Premiers pas en Matlab

Florent Krzakała
Laboratoire P.C.T., UMR CNRS 7083,
ESPCI, 10 rue vauquelin, 75005, Paris, France.

1 Introduction : qu'est-ce que Matlab ?

Le logiciel Matlab constitue un système interactif et convivial de calcul numérique et de visualisation graphique. Destiné aux ingénieurs, aux techniciencs et aux scientifiques, c'est un outil très utilisé, dans les universités comme dans le monde industriel, qui intègre des centaines de fonctions mathématiques et d'analyse numérique (calcul matriciel —le MAT de Matlab—, traitement de signal, traitement d'images, visualisations graphiques, etc.). Notons que d'autres logiciels compatibles avec Matlab existent ; citons par exemple OCTAVE —sous Linux— ou encore SCILAB, développé en France par l'INRIA, qui sont tout deux très performants et de plus gratuits (téléchargement libre).

2 Démarrer et quitter Matlab

Pour lancer Matlab, tapez
≫ **matlab**
dans votre console Linux (ou cliquez simplement sur l'icône dans un environnement Windows). Pour quitter, tapez
≫ **quit**
ou
≫ **exit**
Matlab fonctionne à l'aide de commandes qu'il faut apprendre à utiliser. Vous pouvez à n'importe quel moment taper
≫ **help <command>**
pour savoir comment on utilise telle ou telle commande (par exemple essayez **help exit**). Le module d'aide est aussi assez efficace ; on peut par exemple une recherche de mots clefs en tapant
≫ **lookfor sinus**
on trouvera que la commande **sin** correspond, sans surprise, à la fonction sinus.

On peut enfin assister à une démonstration de Matlab en tapant
≫ **demo matlab**

3 Les variables dans Matlab, premiers calculs

Dans un premiers temps, Matlab peut bien sûr s'utiliser comme une vulgaire calculette. Ainsi
≫ **1+2**
donnera 3,
≫ **4^2**
retourne bien 16 et
≫ **(((1+2)^3)*4)/2**
donne bien 54, puisque Matlab respecte les règles usuelles des mathématiques. Il est aussi possible, comme dans tout langage de programmation, de définir des variables pour stocker un résultat ; ainsi après
≫ **a=1+2**
≫ **b=2+3**

≫ **a+b**
retourne bien 8. Si l'on veut éviter que Matlab affiche les résultats intermédiaires, on ajoute simplement un " ;" à la fin de la commande. On vérifiera que
≫ **a=1+2 ;**
≫ **b=2+3 ;**
≫ **a+b**
effectue les mêmes opérations que précédemment, mais sans afficher les résultats intermédiaires. Enfin, les calculs avec les variables complexes sont parfaitement possibles, en utilisant i ou j (avec bien évidement $i^2 = j^2 = -1$) :
≫ **a=1+2i ;**
≫ **b=2+3i ;**
≫ **a*b**
donne la réponse attendue, à savoir **ans = -4 + 7i**. On peut ainsi vérifier que
≫ **exp(i * pi)+1**
est bien nul (aux approximations numériques près), et que
≫ **sqrt(-1)**
retourne $-i$. Matlab étant très tolérant, il est possible d'appeler une variable i (ou bien j), mais dans ce cas, on ne pourra plus utiliser par la suite i comme étant le générateur des nombres complexes. Il est donc utile de savoir quelles sont les variables que nous avons utilisées depuis le début de la session, et pour cela il suffit d'écrire
≫ **who**
et pour en savoir plus sur elles
≫ **whos**
Enfin, pour les détruire, on utilise
≫ **clear a**
ou encore pour toutes les variables
≫ **clear all**
Certaines variables très utiles sont déjà définies, comme
≫ **pi**
ou
≫ **ans**
qui retourne la dernière réponse donnée par Matlab. Il est aussi utile de savoir qu'en tapant sur ↑ on peut rappeler les commandes précédentes (la flèche ↓ permettant de revenir) cela dispense de taper plusieurs fois des commandes identiques.

4 L'opérateur colon " :"

Avant d'aller plus loin, il est nécessaire de se familiariser avec une notation que l'on retrouve partout en Matlab, celle de l'opérateur " :" (en anglais **colon**). En Matlab, quand l'on écrit "1 :10" par exemple, cela signifie "tous les nombres de 1 a 10" (c.a.d. 1 2 3 4 5 6 7 8 9 10), et "1 :10 :2" signifie "tous les nombres de 1 a 10 avec un pas de 2 (et donc les nombres : 1 3 5 7 9)", le pas étant unitaire par défaut. Cette notation doit être impérativement comprise car elle est utilisée sans arrêt. On s'amusera par exemple a écrire
≫ **1 :10**
≫ **10 :-1 :1**
≫ **0 :pi/4 :pi**
et à observer le résultat.

5 Les vecteurs en Matlab

Nous n'avons vu ici que des variables scalaires (i.e. des nombres) mais Matlab permet d'utiliser aussi bien des vecteurs ou des matrices. Commençons par les vecteurs ; Pour définir un vecteur v, par exemple $v =$ (11 22 33 44 55), on utilise
≫ **v=[11 22 33 44 55]**
La commande

≫ **v'**
donne ensuite la transposée de ce vecteur, c'est-à-dire ici un vecteur colonne. On peut aussi écrire
≫ **z=[11 ; 22 ; 33 ; 44 ; 55 ;]**
pour obtenir directement un vecteur colonne z, mais il est bien plus simple d'écrire
≫ **z=v'**
ou encore
≫ **z=[11 22 33 44 55]'**
pour obtenir

$$z = \begin{pmatrix} 11 \\ 22 \\ 33 \\ 44 \\ 55 \end{pmatrix}$$

Il est aussi possible d'utiliser ces notations avec notre opérateur " :" pour définir un nouveau vecteur. Par exemple si l'on écrit [1 :1 :5], on définit un vecteur allant de 1 à 5 avec une incrémentation unité. Ainsi
≫ **w=[1 :1 :5]**
ou encore
≫ **w=[1 :5]**
nous permettent de définir le vecteur

$$w = (1\ 2\ 3\ 4\ 5)$$

Les mêmes opérations sont bien sûr possibles sur les vecteurs colonnes, par exemple
≫ **vec=[1 :2 :10]'**
définit le vecteur

$$vec = \begin{pmatrix} 1 \\ 3 \\ 5 \\ 7 \\ 9 \end{pmatrix}$$

Pour accéder a une valeur particulière d'un vecteur, on écrira par exemple
≫ **v(2)**
qui donne simplement accès au 2em élément, c.-à.-d. ici la valeur 22. Il est aussi possible d'utiliser les " :" pour sélectionner plusieurs éléments d'un vecteur ! Voici quelques exemples :
≫ **v(1 :3)**
donne accès aux éléments qui vont de 1 à 3 (avec un pas de 1 sans plus de précision, donc les éléments 1, 2 et 3). Cette commande retourne le vecteur (11 22 33). On peut aussi écrire
≫ **v(1 :2 :5)**
qui donne accès aux éléments qui vont de 1 à 5, mais en incrémentant avec un pas de 2. Dans ce cas, le vecteur (11 33 55) est retourné. Si l'on écrit
≫ **v**
tout le vecteur est retourné, mais l'on obtient le même résultat par
≫ **v(:)**
le symbole " :" sans autres précisions signifiant "tout le vecteur". Matlab permet la multiplication des vecteurs, cependant on surveillera l'ordre de ce que l'on écrit : ainsi si **vec*vec**, retourne un message d'erreur, **vec*vec'** retourne une matrice et **vec'*vec** un nombre...

6 Les matrices en Matlab

On vient de voir notre première matrice. Il suffit en fait d'ajouter une dimension aux vecteurs pour créer des matrices avec les mêmes notations. Ainsi
≫ **A=[1 2 3 ; 2 4 5 ; 6 7 8 ;]**

génère la matrice

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 4 & 5 \\ 6 & 7 & 8 \end{pmatrix}$$

Il est important de bien faire attention à la taille des objets que l'on définit sous peine d'obtenir des messages d'erreur. Si l'on veut créer une matrice à partir de ses colonnes, il suffit d'écrire
≫ **A=[[1 2 6]' [2 4 7]' [3 5 8]']**
Pour lire les éléments, on utilise les même méthodes que précédemment. Ainsi la commande
≫ **A(1,3)**
permet d'accéder à l'élément (1,3) de la matrice A, qui est ici 3. De même
≫ **A(1 :3,2 :3)**
retourne les lignes 1 à 3 et les colonnes 2 à 3 :

$$\begin{pmatrix} 2 & 3 \\ 4 & 5 \\ 7 & 8 \end{pmatrix}$$

Le même résultat est obtenu par la notation
≫ **A(1 :1 :3,2 :1 :3)**
Il est enfin souvent utile de pouvoir accéder à l'un des vecteur (ligne ou colonne) d'une matrice. Il faut alors se souvenir que l'on accède à toutes les valeurs d'un vecteur par la notation " :" et que, par conséquent, on écrira **A(:,2)** pour obtenir la deuxième colonne de A et **A(:,3)** pour la troisième. De même **A(1, :)** retourne toute la première ligne.

Enfin, on souhaite fréquemment transposer une matrice ; en Matlab, encore une fois, on procède comme pour les vecteurs :
≫ **A'**
est bien la transposée de **A**. Il existe un certain nombre de commandes qui génèrent automatiquement des matrices particulières :

1. **rand(p)** crée une matrice $p \times p$ dont les éléments sont aléatoires, distribués selon une loi de probabilité uniforme dans l'intervalle $[0, 1[$.
2. **randn(p)** crée une matrice $p \times p$ dont les éléments sont aléatoires, distribués selon une loi de probabilité gaussienne de moyenne nulle et de variance unité.
3. **eye(p)** crée une matrice identité $p \times p$.
4. **zeros(p)** crée une matrice de zéros $p \times p$.
5. **ones(p)** crée une matrices de uns $p \times p$.

Il est aussi possible d'utiliser ces commandes pour créer des matrices non carrées, en écrivant par exemple **eye**$(\mathbf{5}, \mathbf{3})$. On consultera avec profit l'aide en ligne de Matlab. Il est possible, pour finir, de définir les matrices par blocs. Ainsi, pour créer une matrice

$$B = \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \end{pmatrix}$$

il suffira d'écrire
≫ **B=[eye(3) eye(3) ;ones(3) zeros(3) ;]**
ou bien en utilisant les colonnes
≫ **B=[[eye(3) ones(3)]' [eye(3) zeros(3)]']**
ou bien encore
≫ **B=[[eye(3) ; ones(3)] [eye(3) ;zeros(3)]]**

7 Opérations matricielles avec les vecteurs et les matrices

Matlab est vraiment performant pour la manipulation de matrices, de vecteurs, ou de tableaux en général. Pour la clarté de l'exposé, on notera par la suite les vecteurs avec des minuscules et les matrices par des majuscules. La multiplication de vecteurs et de matrices se fait en utilisant la notation "*". Par exemple

≫ **C=[1 3 4 ; 5 2 1 ; 3 1 2]**
≫ **b=[3 1 2]**
≫ **v=[1 3 4]**
≫ **v*b'**
≫ **v'*b**
≫ **A*C**
≫ **C*A**
≫ **A*v'**

Notez bien évidement que cette multiplication n'est pas commutative, comme il se doit. Pour les matrices carrées, il existe aussi des fonctions spécifiques extrêmement utiles :

1. **inv(A)** calcule l'inverse de A.
2. **det(A)** calcule le déterminant de A.
3. **diag(A)** retourne la diagonale.
4. **[V,E]=eig(A)** retourne deux matrices V et E. V est une matrice dont les colonnes contiennent les vecteurs propres de A, et E est une matrice dont la diagonale contient les valeurs propres de A.

Matlab permet aussi d'effectuer des inversion de matrices, et donc de résoudre rapidement des systèmes d'équations linéaires. Par exemple, pour résoudre un système du type $A \cdot x = b$, la solution est donnée par $x = A^{-1} \cdot b$. Cela donne, en Matlab :

≫ **inv(A)*b**

On peut aussi écrire

≫ **A \ b**

où l'on notera le symbole \ qui signifie *diviser par la gauche* ou plus directement, si l'on tient à utiliser la division usuelle (la division à droite)

≫ **eye(3)/A*b**

8 Opérations dans les éléments des matrices/vecteurs

Si nous voulons travailler directement avec les éléments de matrices/vecteurs, par exemple multiplier les éléments de A par ceux de B, il ne s'agit plus alors d'une opération matricielle, mais bien d'une multiplication des éléments de matrices. C'est parfois utile quand on utilise un vecteur comme un tableau. Pour spécifier que l'on utilise les éléments et non pas les matrices, on écrit —par exemple pour la multiplication— ".*" au lieu de "*". Ainsi on aura pour la multiplication élément par élément des vecteurs

≫ **v.*b**

ou

≫ **b.*v**

et de même pour des matrices

≫ **A.*C**
≫ **b./v**
≫ **C./A**
≫ **A.^2**

Certaines fonctions, qui ne sont pas destinées directement aux matrices, s'appliquent directement aux éléments et n'ont donc pas besoin du point :

≫ **sin(v)**
≫ **log(C)**
≫ **exp(A)**
≫ **0.5-C**

Enfin, les fonctions suivantes sont parfois très utiles : **[m,n]=size(A)**, retourne m, le nombre de lignes, et n le nombre de colonnes de A. **min(V)** retourne un vecteur qui contient l'élément le plus petit de chaque colonne de V (ou le plus petit élément de V si V est un vecteur). De même **min(min(V))** retourne l'élément le plus petit de la matrice V et **max(A)** effectue les mêmes opérations pour les maximums.

9 Graphes et dessins

Il est facile d'effectuer diverses représentations graphiques avec Matlab, en $2d$ aussi bien qu'en $3d$. En tapant
≫ **help plot**
on a accès à toutes les options disponibles. On pourra aussi s'aider de la demo Matlab. Pour afficher la fonction $x\sin(x)$ dans l'intervalle $[0:10]$, il suffit d'écrire
≫ **x=[0 :0.1 :10] ;**
≫ **y=x.*sin(x) ;**
≫ **plot(x,y)**
Notons que l'on a ici besoin de la multiplication "avec le point" (.*) puisque x et y sont des vecteurs. Enfin, on peut nommer les différents graphes de la façon suivante
≫ **xlabel('x')**
≫ **ylabel('y')**
≫ **title('...')**
On peut aussi afficher deux courbes a la fois en tapant **hold on** après le premières plot et écrire **hold off** après le dernier. Ou encore plus simplement
≫ **plot(x,sin(x),x,cos(x))**
On peut aussi diviser l'écran en deux parties dans le même graphe
≫ **subplot(1,2,1)**
≫ **plot(x,sin(x),'sin')**
≫ **subplot(1,2,2)**
≫ **plot(x,cos(x),'cos')**

Pour créer un fichier image, on peut soir utiliser les menus deroulants avec la souris ou encore — par exemple pour un fichier PostScript (.ps)— taper
≫ **print -dps name.ps**
Il est souvent utile de tracer des courbes en utilisant des axes logarithmiques. Avec Matlab, il suffira d'utiliser **loglog**, **semilogx** et **semilogy** à la place de **plot**. On pourra aussi créer des vecteurs avec des espacements logarithmiques en utilisant
≫ **y=logspace(-1,2,6)**
pour un vecteur de 6 valeurs allant de 10^-1 à 10^2. Il est enfin possible de tracer des fonctions en $3d$, de la même façon qu'en $2d$, en utilisant la fonction plot3, similaire a la fonction $2d$. Par exemple, pour tracer une hélice on écrira :
≫ **t = 0 :pi/50 :10*pi ;**
≫ **plot3(sin(t),cos(t),t) ;**
Pour tracer des surface en $3d$ dimension, il est nécessaire d'introduire une maille (en anglais *mesh*). L'exemple suivant montre comment utiliser cette nouvelle fonction
≫ **[x,y]=meshgrid(-2 :0.2 :2,-2 :.2 :2) ;**
≫ **z=x.*exp(-x.^2-y.^2)**
≫ **mesh(z)**
On pourra s'amuser a essayer les différentes commandes **mesh**,**surf**, **surfl** ou encore **pcolor** pour les plots "de contours".

Nous avons vu comment tracer des fonctions, mais souvent nous voulons afficher des points écrits dans un fichier. Pour accéder à ces valeurs, le plus simple est d'utiliser la fonction Matlab "load". Si par exemple nous avons un fichier data.txt avec le format suivant
0 1.
1 1.1
2 1.5
3 1.4

4 1.7
... ...
il suffira, pour afficher la seconde colonne en fonction de la première, d'écrire
load data.txt ;
x=data(:.1) ;
y=data(:.2) ;
plot(x,y)
La première commande lit le fichier et crée une matrice (avec le même nom) contenant ces donnés. On copie ensuite la première colonne dans le vecteur x, la seconde dans le vecteur y, et l'on affiche finalement le graphe. On procède trivialement de la même façon en $3d$.

10 Les boucles et les instructions conditionnées

L'utilisation des boucles est le premier pas dans la programmation. En Matlab, les boucles **for** et **while** sont très utilisées pour les processus itératifs. La boucle **for** est associée à une variable, et exécute un processus plusieurs fois en prenant à chaque fois une nouvelle valeur pour cette variable. Pour illustrer cela, créons d'abord un vecteur de taille 5 rempli de valeurs aléatoires
≫ **v=rand(1,5)**
Si l'on veut soustraire à tous les éléments (sauf le premier) de ce vecteur v la première valeur (i.e. v(1)), on écrit
≫ **for** $i = 2 : 5$
v(i)=v(i)-v(1)
end
La première ligne se comprend *pour toutes les valeurs qui vont de 2 à 5, exécute les commandes suivantes, jusqu'à end.* Un exemple simple d'utilisation de boucle pour la résolution d'un vrai problème est celui des équations différentielles. Supposons que l'on veuille calculer $y(x)$, qui vérifie

$$\frac{dy}{dx} = x^2 - y^2$$

avec $y(0) = 1$. On peut discrétiser l'équation différentielle (c'est la méthode d'Euler) ce qui donne $y(x + \Delta x) = y(x) + \Delta x(x^2 - y^2)$. Cette équation nous dit qu'il est possible de calculer $y(x + \Delta x)$ si l'on connaît $y(x)$, ce qui permet une solution itérative. Notant $h = \Delta x$, il suffit ensuite de procéder comme suit pour résoudre l'équation différentielle :
≫ **n=0.1**
≫ **x=[0 :n :2] ;**
≫ **y=0*x ;**
≫ **y(1)=1**
Notons ici un piège classique : Matlab numérote ses éléments de matrices/vecteurs en partant de 1. Ce qui est pour nous la composante y(0) —la valeur en $x = 0$ de $y(x)$— doit donc être placée à la première place du vecteur, c.-à.-d.. en y(1) pour Matlab. De manière générale, on fera **extrêmement** attention aux indices dans la programmation.
≫ **size(x)**
Cette dernière commande donne le résultat [1 21]. On introduit ensuite une boucle
≫ **for i=2 :21**
y(i)=y(i-1)+n*(x(i-1)^2-y(i-1)^2) ;
end
On peut alors afficher un graphe du résultat
≫ **plot(x,y)**
La commande **while** fonctionne de façon similaire ; elle exécute en boucle les commandes *tant qu'une* certaine condition est satisfaite. On aurait donc pu écrire, pour un résultat identique
≫ **i=2**
≫ **y(1)=1**
≫ **while(i<=21)**
y(i)=y(i-1)+n*(x(i-1)^2-y(i-1)^2) ;
i=i+1

end
Les instructions conditionnés, enfin, permettent de n'effectuer une opération *que si* une certaine condition est satisfaite. La plus simple est **if**, qui exécute des commandes **seulement si** une condition est remplie (mais à la différence de while, elle n'exécute ces commandes qu'une seule fois). Par exemple, si l'on veut mettre à zéro toutes les composantes supérieures à 0.2 du vecteur v, on peut écrire
≫ **for i=1 :5**
≫ **if v(i)>0.2**
≫ **v(i)=0**
≫ **end**
≫ **end**

11 Les fonctions dans Matlab

On l'a vu, Matlab a une bonne bibliothèque de fonctions mathématiques (cos,sin,exp,log,sqrt...) et l'on pourra par exemple vérifier que
≫ **cos(0.5)+i*sin(0.5)**
donne bien la même chose que
≫ **exp(i/2)**
comme nous l'a appris Euler. on encore utiliser des fonctions plus exotiques (la fonction erreur **erf**, les fonctions de Bessel que l'on trouvera par **lookfor bessel**.... Plus important encore, on peut définir ses propres fonctions. Pour cela, il faut créer un fichier Matlab (on dit encore un *m-file*), ce que l'on peut faire en utilisant l'éditeur prévu par matlab (fichier-¿Nouveau-¿m-file), ou soit en ouvrant votre éditeur favori. Si l'on crée par exemple un fichier **EulerCheck.m** dans lequel on écrit
≫ **function y=EulerCheck(x)**
≫ **y=cos(x)=i * sin(y)** ;
On notera le ";" pour éviter que Matlab nous affiche ce qu'il calcule dans la fonction. On accédera ensuite aux valeurs de cette fonction dans la fenêtre principale de Matlab par ≫ **EulerCheck(0.5)**
Si plusieurs arguments sont fournis, on doit les séparer par une virgule ; et si plusieurs sont retournés, il faut les mettre entres crochets. Par exemple (pour un fichier nommé distance.m)
≫ **function [d,dx,dy]=distance(x1,y1,x2,y2)**
≫ **d=sqrt((x2-x1)^2 + (y2-y1)^2)**
≫ **dx=x2-x1**
≫ **dy=y2-y1**
permet de retourner la distance absolue et les distance en x et y entre deux points de coordonnées (x_1, y_1) et (x_2, y_2). En tapant dans Matlab
≫ **[dis,disx,disy]=distance(0,0,1,1) ;**
On vérifiera ensuite que $dis = 1.4142$ et que $dis_x = dis_y = 1$.

12 Fichiers exécutables : les scripts

Il arrive que l'on doive exécuter la même tâche plusieurs fois mais en changeant seulement quelques paramètres. Une bonne façon de faire cela est de créer un fichier exécutable, ou encore *script*. Continuons avec la résolution eulérienne de l'équation $y' = 1/y$. Il faut créer un fichier dont le nom porte l'extension .m. Appelons-le **SimpleEuler.m**. Ouvrez ce fichier en utilisant votre éditeur favori,
≫ **emacs SimpleEuler.m**
(ou nedit, ou vi...) ou bien encore utiliser l'éditeur intégré dans matlab pour créer un nouveau m-file, puis tapez
% file : SimpleEuler.m
% To find an approximation of dy/dx=1/y
% with y(t=0)=starty
% you need first to specify h and starty
x=[0 :h :1] ;

y=0*x ;
y(1)=starty
for i=2 :max(size(y))
y(i)=y(i-1)+h/y(i-1) ;
end
Sauvegardez-le, puis tapez
≫ **help SimpleEuler**
Ce qui retourne les commentaires après "%". Pour l'exécuter, on a besoin de h et de *starty*
≫ **h=0.01 ;**
≫ **starty=1 ;**
≫ **SimpleEuler**
≫ **plot(x,y)**
Il est alors simple de recommencer autant de fois que l'on veut pour différentes valeurs initiales.

13 Fichiers sous-routines : les fonctions (bis)

Il est possible de généraliser les fichiers exécutables pour créer des fonctions, ou encore des sous-routines. On l'a vu, matlab permet de créer des fonctions mathématiques mais l'on peut demander (presque) tout à une telle fonction !

Du point de vue de la programmation, la différence principale entre les fonctions et les programmes est que 1) quand on calcule des variables dans des fonctions, elles sont complètement effacées à la fin du calcul à l'exception de celles dont on désire *retourner* la valeur, et 2) qu'il est bien sur aussi possible d'envoyer des paramètres à la fonction, ce qui fait toute sa force. Revenons encore à notre exemple eulérien, et créons un fichier **EulerApprox.m** dans lequel on va définir une fonction du même nom (il est important que le nom de la fonction et celui du fichier soient identiques)
function [x,y]=EulerApprox(startx,h,endx,starty)
% Find the Euler approximation of dy/dx=1/y
% in the range [startx :endx]
% with the starting condition starty
% and using h as the time step
x=[startx :h :endx] ;
y=0*x ;
y(1)=starty
for i=2 :max(size(y))
y(i)=y(i-1)+h/y(i-1) ;
end
On peut alors écrire dans Matlab
≫ **[x,y]=EulerApprox(0,0.01,1,1) ;**
≫ **plot(x,y)**
pour obtenir le même résultat que précédemment. On notera que les variables intermédiaires (c.-à.-d. ici la variable i par exemple) n'existent plus en dehors de la boucle. Notez la différence si l'on tape
≫ **clear all ;**
≫ **h=0.01 ;**
≫ **starty=1 ;**
≫ **SimpleEuler**
≫ **i**
ou si l'on tape
≫ **clear all ;**
≫ **[x,y]=EulerApprox(0,0.01,1,1)**
≫ **i**
Cela illustre le concept de variable locale : les variables utilisées dans une fonction n'existent plus en dehors de cette fonction : **c'est une notion fondamentale** que l'on retrouve dans tous les langage de programmation. Notons que dans la pratique, pour résoudre une équation différentielle, il est préférable d'utiliser plutôt les fonctions spécifiques de Matlab ; essayez **help ode23** ou **help ode45** pour apprendre à utiliser ces (puissantes)

fonctions.

14 Avant d'aller plus loin, quelques conseils ...

1. Il faut se souvenir que Matlab distingue les majuscules et minuscules et qu'ainsi la matrice **A** et la matrice **a** ne sont pas identiques ! Les lettres accentuées sont interdites également.
2. Il ne faut **pas** donner des noms aux variables qui correspondent à des noms de fonctions (soit celles de Matlab, soit celles de l'utilisateur) : dans ce cas Matlab ne peut plus accéder à ces fonctions. C'est particulièrement vrai aussi pour les variables complexes i et j que l'on désactive complément quand l'on nomme une variable i ou j dans un boucle !
3. On tachera d'utiliser au maximum les **fichiers .m** (c.a.d. les scripts et les fonctions) ; l'idée est d'utiliser la page principale de Matlab comme une feuille de papier brouillon et de réserver les calculs difficiles aux fichiers .m.
4. Il est primordial d'insérer des commentaires dans les fichiers .m, et ce *au fur et à mesure de la programmation* (et pas seulement lorsque la programmation est terminée).
5. De même, toujours garder à l'esprit, lorsque l'on écrit des fonctions, la notion de variable locale. En général, une variable n'existe que dans un certain contexte.
6. Enfin, il faut vectoriser le plus possible les opérations et éviter les boucles, et **ceci est aussi fondamental !** puisque Matlab est un langage interprété. Il est souvent plus simple de tout écrire sous forme de boucles ; si cela n'est pas un problème pour des petites opérations, un gain de temps *très considérable* est obtenu en vectorisant les calculs. Le petit exemple suivant est très instructif ; pour trouver la différence maximale entre le vecteur x et le vecteur y définis par
   ≫ **x=rand(1,100000) ;y=rand(1,100000) ;**
   on vérifiera que
   ≫ **maxdif=max(x-y) ;**
   donne un résultat immédiatement tandis que
   ≫ **for i=1 :100000 ;dif(i)=x(i)-y(i) ; end ; maxdif=max(dif) ;**
   est déjà beaucoup plus lent !