonctions implémentées différemment ou non disponibles...), si l'on souhaite réaliser des **scripts portables** (i.e. qui tournent à la fois sous MATLAB et Octave, ce qui est conseillé !) on peut implémenter du **code conditionnel** relatif à chacun de ces environnements en réalisant, par exemple, un test via une fonction built-in appropriée.

Ex: on test ici l'existence de la fonction built-in **`OCTAVE_VERSION`** (n'existant que sous Octave) :

```
if ~ exist('OCTAVE_VERSION')  % MATLAB
  % ici instruction(s) pour MATLAB
else                          % Octave
  % ici instruction(s) équivalente(s) pour Octave
end
```

7.8 Fonctions, P-Code

7.8.1 Principes de base relatifs aux fonctions

Également programmées sous forme de M-files, les "**fonctions**" MATLAB se distinguent des "scripts" par leur mode d'invocation qui est fondamentalement différent :

```
variable(s)_de_sortie = nom_fonction(argument(s)_d_entree, ...)
```

- On invoque donc une fonction en l'appelant par son nom et en lui passant, **entre parenthèses**, des **paramètres d'entrée** (valeurs, noms de variables, expressions) ;
  certaines fonctions peuvent ne pas avoir d'argument, par exemple : `beep()`
- La fonction retourne en général une(des) valeur(s) de **sortie** que l'on récupère alors sur la(les) variable(s) à laquelle (auxquelles) la fonction est affectée lors de l'appel

Le mécanisme de passage des **paramètres** à la fonction se fait "**par valeur**" (c'est-à-dire copie) et non pas "par référence". La fonction ne peut donc pas modifier les variables d'entrée au niveau du script appelant ou du workspace principal.

Les **variables** créées **à l'intérieur** de la fonction sont dites "**locales**" car elles sont, par défaut, inaccessibles en dehors de la fonction (que ce soit dans le workspace principal ou dans d'autres fonctions ou scripts). Chaque fonction travaille donc dans son **propre workspace** local.

- Si l'on tient cependant à ce que certaines variables de la fonction soient visibles et accessibles à l'extérieur de celle-ci, on peut les rendre "**globales**" en les définissant comme telles dans la fonction, *avant* qu'elles ne soient utilisées, par une déclaration `global variable(s)` (voir **plus haut**). Il faudra aussi faire une telle déclaration dans le workspace principal (et avant d'utiliser la fonction !) si l'on veut pouvoir accéder à ces variables dans le workspace principal ou ultérieurement dans d'autres scripts !
- Une autre possibilité consiste à déclarer certaines variables comme "**statiques**" avec la déclaration `persistent` (voir aussi **plus haut**) au cas où l'on désire, à chaque appel à la fonction, retrouver les variables internes dans l'état où elles ont été laissées lors de l'appel précédent. Cela ne devrait pas être utilisé pour les fonctions qui seront utilisées de façon récursive !

Contrairement aux scripts dans lesquels on n'a pas le droit de définir de fonctions internes, il est possible, **dans un M-file de fonction**, de définir **plusieurs fonctions** à la suite les unes des autres. Cependant seule la première fonction du M-file, appelée **fonction principale** (*main function*), sera accessible de l'extérieur. Les autres, appelées **fonctions locales** (ou *subfunctions*), ne pourront être appelées que par la fonction principale ou les autres fonctions locales du M-file.

Il est finalement possible de définir des fonctions à l'intérieur du corps d'une fonction. Ces **fonctions imbriquées** (*nested function*) ne pourront cependant être invoquées que depuis la fonction dans laquelle elles sont définies.

Toute fonction débute par une ligne de **déclaration de fonction** qui définit son `nom_fonction` et ses arguments `args_entree` et `args_sortie` (séparés par des virgules), selon la syntaxe :

```
function [argument(s)_de_sortie, ...] = nom_fonction(argument(s)_d_entree, ...)
```

(les crochets ne sont pas obligatoires s'il n'y a qu'un *arg_sortie*)

Octave affiche un warning si le **nom de la fonction principale** (tel que défini dans la 1ère déclaration de fonction) est différent du **nom du M-file**. Sous MATLAB, le fait que les 2 noms soient identiques n'est pas obligatoire mais fait partie des règles de bonne pratique.

Se succèdent donc, dans le code d'une fonction (et dans cet ordre) :

a. déclaration de la fonction principale (ligne `function...` décrite ci-dessus)
b. lignes de commentaires (commençant par le caractère `%`) décrivant la fonction pour le système d'aide en-ligne, à savoir :
   • la "**H1-line**" précisant le nom de la fonction et indiquant des *mots-clé* (ligne qui sera retournée par la commande `lookfor mot-clé`)
   • les lignes du **texte d'aide** (qui seront affichées par la commande `help nom_fonction`)
c. déclarations d'éventuelles variables **globale** ou **statique**
d. code proprement dit de la fonction (qui va affecter les variables *argument(s)_de_sortie*)
e. éventuelle(s) instruction(s) `return` définissant de(s) point(s) de sortie de la fonction
f. instruction `end` signalant la fin de la fonction

L'instruction `end` peut être omise, sauf lorsque l'on définit plusieurs fonctions dans un même M-file. Les templates de fonctions Octave se terminent par `endfunction`, mais nous vous suggérons d'éliminer cette instruction afin que vos fonctions soient compatibles à la fois pour MATLAB et Octave.

Exemple de fonction: On présente, ci-dessous, deux façons de réaliser une petite fonction retournant le produit et la somme de 2 nombres, vecteurs ou matrices. Dans les deux cas, le M-file doit être nommé **`fsomprod.m`** (c'est-à-dire identique au nom de la fonction). On peut accéder à l'aide de la fonction avec **`help fsomprod`**, et on affiche la première ligne d'aide en effectuant par exemple une recherche **`lookfor produit`**. Dans ces 2 exemples, mis à part les arrêts en cas d'erreurs (instructions **`error`**), la sortie s'effectue à la fin du code mais aurait pu intervenir ailleurs (instructions **`return`**) !

Fonction	Appel de la fonction
`function [resultat]=fsomprod(a,b)` `%FSOMPROD somme et produit de 2 nombres ou vecteurs-ligne` `%   Usage: R=FSOMPROD(V1,V2)` `%      Retourne matrice R contenant: en 1ère ligne la` `%      somme de V1 et V2, en seconde ligne le produit de` `%      V1 et V2 élément par élément` `if nargin~=2` `  error('cette fonction attend 2 arguments')` `end` `sa=size(a); sb=size(b);` `if ~ isequal(sa,sb)` `  error('les 2 arguments n'ont pas la même dimension')` `end` `if sa(1)~=1 || sb(1)~=1` `  error('les arg. doivent être scalaires ou vecteurs-ligne')` `end` `resultat(1,:)=a+b;   % 1ère ligne de la matrice-résultat` `resultat(2,:)=a.*b;  % 2ème ligne de la matrice-résultat,` `                     %      produit élément par élément !` `return     % sortie de la fonction (instruction ici pas` `           % nécessaire vu qu'on a atteint fin fonction)` `end  % pas nécessaire si le fichier ne contient que cette fct`	**Remarque** : cette façon de retourner le résultat (sur une seule variable) ne permet pas de passer à cette fonction des matrices. **`r=fsomprod(4,5)`** retourne la vecteur-colonne r=[9 ; 20] **`r=fsomprod([2 3 1],[1 2 2])`** retourne la matrice r=[3 5 3 ; 2 6 2]
`function [somme,produit]=fsomprod(a,b)` `%FSOMPROD somme et produit de 2 nombres, vecteurs ou matrices` `%   Usage: [S,P]=FSOMPROD(V1,V2)` `%      Retourne matrice S contenant la somme de V1 et V2,` `%      et matrice P contenant le produit de V1 et V2` `%      élément par élément` `if nargin~=2` `  error('cette fonction attend 2 arguments')` `end` `if ~ isequal(size(a),size(b))` `  error('les 2 arg. n'ont pas la même dimension')` `end` `somme=a+b;` `produit=a.*b;  % produit élément par élément !` `return         % sortie de la fonction (instruction ici pas` `               % nécessaire vu qu'on a atteint fin fonction)` `end  % pas nécessaire si le fichier ne contient que cette fct`	**Remarque** : cette façon de retourner le résultat (sur deux variable) rend possible le passage de matrices à cette fonction ! **`[s,p]=fsomprod(4,5)`** retourne les scalaires s=9 et p=20 **`[s,p]=fsomprod([2 3;1 2],[1 2; 3 3])`** retourne les matrices s=[3 5 ; 4 5] et p=[2 6 ; 3 6]

7.8.2 P-Code

Lorsque du code MATLAB est exécuté, il est automatiquement interprété et traduit ("**parsing**") dans un **langage de plus bas niveau** qui s'appelle le **P-Code** (pseudo-code). Sous **MATLAB** seulement, s'agissant d'une fonction souvent utilisée, on peut éviter que cette "passe de traduction" soit effectuée lors de chaque appel en **sauvegardant le P-Code sur un fichier** avec la commande **`pcode`** *`nom_fonction`*. Un fichier de nom *`nom_fonction.p`* est alors déposé dans le répertoire courant (ou dans le dossier où se trouve le M-file si l'on ajoute à la commande **`pcode`** le paramètre **`-inplace`**), et à chaque appel la fonction pourra être directement exécutée sur la base du P-Code de ce fichier sans traduction préalable, ce qui peut apporter des gains de performance.

Le mécanisme de conversion d'une fonction ou d'un script en P-Code offre également la possibilité de distribuer ceux-ci à d'autres personnes sous forme binaire en conservant la **maîtrise du code source**.

7.9 Entrées-sorties formatées, manipulation de fichiers

Lorsqu'il s'agit de charger, dans MATLAB/Octave, une matrice à partir de données externes stockées dans un fichier-texte, les commandes `load -ascii` et `dlmread` / `dlmwrite`, présentées au chapitre "**Workspace MATLAB/Octave**", sont suffisantes. Mais lorsque les données à **importer** sont dans un format plus complexe ou qu'il s'agit d'importer du texte ou d'**exporter** des données vers d'autres logiciels, les fonctions présentées ci-dessous s'avèrent nécessaires.

7.9.1 Vue d'ensemble des fonctions d'entrée/sortie de base

Le tableau ci-dessous donne une **vision synthétique** des principales fonctions d'entrée/sortie (présentées en détail dans les chapitres qui suivent). Notez que :

- le caractère `s` débutant le nom de certaines fonctions signifie "**s**tring", c'est-à-dire que l'on manipule des chaînes
- le caractère `f` débutant le nom de certaines fonctions signifie "**f**ile", c'est-à-dire que l'on manipule des fichiers
- le caractère `f` terminant le nom de certaines fonctions signifie "**f**ormaté"

	Écriture	**Lecture**
Interactivement	Écriture à l'écran (*sortie standard*) • non formaté: `disp(variable|chaîne)` (avec un seul paramètre !) • formaté: `fprintf(format, variable(s))` (ou `printf` ...)	Lecture au clavier (*entrée standard*) • non formaté: *var* = `input(prompt {, 's'} )` • formaté: *var* = `scanf(format)`
Sur **chaîne** de caractères	• *string* = `sprintf(format, variable(s))` • autres fonctions : `mat2str` ...	• *var*|*mat* = `sscanf(string, format {,size})` • autres fonctions : `strread`, `textscan` ...
Sur **fichier** texte	• `fprintf(file_id, format, variable(s)... )` • autres fonctions : `save -ascii`, `dlmwrite` ...	• *var* = `fscanf(file_id, format {,size})` • *line* = `fgetl(file_id)` • *string* = `fgets(file_id {,nb_car})` • autres fonctions : `load(fichier)`, `textread`, `textscan`, `fileread`, `dlmread` ...
Via **internet** (protocoles HTTTP, FTP ou FILE)	• *string* = `urlread(url, method, param)` • `urlwrite(url, fichier, method, param)` (pour `url` de type HTTP, `method` : `'get'` ou `'post'`)	• *string* = `urlread(url)` • `urlwrite(url, fichier)`
Sur **fichier binaire**	• autres fonctions : `fwrite`, `xlswrite` ...	• autres fonctions : `fread`, `xlsread` ...

7.9.2 Formats de lecture/écriture

Les différentes fonctions de **lecture/écriture** sous forme **texte** présentées ci-dessous font appel à des "**formats**" (parfois appelés "**templates**" dans la documentation). Le but de ceux-ci est de :

- décrire la manière selon laquelle il faut interpréter ce que l'on **lit** (s'agit-il d'un nombre, d'une chaîne de caractère...)
- dlfinir sous quelle forme il faut **écrire** les données (pour un nombre: combien de chiffres avant/après la virgule ; pour une chaîne: type de justification...)

Les formats MATLAB/Octave utilisent un sous-ensemble des conventions et spécifications de formats du **langage C**.

Les formats sont des **chaînes** de caractères se composant de "**spécifications de conversion**" dont la syntaxe est décrite dans le tableau ci-dessous.

ATTENTION : dans un format de **lecture**, on ne préfixera en principe pas les "spécifications de conversion" de nombres (`u` `d` `i` `o` `x` `X` `f` `e` `E` `g` `G`) par des valeurs *n* (taille du champ) et *m* (nombre de décimales), car le comportement de **MATLAB** et de **Octave** peut alors conduire à des résultats différents (découpage avec MATLAB, et aucun effet sous Octave).

Ex: `sscanf('560001','%4f')` retourne : M sous MATLAB le vecteur [5600 ; 1] , et O sous Octave la valeur 560001

Spécifications	Description
espace	• En lecture: les caractères espace dans un format n'ont aucune signification (i.e. sont ignorés) • En écriture: les caractères espace sont écrits dans la chaîne résultante
`%u` `%nu`	Correspond à un nombre **entier positif** (non signé) • En lecture: Ex: `sscanf('100 -5','%u')` => 100 et 4.295e+09 (!) • En écriture: si *n* est spécifié, le nombre sera justifié à droite dans un champ de *n* car. au min. Ex: `sprintf('%5u ', 100, -5)` => M ' 100 -5.000000e+000' et O ' 100 -5'
`%d` `%i` `%nd` `%ni`	Correspond à un nombre **entier positif ou négatif** • En lecture: Ex: `sscanf('100 -5','%d')` => 100 et -5 • En écriture: si *n* est spécifié, le nombre sera justifié à droite dans un champ de *n* car. au min. Ex: `sprintf('%d %03d ', 100, -5)` => '100 -05'
`%o` `%no`	Correspond à un nombre **entier positif** en base **octale** • En lecture: Ex: `sscanf('100 -5','%o')` => 64 et 4.295e+09 (!) • En écriture: si *n* est spécifié, le nombre sera justifié à droite dans un champ de *n* car. au min. Ex: `sprintf('%04o ', 64, -5)` => M '0100 -5.000000e+000' et O '0100 -005'
`%x` `%X` `%nx` `%nX`	Correspond à un nombre **entier positif** en base **hexadécimale** • En lecture: Ex: `sscanf('100 -5','%x')` => 256 et 4.295e+09 (!) • En écriture: si *n* est spécifié, le nombre sera justifié à droite dans un champ de *n* car. au min. Ex: `sprintf('%x %04X ', 255, 255, -5)` => M 'ff 00FF -5.000000e+000' et O 'ff 00FF -5'
`%f` `%nf` `%n.mf`	Correspond à un nombre **réel sans exposant** (de la forme `{-}mmm.nnn`) • En lecture: Ex: `sscanf('5.6e3 xy -5','%f xy %f')` => [5600 ; -5] • En écriture: si *n* est spécifié, le nombre sera justifié à droite dans un champ de *n* car. au min., et affiché avec *m* chiffres après la virgule. Ex: `sprintf('%f %0.2f ', 56e002, -78e-13, -5)` => '5600.000000 -0.00 -5.000000'
`%e` `%E` `%ne` `%nE` `%n.me` `%n.mE`	Correspond à un nombre **réel en notation scientifique** (de la forme `{-}m.nnnnnE{+|-}xxx`) • En lecture: Ex: `sscanf('5.6e3 xy -5','%e %*s %e')` => [5600 ; -5] • En écriture: si *n* est spécifié, le nombre sera justifié à droite dans un champ de *n* car. au min., et affiché avec *m* chiffres après la virgule. Avec `e` , le caractère 'e' de l'exposant sera en minuscule ; avec `E` il sera en majuscule. Ex: `sprintf('%e %0.2E ', 56e002, -78e-13, -5)` => '5.600000e+003 -7.80E-12 -5.000000e+00'
`%g` `%G` `%ng` `%nG`	Correspond à un nombre **réel en notation scientifique compacte** (de la forme `{-}m.nnnnnE{+|-}x`) • En lecture:

	Ex: `sscanf('5.6e3 xy -5','%g %*2c %g')` => [5600 ; -5] • En écriture: donne lieu à une forme plus compacte que `%f` et `%e`. Si *n* est spécifié, le nombre sera justifié à droite dans un champ de *n* car. au min. Avec `g`, le caractère 'e' de l'exposant sera en minuscule ; avec `G` il sera en majuscule. Ex: `sprintf('%g %G ', 56e002, -78e-13, -5)` => '5600 -7.8E-12 -5'
`%c` `%nc`	Correspond à 1 ou *n* **caractère(s)**, y compris d'éventuels caractères espace • En lecture: Ex: `sscanf('ab1 2cd3 4ef', '%2c %*3c')` => 'abcdef' • En écriture: Ex: `sprintf(' %c: (ASCII: %3d)\n', 'aabbcc')` => cela affiche : a: (ASCII: 97) b: (ASCII: 98) c: (ASCII: 99)
`%s` `%ns`	Correspond à une **chaîne** de caractères • En lecture: les chaînes sont délimitées par un ou plusieurs caractères espace Ex: `sscanf('123 abcd', '%2s %3s')` => '123abcd' • En écriture: si *n* est spécifié, la chaîne sera justifiée à droite dans un champ de *n* car. au min. Ex: `sprintf('%s|%5s|%-5s|', 'blahblah...', 'abc', 'XYZ')` => 'blahblah...| abc|XYZ |'
Caractères spéciaux	Pour faire usage de certains **caractères spéciaux**, il est nécessaire de les encoder de la façon suivante : • `\n` pour un saut à la ligne suivante (new line) • `\t` pour un tab horizontal • `%%` pour le caractère "%" • `\\` pour le caractère "\"
Tout autre caractère	Tout autre caractère (ne faisant pas partie d'une spécification `%...`) sera utilisé de la façon suivante : • En lecture: le caractère "matché" sera sauté. Exception: les caractères espace d'un format de lecture sont ignorés Ex: `sscanf('12 xy 34.5 ab 67', '%f xy %f ab %f')` => [12.0 ; 34.5 ; 67.0] • En écriture: le caractère sera écrit tel quel dans la chaîne de sortie Ex: `article='stylos' ; fprintf('Total: %d %s \n', 4, article)` => 'Total: 4 stylos'

De plus, les "spécifications de conversion" peuvent être modifiées (préfixées) de la façon suivante :

Spécifications	Description
`%-n` ...	• En écriture: l'élément sera **justifié à gauche** (et non à droite) dans un champ de *n* car. au min. Ex: `sprintf('|%-5s|%-5.1f|', 'abc', 12)` => '|abc |12.0 |'
`%0n` ...	• En écriture: l'élément sera **complété** à gauche **par des '0'** (chiffres zéros, et non caractères espace) dans un champ de *n* car. au min. Ex: `sprintf('|%05s|%05.1f|', 'abc', 12)` => '|00abc|012.0|'
`%*` ...	• En lecture: **saute** l'élément qui correspond à la spécification qui suit Ex: `sscanf('12 blabla 34.5 67.8', '%d %*s %*f %f')` => [12 ; 67.8]

7.9.3 Lecture/écriture formatée de chaînes

Lecture/décodage de chaîne

La fonction `sscanf` ("**s**tring **scan f**ormated") permet, à l'aide d'un **format** de lecture, de décoder le contenu d'une **chaîne de caractère** et d'en récupérer les données **sur un vecteur ou une matrice**. La lecture s'effectue en "**format libre**" en ce sens que sont considérés, comme **séparateurs** d'éléments dans la chaîne, un ou plusieurs espace ou tab.

Si la chaîne renferme davantage d'éléments qu'il n'y a de "spécifications de conversion" dans le format, le format sera "**réutilisé**" autant de fois que nécessaire pour lire toute la chaîne. Si, dans le format, on **mélange des spécifications de conversion numériques et de caractères**, il en résulte une variable de sortie (vecteur ou matrice) entièrement numérique dans laquelle les caractères des chaînes d'entrée sont stockés, à raison d'un caractère par élément de vecteur/matrice, sous forme de leur code ASCII.

vec = `sscanf(string, format)`
[*vec*, *count*] = `sscanf(string, format)`
Décode la chaîne `string` à l'aide du `format` spécifié, et retourne le **vecteur-colonne** `vec` dont tous les éléments seront **de même type**. La seconde forme retourne en outre, sur `count`, le nombre d'éléments générés.

Ex:
- `vec=sscanf('abc 1 2 3 4 5 6', '%*s %f %f')` => vec=[1;2;4;5]
  Notez que, en raison de la "réutilisation" du format, les nombres 3 et 6 sont ici sautés par le `%*s` !
- `vec=sscanf('1001 1002 abc', '%f %f %s')` => vec=[1001;1002;87;98;99]
  Mélange de spécifications de conversion numériques et de caractères => la variable 'vec' est de type nombre, et la chaîne 'abc' y est stockée par le code ASCII de chacun de ses caractères

mat = `sscanf(string, format, size)`
[*mat*, *count*] = `sscanf(string, format, size)`
Permet de remplir une **matrice** *mat*, **colonne après colonne**. La syntaxe du paramètre `size` est :
- `nb` => provoque la lecture des *nb* premiers éléments, et retourne un vecteur colonne
- `[nb_row, nb_col]` => lecture de *nb_row* x *nb_col* éléments, et retourne une matrice de dimension *nb_row* x *nb_col*

Ex:
- `vec=sscanf('1 2 3 4 5 6', '%f', 4)` => vec=[1;2;3;4]
- `[mat,ct]=sscanf('1 2 3 4 5 6', '%f', [3,2])` => mat=[1 4 ; 2 5 ; 3 6], ct=6
- `[mat,ct]=sscanf('1 2 3 4 5 6', '%f', [2,3])` => mat=[1 3 5 ; 2 4 6], ct=6

[*var1*, *var2*, *var3* ...] = `sscanf(string, format, 'C')`
(Proche du langage C, cette forme très flexible n'est disponible que sous Octave)
À chaque "spécification de conversion" du `format` utilisé est associée une variable de sortie *var-i*. Le **type** de chacune de ces variables peut être **différent** !

Ex: • `[str,nb1,nb2]=sscanf('abcde 12.34 45.3e14 fgh', '%3c %*s %f %f', 'C')` => str='abc', nb1=12.34, nb2=4.53e+15

Si une chaîne ne contient que des nombres, on peut aussi aisément récupérer ceux-ci à l'aide de la fonction `str2num` présentée au chapitre sur les "**Chaînes de caractères**".

Il existe encore la fonction de décodage de chaîne `strread` qui est extrêmement puissante ! Nous vous laissons la découvrir via l'aide ou la documentation.

Voyez aussi `textscan` qui est capable de lire des chaînes mais aussi des fichiers !

Écriture formatée de variables sur une chaîne

La fonction `sprintf` ("**s**tring **print f**ormated") lit les **variables** qu'on lui passe et les retourne, de façon formatée, **sur une chaîne de caractère**. S'il y a davantage d'éléments parmi les variables que de "spécifications de conversion" dans le format, le format sera "réutilisé" autant de fois que nécessaire.

string = `sprintf(format, variable(s)... )`
La variable *string* (de type chaîne) reçoit donc la(les) `variable(s)` formatée(s) à l'aide du `format` spécifié. Si, parmi les variables, il y a une ou plusieurs **matrice(s)**, les éléments sont envoyés colonne après colonne.

Ex:
• `nb=4 ; prix=10 ; disp(sprintf('Nombre d''articles: %04u     Montant: %0.2f Frs', nb, nb*prix))`
ou, plus simplement: `fprintf('Nombre d''articles: %04u     Montant: %0.2f Frs \n', nb, nb*prix)`
=> affiche: Nombre d'articles: 0004   Montant: 40.00 Frs

La fonction `mat2str` ("**mat**rix to **str**ing") décrite ci-dessous (et voir chapitre "**chaînes de caractères**") est intéressante

pour sauvegarder de façon compacte sur fichier des matrices sous forme texte (en combinaison avec **fprintf**) que l'on pourra relire sous MATLAB/Octave (lecture-fichier avec **fscanf** , puis affectation a une variable avec **eval**).

string = **mat2str(***mat {,n}***)**
Convertit la matrice *mat* en une chaîne de caractère *string* incluant les crochets [ ] et qui serait dont "évaluable" avec la fonction **eval** . L'argument *n* permet de définir la précision (nombre de chiffres).

Ex:
- **str_mat = mat2str(eye(3,3))** produit la chaîne "[1 0 0;0 1 0;0 0 1]"
- et pour affecter ensuite les valeurs d'une telle chaîne à une matrice x, on ferait **eval(['x=' str_mat])**

Voir aussi les fonctions plus primitives **int2str** (conversion nombre entier->chaîne) et **num2str** (conversion nombre réel->chaîne).

7.9.4 Lecture/écriture formatée de fichiers

Lecture de données numériques structurées

S'il s'agit de lire/écrire des fichiers au format texte contenant des données purement **numériques** et avec le même nombre de données dans chaque ligne du fichier (i.e. des tableaux de nombres), la technique la plus efficace consiste à utiliser les fonctions suivantes décrites plus en détail au chapitre "**Sauvegarde et chargement de données numériques via des fichiers-texte**" :

- données délimitées par un ou plusieurs espace et/ou tab :
  lecture avec : *var*= **load('***file_name***')** ; écriture avec : **save -ascii {-double}** *file_name var*
- données séparées par un *délimiteur* spécifié :
  lecture avec : *var*= **dlmread('***file_name***',** *délimiteur***)** ; écriture avec : **dlmwrite('***file_name***',** *var, délimiteur***)**
  voir aussi les fonctions **csvread** et **csvwrite**

Lecture intégrale d'un fichier sur une chaîne

string = **fileread('***file_name***')**
Cette fonction lit d'une traite l'intégralité du fichier-texte nommé *file_name* et retourne son contenu sur un vecteur colonne *string* de type chaîne

Ex: Dans l'exemple ci-dessous, la première instruction "avale" le fichier **essai.txt** sur le vecteur ligne de type chaîne **fichier_entier** (vecteur ligne, car on transpose le résultat de **fileread**). La seconde découpe ensuite cette chaîne selon les sauts de ligne (**\n**) de façon à charger le tableau cellulaire **tabcel_lignes** à raison d'une ligne du fichier par cellule.

```
fichier_entier =  fileread ('essai.txt')' ;
tabcel_lignes  =  strread (fichier_entier, '%s', 'delimiter', '\n') ;
```

[*status*, *string*] = **dos('type** *file_name***')**
[*status*, *string*] = **unix('cat** *file_name***')**
Ces instructions lisent également l'intégralité du fichier-texte nommé *file_name*, mais le retournent sur un vecteur ligne *string* de type chaîne. On utilisera la première forme sous Windows, et la seconde sous Linux ou MacOS.

Fonction textread

[*vec1*, *vec2*, *vec3* ...] = **textread(***file_name, format {,n}***)**
Fonction simple et efficace de **lecture d'un fichier**-texte nommé *file_name* dont l'ensemble des données répond à un *format* homogène. Les données peuvent être délimitées par un ou plusieurs espace, tab, voire même saut(s) de ligne newline. La lecture s'effectue ainsi en "**format libre**" (comme avec **sscanf** et **fscanf**).
• Le vecteur-colonne *vec1* recevra la 1ère "colonne" du fichier, le vecteur *vec2* recevra la 2e colonne, *vec3* la 3e, et ainsi de suite... La lecture s'effectue jusqu'à la fin du fichier, à moins que l'on spécifie le nombre *n* de fois que le *format* doit être réutilisé.

• Le nombre de variables *vec1 vec2 vec3*... et leurs types respectifs découlent directement du `format`
• Si *vec-i* réceptionne des chaînes de caractères, il sera de type "tableau cellulaire", en fait vecteur-colonne cellulaire (sous Octave jusqu'à la version 2.1.x c'était un objet de type "liste")

Les "spécifications de conversion" de format `%u` , `%d` , `%f` et `%s` peuvent être utilisées avec `textread` sous MATLAB et Octave.
Sous Octave seulement on peut en outre utiliser les spécifications `%o` et `%x` .
Sous MATLAB seulement on peut encore utiliser :
- `%[...]` : lit la plus longue chaîne contenant les caractèrens énumérés entre [ ]
- `%[^...]` : lit la plus longue chaîne non vide contenant les caractèrens non énumérés entre [ ]

Ex:
Soit le **fichier-texte** de données `ventes.txt` suivant :

```
10001     Dupond    Livres       12    23.50
10002     Durand    Classeurs    15     3.95
10003     Muller    DVDs          5    32.00
10004     Smith     Stylos       65     2.55
10005     Rochat    CDs          25    15.50
10006     Leblanc   Crayons     100     0.60
10007     Lenoir    Gommes       70     2.00
```

et le **script** MATLAB/Octave suivant :

```
[No_client, Nom, Article, Nb_articles, Prix_unit] = ...
   textread('ventes.txt', '%u %s %s %u %f') ;

Montant = Nb_articles .* Prix_unit ;

disp('     Client [No   ]      Nb  Articles     Prix unit.         Montant  ')
disp('  --------- -------   -----  ---------   -----------      ------------')
format = ' %10s [%d]   %5d  %-10s %8.2f Frs      %8.2f Frs\n' ;

for no=1:1:length(No_client)
  fprintf(format, Nom{no}, No_client(no), Nb_articles(no), ...
                  Article{no}, Prix_unit(no), Montant(no) ) ;
end

disp(' ')
fprintf('                                                TOTAL      %8.2f Frs \n', ...
                      sum(Montant) )
```

Attention : bien noter, ci-dessus, les accolades pour désigner éléments de `Nom{}` et de `Article{}` . Ce sont des "tableaux cellulaires" dont on pourrait aussi récupérer les éléments, sous forme de chaîne, avec : `char(Nom(no))` , `char(Article(no))` .

L'**exécution** de ce script: lit le fichier, calcule les montants, et affiche ce qui suit :

```
    Client [No   ]      Nb  Articles     Prix unit.         Montant
 --------- -------   -----  ---------   -----------      ------------
    Dupond [10001]      12  Livres        23.50 Frs        282.00 Frs
    Durand [10002]      15  Classeurs      3.95 Frs         59.25 Frs
    Muller [10003]       5  DVDs          32.00 Frs        160.00 Frs
     Smith [10004]      65  Stylos         2.55 Frs        165.75 Frs
    Rochat [10005]      25  CDs           15.50 Frs        387.50 Frs
   Leblanc [10006]     100  Crayons        0.60 Frs         60.00 Frs
    Lenoir [10007]      70  Gommes         2.00 Frs        140.00 Frs

                                             TOTAL       1254.50 Frs
```

Fonctions classiques de manipulation de fichiers (de type ANSI C)

file_id = `fopen(file_name, mode)`
`[file_id, message_err ] = fopen(file_name, mode)`

Ouvre le fichier de nom défini dans la variable-chaîne `file_name` , et retourne le "handle" *file_id* qui permettra de le manipuler par les fonctions décrites plus bas.

Le `mode` d'accès au fichier sera défini par l'une des chaînes suivantes :
- `'rt'` ou `'rb'` ou `'r'` : lecture seule (**r**ead)
- `'wt'` ou `'wb'` ou `'w'` : écriture (**w**rite), avec création du fichier si nécessaire, ou écrasement s'il existe
- `'at'` ou `'ab'` ou `'a'` : ajout à la fin du fichier (**a**ppend), avec création du fichier si nécessaire

- `'rt+'` ou `'rb+'` ou `'r+'` : lecture et écriture, sans création
- `'wt+'` ou `'wb+'` ou `'w+'` : lecture et écriture avec écrasement du contenu
- `'at+'` ou `'ab+'` ou `'a+'` : lecture et ajout à la fin du fichier, avec création du fichier si nécessaire

Le fait de spécifier `t` ou `b` ou aucun de ces deux caractères dans le *mode* a la signification suivante :

- `t` : ouverture en mode "texte"
- `b` ou rien : ouverture en mode "binaire" (mode par défaut)

⚠ Sous **Windows** ou **Macintosh**, il est important d'utiliser le mode d'ouverture "texte" si l'on veut que les fins de ligne soient correctement interprétées !

En cas d'échec (fichier inexistant en lecture, protégé en écriture, etc...), *file_id* reçoit la valeur "-1". On peut aussi récupérer un message d'erreur sous forme de texte explicite sur *message_err*

Handle prédéfinis (toujours disponibles, correspondant à des canaux n'ayant pas besoin d'être "ouverts") :

- `1` ou `stdout` : correspond à la *sortie standard* (*standard output*, c'est-à-dire fenêtre **console** MATLAB/Octave), pouvant donc être utilisé pour l'affichage à l'écran
- `2` ou `stderr` : correspond au canal *erreur standard* (*standard error*, par défaut aussi la fenêtre **console**), pouvant aussi être utilisé par le programmeur pour l'affichage d'erreurs
- `0` ou `stdin` : correspond à l'*entrée standard* (*standard input*, c'est-à-dire saisie au **clavier** depuis console MATLAB/Octave).

Pour offrir à l'utilisateur la possibilité de désigner le nom et emplacement du fichier à ouvrir/créer à l'aide d'une **fenêtre de dialogue** classique (interface utilisateur graphique), on se réfèrera aux fonctions `uigetfile` (lecture de fichier), `uiputfile` (écriture de fichier) et `zenity_file_selection` présentées au chapitre "**Interfaces-utilisateur graphiques**". Pour sélectionner un répertoire, on utilisera la fonction `uigetdir`.

[*file_name*, *mode*] = `fopen(file_id)`

Pour un fichier déjà ouvert de handle *file_id* spécifié, retourne son nom *file_name* et le *mode* d'accès.

`freport()`

Affiche la **liste** de tous les fichiers ouverts, avec *file_id*, *mode* et *file_name*.
On voit que les canaux `stdin`, `stdout` et `stderr` sont pré-ouverts

{*status*=} `fclose(file_id)`
`fclose('all')`

Referme le fichier de handle *file_id* (respectivement tous les fichiers ouverts). Le *status* retourné est "0" en cas de succès, et "-1" en cas d'échec.
A la fin de l'exécution d'un script ayant ouvert des fichiers, tous ceux-ci sont automatiquement refermés, même en l'absence de `fclose`.

variable = `fscanf(file_id, format {,size})`
[*variable*, *count*] = `fscanf(file_id, format {,size})`

Fonction de **lecture formatée** ("***f**ile **scan** f**ormated***") du fichier-texte identifié par son handle *file_id*.
Fonctionne de façon analogue à la fonction `sscanf` vue plus haut (à laquelle on renvoie le lecteur pour davantage de précision), sauf qu'on lit ici sur un fichier et non pas sur une chaîne de caractères.
Remarque importante : en l'absence du paramètre *size* (décrit plus haut sous `sscanf`), `fscanf` tente de lire (avaler, "slurp") l'intégralité du fichier (et non pas seulement de la ligne courante comme `fgetl` ou `fgets`).

Ex :
Soit le **fichier-texte** suivant :

```
10001     Dupond
  Livres      12    23.50
10002     Durand
  Classeurs   15     3.95
```

La **lecture** des données de ce fichier avec `fscanf` s'effectuerait de la façon suivante :

```
file_id = fopen('fichier.txt', 'rt') ;
no = 1 ;
while ~ feof(file_id)
  No_client(no)    = fscanf(file_id,'%u',1) ;
  Nom{no,1}        = fscanf(file_id,'%s',1) ;
  Article{no,1}    = fscanf(file_id,'%s',1) ;
  Nb_articles(no)  = fscanf(file_id,'%u',1) ;
  Prix_unit(no)    = fscanf(file_id,'%f',1) ;
  no = no + 1 ;
end
status = fclose(file_id) ;
```

[*variable*, *count*] = `scanf(format {,size})`
Fonction spécifiquement Octave de lecture formatée sur l'entrée standard (donc au clavier, handle `0`). Pour le reste, cette fonction est identique à `fscanf`.

line = `fgetl(file_id)`
Lecture, **ligne par ligne** ("***f**ile **get** **l**ine*"), du fichier-texte identifié par le handle `file_id`. A chaque appel de cette fonction on récupère, sur la variable *line* de type chaîne, la ligne suivante du fichier (sans le caractère de fin de ligne).

string = `fgets(file_id {,nb_car})`
Lecture, par **groupe** de `nb_car` ("***f**ile **get** **s**tring*"), du fichier-texte identifié par le handle `file_id`. En l'absence du paramètre `nb_car`, on récupère, sur la variable *string*, la ligne courante inclu le(s) caractère(s) de fin de ligne (<cr> <lf> dans le cas de Windows).

{*count*=} `fskipl(file_id ,nb_lignes)`
Avance dans le fichier `file_id` en **sautant** `nb_lignes` ("***f**ile **skip** **l**ines*"). Retourne le nombre *count* de lignes sautées (qui peut être différent de *nb_lignes* si l'on était près de la fin du fichier).

{*status*=} `feof(file_id)`
Test si l'on a atteint la **fin du fichier** identifié par le handle `file_id` : retourne "1" si c'est le cas, "0" si non.
Utile pour implémenter une boucle de lecture d'un fichier.

Ex : voir l'usage de cette fonction dans l'exemple `fscanf` ci-dessus

`frewind(file_id)`
Se (re)positionne au **début** du fichier identifié par le handle `file_id`.
Pour un positionnement précis **à l'intérieur** d'un fichier, voyez les fonctions :
- `fseek(file_id, offset, origin)` : positionnement `offset` octets après `origin`
- `position = ftell(file_id)` : retourne la `position` courante dans le fichier

{*count*=} `fprintf(file_id, format, variable(s)... )`
Fonction d'**écriture formatée** ("***f**ile **print** **f**ormated*") sur un fichier-texte identifié par son handle `file_id`, et retourne le nombre *count* de caractères écrits. Fonctionne de façon analogue à la fonction `sprintf` vue plus haut (à laquelle on renvoie le lecteur pour davantage de précision), sauf qu'on écrit ici sur un fichier et non pas sur une chaîne de caractères.

{*count*=} `fprintf(format, variable(s)... )`
{*count*=} `printf(format, variable(s)... )`
Utiliser `fprintf` en omettant le `file_id` (qui est est identique à utiliser le `file_id` "`1`" représentant la sortie standard) ou `printf` (spécifique à Octave), provoque une écriture/affichage à l'écran (i.e. dans la fenêtre de commande MATLAB/Octave).

Ex : affichage de la fonction y=exp(x) sous forme de tableau avec :
```
x=0:0.05:1 ; exponentiel=[x;exp(x)] ; fprintf('   %4.2f   %12.8f \n',exponentiel)
```

{*status*=} `fflush(file_id)`
Pour garantir de bonnes performances, Octave utilise un mécanisme de mémoire tampon (*buffer*) pour les opérations d'écriture. Les écritures ainsi en attente sont périodiquement déversées sur le canal de sortie, au plus tard lorsque l'on fait un `fclose`. Avec la fonction `fflush`, on force le vidage (*flush*) du *buffer* sur le canal de sortie.
Dans des scripts interactifs ou affichant des résultats en temps réel dans la fenêtre console, il peut être utile dans certaines situations de *flusher* la sortie standard avec `fflush(stdout)`. Voyez aussi la fonction `page_output_immediately` pour carrément désactiver le buffering.

Fonction textscan

Cette fonction est capable à la fois de lire un **fichier** ou de décoder une **chaîne**.

vec_cel = `textscan(file_id, format {,n})`
Lecture formatée d'un **fichier**-texte identifié par son handle `file_id` (voir ci-dessus) et dont l'ensemble des données répond à un `format` homogène.
La lecture s'effectue jusqu'à la fin du fichier, à moins que l'on spécifie le nombre `n` de fois que le `format` doit être réutilisé.
Dans le format, on peut notamment utiliser `\r`, `\n` ou `\r\n` pour *matcher* respectivement les caractères de fin

de lignes "carriage-return" (Mac), "line-feed" (Unix/Linux) ou "carriage-return + line-feed (Windows)
La fonction retourne un vecteur-ligne cellulaire *vec_cel* dont la longueur correspond au nombre de spécifications du format. Chaque cellule contient un vecteur-colonne de type correspondant à la spécification de format correspondante.

Ex: on peut lire le fichier **`ventes.txt`** ci-dessus avec :

```
file_id = fopen('ventes.txt', 'rt');
  vec_cel = textscan(file_id, '%u %s %s %u %f');
fclose(file_id);
```

et l'on récupère alors dans **`vec_cel{1}`** le vecteur de nombre des No, dans **`vec_cel{2}`** le vecteur cellulaire des Clients, dans **`vec_cel{3}`** le vecteur cellulaire des Articles, etc...

vec_cel = **`textscan(`***`string, format`*** **`{,`***`n`***`})`**
Opère ici sur la **chaîne** ***`string`***

7.9.5 Fonctions de lecture/écriture de fichiers spécifiques/binaires

`fread(...)` et **`fwrite(...)`**
Fonctions de lecture/écriture **binaire** (non formatée) de fichiers, pour un stockage plus compact qu'avec des fichiers en format texte. Voyez l'aide pour davantage d'information.

`xlsread(...)` et **`xlswrite(...)`**
Fonctions de lecture/écriture de classeurs **Excel** (binaire). Voyez l'aide pour davantage d'information.

`odsread(...)` et **`odswrite(...)`**
Fonctions de lecture/écriture de classeurs **OpenOffice/LibreOffice** (binaire). Voyez l'aide pour davantage d'information.

7.10 Réalisation d'interfaces-utilisateur graphiques (GUI)

7.10.1 Fonctions GUI communes à MATLAB et GNU Octave

Désignation de fichiers et de répertoires

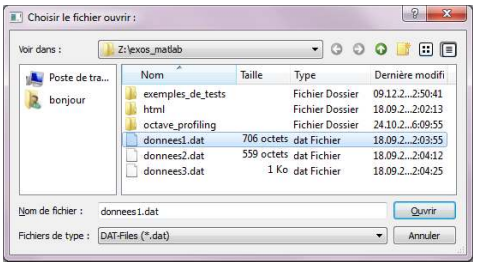
`[file_name, path] = uigetfile('filtre' {,'titre_dialogue'} {,x,y} )`

Fait apparaître à l'écran une fenêtre graphique de dialogue standard de désignation de fichier (selon figures ci-dessous). Fonction utilisée pour désigner un fichier à ouvrir en **lecture**, en suite de laquelle on utilise en principe la fonction `fopen` ... Une fois le fichier désigné par l'utilisateur (validé par bouton Ouvrir ou OK), le nom du fichier est retourné sur la variable `file_name`, et le chemin d'accès complet de son dossier sur la variable `path`. Si l'utilisateur referme cette fenêtre avec le bouton Annuler ou Cancel, cette fonction retourne `file_name = path = 0`

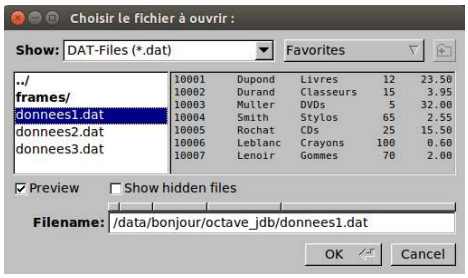
- La chaîne `titre_dialogue` s'inscrit dans la barre de titre de cette fenêtre
- Le `filtre` permet de spécifier le type des fichiers apparaissant dans cette fenêtre. Par exemple `*.dat` ne présentera que les fichiers ayant l'extension `.dat` (à moins que l'utilisateur ne choisisse `All files (*.*)` dans le menu déroulant `Fichiers de type:`)
- La fenêtre sera positionnée à l'écran de façon que son angle supérieur gauche soit aux coordonnées `x,y` par rapport à l'angle supérieur gauche de l'écran

Ex: le code ci-dessous fait désigner par l'utilisateur un fichier, puis affiche son contenu
(pour les fonctions `fopen`, `feof`, `fgetl`, `fprintf` et `fclose`, voir le chapitre **Entrées-sorties...**)

```
[fichier, chemin] = uigetfile('*.dat','Choisir le fichier à ouvrir :');
if fichier == 0
   disp('Aucun fichier n''a été désigné !')
else
   fid = fopen([chemin fichier], 'rt');  % entre crochets, concaténation
                                         % du chemin et du nom de fichier
   while ~ feof(fid)
      ligne = fgetl(fid);
      fprintf('%s\n', ligne)
   end
   status=fclose(fid);
end
```



Fenêtre uigetfile sous Windows 7



Fenêtre uigetfile sous Linux/Ubuntu
(remarquez l'option et la zone "preview")

`[file_name, path] = uiputfile('fname' {,'titre_dialogue'} {,x,y} )`

Fait apparaître une fenêtre de dialogue standard de **sauvegade** de fichier (en suite de laquelle on fait en principe un `fopen` ...). Le nom de fichier `fname` sera pré-inscrit dans la zone "Nom de fichier" de cette fenêtre. De façon analogue à la fonction `uigetfile`, le nom de fichier défini par l'utilisateur sera retourné sur la variable `file_name`, et le chemin complet d'accès au dossier sélectionné sur la variable `path`. Si un fichier de même nom existe déjà, MATLAB/Octave demandera une confirmation d'écrasement.

Ex: `[fichier, chemin] = uiputfile('resultats.dat','Sauver sous :');`

`path = uigetdir( {'path_initial' {,'titre_dialogue' } } )`

Fait apparaître à l'écran une fenêtre graphique de dialogue standard de **sélectionnement de répertoire**, et retourne le chemin de celui-ci sur *path*

- Le `path_initial` permet de positionner la recherche à partir du path ainsi spécifié. Si ce paramètre est omis, le positionnement initial s'effectue sur le répertoire courant
- La chaîne `titre_dialogue` s'inscrit dans la barre de titre de cette fenêtre

Fenêtres de dialogue standards

Les fonctions présentées ici permettent d'afficher des **fenêtres de dialogues standards**. Elles sont toutes "**modale**", c'est-à-dire que le programme est suspendu tant que l'utilisateur n'a pas répondu à la sollicitation de la fenêtre.

Nous les présentons sous forme d'exemples, et les illustrations proviennent de Octave sous Windows 7.

S'agissant de Octave, ces fonctions apparaissent avec la version 3.8 et s'appuient sur le package Java qui est désormais intégré à Octave Core.

Fenêtre d'information :

```
msg={'Message d''information', 'sur plusieurs lignes'};

msgbox(msg, 'Titre fenetre', 'none');
```

Le premier paramètre `msg` peut être une chaîne simple (qui sous Octave peut contenir `\n` pour générer un saut à la ligne) ou un vecteur cellulaire de chaînes (chaque élément étant alors affiché dans la fenêtre sur une nouvelle ligne).
Le 3e paramètre peut être `'warn'`, `'help'` ou `'error'` (=> affichage d'une icône dans la fenêtre), ou être ignoré ou valoir `'none'` (=> pas d'icône).

Remarque : notez la différence de comportement entre MATLAB et Octave :
Sous Octave, l'affichage de cette fenêtre suspend le programme qui redémarre une fois qu'on a cliqué OK
Cette fenêtre ne suspend pas l'exécution du programme (pour cela il faudrait faire `uiwait(msgbox(...))`)

Fenêtre d'avertissement :

```
msg={'Message d''avertissement', 'sur plusieurs lignes'};

warndlg(msg, 'Titre fenetre');
```

Cette fonction fournit le même résultat que `msgbox(msg, titre, 'warn')`

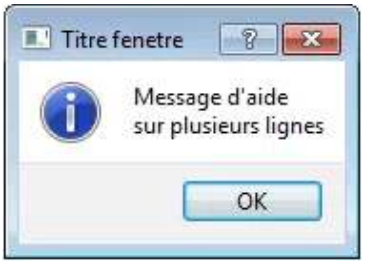
Remarque : cette fonction suspend l'exécution du programme sous Octave 3.8, mais pas sous MATLAB (idem que pour `msgbox`)

Fenêtre d'aide :

```
msg={'Message d''aide', 'sur plusieurs lignes'};

helpdlg(msg, 'Titre fenetre');
```

Cette fonction fournit le même résultat que `msgbox(msg, titre, 'help')`

Remarque : cette fonction suspend l'exécution du programme sous Octave 3.8, mais pas sous MATLAB (idem que pour `msgbox`)

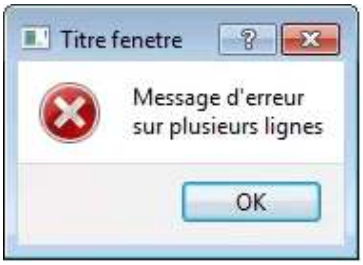


Fenêtre d'erreur :

```
msg={'Message d''erreur', 'sur plusieurs lignes'};

errordlg(msg, 'Titre fenetre');
```

Cette fonction fournit le même résultat que `msgbox(msg, titre, 'error')`

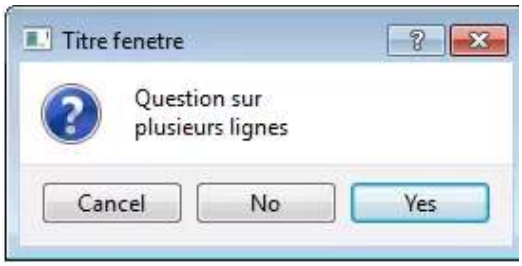
Remarque : cette fonction suspend l'exécution du programme sous Octave 3.8, mais pas sous MATLAB (idem que pour `msgbox`)



Fenêtre de question :

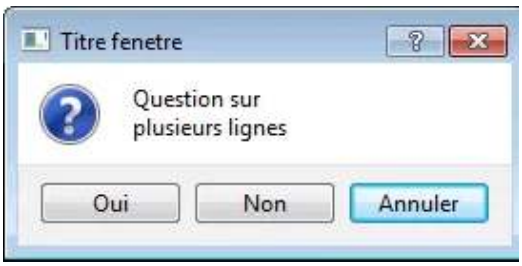
Cette fonction suspend l'exécution du programme et affiche une fenêtre de question. L'exécution se poursuit lorsque l'on a cliqué sur l'un des 3 boutons.

```
msg={'Question sur', 'plusieurs lignes'};

bouton = questdlg(msg, 'Titre fenetre','Yes')
```



Avec la première forme ci-dessus, affiche en-dessous du message *msg* les 3 boutons Cancel, No et Yes. Le 3e paramètres (facultatif, `'Yes'` dans l'exemple ci-dessus) indique quel bouton est pré-sélectionné et sera activé si l'on presse enter. Retourne sur *bouton* la chaîne de caractère correspondant au nom du bouton pressé.
Différence de comportement entre MATLAB et Octave 3.8 si on utilise la case de fermeture : MATLAB retourne une chaîne vide, Octave retourne `'Cancel'`

```
bouton = questdlg(msg, 'Titre fenetre', ...
                  'Annuler', 'Non', 'Oui', 'Annuler')
```



Avec la seconde forme ci-dessus, on spécifie le **nom des boutons**. Il peut y en avoir 2 ou 3 (ici 3), et on doit obligatoirement indiquer une chaîne supplémentaire spécifiant le nom du bouton activé par défaut si l'on presse enter.

Fenêtre de saisie de champs de **texte** :

Cette fonction suspend l'exécution du programme et affiche une fenêtre de saisie multi-champs. L'exécution se poursuit lorsque l'on a cliqué sur l'un des 2 boutons.

```
labels  = {'Prenom', 'Nom', 'Pays', 'Commentaire'};
li_cols = [1, 15; 1, 15; 1, 12; 2, 20];
val_def = {'', '', 'Suisse', ''};

vec_cel = inputdlg(labels, 'Titre fenetre', li_cols, val_def)
```

Affiche les champs de saisie étiquetés par les chaînes du vecteur cellulaire `labels`.

Le paramètre `li_cols` (facultatif, mais nécessaire si on veut passer le paramètre *val_def*) permet de définir la taille des champs :
• scalaire : nombre de lignes de tous les champs
• vecteur à 2 éléments [*lignes*, *colonnes*] : nombre de *lignes* et de *colonnes* de tous les champs
• vecteur (avec autant d'éléments que *labels*) : nombre de lignes de chaque champ
• matrice (de 2 colonnes et autant de lignes que d'éléments dans *labels*) : nombre de lignes et de colonnes de chaque champ
Notez que sous MATLAB le paramètre *colonnes* affecte non seulement la largeur du champ mais aussi la largeur d'affichage des étiquettes *labels* !

Le vecteur cellulaire de chaînes `val_def` (paramètre facultatif) permet de pré-remplir les champs par des valeurs par défaut (ici le Pays Suisse).

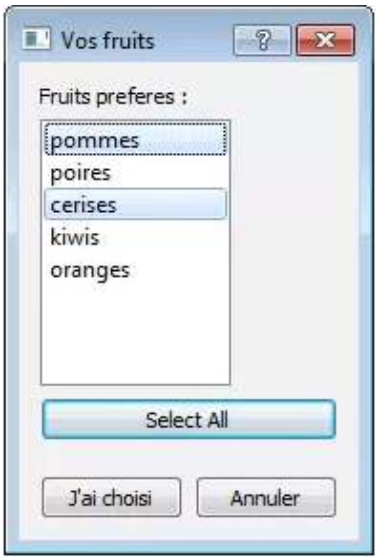
Lorsque l'on clique Ok, la fonction retourne les données saisies dans les différents champs sur un vecteur cellulaire de chaînes `vec_cel`.

Fenêtre de sélection dans une **liste** :

Cette fonction suspend l'exécution du programme et affiche une fenêtre de sélection. L'exécution se poursuit lorsque l'on a cliqué sur l'un des 2 boutons inférieurs.

```
valeurs={'pommes','poires','cerises','kiwis','oranges'};

[sel_ind, ok] = listdlg('ListString', valeurs, ...
        'Name', 'Vos fruits', ...
```

```
        'PromptString', 'Fruits preferes :', ...
        'SelectionMode', 'Multiple', ...
        'ListSize', [100 140], ...
        'InitialValue', [1, 3], ...
        'OKString', 'J''ai choisi', ...
        'CancelString', 'Annuler');
if ok
  fprintf('Fruits choisis: ')
  for ind = 1:numel(sel_ind)
    fprintf([valeurs{sel_ind(ind)} ' '])
  end
  fprintf('\n')
end
```



Les paramètres d'entrée de **`listdlg`** sont définis par paire : 'mot-clé', valeur

Paramètre d'entrée obligatoire :
• **`'ListString'`** réfère le vecteur cellulaire *valeurs* des éléments à afficher dans la liste

Paramètres d'entrée facultatifs :
• **`'Name'`** définit le titre de la fenêtre
• **`'PromptString'`** définit l'invite affichés au haut de la liste
• **`'SelectionMode'`** peut être 'Multiple' (sélection possible de plusieurs valeurs avec ctrl-clic, mode par défaut) ou 'Single'
• **`'ListSize'`** définit en pixels la taille [*largeur*, *hauteur*] de la zone d'affichage des *valeurs*
• **`'InitialValue'`** réfère un vecteur contenant les indices des éléments de *valeurs* qu'il faut pré-sélectionner
• **`'OKString'`** permet de redéfinir le texte du bouton Ok
• **`'CancelString'`** permet de redéfinir le texte du bouton Cancel

Paramètres de sortie :
• *`sel_ind`* est un vecteur ligne contenant les indices des éléments de *valeurs* que l'on a sélectionnés
• *`ok`* retourne **`1`** si on a cliqué Ok, et **`0`** si on a cliqué Cancel

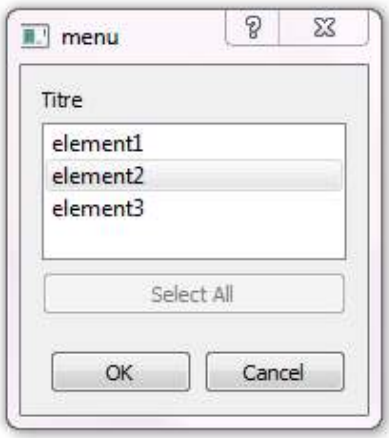
Autres accessoires GUI

`choix` **`= menu('`***`Titre`***`','`***`element1`***`','`***`element2`***`',...)`**

Implémente un **menu** retournant dans *`choix`* le **numéro d'ordre** de l'*`élément`* sélectionné :

- **MATLAB** : ce menu apparaît sous forme d'une fenêtre dans laquelle chaque *élement* fait l'objet d'un bouton (voir figure de gauche ci-contre). Il suffit de cliquer sur un bouton pour refermer la fenêtre et continuer l'exécution du programme.

- **Octave** : Sous Octave ≥ 4.0, ce menu est implémenté par une fenêtre analogue à celle de la fonction **`listdlg`** (voir figure de droite ci-contre). Le bouton Select All est désactivé. Il faut sélectionner l'*élément* désiré puis cliquer sur Ok pour refermer la fenêtre et continuer l'exécution du programme.



  Sous Octave ≤ 3.8, ce menu apparaissait sous forme texte dans la fenêtre de commande, et il fallait répondre en frappant le numéro de la ligne de menu

désirée, celle-ci étant alors retournés sur *choix*.

Si vous souhaitez implémenter un choix permettant de sélectionner plusieurs éléments, utilisez la fonction `listdlg` présentée au chapitre "**Interfaces-utilisateur graphiques**"

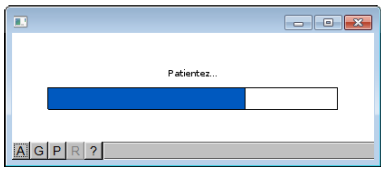
handle= `waitbar(x {,'texte'} )`

Affiche une barre de progression ("thermomètre") de longueur définie par le paramètre *x* dont la valeur doit être comprise entre 0.0 (barre vide) et 1.0 (barre pleine). On utilise cette fonction pour faire patienter l'utilisateur en lui indiquant la progression d'un traitement d'une certaine durée.
Cette barre se présente sous la forme d'une fenêtre graphique que l'on pourra refermer avec `close(handle)`.
Attention, seul le 1er appel à `waitbar` peut contenir le paramètre *texte* (qui s'affichera au-dessus de la barre), sinon autant de fenêtre seront crées que d'appels à cette fonction !

Sous Octave, voyez aussi la fonction analogue `zenity_progress`

Ex

```
barre = waitbar(0,'Patientez...');
for k=1:50
   pause(0.1)
     % pour les besoins de l'exemple, on
     %    fait ici une pause en lieu et
     %    place d'autres instructions...
   waitbar(k/50)
end
close(barre)
```



Ci-contre effet de ce code sous Octave Windows avec backend FLTK

Fonctions GUI propres aux fenêtres de figures


Chapitre en cours d'élaboration... Merci de patienter !

Pour développer de véritables interfaces graphiques sous MATLAB ou Octave, on réalise généralement cela dans des fenêtres de figures. Notez que ces possibilités apparaissent progressivement sous Octave depuis la version 3.8 et ne sont disponibles qu'avec les backends graphiques Qt et FLTK.

Nous énumérons ci-après les fonctions principales disponibles à la fois sous MATLAB et Octave en vous renvoyant à l'aide pour plus de détails :

- `uimenu` : création d'un **menu** (titre, articles de menu avec raccourcis et fonctions callback associés...) dans la barre de menus d'une figure
- `uicontextmenu` : définition de **menus contextuels** (i.e. apparaissant par un clic-droite sur un objet de la figure)

- `uitoolbar` : création d'une **barre d'outils** au haut de la figure, sous la barre de menus
- `uipushtool` : définition de **push-buttons** dans la barre d'outils
- `uitoggletool` : définition de **toggle-buttons** dans la barre d'outils

- `uicontrol` : définition de **contrôles** (popup-menus, boutons, sliders...) dans la zone de figure
- `uipanel` : permet de **grouper** les contrôles dans la zone de figure (containers, qui peuvent être imbriqués)

7.10.2 Fonctions GUI spécifiques à GNU Octave

Possibilités offertes par Zenity

Zenity est un outil du monde GNU/Linux sous GNOME permettant d'afficher aisément, depuis des scripts, des widgets basées sur les librairies GTK+ et Glade. Un package Octave-Forge, également nommé `zenity`, permet d'accéder aux possibilités de cet outil via des fonctions Octave nommées `zenity_*`

Pour plus de détails (installation, possibilités, utilisation...) : Afficher les détails ...

7.11 "Publier" un code MATLAB/Octave

Se rapprochant du concept de *notebook*, la fonction **publish** permet d'exécuter un script MATLAB/Octave en produisant un rapport comportant 3 parties :

- le code source du script
- les résultats d'exécution du script
- les éventuels graphiques produits par le script

publish('*script*{.m}' {, 'format','*fmt_rapport*', 'imageFormat','*fmt_images*', 'showCode',true|false, 'evalCode',true|false })

Avec Octave, le rapport est créé dans le répertoire courant ; sous MATLAB, dans un sous-répertoire nommé "html".
Les paramètres optionnels sont :

- **format** : valeurs possibles : **html** , **latex** , **doc** , **pdf** ; les valeurs par défaut sont : **html** sous MATLAB, **latex** sous Octave
- **imageFormat** : valeurs possibles : **png** , **jpg** , **pdf** ; les valeurs par défaut sont : **png** sous MATLAB, **pdf** sous Octave
- **showCode** : affichage du code dans le rapport, par défaut **true**
- **evalCode** : exécution du code, par défaut **true** ; si mis à **false** , le code n'est pas exécuté et aucun graphique n'est bien évidement produit

Ex: Soit le script ci-dessous nommé **code.m** :

```
% Exécution/publish de ce code sous Octave

% Données
    x= 4*rand(1,50) -2       % val. aléatoires entre -2 et +2
    y= 4*rand(1,50) -2       % val. aléatoires entre -2 et +2
    z= x.*exp(-x.^2 - y.^2)  % Z en ces points

% Premier graphique (double, de type multiple plots)
    figure(1)
    subplot(1,2,1)
    plot(x,y,'o')
    title('semis de points aleatoire')
    grid('on')
    axis([-2 2 -2 2])
    axis('equal')
    subplot(1,2,2)
    tri_indices= delaunay(x, y);   % form. triangles => matr. indices
    trisurf(tri_indices, x, y, z)  % affichage triangles
    title('z = x * exp(-x^2 - y^2) % sur ces points')
    zlim([-0.5 0.5])
    set(gca,'ztick',-0.5:0.1:0.5)
    view(-30,10)

% Second graphique (simple)
    figure(2)
    xi = -2:0.2:2 ;
    yi = xi';
    [XI,YI,ZI] = griddata(x,y,z,xi,yi,'nearest');  % interp. grille
    surfc(XI,YI,ZI)
    title('interpolation sur grille')
```

Son exécution avec la commande :

publish('code.m', 'format','html', 'imageFormat','png')

produit le rapport accessible **sous ce lien**

Octave Quick Reference

Octave Version 3.0.0

Starting Octave

`octave`	start interactive Octave session
`octave` *file*	run Octave on commands in *file*
`octave --eval` *code*	Evaluate *code* using Octave
`octave --help`	describe command line options

Stopping Octave

`quit` or `exit`	exit Octave
`INTERRUPT`	(*e.g.* `C-c`) terminate current command and return to top-level prompt

Getting Help

`help`	list all commands and built-in variables
`help` *command*	briefly describe *command*
`doc`	use Info to browse Octave manual
`doc` *command*	search for *command* in Octave manual
`lookfor` *str*	search for *command* based on *str*

Motion in Info

`SPC` or `C-v`	scroll forward one screenful
`DEL` or `M-v`	scroll backward one screenful
`C-l`	redraw the display

Node Selection in Info

`n`	select the next node
`p`	select the previous node
`u`	select the 'up' node
`t`	select the 'top' node
`d`	select the directory node
`<`	select the first node in the current file
`>`	select the last node in the current file
`g`	reads the name of a node and selects it
`C-x k`	kills the current node

Searching in Info

`s`	search for a string
`C-s`	search forward incrementally
`C-r`	search backward incrementally
`i`	search index & go to corresponding node
`,`	go to next match from last 'i' command

Command-Line Cursor Motion

`C-b`	move back one character
`C-f`	move forward one character
`C-a`	move to the start of the line
`C-e`	move to the end of the line
`M-f`	move forward a word
`M-b`	move backward a word
`C-l`	clear screen, reprinting current line at top

Inserting or Changing Text

`M-TAB`	insert a tab character
`DEL`	delete character to the left of the cursor
`C-d`	delete character under the cursor
`C-v`	add the next character verbatim
`C-t`	transpose characters at the point
`M-t`	transpose words at the point

[ ] surround optional arguments ... show one or more arguments

Killing and Yanking

`C-k`	kill to the end of the line
`C-y`	yank the most recently killed text
`M-d`	kill to the end of the current word
`M-DEL`	kill the word behind the cursor
`M-y`	rotate the kill ring and yank the new top

Command Completion and History

`TAB`	complete a command or variable name
`M-?`	list possible completions
`RET`	enter the current line
`C-p`	move 'up' through the history list
`C-n`	move 'down' through the history list
`M-<`	move to the first line in the history
`M->`	move to the last line in the history
`C-r`	search backward in the history list
`C-s`	search forward in the history list
`history` [-q] [*N*]	list *N* previous history lines, omitting history numbers if `-q`
`history -w` [*file*]	write history to *file* (`~/.octave_hist` if no *file* argument)
`history -r` [*file*]	read history from *file* (`~/.octave_hist` if no *file* argument)
`edit_history` *lines*	edit and then run previous commands from the history list
`run_history` *lines*	run previous commands from the history list
[*beg*] [*end*]	Specify the first and last history commands to edit or run.

If *beg* is greater than *end*, reverse the list of commands before editing. If *end* is omitted, select commands from *beg* to the end of the history list. If both arguments are omitted, edit the previous item in the history list.

Shell Commands

`cd` *dir*	change working directory to *dir*
`pwd`	print working directory
`ls` [*options*]	print directory listing
`getenv (`*string*`)`	return value of named environment variable
`system (`*cmd*`)`	execute arbitrary shell command string

Matrices

Square brackets delimit literal matrices. Commas separate elements on the same row. Semicolons separate rows. Commas may be replaced by spaces, and semicolons may be replaced by one or more newlines. Elements of a matrix may be arbitrary expressions, assuming all the dimensions agree.

`[` *x*`, ` *y*`, ... ]`	enter a row vector
`[` *x*`; ` *y*`; ... ]`	enter a column vector
`[` *w*`, ` *x*`; ` *y*`, ` *z* `]`	enter a 2×2 matrix

Multi-dimensional Arrays

Multi-dimensional arrays may be created with the *cat* or *reshape* commands from two-dimensional sub-matrices.

`squeeze (`*arr*`)`	remove singleton dimensions of the array.
`ndims (`*arr*`)`	number of dimensions in the array.
`permute (`*arr*`, ` *p*`)`	permute the dimensions of an array.
`ipermute (`*arr*`, ` *p*`)`	array inverse permutation.
`shiftdim (`*arr*`, ` *s*`)`	rotate the array dimensions.
`circshift (`*arr*`, ` *s*`)`	rotate the array elements.

Sparse Matrices

`sparse (...)`	create a sparse matrix.
`speye (`*n*`)`	create sparse identity matrix.
`sprand (`*n*`, ` *m*`, ` *d*`)`	sparse rand matrix of density *d*.
`spdiags (...)`	sparse generalization of *diag*.
`nnz (`*s*`)`	No. non-zero elements in sparse matrix.

Ranges

base `:` *limit*
base `:` *incr* `:` *limit*
Specify a range of values beginning with *base* with no elements greater than *limit*. If it is omitted, the default value of *incr* is 1. Negative increments are permitted.

Strings and Common Escape Sequences

A *string constant* consists of a sequence of characters enclosed in either double-quote or single-quote marks. Strings in double-quotes allow the use of the escape sequences below.

`\\`	a literal backslash
`\"`	a literal double-quote character
`\'`	a literal single-quote character
`\n`	newline, ASCII code 10
`\t`	horizontal tab, ASCII code 9

Index Expressions

var `(`*idx*`)`	select elements of a vector
var `(`*idx1*`, ` *idx2*`)`	select elements of a matrix
scalar	select row (column) corresponding to *scalar*
vector	select rows (columns) corresponding to the elements of *vector*
range	select rows (columns) corresponding to the elements of *range*
`:`	select all rows (columns)

Global and Persistent Variables

`global` *var1* `...`	Declare variables global.
`global` *var1* `=` *val*	Declare variable global. Set initial value.
`persistent` *var1*	Declare a variable as static to a function.
`persistent` *var1* `=` *val*	Declare a variable as static to a function and set its initial value.

Global variables may be accessed inside the body of a function without having to be passed in the function parameter list provided they are declared global when used.

Selected Built-in Functions

`EDITOR`	editor to use with `edit_history`
`Inf, NaN`	IEEE infinity, NaN
`NA`	Missing value
`PAGER`	program to use to paginate output
`ans`	last result not explicitly assigned
`eps`	machine precision
`pi`	π
`1i`	$\sqrt{-1}$
`realmax`	maximum representable value
`realmin`	minimum representable value

Assignment Expressions

var = *expr*	assign expression to variable
var (*idx*) = *expr*	assign expression to indexed variable
var (*idx*) = []	delete the indexed elements.
var {*idx*} = *expr*	assign elements of a cell array.

Arithmetic and Increment Operators

x + *y*	addition
x - *y*	subtraction
x * *y*	matrix multiplication
x .* *y*	element by element multiplication
x / *y*	right division, conceptually equivalent to `(inverse (y') * x')'`
x ./ *y*	element by element right division
x \ *y*	left division, conceptually equivalent to `inverse (x) * y`
x .\ *y*	element by element left division
x ^ *y*	power operator
x .^ *y*	element by element power operator
- *x*	negation
+ *x*	unary plus (a no-op)
x '	complex conjugate transpose
x .'	transpose
++ *x* (-- *x*)	increment (decrement), return *new* value
x ++ (*x* --)	increment (decrement), return *old* value

Comparison and Boolean Operators

These operators work on an element-by-element basis. Both arguments are always evaluated.

x < *y*	true if *x* is less than *y*
x <= *y*	true if *x* is less than or equal to *y*
x == *y*	true if *x* is equal to *y*
x >= *y*	true if *x* is greater than or equal to *y*
x > *y*	true if *x* is greater than *y*
x != *y*	true if *x* is not equal to *y*
x & *y*	true if both *x* and *y* are true
x | *y*	true if at least one of *x* or *y* is true
! *bool*	true if *bool* is false

Short-circuit Boolean Operators

Operators evaluate left-to-right. Operands are only evaluated if necessary, stopping once overall truth value can be determined. Operands are converted to scalars using the `all` function.

x && *y*	true if both *x* and *y* are true
x || *y*	true if at least one of *x* or *y* is true

Operator Precedence

Table of Octave operators, in order of increasing precedence.

; ,	statement separators
=	assignment, groups left to right
|| &&	logical "or" and "and"
| &	element-wise "or" and "and"
< <= == >= > !=	relational operators
:	colon
+ -	addition and subtraction
* / \ .* ./ .\	multiplication and division
' .'	transpose
+ - ++ -- !	unary minus, increment, logical "not"
^ .^	exponentiation

Paths and Packages

path	display the current Octave function path.
pathdef	display the default path.
addpath(*dir*)	add a directory to the path.
EXEC_PATH	manipulate the Octave executable path.
pkg list	display installed packages.
pkg load *pack*	Load an installed package.

Cells and Structures

var.*field* = ...	set a field of a structure.
var{*idx*} = ...	set an element of a cell array.
cellfun(*f*, *c*)	apply a function to elements of cell array.
fieldnames(*s*)	returns the fields of a structure.

Statements

`for` *identifier* = *expr* *stmt-list* `endfor`

Execute *stmt-list* once for each column of *expr*. The variable *identifier* is set to the value of the current column during each iteration.

`while` (*condition*) *stmt-list* `endwhile`

Execute *stmt-list* while *condition* is true.

break	exit innermost loop
continue	go to beginning of innermost loop
return	return to calling function

`if` (*condition*) *if-body* [`else` *else-body*] `endif`

Execute *if-body* if *condition* is true, otherwise execute *else-body*.

`if` (*condition*) *if-body* [`elseif` (*condition*) *elseif-body*] `endif`

Execute *if-body* if *condition* is true, otherwise execute the *elseif-body* corresponding to the first `elseif` condition that is true, otherwise execute *else-body*.

Any number of `elseif` clauses may appear in an `if` statement.

`unwind_protect` *body* `unwind_protect_cleanup` *cleanup* `end`

Execute *body*. Execute *cleanup* no matter how control exits *body*.

`try` *body* `catch` *cleanup* `end`

Execute *body*. Execute *cleanup* if *body* fails.

Strings

strcmp (*s*, *t*)	compare strings
strcat (*s*, *t*, ...)	concatenate strings
regexp (*str*, *pat*)	strings matching regular expression
regexprep (*str*, *pat*, *rep*)	Match and replace sub-strings

Defining Functions

`function` [*ret-list*] *function-name* [(*arg-list*)]
function-body
`endfunction`

ret-list may be a single identifier or a comma-separated list of identifiers delimited by square-brackets.

arg-list is a comma-separated list of identifiers and may be empty.

Function Handles

@*func*	Define a function handle to *func*.
@(*var1*, ...) *expr*	Define an anonymous function handle.
str2func (*str*)	Create a function handle from a string.
functions (*handle*)	Return information about a function handle.
func2str (*handle*)	Return a string representation of a function handle.
handle (*arg1*, ...)	Evaluate a function handle.
feval (*func*, *arg1*, ...)	Evaluate a function handle or string, passing remaining args to *func*

Anonymous function handles take a copy of the variables in the current workspace.

Miscellaneous Functions

eval (*str*)	evaluate *str* as a command
error (*message*)	print message and return to top level
warning (*message*)	print a warning message
clear *pattern*	clear variables matching pattern
exist (*str*)	check existence of variable or function
who, whos	list current variables
whos *var*	details of the variable *var*

Basic Matrix Manipulations

rows (*a*)	return number of rows of *a*
columns (*a*)	return number of columns of *a*
all (*a*)	check if all elements of *a* nonzero
any (*a*)	check if any elements of *a* nonzero
find (*a*)	return indices of nonzero elements
sort (*a*)	order elements in each column of *a*
sum (*a*)	sum elements in columns of *a*
prod (*a*)	product of elements in columns of *a*
min (*args*)	find minimum values
max (*args*)	find maximum values
rem (*x*, *y*)	find remainder of *x*/*y*
reshape (*a*, *m*, *n*)	reformat *a* to be *m* by *n*
diag (*v*, *k*)	create diagonal matrices
linspace (*b*, *l*, *n*)	create vector of linearly-spaced elements
logspace (*b*, *l*, *n*)	create vector of log-spaced elements
eye (*n*, *m*)	create *n* by *m* identity matrix
ones (*n*, *m*)	create *n* by *m* matrix of ones
zeros (*n*, *m*)	create *n* by *m* matrix of zeros
rand (*n*, *m*)	create *n* by *m* matrix of random values

Linear Algebra

chol (*a*)	Cholesky factorization
det (*a*)	compute the determinant of a matrix
eig (*a*)	eigenvalues and eigenvectors
expm (*a*)	compute the exponential of a matrix
hess (*a*)	compute Hessenberg decomposition
inverse (*a*)	invert a square matrix
norm (*a*, *p*)	compute the *p*-norm of a matrix
pinv (*a*)	compute pseudoinverse of *a*
qr (*a*)	compute the QR factorization of a matrix
rank (*a*)	matrix rank
sprank (*a*)	structural matrix rank
schur (*a*)	Schur decomposition of a matrix
svd (*a*)	singular value decomposition
syl (*a*, *b*, *c*)	solve the Sylvester equation

Equations, ODEs, DAEs, Quadrature

`*fsolve`	solve nonlinear algebraic equations
`*lsode`	integrate nonlinear ODEs
`*dassl`	integrate nonlinear DAEs
`*quad`	integrate nonlinear functions
`perror` (*nm*, *code*)	for functions that return numeric codes, print error message for named function and given error code

`*` See the on-line or printed manual for the complete list of arguments for these functions.

Signal Processing

`fft` (*a*)	Fast Fourier Transform using FFTW
`ifft` (*a*)	inverse FFT using FFTW
`freqz` (*args*)	FIR filter frequency response
`filter` (*a*, *b*, *x*)	filter by transfer function
`conv` (*a*, *b*)	convolve two vectors
`hamming` (*n*)	return Hamming window coefficients
`hanning` (*n*)	return Hanning window coefficients

Image Processing

`colormap` (*map*)	set the current colormap
`gray2ind` (*i*, *n*)	convert gray scale to Octave image
`image` (*img*, *zoom*)	display an Octave image matrix
`imagesc` (*img*, *zoom*)	display scaled matrix as image
`imread` (*file*)	load an image file
`imshow` (*img*, *map*)	display Octave image
`imshow` (*i*, *n*)	display gray scale image
`imshow` (*r*, *g*, *b*)	display RGB image
`imwrite` (*img*, *file*)	write images in various file formats
`ind2gray` (*img*, *map*)	convert Octave image to gray scale
`ind2rgb` (*img*, *map*)	convert indexed image to RGB
`rgb2ind` (*r*, *g*, *b*)	convert RGB to Octave image
`save a matrix to` *file*	

C-style Input and Output

`fopen` (*name*, *mode*)	open file *name*
`fclose` (*file*)	close *file*
`printf` (*fmt*, ...)	formatted output to `stdout`
`fprintf` (*file*, *fmt*, ...)	formatted output to *file*
`sprintf` (*fmt*, ...)	formatted output to string
`scanf` (*fmt*)	formatted input from `stdin`
`fscanf` (*file*, *fmt*)	formatted input from *file*
`sscanf` (*str*, *fmt*)	formatted input from *string*
`fgets` (*file*, *len*)	read *len* characters from *file*
`fflush` (*file*)	flush pending output to *file*
`ftell` (*file*)	return file pointer position
`frewind` (*file*)	move file pointer to beginning
`freport`	print a info for open files
`fread` (*file*, *size*, *prec*)	read binary data files
`fwrite` (*file*, *size*, *prec*)	write binary data files
`feof` (*file*)	determine if pointer is at EOF

A file may be referenced either by name or by the number returned from `fopen`. Three files are preconnected when Octave starts: `stdin`, `stdout`, and `stderr`.

Other Input and Output functions

`save` *file var* ...	save variables in *file*
`load` *file*	load variables from *file*
`disp` (*var*)	display value of *var* to screen

Polynomials

`compan` (*p*)	companion matrix
`conv` (*a*, *b*)	convolution
`deconv` (*a*, *b*)	deconvolve two vectors
`poly` (*a*)	create polynomial from a matrix
`polyderiv` (*p*)	derivative of polynomial
`polyreduce` (*p*)	integral of polynomial
`polyval` (*p*, *x*)	value of polynomial at x
`polyvalm` (*p*, *x*)	value of polynomial at x
`roots` (*p*)	polynomial roots
`residue` (*a*, *b*)	partial fraction expansion of ratio a/b

Statistics

`corrcoef` (*x*, *y*)	correlation coefficient
`cov` (*x*, *y*)	covariance
`mean` (*a*)	mean value
`median` (*a*)	median value
`std` (*a*)	standard deviation
`var` (*a*)	variance

Plotting Functions

`plot` (*args*)	2D plot with linear axes
`plot3` (*args*)	3D plot with linear axes
`line` (*args*)	2D or 3D line
`patch` (*args*)	2D patch
`semilogx` (*args*)	2D plot with logarithmic x-axis
`semilogy` (*args*)	2D plot with logarithmic y-axis
`loglog` (*args*)	2D plot with logarithmic axes
`bar` (*args*)	plot bar charts
`stairs` (*x*, *y*)	plot stairsteps
`stem` (*x*, *y*)	plot a stem graph
`hist` (*y*, *x*)	plot histograms
`contour` (*x*, *y*, *z*)	contour plot
`title` (*string*)	set plot title
`axis` (*limits*)	set axis ranges
`xlabel` (*string*)	set x-axis label
`ylabel` (*string*)	set y-axis label
`zlabel` (*string*)	set z-axis label
`text` (*x*, *y*, *str*)	add text to a plot
`legend` (*string*)	set label in plot key
`grid` [on|off]	set grid state
`hold` [on|off]	set hold state
`ishold`	return 1 if hold is on, 0 otherwise
`mesh` (*x*, *y*, *z*)	plot 3D surface
`meshgrid` (*x*, *y*)	create mesh coordinate matrices
