TP : démarrer sous scilab

(P. Klein)

- **Scilab :** Principe
  - Scilab est constitué de deux fenêtres : l'une, appelée fenêtre de commande, dans laquelle on peut taper des instructions (il y a aussi une barre d'adresse et un historique, entre autres) ; l'autre, correspondant à l'éditeur de scilab, dans laquelle on va archiver les commandes utilisées, et qu'on pourra sauver sous forme de fichier texte afin de reproduire le TP.
  - Utiliser l'éditeur de scilab pour y noter toutes les commandes. S'il n'est pas déjà ouvert, taper `edit` puis Entrée dans la fenêtre de commande pour y accéder.
  - Enregistrer le fichier avec l'extension `.sce` dans un répertoire approprié. Par exemple : `TP1_script.sce`. C'est ce qu'on appelera un fichier script, il contient les commandes successives permettant de répondre aux différentes questions du TP. On peut placer des commentaires en les faisant précéder du double symbole `//`.
  - Placer scilab dans ce répertoire. Pour savoir dans quel répertoire on se trouve, taper `pwd` puis Entrée dans la fenêtre de commande. Pour lister le contenu du répertoire dans lequel on se trouve, taper `ls` puis Entrée. Pour changer de répertoire si nécessaire, choisir le chemin convenable dans la barre d'adresse, ou taper :

    ```
    cd('la_partition/le_repertoire/le_chemin_voulu/')
    ```

    suivi d'Entrée. A chaque étape, on peut utiliser la touche Tabulation pour afficher les différentes possibilités de complétion.
  - Pour nettoyer l'affichage de la fenêtre de commande (sans effacer le contenu des variables) : commande `clc`, Entrée.
  - Pour effacer toutes les variables : `clear`, Entrée.
  - Pour rappeler une commande précédemment tapée dans la fenêtre de commande, il suffit d'utiliser la flèche vers le haut autant de fois que nécessaire pour parcourir l'historique des commandes tapées. On peut alors éventuellement la modifier avant de la valider.
  - Pour faire exécuter la succession des commandes contenues dans le fichier script, il faut l'exécuter. Deux possibilités : via un petit symbole dans le menu graphique (en forme de flèche), ou bien taper dans la fenêtre de commande : `exec('TP1_script.sce')`.
  - Pour exécuter seulement une partie du script, on peut effectuer un copier-coller des instructions concernées dans la fenêtre de commande, ou bien sélectionner les lignes concernées puis faire un clic droit : " valuer la sélection ".
  - Pour obtenir de l'aide sur une commande, ce qu'elle effectue, sa syntaxe : `help commande`.

- **Les fonctions (1) :** La syntaxe pour déclarer des fonctions est :

```
function [sortie1, sortie2] = nom_de_la_fonction (entree1, entree2)
   (des instructions)
   (sur plusieurs lignes)
   (dans lesquelles on definit notamment
    les variables de sortie sortie1, sortie2 (ou davantage)
    en fonction des variables d'entree entree1, entree2)
endfunction
```

Remarques :

- Les mots **function** et **endfunction** sont des mots-clés indispensables.
- Il peut y avoir zéro, un ou plusieurs arguments d'entrée, comme de sortie, toujours séparés par des virgules.
- Les noms `entree1`, `entree2`, `sortie1`, `sortie1` correspondent à des variables muettes, qui n'ont d'existence qu'à l'intérieur de la déclaration de la fonction.

Exemple de fonction :

```
function [y, flag] = second_membre(x, theta)
   if theta==0
      y = 1;
      flag = false;
   else
```

```
      y = 1/sin(theta) * x^2;
      flag = true;
   end
endfunction
```

- **Les fonctions (2) :** Où placer la déclaration d'une fonction ? Les fonctions sont à placer toutes dans le même fichier, qui portera l'extension `.sci` (pour le différencier des fichiers de scripts portant l'extension `.sce`). Ce fichier portera donc un nom générique du type `TP1_fonctions.sci`, et non pas un nom en rapport avec la première fonction qu'on y décrit, puisqu'il y en aura d'autres.

  Le fichier `TP1_fonctions.sci` est enregistré dans le même répertoire que le fichier `TP1_script.sce`.

- **Les fonctions (3) :** Comment prendre en compte les fonctions ? Dans un premier temps, il est nécessaire d'enregistrer le fichier de fonctions après chaque modification. Mais cela ne suffit pas. Il faut également **exécuter** le fichier. Deux possibilités : soit par le menu graphique (avec un symbole en forme de flèche), soit en tapant dans la fenêtre de commande (si on est dans le bon répertoire) :

  ```
  exec('TP1_fonctions.sci')
  ```

  Pour ne pas oublier cette étape, on peut la placer au démarrage du fichier script : la première ligne du fichier `TP1_script.sce` est alors

  ```
  exec('TP1_fonctions.sci')
  ```

  (en présence de plusieurs fichiers de fonctions, il faudrait les exécuter tous en répétant cette ligne avec chaque nom de fichier).

- Une fois le fichier de fonctions exécuté, la fonction est accessible par son nom depuis la fenêtre de commande. En suivant l'exemple précédent, on tapera :

  ```
  second_membre( 1.3, pi/2)
  ```

  ou, si on veut stocker les deux variables de résultat :

  ```
  [valeur, drapeau] = second_membre( 1.3, pi/2)
  ```

  Le point-virgule sert à supprimer l'affichage. Dans l'instruction suivante, on affecte les variables `valeur` et `drapeau`, mais on n'affiche rien :

  ```
  [valeur, drapeau] = second_membre( 1.3, pi/2);
  ```

- **Les boucles :** La syntaxe est classique. Pour un test :

  ```
  if (condition)
     instructions a realiser;
  elseif (autre condition)
     instructions a realiser;
  else
     instructions a realiser;
  end
  ```

  Boucle `for` :

  ```
  for i = debut:pas:fin
     instructions a realiser;
  end
  ```

  Boucle `while` :

  ```
  while (condition1) && (condition2)
     instructions a realiser;
  end
  ```

- **Les vecteurs (1) :** Un grand nombre d'opérations en scilab peuvent être réalisées de faon condensée et efficace à l'aide de vecteurs. Tout d'abord, quelques commandes utiles :

- Accès à la longueur d'un vecteur (qu'il soit ligne ou colonne) :

  `length(u)`

- Accès aux dimensions d'un vecteur ou d'une matrice :

  `size(A)`

  renvoie deux valeurs, la première étant le nombre de lignes, la deuxième le nombre de colonnes. Pour accéder directement, et uniquement au nombre de lignes :

  `size(A,1)`

  et pour accéder directement au nombre de colonnes : `size(A,2)`.

- Pour transposer un vecteur ou une matrice :

  `A.'`

  (NB : l'opération `A'` transpose **et** conjugue).

- **Les vecteurs (2) :** Comment définir un vecteur ?

  - Vecteur colonne constitué de zéros :

    ```
    u = zeros(N, 1);
    ```

    Le premier argument est le nombre de lignes, le deuxième le nombre de colonnes. Ainsi, pour un vecteur ligne, on tapera : `u = zeros(1, N);`, et pour une matrice à N lignes et M colonnes : `A = zeros(N, M);`.

  - Vecteur colonne constitué de 1 :

    ```
    u = ones(N, 1);
    ```

    (même généralisation pour un vecteur ligne ou une matrice).

  - Vecteur déclaré à la main : la concaténation se fait entre crochets, les colonnes sont séparées par des virgules ou des espaces, les lignes sont séparées par des points-virgules.

    ```
    u = [1, 2, 5.3, -1, %pi, sin(1), -5.21]
    ```

    (ou `u = [1 2 5.3 -1 %pi sin(1) -5.21]`) pour un vecteur ligne,

    ```
    u = [1; 2; 5.3; -1; %pi; sin(1); -5.21]
    ```

    pour un vecteur colonne.

  - Matrice déclarée à la main (la déclaration se fait ligne par ligne) :

    ```
    A = [ 1, 1.1, 1.3, 1.05 ; -2.1, -3, -5, 0 ; -1, -7, 4, 10 ]
    ```

    est une matrice à 3 lignes et 4 colonnes.

  - Matrice identité, éventuellement rectangulaire :

    `eye(N, M)`

  - Vecteur constitué d'éléments régulièrement espacés : la syntaxe est `u = debut:pas:fin`. Lorsque le pas est égal à 1, on peut l'omettre. Exemples :

    `u = 1:10`

    renvoie les entiers de 1 à 10,

    `u = 2:2:10`

    renvoie les entiers pairs inférieurs ou égaux à 10,

    `u = 0:-1:-10`

    renvoie les entiers négatifs classés par ordre décroissant jusqu'à -10,

    `u = 3:0.1:4`

    renvoie tous les dixièmes compris entre 3 et 4.

- **Les vecteurs (3) :** Comment accéder aux éléments d'un vecteur ?

  - Le symbole est la parenthèse. Accès au `i`-ième élément d'un vecteur :

`u(i)`

(s'il s'agit d'un vecteur ligne, la commande équivalente est `u(1,i)` ; s'il s'agit d'un vecteur colonne, la commande équivalente est `u(i,1)`).

Important : la numérotation des éléments d'un vecteur/matrice commence toujours à 1. Ainsi, `u(0)` n'est jamais défini. Faire les décalages nécessaires en fonction des notations mathématiques utilisées.

Le premier élément d'un vecteur est `u(1)`, le dernier élément est `u($)`.

- Accès à l'élément `(i,j)` d'une matrice :

  `A(i,j)`

- Accès à certains éléments d'un vecteur : le principe est d'appeler `u(v)`, où `v` est un vecteur d'entiers contenant les indices des éléments de `u` qu'on veut extraire. Souvent, le vecteur `v` lui-même peut être construit à l'aide de la syntaxe utilisant `:`. Exemples :
  Pour prendre tous les éléments, sauf le premier :

  `u(2:$)`

  Pour prendre tous les éléments de `u`, sauf le dernier :

  `u(1:$-1)`

  Pour ne prendre que les éléments de `u` d'indice pair :

  `u(2:2:$)`

  Pour prendre les éléments d'indice `5` à `13` :

  `u(5:13)`

  Pour retourner `u` en prenant les éléments en ordre contraire :

  `u($:-1:1)`

  Et plus généralement :

  `u(v)`

  tant que `v` est un vecteur d'entiers compris entre 1 et la longueur de `v`. Un indice peut être répété.

- Accès à certains éléments d'une matrice :

  `A(v1,v2)`

  extrait les lignes de `A` dont les indices sont donnés par le vecteur `v1`, puis n'en garde que les colonnes d'indices donnés par `v2`.
  Pour prendre toutes les lignes (ou toutes les colonnes), on utilise `:` seul. Exemple :

  `A(2:$,:)`

  renvoie `A` privée de sa première ligne, mais en gardant les colonnes intactes.
  Pour supprimer la première et la dernière ligne de `A`, on écrirait :

  `A( 2:$-1, 2:$-1)`

  Remarque : on utilise ce principe pour modifier un ou plusieurs éléments d'un vecteur ou d'une matrice. Exemples :

```
u(2) = -1;
u(1:2:$) = 0;
A($,$) = 0;
A(1,:) = zeros(1, size(A,2));
```

- **Les vecteurs (4) :** La concaténation. On a déjà vu la concaténation pour assembler un vecteur. On peut également l'utiliser pour ajouter des éléments à un vecteur existant.

  Si `u1` est un vecteur colonne :

  `u2 = [0 ; u1 ; 0 ];`

  (ou `u2 = [0 u1 0 ];` si `u1` est un vecteur ligne).

  Concaténer des vecteurs colonnes de même longueur pour obtenir une matrice :

  `A = [ u1 u2 u3 u4 ];`

Ajouter une ligne à une matrice :

```
B = [ zeros(1,size(A,2)) ; A ];
```

- **Utilisation des vecteurs dans les fonctions :** Il est très utile de pouvoir appliquer une fonction directement à un vecteur, et de récupérer en sortie le vecteur des résultats associés à chaque composante du vecteur d'entrée. Par exemple, si on considère la fonction

```
function y = somme(x)
   y = 1 + 2*x;
endfunction
```

on peut l'appliquer aussi bien à un scalaire `x` qu'à un vecteur `X`, car l'opération $X \mapsto 1 + 2X$ a un sens lorsque X est un vecteur, et ce sens est bien le même que pour $x_i \mapsto 1 + 2x_i$ si les x_i sont les composantes du vecteur X. Ainsi, `Y = somme(1:10)` renvoie `[3 5 7 9 11 13 15 17 19 21 ]`, et remplace avantageusement l'utilisation d'une boucle `for`.

Dès qu'on écrit une fonction scilab, il faut alors se poser la question de savoir si elle se généralisera à des entrées vectorielles. Parfois, une adaptation est nécessaire. Le produit $2X$ a un sens si X est un vecteur, mais pas XZ si X et Z sont des vecteurs. Le sens à donner à ce produit, pour généraliser le cas scalaire, serait un produit composante par composante : $x_i z_i$. L'opérateur effectuant le produit composante par composante est `.*`. Appliqué à des scalaires, il correspond au produit classique. De cette faon, la fonction

```
function y = produit(x,z)
   y = 1 + 2*x.*z;
endfunction
```

peut être appliquée à des entrées X et Z vectorielles, et retourne alors Y tel que $y_i = 1 + 2x_i z_i$. Sans le `.` dans le produit `x.*z`, `produit(X,Z)` n'aurait pas de sens pour `X` et `Z` vecteurs. En revanche, dans le produit `2*x`, le `.` n'est pas nécessaire car on ne multiplie que par un scalaire.

Généralisation d'opérateurs classiques à des opérations composante par composante :

- le produit `*` entre deux variables doit être remplacé par `.*` pour permettre la compatibilité avec des opérations entre vecteurs
- la puissance `^` doit être remplacée par `.^`
- l'inverse ou le quotient `/` doivent être remplacés par `./`, **en respectant une espace avant le point** (sinon, le calcul se fait mais le résultat n'est pas le même...).
- les fonctions racine carrée `sqrt`, logarithme `log`, exponentielle `exp`, trigonométriques `cos`, `sin`, ..., sont directement utilisables avec des entrées vectorielles.

Exemple :

```
function y = second_membre(x)
   y = 1 ./(1+x.^2) .* ( 2*sqrt(x).*sin(x) );
endfunction
```

Cette fonction est compatible avec des opérations entre vecteurs.

- **Les graphiques :** On utilise la commade `plot`, et on représente toujours des vecteurs.

```
plot(X,Y)
```

représente les points de coordonnées (x_i, y_i) (les vecteurs `X` et `Y` doivent être de même longueur). Souvent, `X` peut être tout simplement `1:(length(Y))`.

Pour représenter une fonction f connue sous forme analytique, on commence par l'évaluer sur une grille suffisamment fine, et on représente cette évaluation : pour représenter $f : x \mapsto \cos x$ sur $[-\pi, \pi]$:

```
X = -%pi:0.1:%pi;
Y = cos(X);
plot(X,Y)
```

La commande `plot` admet un troisième argument, qui est une chane de caractère (donc palcée entre apostrophes `'`) pouvant cumuler trois symboles (dont l'un des symboles peut être constitué de deux caractères, NB), juxtaposés dans un ordre indifférent, portant sur :

- la couleur du tracé : `r` pour rouge, `b` pour bleu, `g` pour vert, etc. Valeur par défaut : bleu.
- le style de la ligne rejoignant les points : `-` pour une ligne continue, `:` pour une ligne en pointillés, `--` ou `-.` pour une généralisation des pointillés. Valeur par défaut : ligne continue, sauf si un symbole est indiqué, auquel cas aucune ligne ne rejoint les points.
- le symbole marquant l'emplacement de chaque point représenté : `o`, `x`, `*`, `v`, `<`, `d`, etc. Valeur par défaut : pas de symbole.

Liste complète des possibilités : `help plot`.

Exemples : comparer

```
X = 1:10;
Y = X.^2;
plot(X,Y)
plot(X,Y,'d')
plot(X,Y,':')
plot(X,Y,'d:')
plot(X,Y,'rd-.')
```

On peut représenter simultanément plusieurs courbes sur le même graphique :

```
plot(X,Y,'*:',X1,Y1,'o')
```

Autres attributs d'un graphique :

- titre : `title`
- légende : `legend`
- étiquettes d'axes : `xlabel`, `ylabel`
- juxtaposition de plusieurs graphiques : `subplot`