*Au secours ! Comment ça marche Scilab ?*

# 1. LA CONSOLE : COMPLÉMENTS

## 1.1. Calculs simples dans la console

Dans la console, après l'invite de commande --> , il suffit de saisir une instruction et d'appuyer sur la touche [↵] du clavier pour obtenir le résultat correspondant.
Calculs masqués : ajouter « ; » à la fin de la ligne.
Pour reprendre une instruction antérieure, utiliser la touche du clavier : [↑] ou le début de l'instruction précédé de « ! ».
Autocomplétion : SciLab complète automatiquement les fonctions qu'il connaît si on utilise la touche de tabulation : [⭾].

### Écriture dans Scilab, rappel de l'essentiel

écriture et opérations simples

| vraie vie | $=$ | $a+b$ | $a-b$ | $2,5$ | $a\times 10^n$ | $\pi$ | $\sqrt{x}$ | $e^x$ | $\ln x$ |
|---|---|---|---|---|---|---|---|---|---|
| Scilab | `=` | `a+b` | `a-b` | `2.5` | `aEn` | `%pi` | `sqrt(x)` | `exp(x)` | `log(x)` |

multiplication, division, puissance dans $\mathbb{R}$

| vraie vie | $a\times b$ | $a\div b$ | $a^b$ |
|---|---|---|---|
| Scilab | `a*b` | `a/b` | `a^b` |

opérateurs de comparaison

| $=$ | $\neq$ | $<$ | $\leqslant$ | $>$ | $\geqslant$ |
|---|---|---|---|---|---|
| `==` | `<>` | `<` | `<=` | `>` | `>=` |

multiplication, division, puissance, appliquées aux vecteurs, aux matrices (tableaux)

| | opérations termes à termes (→ graphes) | | | opérations matricielles | |
|---|---|---|---|---|---|
| vraie vie | $a\times b$ | $a\div b$ | $a^b$ | $a\times b$ | $a\div b$ |
| Scilab | `a.*b` | `a␣./b` | `a.^b` | `a*b` | `a/b` |

␣ : espace

Effacer (réinitialiser) toutes les variables : `clear`
Effacer la console (vider l'écran) : `clc` (`cl`*ear* `c`*onsole*), [F2], *Menu Édition* > *Effacer la console*, ou cliquer sur 🧹

## 1.2. Tableaux de nombres

a) <u>petit résumé</u>

- Les tableaux sont délimités par des crochets : « `[` » avant le 1er élément, « `]` » après le dernier.
- On sépare les nombres par un espace (on peut aussi utiliser une virgule à la place).
- On indique la fin d'une ligne en passant à la ligne [↵] ou bien avec un point-virgule.

b) <u>listes à distribution régulière</u>

- *`début:pas:fin`* permet de créer une liste dont les éléments commencent à la valeur de *`début`*, sont espacés de la valeur du *`pas`* et ne dépassent pas la valeur limite *`fin`*. En principe, vous connaissez cette première méthode pour l'avoir utilisée en maths.
- `linspace(`*`début,fin,nb`*`)` permet de créer une liste comportant *`nb`* éléments et allant de la valeur *`début`* à la valeur *`fin`* (*linspace* pour *linearly spaced vector* – vecteur à composantes linéairement espacées). Cette seconde méthode garantit que les bornes sont incluses.

c) <u>transposition</u>

On utilise l'opérateur « `'` » placé après le tableau. Exemple : `L'`

## 1.3. Mise en forme des résultats

a) <u>La fonction `disp`</u>

La fonction `disp` (pour *display* – afficher) est suivie de parenthèses à l'intérieur desquelles on place ce qu'on veut voir. Supposons que nous voulions afficher le 4e élément de la liste `L` précédente ; notez la différence entre le mode d'affichage simple et le mode utilisant `disp` :

```
-->L(4)
 ans  =
    4.712389
-->disp(L(4))
    4.712389
```

b) affichages plus complexes

Pour afficher une chaîne de caractères (en général une phrase), on la met entre guillemets :

```
--> disp("Maurice a gagné")
 Maurice a gagné
```

Pour mélanger des mots et des valeurs, utilisez la commande `string` qui transforme les valeurs en caractères (chaîne de caractères se dit *character string* en anglais), et « + » entre les différentes parties, pour les *concaténer* :

```
--> disp("Maurice a gagné "+string(L(4))+" euros.")
 Maurice a gagné 4.712389 euros.
```

Si la phrase contient une apostrophe, il est nécessaire de la doubler dans la chaîne de caractères pour qu'elle s'affiche correctement :

```
--> disp("Maurice n''a gagné que "+string(L(4))+" euros.")
 Maurice n'a gagné que 4.712389 euros.
```

Très bien, mais au fait : *4,712 389* €… C'est quoi ces 6 chiffres après la virgule ?!

C'est l'occasion de parler rapidement du formatage de nombres. Nous pouvons fixer le nombre de chiffres significatifs avec la fonction `format(n)`, où `n` = nombre de chiffres significatifs + 2 (1 pour la virgule, 1 pour le signe éventuel) ; `n` ⩽ 25 ; `n` = 10 par défaut.

Ici, nous choisirons donc `n` = 5.

```
-->format(5);disp("Maurice n''a gagné que "+string(L(4))+" euros.")
 Maurice n'a gagné que 4.71 euros.
```

Voilà ! Mais il faut maintenant rétablir le format par défaut pour la suite… Vivement les scripts pour automatiser ça !

```
-->format('v',10)
```

NB : `'v'` signifie *variable* (affichage automatique, scientifique ou pas selon le nombre). L'autre option est `'e'` pour *engineer* (affichage scientifique). En réalité, la mention `'v'` est donc ici superflue.

## 2. SCRIPTS

### 2.1. Introduction : SciNotes

Un script est un fichier texte d'extension SCE qui regroupe plusieurs instructions. On peut l'éditer dans le Bloc-notes de Windows ou mieux, dans l'éditeur de Scilab, nommé SciNotes.

Pour lancer SciNotes : taper `editor` dans la console ou cliquer sur .

La fenêtre de SciNotes s'ouvre, je vous conseille de l'arrimer à la fenêtre principale, à côté de la console.

À l'ouverture, un script vide est automatiquement créé. Ensuite, pour créer un nouveau script : *Menu Fichier > Nouveau*, Ctrl {n} ou cliquer sur . Pour ouvrir un script existant : *Menu Fichier > Ouvrir*, Ctrl {o} ou cliquer sur .

### 2.2. Exemple de script

a) présentation

Voici un script qui calcule la période propre d'un pendule pesant simple en fonction de sa longueur $L$.

```
// Période propre d'un pendule pesant simple
mode(0)         // Affichage des résultats dans la console
```

```
clear           // Réinitialisation de toutes les variables
L = input("longueur L (en cm) : ");
L = L/100;    // conversion en m
g = 9.81;       //(en m/s²)
w0 = sqrt(g/L);
T0 = 2*%pi/w0;
format(5)       // 3 chiffres significatifs
disp("La période propre est de "+string(T0)+" s.")
format(10)      // retour au format par défaut
```

b) enregistrement

Il est prudent d'enregistrer le script avant même d'avoir fini de le taper (plutôt que de l'écrire, vous pouvez utiliser le copier-coller à partir du document en ligne – http://www.phycats.plaf.org/docs/index.php#I) : ou Ctrl {s}, classique. Enregistrez votre fichier dans '*Mes documents*', dans un sous-dossier que vous retrouverez facilement, genre *Scilab-physique*, avec un nom facile à identifier, par exemple '*Période propre pendule simple.sce*'.

c) contenu de base

Observer les lignes 2 et 3.

- commentaires

Remarquez les "//" : ils définissent un début de commentaire, le commentaire se terminant en fin de ligne.
Un commentaire est repéré par sa couleur verte. Comme son nom l'indique, il commente, par opposition au reste du code qui correspond à des instructions.
Les commentaires sont facultatifs, mais très utiles pour retrouver rapidement à quoi sert un script.
Remarque : Ctrl {d} transforme une instruction en commentaire, Ctrl ⇧ {D} fait le contraire.

- affichage des résultats

Par défaut, l'exécution d'un script lancé depuis SciNotes se fait en mode "silencieux" soit `mode(-1)`. Les résultats ne sont pas affichés… Le script indique donc qu'on se placera en `mode(0)`, pour afficher les résultats.

*Remarque : le `mode(0)` est le mode d'exécution par défaut d'un script lancé depuis la console par la commande `exec`. Le `mode(2)` est semblable, à ceci près qu'il laisse des lignes vides à la place des calculs cachés et des commentaires.*

Un résultat non souhaité ? Terminez sa ligne par « **;** »

- nettoyage

Pour éviter tout problème de conflit, on demandera au script d'effacer toutes les variables : `clear`

- récapitutatif

Tout script pourrait contenir ces quelques instructions. On pourrait également, au cas où une courbe serait tracée, ajouter une commande fermant toutes les figures (fenêtres contenant des graphiques) : `xdel(winsid())`
⚠ Si vous souhaitez tracer une courbe dans une fenêtre existante, cette commande est évidemment à proscrire.
Vous pouvez donc vous créer (voir §1.1) un "script de départ" pour les scripts à venir :

```
//Titre
mode(0)         // Affichage des résultats dans la console
clear           // Réinitialisation de toutes les variables
//xdel(winsid()) // Fermeture (ou pas) de toutes les figures
```

Vous pouvez le nommer par exemple '*_script_vierge.sce*', et le mettre en lecture seule.

d) commandes particulières

Remarquer la présence des commandes `input`, `format` et `disp`. En gros :

- `input` a une fonction interactive : elle permet d'indiquer à l'utilisateur qu'il doit entrer des données.
- `format(n)` limite le nombre de chiffres significatifs à `n-2`. Neutraliser la ligne 10 avec `//` pour voir la différence.
- `disp` sert à afficher un résultat sous une forme plus élaborée.

Ces commandes ne sont pas essentielles à votre formation. Pour en savoir plus, lire l'*annexe* :
http://www.phycats.plaf.org/docs/I_annexe.pdf

e) exécution d'un script

Si le fichier n'a pas été enregistré, cliquer sur ou faites F5, pour l'enregistrer puis l'exécuter.
Sinon, cliquer sur ▷ (si nécessaire, Scilab vous demandera de l'enregistrer).
C'est le plus simple dans notre contexte ; d'autres méthodes existent.

### 2.3. Arrêter l'exécution d'un script

Avant d'aller plus loin, voici qui pourrait vous être utile, par exemple lorsqu'une exécution ne répond plus. Dans la console, Ctrl {c} suspend l'exécution d'un script, en *coupant* le processus en cours. L'invite de commande prend alors cette allure :

```
-1->
```

La commande `resume` reprend l'exécution, tandis que la commande `abort` l'arrête :

```
-1->abort
-->
```

***Exercice S1***

On donne le mouvement oscillatoire non amorti d'un point M caractérisé par sa position au cours du temps : $x(t) = X\cos(\omega_0 t)$. On connaît l'amplitude du mouvement $X = 10$ cm et la période propre $T_0 = 1$ s.

Écrire un script permettant de calculer $x$ en cm à une date $t$ donnée. Une ligne devra calculer $\omega_0$ à partir de $T_0$.
Enregistrer ce script *par exemple* sous '*ex1_oscillations.sce*'

*Suggestion : pour commencer, si vous l'avez créé, ouvrez '_script_vierge.sce' et enregistrez-le sous un autre nom.*
*Pour entrer la date t, deux solutions : soit une invite à l'aide de la commande* `input`*, soit simplement une ligne affectant une valeur à la variable t (*`t=0.2;` *par exemple) que vous modifierez manuellement avant chaque lancement du script.*

C'est mieux que de réécrire la fonction $x(t)$ à chaque valeur de $t$, mais c'est encore fastidieux. Il y a mieux, en définissant $x(t)$ en tant que *fonction* dans Scilab (voir §2).
Avant cela, je vous indique en §1.4 comment lancer un script depuis un autre script. Cela nous sera utile pour une utilisation particulière des fonctions. Vous pourrez y revenir plus tard. Passez directement au §2.

### 2.4. Script dans un script (pour mémoire)

a) la commande `exec`

On peut lancer un script depuis un script avec la commande `exec('chemin…\script appelé.sce');`
En `mode(0)`, le « `;` » sert à ne pas afficher les commentaires du script exécuté.
La commande d'exécution du script *appelé* sera inscrite dans le script *appelant*. Exemple :

```
//Script dans un script (script appelant - 'Script dans un script.sce' par exemple)
mode(0)         // Affichage des résultats dans la console
clear           // Réinitialisation de toutes les variables
xdel(winsid()) // Fermeture de toutes les figures
exec('P:\chemin…\ex1_oscillations.sce');    //script appelé : corriger le chemin (voir §b ci-dessous)
```

Essayez sans le « `;` » à la fin, pour voir la différence.

b) chemin vers un script

- Si les scripts sont sur un disque local (`C:` par exemple), je vous conseille d'utiliser des chemins relatifs. Si cela fonctionne (ce n'est pas toujours le cas, je ne sais pas pourquoi…), c'est plus souple. Notez que vous pouvez utiliser indifféremment des slash « `/` » ou des antislash « `\` ».
  - 1er cas : l'appelé est dans le même répertoire de l'appelant. Le nom du fichier est juste précédé de « `./` » :
    `'./script appelé.sce'`
  - 2e cas : l'appelé est dans un sous-répertoire de l'appelant. Le chemin est :
    `'./sous-répertoire/script appelé.sce'`
  - 3e cas : l'appelé est dans un répertoire voisin de celui de l'appelant. Le chemin est :
    `'./../répertoire voisin/script appelé.sce'`
  - et ainsi de suite, « `../` » permettant de remonter dans l'arborescence.
- Si les scripts sont sur un disque réseau (`P:` par exemple), vous n'aurez probablement pas d'autre choix que d'utiliser des chemins absolus :
  `'P:\chemin…\script appelé.sce'`

### 2.5. Autre exemple de script

Voici un script qui définit le carré magique de Dürer, et qui vérifie ensuite quelques-unes de ses nombreuses propriétés "magiques" (*Le carré magique d'ordre 4 dit de Dürer a été utilisé par le peintre allemand dans sa gravure Melencolia en 1514*).

C'est l'occasion, comme vous pouvez le constater, de réviser quelques opérations sur les tableaux (extraction partielle) et de voir deux nouvelles opérations :

`sum`, qui fait la somme de certains éléments d'un tableau, et `diag(M)`, qui extrait la diagonale principale d'un tableau `M`, c'est-à-dire le tableau contenant les termes `M(i,i)`.

```
//Définition du carré magique de Dürer
//Puis vérification de quelques-unes de ses propriétés
magiques
mode(0)         // Affichage des résultats dans la console
clear           // Réinitialisation de toutes les variables
xdel(winsid())  // Fermeture de toutes les figures
M = "carré magique de Dürer"
M = [16 3 2 13
     5 10 11 8
     9 6 7 12
     4 15 14 1]                        //somme des éléments…
Somme_ligne1 = sum(M(1,1:4))           //…de la 1e ligne
Somme_ligne2 = sum(M(2,1:4))           //…de la 2e ligne
Somme_colonne3 = sum(M(1:4,3))         //…de la 3e colonne
Somme_diagoPrincipale = sum(diag(M))   //…diagonale principale
```

Exécuter le script. Étonnant, le carré magique, non ?

## 2.6. interactivité : la commande input

Au cours de son exécution, un script peut demander à l'utilisateur de saisir une donnée. Cette demande se fait avec la commande `input`, qui affiche un message d'invite. La saisie sera alors affectée à une variable que nous nommerons `x` pour la suite de l'explication, cette variable étant une valeur numérique ou une chaîne de caractères.

- si `x` est un nombre réel, on écrit : `x = input("`*`message`*`")`
  `"`*`message`*`"` désigne le message d'invite, à placer entre guillemets.
- si `x` est une chaîne de caractères, on écrit : `x = input("`*`message`*`","string")` ou
  `x = input("`*`message`*`","s")`

Testons `input` dans la console :

```
-->a = input("Quel est ton âge ? ")
Quel est ton âge ? 37
 a  =
    37.
-->n = input("Quel est ton nom ? ","s")
Quel est ton nom ? Maurice
 n  =
Maurice
```

## 2.7. la commande input dans SciNotes

Essayons ceci dans SciNotes, ce sera aussi l'occasion de réutiliser la fonction `disp` (§1.3).

Enregistrons ce script sous '*âge et nom.sce*' par exemple.

Quel est l'intérêt des « `;` » ? Essayez sans, pour voir.

Autres solutions possibles : puisque nous sommes en `mode(0)`, nous pouvons nous affranchir de la fonction `disp`. Le résultat est cependant un peu moins convivial. Testez les deux alternatives proposées.

Rappel : Ctrl {d} transforme une instruction en commentaire, Ctrl ⇧ {D} fait le contraire.

```
// âge et nom
mode(0)         // Affichage des résultats dans la console
clear           // Réinitialisation de toutes les variables
xdel(winsid())  // Fermeture de toutes les figures
a = input("Quel est ton âge ? ");
n = input("Quel est ton nom ? ","s");
```

```
disp(string(n)+", tu as donc "+string(a)+" ans.")
//string(n)+", tu as donc "+string(a)+" ans."                //← alternative 1
//conclusion = string(n)+", tu as donc "+string(a)+" ans."  //← alternative 2
```

# 3. FONCTIONS : COMPLÉMENT

## 3.1. Fonction constante

a) définition simpliste, applicable à des valeurs discrètes seulement

Fonction définie par un script :

```
function f=f(x)
    f = 2
endfunction
```

Utilisation dans la console :

```
-->f(0)
 ans  =
    2.

-->f(1)
 ans  =
    2.
```

Et ainsi de suite, c'est sans surprise.
Par contre, s'il s'agit d'appliquer la fonction à une *liste* de valeurs…

```
-->f(linspace(0,5,6))'
 ans  =
    2.
```

Nous n'obtenons pas une *liste* d'images, représentée par un vecteur colonne comportant les différentes valeurs de $f(x)$.
Conséquence : pour tracer un graphe à partir de séries de données, ça va coincer…
Il faut donc corriger la définition de notre fonction constante pour pouvoir l'appliquer à des listes de valeurs (vecteurs).

b) définition complète, applicable à des listes de valeurs

Il faut prendre en compte la possibilité que `x` soit un vecteur. Nous pouvons utiliser la fonction `ones(x)` qui renvoie un tableau rempli de `1` de même taille que `x`. Nous multiplions ensuite `ones(x)` par la valeur de la constante :

```
function cste=cste(x)
    cste = ones(x)*2
endfunction
```

Utilisation dans la console :

```
-->cste(0)
 ans  =
    2.

-->cste(1)
 ans  =
    2.
```

```
-->cste(linspace(0,5,6))'
 ans  =
    2.
    2.
    2.
    2.
    2.
    2.
```

Remarque : il y aurait d'autres solutions pour définir cette fonction, comme par exemple écrire `cste = 0*x + 2`.

Mais pourquoi faire, au fait ? Quel intérêt peut-il bien y avoir à tracer une droite horizontale dans Scilab ?
Cette situation se rencontre, par exemple, dans le cas d'une fonction définie par morceaux, où sur un certain intervalle la fonction reste constante. Du coup, cela soulève la question de la définition des fonctions par morceaux dans Scilab.

## 3.2. Fonction par morceaux

a) cas d'une fonction continue

◊ *définition de la fonction (par un script)*

Exemple d'une fonction affine par morceaux : si $x \leqslant 1$ alors $f(x) = -2x + 4$ et si $x \geqslant 1$ alors $f(x) = 2x$

```
function f=f(x)
    f(x<1) = -2*x(x<1) + 4
    f(x>=1) = 2*x(x>=1)
endfunction
```

◊ *Utilisation dans la console :*

```
-->f(-1)
 ans  =
    6.

-->f(0)
 ans  =
    4.

-->f(1)
 ans  =
    2.

-->f(2)
 ans  =
    4.
```

```
-->x=(linspace(-1,2,4))'
 x  =
  - 1.
    0.
    1.
    2.

-->f(x)
 ans  =
    6.
    4.
    2.
    4.
```

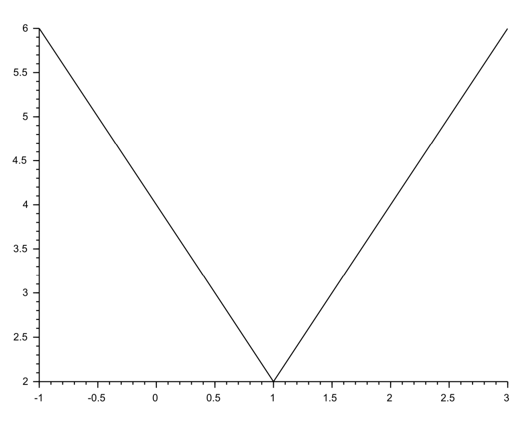
Remarque : on aurait pu penser à une définition utilisant un test `if… then… else…`, mais cela ne fonctionne pas avec les listes de valeurs, car Scilab ne retient que la dernière définition de la fonction.

◊ *graphe*

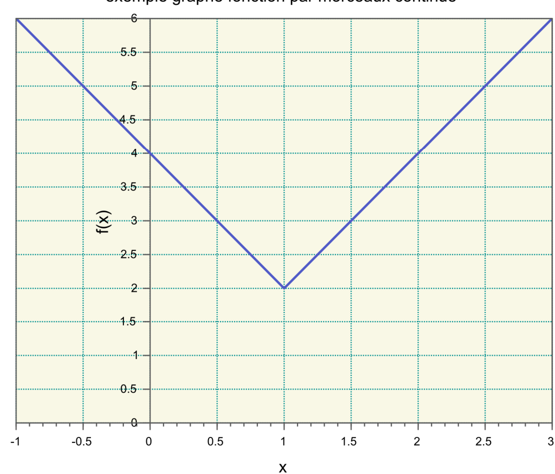
Dans la console :

```
-->x=(linspace(-1,3,1000))';
-->plot2d(x,f(x))
```

On obtient :



*Graphe, version "brute"*




*Version mise en forme avec 'options_graphes.sci'*

b) <u>cas d'une fonction discontinue</u>

◊ *définition de la fonction (par un script)*

Exemple : si $x < 1$ alors $f(x) = -2x + 1$ et si $x > 1$ alors $f(x) = 2x + 1$ (discontinuité en $x = 1$)

```
function f=f(x)
```

```
    f(x<1)= -2*x(x<1) + 1
    f(x>1)= 2*x(x>1) + 1
endfunction
```

◊ *Utilisation dans la console :*

```
-->f(-1)
 ans  =
    3.

-->f(0)
 ans  =
    1.

-->f(1)
 ans  =
    []

-->f(2)
 ans  =
    5.
```

```
-->x=(linspace(-1,2,4))'
 x  =
  - 1.
    0.
    1.
    2.

-->f(x)
 ans  =
    3.
    1.
    0.
    5.
```

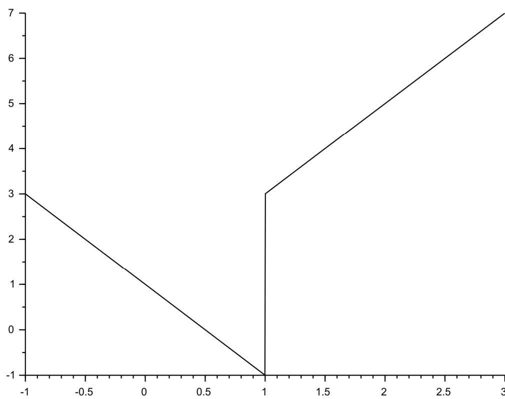
On note d'une part que la discontinuité se traduit par `f(1) = []` (`[]` représentant un vecteur de longueur nulle), d'autre part que lorsqu'on applique `f` à une liste contenant `1`, on obtient `f(1) = 1` (bug).

◊ *graphe*

Si on procède comme dans le cas d'une fonction continue, avec une seule liste de valeurs,

```
-->x=(linspace(-1,3,1000))';
-->plot2d(x,f(x))
```

on obtient le graphe d'une fonction qui semble continue (voir ci-contre).

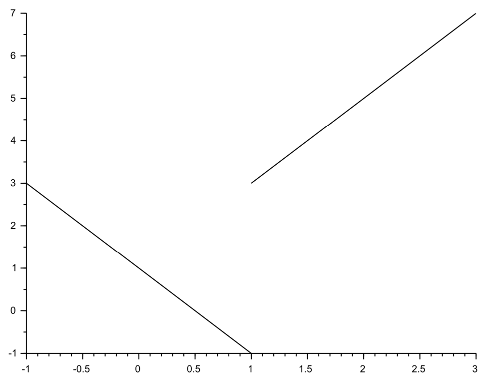


Il faut donc tracer l'un après l'autre les deux graphes correspondant aux intervalles $[-1;1[$ et $]1;3]$ :

```
-->x=(linspace(-1,3,1000))';
-->plot2d(x(x<1), f(x(x<1)))
-->plot2d(x(x>1), f(x(x>1)))
```

on obtient le graphe ci-contre.

Remarque : il ne faut évidemment pas fermer la figure après le premier tracé.

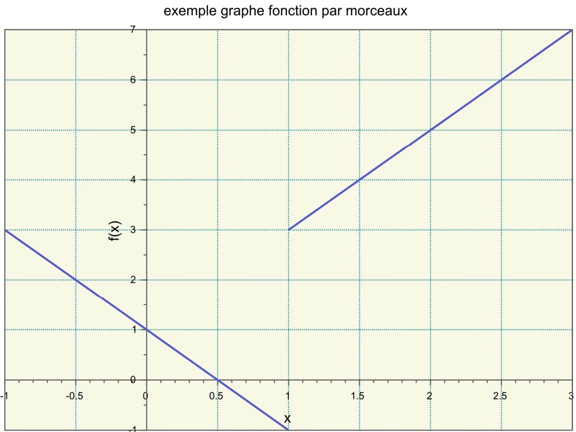


Variante, en passant par un script pour le graphe (le script de la fonction, situé dans un sous-répertoire '*_fct*' s'appelant '*fonction par morceaux discontinue.sci*') et en améliorant le graphique avec '*options_graphes.sci*' :

```
// exemple graphe fonction par morceaux
mode(0)         // Affichage des résultats dans la console
clear           // Réinitialisation de toutes les variables
xdel(winsid()) // Fermeture de toutes les figures

// intervalles d'abscisses
x=(linspace(-1,3,500))';

// appel fonction
exec('./_fct/fonction par morceaux discontinue.sci', -1)  // corriger le chemin

// tracé
plot2d( x(x<1), f(x(x<1)) )
```

```
plot2d( x(x>1), f(x(x>1)) )

//mise en forme
exec('./options_graphes.sci', -1)           // corriger le chemin
```

Graphe obtenu :




*Remarque : les formulations correspondant à des superpositions de graphes (tracés simultanés) conduisent toutes à des tracés continus. En voici diverses variantes, toutes infructueuses :*

```
plot2d(x, [f(x(x<1)),f(x(x>1))] )

plot2d( [x(x<1),x(x>1)], [f(x(x<1)),f(x(x>1))] )

y=f(x)
plot2d( [x(x<1),x(x>1)], [y(x<1),y(x>1)] )
```

*Il faut donc retenir que durant un* `plot2d` *Scilab "ne relève pas le stylo", puisque la fonction de* `plot2d` *est de relier des points. S'il y a discontinuité, il faut écrire deux* `plot2d`*.*

*Remarque complémentaire : avec* `plot` *(tout court) le problème se serait posé différemment, mais nous n'avons pas appris à l'utiliser.*

```
plot(x(x<1),f(x(x<1)),'b',x(x>1),f(x(x>1)),'b')
```

## 3.3. Fonctions à plusieurs variables

Par exemple, pour transformer des euros (`e`) en dollars (`d`) avec un taux de change (`t`), définissons la fonction `dollars`. Les variables sont `e` et `t` et l'image est `d`.

```
-->function dollars = dollars(e,t)
--> dollars = e*t
-->endfunction

-->dollars(200,1.4)
 ans  =
    280.
```

# 4. GRAPHE D'UNE FONCTION À 1 VARIABLE : MISE EN FORME

## 4.1. Mise en forme sommaire : options de plot2d

a) introduction

Commençons par explorer les options de la commande `plot2d`.

```
plot2d(x, y, arguments optionnels)
```

*x* et *y* sont les séries d'abscisses et d'ordonnées sous forme de vecteurs-colonne.
Les arguments sont de la forme *mot-clef=valeur*.

b) couleur et aspect de la courbe : `style`

`style=n` définit le type de tracé (couleur, marqueur). $n \in \mathbb{Z}$ :

- si $n = 1$ la courbe sera en trait plein noir (valeur par défaut si aucun style n'est indiqué)
- si $n > 1$ la courbe sera en couleur ; on peut voir la correspondance entre le nombre et la couleur (carte des couleurs) grâce à la commande `getcolor()`
  Exemple : `plot2d(x,y,style=5)` permet d'obtenir une courbe rouge.
- si $n \leqslant 0$ la courbe sera remplacée par des marqueurs (prévoir d'espacer les points, pour plus de visibilité) :
  0 → · −1 → + −2 → × ... −9 → ○ −10 → ☆

`style=getcolor()` permet à l'utilisateur, au moment du tracé, de choisir la couleur dans la palette affichée (valider en cliquant sur [OK]).

c) échelles

`logflag="valeur"` : définit l'échelle (linéaire ou logarithmique) le long des axes. Valeurs possibles : `"nn"`, `"nl"`, `"ln"` et `"ll"` (n pour *normal*, c'est-à-dire linéaire, et `l` pour *logarithmique*). Par défaut `logflag="nn"`.

`frameflag=n` $(0 \leqslant n \in \mathbb{N} \leqslant 9)$ permet d'adapter la graduation des axes selon diverses priorités. Les options par défaut sont souvent les meilleures, on se passera donc le plus souvent de cette option. Un cas utile cependant : si abscisses et ordonnées sont de même dimension, il peut être intéressant d'avoir un repère orthonormé : `frameflag=4`.
Pour en savoir plus :

```
-->help frameflag
```

## 4.2. Paramètres graphiques

Pour Scilab, les graphiques sont des objets, de type *polyline*. On peut donc ajuster leur apparence en modifiant les propriétés de ces objets. Première chose, nous devons récupérer la "poignée" (*handle* en anglais) de l'objet, pour pouvoir l'attraper. Pour cela, il faut utiliser la fonction `get("current_axes")` juste après le tracé. Le but étant de se servir de la poignée en question, il faut la garder en réserve dans une variable, que nous nommerons ici `a` :

```
-->a = get("current_axes");
```

NB : la fonction `get("current_axes")` pouvant s'abbréger en `gca()`, on peut donc entrer :

```
-->A = gca();
```

Vous voulez vous faire peur en voyant toutes les propriétés existantes, autrement dit tous les paramètres disponibles, et donc en affichant les réglages actuels ? Enlevez le « `;` ».

Pour définir la propriété de cet objet, on fait alors

```
A.propriété = "valeur";
```

### 4.3. Mise en forme de base

a) repère orthonormé

La propriété est `isoview`, par défaut réglée sur `off`. Dans la majorité des cas, en physique, abscisse et ordonnée ne sont pas de la même dimension (par exemple angle et énergie). Avoir un repère orthonormé dans ce cas ne présente pas d'intérêt. Dans le cas contraire, on utilisera :

```
-->A.isoview="on";
```

b) échelle logaritmique

Le choix du type d'échelle se fait à l'aide de la propriété `log_flags` : la valeur est constituée de trois lettres correspondant respectivement aux axes $x$, $y$ et $z$ (que $z$ soit utilisé ou non). Lorsqu'on veut une échelle *normale* (linéaire), on met la lettre `n`, pour une échelle *logarithmique*, la lettre `l`.
Par défaut, les échelles sont linéaires : `log_flags = "nnn"`
Dans le cas où vous avez besoin d'une échelle logarithmique ou semi-logarithmique, cette propriété doit donc être redéfinie. Exemple :

```
-->log_flags = "nln";
```

c) position des axes

Les propriétés sont `x_location` et `y_location`. Par défaut, `x_location="bottom"` et `y_location="left"`.
Pour déplacer l'axe des abscisses, on règle la propriété `x_location` sur

- `"top"`, pour le mettre en haut
- `"origin"`, pour le faire passer par l'origine du repère.

Pour déplacer l'axe des ordonnées, on règle la propriété `y_location` sur :

- `"right"`, pour le mettre à droite ;
- `"origin"`, pour le faire passer par l'origine.

Le plus souvent, on utilisera :

```
-->A.x_location="origin";A.y_location="origin";
```

d) titre du graphique

Contenu du titre : `title.text="`*`chaîne de caractères`*`"`. Exemple :

```
-->A.title.text="Quelle belle courbe !";
```

Taille de la police : `title.font_size=`*n*. $n \in \mathbb{N}$. Exemple, pour une taille moyenne :

```
-->A.title.font_size=3;
```

Position du titre : par défaut, il est en haut, au milieu. Si on veut le positionner manuellement, on indique l'abscisse et l'ordonnée du début du titre. Exemple :

```
-->A.title.position=[0,2.25];
```

e) légende des axes

On utilise : `x_label.text="`*`chaîne de caractères`*`"` et `y_label.text="`*`chaîne de caractères`*`"`.
Exemple, pour $\mathscr{E}_p(\theta)$ :

```
-->A.x_label.text="θ";A.y_label.text="Ep";
```

Remarque : si vous n'arrivez pas à entrer `θ`, vous pouvez faire du copié-collé ou vous contenter de `thêta`.
On peut également modifier la taille des caractères par : `x_label.font_size=`*n* et `y_label.font_size=`*n*.

f) courbe : épaisseur et couleur du tracé

Nous devons commencer par désigner la courbe, afin de lui appliquer des propriétés. Nous avons vu comment récupérer l'objet global "graphique", que nous avons nommé `a`, il nous faut maintenant récupérer l'objet "courbe", qui est un élément

du graphique. Pour Scilab, un graphique est structuré en arborescence et la courbe est un descendant de descendant de l'objet "graphique". Concrètement, notre courbe est l'objet désigné par : `A.children.children`.

Pour modifier sa couleur, nous disposons de la propriété `foreground`.

`foreground=`$n$ définit la couleur du tracé, $n \in \mathbb{N}^*$ :

- si $n = 1$ la courbe sera en trait plein noir (valeur par défaut si aucun style n'est indiqué)
- si $n > 1$ la courbe sera en couleur ; on peut voir la correspondance entre le nombre et la couleur (carte des couleurs) grâce à la commande `getcolor()`

Exemple : `foreground=5` permet d'obtenir une courbe rouge.

Il existe également une syntaxe alternative `foreground =color(`$r,v,b$`)`, permettant d'élargir la palette de couleur à l'aide du code RVB : les nombres $r$, $v$ et $b$ sont compris entre 0 et 255 et représentent respectivement le dosage du rouge, du vert et du bleu (cf. synthèse additive des couleurs) .

Exemple, toujours pour du rouge : `foreground=color(255,0,0)`

Pour modifier l'épaisseur de la courbe, nous disposons de la propriété `thickness`.

`thickness=`$n$ définit l'épaisseur du tracé en pixels. Par défaut $n$=1.

Exemple : pour doubler l'épaisseur de la courbe et la colorer en rouge, nous saisissons :

```
-->A.children.children.thickness=2; A.children.children.color(255,0,0);
```

### 4.4. Mise en forme complémentaire

a) <u>couleur du fond, encadrement et marges</u>

▪ Couleur du fond : la propriété est `background`, réglée par défaut sur `-2` : fond blanc. La syntaxe de base est `background=`$n$, $n \in \mathbb{Z}$. Rappel : la correspondance entre le nombre et la couleur est donnée par la *carte des couleurs*, accessible par la commande `getcolor()`. La syntaxe alternative `background=color(`$r,v,b$`)`, s'applique également. Exemple :

```
-->A.background=color(250,250,245);
```

▪ Encadrement (ou boîte) : la propriété est `box`, réglée par défaut sur `"off"`. Si on veut un cadre :

```
-->A.box="on";
```

▪ Marges : par défaut le réglage est automatique, et correspond à la propriété : `auto_margins = "on"`.

Pour modifier les marges, on utilise la propriété `margins`, réglée par défaut sur `[0.125,0.125,0.125,0.125]`, à peu de chose près… Ces 4 valeurs représentent respectivement les marges gauche, droite, supérieure, inférieure, indiquées en valeurs relatives : cela signifie ici que les marges représentent 12,5 % de la figure. Exemple de modification :

```
-->A.margins=[0.05 0.05 0.15 0.05];
```

Et pour revenir au réglage par défaut :

```
-->A.auto_margins = "on";
```

b) <u>épaisseur et couleur des axes</u>

▪ Épaisseur : la propriété est `thickness`, réglée par défaut sur `1`.

Si votre graphe est destiné à être projeté, épaissir les axes à `2` peut être intéressant :

```
-->A.thickness=2;
```

▪ Couleur : la propriété est `foreground`, réglée par défaut sur `-1`, équivalent à `0` ou `1` : axes noirs. La syntaxe est la même que pour `background`. Exemple :

```
-->A.foreground=color(90,90,90);
```

c) <u>ajouter une grille</u>

Il peut parfois être utile, pour aider à la lecture du graphe, d'ajouter une grille dans le fond. La propriété est `grid`, réglée par défaut sur `[-1,-1]` : pas de grille. La syntaxe est `grid=[`$n_x, n_y$`]` où $n_x$ et $n_y$ sont des nombres entiers issus de la *carte des couleurs*, indiquant les couleurs de la grille suivant x et y.

Outre la couleur, un autre réglage utile est le style de trait. La propriété est `grid_style`, réglée par défaut sur `[3,3]`. Pour connaître ces codes : `help polyline_properties` (c'est en bas de page).
Pour contrôler l'épaisseur de la grille, la propriété est `grid_thickness`. Par défaut, l'épaisseur de la grille est identique à celle des axes, soit `grid_thickness=[1,1]`, si on n'a pas modifié `thickness`. Mais si on a épaissi les axes, la grille le sera aussi. On peut donc être amené à rétablir le réglage initial de l'épaisseur de la grille.

En résumé, on peut donc utiliser par exemple :

```
-->A.grid=[16,16];A.grid_style=[7,7];A.grid_thickness=[1,1];
```

d) recadrage

La propriété est `data_bounds`. Le cadrage par défaut est automatique, de façon à afficher toutes les valeurs.
Syntaxe : `data_bounds=[`$x_A, y_A; x_B, y_B$`]`, où A($x_A, y_A$) et B($x_B, y_B$) sont deux points situés respectivement en bas à gauche de la figure et en haut à droite. Exemple :

```
-->A.data_bounds=[-%pi,0;%pi,2.2];
```