PETITE INTRODUCTION AU LOGICIEL SCILAB

FRANÇOIS DUCROT

Ce petit document est conçu pour être utilisé par quelqu'un qui fait les expérimentations sur machine au fur et à mesure qu'elles sont proposées dans le texte.

1. Généralités

Scilab est un clône du logiciel commercial Matlab. Il est développé par l'INRIA et l'ENPC et est distribué gratuitement sur Internet sous une licence Open Source. Il s'agit d'un logiciel d'analyse numérique, destiné à traiter des (gros) tableaux de nombres, écrits en virgule flottante (le nom matlab vient de MATrix LABoratory). Il est muni d'un langage de programmation interprété (un script est lu ligne par ligne par la machine, qui exécute au fur et à mesure chaque ligne), mais peut aussi inclure des sous-programmes écrits en fortran ou en C, et transformés par un compilateur en un programme directement exécutable par la machine. Les sous-programmes ainsi compilés s'effectuent bien plus rapidement que des scripts interprétés ; cependant, on se bornera ici à l'utilisation de petits scripts interprétés.
La version de scilab actuellement installée dans les salles d'enseignement est la 5.2.2, et le système d'exploitation est Linux Fedora 13. Vous pouvez télécharger gratuitement scilab pour Windows/Linux/MacOS sur le site `http://www.scilab.org`.

2. L'interface du logiciel

On décrit dans cette section les différentes fenêtres que Scilab fait apparaitre à l'écran, et la façon dont l'utilisateur interagit avec le logiciel.

2.1. **L'interpréteur de commande.** Quand on lance scilab, la fenêtre qui apparait est un interpréteur de commande. Le logiciel affiche un signe d'invite (`>`), en face duquel on tape une ligne de commandes. A chaque fois qu'on presse la touche entrée le logiciel évalue les instructions contenues dans la ligne courante, et affiche soit un message d'erreur, soit le résultat des calculs. Voici quelques exemples :

```
> 1+1
> a=2+1
> b=5 ;
> a * b
> M=[1,2;3,4]
> det(M)
> S=0 ; for i=1:1000, S=S+i; end
> S
> 1==2
> sin(%pi/4)
> sqrt(1+5*%i)
```

On notera que lorsqu'on fait suivre une commande par un point-virgule, la commande est effectuée, mais le résultat du calcul n'est pas affiché à l'écran ; ça peut être utile si le résultat est une matrice 100×100.

Date: Septembre 2010.

Il est important de garder en tête que la fenêtre est uniquement un interprèteur de commande, et en aucun cas un editeur de texte. Donc quand une commande est effectuée on ne peut pas revenir en arrière. Par contre on dispose d'une fonction de rappel de commande (la touche "flèche vers le haut") qui permet de faire reapparaitre toutes les commandes précédemment tapées.

2.2. **L'aide.** On peut obtenir de l'aide sur une fonction en tapant par exemple :
`> help intg`
ce qui demande bien entendu de connaitre le nom de la fonction recherchée. En cliquant sur le menu help, on fait apparaitre une fenêtre d'aide, qui permet une recherche plus intuitive. Pensez toujours à regarder les quelques exemples des pages d'aide, qui sont souvent plus explicites que le texte. A titre d'exemple cherchez, en utilisant l'aide, comment dessiner le graphe de la fonction $[0, 10] \longrightarrow \mathbb{R}, x \mapsto \sin(x)$.

2.3. **La fenêtre graphique.** Les graphiques sont affichés dans une fenêtre séparée. Ainsi :
```
> x=[0:.1:2]; z=x'*x;
> xbasc() , plot3d(x,x,z)
```
Scilab écrit par défaut tous les graphiques dans une seule fenêtre graphique, sans effacer la fenêtre au préalable. La commande `xbasc()` permet d'effacer le contenu de la fenêtre graphique courante. Les différents menus de la fenêtre graphique permettent d'agir sur son contenu (zoom, rotation), ou de le sauvegarder par exemple dans un fichier postscript (menu File/Export).

Il est également possible de passer des paramètres au moteur d'affichage par l'intermédiaire de la ligne de commande par la commande `xset`. Ainsi la ligne `xset('window',1)` crée une nouvelle fenêtre graphique (de numéro 1) et envoie les ordres d'affichage suivants vers cette fenêtre, ce qui permet d'avoir simultanéments plusieurs graphiques affichés. On peut employer aussi cette commande sans argument en tapant `xset()`, ce qui fait apparaitre une fenetre de réglage graphique.

On peut aussi afficher plusieurs graphiques à l'intérieur d'une seule fenêtre grâce à `xsetech` comme dans l'exemple :
```
> x=linspace(0,%pi,50); y1=sin(x) ; y2=%e^(-x); y3=y1.*y2;
> xbasc() , xsetech([0 0 .5 .5]), plot2d(x,y1) , xtitle("sin(x)")
> xsetech([.5 0 .5 .5]), plot2d(x,y2) , xtitle("exp(-x)")
> xsetech([0 0.5 1 .5]), plot2d(x,y3) , xtitle("exp(-x) sin(x)")
```

3. Les variables

Tout ordre dans scilab est un ordre d'affectation de variable. Ainsi la commande `a=1` crée, si elle n'existe pas déja la variable `a` et lui affecte la valeur 1 . De même la commande `1+1` affecte à la variable système `ans` la valeur obtenue en faisant le calcul `1+1`. Enfin la commande`a=1+1;` affecte à `a` le résultat du calcul, mais n'affiche pas ce résultat à l'écran (c'est utile si le résultat est une matrice 100×100)

On peut voir à tout moment quelles sont les variables déclarées en tapant la commande `> who`. On voit alors apparaître d'une part les variables définies par l'utilisateur, et d'autre part des variables définies dans le système. On trouve ainsi les variables %pi, %e, %T, %F, %i, etc.

Chaque variable a un type (matrice de nombres, matrice booléenne, fonction, etc), et on peut connaitre le type du variable `a` par la commande `type(a)` (regarder l'aide au sujet de la commande `type`, pour voir les tifférents types de varibles reconnus par scilab). Décrivons dans la section suivantes quelques types de données.

4. Les types de données

4.1. **Les nombres.** Scilab manipule des nombres en virgule flottante, écrits sous la forme $a_0, a_1 \dots a_k \,.\, 10^{\pm n}$. Par défaut il affiche 7 chiffres après la virgule, mais les calculs sont effectués avec 16 chiffres significatifs (changement de l'affichage avec la commande `format`). Les expérimentations suivantes aident à comprendre comment travaille scilab

```
> a= 57895*10^150
> b=7*10^200
> a/b
```

On voit donc qu'on peut manipuler des nombres très grands ou très petits, dans des limites raisonnables cependant :

```
> 10^400
> 10^(-400)
```

On peut aussi calculer avec des complexes ; dans l'exemple suivant, on voit que scilab a calculé une racine dixième d'un complexe, mais il en a choisi une de manière arbitraire parmi les dix possibles, ce qui nécessite donc une certaine prudence dans l'utilisation :

```
> (1+5*%i)^(1/10)
```

Quand on effectue des additions, il faut tenir compte de la précision machine :

```
> c=1+10^(-20)
> d=10^(-20) +10^(-33)
> c-1 , d-10^(-20)
```

Expliquez pourquoi scilab a répondu 0 dans un cas et pas dans l'autre. Signalons que scilab a une constante machine `%eps`, qui est le plus grand nombre ε tel que $1 + \varepsilon = 1$. Enfin comment expliquez vous le calcul matriciel suivant ?

```
> M=[1,2,3;4,5,6;7,8,9]
> det(M)
> inv(M)
```

On voit donc ici que quand on travaille avec des nombres en virgule flottante, il n'y a souvent aucune raison de distinguer un nombre très petit (comme 10^{-16}) de 0.

4.2. **Les matrices.** Les objet pricipaux que manipule Scilab sont les matrices et les vecteurs de nombres en virgule flottante ; un nombre n'étant pour Scilab qu'une matrice de taille 1×1. On donne ici quelques exemples de ce qu'on peut faire avec les matrices. Pour voir d'autres possibilités, regardez l'aide de scilab dans la rubrique "linear algebra".

4.2.1. *Fabriquer une matrice.* Voici quelques exemples:

```
> [1,2,3;4,5,6;7,8,9]          matrice définie coefficient par coefficient
> rand(5,5)          matrice aléatoire
> [1:5]          vecteur ligne des nombres entre 1 et 5 par incrément de 1
> [1:.07:2]          vecteur ligne des nombres entre 1 et 2 par incrément de 0.07
> a=eye(3,3), b=eye(3,4)          matrice identité (pas forcément carrée !)
> b=zeros(3,2)          matrice nulle
> c=ones(2,3)          matrice remplie de 1
> a=[1 2 3 4]; b=diag(a) , c=diag(a,1)          matrices diagonales
```

On peut utiliser ces différentes briques pour construire des matrices plus grandes par concaténation :

```
> a=eye(3,3), b=[1 2 3], c=[1;2;3;4], d=[a;b] , e=[d, c]
```

4.2.2. *Extraire une partie d'une matrice.* Donnons nous une matrice

```
> M=rand(6,5)
```

On peut alors en extraire différents morceaux

```
a=M(2,2)
B=M([1 3;4 5])
L=M(3,:)        troisième ligne
C=M(:,$)        dernière colonne
```

4.2.3. *Modifier une matrice.* Partons d'une matrice
`> M=rand(6,5)`
on peut modifier un coefficient :
`> M(2,2)=0`
intervertir deux lignes
`> M([1,2],:)= M([2,1],:)`
ajouter à la deuxième colonne 3 fois la première
`> M(:,2)= M(:,2)+3* M(:,1)`
effacer la dernière ligne
`> M($,;)=[ ]` , et bien d'autres choses encore.
Attention : Quand on tape une commande du type `truc(2,3)=5`, scilab peut avoir deux comportement distincts : soit truc existait déja en tant que matrice, et il donnera la valeur 5 à son coefficient en position (2,3), soit truc n'existait pas auparavant et il créera une matrice truc de taille (2,3), dont un coefficient sera 5 et les autres zero. Ce procédé de construction est très rapide, mais on ne maîtrise pas vraiment ce qu'on fait (qui sait si truc n'existait pas avant). Il vaut donc mieux créer truc par une commande du type `truc=zeros(2,3)` et ensuite définir ses coefficients.

4.2.4. *Opérations sur les matrices.* Les opérations qu'on peut faire subir aux matrices sont très nombreuses. Commençons par définir quelques matrices
`> x=[0:.2:10], y=[1:51], A=rand(51,51), M=[1 1+%i 1+2*%i ; 2 2+%i 2+2*%i]`
On peut vouloir connaitre la taille de ces matrices (utile en programmation)
`> size(x) , size(A) , size(M,1) , size(M,2)`
Matrice transposée et matrice adjointe (conjuguée et transposée) : `M' , M.'`
On peut appliquer une fonction à chaque terme d'une matrice
`> sin(x) , sqrt(x)`
On dispose bien entendu des opérations matricielles élémentaires ; on distinguera entre `*` (multiplication matricielle) et `.*` (multiplication coefficient par coefficient) .
`> x+y, 2*x , x.*y , x'*y`
Enfin, résolvons un système linéaire :
`> b=A*x'; z=A\b;`
`> norm(z-x')` permet de mesurer l'erreur commise
La plupart des opérations matricielles usuelles aux quelles vous pouvez penser, ainsi que beaucoup d'autres, sont implémentées dans scilab; le chapitre "linear algebra" de l'aide vous en donnera une liste.

L'appel aux fonctions internes de scilab utilisant des matrices est beaucoup plus rapide que les procédures que vous pourriez écrire comme le montre l'exemple suivant :
```
> timer()
> sum(1:10000)
> timer()
> S=0; for i=1:10000, s=s+i ; end ; S
> timer()
```
Ceci est du au fait que les fonctions internes de scilab sont compilées et non interprétées, et

qu'elles manipulent les matrices d'un coup (on parle de fonction vectorielles) et non coefficient par coefficient. Un principe à en déduire est qu'il faut vectorialiser au maximum les calculs.

4.3. **Les booléens.** Les valeurs booléennes sont vrai (`%T`) et faux (`%F`). Ces valeurs sont généralement obtenues comme résultat d'un test

```
> 1==3 , 1<>3 , 1<3 , 1<=3
```

Les booléens peuvent être reliés par des opérateurs logiques :

```
> %F | %T , %F & %T , ~%F , 1<2 | 4<4
```

Les expressions booléennes seront utilisées pour des tests en programmation. On peut aussi les utiliser pour trouver les coefficients d'un vecteurs qui satisfont une condition booléenne

```
> a= 1:2:12
> b=find(a>3)        indices des coefficients de a qui sont > 3
> a(b)        vecteur composé des coefficients de a qui sont > 3
```

Regarder l'aide au sujet de `find` pour plus d'applications.

4.4. **Les fonctions.** Une fonction en mathématiques est définie par deux expressions comme dans l'exemple $f : \mathbb{R}^2 \longrightarrow \mathbb{R}\,,\ (x, y) \mapsto x^2 + y^3$. La définition d'une fonction dans scilab est tout à fait analogue, comme dans la commande suivante qui affecte à la variable "mafonction" la fonction en question :

```
> deff('z=mafonction(x,y)','z=x^2+y^3')
> mafonction(3,4)
```

On constate que pour définir une fonction, on utilise deux expressions (mises entre apostrophes), la première donnant le nom de la fonction et de ses arguments, et la deuxième donnant la formule qui la calcule. Une fois qu'on a créé une fonction, on peut y faire appel comme à n'importe quelle autre variable, comme dans les exemples suivants :

```
> deff('z=f(x)','z=x^2')
> deff('z=eval2(g)','z=g(2)')
> eval2(f)
> x=1:2:10 ; feval(x,f)
```

Remarquez ici l'usage de la fonction `feval` pour appliquer terme à terme une fonction à un vecteur ; elle permet une programmation efficace et rapide, et sera utilisée pour tracer des graphes de fonctions.

5. Importer ou exporter des données

On pourrait envisager de ne travailler avec scilab que de façon interactive en tapant toutes les commandes dans l'interprèteur de commandes, mais ça se révèlera très vite inconfortable. On préferera souvent taper des fichier texte contenant des listes d'instructions à l'aide d'un éditeur de textes puis les importer dans scilab. L'éditeur de texte emacs qui vous est proposé sous linux est parametré pour traiter au mieux les fichiers de code emacs, en fournissant en particulier des possibilités de coloration syntaxique et de complétion automatique (à condition que les fichiers aient une xetension .sci ou .sce).
Attention à bien utiliser un éditeur de texte (bloc-note sous Windows, emacs, ...) et surtout pas un traitement de texte (word, ...), qui introduirait des caractères parasites. On peut procéder de différentes façons :

5.1. **Définir des fonctions par `getf`.** Plutot que de définir la fonction `mafonction` comme précedemment, créons grâce à un éditeur de textes le fichier "truc.sci" contenant les trois lignes suivantes :

```
function z=lamemefonction(x,y)
   z=x^2+y^3
```

```
endfunction
```
demandons ensuite a scilab de lire ce fichier (par la commande `getf("chemin-vers-truc.sci")`, où "chemin-vers-truc.sci" désigne le chemin d'accès complet vers le fichier truc.sci (par exemple D:\ truc.sci), ou en utilisant une option du menu "file" ("File operations" ou "getf" suivant les systèmes). Scilab sait alors calculer `lamemefonction(3,4)`. Le fichier truc.sci aurait pu contenir plusieurs définitions de fonctions. Dans ce cas elles doivent être séparées par une ligne blanche, qui fait office de délimiteur. Une façon de travailler peut être de créer toutes ses fonctions dans un seul fichier texte, qu'on importera après chaque modification (ce n'est pas fait automatiquement). On prendra garde que chaque définition de fonction, même si il n'y en a qu'une, doit être suivie d'un retour à la ligne et d'une ligne blanche ; c'est cette ligne qui indique à la machine que la définition de la fonction est complète.

5.2. **Exécuter un script avec `exec`.** Un script est un ensemble d'instructions rassemblées dans un fichier texte. Créons ainsi le fichier texte "monpremierscript.sce" comprenant uniquement la ligne suivante :
```
for i=1:5 , i^2, end
```
La commande `exec("monpremierscript.sce")` exécute le script en question (il peut etre plus simple d'appeler le script à l'aide du menu "file/File operations"). Heureusement on peut faire des scripts moins ininteressants. Le script suivant demande à l'utilisateur de rentrer les coefficients a, b, c d'un polynôme du second degré et en trace le graphe sur l'intervalle $[0, 1]$:
```
// un script passionnant
a=input(" Rentrer la valeur de a :  ");
b=input(" Rentrer la valeur de b :  ");
c=input(" Rentrer la valeur de c :  ");
x=[0:.01:1] ;
y=a*x.*x+b*x+c*ones(x);
plot2d(x,y)
```
Notez ici l'usage de // pour commencer une ligne de commentaires.

5.3. **Fichiers de données avec `write` et `read`.** Le résultat d'un calcul, par exemple une grosse matrice, peut être sauvegardé dans un fichier texte, puis chargé quand on en a besoin, comme le montre l'exemple suivant :
```
> a=rand(50,50);
> write('truc.dat',a)
> a=zeros(1)
> a=read('truc.dat',2,2)
a =
!  0.2113249 0.4094825 !
!  0.4256872 0.2659857 !
```
Ici, la variable a contient maintenant la sous-matrice principale de taille 2x2, du résultat de `a=rand(50,50)`.

5.4. **Remarque sur les formats de fichiers.** Scilab recommande d'utiliser les suffixes ".sci" pour les fichiers de définitions qu'on charge avec "getf", ".sce" pour les scripts qu'on éxécute avec "exec", et ".dat" pour les fichiers de données qu'on lit avec "read". Ce n'est pas impératif, et c'est heureux puisque l'éditeur de texte "bloc-note" de windows enregistre systématiquement ses fichiers avec le suffixe ".txt".

6. Programmation

Scilab permet d'écrire des programmes interprètés, en utilisant une syntaxe proche de celle de pascal ou C. Quand on écrit un tel programme, il est recommendé d'utiliser à bon escient des retours à la ligne et des indentations, qui rendent le programme plus lisible et plus facile à corriger. Cependant, dans le cas d'une utilisation interactive, on peut remplacer les retours à la ligne par des virgules.

6.1. **La boucle for.** La syntaxe générale d'une boucle `for` est

```
for i=vecteur
   suite-d-instructions
end
```

L'indice i prend successivement la valeur de chaque coordonnée du vecteur. La fin de la liste d'instruction est marquée par le `end`. Donnons un exemple trivial :

```
> s=1;
> for i=1:10, s=s*i; end;
> s;
```

6.2. **La boucle while.** Une variante de ce qui précède est la boucle `while` (tant que), dont la syntaxe est

```
while condition
   suite-d-instructions
end
```

6.3. **Les instructions conditionnelles.** On utilisera essentiellement des constructions `if then else`, dont la syntaxe est la suivante

```
if condition-principale then
   suite-d-instructions
elseif autre-condition then
   autre-suite-d-instructions
else
   encore-des-instructions
end
```

Les conditions sont données par des opérateurs booléens. Il peut avoir un nombre quelconque (éventuellement nul) de `elseif`, mais au plus un seul `else`.

6.4. **Appels récursifs.** Une fonction peut s'appeler elle même. l'exemple le plus classique est la définition de la factorielle :

```
function f=fact(n)
  if n<=1 then
    f=1
  else
    f=n*fact(n-1)
  end
endfunction
```

Ceci donne une programmation élégante, mais souvent inefficace en termes d'occupation mémoire. C'est en pratique peu utilisé pour du calcul numérique, travaillant avec des données de grande taille.

6.5. **Débogage.** Il faut quelques fois faire la chasse aux erreurs de programmation. Une méthode consiste à introduire aux endroits stratégiques d'une procédure des instructions `pause`. Lors de l'exécution d'une telle procédure, scilab s'arrête quand il rencontre une pause et vous avez la main pour demander à scilab d'afficher les valeurs des différentes variables à cet instant de l'éxécution du programme ; vous pouvez ensuite dire à scilab de repartir avec l'instruction `resume`. Il repartira alors jusqu'à la prochaine pause, comme si rien ne s'était passé pendant l'interruption.

7. Afficher des graphes

7.1. **Graphes 2D.** Un graphe en dimension 2 est un ensemble (fini) de points (x_i, y_i). La commande qui permet de le dessiner est

```
> plot2d(x,y)
```

où `x` (resp. `y`) est le vecteur des x_i (resp. y_i). Les exemples suivants permettent de tester quelques possibilités :

```
> x=0:.1:10 ; y1=sin(x) ; y2=sin(2*x) ; y3=sin(3*x);
> plot2d(x,y1)
> plot2d(x,y2)
> plot2d(2*x,2*y2)
> xbasc()
> plot2d(cos(x),sin(x))
> xbasc()
> plot2d([x' x'],[y1' y2'])
> xbasc()
> plot2d([x' x' (2*x)'],[y1' y2' (2*y2)'])
> xbasc()
> plot2d([x' x' (2*x)'],[y1' y2' (2*y2)'],[-2,1,6],'151','C1@C2@C3',[0,-2,10,2])
```

Ici [-2,1,6] désigne le style (croix ou couleurs) affecté à chaque courbe. '151' signifie qu'il y a une légende (1er 1), que les limites de la représentation seront fixées dans un autre argument (5), et que les axes sont gradués (2ème 1). 'C1@C2@C3' définit les légendes et le dernier argument donne les limites du graphique.

Consulter l'aide pour trouver toutes les ressources de `plot2d`.

Pour dessiner le graphe d'une fonction, on a plusieurs façons de procéder.

```
> deff('y=f(x)','y=%e^(-x)*sin(x)')
> x=[0:.1,3];
> y=feval(x,f); plot2d(x,y)
> xbasc() , fplot2d(x,f)
```

La fonction `fplot2d` utilisée pour dessiner le graphe d'une fonction est en réalité un script écrit en scilab, qui appelle la fonction `feval`.

7.2. **Graphes 3D et fonctions de deux variables.** De même, la commande `plot3d(x,y,z)`, où `x` (resp. `y`) est un vecteur de taille m (resp. n), et `z` est une matrice de taille $m \times n$ trace une représentation en perspective de l'ensemble des $(x_i, y_j, z_{i,j})$. On notera que, contrairement au cas du dessin en dimension 2, plus le nombre de points dessinés est grand, moins le graphe est lisible. Voici une suite d'expérimentations :

```
x=-% pi:.05:% pi; y=x; z=sin(x)'*ones(y)+ones(x)'*sin(y);
xbasc(); plot3d(x,x,z)
x=-% pi:.25:% pi; y=x; z=sin(x)'*ones(y)+ones(x)'*sin(y);
xbasc(); plot3d(x,x,z)
xbasc(); xset('colormap',hotcolormap(64)) ; plot3d1(x,x,z)
```

`xset('window',2) , xset('colormap',hotcolormap(64)), grayplot(x,x,z)`
`contour2d(x,x,z)`
La commande "`xset('colormap',hotcolormap(64))`" a été utilisée dans cet exemple pour choisir la palette de couleurs utilisée, en l'occurence une palette de 64 couleurs allant du rouge noir (froid) au jaune blanc (très chaud).

Il existe également une commande `champ` destinée à afficher des champs de vecteurs (représentés par des flèches). Chacune des commandes précédentes a un analogue dont le nom commence par un "f" (`fplot3d`, `fcontour`, `fcontour2d`, `fgrayplot`, `fchamp`, ...), et qui prend pour argument une fonction. Cependant il est souvent plus efficace d'utiliser au maximum les opérations de type matricielles.

Exercice : Utilisez au mieux les capacités graphiques de scilab pour faire comprendre la géométrie du graphe de la fonction

$$(x, y) \mapsto \frac{2x^2y + y^2}{x^2 + 2y^2}$$

8. Un exemple illustratif : la méthode du pivot

Soient A une matrice carrée $n \times n$ et b un vecteur de taille $1 \times n$, on veut résoudre l'équation $A.x = {}^t b$ par la méthode du pivot partiel. Pour s'entrainer à écrire un programme, il est bon de commencer à écrire des programmes plus petits, ne réalisant qu'une partie de ce qui est demandé, puis de les intégrer pour obtenir le programme définitif :

- Ecrire une fonction, qui prend comme argument une matrice A et la met sous forme triangulaire supérieure en effectuant des opérations élémentaires sur les lignes, en prenant pour pivots siccessifs les termes sur la diagonale.
- Ecrire une fonction prenant A et b comme arguments, qui vérifie qu'is sont de tailles adéquates (fonction `size`), puis résoud le le système par la méthode du pivot, toujours en prenant naivement les pivots $a_{11}, a_{22}, \cdots$.
- Raffiner cette méthode en choisissant pour j-ième pivot le coefficient le plus grand de la j-ème colonne (méthode du pivot partiel), afin de minimiser les erreurs d'arrondi.

Voici un exemple de script qui fait ce qui est demandé, mais il est important que vous le trouviez par vous même :

```
function x=sol(A,b)
// Resolution d'un syteme lineaire carre par pivot partiel.
[n,m]=size(A)
[l,k]=size(b)
// on fait d'abord des tests de consistance
if n<>m then
   error("matrice non carree ")
end
if (l<>n | k<>1) then
   error("tailles non compatibles")
end
// On rend ensuite le système triangulaire
a=A
for j=[1:n-1]
   npivot=j
   pivot=a(npivot,j)
   for i=[j+1:n]
     if abs(a(i,j))>abs(pivot) then
       npivot=i
       pivot=a(i,j)
     end
   end
   a([j,npivot],:)=a([npivot,j],:)
   b([j,npivot])=b([npivot,j])
   for i=[j+1:n]
     m=a(i,j)/pivot
     a(i,:)=a(i,:)-m*a(j,:)
     a(i,j)=0
     b(i)=b(i)-m*b(j)
   end
end
// On résoud enfin le sytème triangularisé
x=zeros(n,1)
x(n)=b(n)/a(n,n)
for i=n-1:-1:1
   x(i)=(b(i)-a(i,i+1:n)*x(i+1:n))/a(i,i)
end
```