Chapitre S1 : Introduction à Scilab et applications aux tracés

Une remarque introductive

Le logiciel SciLab a une syntaxe assez proche de Python en bien des points. Il y a aussi bien des différences. Un avantage pour nous est l'aide en ligne très complète, une gestion plus légère sans avoir à charger des bibliothèques, des fonctions mathématiques et graphiques.

L'aide s'obtient avec `help` ou `help plot` (sans parenthèses[1]), si on veut de l'aide sur la fonction `plot`. En outre sur les machines du lycée des fichiers d'aide sont à disposition. Ouvrir firefox et accéder à l'aide en ligne : `http://192.168.numero_de_la_salle.1` Par exemple `http://192.168.18.1`

Table des matières

1. une différence avec Python

1 Un paradigme : en Scilab tout est vecteur ou matrice...

Le logiciel SciLab(pour *Scientific Laboratory*) a été développé par l'INRIA comme une alternative libre à Matlab. Or Matlab signifie *Matrix Laboratory*, donc ces deux logiciels font des vecteurs et des matrices (tableaux) leurs objets de base.

1.1 Comment rentrer un vecteur ou une matrice ?

A la main :

Entre crochets `[ ]`, dans une ligne les entrées sont séparées par des virgules, les lignes sont séparées par des points virgules.

```
-->u=[1,2,3] // vecteur ligne
 u  =

    1.    2.    3.

-->u=[1; 2;3] // vecteur colonne
 u  =

    1.
    2.
    3.
 -->A=[1,2,3; 4,5,6]
 A  =

    1.    2.    3.
    4.    5.    6.
```

Remarque : L'opération de *transposition* qui transforme les lignes en colonnes et inversement, se note avec une *prime*. Par exemple :

```
-->u=[1,2,3]
 u  =

    1.    2.    3.

-->u=u'
 u  =

    1.
    2.
    3.
```

Plus automatique :

```
-->u=0:0.1:1 // de 0 à 1 avec un pas de 0.1
 u  =

    0.    0.1    0.2    0.3    0.4    0.5    0.6    0.7    0.8    0.9    1.
 -->u=linspace(0,1,11) // subdivision régulière à 11 points de l'intervalle [0,1]
 u  =

    0.    0.1    0.2    0.3    0.4    0.5    0.6    0.7    0.8    0.9    1.
```

1.2 Les réels sont des tableaux 1×1 :

Par exemple que répond SciLab à :

```
-->x=2; // le point virgule empêche l'affichage
-->x==[2] // test d'égalité comme en Python
```

1.3 Les variables en mémoire :

Pour voir la liste des variables utilisées

La commande `who` ou mieux `who_user` donne la liste des variables en mémoire.

Pour effacer

```
-->u=2
-->clear('u')// vide le contenu de u
-->clear // efface toutes les variables (non protégées) que nous avons déclarées.
```

Remarque sur les variables prédéfinies et protégées :

On a déjà rencontré les variables mathématiques `%e,%pi %i` .

Elles sont toutes précédées d'un %. Ce signe % signifie que ces variables sont *protégées* i.e. résistent à la commande `clear`.

La commande `predef()` permet de gérer ces variables prédéfinies, nous ne l'utiliserons pas forcément.

1.4 Pour accéder aux entrées d'un vecteur ou d'une matrice :

Attention deux différences avec Python :
- parenthèses et pas crochets,
- la numérotation commence à 1 et pas à 0.

Pour les tableaux, toujours indice Ligne puis indice Colonne (notation standard en math : LiCo).

```
-->A=[1,2,3;4,5,6]
 A  =

    1.    2.    3.
    4.    5.    6.

-->A(1,2)
 ans  =

    2.

-->A(2,2)
 ans  =
```

1.5 Ce qu'on ne pouvait pas faire avec une liste Python

On peut créer un vecteur `u` en définissant les `u(n)`

```
for n=1:50
    u(n)=(-0.9)^n
end
```

2 Les opérations et fonctions sur les vecteurs

2.1 Les opérations usuelles sur les vecteurs et matrices

Les deux lois évidentes d'espace vectoriel

Pas de surprise pour `+` : ajoute entrée par entrée des vecteurs (resp. tableaux) de même taille.
Pas de surprise pour la multiplication par un scalaire notée `*`

Deux notions distinctes de multiplication entre vecteurs, matrices

a) La multiplication *entrée par entrée* entre deux vecteurs (tableaux) de même taille se note `.*` attention : il y a un point avant l'étoile ! Par exemple :

```
->u=[1,2,3]
 u  =

    1.    2.    3.

-->v=[-1,2,1]
 v  =

  - 1.    2.    1.

-->u*v
    !--error 10
Multiplication incohérente.
 -->u.*v
 ans  =

  - 1.    4.    3.
```

b) Le symbole `*` seul correspond à la *multiplication des matrices* que nous étudierons en maths

Nous aurons besoin ici seulement de l'exemple de la multiplication à droite d'un tableau (matrice) par un vecteur colonne :

Idée expliquée en 2×2 : à partir d'un tableau $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$ et d'un vecteur colonne $\begin{pmatrix} x \\ y \end{pmatrix}$, on note :

$\begin{pmatrix} a & b \\ c & d \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \stackrel{def}{=} \begin{pmatrix} ax_1 + bx_2 \\ cx_1 + dx_2 \end{pmatrix}$.

Ceci permet p. ex. d'écrire le système $\begin{cases} ax_1 + bx_2 = y_1 \\ cx_1 + dx_2 = y_2 \end{cases}$ sous la forme $\begin{pmatrix} a & b \\ c & d \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}$.

2.2 Les fonctions usuelles opèrent sur les vecteurs :

Attention aux points à mettre devant les multiplications, divisions, puissances...

Parfois ce point n'est pas nécessaire, mais dans le doute mieux vaut le mettre ! Pas de . en revanche pour appliquer `sin`, `log` etc...

```
->v=[1,2,3]; // le point virgule empêche l'affichage

-->v.^2
 ans  =

    1.    4.    9.
```

```
-->u=[%pi,0, %pi/2]
 u  =

    3.1415927    0.    1.5707963

-->sin(u)
 ans  =

    1.225D-16    0.    1.
```

Eh oui SciLABfait du calcul *numérique*!

2.3 Un autre paradigme : le calcul *numérique* et le ε-machine

Comme vu au dernier exemple, en SciLab, $\sin(\pi)$ ne donne pas 0, mais `1.225D^-16`. Pour tester si quelque chose peut être considéré comme nul, on peut le comparer avec l'écart le plus petit entre deux flottants [2], appelé ε-machine en informatique stocké dans la variable `%eps` en SciLab.

```
-->%eps
 %eps  =

    2.220D-16

-->sin(%pi)<%eps
 ans  =
   T
```

3 Définition et manipulations de fonctions en Scilab

3.1 La déclaration de fonction dans l'éditeur de texte

La syntaxe pour la définition de fonctions en SciLab est la suivante :

```
function y=MaFonction(paramètres); // ce point virgule là est obligatoire
    instructions; // ceci est un commentaire
    instructions;  // les ; permettent d'éviter un affichage
    y=...
endfunction
```

Par exemple :

```
function y=f(x);
    y=sqrt(x.^2+1)
endfunction
```

Ensuite *exécuter* la fonction pour qu'elle soit utilisable dans le *shell* de SciLab.

N.B. Ma fonction `f` peut opérer sur un vecteur (ou un tableau) `x` et dans ce cas renverra `y` vecteur (resp. tableau) de mêmes taille. S'il y a plusieurs variables arguments et plusieurs variables de sortie, la syntaxe est :

```
function [x,y]=g(u,v);
    x=u+v
    y=u*v
endfunction
```

2. revoir le chapitre 6

3.2 Une autre possibilité de déclaration sur une ligne

```
-->deff('y=f(x)','y=sqrt(x.^2+1)')
```

On peut aussi appliquer la fonction à des couples, même avec cette façon de la définir :

```
-->deff('[x,y]=g(u,v)',['x=u+v','y=u*v'])
```

C'est économique mais la syntaxe n'est pas forcément légère avec les *quotes*.

4 Au propos de l'affichage graphique

4.1 L'essentiel sur ce qui fait `plot` : affichage à partir de deux vecteurs

Aveu : je n'ai pas compris toutes les différences `plot/plot2d`

En fait `plot2d` a plus d'options que `plot`, qui lui, permet d'écrire des scripts compatibles avec MatLab ?

L'utilisation en est la même, tant qu'on ne modifie pas les options d'affichages !

Remarque préliminaire : *Il existe plusieurs façons d'utiliser* `plot`*, qui font qu'il n'est pas forcément facile de dégager d'emblée une logique commune. Je vais partir du point de vue qui me semble le plus commode parce qu'il s'applique à des situations très générales.*

L'essentiel : Comme vu au T.P. 6, il faut penser que la commande `plot` de SciLab prend comme arguments deux vecteurs de même taille, disons `x` et `y`, place les points de coordonnées `(x(i),y(i))`, les points successifs étant, par défaut, reliés par des segments.

Ce point de vue permet de tracer beaucoup de choses : pas *que* des graphes de fonctions !

4.2 Premières options d'affichage

4.2.1 Affichage par défaut : points noirs reliés par des segments noirs

```
x=[0, 1,2];
y=[1,2,1];
plot(x,y)
```

4.2.2 Pour modifier : points séparés ou pas, couleurs..

a) **Pour la commande** `plot` : on rajoute comme arguments supplémentaires dans `plot` avec des guillemets :
   - Les couleurs simples : "b"=blue, "r"=red,"g"=green,"y"=jaune,"m"=magenta,"w"=white,
   - Les formes de points : par défauts les points sont reliés, sinon, on met : ".","+","o","x","*".

```
x=0:0.1:%pi/2
y=sin(x)
plot(x,y,"r+") \\ points rouges (r) en forme de +, on pourrait mettre  "g*", "yx" etc...
```

b) **Pour la commande** `plot2d` : on utilise plutôt les *numéros* des couleurs et des marques. Ces numéros sont donnés dans des tableaux `getcolor()` et `getmark()`. La commande `color()` donne aussi le numéro d'une couleur donnée par son nom :

```
-->color("red")
 ans  =

    5.
```

Dans `plot2d`, les trois commandes suivantes sont équivalentes :

```
plot2d(x,y,5)
plot2d(x,y,color("red")
plot2d(x,y,style=[5])
```

Sachant que les marques (+,*,o) sont données par un nombre négatif, on peut l'équivalent du a) avec `plot2d` :

```
x=0:0.1:%pi/2
y=sin(x)
plot2d(x,y,[5,-1])\\points rouges en forme de +.
```

4.3 Pour tracer des graphes de fonctions

4.3.1 Avec les deux vecteurs (le point de vue qui marche toujours)

```
clf; // pour clear figure : efface la figure courante
x=linspace(-%pi,+%pi,100);
y=x.^2+x+1
plot(x,y)
```

4.3.2 Le second vecteur peut être calculé directement dans l'appel de `plot`

Comme dit dans le titre, un script équivalent au précédent est :

```
clf; // pour clear figure : efface la figure courante
x=linspace(-%pi,+%pi,100);
plot(x,x.^2+x+1)
```

4.3.3 On peut remplacer le second vecteur par le nom de la fonction

Si on a définit une fonction $f : x \mapsto x^2 + x + 1$ en SciLab, on peut aussi tracer ainsi :

```
clf; // pour clear figure : efface la figure courante
x=linspace(-%pi,+%pi,100);
plot(x,f)
```

Autrement dit : quand le second argument de `plot` est une fonction et pas un vecteur, il comprend qu'il doit tracer `x,f(x)`.

4.4 Des fonctions particulières : les suites

On a vu plus haut que pour une suite définie explicitement, on peut utiliser une simple boucle `for`

```
for n=1:50
    u(n)=(-0.9)^n
end
```

Remarque : les opérations sur les vecteurs permettent de court-circuiter l'écriture de la boucle for On peut ainsi définir `u` plutôt comme suit :

```
u=(-0.9)^(1:50)
```

On peut alors tracer sans surprise :

```
N=1:50
plot(N,u,"*r")
```

Mais, en fait, pour un vecteur ayant n entrées, le premier argument peut ici être enlevé :

Par défaut, le premier argument de `plot` appliqué à un vecteur de longueur n sera le vecteur `1 :n` : c'est donc très pratique pour visualiser des suites.

```
for n=1:50
    v(n)=sin(n)
end
plot(v,"*b")
```

4.5 Pour tracer des *courbes paramétrées* planes

Définition : Tracer une *courbe paramétrée plane* est tracer l'ensemble des points $M(t) = (x(t), y(t))$ où $t \mapsto x(t)$ et $t \mapsto y(t))$ sont des fonctions quelconques.

L'essentiel : la variable t ne se voit pas sur la figure. On peut penser que c'est le temps.

4.5.1 Un premier exemple de tracé :

Par exemple le mouvement d'un mobile $M(t)$ en fonction du temps t est défini par $\begin{cases} x = \cos(3t), \\ y = \sin(3t) \end{cases}$: mouvement circulaire uniforme.

Pour afficher la trajectoire avec `plot` (ou `plot2d`) :

```
t=linspace(0,2*%pi,100)
x=cos(3*t)
y=sin(3*t)
plot(x,y)
```

Problème : on voit un ovale au lieu d'un cercle : voir § 4.6.2 ci-dessous

4.5.2 Un enjeu pour les courbes paramétrées : savoir suivre la courbe

- **Propriétés de symétries :** *voir notes manuscrites*
- **Vecteurs tangents et asymptotes :** *voir notes manuscrites*

4.6 Commandes à connaître pour gérer les graphiques :

4.6.1 Gestion des fenêtres des figures

Au premier appel d'un `plot` (ou autre), il y a création d'une fenêtre graphique, appelée Figure 0.

Par défaut, les différentes figures vont se supposer sauf si :

On efface : avec `clf()` (*clear figure*) ;
On change de fenêtre : avec `scf()` (*set current figure*)

```
scf(1)// set current figure numéro de la figure
scf(0) // revient à la figure 0
```

4.6.2 Gestion des axes et autres

Le plus simple : en mode graphique

- Avec les boutons dans la fenêtre graphique : cadrage, zoom
- Avec le menu Edition de la fenêtre de la figure.

En terme de commandes :
Chaque fenêtre graphique est gérée comme un objet de type *Figure*. Pour la manipuler, on peut utiliser la command `gcf()` pour *get current figure*. Pour les axes, on peut utiliser `gca()` pour *get current axes*.

```
mafig=gcf()// les données de la figure courante sont stockées dans mafig
a=gca()//  les données des axes...
a.isoview='on'; // méthode qui agit sur a et met à  la même échelle les deux axes
```

Cette commande `isoview` peut être obtenue en cochant dans le menu Edition...

4.6.3 Numéros des couleurs

Pour beaucoup de commandes, il vaut mieux rentrer la couleur avec un numéro. Les numeros et les noms des couleurs basiques se voient avec la commande `getcolor()`. On peut aussi rentrer une couleur avec son numéro RGB : *cherchez comment !*

5 Recherches de zéros de fonctions avec SciLab

5.1 Utilisation du mode graphique : `xclick`

La commande `xclick()` met le programme en pause dans l'attente d'un appui sur un des boutons de la souris à l'intérieur du cadre de la fenêtre graphique.

Au moment d'un click, elle renvoie *la nature du clic (gauche, droit)* et surtout *les coordonnées du points du clic* : soit au total un vecteur formé de *trois nombres*.

Par exemple en déclarant `u=xclick()` et avec un clic droit sur le point de coordonnées (0.3,1.7), après le clic, on aura `u=[0, 0.3,1.7]`

Cette commande est intéressante pour permettre de relever des coordonnées de points sur lesquels on clique.

Un exemple : On cherche à résoudre graphiquement l'équation $\sin(x) = x/2$.

5.2 La fonction fsolve

Pour trouver une solution approchée de l'équation $g(x) = 0$ au voisinage d'un point x_0, on utilise la commande suivante, qui stocke le résultat dans la variable `x` :

```
x=fsolve(x0,g)
```

Pour l'exemple précédente, après avoir repéré une valeur proche de 4 avec `xclick`, on peut donc définir :

```
-->deff('y=f(x)','y=sin(x)-x/2')
--> x=fsolve(2,f)
```

On peut alors faire figurer le point sur la figure.

```
plot2d(x,f(x),-3)
```