Un livre de Wikilivres.

Découvrir Scilab

Une version à jour et éditable de ce livre est disponible sur Wikilivres,
une bibliothèque de livres pédagogiques, à l'URL :
http://fr.wikibooks.org/wiki/D%C3%A9couvrir_Scilab

Vous avez la permission de copier, distribuer et/ou modifier ce document
selon les termes de la Licence de documentation libre GNU, version 1.2
ou plus récente publiée par la Free Software Foundation ; sans sections
inaltérables, sans texte de première page de couverture et sans Texte de
dernière page de couverture. Une copie de cette licence est inclue dans
l'annexe nommée « Licence de documentation libre GNU ».

Interface

Table des matières — Index

---

1. Interface utilisateur

Saisie des commandes et affichage numérique

Scilab est un logiciel qui s'utilise en ligne de commande. L'écran se compose d'une zone de saisie dans laquelle on va taper des commandes.

L'invite de la ligne de commande *(prompt)* est constituée d'une « flèche » : deux tirets et un signe supérieur `-->`. L'instruction est tapée puis validée avec la touche de retour chariot (`[↲]`, `[Enter]` ou `[Return]`). Le résultat est affiché à la suite, sauf si la ligne se termine par un point-virgule auquel cas le résultat est caché.

Par exemple

- sans point-virgule

```
-->a=1[↲]
 a  =

    1.

-->A=2[↲]
 A  =

    2.

-->a+A[↲]
ans  =

   3.
```

- avec points-virgules

```
-->a=1;[↲]

-->A=2;[↲]

-->a+A[↲]
ans  =

   3.
```

Lorsque l'on écrit une ligne, on peut utiliser les touches classiques d'édition (variable selon le système et le type du clavier) :

- `[→]` : avancer d'une lettre ;
- `[←]` : reculer d'une lettre ;

- touches de suppression ([Del], [Suppr], [⟵]) ;
- touche de retour chariot [↵], [Enter] ou [Return] : validation et exécution de la commande ;
- [↑] et [↓] : permettent de faire défiler les commandes tapées précédemment (historique des commandes).

L'endroit où se place le caractère (lettre, chiffre, signe…) tapé est signalé par le curseur, un trait de soulignement clignotant (Windows XP) ou un rectangle (MacOS X). L'édition se fait en insertion, c'est-à-dire que si le curseur est au milieu d'une ligne, les caractères suivants sont décalés d'une position à droite lorsque l'on entre un caractère.

On peut, si on le désire, fractionner l'édition sur plusieurs ligne. Pour cela, il faut terminer la ligne par trois points « . . . ».

Exemple

```
-->a=...[↵]
-->5[↵]
 a  =

    5.
```

Ceci est bien sûr plutôt intéressant lorsque la ligne à taper est longue.

À l'inverse, on peut mettre plusieurs instructions sur une même ligne en les séparant d'une virgule, ou bien d'un point-virgule si l'on ne veut pas que le résultat soit affiché.

Note

Par la suite, le retour chariot [↵] sera omis, un retour à la ligne après une ligne commençant par l'invite --> l'indiquera implicitement.

Chaînes de caractère

Par défaut, une suite de caractères est interprété comme une commande ou comme une variable. Pour indiquer que c'est une chaîne de caractères, il faut la mettre entre guillemets simples (apostrophes, *single quote*) « ' » ou entre guillemets doubles *(double quote)* « " ».

Lorsque la chaîne doit contenir une apostrophe, on redouble l'apostrophe. De même, lorsque la chaîne doit contenir des guillemets, on redouble ces guillemets.

Exemple

```
-->l'amertume de la "bière"
  !--error 4
undefined variable : l


-->'l''amertume de la ""bière""'
 ans  =

 l'amertume de la "bière"

-->"l''amertume de la ""bière"""
 ans  =
```

```
l'amertume de la "bière"
```

Les chaînes de caractères sont utilisées pour définir certains paramètres de fonctions, ou bien pour afficher des étiquettes et légendes sur les graphiques.

Éditeur SciNotes

Scilab dispose d'un éditeur de programme, SciNotes, qui peut être lancé en tapant `edit` dans la ligne de commande. On peut aussi taper `scinotes`, mais cette solution est sensible à un changement du nom de l'éditeur ; par exemple, avant la version 5.3.0 (2010), l'éditeur était SciPad[1].

Il s'agit d'un éditeur multiligne : lorsque l'on appuie sur le retour chariot, cela ne valide pas la commande mais passe simplement à la ligne suivante. Les touches `[↑]` et `[↓]` permettent de changer de ligne, d'aller éditer une autre ligne.

Ce qui est tapé dans cet éditeur peut ensuite être sauvegardé dans un fichier .SCI ou bien directement exécuté dans Scilab (l'ancienne extension était .SCE).

Affichage graphique

L'affichage graphique (tracé d'une courbe, dessin…) se fait dans une fenêtre dédiée que nous appelerons « fenêtre graphique ». Cette fenêtre s'affiche automatiquement dès la première instruction graphique. Par exemple, taper `clf` puis valider : cela affiche une fenêtre graphique vide.

Aide en ligne

Scilab dispose d'une aide en ligne. Pour cela, taper `help`.

Cela affiche une fenêtre Browse Help (« naviguer dans l'aide ») avec une arborescence à gauche. Lorsque l'on clique sur un dossier de l'arborescence, une branche se développe, faisant apparaître les articles. Chaque article correspond à une commande ou à une fonction prédéfinie ; il suffit de cliquer dessus pour que le contenu de l'aide — disponible en français, mais quelques articles restent en anglais — s'affiche dans la partie droite de la fenêtre.

Lorsque l'on clique sur le bouton **Rechercher**, un champ de recherche apparaît dans lequel on peut entrer un mot-clef ; lorsque l'on appuie sur le retour chariot `[↵]`, le programme effectue une recherche et affiche les article correspondants dans la zone de texte située dans la partie gauche de la fenêtre. il suffit de cliquer dessus pour que le contenu de l'article s'affiche dans la partie droite de la fenêtre. On revient à une navigation par arborescence en cliquant sur le bouton **Arborescence**.

Fermer le programme

Scilab peut être fermé de la manière habituelle selon les systèmes d'exploitation : en général bouton en forme de croix ou menu **Fichier | Quitter**. On peut aussi

- taper `exit` en ligne de commande ;
- lorsque l'on est dans l'environnement initial, on peut également taper `quit` (voir aussi *Environnement*).

Spécificités selon les systèmes d'exploitation

Sous Microsoft Windows

Éditeur de ligne de commande

Voici les effets des touches suivantes dans l'éditeur de ligne de commande :

- `[↖]` ou `[Home]` : aller en début de ligne ;
- `[Fin]` ou `[End]` : aller en fin de ligne ;
- `[Échap]` ou `[Esc]` efface la ligne courante.

On peut utiliser les actions classiques de la souris, par exemple sélectionner une ligne de texte et faire du copier-coller :

- en utilisant le menu contextuel accessible en cliquant avec le bouton droit [2],
- avec les boutons d'édition situés en haut de la fenêtre, ou
- avec les raccourcis clavier, par exemple `[Ctrl]+[C]` et `[Ctrl]+[V]`.

Attention

l'utilisation de `[Ctrl]+[C]` lorsqu'aucun texte n'a été sélectionné avec la souris a un effet totalement différent, voir le chapitre *Environnement*. Il est donc préférable d'éviter le raccourci clavier pour la fonction copier.

Le texte ne peut être sélectionné qu'avec la souris ; la combinaison de touches `[⇑]+[→]` ou `[⇑]+[←]`, utilisable dans de nombreuses autres applications, ne fonctionne pas[3]. Il est possible de copier du texte dans une autre application (par exemple un éditeur de texte ou bien du texte affiché par un navigateur Internet) et de le coller dans l'éditeur de ligne de commande de Scilab. On peut ainsi utiliser des instructions écrites au préalable, éventuellement par d'autres personnes (voir aussi *Programmation* > *Chargement d'une fonction*).

Notons que le raccourci Scilab pointe vers un programme nommé WScilex.EXE situé dans `~\bin\`, le tilde `~` indiquant ici le répertoire (dossier) d'installation (par défaut `C:\Program Files\Scilab-4.0\` pour la version 4.0). Il existe un autre programme dans le même répertoire, Scilex.EXE, qui est un éditeur en ligne de commande Scilab mais sans « l'environnement habituel » Microsoft Windows : la fenêtre n'a pas de menu ni de bouton, seule la ligne de commande est active, et on ne peut pas copier le texte de la fenêtre (on en peut pas le sélectionner avec la souris), ni coller du texte copié d'une autre application Microsoft Windows. On peut tout de même lancer l'aide en tapant `help`.

Éditeur de programme SciNotes

Concernant l'éditeur de programme SciNotes, il est possible de faire du copier-coller depuis ou vers une autre application Microsoft Windows, et les combinaisons de touches `[⇑]+[→]` et `[⇑]+[←]` permettent de sélectionner du texte.

Aide en ligne

On peut lire directement le fichier d'aide au format CHM (*compiled HTML*, un format d'aide propre à Microsoft Windows). Ce fichier contient les mêmes informations que l'aide en ligne accessible par `help`, seule change l'interface ; le fichier s'appelle `scilab_fr_FR_help.chm` (la nomenclature était différente dans les versions précédentes, le fichier s'est appelé `sciman-fr-3.1.1.CHM`, `man-fr-scilab-4.0.CHM`)[4]. Il se trouve dans le dossier `C:\Program Files\scilab-5.4.0\modules\helptools\chm`.

On lance cette aide avec un double-clic sur l'icône dans l'Explorateur Windows. La fenêtre comporte deux parties :

- à gauche, un panneau de navigation avec trois onglets :
  - **sommaire** : c'est une arborescence, similaire à l'aide `help` ;
  - **index** : permet une recherche par mots-clefs (des mots-clefs sont associés aux articles) ;
  - **rechercher** : permet une recherche par chaîne de caractère (lorsque l'on cherche un mot qui n'a pas été entré comme mot-clef) ;
- à droite, un panneau où s'affiche l'article sélectionné dans le panneau de navigation.

Sous MacOS X

Scilab fonctionne dans l'environnement X11 ; on ne peut pas copier un texte dans une autre application puis le coller dans l'éditeur en ligne de Scilab. Par contre, c'est possible dans SciPad. SciPad peut être lancé en cliquant sur le bouton **Editor** en haut de la fenêtre.

Il est possible de copier du texte d'une application extérieure (y compris sous Aqua[5]) de manière classique (par exemple avec +`[C]`), puis de le coller dans SciPad avec le menu **Edit | Paste** ou bien le raccourci clavier `[Ctrl]+[V]`.

Note

Les raccourcis pour le copier-coller sont différents entre les applications sous Aqua (touche) et SciPad (touche `[Ctrl]`).

Les touches de fonction classiques fonctionnent normalement dans l'éditeur de ligne de commande :

- `[↖]` : aller en début de ligne ;
- `[↘]` : aller en fin de ligne.

Voir aussi

Dans Wikipédia

- Aqua
- Chaîne de caractères
- Man
- X11

Notes

1. SciPad est toujours disponible en tant que produit à part, sur http://sourceforge.net/p/scipad/
2. pour les droitiers, ou le bouton gauche si la souris est configurée pour les gauchers
3. `[⇑]` désigne la touche permettant d'accéder aux capitales, souvent notée `[Maj]` pour « majuscules » bien que le terme soit impropre (voir *capitale et majuscule sur Wikipédia*) ; cette touche est appelée *« shift »* en anglais
4. le terme « man » fait référence à la fonction Unix permettant d'afficher les pages de manuel.
5. Aqua est l'environnement graphique « naturel » de MacOS X

Introduction < ↑ > *Calculs élémentaires*

Calculs élémentaires

Table des matières — Index

---

2. Calculs élémentaires

Scilab peut s'utiliser comme une calculatrice scientifique, de type calculatrice graphique. Il suffit de taper une formule dans une syntaxe relativement standard, et d'appuyer sur le retour chariot pour avoir le résultat.

Opérateurs et fonctions classiques

Scilab utilise les fonctions et opérateurs classiques :

- `+` (addition), `-` (soustraction), `*` (produit), `/` (division), `^` ou `**` (élévation à une puissance),
- `modulo(a, b)` : reste entier de *a*/*b*, `pmodulo(a, b)` donne un résultat toujours positif,
- `sqrt()` (racine carrée), `nthroot(x, n)` (racine *n*-ième de *x*)
- `cos()`, `sin()`, `tan()` pour les fonctions trigonométriques de base, `acos()`, `asin()`, `atan()` pour leurs réciproques, `cotg()` pour la cotangente ;
- les fonctions trigonométriques hyperboliques `cosh()`, `sinh()`, `tanh()`, `acosh()`, `asinh()`, `atanh()`
- `log()` pour le logarithme népérien (ln), `log10()` pour le logarithme en base 10 ($\log_{10}$), `log2()` pour le logarithme en base 2 ($\log_2$),
- `exp()` pour la fonction exponentielle,
- `floor()` pour la partie entière, `int()` pour la troncature (arrondit à l'entier vers 0, différent de `floor` pour les nombres négatifs), `round()` pour l'arrondi au plus proche (0,5 → 1), `abs()` pour la valeur absolue,
- `erf()` pour la fonction d'erreur de Gauss,
- `rand` pour avoir un nombre aléatoire entre 0 et 1, suivant une loi uniforme ; `rand(1,'normal')` pour avoir un nombre aléatoire suivant une loi normale centrée réduite (moyenne 0, variance 1) ;
- la factorielle de *n* peut s'obtenir par `prod(1:n)` ou bien par `gamma(n+1)` ;
- …

Le séparateur décimal est le point.

Les puissances de 10 se notent avec la lettre « e » en minuscule ou en capitale : $5{,}2 \cdot 10^3$ s'écrit `5.2e3` ou `5.2E3`, $5{,}2 \cdot 10^{-3}$ s'écrit `5.2e-3` ou `5.2E-3`. On peut aussi utiliser le « d » minuscule ou capital.

Par défaut, Scilab affiche les résultats avec dix caractères (signe et point décimal inclus). On peut modifier le nombre de caractères avec la fonction `format(n)` où *n* est le nombre de caractères.

Mentionnons également la fonction `clear` qui permet d'effacer toutes les variables (dont les fonctions définies par l'utilisateur). Cela peut être utile lorsque l'on a procédé par essai-erreur et que l'on veut remettre les « compteurs à zéro ».

Vocabulaire

Dans Scilab, une « fonction interne[1] » est une fonction prédéfinie. Par opposition, les fonctions définies par l'utilisateur sont appelées « fonctions extérieures ».

Constantes particulières

Pour entrer l'imaginaire *i*, il faut utiliser `%i` ; il figure simplement sous la forme « `i` » dans les résultats.

Pour entrer l'infini ∞, il faut utiliser `%inf` ; il figure simplement sous la forme « `inf` » dans les résultats.

La valeur de π s'obtient par `%pi`, et la constante de Neper *e* par `%e`.

Lorsqu'une opération invalide ce produit, certains programmes retournent un nombre particulier appelé « NaN » pour *not a number* (littéralement « ceci n'est pas un nombre »)[2]. Dans Scilab, cette valeur est obtenue par `%nan` et s'affiche sous la forme `Nan`.

Variables

On peut définir des variables simplement sous la forme *`NomVariable=valeur`*. *NomVariable* est une chaîne de caractères commençant par une lettre ; Scilab est sensible à la casse, c'est-à-dire que les variables « a » et « A » sont deux variables différentes. Le type de la variable (entier, réel, imaginaire) est géré automatiquement.

La variable `ans` contient le dernier résultat (pour *answer*, « réponse » en anglais).

La fonction `who` affiche les variables déclarées (*who?* signifie « qui ? » en anglais). On remarque que dès l'ouverture, Scilab a défini de nombreuses variables, notamment des variables dites « d'environnement », c'est-à-dire des paramétrages de fonctionnement de Scilab.

Exemples

Exemple 1

```
-->asin(sqrt(2)/2)
 ans  =

    0.7853982

-->ans/%pi*180
 ans  =

    45
```

puisque sin(45°) = √2/2.

Exemple 2

```
-->%e^(%i*%pi/2)
 ans  =

    6.1273D-17 + i
```

notez que dans l'absolu, on devrait obtenir *i*.

Exemple 3

Calcul de la pression, en Pa, de 2,5 moles de gaz parfait dans un volume de un litre (10^{-3} m^3) à une température de 300 K (26,85 °C)

```
-->n=2.5;R=8.314472;T=300;V=1e-3;

-->n*R*T/V
 ans  =

    6235854
```

soit environ $6{,}2 \cdot 10^6$ Pa.

Complexes

Les nombres complexes sont entrés la forme `a + b*%i`. On peut utiliser la commande `complex(a, b)` pour créer ce nombre. Les lignes suivantes sont équivalentes :

```
z = 1 + 2*%i
z = complex(1,2)
```

Les commandes `real(z)` et `imag(z)` donnent respectivement la partie réelle et la partie imaginaire du nombre complexe. la conjugaison complexe $\bar{z}$ s'obtient par `conj(z)`.

Le module d'un nombre complexe s'obtient avec la commande `abs(z)`. L'argument complexe de $z = a + bi$ s'obtient par

```
phi = atan(b, a)
phi = atan(imag(z), real(z))
```

La commande `isreal(z)` teste si un nombre est réel, ou bien si une matrice ne contient que des réels.

Pour multiplier par l'imaginaire pur i, il vaut mieux utiliser la commande `imult(a)` que `%i*a`.

Booléens

Lorsque l'on assigne une valeur booléenne, on utilise `%t` pour « vrai » *(true)* et `%f` pour « faux » *(false)*. Le résultat de l'opération affiché est respectivement `T` ou `F`.

L'opérateur *ou* (OR) est noté « `|` » (tube), *et* (AND) est noté « `&` » (esperluette), *non* (NOT) est noté « ~ » (tilde). Le *ou exclusif* (XOR) s'obtient avec ~=.

Exemple

```
-->(%t & %f) | %f
ans =

F.
```

Un booléen s'obtient en général en comparant deux valeurs, avec les relations d'égalité == et de différence <> ou ~=, et les relations d'ordre <, <=, > et >=.

Exemple

```
-->a=3
 a  =

    3.

-->a==3
 ans  =

  T

-->a<2
 ans  =

  F
```

Il existe également un certain nombre de commandes de test renvoyant un booléen, par exemple :

- `isempty(a)` : renvoie « vrai » si c'est la variable vide, « faux » sinon[3] ;
- `isnan(a)` : vérifie si la variable a la valeur NaN (*not a number*, résultat d'une opération invalide) et renvoie « vrai » dans ce cas ;
- `isinf(a)` : vérifie si la variable a une valeur infinie et renvoie « vrai » dans ce cas ;
- `isequal(a, b, …)` : vérifie si les variables *a*, *b*, … sont toutes égales.

Si la variable *x* est un booléen, la commande `bool2s(x)` *(boolean to standard)* transforme la valeur « faux » en « 0 » et la valeur « vrai » en « 1 ». Si c'est un nombre, la commande transforme toute valeur non nulle en « 1 », et une valeur nulle en « 0 ».

Polynômes et fractions rationnelles

Scilab manipule les polynômes et fractions rationnelles formels, c'est-à-dire non pas en tant que fonction, mais en tant qu'objets mathématiques, éléments du corps des polynômes ou des fractions rationnelles.

La première opération pour définir un polynôme est de définir le monôme *x*. Ceci se fait par la commande `x = poly(0, 'x')`. Cette commande indique que la variable *x* est un polynôme (commande `poly()`) dont l'indéterminée est notée « x » (le `'x'` situé à l'intérieur de la parenthèse) et la racine est 0.

Puis, le polynôme ou la fraction rationnelle est définie par une formule contenant ce *x*.

Exemple de définition de polynôme

```
-->x = poly(0, 'x'); p = x ^ 2 + 2 * x + 1
 p  =

               2
    1 + 2x + x
```

Exemple de définition de fraction rationnelle

```
-->x = poly(0, 'x'); q = (1 + x) / (1 - x)
 q  =

    1 + x
```

```
   -----
   1 - x
```

On peut aussi utiliser directement les constantes spéciales `%s` et `%z`, qui sont l'équivalent de `%s = poly(0, 's')` et `%z = poly(0, 'z')` :

```
-->p = %s^ 2 + 2*%s + 1
 p  =

              2
   1 + 2s + s
```

Les polynôme et fractions rationnelles se manipulent ensuite de manière formelle comme les autres variables. Voici des exemples à partir des objets définis ci-dessus.

Exemple

```
-->1/q
 ans  =

   1 - x
   -----
   1 + x

--> p*q
 ans  =

               2    3
   1 + 3x + 3x  + x
   ----------------
        1 - x
```

La fonction `derivat()` fait la dérivée formelle du polynôme ou de la fonction rationnelle.

De manière interne, Scilab représente les polynômes et les fractions rationnelles comme des listes typées. Ainsi, avec les exemples précédents,

- `p(1)` va nous renvoyer $1+2x+x^2$;
- `q(1)` renvoit la liste des noms des champs de la liste ;
- `q(2)` (ou `q('num')`, `q.num`) donne `1+x` ;
- `q(3)` (ou `q('den')`, `q.den`) donne `1-x` ;

et `p(a)` donne une erreur quelque soit la valeur de *a* (sauf 1), `q(a)` donne une erreur quelque soit la valeur de *a* (sauf 1, 2, 3 et 4).

On ne peut donc pas utiliser un polynôme ou une fraction rationnelle comme une fonction classique. L'évaluation, c'est-à-dire l'utilisation du polynôme ou de la fraction rationnelle comme fonction numérique, se fait avec la fonction `horner` :

`horner(f,val)` calcule $f(val)$.

Exemple

```
-->horner(p,2)
```

```
ans  =

   9.
```

Fonctions spécifiques aux polynômes

La fonction `roots(p)` donne les racines du polynôme *p*. La fonction `factors(p)` donne la décomposition en facteurs irréductibles.

La fonction `coeff(p)` donne une matrice dont les coefficients sont les coefficients du polynôme *p*. ; on peut extraire le coefficient de degré *n* avec `coeff(p, n)`.

La fonction `varn(p)` renvoie le nom de l'indéterminée de *p* (ici, *x*).

En fait, la commande `x = poly(0, 'x')` définit que *x* est le polynôme le plus simple dont l'indéterminée est le caractère « x » et dont la racine est 0, c'est-à-dire le monôme *x*. La commande `A = poly(2, 'x')` définit de même le polynôme le plus simple ayant pour racine 2, c'est-à-dire *x* - 2.

Note

Si l'on écrit `x = poly(2, 'x')`, alors la variable *x* sera le polynôme « *x* - 2 ». Il faut bien distinguer la notion d'indéterminée (chaîne de caractère attachée au polynôme) et la variable *x*. Si l'on écrit `A = poly(2, 'x')` puis `x = 1`, A sera toujours le polynôme « *x* - 2 », l'indéterminée ne sera pas remplacée par la valeur de la variable *x*.

Cette fonction permet de définir un polynôme ayant plusieurs racines, en utilisant une matrice ligne au lieu d'un simple nombre. Par exemple, `A=poly([1 2],'x')` définit le polynôme le plus simple ayant pour racines 1 et 2, soit le polynôme x^2 - 3*x* + 2.

On peut aussi utiliser cette fonction pour générer un polynôme dont les coefficients sont contenus dans une matrice ligne, en ajoutant le caractère `'c'` à la fin des arguments. Par exemple, `A = poly([a0, a1, …, an], 'x', 'c')` définit le polynôme $a0 + a1 \cdot x + \ldots + an \cdot x^n$.

Fonctions spécifiques aux fractions rationnelles

La fraction rationnelle *q* est le rapport de deux polynômes, notés *q.num* (numérateur) et *q.den* (dénominateur), qui peuvent se manipuler comme des polynômes normaux.

Exemple

```
-->q.num
 ans  =

   1+x

-->q.num^2
 ans  =

              2
   1 - 2x + x
```

La fonction `simp()` fait la simplification de la fraction rationnelle.

Chaînes de caractères

Créer une chaîne

Comme indiqué précédemment, une chaîne de caractères est simplement délimitée par des guillemets simples ou doubles :

```
'a', "a"
```

> **Nous recommandons l'utilisation des guillemets doubles** : en effet, le guillemet simple sert aussi à la transposition des matrices, ce qui peut porter à confusion.

Rappelons que pour faire figurer une apostrophe ou un guillemet dans la chaîne de caractères, il faut le doubler, `''` ou `""` (voir *Interface utilisateur* > *Chaînes de caractère*).

Mais une chaîne de caractères peut aussi être obtenues à partir des codes ASCII[4]. La fonction `ascii()` renvoit :

- si on lui donne un caractère ou une chaîne : le code ASCII du caractère ou le vecteur contenant les codes ASCII de chaque caractère de la chaîne ;
- si on lui donne un entier ou un vecteur d'entiers : le caractère dont le code ASCII est l'entier, ou la chaîne de caractères.

Exemple

```
--> ascii("a")
 ans  =

    97.

--> ascii(65)
 ans  =

 A
```

La fonction `char()` crée un vecteur colonne de caractères à partir d'une matrice de nombres, par exemple

```
-->char([64, 65])
 ans  =

 @A
```

Notons que `@A` est une chaîne de caractères.

```
-->char([64, 65 ; 66, 67])
 ans  =

!@A  !
!    !
!BC  !
```

Par contre, appliquée à une matrice de chaînes de caractères, cette fonction concatène les lignes.

```
-->char(["a", "b" ; "c", "d"])
 ans  =

!ab  !
!    !
!cd  !
```

Rappelons quelques caractères ASCII spéciaux utiles :

- `9` : tabulation (*horizontal tabulation*, `HT`) ;
- `10` : saut de ligne (*line feed*, `LF`) ;
- `13` : retour de chariot (*carriage return*, `CR`) ;
- les caractères affichables/imprimables vont du code `32` au code `126` :
  - les chiffres vont de `48` (pour 0) à `57` (pour 9),
  - les lettres capitales vont de `65` (A) à `90` (Z),
  - les lettres minuscules vont de `97` (a) à `122` (z).

> *Pour plus de détails voir :* ***w:fr:American Standard Code for Information Interchange#Table des 128 caractères ASCII*** .

Rechercher, remplacer, découper une chaîne

La commande `strsubst()` fait un « rechercher/remplacer » dans une chaîne. Par exemple, pour remplacer une virgule par un point dans la chaîne de caractères X :

```
strsubst(X, ",", ".")
```

La commande `length(X)` indique la longueur de la chaîne X.

La commande `part` permet d'extraire des caractères d'une chaîne. Les positions des caractères sont décrites dans un vecteur. Par exemple, pour extraire les cinq premiers caractères d'une chaîne X :

```
part(X, 1:5)
```

D'autres commandes sont présentées dans le chapitre *Programmation > Analyse et construction de chaînes de caractères*.

Utiliser une chaîne

L'opérateur « `+` » permet la concaténation de deux chaînes de caractère. La fonction `strcat()` concatène une matrice de chaînes.

```
-->a = strcat(["a", "b" ; "c", "d"])
 a  =

 acbd
```

La fonction `string` permet de transformer un nombre, un booléen, un vecteur, une matrice, … en une chaîne

de caractères, par exemple pour permettre son utilisation par une fonction d'affichage (avec la fonction `xstring`, `input`, ...) ou bien pour fabriquer un nom de fichier.

La fonction `evstr()` permet de transformer une chaîne en une expression Scilab (on parle « d'évaluation de la chaîne »). La fonction `execstr()` permet d'exécuter le contenu de la chaîne.

Exemple

```
-->a = "1";

-->evstr(a)*2
 ans  =

    2.

-->b = "x"; c = "=5"; execstr(b+c)

-->x^2
 ans  =

    25.
```

la chaîne b+c est `"x=5"`, cette chaîne est exécutée, ce qui signifie que la valeur 5 est attribuée à la variable x, ce que confirme le calcul fait par la suite.

La commande `evstr()` permet donc de transformer une chaîne de caractères — ou une matrice de chaînes — ne contenant que des chiffres en un nombre — ou une matrice de nombres. Si la chaîne de caractères mélange des chiffres et d'autres caractères, on peut alors utiliser la commande `strtod()` *(string to double)*.

Variable chaîne comme matrice

Une variable contenant une chaîne de caractères peut être utilisée pour générer une matrice. Pour cela, on fait comme si la variable était elle-même une matrice acceptant une matrice rectangulaire ne contenant que des uns (`ones()`) en indice. Par exemple :

```
-->chaine = "a"
 chaine  =

 a

-->chaine([1, 1 ; 1, 1])
 ans  =

!a  a  !
!      !
!a  a  !

-->chaine(ones(1, 5))
 ans  =

!a  a  a  a  a  !
```

La concaténation d'un vecteur ligne avec une chaîne permet alors d'intercaler des caractères.

```
-->strcat(chaine(ones(1, 5)), "b")
```

```
ans  =

abababababa
```

Cela permet de fabriquer des chaînes particulières de manière rapide.

Matrices

Scilab a été conçu pour le calcul matriciel. Les éléments des matrices peuvent être de tout type (nombre réel, nombre complexe, booléen, polynôme, fraction rationnelle, chaîne de caractères…).

Définir une matrice

Pour définir une matrice à partir de ses coefficients, on les place entre deux crochets `[…]`. Les éléments d'une ligne sont séparés d'un espace ou d'une virgule, les lignes sont séparées retour à la ligne ou d'un point-virgule. À l'affichage, une matrice de nombres est représentée comme un tableau sans encadrement ; une matrice de chaînes de caractères est encadré par des points d'exclamation.

Exemple

```
-->[1,0,0;0,1,0;0,0,1]
ans =

1.    0.    0.
0.    1.    0.
0.    0.    1.
```

La matrice vide est notée par `[]`.

L'expression `M(i,j)` désigne l'élément (*i*, *j*) de la matrice M (ligne *i*, colonne *j*).

Le caractère `:` (deux-points) signifie « tous les indices », par exemple `M(1, :)` est la première ligne de la matrice (c'est un vecteur ligne). Le caractère `$` (dollar) désigne le dernier indice (ligne ou colonne) d'une matrice. L'expression `i1:i2` désigne tous les indices entre i_1 et i_2. Par exemple, si l'on veut effectuer l'opération M(*i* + 1, *j*) - M(*i*, *j*) pour tous les indices, on peut écrire simplement

```
difference = M(2:$, :) - M(1:$-1, :)
```

(note que la fonction `diff(M)` donne le même résultat).

Par ailleurs, l'expression `N1:N2` permet aussi de générer une matrice-ligne dont le premier coefficient est N_1, le dernier est inférieur ou égal à N_2, et le pas entre les coefficients est 1. L'expression `N1:pas:N2` permet de générer une matrice-ligne en choisissant le pas. Par exemple

```
-->1.1:5.2
ans =

1.1    2.1    3.1    4.1    5.1

--> 1:2:5
ans =
```

```
1    3    5

-->"a":"d"
ans =

abcd
```

On notera que dans le dernier exemple, `"a":"d"` ne crée pas une matrice mais une chaîne de caractères.

On peut aussi utiliser la fonction

```
linspace(x1, x2, n)
```

pour générer une matrice ligne de n valeurs espacées régulièrement entre x_1 et x_2.

La fonction

- `zeros(`*m*`,`*n*`)` crée une matrice $m \times n$ remplie de 0 ;
- `emptystr(`*m*`, `*n*`)</codes> crée une matrice `*m*`×`*n*` de chaînes de caractère vides ;`
- `ones(`*m*`,`*n*`) crée une matrice `*m*`×`*n*` remplie de 1 ;`
- `eye(`*n*`,`*n*`)` crée une matrice unité $n \times n$;
- `diag(V)`, ou V est une matrice ligne ou une matrice colonne, crée la matrice diagonale : matrice carrée dont les termes de la diagonale sont les éléments de V et les autres termes sont nuls.

On peut aussi passer une matrice M en paramètre de ces fonctions ; elles créent alors une matrice de même dimension que la matrice M. Par exemple, `M = zeros(M)` met tous les coefficients de M à zéro.

Par défaut, une matrice numérique est de type « double précision ». Si l'on n'a pas besoin d'une telle précision, c'est un gaspillage de mémoire. Pour créer une matrice $m \times n$ d'entiers non signés codés sur 8 bits (soit des valeurs allant de 0 à 255), on peut utiliser la fonction `uint8()` *(unsigned integer)* sur une matrice existante :

```
uint8(zeros(m, n))
```

Mais cela revient à créer une énorme matrice ($m \times n \times 8$ octets) pour ensuite la réduire (à $m \times n \times 1$ octets). Or, nous supposons ici que la taille de la matrice $m \times n$ est important.

L'idéal est donc de :

- partir du nombre en double précision 0 (zéro) et le transformer en entier non signé, avec la fonction `uint8()` ;
- redimensionner cette matrice 1×1 avec la fonction `resize_matrix()`

ce qui initialise la matrice avec des zéros :

```
resize_matrix(uint8(0), m, n)
```

Si f est une fonction extérieure (c'est-à-dire par définie par `deff` ou par `function`, voir *Calcul numérique* > *Définition de fonctions extérieures*), et que x et y sont des vecteurs, alors la fonction `feval` permet de bâtir une matrice $z = f(x, y)$

`z = feval(x, y, f)` : on a $z(i, j) = f(x(i), y(j))$

Ajouter ou supprimer des lignes, des colonnes

Considérons une matrice M. Comme indiqué précédemment, `M(i, :)` crée le vecteur ligne composé des coefficients de la ligne *i*, et `M(:, j)` crée le vecteur colonne composé des coefficients de la colonne *j*.

Si la matrice M est une matrice par blocs composée de quatre blocs A, B, C et D (eux-même des matrices), on peut créer M simplement par

```
M = [A, B ; C, D]
```

Si l'on veut insérer un vecteur ligne L après la ligne *i* de la matrice, on peut écrire

```
M = [M(1:i, :) ; L ; M(i+1:$, :)]
```

et si l'on veut supprimer la ligne *i*, en faisant remonter les lignes suivantes, il suffit d'y mettre la matrice vide :

```
M(i, :) = []
```

On peut procéder de même avec les colonnes.

Fonctions et opérateurs sur les matrices

Toutes les fonctions numériques s'appliquent à des matrices ; par exemple si M est une matrice, alors `cos(M)` sera la matrice dont les coefficients sont les cosinii des coefficients de M.

La fonction `exp(M)` calcule l'exponentielle de chaque terme. L'exponentielle de la matrice $(\sum(1/k!)M^k)$ s'obtient avec la fonction `expm(M)`.

La fonction `size(M)` renvoie la taille de la matrice sous la forme d'une matrice 2×1 contenant le nombre de lignes puis le nombre de colonnes. La fonction `size(M, "*")` renvoie le produit des deux valeurs, soit le nombre total d'éléments dans la matrice. On peut avoir le nombre de lignes avec `size(M, "r")` *(row)*, et le nombre de colonnes avec `size(M, "c")`.

Les opérations spécifiques aux matrices classiques sont :

- la transposition : il suffit de mettre une apostrophe `'` après la matrice ;
- somme, différence et produit de matrices : `+`, `-` et `*` ;
- produit, rapport et élévation à la puissance élément par élément : `.*`, `./` et `.^` ;
- le produit tensoriel `.*.` ;
- le déterminant d'une matrice carrée M : `det(M)` (`determ()` pour une matrice de polynômes et `detr()` pour une matrice de fractions rationnelles) ;
- la trace d'une matrice carrée M : `trace(M)` ;
- l'inverse d'une matrice inversible M : `inv(M)` ;

Vecteur

Un vecteur est une matrice ligne ou colonne (matrice $1 \times n$ ou $n \times 1$) ; l'expression `V(i)` désigne la composante *i* du vecteur V.

Si l'on veut avoir la taille d'un vecteur `v` sous la forme d'un entier, on peut utiliser

vecteur ligne : `size(v, "r")`
vecteur colonne : `size(v, "c")`
dans les deux cas : `max(size(v))` ou bien `size(v, "*")`

La norme d'un vecteur `v` s'obtient simplement par

```
nv = norm(v);
```

Si V1 et V2 sont des vecteurs colonnes, alors le produit scalaire est

```
V1' * V2;
```

si ce sont des vecteurs lignes, le produit scalaire est

```
V1 * V2'
```

Il n'existe pas de fonction « toute faite » pour calculer le produit vectoriel *(cross product)*.

Vecteurs ou matrices utilisés comme suite de nombres

Un vecteur peut être utilisé pour stocker une suite de nombre. cela peut être les valeurs successives d'une suite, au sens mathématique du terme, ou bien des données recueillies expérimentalement. Quelques fonctions sont relatives à ces applications.

On peut faire :

- la somme de tous les éléments de la matrice : `sum(M)` ;
- le produit de tous les éléments de la matrice : `prod(M)`.

La fonction `max(M)` renvoie la plus grand élément de la matrice, et la fonction `min(M)` renvoie le plus petit.

Si l'on tape `[m,k]=max(A)`, alors Scilab met la valeur du maximum dans la variable *m* et sa position dans la matrice (sous la forme d'un vecteur ligne [*i*,*j*]) dans la variable *k* — on peut utiliser d'autres noms de variable que *m* et *k*. De même pour `[m,k]=min(A)`.

On peut ajouter un paramètre après la matrice, sous la forme d'une caractère :

- `max(A, "c")` et `min(A, "c")` renvoient les *extrema* de chaque colonne de la matrice ;
- `max(A, "r")` et `min(A, "r")` renvoient les *extrema* de chaque ligne de la matrice (« r » pour *row*).

On peut mettre une suite de matrices de même taille dans `min` et max. Alors, Scilab renvoit une matrice *m* de taille identique dont chaque élément *m*(*i*, *j*) est l'*extremum* des éléments (*i*, *j*) des matrice, et éventuellement une matrice *k* dont chaque élément *k*(*i*, *j*) contient le numéro d'ordre de la matrice (dans la liste) qui contient ce maximum.

Exemple

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

    1.    2.
    3.    4.
 B  =

    4.    3.
    2.    1.

-->[m,k] = max(A,B)
 k  =

    2.    2.
    1.    1.
 m  =

    4.    3.
    3.    4.
```

La matrice *m*(1,1) contient « 4 » qui est le maximum entre A(1,1) et B(1,1), et *k*(1,1) contient « 2 » ce qui indique que c'est la seconde matrice (donc B) qui contient ce maximum.

La fonction `cumsum(A)` fait la somme cumulée des valeurs de la matrice A. Si A est un vecteur de dimension *n*, cela correspond à la somme de la série, la fonction renvoie un vecteur de dimension *n* contenant

$$\left(\mathrm{A}(1),\quad \mathrm{A}(1)+\mathrm{A}(2),\quad \ldots,\quad \textstyle\sum_{i=1}^{n}\mathrm{A}(i)\right).$$

Si A est une matrice rectangulaire *m*×*n*, la somme se fait par colonne, de haut en bas, le résultat est une matrice de même dimension contenant

$$\begin{pmatrix} \mathrm{A}(1,1) & \mathrm{S}_1+\mathrm{A}(1,2) & \cdots & \sum_{j=1}^{n-1}\mathrm{S}_j+\mathrm{A}(1,n) \\ \mathrm{A}(1,1)+\mathrm{A}(2,1) & \mathrm{S}_1+\mathrm{A}(1,2)+\mathrm{A}(2,2) & \cdots & \\ \vdots & \vdots & & \vdots \\ \mathrm{S}_1=\sum_{i=1}^{m}\mathrm{A}(i,1) & \mathrm{S}_1+\mathrm{S}_2 & \cdots & \sum_{j=1}^{n}\mathrm{S}_j \end{pmatrix}$$

La fonction `cumprod()` fait le produit cumulé, selon la même démarche.

La fonction `gsort()` effectue un tri :

```
gsort(A) # tri par ordre décroissant
gsort(A, 'i') # tri par ordre croissant
```

Pour plus de détails voir : ***Découvrir Scilab/Programmation#Tri, recherche et sélection***.

Si l'on effectue des comparaisons sur des matrices de même dimensions, cela donne une matrice de booléens de même dimensions, correspondant à la comparaison terme à terme. Par exemple

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

    1.    1.
    1.    1.
```

```
-->B = [1, 2 ; 3, 4]
 B  =

    1.    2.
    3.    4.

-->A == B
 ans  =

  T F
  F F

-->A < B
 ans  =

  F T
  T T
```

On peut ainsi utiliser les opérateurs ~=, <>, <, <=, > et >=.

Les fonctions `and()` et `or()` appliquent les opérateurs respectivement ET et OU à la totalité des éléments de la matrice. Par exemple, si l'on veut déterminer si deux matrices sont différentes, on peut utiliser `and(A ~= B)` ou `and(A <> B)`.

Importance des matrices dans Scilab

Les matrices sont très utilisées dans Scilab pour définir les paramètres. Ainsi, pour tracer une courbe, il utilise un nuage de point sous forme de matrice, ou encore, pour effectuer une boucle itérative, il prend les indices dans une matrice ligne ou colonne.

La syntaxe propre aux matrices se retrouve dans de nombreux endroits.

Date et heure

Pour certaines applications, il est nécessaire de manipuler les dates et heures. Scila fournit les commandes suivantes :

- `datenum()` ou `now` : indique le numéro du jour actuel, le premier jour étant le 1er janvier de l'an 0 s'il avait existé (en fait, l'an -1, puisque le calendrier grégorien passe de l'an -1 à l'an 1) ; la partie décimale correspond à l'heure ;
- `getdate()` : *idem*, mais avec un format différent : année, mois, numéro de la semaine dans le mois, numéro du jour d'ans l'année, numéro du jour dans la semaine (1 pour dimanche, 2 pour lundi, …), numéro du jour dans le mois, heure, minutes, secondes (entier), millisecondes ;
- `datenum([aaaa, mm, jj])` indique le numéro du jour *jj* du mois *mm* de l'année *aaaa* ; on peut aussi ajouter l'heure pour avoir la partie décimale : `datenum([aaaa, mm, jj, hh, mm, ss])` ;
- `clock` donne directement la date et l'heure au format [*aaaa*, *mm*, *jj*, *hh*, *mm*, *ss*] ;
- `datevec(numjour)` : convertit le numéro du jour en un vecteur contenant l'année, le mois, le jour, l'heure, les minutes et les secondes (opération inverse de la commande précédente) ;
- `weekday(numjour)` indique le jour de la semaine du jour *numjour* : 1 pour dimanche, 2 pour lundi, … Avec la syntaxe `[N, S] = weekday(numjour)`, Scilab indique le numéro du jour de la semaine, dans N, ainsi que le nom du jour selon l'abréviation anglaise, dans S ;
- `eomday(année, mois)` : donne le nombre de jours du mois (indiquer sous forme de nombre, de 1 à 12) et de l'année indiqué ;
- `calendar()` : donne le calendrier du mois en cours ;

- `calendar(aaaa, mm)` : donne le calendrier du mois *mm* de l'année *aaaa*.

Par exemple :

```
aujourdhui = datenum()
datevec(aujourdhui)
clock
calendar()
[N, S] = weekday(aujourdhui)
datenum([1969, 07, 21])
eomday([1969, 07])
```

Si l'on s'intéresse à l'intervalle entre deux dates :

- `etime(t1, t2)` : indique le nombre de secondes écoulées *(elapsed time)* entre les dates t_1 et t_2, sous la forme de vecteurs [*aaaa*, *mm*, *jj*, *hh*, *mm*, *ss*] ;
- `getdate("s")` : indique le nombre de secondes écoulée entre le 1er janvier 1970 et l'instant actuel ;
- `getdate(numsec)` : indique le jour et l'heure, au format `getdate()`, à partir du nombre de secondes écoulées depuis le 1er janvier 1970 ;
- `tic()` déclenche un chronomètre, et `toc()` renvoie le nombre de secondes écoulées depuis ce déclenchement.

Précision des calculs

Dans Scilab, les nombres sont stockés sour la forme de nombre décimaux de double précision *(doubles)*, selon la norme IEEE 754[5]. Ce sont des nombres à virgule flottante *(floating point)* codés sur 64 bits (8 octets). Il s'ensuit que :

- la précision d'un résultat est au mieux de 10^{-16} ; cette précision est indiquée par la variable `%eps` ;
- la valeur absolue la plus petite que l'on puisse représenter est environ 10^{-307} ; un nombre ayant une valeur absolue plus petite sera considéré comme nul, 0 (soupassement, *underflow*) ;
- la valeur absolue la plus grande que l'on puisse représenter est environ 10^{308} ; un nombre ayant une valeur absolue plus grande sera considéré comme infini, `inf` (dépassement, *overflow*).

Un certain nombre de fonction internes de Scilab utilisent des méthodes qui minimisent l'erreur de calcul, et font donc mieux qu'une codage « naïf » de ces méthode. Par exemple, pour résoudre une équation du second degré, il vaut mieux utiliser la commande `roots()` que calculer le résultat à partir du discriminant ; la différence ne sera pas visible pour la plupart des cas, mais peut être flagrante pour des cas extrêmes.

Lire les valeurs dans un fichier

Il existe plusieurs manières de lire des données écrites dans un fichier. Nous nous limitons ici à deux cas :

- fichier CSV, par exemple créé avec un tableur comme LibreOffice Calc ou Microsoft Office Excel : `M = csvRead("nom_du_fichier.csv");`
- fichier de texte pur (ASCII ou Unicode) dans lequel les données sont sous forme de tableau : `M = fscanfMat("nom_du_fichier.txt")`.

Dans les deux cas, les donées sont placées dans la matrice M.

Pour plus de détails voir : ***Découvrir Scilab/Gestion des fichiers***.

Voir aussi

Dans Wikipédia

- Algèbre booléenne
  - Opérateur logique
- American Standard Code for Information Interchange (ASCII)
- Concaténation
- Fraction rationnelle
- IEEE 754
- Matrice
  - Calcul matriciel
  - Déterminant
  - Matrice inversible
  - Matrice unité
  - Produit tensoriel
  - Trace
- NaN
- Polynôme
  - Fonction polynôme
- Relation d'ordre
- Vecteur
  - Produit scalaire
  - Vecteur colonne
  - Vecteur ligne

Notes

1. la documentation utilise le terme de « primitive » ; il y a une ambiguité en français que nous avons décidé de lever dans le présent ouvrage en changeant le terme (en anglais, le problème ne se pose pas puisque la primitive d'une fonction est appelée *« antiderivative »*)
2. dans la représentation des nombres à virgule flottante IEEE 754, un NaN est un nombre dont l'exposant ne contient que des 1 et la mantisse n'est pas nulle
3. En particulier, il faut éviter le test `a == []`, car la notion de « variable vide » peut évoluer. Par exemple, on peut envisager qu'il existera un jour un tableau de *n* lignes et 0 colonne, qui est équivalent à la variable vide d'un point de vue mathématiques, mais pas à `[]` d'un point de vue informatique. La commande `isempty()` est donc plus robuste aux évolutions du logiciel [1] (http://mailinglists.scilab.org/Scilab-users-gcf-tp4027097p4027114.html).
4. les commandes `code2str()` et `str2code()` permettent aussi de passer de nombres aux caractères et *vice versa* (il s'agit par contre d'une table de conversion propre à Scilab), mais sont déclarées obsolètes et seront supprimées dans la version 6
5. voir *Scilab is not naive (http://www.scilab.org/content/download/249/1710/file/scilabisnotnaive.pdf)*, Scilab Enterprises, décembre 2010

Interface < ↑ > *Calcul numérique*

Calcul numérique

Table des matières — Index

---

3. Calcul numérique

Vocabulaire

Dans le vocabulaire habituel de la programmation, une procédure est un sous-programme, c'est-à-dire un bout de programme auquel on donne un nom, et que l'on exécute à la demande en « appelant » ce nom. On peut voir cela comme une macro-définition (ou simplement « macro ») : le nom de la procédure est « remplacé » par le contenu du sous-programme. Une fonction est une procédure qui renvoit une valeur.

Dans la documentation officielle de Scilab, les termes *function*, *procedure* et *macro* sont équivalents ; par contre, ils sont utilisés pour les sous-programme définis par l'utilisateur. Les fonctions intégrées dans Scilab, dites *built-in*, sont appelées « primitives ».

Lorsqu'une primitive a pour argument une fonction, cette fonction est dite « externe » *(external)*. C'est le cas par exemple des primitives de résolution d'équations différentielle `ode()` ou d'optimisation `optim()` : on indique à ces primitves quelle est la fonction à utiliser.

Par soucis de simplicité, nous ne feront pas la différence entre les fonctions définies par l'utilisateur et les primitives, sauf lorsque cette nuance est pertinente. Les primitives sont parfois appelées « commandes » ou « instructions » dans le présent document.

Définition de fonctions extérieures

Une fonction extérieure est une fonction numérique définie par l'utilisateur (notez la différence entre « externe » et « extérieure »).

Pour les cas simples, la commande `deff` permet de définir de nouvelles fonctions pouvant s'exprimer avec les opérateurs déjà prédéfinis (les « fonctions internes » ou « primitives »). On passe deux chaînes de caractères comme paramètre :

- la première indique le nom de la fonction et les variables utilisées en entrée et en sortie, et
- la deuxième indique la formule.

Par exemple, la fonction

$$f(x) = 2 \cdot x$$

peut se définir par

```
deff("[y] = f(x)", "y = 2*x")
```

On peut aussi utiliser l'instruction `function()` sous la forme

```
-->function [y] = f(x)
```

```
--> …
--> endfunction
```

la formule, sous la forme `y = expression`, étant placée à la place des trois points. Avec l'exemple ci-dessus, cela donne :

```
-->function [y] = f(x)
-->y=2*x
-->endfunction
```

(voir aussi le chapitre *Programmation*).

La fonction s'utilise par la suite de la même manière qu'une fonction interne (sur l'exemple ci-dessus) :

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

    4.

-->feval(1,f) // calcule f(1)
 ans  =

    2.
```

Graphe d'une fonction

Il est souvent utile de pouvoir afficher des fonctions, ainsi voici un exemple de la syntaxe Scilab pour afficher la représentation graphique $y = f(x)$ d'une fonction f.

```
t=[1:0.1:30]; // t est un vecteur des valeurs de 1 à 30 par pas de 0,1

deff("[y] = f(x)", "y = cos(x)./x")  // définition de la fonction f

plot(t, f(t)) // tracé de la fonction
```

Pour plus de détails voir : ***Découvrir Scilab/Graphiques et sons#Tracé de fonction***.

Statistiques

Tendance centrale

Considérons une matrice X. On peut calculer les moyennes suivantes :

- `mean(X)` : calcule la moyenne de tous les éléments de X ;
- `mean(X, "r")` : donne un vecteur ligne *(row)* contenant les moyennes colonne par colonne ;
- `mean(X, "c")` : donne un vecteur colonne contenant les moyennes ligne par ligne.

On peut calculer la moyenne géométrique avec `geomean()`, et la moyenne harmonique avec `harmean()`, avec la même syntaxe.

De la même manière, on peut calculer les médianes avec `median(X)`, `median(X, "r")` et `median(X, "c")`.

La commande `trimmean(X, p)` permet de calculer la moyenne en éliminant les *p* pourcents extrêmes (les *p*/2 % valeurs les plus élevées, et les *p*/2 % valeurs les plus faibles). On peut également faire le calcul par ligne ou par colonne — `trimmean(X, p, "r")` ou `trimmean(X, p, "c")`. notons que l'on a :

- `mean(X) = trimmean(X, 0)` ;
- `median(X) = trimmean(X, 100)`.

Dispersion

La commande

```
strange(X)
```

calcule l'écart *(range)* entre la valeur maximale et la valeur minimale de la matrice. On peut calculer les valeurs pour chaque colonne ou chaque ligne avec `strange(X, "r")` et `strange(X, "c")`. La commande

```
mad(X)
```

calcule la déviation absolue moyenne *(mean absolute deviation)*

$$\frac{1}{n}\sum|x_i - \bar{x}|$$

on peut de même la calculer pour chaque colonne ou ligne.

La commande

```
iqr(X)
```

calcule l'écart interquartile *(interquartile range)*.

La commande `stdev` calcule l'écart type *(standard deviation)* et la commande `variance` la variance des éléments de la matrice (entière, par ligne ou par colonne). Par défaut, la variance est sans biais : normalisée par *m*×*n* - 1 pour une matrice de *m*×*n* éléments, par *m* - 1 par colonne et par *n* - 1 par ligne. Si l'on calcule la variance par ligne ou par colonne, on peut ajouter l'option `1` pour calculer le moment d'ordre 2 (normalisation par *m* ou *n*), par exemple

```
mom2 = variance(X, "r", 1) // pour une matrice non-vecteur
mom2 = variance(X, "*", 1) // pour un vecteur
```

L'écart type est toujours sans biais. Si l'on veut l'estimateur normalisé par *m* ou *n*, il faut utiliser

```
estim_ecart_type = sqrt(variance(X, "*", 1)) // pour une matrice non-vecteur
```

Normalisation

La commande `center(X)` permet de centrer la matrice (équivalent de `X - mean(X)`). La commande `wcenter(X)` donne la matrice centrée réduite (équivalent de `1/stdev(X)*(X - mean(X))`).

Fréquences et quartiles

La commande `tabul(X)` calcule les fréquences d'apparition des valeurs dans la matrice X. Le résultat est classé par défaut par ordre décroissant des valeurs ; pour avoir un classement croissant, on ajoute le paramètre <code"'i" *(increasing order)* :

```
tabul(X, "i")
```

La commande `quart(X)` calcule les quartiles et les renvoie dans un vecteur de dimension 3. On peut calculer les quartiles pour chaque colonne ou chaque ligne, avec `quart(X, "r")` et `quart(X, "c")`.

On peut aussi travailler avec des valeurs et leur fréquence. Si la matrice X contient les valeurs et la matrice F leur fréquence — F(*i*, *j*) est la fréquence de X(*i*, *j*) —, alors

- `meanf(X, F)` donne la moyenne ;
- `stdevf(X, F)` donne l'écart type ;
- `variancef(X, F)` donne la variance.

Notons que F n'est pas nécessairement normalisé à 1 ou à 100 ; il peut par exemple s'agir d'un nombre d'occurrences.

Exemple

```
X = int(10*rand(100,1, "normal")); // génère 100 valeurs entières avec une loi normale ce
F = tabul(X); // calcule les fréquences d'apparition
x = F(:, 1); // extraction de la liste des valeurs
f = F(:, 2); // extraction de la liste des fréquences

quartiles = quart(X); // calcule les quartiles de la population

moyenne = meanf(x, f); // calcul de la moyenne pondérée par les fréquences
```

Considérons maintenant deux vecteurs X et Y de même dimension, et la matrice F des fréquences de X et Y : F(*i*, *j*) est la fréquence de X(*i*) & Y(*j*). Alors :

- `covar(X, Y, F)` donne la covariance ;
- `correl(X, Y, F)` donne le coefficient de corrélation.

Exemple

```
n = 10; // nombre de valeurs
X = 1:n; // abscisses
Y = 2*X + 5 + 3*rand(X); // ordonnées : droite "bruitée"
plot(X, Y); // tracé des données
F = eye(n, n); // couplage (x, y) : chaque couple est présent une fois, (X(i) & Y(i)) = 1

covariance1 = covar(X, Y, F);
covariance2 = 1/n*sum(X.*Y) - 1/n^2*sum(X)*sum(Y); // calcul "à la main" pour comparer

correlation1 = correl(X, Y, F);
correlation2 = covariance2/sqrt(variance(X, "*", 1))/sqrt(variance(Y, "*", 1)); // calcul
```

Fonctions de distribution et de répartition de lois statistiques

Par défaut, Scilab propose une loi de distribution, pour la loi binomiale, et onze lois de répartition (probabilités cumulatives).

Pour la loi binomiale, la fonction de répartition pour *n* épreuves (nombre entier) avec une probabilité *p* (nombre réel entre 0 et 1) est donnée par

```
f = binomial(p, n)
```

La variable `f` est un vecteur de *n* + 1 composantes, avec

$$\mathtt{f}(k+1) = \binom{n}{k} \cdot p^k \cdot (1-p)^{n-k} = \mathrm{C}_n^k \cdot p^k \cdot (1-p)^{n-k}$$

La fonction de répartition s'obtient avec `cdfbin()` :

```
P =  = cdfbin("PQ", k, n, p, 1-p)
```

donne la probabilité P d'avoir entre 0 et *k* succès pour *n* tirages avec une probabilité de réussite *p*. On peut d'ailleurs le comparer avec

```
f = binomial(p, n)
c = sum(f(1:k+1))
```

De manière générale, les fonctions de répartition commencent par `cdf…` *(cumulative distribution function)*, par exemple :

- `P = cdfchi("PQ", x, k)` : loi du χ^2 à *k* degrés de liberté ;
- `P = cdfnor("PQ", x, mu, sigma)` : loi normale d'espérance *mu* et d'écart type *sigma* ;
- `P = cdfpoi("PQ", k, lambda)` : loi de Poisson de paramètre *lambda* ;
- `P = cdft("PQ", t, k)` : loi t de Student à *k* degrés de liberté ;
- … voir *Scilab help >> Statistics > Cumulated Distribution Functions (http://help.scilab.org/docs/5.4.1 /en_US/section_e9886297dc5f06a7c11cf8b143ab8686.html)*

Notons que pour chaque cas, on peut aussi avoir la valeur complémentaire Q = 1 - P avec la syntaxe

```
[P, Q] = cdf…("PQ", …)
```

Exemple

Ci-dessous, nous traçons la fonction de répartition de la loi normale centrée réduite (μ = 0, σ = 1). Notons que pour des calculs vectoriels, l'espérance et l'écart type sont des vecteurs de même dimension que X.

```
X = -3:0.1:3;
mu = zeros(X);
sigma = ones(X);
[P, Q] = cdfnor("PQ", X, mu, sigma);
plot(X', [P', Q'])
```

Les mêmes fonctions permettent de rechercher les quantiles ou fractiles :

- `k = cdfbin("S", n, P, 1-P, p, 1-p)` : nombre de succès *k* sur *n* épreuves ayant pour paramètre *p*, donnant une probabilité cumulée P ;
- `x = cdfchi("X", k, P, 1-P)` : valeur de *x* pour laquelle on a une probabilité cumulée P avec une loi du χ^2 à *k* degrés de liberté ;
- `x = cdfnor("X", mu, sigma, P, 1-P)` : valeur de *x* pour laquelle on a une probabilité cumulée P avec une loi normale d'espérance *mu* et d'écart type *sigma* ;
- `k = cdfpoi("S", lambda, P, 1-P)` : valeur de *k* pour laquelle on a une probabilité cumulée P avec une loi de Poisson de paramètre *lambda* ;
- `t = cdft("T", k, P, 1-P)` : valeur de *t* pour laquelle on a une probabilité cumulée P avec une loi t de Student à *k* degrés de liberté ;
- …

Exemple

Ci dessous, nous déterminons le quantile pour un risque α bilatéral de 5 %, c'est-à-dire que l'on rejette 0,25 % des cas les plus hauts et les plus bas, pour une loi normale centrée réduite puis pour une loi t de Student à 30 degrés de liberté

```
-->x = cdfnor("X", 0, 1, 0.975, 0.025)
 x  =

    1.959964

-->t = cdft("T", 30, 0.975, 0.025)
 t  =

    2.0422725
```

Enfin, ces fonctions permettent de déterminer les paramètres des loi pour atteindre un quantile visé :

- loi binomiale :
  - `n = cdfbin("Xn", p, 1-p, P, 1-P, k)` : nombre d'épreuves ayant pour paramètre *p* afin d'avoir un quantile P valant *k*,
  - `[p, q] = cdfbin("PrOmpr", P, 1-P, k, n)` : paramètre *p* de chaque épreuve afin d'avoir un quantile P pour *k* succès parmi *n* ;
- loi du χ^2 :
  `k = cdfchi("Df", P, 1-P, x)` : valeur du nombre de degrés de liberté *(degrees of freedom) k* pour avoir un quantile P valant *x* ;
- loi normale :
  - `mu = cdfnor("Mean", sigma, P, 1-P, x)` : valeur de l'espérance (moyenne) pour avoir un quantile P valant *x* si l'on a un écart type *sigma*,
  - `sigma = cdfnor("Std", P, 1-P, X, mu)` : valeur de l'écart type *(standard deviation)* pour avoir un quantile P valant *x* si l'on a une espérance de *mu* ;
- loi de Poisson :
  `lambda = cdfpoi("Xlam", P, 1-P, k)` : valeur du paramètre λ pour avoir un quantile P valant *k* ;
- loi t de Student :
  `k = cdft("Df", P, 1-P, t)` : nombre de degrés de liberté *k* pour avoir un quantile P valant *t*.

Générateurs pseudo-aléatoires

Les générateurs de nombres aléatoires permettent de tester des programmes ou bien de faire des simulations

par la méthode de Monte Carlo.

Scilab possède une fonction `rand()` qui par défaut génère un nombre entre 0 et 1 avec une loi de distribution uniforme. Par exemple, pour simuler un jet de dé à six faces, on peut utiliser :

```
de = 1 + int(6*rand())
```

La fonction peut générer une matrice *m*×*n* avec les syntaxes

```
rand(m, n)
rand(X)
```

où X est une matrice *m*×*n*. Notons que si l'on veut générer *n* valeurs, il faut bien taper `rand(n, 1)` ou `rand(1, n)`.

On peut lui faire générer une valeur selon une loi normale centrée réduite avec la clef `"normal"` :

```
rand(m, n, "normal")
```

Exemple

Supposons par exemple que l'on a une poutre faite dans un acier dont la résistance intrinsèque, la limite d'élasticité R_e, est garantie au minimum à 235 MPa. En réalité, R_e suit une loi normale d'espérance μ_{Re} = 269 MPa et d'écart type σ_{Re} = 17 MPa. Dans les conditions d'utilisation, l'acier de cette poutre doit résister à une contrainte S *(stress)* de 214 MPa ; en fait, les conditions de chargement sont variables et suivent une loi normale de paramètres μ_S = 214 MPa et σ_S = 15 MPa.

On veut déterminer statistiquement dans combien de cas la contrainte S est supérieure à la résistance R_e.

```
n = 1e6; // nombre d'événements
R = 269 + 17*rand(n, 1, "normal"); // résistance
S = 214 + 14*rand(n, 1, "normal"); // contrainte
Mbool = S>R; // comparaison des valeurs (matrice de booléens)
moyenne = sum(Mbool)/n; // les "vrai" comptent pour 1 et les "faux" pour 0
disp(string(100*moyenne)+" %") // affichage du résultat
```

La commande `grand()` *(get random number)* permet d'avoir des nombres aléatoires selon de nombreuses lois, par exemple :

- `Y = grand(m, n, "bin", N, p)` : matrice *m*×*n* de nombres aléatoires tirés selon une loi binomiale de N essais avec une probabilité *p* chacun ;
- `Y = grand(m, n, "nor", mu, sigma)` : pour une loi normale d'espérance *mu* et d'écart type *sigma* ;
- `Y = grand(m, n, "unf", a, b)` : pour une loi uniforme sur [*a' ;* b*[(*b *exclus) ;*
- `Y = grand(m, n, "uin", a, b)` : pour une loi uniforme sur [*a' ;* b*] (*b *inclus) ;*
- …

Dans l'exemple précédent, on aurait donc :

```
R = grand(n, 1, "nor", 369, 17); // résistance
S = grand(n, 1, "nor", 314, 14); // contrainte
```

Autres lois statistiques

Des modules complémentaires Atoms permettent d'avoir accès à d'autres loi.

Par exemple le module CASCI (http://atoms.scilab.org/toolboxes/casci) donne accès à de nombreuses fonctions de distribution et de répartition, notamment des lois log-normale (`pdflognormal()`, `cdflognormal()`), exponentielle (`pdfexponential()`, `cdfexponential()`) ou de Weibull (`pdfweibull()`, `cdfweibull()`). Ainsi :

```
p = pdfnormal(x, mu, sigma); // densité d'une loi normale en x
y = cdflognormal(x, lambda); // répartition d'une loi log-normale d'espérance mu et d'éca
y = cdfexponential(x, mu, sigma); // répartition d'une loi exponentielle de paramètre lam
y = cdfweibull(x, k, lambda, theta); // répartition d'une loi de Weibull
   // de paramètre de forme k, de paramètre d'échelle lambda et de paramètre de position
```

Le module fournit également le calcul des quantiles avec les fonctions `idf…` *(inverse cumulative distribution functions)*, et des générateurs de nombres aléatoires avec les fonctions `rnd…`. Par exemple :

```
x = idfweibull(y, k, lambda, theta); // quantile y d'une loi de Weibull
   // de paramètre de forme k, de paramètre d'échelle lambda et de paramètre de position
X = rndweibull(n, k, lambda, theta); // génère n nombres aléatoires selon la loi de Weibu
```

Interpolation, extrapolation et lissage

Interpolation dans le plan

Présentation du problème

On dispose de points expérimentaux (x_i, y_i), qui sont censés suivre une loi $y = f(x)$ inconnue, et l'on désire avoir une valeur y correspondant à une valeur x arbitraire, mais contenue dans l'intervalle $[\min\{x_i\}, \max\{x_i\}]$. Une manière de faire consiste à utiliser une interpolation ; Scilab propose deux démarches :

- interpolation linéaire : si $x_i \leq x \leq x_{i+1}$, alors on prend y sur le segment de droite $[(x_i, y_i) ; ((x_{i+1}, y_{i+1})]$: $y = y_i + \frac{y_{i+1} - y_i}{x_{i+1} - x_i}(x - x_i)$;
- interpolation par un polynôme de degré trois, en utilisant des *splines* cubiques d'Hermite :
  $y = p(x)$
  la fonction p étant un polynôme de degré trois par parties, les portions de polynôme étant raccordées aux points expérimentaux (même valeur, même pente).

> *Pour plus de détails voir :* ***wikipedia:fr:Interpolation linéaire*** *et* ***wikipedia:fr:Spline cubique d'Hermite***.

Interpolation linéaire

Le nuage de n points $(x_i, y_i)_{1 \leq i \leq n}$ est décrit par deux vecteurs : `x`, qui contient les x_i classés par ordre croissant, et `y`, qui contient les y_i correspondant.

Pour un interpolation linéaire, on considère une valeur `xp` arbitraire, ou un vecteur de valeurs, les valeurs de `yp` correspondantes sont obtenues par

```
yp = interp1(x, y, xp)
```

Plutôt que d'avoir les valeurs interpolées, on peut préférer les valeurs les plus proches ; pour cela, on utilise l'option `"nearest"` :

```
yp = interp1(x, y, xp, "nearest")
```

On peut aussi utiliser la commande d'interpolation à *n* dimensions `linear_interpn()` :

```
yp = linear_interpn(xp, x, y)
```

Si l'on veut faire une *extrapolation*, c'est-à-dire avoir une valeur hors de l'intervalle $[x_1, x_n]$, on peut ajouter une option :

- `"by_zero"` : $y = 0$ hors intervalle ;
- `"C0"` : $y = y_1$ si $x < x_1$, $y = y_n$ si $x > x_n$;
- `"natural"` : si la fonction est supposée monotone hors de l'intervalle de définition, on utilise le segment de droite le plus proche ;
- `"periodic"` : si la fonction est supposée périodique, on répète le motif obtenu.

Par exemple :

```
yp = linear_interpn(xp, x, y, "natural")
```

Interpolation par une *spline* cubique

Pour une interpolation par *spline* cubique d'Hermite, la fonction d'interpolation est décrite par les coordonnées des points et un vecteur de coefficients `d` (les dérivées), soit trois vecteurs (`x`, `y`, `d`). Le vecteur `d` est obtenu par la commande `plsin()` :

```
d = splin(x, y)
```

si l'on a maintenant une valeur `xp` arbitraire, ou un vecteur de valeurs, les valeurs de `yp` correspondantes sont obtenues par

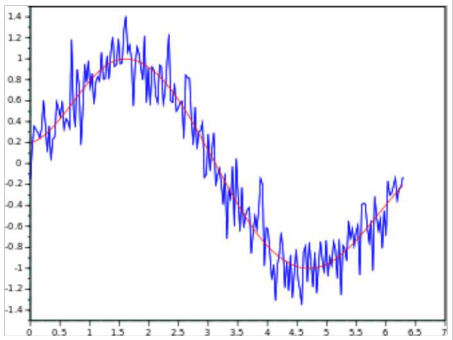
```
yp = interp(xp, x, y, d)
```

Lissage

L'interpolation considère que les données sont peu bruitées ; les points mesurés sont « fiables », les valeurs des `x` et `y` sont « exactes ». Si le signal est bruité, il faut alors effectuer un lissage.

Scilab propose la commande `lsq_splin()` qui utilise des *splines* cubique comme modèle, déterminées par

la méthode des moindres carrés *(least square)*. Si l'on a deux vecteurs *x* et *y* de *m* valeurs chacun, on choisit *n* abscisses *xs* pour déterminer la *spline*, avec dans l'idéal *n* << *m*. Typiquement :



Lissage par une *spline* cubique.

```
// ***** Génération des données *****

m = 200; // nombre de points mesurés
bruit = 0.2; // amplitude du bruit
x = linspace(0, 2*%pi, m);
y = sin(x) + 0.2*rand(x, "normal"); // valeurs bruitées

// ***** Détermination de la spline *****

n = 6; // nombre de points de contrôle de la spline
xs = linspace(x(1), x($), n); // abscisse des points de c
// répartition uniforme
[ys, d] = lsq_splin(x, y, xs); // détermination de la spl

// ***** Calcul des valeurs lissées de y *****

yliss = interp(x, xs, ys, d); // calcul des données lissé

// ***** Tracé *****

plot(x, y) // données originales
plot(x, yliss, "r") // données lissées
```

Ainsi, Scilab a construit une *spline* cubique passant par les abscisses *xs* (en l'occurrence 0, 2π/5, 4π/5, 6π/5, 8π/5 et 2π), et a déterminé les dérivées *d* en ces *n* points.

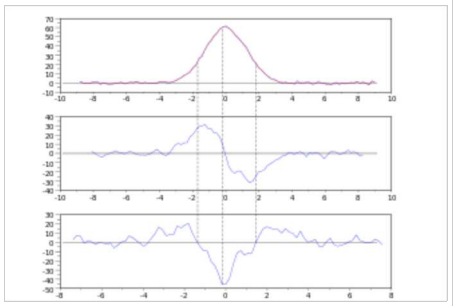
Si l'on veut éviter des effets de bord, on peut choisir une répartition différente des points de contrôle. On utilise fréquemment une répartition sinusoïdale pour avoir plus de points aux bords qu'au centre (comme pour les polynômes de Tchebychev).

```
theta = linspace(-%pi, 0, n); // angles uniformément répartis
xs = mean([x(1), x($)]) + 0.5*(x($) - x(1))*cos(theta);
// abscisse des points de contrôle
```

Ceci est en particulier important si l'on veut extrapoler, c'est-à-dire calculer des valeurs de *y* pour des *x* hors des points d emesure.

Cela ne présente d'intérêt que pour $n \geq 4$.

L'utilisateur peut évidemment coder un autre algorithme. L'image ci-contre montre le calcul de la dérivée et de la dérivée seconde par l'algorithme de Savitzky-Golay ; cliquer sur l'image pour avoir accès à la page de description indiquant le code utilisé.



Lissage et détermination des dérivée et dérivée seconde par l'algorithme de Savitzky-Golay.

On peut aussi effectuer un simple lissage par la méthode des moyennes glissantes en utilisant la fonction `convol()` (voir ci-après). Si l'on appelle *p* le nombre de points de l'intervalle glissant :

```
p = 10; // largeur de l'intervalle

h = 1/p*ones(1, p); // fonction porte pour le lissage
```

```
yliss = conv(h, y);

plot(x, y) // données originales
plot(x, yliss(1+p/2:$-p/2+1), "r") // données lissées
```

Interpolation d'une surface

Présentation du problème

On dispose de valeurs z_i à des positions déterminées (x_i, y_i), qui sont censés suivre une loi $z = f(x, y)$ inconnue, ce qui définit donc une surface dans l'espace, et l'on désire avoir une valeur z correspondant à un point (x, y) arbitraire.

Interpolation linéaire sur un quadrillage

Les données sont décrites par :

- un vecteur `x` de dimension m et un vecteur `y` de dimension n, strictement croissants, qui définissent un quadrillage du plan, donc un ensemble de $m \times n$ points $\{(x_1, y_1) ; (x_1, y_2) ; \dots ; (x_1, y_n) ; (x_2, y_1) ; \dots ; (x_m, y_n)$;
- une matrice `z` de dimension $m \times n$, contenant les valeurs correspondante : $z_{i,j} = f(x_i, y_j)$.

L'interpolation linéaire se fait de la même manière que précédemment : on définit un nouveau couple de valeurs `xp` et `yp` (ou un couple de vecteurs défnissant un nouveau quadrillage), et l'on a :

```
zp = linear_interp(xp, yp, x, y, z)
```

c'est-à-dire qu'à partir des points (x, y, z), on définit les points interpolés (xp, yp, zp).

Interpolation cubique sur un quadrillage

Comme précédemment, les données sont décrites par :

- un vecteur `x` de dimension m et un vecteur `y` de dimension n, strictement croissants, qui définissent un quadrillage du plan, donc un ensemble de $m \times n$ points $\{(x_1, y_1) ; (x_1, y_2) ; \dots ; (x_1, y_n) ; (x_2, y_1) ; \dots ; (x_m, y_n)$;
- une matrice `z` de dimension $m \times n$, contenant les valeurs correspondante : $z_{i,j} = f(x_i, y_j)$.

Et comme dans le plan, la fonction d'interpolation est décrite par (`x`, `y`, `C`), les valeurs coordonnées des points du quadrillage et une matrice de coefficients `C` obtenue par

```
C = splin2d(x, y, z)
```

Si l'on considère un point du plan (`xp`, `yp`) (ou un couple de vecteurs défnissant un nouveau quadrillage), les valeurs de z sur ces nouveaux points sont :

```
zp = interp2d(xp, yp, x, y, C)
```

Interpolation cubique sur un nuage de points dispersés

On ne dispose pas toujours de relevés de valeurs sur des coordonnées régulières formant un quadrillage. Dans ce cas-là, on utilise la méthode de Shepard (pondération des plus proches voisins). Les *n* points expérimentaux (x_i, y_i, z_i) sont décrits par une matrice M de trois colonnes et de *n* lignes

$$\mathrm{M} = \begin{pmatrix} x_1 & y_1 & z_1 \\ \vdots & \vdots & \vdots \\ x_n & y_n & z_n \end{pmatrix}$$

La fonction d'interpolation est définie par une liste typée T obtenue par

```
T = cshep2d(M)
```

Si l'on considère un point du plan (`xp`, `yp`) (ou un couple de vecteurs défnissant un nouveau quadrillage), les valeurs de *z* sur ces nouveaux points sont :

```
zp = eval_cshep2d(xp, yp, T)
```

Régression

Présentation du problème

Le problème a des similarités avec celui de l'interpolation : on a des points expérimentaux (x_i, y_i), et l'on détermine une fonction décrivant ces points. Les différences majeures sont :

- la fonction est déterminée sur l'ensemble de définition complet, à partir de tous les points, et non pas par morceaux entre deux points ;
- la fonctions ne passe en général pas *par* les points expérimentaux mais *près* de ces points.

Par ailleurs, la forme de la fonction est souvent déterminée par le type de problème auquel on s'attaque, elle a un « sens physique », là où l'interpolation se contente de prendre des fonctions simples.

Considérons un nuage de *n* points « expérimentaux » (X, Y) : X et Y sont des vecteurs

- $\mathrm{X} = [\mathrm{X}_1 ; \mathrm{X}_2 ; \ldots ; \mathrm{X}_n]$;
- $\mathrm{Y} = [\mathrm{Y}_1 ; \mathrm{Y}_2 ; \ldots ; \mathrm{Y}_n]$.

Considérons une fonction « modèle » $y = f(\mathrm{A}, x)$ où A est un vecteur ; les valeurs de A sont des paramètres de la fonction f, par exemple :

- $f(\mathrm{A}, x) = \mathrm{A}(1) \cdot \exp((x - \mathrm{A}(2))^2/\mathrm{A}(3))$: une fonction gaussienne ;
- $f(\mathrm{A}, x) = \mathrm{A}(1) \cdot x^3 + \mathrm{A}(2) \cdot x^2 + \mathrm{A}(3) \cdot x + \mathrm{A}(4)$: un polynôme de degré 3.

La régression consiste à touver le vecteur A tel que la fonction f « colle le mieux » au nuage de points. On utilise pour cela la méthode des moindres carrés *(least square)* :

- on définit la fonction « résidus » $r(\mathrm{A}, x, y)$ entre le modèle et les points expérimentaux :
  $r(\mathrm{A}, x, y) = y - f(\mathrm{A}, x)$;

- on définit la somme S des carrés des résidus comme étant :

$$S = \sum_i r^2(A, X_{i,\ Y_i})\ ;$$

la régression par la méthode des moindres carrés consiste donc à trouver le vecteur A donnant la valeur minimale de S.

Régression linéaire

Régression linéaire simple

Dans le cas de la régression linéaire, la fonction modèle f est une fonction affine :

$$f(A, x) = A(1)\cdot x + A(2).$$

Comme il n'y a que deux paramètres dans le modèle, on note ceux-ci a et b :

$$f(a, b, x) = a\cdot x + b.$$

D'une certaine manière, la régression consiste à résoudre un système d'équations surdéterminé (chaque point expérimental étant une équation, il y a plus d'équations que d'inconnues) :

$$\begin{cases} ax_1 + b = y_1 \\ ax_2 + b = y_2 \\ \dots \\ ax_n + b = y_n \end{cases}$$

ce qui peut s'écrire sous forme matricielle

$$A \times X = Y$$

avec

$$A = (a\ b)$$
$$X = \begin{pmatrix} x_1 & x_2 & \dots & x_n \\ 1 & 1 & \dots & 1 \end{pmatrix}$$
$$Y = (y_1\ y_2\ \dots\ y_n)$$

Si l'on demande à Scilab de résoudre le système avec la division matricielle alors que la matrice X n'est pas carrée, il effectue automatiquement une régression linéaire des moindres carrés :

```
\\ génération des données de test
x = 1:10
y = x + 0.3*rand(x);

\\ régression
X = [x ; ones(x)];
A = y/X
```

Si l'on veut forcer l'ordonnée à l'origine à 0, il suffit de ne pas mettre la ligne de 1 :

```
A = y/x
```

On peut aussi travailler avec des vecteurs colonne, et utiliser l'autre diviseur matriciel `A = x\y`.

Mais la méthode de la division matricielle ne donne pas d'autre information que les coefficients, et en particulier ne donne pas le résidus.

Pour cela, on peut utiliser la fonction `reglin()`, avec la syntaxe :

```
// définition du nuage de points
X = […];
Y = […];

// régression
[a, b, sigma] = reglin(X, Y);
```

Remarque : Les matrices X et Y doivent être des vecteurs *ligne*.

On peut aussi utiliser la commande `regress()`. Elle ne permet pas d'obtenir l'écart type, et ne fonctionne qu'avec deux variables, mais par contre, les vecteurs peuvent être indifféremment ligne ou colonne ; ils peuvent même être de types différents (un vecteur ligne et l'autre colonne).

```
coefs = regress(X, Y)
```

Exemple

Dans l'exemple suivant, on génère des point dispersés aléatoirement autour d'une droite `y = pente*x + ordonnee`, puis on fait la régression linéaire de ce nuage de points.

Pour clarifier le code source, le programme est scindé en trois fichiers situés dans le même répertoire (dossier) appelé ici `monrepertoire/` :

- le fichier `fonction.sce` qui contient la fonction affine ;
- le fichier `generation.sce` qui crée un nuage de points et le met dans le fichier `donnees.txt` ;
- le fichier `regression.sce` qui lit le fichier `donnees.txt` et effectue la régression.

Fichier `fonction.sce`

```
chdir("monrepertoire/");

// **********
// Constantes
// **********

// **********
// Fonctions
// **********

deff("[y] = affine(a, b, x)", "y = a*x + b")
```

Fichier `generation.sce`

```
chdir("monrepertoire/");

// **********
```

```
// Constantes
// **********

// paramètres de la loi affine
pente = 1;
ordonnee = 10;

// paramètres de la loi normale
var = 1; // variance

// nombre de points
nbpts = 20;

// **********
// Initialisation
// **********

X = zeros(nbpts)';
Y = zeros(nbpts)';

// **********
// Fonctions
// **********

exec("fonction.sce", -1) // charge le fichier (-1 : sans commentaire)

// **********
// Programme principal
// **********

// nuage de points
for i = 1:nbpts
    X(1,i) = i + var*rand(1, "normal");
    Y(1,i) = affine(pente, ordonnee, i) + var*rand(1, "normal");
end

// enregistrement

write("donnees.txt", [X,Y]);
```

Fichier `regression.sce`

```
clf;
chdir("monrepertoire/");

// **********
// Programme principal
// **********

// lecture des données
donnees = read("donnees.txt", -1, 2)
Xexp = donnees(:,1);
Yexp = donnees(:,2);


// regression
[aopt, bopt, sigma] = reglin(Xexp, Yexp)

// loi modèle
Yopt = affine(aopt, bopt, X);

// affichage
plot(Xexp, Yexp, "+b")
```

```
plot(Xexp, Yopt, "-r")
xstring(0, ordonnee, "a = "+string(aopt)+" pour "+string(pente)...
+" ; b = "+string(bopt)+" pour "+string(ordonnee)...
+" ; sigma = "+string(sigma))
```

Comme la forme de la fonction f est connue, il est inutile de la définir. La fonction `reglin()` retourne un vecteur de trois composantes : les paramètres a et b du modèle, et l'écart type σ du résidu (qui est la racine carrée de la somme des carrés des résidus, $\sigma = \sqrt{S}$).

Régression multilinéaire

Notons que a, b, x et y peuvent être des vecteurs. Si par exemple on a un modèle linéaire à plusieurs variables :

$$f(a_1, a_2, \ldots, a_m, b, x_1, x_2, \ldots x_n) = a_1 \cdot x_1 + a_2 \cdot x_2 + \ldots + a_n \cdot x_n + b$$

alors il suffit de définir X comme étant une matrice $m \times n$

```
X = [X11, X12, … X1n ;
     X21, X22, … X2n ;
     …
     Xm1, Xm2, … Xmn]
```

où X_{ij} est la valeur de x_j pour le point de mesure i ; et de définir Y comme étant un vecteur ligne de n valeurs

```
Y = [Y1, Y2, … , Yn]
```

et le paramètre a retourné par la division matricielle ou par la fonction `reglin()` est le vecteur $[a_1, a_2, \ldots, a_n]$.

Régression polynomiale

La régression polynomiale est un cas particulier de régression multilinéaire.

Pour plus de détails voir : ***wikipedia:fr:Régression polynomiale***.

Voici par exemple un programme générant n points pour x allant de 0 à 5 (valeurs exactes), et y suivant le polynôme $x^3 - 2 \cdot x^2 + 3 \cdot x - 4$, bruité :

```
// n points aléatoires approchant un  polynôme de degré 3

// chdir("mon_chemin_d_acces"); // répertoire où enregistrer le fichier

n = 10;
x = linspace(0, 5, n);
P = %s^3 - 2*%s^2 - 3*%s + 4;

y = horner(P, x); // valeurs exactes du polynôme
```

```
clf;
plot(x, y); // tracé du polynôme

y = y + 3*rand(y, "normal"); // valeurs bruitées

plot(x, y, "+"); // tracé des points bruités

write("polynome_bruite.txt", [x' , y']);
```

Et voici un programme permettant d'effectuer la régression sur ces données :

```
// Régression polynomiale de degré 3

// chdir("mon_chemin_d_acces"); // répertoire où chercher le fichier


M = read("polynome_bruite.txt", -1, 2); // lecture des données

x = M(:, 1);
y = M(:, 2);

X = [x.^3, x.^2, x, ones(x)];

A = X\y; // régression linéaire par les moindres carrés

P = [%s^3, %s^2, %s, 1]*A; // définition du polynôme

yth = horner(P, x); // valeurs correspondant au polynôme

// tracé du résultat
clf;
plot(x, yth)
plot(x, y, "+")
```

Régression non-linéaire

Dans le cas de la régression non-linéaire, f est une fonction quelconque. Pour effectuer la régression, on utilise la fonction `leastsq()` avec la syntaxe suivante :

```
// fonction modèle
function y = f(A, x)
   ...
endfunction

// fonction résidus
function e = r(A, x, y)
   e = f(A, x) - y
endfunction

// initialisation des paramètres du modèle
Ainit = […];

// définition du nuage de points
X = […];
Y = […];

[S, Aopt] = leastsq(list(r, X, Y), Ainit)
```

Avec cette syntaxe, la fonction `leastsq()` retourne donc un vecteur :

- le premier élément, `S`, est la valeur de la somme des carrés des écarts ;
- le second élément, `Aopt`, est le vecteur contenant la valeur « optimisée » des paramètres du modèle.

> **Remarque :** Les matrices X et Y doivent être ici des vecteurs *colonne*. Par ailleurs, la fonction de calcul du résidus doit s'appliquer aux *vecteurs* X et Y, pour renvoyer un *vecteur* e.

Exemple 1

Nous reprenons l'exemple de la régression linéaire, mais en utilisant cette fois-ci la fonction `leastsq()`. Lorsque l'on enregistre les données générées, on utilise :

```
write("donnees.txt", [X', Y']);
```

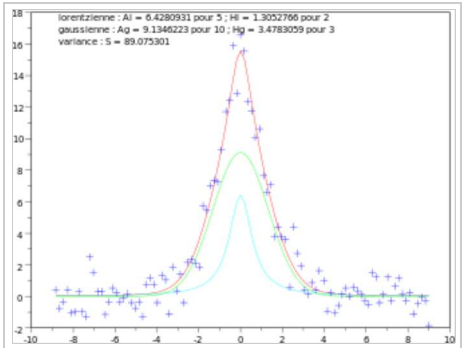
afin d'avoir des vecteurs colonne.

```
clf;
chdir("monrepertoire/");

// **********
// Initialisation
// **********

Ainit=[1;1];

// **********
// Fonctions
// **********

deff("[y] = f(a, b, x)", "y = a*x + b")

deff("[e] = res(A, x, y)", "e = f(A(1),A(2),x) - y")

// **********
// Programme principal
// **********

// lecture des données
donnees = read("donnees.txt", -1, 2)
Xexp = donnees(:, 1);
Yexp = donnees(:, 2);

// regression
[S, Aopt] = leastsq(list(res, Xexp, Yexp),Ainit)

// loi modèle
Ymod = f(Aopt(1), Aopt(2), Xexp);

// affichage
plot(Xexp, Ymod, "-r")
plot(Xexp, Yexp, "+b")
xstring(0, ordonnee, "a = "+string(Aopt(1))+" pour "+string(pente)...
+" ; b = "+string(Aopt(2))+" pour "+string(ordonnee)...
+" ; variance = "+string(S))
```

Exemple 2

Nous travaillons ici avec un pic généré par la somme d'une fonction gaussienne et d'une fonction lorentzienne. Ce genre de profil est classiquement utilisé pour décrire la forme des pics obtenus par diffractométrie X. On fait ce que l'on appelle une désommation de pics (on parle souvent de

« déconvolution », mais ce terme est abusif puisqu'il ne s'agit pas de produit de convolution).



Résultat de la régression.

```
clf

// **********
// Constantes
// **********

paramlorentz(1) = 5; // hauteur de la courbe lorentzienne
paramlorentz(2) = 2; // largeur de la courbe lorentzienne
paramgauss(1) = 10; // hauteur de la courbe gaussienne
paramgauss(2) = 3; // largeur de la courbe gaussienne
var=0.02; // variance de la loi normale
nbpts = 100 // nombre de points
demielargeur = max(3*paramgauss(2), 3*paramlorentz(2)) //
pas = 2*demielargeur/nbpts;
Ainit = [1;1;1;1]; // paramètres initiaux du modèle

// **********
// Initialisation
// **********

X = zeros(nbpts,1);
Y = zeros(nbpts,1);
Yopt = zeros(nbpts,1);

// **********
// Fonctions
// **********

// Fonction gaussienne centrée sur 0

function [y] = gauss(A, x)
    // A(1) : hauteur de pic
    // A(2) : "largeur" du pic
    y = A(1)*exp(-x^2/A(2));
endfunction

// Fonction lorentzienne centrée sur 0

function [y] = lorentz(A, x)
    // A(1) : hauteur de pic
    // A(2) : "largeur" du pic
    y = A(1)./(1 + (2.*x/A(2))^2);
endfunction

function [y] = profil(A, x)
    // A(1) : hauteur de pic lorentzien
    // A(2) : "largeur" du pic lorentzien
    // A(3) : hauteur de pic gaussien
    // A(4) : "largeur" du pic gaussien
    L = A(1:2);
    G = A(3:4);
    y = lorentz(L, x) + gauss(G, x);
endfunction

// Résidus

function [e] = residus(A, x, y)
    e = profil(A, x) - y;

endfunction
```

```
// **********
// Nuage de points
// **********

for i=1:nbpts
    x = pas*i - demielargeur;
    X(i) = x;
    Y(i) = profil([paramlorentz;paramgauss], x) + rand(var, "normal");
end

// **********
// Programme principal
// **********

[S, Aopt] = leastsq(list(residus, X, Y), Ainit)
Yopt = profil(Aopt, X);
YLopt = lorentz(Aopt(1:2),X);
YGopt = gauss(Aopt(3:4),X);

// affichage

plot(X, Yopt, "-r")
plot(X, YLopt, "-c")
plot(X, YGopt, "-g")
plot(X, Y, "+b")

hauteur = paramlorentz(1)+paramgauss(1);

xstring(-demielargeur, hauteur,...
 "lorentzienne : Al = "+string(Aopt(1))+" pour "+string(paramlorentz(1))...
+" ; Hl = "+string(Aopt(2))+" pour "+string(paramlorentz(2)))
xstring(-demielargeur, 3*hauteur/4,...
 "gaussienne : Ag = "+string(Aopt(3))+" pour "+string(paramgauss(1))...
+" ; Hg = "+string(Aopt(4))+" pour "+string(paramgauss(2)))
xstring(-demielargeur, hauteur/2,...
 "variance : S = "+string(S))
```

L'algorithme par défaut de `leastsq()` est l'algorithme de pseudo-Newton. On peut aussi utiliser l'algorithme du gradient conjugué en ajoutant `"gc"` aux paramètres, ou bien algorithme foncitonnant avec les fonctions non-différentiables avec `"nd"`. On peut enfin utiliser la commande `lsqrsolve()`, qui utilise l'algorithme de Levenberg-Marquardt.

Dérivation

La dérivation peut s'aborder de différentes manières selon les données disponibles.

Pour une fonction définie

S'il s'agit d'un polynôme ou d'une fraction rationnelle, la commande `derivat()` effectue la dérivée formelle, par exemple

```
-->q = (1 + %s^2)/(1+%s)
 q  =

         2
    1 + s
    -----
    1 + s
```

```
-->derivat(q)
 ans  =

              2
  - 1 + 2s + s
    ----------
              2
    1 + 2s + s
```

Si l'on dispose d'une fonction extérieure (définie par `function()` … `endfunction` ou bien par `deff()`), alors on peut utiliser la fonction `derivative()` pour obtenir l'approximation de la matrice jacobienne et hessienne :

```
-->deff("y = f(x)", "y = x.^2")

-->derivative(f, 2) // f'(x) = 2*x ; J = f'(2) = 4
 ans  =

    4.

-->[J, H] = derivative(f, 2) // f''(x) = 2 ; H = f''(2) = 2
 H  =

    2.
 J  =

    4.

-->x=(1:3)';

-->derivative(f, x)
 ans  =

    2.    0.    0.
    0.    4.    0.
    0.    0.    6.

-->[J, H] = derivative(f, x)
 H  =

    2.    0.    0.    0.    0.    0.    0.    0.    0.
    0.    0.    0.    0.    2.    0.    0.    0.    0.
    0.    0.    0.    0.    0.    0.    0.    0.    2.
 J  =

    2.    0.    0.
    0.    4.    0.
    0.    0.    6.
```

La fonction `numdiff()` calcule le gradient. Avec la même fonction *f* et le même vecteur *x* :

```
-->numdiff(f, 2)
 ans  =

    4.

-->numdiff(f, x)
 ans  =

    2.    0.    0.
```

```
    0.    4.    0.
    0.    0.    6.
```

Pour une liste de valeurs

Si l'on dispose d'une liste de valeurs, typiquement un vecteur *x* et un vecteur *y* de même dimension, il faut alors distinguer plusieurs cas.

Valeurs peu bruitées

Si l'on suppose que le bruit est faible, donc que les valeurs sont « exactes », et que de plus la dérivée seconde est faible, alors on peut se contenter de déterminer les pentes entre les points, c'est-à-dire de calculer le taux de variation, faire une interpolation linéaire. Par exemple :

```
x = [0 2 5 20 40 60 80];
y = [0.0956820 0.0480457 0.0277857 0.0036214 0.0002543 0.0002543 0.0001265];

yprime = diff(y)./diff(x);

subplot(2, 1, 1)
plot(x, y)

subplot(2, 1, 2)
plot(x(1:$-1), yprime)
```

Si la dérivée seconde n'est pas négligeable, on peut alors utiliser une interpolation par un polynôme de degré trois (*spline* cubique) :

```
yprime1 = splin(x, y);

subplot(2, 1, 2)
plot(x, yprime1, "r")
```

Valeurs bruitées : régression et lissage

Les méthodes ci-dessus sont très sensibles au bruit. Si les valeurs sont bruitées, alors il faut lisser la courbe d'une manière ou d'une autre.

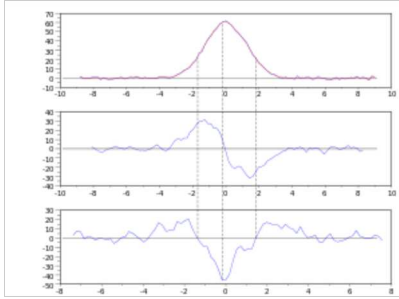
Si l'on dispose d'un modèle analytique pour les données, on peut déterminer les paramètre de ce modèle par régression, voir la section ci-dessus. Sinon, l'utilisation d'une *spline* par la commande `lsq_splin()` présentée ci-dessus permet également d'obtenir la dérivée : il suffit d'évaluer la *spline* avec la syntaxe suivante :

```
[yi, yprime2] = interp(x, xs, ys, d); // valeurs de la spline aux abscisses x
// et valeurs de la dérivée
```

L'algorithme de Savitzky-Golay présenté ci-dessus permet également d'obtenir la dérivée : comme on détermine localement un polynôme, on a les dérivées successives de ce polynôme.

Intégration

Intégrale numérique



Détermination des dérivée et dérivée seconde par l'algorithme de Savitzky-Golay.

Scilab peut calculer l'intégrale numérique d'une fonction entre deux bornes :

```
integrate("expression", "x", x0, x1)
```

calcule l'intégrale de la fonction décrite par *expression* ; *expression* est une chaîne de caractères interprétable par Scilab, comme par exemple `"sin(x)"`. Le paramètre *x* est la variable d'intégration, et les bornes sont *x0* et *x1*.

Par exemple :

```
// fonction carré
function [y] = f(x)
    y = x.^2
endfunction

// intégrale entre 0 et 4
Y = integrate("f(x)", "x", 0, 4)

// vérification
Y1 = 4^3/3
```

On peut aussi utiliser la fonction `intg(a, b, f)` où *a* et *b* sont les bornes d'intégration. Avec l'exemple précédent :

```
-->Y = intg(0, 4, f)
 Y  =

    21.333333
```

Il peut également effectuer une intégration numérique à partir d'une suite de points ($x(i)$, $y(i)$), par la méthode des trapèzes, avec la commande `inttrap()` :

```
x = linspace(0, 4, 50);
y = f(x);
inttrap(x, y)
```

Produit de convolution

Considérons deux suite de valeurs (vecteurs ligne) $(h_i)_{1 \leq i \leq m}$ et $(x_i)_{1 \leq i \leq n}$, avec $m < n$. Le produit de convolution discret est défini par :

$$y_j = \sum_{i=1}^{m} h_j \times x_{k+1-j}$$

Il est obtenu par la commande `conv(h, x)` ou bien la commande `convol(h, x)`. Par exemple, pour convoluer une fonction sinus avec une fonction porte de largeur 5 points :

```
t = linspace(0, %pi, 50);
x = sin(t);
h = 1/5*ones(1, 5);
y = conv(h, x);
plot(x)
plot(y, "r")
```

Les commandes `conv` et `convol` n'utilisent pas le même algorithme. La fonction `convol` utilise la transformée de Fourier rapide, et est mieux adaptée lorsqu'il y a beaucoup de valeurs.

Résolution d'équations

Scilab dispose de plusieurs primitives permettant la résolution d'équations. Nous présentons ci-après quatre fonctions internes, mais il en existe d'autres pour des systèmes spécifiques.

Équation polynomiale

Rappelons que la commande `root(p)` détermine les racines du polynôme *p*, c'est-à-dire que cette fonction résoud l'équation

$$p(x) = 0.$$

Équation linéaire matricielle

La commande `kernel()` permet de déterminer le noyau de la matrice, c'est-à-dire, pour une matrice M, de résoudre l'équation matricielle

$$MX = 0$$

la syntaxe est simplement :

```
X = kernel(M)
```

La division de matrice consiste à résoudre une équation linéaire :

- à droite : `X = A/B` est la solution de XB = A,
- à gauche : `X = A\B` est la solution de AX = B,
- on a `B/A = (A' \ B')'` ;

Notons que l'on peut représenter un système d'équations linéaires par une matrice ; la division matricielle est donc un moyen de résoudre un tel système.

Si la matrice A est creuse, il est recommandé d'utiliser `umfpack()` ; pour résoudre, `X = A\B` :

```
X = umfpack(A, "\" , B)
```

Par ailleurs, les résolutions numériques d'équations matricielles font souvent intervenir des décompositions d'une matrice :

- `lu(M)` : factorisation LU d'une matrice carrée, décomposition en un produit d'une matrice triangulaire inférieure L et d'une matrice triangulaire supérieure U,

- `qr(M)` : factorisation QR, décomposition en un produit d'une matrice orthogonale Q et une matrice triangulaire supérieure R,
- `svd(M)` : décomposition en valeurs singulières *(singular value decomposition)*.

Système d'équations linéaires

Scilab permet la résolution numérique d'un système d'équations linéaires, avec la fonction `linsolve`

```
[x, kerA] = linsolve(A, b)
```

où A est la matrice réelle des coefficients du système d'équations, et *b* un vecteur de constantes. Les solutions de

$A \times x + b = 0$

sont de la forme

$x_0 + w \times \mathrm{kerA}$,

w étant un réel arbitraire.

Exemple

Le système d'équation

$$\begin{cases} 3x + y = 5 \\ 4x - y = 9 \end{cases}$$

est décrit par

$$A = \begin{pmatrix} 3 & 1 \\ 4 & -1 \end{pmatrix},\ b = \begin{pmatrix} -5 \\ -9 \end{pmatrix}$$

donc

```
-->A = [3 1 ; 4 -1];

-->b = [-5 ; -9];

-->linsolve(A, b)
 ans  =

    2.
  - 1.
```

le vecteur

$$\begin{pmatrix} 2 \\ -1 \end{pmatrix}$$

représente la solution $x = 2$ et $y = -1$.
Si maintenant on tape

```
-->[x0, kerA] = linsolve(A,b)
 kerA  =
```

```
    []
 x0  =

    2.
  - 1.
```

indiquant que la solution est unique.

Exemple

S'il y a plus d'inconnues que d'équations :

$$3x + y = 5$$

```
-->A = [3 1] ; b = -5;

-->linsolve(A, b)
 ans  =

    1.5
    0.5

-->[x0, kerA] = linsolve(A,b)
 kerA  =

  - 0.3162278
    0.9486833
 x0  =

    1.5
    0.5
```

la première commande renvoie une seule solution ; la deuxième commande indique que les solutions sont de la forme

$$\begin{cases} x = 1,5 - 0,316\,227\,8 \cdot t \\ y = 0,5 + 0,948\,683\,3 \cdot t \end{cases}$$

notons qu'il s'agit d'une solution approchée, puisque la vraie solution est du type :

$$\begin{cases} x = 1,5 - \frac{1}{3} \cdot t \simeq 1,5 - 0,33\ldots \cdot t \\ y = 0,5 + t \end{cases}$$

Si le système n'admet pas de solution, Scilab affiche le message d'erreur : `WARNING:Conflicting linear constraints!.`

Scilab permet également la résolution symbolique d'un système linéaire avec la fonction `solve`

```
w = solve(A, c)
```

où A est une matrice triangulaire supérieure de réels (les coefficients du système d'équation), *c* est un vecteur de chaînes de caractères (la partie à droite des signes égal), et *w* est la matrice résultat de

A×*w* = *c*.

Exemple

Le système d'équation

$$\begin{cases} x + y = a_1 \\ y = a_2 \end{cases}$$

est décrit par

$$A = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix},\ c = \begin{pmatrix} a_1 \\ a_2 \end{pmatrix}$$

```
-->A = ["1", "1" ; "0", "1"] ; c = ["a1" ; "a2"];

-->solve(A,c)
 ans  =

!a1-a2  !
!       !
!a2     !
```

indiquant que la solution est

$x = a_1 - a_2$ et $y = a_2$.

Système d'équations quelconque

Scilab permet la résolution numérique d'un système d'équations quelconques, avec la fonction `fsolve`.

Considérons un système de *n* équations à *n* inconnues (x_1, …, x_n), de la forme

$$\begin{cases} f_1(x_1, \ldots, x_n) = 0 \\ \vdots \\ f_n(x_1, \ldots, x_n) = 0 \end{cases}$$

On définit, en tant que fonction externe (voir *Programmation* > *Fonctions*), la fonction f qui à un vecteur **x**(x_1, …, x_n) associe un vecteur **v**(vx_1, …, v_n) :

$$\mathbf{v} = f(\mathbf{x}) : \begin{cases} v_1 = f_1(x_1, \ldots, x_n) \\ \vdots \\ v_n = f_n(x_1, \ldots, x_n) \end{cases}$$

Le système d'équation consiste donc à résoudre

$f(\mathbf{x}) = \mathbf{0}$.

La syntaxe utilisée est :

```
[x] = fsolve(x0, f)
```

où `f` est la fonction externe telle que définie ci-dessus, et `x0` est une estimation de la solution. Dans le cas d'un *système* d'équations, l'entrée **x** est donc un vecteur de dimension *n*, et la fonction `f` renvoie un vecteur de dimension *n*.

Scilab utilise un algorithme du type méthode hybride de Powell. La fonction peut donner le vecteur **v** correspondant à la solution **x** estimée (**v** devant être proche de **0**), avec la syntaxe :

```
[x, v] = fsolve(x0, f)
```

Il est recommandé de fournir la jacobienne de f, sous la forme d'une fonction externe f_J :

$$\mathbf{w} = f_J(\mathbf{x}) = J\mathbf{x}$$

où J est la matrice jacobienne :

```
[x] = fsolve(x0, f, fJ)
```

Exemple

En reprenant le système d'équation ci-dessus

$$\begin{cases} 3x + y = 5 \\ 4x - y = 9 \end{cases}$$

on écrit donc

```
function [y] = f(x)
    y = [3, 1 ; 4, -1]*x - [5 ; 9];
endfunction

x0 = [0 ; 0];
solution = fsolve(x0, f)
```

ce qui donne bien (2 ; -1).

Si la fonction est paramétrée, alors

- dans la définition de la fonction, la variable doit venir avant les paramètres ;
- lors de l'appel de `fsolve()`, la fonction doit être encapsulée avec ses paramètres dans une liste.

Exemple

Considérons le système

$$\begin{cases} a \cdot x + b = 0 \\ c \cdot y + d = 0 \end{cases}$$

que l'on veut résoudre pour $a = 1$, $b = 2$, $c = 3$, $d = 4$. On écrit donc

```
function [Y] = f(X, A, b)
    Y = A*X + b;
endfunction

param1 = [1, 0 ; 0, 3];
param2 = [2 ; 4];

X0 = [0 ; 0]
```

```
solution = fsolve(X0, list(f, A = param1, b = param2));
// alternative : solution = fsolve(X0, list(f, A = param1, b = param2));
disp(solution);
```

ce qui donne comme résultat (-2 ; -1,333 333 3).

Équation différentielle ordinaire

Une d'équation différentielle ordinaire *(ordinary differential equation)*, en abrégé EDO *(ODE)*, peut être résolue de manière numérique avec la fonction `ode`. Soit une fonction $y(t)$; si l'équation différentielle est

$dy/dt = f(t,y)$,

alors f ayant été définie (fonction externe), la syntaxe pour déterminer $y(t)$ est

`y = ode(y0,t0,t,f)`

où

- *y0* et *t0* sont les valeurs initiales du système, $y(t0) = y0$,
- *t* est un vecteur de valeurs pour lesquelles on calcule les solutions, et
- *y* est le vecteur de solutions (`plot(t,y)` permet de tracer le graphique des solutions).

La fonction interne `ode` admet des arguments permettant de résoudre des situations spécifiques :

- `"roots"` pour rajouter une équation $g(t,y) = 0$,
- `"discrete"` pour calculer de manière récursive $y(k+1) = f(k,y(k))$ à partir d'un état initial $y(k0)$.

Exemple

On se place sur un segment [0,5;1] ; on veut déterminer les valeurs numériques de la fonction *y* vérifiant

$$\frac{dy}{dt} = t \text{ et } y(0) = 0$$

on a donc en fait $f(t,y) = t$, t0 = y0 = 0

```
-->t=[0.5:0.1:1]; // points de [0,5;1] espacés de 0,1

-->deff("[u] = f(t,y)", "u = t") // définition de la fonction f

-->ode(0,0,t,f)
 ans  =
    0.125    0.18    0.245    0.32    0.405    0.5
```

qui sont bien les valeurs de $1/2 \times t^2$

Note

La fonction f doit être continue et différentiable en tout point. Si la fonction présente des singularité, alors il faut la résoudre sur la première partie D^1 du domaine, puis sur la seconde partie en prenant comme conditions limites les résultats du premier calcul.

Optimisation d'une fonction

Soit une fonction f de $\mathbb{R}^n \rightarrow \mathbb{R}$. L'optimisation consiste à trouver le vecteur **x** (vecteur à *n* composantes)

donnant la valeur minimale de f.

Optimisation linéaire

La fonction f est une application linéaire ; elle peut donc s'écrire :

$$f(\mathbf{x}) = \mathbf{c}^{\mathrm{t}}\mathbf{x}$$

où **c** est un vecteur de $\mathbb{R}^n$ et $\mathbf{c}^{\mathrm{t}}$ est sa transposée. On peut par ailleurs restreindre le domaine de recherche à un polyèdre convexe décrit par *m* inéquations :

$$\mathrm{A}\mathbf{x} \leqslant \mathbf{b}$$

où

- A est une matrice $m \times n$;
- **b** est un vecteur de *m* dimensions.

Scilab dispose de la commande `karmarkar()` qui permet de résoudre le problème avec l'algorithme de Karmarkar.

La syntaxe la plus simple permet de résoudre le problème sur la frontière du polyèdre, donc avec des égalités partout :

```
xopt = karmarkar(Ae, be, c)
```

résout

$$\begin{cases} \inf_{\mathbf{x}} \mathbf{c}^{\mathrm{t}}\mathbf{x} \\ \mathrm{A_e}\mathbf{x} = \mathbf{b_e} \end{cases}$$

On peut y ajouter un système d'inéquations :

```
xopt = karmarkar(Ae, be, c, [], [], [], [], [], Ai, bi)
```

résout

$$\begin{cases} \inf_{\mathbf{x}} \mathbf{c}^{\mathrm{t}}\mathbf{x} \\ \mathrm{A_e}\mathbf{x} = \mathbf{b_e} \\ \mathrm{A_i}\mathbf{x} \leqslant \mathbf{b_i} \end{cases}$$

et si l'on n'a que des inéquations :

```
xopt = karmarkar([], [], c, [], [], [], [], [], Ai, bi)
```

résout

$$\begin{cases} \inf_{\mathbf{x}} \mathbf{c}^t\mathbf{x} \\ \mathbf{A_i x} \leqslant \mathbf{b_i} \end{cases}$$

On peut indiquer un vecteur de départ $\mathbf{x_0}$:

```
xopt = karmarkar(A, b, c, x0)
xopt = karmarkar(A, b, c, x0, [], [], [], [], Ai, bi)
xopt = karmarkar([], [], c, x0, [], [], [], [], Ai, bi)
```

et l'on peut demander de calculer la valeur de $f(\mathbf{x_{opt}})$:

```
[xopt,fopt] = karmarkar(…)
```

Optimisation non linéaire

Ici, la fonction f de $\mathbb{R}^n \rightarrow \mathbb{R}$ est non linéaire. On utilise pour cela la fonction `optim` :

```
[fopt, xopt] = optim(costf, x0)
```

où

- `costf` est la « fonction de coût » de f ; c'est une fonction qui renvoit la valeur de la fonction f en **x** et le gradient de f en **x**, défini sous la forme
  `function [f, g, ind] = costf(x, ind)`
  où `f` désigne $f(\mathbf{x})$, `g` est le gradient et `ind` est un index, un entier permettant de modifier le comportement de `costf` ;
- `x0` est une estimation de la solution ;
- `xopt` est le vecteur trouvé ;
- `fopt` = f(`xopt`), valeur estimée du minimum.

On peut indiquer des bornes inférieures et supérieure de **x**, sous al forme de vecteurs $\mathbf{x}_{\text{inf}}$ et $\mathbf{x}_{\text{sup}}$:

```
[fopt, xopt] = optim(f, x0, "b", xinf, xsup)
```

La fonction `optim()` est une méthode de quasi-Newton utilisant les critères de Wolfe.

Optimisation quadratique

Soit f une fonction quadratique de n variables $(x_i)_{1 \leq i \leq n}$, c'est-à-dire une combinaison linéaire de $x_i x_j$. Cette fonction peut se décrire par deux matrices, Q, définie positive de dimension $n \times n$, et p, matrice colonne de dimension n :

> si l'on appelle **x** la matrice colonne $[x_1 ; x_2 ; \ldots ; x_n]$
> $f(x_1, \ldots, x_n) = \frac{1}{2}{}^t\mathbf{x}\mathrm{Q}\mathbf{x} + {}^t p\mathbf{x}$
> où ^{t}M est la transposée de la matrice M.

L'optimisation quadratique consiste à trouver le vecteur **x** donnant la valeur minimum de f, en imposant de

plus que la solution se trouve dans un espace convexe, ce qui se traduit par m contraintes linéaires : m_e conditions d'égalité

$$\sum_{1 \leq j \leq n} C_{ij} x_j = b_i,\ 1 \leq i \leq m_e$$

et m - m_e conditions d'inégalité

$$\sum_{1 \leq j \leq n} D_{ij} x_j \leq d_i,\ 1 \leq i \leq m - m_e$$

soit sous forme matricielle

$$C_1 \mathbf{x} = b_1$$
$$C_2 \mathbf{x} \leq b_2$$

où

- C_1 est une matrice $m_e \times n$;
- b_1 est une matrice colonne de dimension m_e ;
- C_2 est une matrice $(m - m_e) \times n$;
- b_2 est une matrice colonne de dimension $(m - m_e)$.

On rassemble les matrices :

- $C = [C_1 ; C_2]$;
- $b = [b_1 ; b_2]$.

Pour résoudre un tel système, Scilab propose deux commandes.

La commande `qpsolve`, sous la forme :

```
[x] = qpsolve(Q, p, C, b, ci, cs, me)
```

utilise la fonction Fortran `qpgen1.f` (également appelée `QP.solve.f`), developpée par Berwin A. Turlach selon l'algorithme de Goldfarb/Idnani.

Les paramètres c_i et c_s sont des vecteurs colonne de dimension n contenant les limites inférieures et supérieures aux valeurs des x_i

$$c_i \leq \mathbf{x} \leq c_s.$$

Si l'on n'impose pas de limite, on utilise des matrices vides `[]`.

La commande `qld`, sous la forme :

```
[x, lagr] = qld(Q, p, C, b, ci, cs, me)
```

utilise la méthode de Powell modifiée par Schittkowski.

La variable *lagr* est un vecteur de multiplicateurs lagrangiens.

Le problème de la précision et de l'exactitude

Scilab est destiné à faire du calcul. La précision, l'exactitude des résultats est donc une préoccupation primordiale.

La première chose est bien évidemment d'utiliser des méthodes, des algorithmes corrects, validés, documentés. Mais la mise en code de ces algorithmes doit également être correcte. Bien évidemment, la méthode choisie doit être retranscrite fidèlement et sans erreur, mais il faut aussi prendre en compte certains « effets de bord » dus à la représentation des nombres.

En effet, Scilab calcule par défaut avec des nombre décimaux codés en virgule flottante à double précision selon la norme IEEE754. Cela permet de représenter des nombres dont la valeur absolue est comprise entre 10^{-307} et 10^{308} ; si un calcul donne une valeur inférieure, cela retourne la valeur 0 (erreur de soupassement), et s'il donne une valeur supérieure, cela retourne la valeur Inf ou -Inf (erreur de dépassement). Le nombre de chiffres significatifs est de l'ordre de 16 ; ainsi, si deux nombres diffèrent au 17^{e} chiffre significatif, ils seront considérés comme égaux (erreur d'arrondi), leur différence sera nulle, leur rapport sera de 1.

Il faut donc s'assurer d'une part que le valeur finale est représentable, mais aussi que les résultats des calculs intermédiaires sont eux aussi représentables. Par exemple le calcul de `abs(-10^-200)` donne bien le résultat 10^{-200}, alors que `sqrt((-10^-200)^2)` donne 0, puisque le résultat intermédiaire $(-10^{200})^2$ n'est pas représentable (soupassement). Dans certains cas, un algorithme itératif, cherchant une valeur approchée, peut donner un meilleur résultat qu'un calcul direct utilisant une formule exacte…

Si l'on doit faire des calculs avec des cas « extrêmes » — nombres très proches, ou bien ayant une valeur absolue élevée ou faible —, on s'intéressera à des algorithmes « robustes ». Ceux-ci font fréquemment intervenir une normalisation des données ou une factorisation pertinente. Dans la mesure du possible, si l'on a le choix entre une fonction « fait maison » et une fonction Scilab faisant le même travail, on favorisera la fonction Scilab, qui est censée être optimisée.

Si des calculs intermédiaires font intervenir des nombres positifs très grands ou très petits, on peut utiliser la fonction logarithme, qui a initialement été inventée pour cela. Par exemple, pour calculer

$$1 - \frac{365!}{(365-23)! \times 365^{23}} \simeq 0,507 \text{ avec } 365! \simeq 10^{778} \gg 10^{308}$$

on peut utiliser la fonction logarithme de la fonction gamma, `gammaln()`[1] :

```
1-exp(gammaln(365 + 1)-(gammaln(365 - 23 + 1) + 23*log(365)))
```

On peut également retravailler la formule en

$$1 - \frac{365-22}{365} \times \frac{365-21}{365} \times \ldots \times \frac{365-0}{365}$$

et utiliser

```
1 - prod( (343/365):(1/365):(364/365) )
```

Quoi qu'il en soit, la mise en œuvre d'un algorithme doit toujours s'accompagner de tests avec des valeurs pour lesquelles le résultat est connu. Les résultats de référence peuvent être calculés à la main, obtenus avec un logiciel de référence, ou bien les données peuvent être générées en fonction du résultat attendu.

Par ailleurs, un résultat de qualité fait figurer la précision du calcul.

On pourra se reporter à *(anglais)* Michaël Baudin, « Scilab is not naive (http://www.scilab.org/content/download/249/1710/file/scilabisnotnaive.pdf) » sur *Consortium Scilab*, 2010.

Voir aussi

Dans Wikipédia

- Équation différentielle ordinaire
- Solveur d'équations
- Système d'équations
- Récursivité

Liens externes

- QL - Quadratic Programming (http://www.ai7.uni-bayreuth.de/ql.htm), Université de Bayreuth
- `[pdf]` anglais Alan Edelman (MIT), *Open Source and Traditional Technical Computing* (http://www.scilab.org/content/download/395/2850/file/ScilabTec_Keynote.pdf), Scilabtec10, 16 juin 2010

Notes

1. voir *(anglais)* Computing with very large numbers in Scilab (http://mailinglists.scilab.org/Scilab-users-Computing-with-very-large-numbers-in-Scilab-tp4031570.html) sur *liste de discussion Scilab-users* et gammaln (http://help.scilab.org/docs/5.5.1/fr_FR/gammaln.html) sur *Aide de Scilab*

Calculs élémentaires < ↑ > *Graphiques et sons*

Matrices creuses

Table des matières — Index

---

4. Matrices creuses

Qu'est-ce qu'une matrice creuse ?

Une matrice creuse est une matrice dont seuls les éléments non nuls sont stockés en mémoire, par opposition à une matrice pleine dont tous les termes sont rangés en mémoire. Dans le cas de matrices diagonales, ou de matrices ne comportant qu'un faible nombre d'éléments non nuls, l'économie en terme de mémoire peut être considérable.

Outre cette économie en terme de mémoire, l'utilisation de matrices creuses peut accélérer fortement certains calculs.

Création d'une matrice creuse

La fonction `sparse` (http://help.scilab.org/docs/5.3.0/fr_FR/sparse.html) permet de créer des matrices creuses, ou de convertir une matrice pleine en matrice creuse.

Syntaxe	Description	Exemple

`sparse (A)`	Transforme la matrice `A` en une matrice creuse	`-->A = [0, 1; 2, 0]`
		` A  =`
		
		`    0.    1.`
		`    2.    0.`
		
		`-->sparse (A)`
		` ans  =`
		
		`(    2,    2) sparse matrix`
		
		`(    1,    2)        1.`
		`(    2,    1)        2.`

`sparse (ij, v [,mn])`	Crée une matrice creuse :
	■ `ij` : matrice à deux colonnes donnant la position des éléments non nuls ;
	■ `vv` : vecteur donnant la valeur des éléments non nuls ;
	■ `nm` : vecteur à deux éléments donnant la dimension de la matrice.
		`-->A = sparse ([1, 2; 4, 3], [1, 1])`
		` A  =`
		
		`(    4,    3) sparse matrix`
		
		`(    1,    2)        1.`
		`(    4,    3)        1.`
		
		`-->full (A)`
		` ans  =`
		
		`    0.    1.    0.`
		`    0.    0.    0.`
		`    0.    0.    0.`
		`    0.    0.    1.`

`sparse ([], [], [n, m])`	Crée une matrice creuse "vide" de dimension `n*m`. Équivalent au code `matlab` :
	`sparse (n, m)`
		`-->A = sparse ([], [], [2, 3])`
		` A  =`
		
		`(    2,    3) zero sparse matrix`
		
		`-->full (A)`
		` ans  =`
		
		`    0.    0.    0.`
		`    0.    0.    0.`

Matrice diagonale

La fonction `matlab spdiags` n'existe pas sous `scilab`. Pour créer une matrice diagonale, il faut passer par la fonction générique `sparse` de création d'une matrice creuse :

```
sparse ([1:n; 1:n]', d)
```

- `n` : dimension de la matrice;

- `d` : vecteur des valeurs de la diagonale.

Exemple :

```
d = [1, 5, 2, 4]
```

```
 d  =

    1.    5.    2.    4.
```

```
A = sparse ([1:4; 1:4]', d)
```

```
 A  =

(    4,    4) sparse matrix

(    1,    1)        1.
(    2,    2)        5.
(    3,    3)        2.
(    4,    4)        4.
```

```
full (A)
```

```
 ans  =

    1.    0.    0.    0.
    0.    5.    0.    0.
    0.    0.    2.    0.
    0.    0.    0.    4.
```

Notes

Voir aussi

Liens externes

- Introduction to sparse matrices in Scilab, Consortium Scilab - Digiteo - Michael Baudin (http://forge.scilab.org/index.php/p/docscisparse/downloads/)

Calcul numérique < ↑ > *Graphiques et sons*

Graphiques et sons

Table des matières — Index

---

5. Graphiques et sons

Ce chapitre traite de la création de graphique et de sons. Pour la création et la lecture de fichiers graphiques et sonores, voir *Gestion des fichiers*.

Tracé de fonction

Tracé 2D

En Coordonnées cartésiennes

Le tracé d'une fonction se fait en deux étapes

1. définir l'étendue de la variable abscisse et le pas, sous la forme d'un vecteur ligne ou colonne, par exemple
   `x = [début:pas:fin]`, ou
   `x = linspace(début, fin, nombre_de_points)` ;
2. tracer la fonction avec la commande
   `plot(x, f(x))`
   si f est la fonction.

On note qu'en fait, $f(x)$ est elle-même un vecteur. On peut de manière générale définir un vecteur des valeurs de x et un vecteur des valeurs de y, et la fonction `plot(x, y)` tracera le nuage de points $(x(i),y(i))$.

Si f est une fonction externe (par exemple définie avec `deff` ou `function`, voir *Calcul numérique*), alors on peut tracer directement la fonction avec

```
fplot2d(x, f).
```

On peut aussi définir le vecteur y par

```
y = feval(x, f),
```

puis tracer avec

```
plot(x,y).
```

Les fonctions `plot2di`, utilisées à la place de `plot`, permettent de faire varier l'apparence générale du tracé :

- `plot2d` : trait « normal » ; identique à `plot`, mais l'utilisation de marqueurs et des couleurs est différente (voir ci-après) ;
- `plot2d2` : trait en escalier ;
- `plot2d3` : tracé en barres ;
- `plot2d4` : tracé en « flèches » : les points adjacents sont reliés par une flèche ;

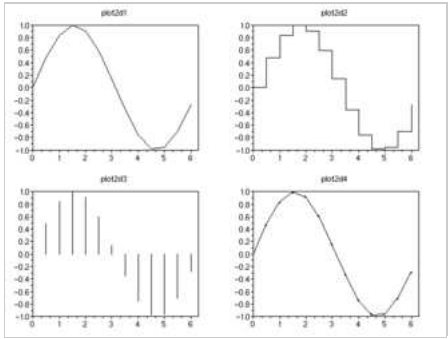
Ces fonctions `plot2di` acceptent des arguments modifiant le tracé, sous la forme

`plot2di(x, y, arguments)`.

Les arguments sont de la forme *`mot-clef = valeur`*. L'argument `rect = [xmin, ymin, xmax, ymax]` permet de limiter le tracé à la zone comprise dans le rectangle défini par les valeurs dans la matrice. D'autres arguments sont exposés plus loin.

Exemple

```
clear; clf;
x = [0:0.5:2*%pi]';

subplot(2,2,1)
plot2d1(x,sin(x),rect=[0,-1,2*%pi,1])
xtitle("plot2d1")

subplot(2,2,2)
plot2d2(x,sin(x),rect=[0,-1,2*%pi,1])
xtitle("plot2d2")

subplot(2,2,3)
plot2d3(x,sin(x),rect=[0,-1,2*%pi,1])
xtitle("plot2d3")

subplot(2,2,4)
plot2d4(x,sin(x),rect=[0,-1,2*%pi,1])
xtitle("plot2d4")
```



Modes de tracé 2d

Tracé de plusieurs courbes

On peut superposer plusieurs courbes. Il suffit d'avoir :

- pour les abscisses *x*, un vecteur colonnes ;
- pour les ordonnées *y*, une matrice Y dont chaque colonne contient les ordonnées d'une courbe.

Par exemple, si l'on a deux courbes, la matrice Y a deux colonnes, et les courbes tracées sont les courbes ($x(i)$, $Y(i, 1)$) et ($x(i)$, $Y(i, 2)$) :

```
x = (0:0.05:2*%pi)';
Y = [cos(x), sin(x)];
plot(x, Y)
```

On peut aussi simplement utiliser `plot(x, [cos(x), sin(x)])`.

Dans ce cas-là, Scilab utilise une couleur différente pour chaque courbe ; on peut inscrire une légende sous le graphique pour chaque couleur, avec la commande `legend()` :

```
legend("cosinus", "sinus");
```

Les légendes des axes peuvent contenir des expressions mathématiques LaTeX :

```
legend("$\cos \alpha$", "$\sin \alpha$");
```

Si l'on a de nombreuses courbes, on peut afficher les légendes sur plusieurs colonnes avec la fonction `legends_multicolumn()` développée par Samuel Gougeon, et téléchargeable sur le Scilab FileExchange (http://fileexchange.scilab.org/toolboxes/274000).

Si les listes d'abscisses sont différentes, alors on répète la liste des abscisses et ordonnées :

```
plot(x1, f1(x1), x2, f2(x2), x3, f3(x3))
```

On peut définir la manière dont est tracée la courbe en mettant une chaîne de caractère après la matrice des *y*. Cette chaîne de caractère peut contenir :

- un signe moins « `-` » pour indiquer que l'on trace la ligne en continu ;
- un signe pour tracer les ligne en discontinu : deux moins « `--` » (tirets longs), deux-points « `:` » (pointillets), un moins et un point « `-.` » (trait mixte) ;
- un signe pour indiquer que l'on met un marqueur : un plus « `+` », un astérisque « `*` », la lettre « `o` » (rond), un accent circonflexe « `^` » (triangle pointe en haut), inférieur « `<` » (triangle pointe à gauche), supérieur « `>` » (triangle pointe à droite), la lettre « `v` » (triangle pointe en bas), la lettre « `d` » (losange, *diamond*), la lettre « `p` » pour une étoile à cinq branches *(pentagram)*, la lettre « `s` » pour un carré *(square)*, la lettre « `x` » pour des croix de saint André ;
- une lettre indiquant la couleur : « `c` » pour cyan, « `g` » pour vert *(green)*, « `k` » pour noir *(black)*, « `m` » pour magenta, « `r` » pour rouge, « `w` » pour blanc *(white)*, « `y` » pour jaune *(yellow)*.

Ces paramètres ne fonctionnent qu'avec la fonction `plot` et pas avec les fonctions `plot2d`.

Exemple

```
x=linspace(0,%pi,50);
plot(x, sin(x), "+g", x, cos(x), "-^r")
```

trace un sinus en croix droites vertes et un cosinus en trait plein et avec des triangles en rouge.

Avec les commandes `plot2di`, on change le type de tracé (couleur, marqueur) en utilisant l'argument `style = n` ou *n* est un entier positif ou négatif (par exemple `plot2d(x, y, style = 1)`) :

- un nombre négatif remplace les points par des marqueurs : une étoile pour -10, des ronds pour -9, …, des petits points pour 0 ; la liste s'obtient en tapant la commande `getsymbol()` ;
- un nombre positif indique un trait plein mais de couleur déterminée ; on peut voir la correspondance entre le nombre et la couleur (carte des couleurs) grâce à la commande `getcolor()` (voir ci-après pour la modification de la cartes de couleur).

Avec plusieurs courbes, on place les différents styles dans un vecteur ligne, par exemple

```
plot2d(x, [cos(x), sin(x)], style = [-1, -2])
```

Notons enfin que pour Scilab, les courbes sont des objets de type *polyline*. On peut donc ajuster leur apparence (couleur, marqueur, type de tracé, …) en modifiant les propriétés de ces objets, voir la section *Propriétés des objets*.

On peut également afficher les courbes dans des sous-fenêtres séparées, voir *Utilisation de sous-fenêtres*.

En coordonnées polaires

La fonction `polarplot` permet en tracé en coordonnées polaires. La fonction *r*(*theta*) se trace par la commande :

```
polarplot(r, theta)
```

où *r* et *theta* sont deux matrices de même dimension.

Tracé de statistiques

La commande `bar()` permet de tracer un histogramme à barres verticales, et `barh()` un histogramme à barres horizontales. Les barres sont équidistantes et de même largeur. Dans sa forme la plus simple, il suffit de fournir un vecteur de valeurs à tracer ; les barres sont centrées sur les indices des valeurs dans le vecteur. Par exemple

```
y = [1 -3 5];
bar(y)
```

Trace un histogramme à trois barres centrées sur $x = 1$, $x = 2$ et $x = 3$, ayant pour altitude respective 1, -3 et 5. On peut préciser la largeur *l* sous la forme d'un nombre *l*, et éventuellement la couleur en donnant le nom en anglais entre guillemets :

```
bar(y, l, "green")
```

La variable *y* peut aussi être une matrice de *m* lignes et *n* colonnes. Alors, la commande trace *m* histogrammes, un par ligne. Chaque histogramme est centré sur l'abscisse correspondant au numéro de la ligne. Mais si l'on ajoute l'option `stacked"` (empilé), alors la commande trace un histogramme de *m* barres, une par ligne, et les barres correspondant aux valeurs de chaque ligne sont empilées. Par exemple

```
y = [1 3 5 ; 2 4 6];
bar(y, 0.3, "stacked")
```

La fonction `histplot()` analyse un jeu de données et trace l'histogramme correspondant. Si *x* est un vecteur, la fonction `histplot(`*n*`, `*x*`)`, *n* étant un entier, va découper l'intervalle de valeurs prises par les coefficients de *x* en *n* tranches d'égale largeur, et tracer l'histogramme de répartition des valeurs selon ces tranches. Si *n* est un vecteur dont les coefficients sont strictements croissants, les valeurs des coefficients de *n* servent à déterminer les tranches.

Si *x* est une matrice, `hist3d(`*x*`)` trace un histogramme 3D tel que le parallélépipède situé en (*i*, *j*) a pour hauteur *x*(*i*, *j*). Comme pour toutes les fonctions de tracé en trois dimension (voir plus loin), on peut définir l'angle de vue avec θ et α.

Carte d'un champ

Considérons un champ de scalaires *z* défini sur un plan (*x*, *y*). On a donc :

- un vecteur colonne *x* de *m* éléments,
- un vecteur colonne *y* de *n* éléments et
- *z* une matrice *m*×*n*.

Ainsi, au point de coordonnées ($x(i)$, $y(j)$) est associée la valeur $z(i, j)$.

Cartographie couleur

La fonction

```
grayplot(x, y, z)
```

associe une couleur à chaque valeur de z et trace une carte de couleurs, chaque point ($x(i)$, $y(j)$) ayant la couleur associée au coefficient $z(i, j)$. Si l'on veut afficher la légende des couleurs, il faut utiliser la commande `colobar()` *avant* la commande `grayplot()`, en lui indiquant les valeurs extrêmes :

```
zmin = min(z); zmax = max(z);
colorbar(zmin, zmax)
grayplot(x, y, z)
```

Notez que la barre de couleurs est un objet de type composé *(compound)* encapsulé dans un objet de type axes *(axes)*. Le type axes possède une propriété `title` (objet de type étiquette, *label*), qui permet de mettre un titre à la barre :

```
colorbar(zmin, zmax)
e = gce();
e.parent.title.text = "Série 1"
```

(voir plus bas *Propriétés des objets*).

La commande

```
Sgrayplot(x, y, z)
```

fait un lissage *(smoothing)* des valeurs pour avoir des dégradés de couleurs plutôt qu'un pavage. La fonction

```
Sfgrayplot(x, y, f)
```

fait une cartographie couleur lissée de $f(x, y)$.

Les fonctions `Sgrayplot()` et `Sfgrayplot()` font en fait appel à la fonction `fec()`. Cette fonction permet de faire une interpolation des valeurs et donc de représenter un dégradé, on considérant une maillage triangulaire comme ceux utilisés dans la méthode des éléments finis (FE, *finite elements*) :

- on a n nœuds, le nœud i étant défini par les coordonnes $x(i)$, $y(i)$, $z(i)$; x, y et z sont donc des vecteur de dimension n ;
- un nombre t de mailles triangulaires, une maille étant définie par son numéro j, les numéros i des trois nœuds la composant, et un entier qui n'est pas utilisé ici (utiliser la valeur 1) ; T est donc une matrice de dimension t×5 ;

l'affichage « vu de dessus » se fait par

```
fec(x, y, T, z);
```

La commande

```
Matplot(A)
```

fait une carte couleur des valeurs de la matrice A. L'abscisse *x* est le numéro de colonne, l'ordonnée *y* est le numéro de ligne, la première ligne étant en haut, respectant en cela l'organisation du tableau. Cette fonction utilise la couleur correspondante à la partie entière de la valeur.

Les niveaux de couleur sont indiqués par la fonction

```
set(gcf(), "color_map", cmap)
```

où *cmap* est une matrice de trois colonnes dont chaque ligne contient la couleur associée à un niveau, sous la forme RVB (les éléments de la matrice allant de 0 à 1). La première ligne de la matrice correspond au plus bas niveau, la dernière ligne au plus haut.

Exemple de codage RVB de couleurs

Couleur	Ligne de la matrice
Rouge	`[1, 0, 0]`
Jaune	`[1, 1, 0]`
Vert	`[0, 1, 0]`
Cyan	`[0, 1, 1]`
Bleu	`[0, 0, 1]`
Magenta vidéo	`[1, 0, 1]`
Magenta imprimerie	`[0.859, 0, 0.451]`
Blanc (100 %)	`[1, 1, 1]`
Gris clair (75 %)	`[0.75, 0.75, 0.75]`
Gris moyen (50 %)	`[0.5, 0.5, 0.5]`
Gris foncé (25 %)	`[0.25, 0.25, 0.25]`
Noir (0 %)	`[0, 0, 0]`

Cette matrice peut être générée de manière automatique par les fonctions …`colormap(`*n*`)`, où *n* est le nombre de couleurs (entier) :

`get(sdf(), "color_map")` : carte de couleurs par défaut ;
`coolcolormap(`*n*`)` : dégradé entre le cyan et le magenta ;
`coppercolormap(`*n*`)` : dégradé entre le noir et le cuivre clair (codage RVB `[1, 0.8, 0.5]`) ;
`graycolormap(`*n*`)` : niveaux de gris entre le noir et le blanc ;
`hotcolormap(`*n*`)` : dégradé entre le rouge et le jaune ;
`hsvcolormap(`*n*`)` : dégradé entre suivant le cercle chromatique *(hue, saturation, value)* : rouge, jaune, vert, cyan, bleu, magenta, et retour au rouge ;
`jetcolormap(`*n*`)` : dégradé entre le bleu et le rouge ;

`oceancolormap(n)` : dégradé de bleus ;
`pinkcolormap(n)` : sépia ;
`rainbowcolormap(n)` : dégradé selon les couleurs de l'arc-en-ciel (rouge, orange, jaune, vert, bleu, violet) ;
`springcolormap(n)` : dégradé entre le magenta et le jaune ;
`summercolormap(n)` : dégradé entre le vert et le jaune ;
`whitecolormap(n)` : blanc uniforme (matrice remplie de 1, équivalent à `ones(n, 3)`) ;
`wintercolormap(n)` : dégradé entre le bleu et le vert.

On peut par exemple utiliser

```
set(gcf(), "color_map", graycolormap(32))
```

pour avoir 32 niveaux de gris. On peut réaliser un dégradé du noir vers le rouge avec

```
cmap = graycolormap(32); cmap(:, 2:3) = 0;
set(gcf(), "color_map", cmap)
```

ou avec

```
r = linspace(0, 1, 32)'; cmap = [r zeros(32, 2)];
set(gcf(), "color_map", cmap)
```

et un dégradé du bleu vers le blanc avec

```
cmap = graycolormap(32); cmap(:, 2:3) = 1;
set(gcf(), "color_map", cmap)
```

ou avec

```
r = linspace(0, 1, 32)'; cmap = [r ones(32, 2)];
set(gcf(), "color_map", cmap)
```

Les niveaux de couleur sont également utilisés lorsque l'on trace plusieurs courbes sur le même graphique. Dans ce cas-là, des dégradés ne fournissent pas un contraste permettant de distinguer facilement des courbes voisines. La carte par défaut est bien conçue pour ce cas-là, et on peut la restaurer par

```
cmap = get(sdf(), "color_map");
set(gcf(), "color_map", cmap)
```

Courbes de niveau

On peut tracer des courbes de niveau avec la fonction

```
contour2d(x, y, z, n)
```

où n est le nombre de niveaux que l'on veut voir figurer. On peut aussi donner les valeurs des niveaux z_1, z_2,

…, z_n par un vecteur

```
contour2d(x, y, z, [z1,z2,...,zn])
```

Champ de vecteurs

On peut également tracer un champ de vecteurs, sous la forme de flèches. Pour cela, il faut un vecteur colonne *vx* ayant les composante selon *x* du champ de vecteur, un vecteur colonne *vy* ayant les composantes selon *y* de ce champ, et utiliser la fonction

```
champ(x, y, vx, vy)
```

le vecteur de composantes ($vx(i,j)$,$vy(i,j)$) étant placé en ($x(i)$, $y(j)$).

Avec la fonction `champ1`, les vecteurs tracés ont tous la même longueur, la norme du champ est indiquée par la couleur du vecteur, suivant le principe exposé pour `grayplot`.

Tracé 3D

Scilab permet également le tracé de surfaces à trois dimensions. Il s'agit en général de nuages de points de la forme(x, y, z).

De même que les courbes sont des objets « polyline », les surfaces sont des objets « fac3d » (facettes à trois dimensions).

Tracé de fonctions de deux variables

Nous voulons tracer la surface représentative d'une fonction f, on a

$$z = f(x, y).$$

Si x est une matrice colonne de m éléments, y une matrice colonne de n éléments, alors z est une matrice $m \times n$ définie par

$$z(i, j) = f(x(i), y(j))$$

alors la fonction

```
plot3d(x, y, z)
```

va tracer la surface générée par le nuage de points ($x(i)$, $y(j)$, $z(i, j)$).

Si f est une fonction « externe » (par exemple définie par `deff` ou `function`, voir *Calcul numérique*), on peut définir z avec la fonction `feval` (voir plus haut), ou bien utiliser

```
fplot3d(x, y, f)
```

On peut tracer une surface paramétrique définie par trois fonctions

$$\begin{cases} x = f_x(t) \\ y = f_y(t) \\ z = f_z(t) \end{cases}$$

Pour cela, on utilise la fonction

```
param3d(fx(t), fy(t), fz(t))
```

trace la surface paramétrique ; *t* est un vecteur contenant les valeurs successives du paramètre. Si par exemple *x*, *y* et *z* sont des vecteurs colonne de même dimension, alors

```
param3d(x, y, z)
```

trace la courbe passant par les points ($x(i)$, $y(i)$, $z(i)$).

Niveaux de couleur

Les fonctions `plot3d1`, `fplot3d1` et `param3d1` permettent d'utiliser des niveaux de couleurs pour indiquer la valeur de *z*. L'utilisation est identique à celle de `plot3d`, mais les niveaux de couleur sont indiqués par la fonction

```
set("colormap", cmap)
```

(voir ci-dessus).

Tracé de surfaces pavées

Considérons un pavage (par exemple une triangulation ou quadrillage) du plan (*x*, *y*) ; les pavés peuvent être disjoints. On a *n* pavés ; le pavé *i* est défini par un polygone de *m* points $A_{i,1}$, $A_{i,2}$, …, $A_{i,m}$ ($m = 3$ pour une triangulation, 4 pour un quadrillage).

Un point $A_{i,j}$ a pour coordonnées ($X_{i,j}$; $Y_{i,j}$). À chaque point $A_{i,j}$ est associé une cote $Z_{i,j}$.

On a donc trois matrices X, Y et Z de dimensions $m \times n$.

On trace cette surface simplement avec la fonction `plot3d` :

```
plot3d(X, Y, Z)
```

Par rapport au tracé de fonctions de deux variables, on peut passer d'une description

$M(x(i), y(j), z(i, j)$

à une description

$N(X(i, j)$; $Y(i, j)$; $Z(i, j))$

par la commande `genfac3d` :

```
[X, Y, Z] = genfac3d(x, y, z);
```

Angle de vue de la surface

Le point de vue de la surface 3D est défini par deux angles en degrés, θ, rotation autour de l'axe des *z*, et α, rotation autour de l'axe « horizontal » (une fois effectuée la rotation en *z*) : `plot3d(x, y, z, theta, alpha)`, `param3d(x, y, z, theta, alpha)...` ;

- `plot3d(x, y, z, 0, 0)` donne une vue de dessus, semblable aux cartes 2D, projection de la surface sur le plan *xy* selon l'axe *z* ; l'axe des *x* est vertical, l'axe des *y* est horizontal ;
- `plot3d(x, y, z, 90, 0)` donne aussi une vue de dessus, mais tournée de 90°, l'axe des *x* est horizontal, l'axe des *y* est vertical ;
- `plot3d(x, y, z, 0, 90)` donne une vue de face, projection de la surface sur le plan*yz* selon l'axe *x* ;
- `plot3d(x, y, z, 90, 90)` donne une vue de côté, projection de la surface sur le plan*zx* selon l'axe *y* ;
- `plot3d(x, y, z, 45, 35.5)` est la vue par défaut, de type perspective isométrique.

Plus exactement, (θ, α) désigne la direction d'observation en coordonnées sphériques.

On peut également définir les limites des axes avec le mot-clef `ebox` :

- `plot3d(x, y, z, ebox=[xmin, xmax, ymin, ymax, zmin, zmax])`
- `plot3d(x, y, z, theta, alpha, ebox=[xmin, xmax, ymin, ymax, zmin, zmax])`

Rappelons que la propriété *isoview* permet d'avoir des axes identiques :

```
a = get("current_axes"); // a = gca();
a.isoview = "on";
```

Tracé de plusieurs surfaces

On peut placer plusieurs surfaces sur le même graphique. Pour les distinguer, on peut modifier leur couleur et/ou leur style, mais cela nécessite de modifier les attributs des objets (voir ci après *Propriétés des objets*). Par exemple, pour changer la couleur

```
t = linspace(0, 2*%pi, 20)';
z1 = sin(t)*cos(t');
z2 = 10 + sin(2*t)*cos(2*t');

plot3d(t, t, z1);
s1 = gce();
s1.color_flag = 0;
s1.color_mode = 2;

plot3d(t, t, z2);
s2 = gce();
s2.color_flag = 0;
s2.color_mode = 3;

legends(["surface 1", "surface 2"], [s1.color_mode, s2.color_mode], "ur")
```

L'attribut `color_flag = 0` indique que l'on veut une couleur uniforme, et `color_mode = …` indique la couleur selon la carte de couleurs.

Pour tracer des nuages de points avec des marqueurs :

```
t = linspace(0, 2*%pi, 20)';
z1 = sin(t)*cos(t');
z2 = 10 + sin(2*t)*cos(2*t');

plot3d(t, t, z1);
s1 = gce();
s1.surface_mode = "off";
s1.mark_mode = "on";
s1.mark_style = 1;

plot3d(t, t, z2);
s2 = gce();
s2.surface_mode = "off";
s2.mark_mode = "on";
s2.mark_style = 2;

legends(["surface 1", "surface 2"], [-s1.mark_style, -s2.mark_style], "ur")
```

Concernant l'adjonction de légendes : la fonction `legend()` (au singulier) ne reconnaît que des objets de type *polyline*. Il faut donc « construire soi-même » la légende ; cela se fait avec la commande `legends()` (au pluriel) :

```
legends(["titre 1", "titre 2"], [style1, style2], placement)
```

où

- *style1* et *style2* sont des entiers positifs correspondant à la couleur de la surface, selon la carte de couleurs définie, ou bien des entiers négatifs correspondant au type de marqueur ;
- *placement* est une chaîne de caractères de type `"ul"` *(upper left)*, `"ur"` *(upper right)*, `"ll"` *(lower left)*, `"lr"` *(lower left)* ou `"below"`.

Cette fonction `legends()` crée des objets de type *texte* (les textes des légendes) et *polyline* (les traits ou symboles faisant référence aux surfaces) ; les pointeurs faisant références à ces objets (voir ci après *Propriétés des objets*) sont placés dans une matrice. Si l'on veut modifier les propriétés de la légende (police du texte, couleur des symboles), il faut identifier les pointeurs et modifier les attributs. Par exemple :

```
legends(["surface 1", "surface 2"], [s1.color_mode, s2.color_mode], "ur")
leg = gce();

leg.children(1).font_style = 2;
leg.children(3).font_style = 2;
```

la variable `leg.children` est une matrice de pointeurs ; les pointeurs 1 et 3 correspondent aux textes[1], que l'on passe en police de type Times (voir ci-après *Texte*).

Dessin

Texte

La fonction

```
xstring(x, y, "chaîne de caractères");
```

écrit « chaîne de caractère » sur le graphique, le point en bas à gauche de la boîte contenant le texte étant au point (*x*,*y*). Cela permet par exemple de placer une étiquette *(label)* sur le graphique.

On peut choisir la fonte — police (famille) et corps (taille) — de la manière suivante :

```
xstring(x, y, "chaîne de caractères");
t = gce(); // t pointe vers le texte qui vient d'être déssiné
t.font_style = 2 // référence de la police
t.font_size = 2 // corps (taille de caractères)
```

Les références de la police sont (voir l'aide *graphics_fonts*) :

- `0` : police à chasse fixe (type `Courier`) ;
- `1` : police de lettres grecques (type Symbol) ;
- `2` à `5` : police à empattement (type Times),
  - `2` : romaine, maigre,
  - `3` : *italique, maigre*,
  - `4` : **romaine, gras**,
  - `5` : ***italique, gras*** ;
- `6` à `9` : police sans empatement (type Arial),
  - `6` : romaine, maigre,
  - `7` : *italique, maigre*,
  - `8` : **romaine, gras**,
  - `9` : ***italique, gras***.

Les tailles de police vont de `0` à `10` ; `0` correspond à un corps de 8 points, `1` à un corps de 10 points, …, `10` à un corps de 54 points.

Si l'on connaît le point en *haut* à gauche de la boîte devant contenir la chaîne, on peut déterminer le point en *bas* à gauche par la commande `stringl()` :

```
rectangle = xstringl(x, yhaut, "chaîne de caractères")
```

la variable `rectangle` est une matrice `[x, ybas, largeur, hauteur]` décrivant la boîte contenant le texte. On peut préciser la fonte : référence de la police et corps (taille), par exemple

```
rectangle = xstringl(x, yhaut, "chaîne de caractères", 2, 10)
```

La chaîne de caractères peut contenir des expressions mathématiques LaTeX [2], par exemple

```
xstring(0, 0, "$x_{1, 2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$")
```

Notons que pour que le code soit interprété correctement, il faut que la chaîne commence et finisse par le signe `$`. Par exemple, si l'on veut intercaler la valeur d'une variable, il faut utiliser

```
xstring(0, 0, "$a = "+string(a)+"$")
```

Signalons que l'on peut introduire du texte « normal » au sein d'une formule LaTeX avec la commande `\text{}`. Pour utiliser la police par défaut de Scilab, il faut utiliser la commande `jmlText{}`, par exemple

```
xstring(0, 0, "$a = \jmlText{"+string(a)+"}$")
```

Formes simples

Il est possible de dessiner directement des figures géométriques sur le graphique.

La fonction

```
xpoly(x, y, "lines", 1)
```

permet de tracer un polygone fermé, `x` étant la liste des abscisses des sommets (sous la forme d'un vecteur ou d'une matrice) et `y` étant la liste des ordonnées des sommets (c'est une matrice de même dimension que `x`) ; la ligne relie les points ($x(i)$, $y(i)$). La fonction

```
xfpoly(x, y, "lines", 1)
```

permet de tracer un polygone rempli.

La fonction

```
xrect(x, y, l, h)
```

trace un rectangle dont le point en haut à gauche est (x, y), de largeur l et de hauteur h. La fonction

```
xfrect(x, y, l, h)
```

trace un rectangle rempli.

La fonction

```
xarc(x, y, l, h, a1, a2)
```

trace un arc d'ellipse compris dans le rectangle dont le point en haut à gauche est (x, y), de largeur l et de hauteur h. Chaque degré d'angle est divisé en 64 secteurs, l'arc part de l'angle trigonométrique $a1$×64 ° et va jusqu'à l'angle ($a1$+$a2$)×64 °. Pour tracer une ellipse entière, il faut donc entrer :

```
xarc(x, y, l, h, 0, 360*64)
```

La fonction

```
xfarc(x, y, l, h, a1, a2)
```

trace un arc d'ellipse rempli.

La fonction

```
xarrows(nx, ny)
```

permet de tracer des flèches. Deux cas se présentent :

- `nx` et `ny` sont des vecteurs : Scilab trace un polygone de flèches reliant les points $(n_x(i), n_y(i)) \rightarrow (n_x(i+1), n_y(i+1))$;
- `nx` et `ny` sont des matrices de deux lignes et ayant le même nombre de colonnes :
  - la première ligne de `nx` indique les abscisses des points de départ des flèches, la première ligne de `ny` en indique les ordonnées ;
  - la deuxième ligne de `nx` indique les abscisses des points d'arrivée des flèches, la deuxième ligne de `ny` en indique les ordonnées.

Propriétés des objets

Pour changer les propriétés des traits des dessins, il faut utiliser la commande :

```
set(gce(), "propriété", valeur);
```

sur l'objet courant, où *propriété* est un mot-clef (voir ci-après). On peut aussi associer une variable à l'élément dessiné ; cela se fait juste après la création de l'objet, avec la fonction `get()` :

```
variable = get("hdl");
variable = get("current_entity");
```

les deux formes sont équivalentes ; le terme `hdl`, signifiant *handle* (littéralement « poignée », intermédiaire d'action), désigne le pointeur vers l'objet. On peut aussi utiliser l'abréviation

```
variable = gce();
```

(pour *get current entity*).

L'avantage est qu'ainsi, on peut modifier les propriétés de l'objet plus tard, alors qu'il n'est plus l'objet courant.

Pour définir la propriété de cet élément, on fait alors

```
variable.propriété = valeur;
```

Ces deux méthodes sont valables quelque soit l'objet (tracé de fonction, courbe, polygone, arc d'ellipse, texte, …).

Les principales propriétés d'un objet sont (voir *Polyline_properties* dans l'aide Scilab) :

- `foreground` : couleur, désigné par son index dans la table des couleurs ;
- `line_mode` : `"on"` si la ligne est tracée, `"off"` si elle est omise ;
- `line_style` : type de trait ; c'est un entier, 0 pour un trait plein, 2 pour des pointillés longs, 3 pour des pointillés courts, 4 pour un trait d'axe (alternance trait-point)…

- `thickness` : épaisseur du trait, sous la forme d'un entier positif ;
- `mark_mode` : `"on"` si les marqueurs sont affichés, `"off"` sinon ;
- `mark_style` : nature du marqueur, `0` pour un disque plein, `1` pour une croix, …
- `mark_size` : dans le cas d'un tracé par marqueurs, taille des marqueurs ;
- `mark_foreground`, `mark_background` : couleur respectivement de la ligne et de la surface des marqueurs, désigné par son index dans la table des couleurs.

Par exemple

```
xarc(0, 1, 0.5, 0.5, 0, 360*64) // ellipse
set(gce(), "line_style", 2) // pointillés
```

ou bien

```
xarc(0, 1, 0.5, 0.5, 0, 360*64) // ellipse
a = gce(); // ou bien get("hdl")
a.line_style = 2 // pointillés
```

L'utilisation des propriétés permet d'ajuster finement le rendu du graphique, en particulier pour les tracés en perspective : les fonctions `plot3d` et `param3d` ne permettent pas de définir le style d'affichage.

Placement des objets

Il est possible de déplacer un objet de pointeur *h* d'un vecteur *v* (de dimension 2 ou 3) avec la commande `move()` :

```
move(h, v)
```

Mise en forme, axes et fenêtrage

Utilisation de sous-fenêtres

L'exemple ci-dessus montre l'utilisation de la commande `subplot()`, qui permet de diviser la fenêtre graphique. On peut ainsi mettre plusieurs graphiques côte à côte :

```
subplot(m, n, i)
```

place le tracé qui suit l'instruction dans la i^{e} case (dans l'ordre de lecture européen) d'un tableau $m \times n$. Notez que l'on peut utiliser différentes valeurs de *m* et de *n* au sein d'une fenêtre ; si par exemple on veut un graphique occupant la partie gauche dans toute sa hauteur, et deux graphiques l'un au dessus de l'autre à droite :

```
x = linspace(0, 2*%pi, 20);
y1 = x^2 ; y2 = cos(x) ; y3 = sin(x) ;

subplot(1, 2, 1) // 1re colonne, ligne unique
plot(x, y1)

subplot(2, 2, 2) // 2e colonne, 1re ligne
```

```
plot(x, y2)

subplot(2, 2, 4) // 2e colonne, 2e ligne
plot(x, y3)
```

On peut aussi utiliser la commande `xsetech()`, qui définit une fenêtre à l'intérieur de la fenêtre graphique :

```
xsetech([x, y, l, h])
```

où *x* et *y* sont les coordonnées du point en haut à gauche de la sous-fenêtre, et *l* et *h* sont la largeur et la hauteur ; ces valeurs sont des *fractions* de la fenêtre globale (donc allant de 0 à 1). On peut également définir les valeurs extrêmes des axes de cette sous-fenêtre avec un second vecteur, avec deux syntaxes possibles :

```
xsetech([x, y, l, h], [xmin, ymin, xmax, ymax])
xsetech(wrect = [x, y, l, h], frect = [xmin, ymin, xmax, ymax])
```

Organisation des entités graphiques

Pour Scilab, les entités graphiques sont organisées de manière arborescente, hiérarchique. Chaque entité a des propriétés, et les enfants de cette entité héritent de certaines propriétés par défaut, voir l'entrée *graphic_entities* dans l'aide de Scilab. Les principales entités sont, de la racine vers les feuilles :

- une fenêtre graphique appelée `figure` ;
  - un ou plusieurs systèmes d'axes, que l'on peut voir comme des sous-fenêtres, correspondant au mot-clef `axes` (pluriel) ;
    - des composés, `compound`, qui sont une enveloppe contenant un ou plusieurs objets graphiques décrits ci-dessous :
      - des axes, au sens graphique du terme (ligne graduée), correspondant au mot-clef `axis` (singulier),
      - des lignes brisées, `polyline`, qui peuvent être tracée à partir de points mais sont aussi utilisées par Scilab pour le tracé de fonctions,
      - des cartes de couleurs, `grayplot`, par exemple représentant un champ scalaire avec la commande `grayplot()`,
      - des champs de vecteurs, `champ`, obtenus par exemple avec la commande `champ`,
      - des légendes, `legend`,
      - des objets tracés directement comme des arcs d'ellipse, `arc`, des rectangles, `rectangle`, du texte, `text`,
      - des étiquettes, `label`, qui correspondent aux titres des tracés et des axes.

Chaque entité a un pointeur *(handle)*, que l'on peut obtenir avec la commande `get()`, et qui permet de faire référence à l'entité, par exemple :

```
f = get("current_figure"); // ou gcf() ; permet d'agir sur la figure
a = get("current_axes"); // ou gca() ; permet d'agir sur le système d'axes
h = get("current_entity"); // ou gce() ; permet d'agir sur l'entité (polyline, arc, ...)
```

où *f*, *a* et *h* sont des variables.

On peut fixer le contenu et un certain nombre de propriétés de ces entités par des fonctions spécifiques, comme nous l'avons vu précédemment. Mais l'on peut modifier directement une propriété. Par exemple, les commandes suivantes sont équivalentes :

```
plot(X, Y, "r");

plot(X, Y)
h = gce();
h.children.foreground = 5;
```

dans le deuxième exemple, la commande `gce()` met dans *h* le pointeur vers le dernier composé (enveloppe `compound`) créé, et `h.children` fait référence au fils de ce composé, c'est-à-dire le tracé graphique contenu dans l'enveloppe. On fixe la propriété `foreground` (couleur de premier plan) à 5, qui correspond au rouge dans la carte de couleurs par défaut.

Notons que si une entité enveloppe `compound` contient plusieurs tracés, c'est alors un vecteurs d'entités. On peut changer les propriétés des différents objets par

```
h.children(1).foreground = 3;
h.children(2).foreground = 4;
```

Création d'une fenêtre

Lorsque l'on demande à Scilab d'exécuter un tracé, il crée automatiquement une fenêtre graphique.

La commande `scf(`*n*`)` crée une fenêtre graphique dont l'étiquette (identification de figure, *figure id*) est le nombre *n*. Cela permet d'avoir plusieurs fenêtres, et donc de faire des tracés dans plusieurs fenêtres.

Le fenêtre dans laquelle se fait le tracé est la dernière fenêtre créée. On peut aussi « mettre la fenêtre dans une variable » — pointeur (*handle*, litt. poignée) — et ainsi passer d'une fenêtre à l'autre, par exemple

```
f1 = scf(1) // créée la fenêtre 1 et crée la poignée f1
f2 = scf(2) // créée la fenêtre 2 et crée la poignée f2
// La fenêtre active est la fenêtre 2 (dernière créée)
scf(f1)
// la fenêtre active est la fenêtre 1
```

Si la fenêtre existe déjà, on peut récupérer son pointeur par la commande `gcf()` *(get current figure)* :

```
f0 = gcf()
```

Paramètres de la fonction plot2d

Dans un tracé 2D, le format des axes peut se définir avec des arguments des fonctions `plot2di`.

Rappel

les arguments se placent après les vecteurs de valeurs
`plot2di(x, y, arguments)`.

On peut choisir le type d'axe avec l'argument `axesflag=`*n* où *n* est un entier positif :

- 0 pour ne pas avoir d'axe,
- 1 pour des axes « classiques » se recoupant en bas à droite,

- 4 pour des axes « classiques » se coupant au centre,
- 5 pour des axes se coupant en (0, 0)…

On peut définir le nombre de graduations et de sous-graduations des axes avec l'argument `nax = [nx, Nx, ny, Ny]` où *Nx* est le nombre de graduations de l'axe *x*, *nx* le nombre de sous-graduations…

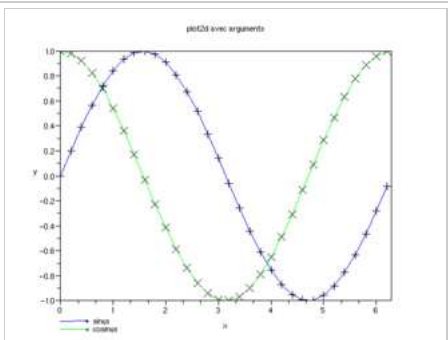
Pour une échelle logarithmique, on utilise l'argument `logflag = type` où *type* est une chaîne de deux caractères, « n » (pour normal) ou « l » (pour logarithmique), le premier caractère correspondant à l'axe des *x* et le second à l'axe des *y*. Par exemple

- `plot2d(x, y, logflag = "nl")` pour un axe des *x* linéaire et un axe des *y* logarithmique ;
- `plot2d(x, y, logflag = "ll")` pour avoir deux échelles logarithmiques.

.

Si le tracé comporte plusieurs courbes (chaque liste d'ordonnées étant une colonne d'une matrice Y), l'argument `style` est une matrice, la valeur de chaque élément indiquant le style de chaque courbe, par exemple

```
x = (0:0.05:2*%pi)';
Y = [cos(x), sin(x)];
plot2d(x, Y, style = [-2, -1])
```



Deux tracés superposés, l'un avec des styles positifs (traits pleins en couleur), l'autre avec des styles négatifs (marqueurs)

Dans ce cas-là, la command `elegend()` reprend les symboles :

```
legend("$\cos \alpha$", "$\sin \alpha$");
```

On peut aussi définir les légendes dans la commande `plot2d()` avec l'argument `leg = ("texte1@texte2")`, l'arobase « @ » servant à séparer les légendes, par exemple

```
plot2d(x, Y, style = [-2, -1], leg = "$\cos \alpha$@$\sin \alpha$")
```

Axes

Comme pour tous les objets graphiques, on peut changer les propriétés des axes en les affectant à une variable par la commande `var = get("current_axes")` ; on peut aussi utiliser l'abréviation `var = gca()`. Les principales propriétés sont (voir l'aide Scilab, entrée *axes_properties*) :

- `x_location` et `y_location` : prennent les valeurs `"top"`, `"middle"` ou `"bottom"` selon que l'on veut que l'axe soit en haut, passe par l'origine, ou soit en bas ;
- `isoview` : vaut `"on"` si l'on veut que les échelles soient identiques sur les axes, `"off"` si l'échelle est libre ;
- `log_flags` : chaîne de trois caractères, indiquant dans l'ordre qi l'axe des *x*, des *y* et le cas échéant des *z* est à graduations linéaires (`n` pour normal) ou logarithmique (`l` ; par exemple `"lln"` si les échelles en *x* et *y* sont logarithmiques et l'échelle de *z* est linéaire ;
- `sub_ticks` : matrice `[nx,ny]` où *nx* et *ny* est le nombre de sous-graduation (il faut 4 sous-graduations pour que la graduation principale soit divisée en 5) ;
- `axes_reverse` : vecteur de trois chaînes de caractères `"on"` ou `"off"` ; si une valeur est sur `"on"`, l'axe est inversé, par exemple `["off", "on", "off"]` pour inverser l'axe *y* ;
- `margin` : vecteur de quatre valeurs (marges gauche, droite, haut, bas) définissant la marge en proportion de

la taille de l'axe (`1` pour une marge de la taille de l'axe) ; cela permet par exemple de laisser de la place pour des légendes.

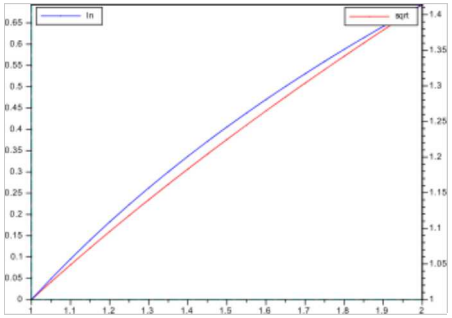
Exemple

```
a = gca();
a.x_location = "middle";
a.y_location = "middle";
a.axes_reverse(2) = "on";
a.isoview = "on";
a.sub_ticks = [4,4];
```

Comme indiqué plus haut, les objets graphiques sont organisés de manière hiérarchisée, avec des relations « parent-enfant ». Les axes sont les enfants d'une figure, et sont les parents de « composés » (en particulier des lignes polygonales obtenues par les commandes de type `plot`). On peut ainsi définir plusieurs axes sur une même figure, les composés rattachés à chaque système d'axe ayant la même échelle. Cela permet de supperposer des graphiques d'échelles différentes, avec la commande `newaxes()`, en prenant soin de rendre le fond transparent avec la propriété `filled`, et de ne pas afficher deux fois l'axes des *x* avec la propriété `axes_visible`.

Exemple

```
// Ensemble de départ
x = [1:0.05:2];

// Création de l'axe de gauche et tracé de la 1re fonctio
a1 = newaxes();
a1.tight_limits = "on";

plot(x, log(x));

legend("ln", 2);

// Création de l'axe de droite et tracé de la 2nde fonctio
a2 = newaxes();
a2.filled = "off"; // pas de couleur de fond pour ce trac
a2.axes_visible(1) = "off"; // 2nd axe x caché
a2.y_location = "right";
a2.tight_limits = "on";

plot(x, sqrt(x), "r");

legend("sqrt", 1);
```



Superposition de deux graphiques (logarithme et racine carrée entre 1 et 2) avec des échelles différentes : utilisation d'un axe à droite avec Scilab.

Enfin, il est possible de dessiner soi-même un axe avec la commande `drawaxis()`. La syntaxe globale est

```
drawaxis(x = …, y = …, dir = …)
```

Pour un axe horizontal :

- `x` doit être un vecteur de scalaires, donnant les positions des graduations ;
- `y` est un scalaire donnant la position verticale de l'axe ;
- `dir` est une chaîne de caractères valant `"u"` si les graduations sont vers le haut *(up)*, ou `"d"` si les graduations sont vers le bas *(down)* ;

par exemple

```
x = 1:10;
plot(x, x.^2);
drawaxis(x = 1:5, y = -10, dir = "u");
```

Pour un axe vertical :

- `x` est un scalaire donnant la position horizontale de l'axe ;
- `y` est un vecteur de scalaires, donnant les positions des graduations ;
- `dir` vaut `"l"` si les graduations sont du côté gauche *(left)*, ou `"r"` si elles sont du côté droit *(right)*.

On peut changer le comportement du paramètre `x` — pour un axe horizontal — ou `y` — pour un axe vertical — avec le paramètre `tics` :

- `tics = "v"` *(values)* : comportement par défaut, le vecteur indique les valeurs des graduations ;
- `tics = "r"` *(range)* : le vecteur a trois éléments, la valeur minimale, la valeur maximale et le nombre de graduations ; `x = [xmin, xmax, n]` pour un axe horizontal ;
- `tics = i` : pour des graduations de la forme mantisse *m* et exposant *a* ; le vecteur a quatre éléments, la mantisse minimale, la mantisse maximale, l'exposant et le nombre de graduations ; `x = [m1, m2, a, n]` pour un axe horizontal, les graduations vont de $m_1 \cdot 10^a$ à $m_2 \cdot 10^a$.

Autres fonctions

En tracé 2D, la fonction `xgrid` permet d'afficher une grille correspondant aux graduations.

La fonction `set` permet de définir l'apparence :

- police des graduations et titres : `set(pointeur, "font", type, taille)`, où
  - *taille* est un nombre désignant la taille des caractères, en unité arbitraire ;
  - *type* est un nombre désignant le type de police : 1 pour les lettres grecques (police Symbol), 2–5 pour une police avec empattements type Times (2 pour des lettre romanes, 3 en italique, 4 en gras, 5 en gras italique) ; 6–9 pour une police sans empattement type Arial (6 pour des lettres romaines, 7 en italique, 8 en gras, 9 en gras italique) ;
- épaisseur des traits d'axe et du tracé : `set("thickness", e)` où *e* est un nombre désignant l'épaisseur en unité arbitraire ;
- taille des marqueurs (le cas échéant, voir ci-après) avec `set("mark size", taille)`.

Le tracé peut comporter plusieurs instructions `plot()`, `set()`, `xtitle()`.

La fonction

```
xtitle("nom du graphique", "axe des x", "axe des y")
```

met un titre au graphique et aux axe. La fonction

```
titlepage("titre")
```

met un titre au milieu du graphique. Les chaînes des deux fonctions (`xtitle` et `titlepage`) peuvent contenir des expressions mathématiques LaTeX ou Mathml (voir la section *Texte*).

La fonction

```
clf
```

efface la fenêtre graphique *(clear figure)*, et la fonction

```
xdel
```

supprime toutes les fenêtres graphiques.

Scilab Graphic Editor (ged)

Le Scilab Graphic Editor est un ensemble de fonctions permettant de modifier ue fenêtre graphique de manière interactive. On appelle ces fonctions avec la commande

```
ged(action, numfig)
```

où

- *action* est un nombre entier désignant la fonction à metre en œuvre ;
- *numfig* est le numéro de la figure concernée (voir *scf() et gcf()*).

Par exemple, l'action 8 permet d'éditer les propriétés de la fenêtre et l'action 9 permet d'éditer les propriétés des axes. Par exemple

```
scf(0);
plot2d();
ged(8, 0);
```

Export direct vers un fichier

Nous avons vu précédemment comment enregistrer un graphique (voir *Gestion des fichiers > Enregistrer un graphique*.)

Il est également possible de rediriger la sortie graphique directement vers un fichier ; il n'y a donc pas d'affichage sur écran. Cela se fait en changeant le pilote *(driver)*, avec la commande `driver()`, puis en effectuant les tracés (`plot()`) dans une session d'export (entre les commandes `xinit()` … `xend()`).

Outre le fait de changer de pilote, la commande `driver()` indique le pilote courant ; par défaut, il s'agit du pilote d'affichage sur écran, qui selon les système s'appelle `Rec` ou `X11`. Les pilotes d'export disponibles sont :

- export en image vectorielle : `Pos` ou `ps` (PostScript), `PDF` ou `SVG` ;
- export en image matricielle : `PNG`, `GIF`, `JPEG` ou `JPG`, `BMP`, `PPM`.

Le code typique d'une session d'export au format SVG est le suivant :

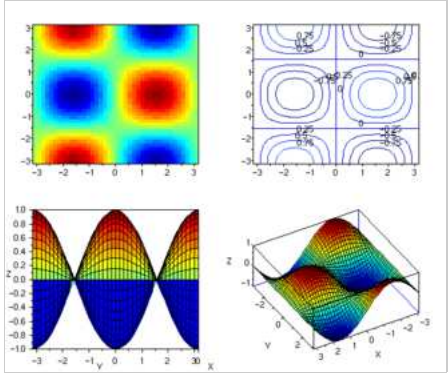
```
pilote_original = driver("SVG");
```

```
xinit("nom_de_fichier.svg")
plot(…)
xend()

driver(pilote_original)
```

Exemple

Voici le graphique que l'on obtient avec les instructions ci-dessous.

```
clear; clf;

deff("[z] = f(x, y)", "z = sin(x) * cos(y)")

set(gcf(), "color_map", jetcolormap(32))

x = %pi * [-1:0.05:1]';
y = x;
z = feval(x, y, f);

subplot(2, 2, 1)
grayplot(x, y, z)

subplot(2, 2, 2)
contour2d(x, y, z, 5)

subplot(2, 2, 3)
plot3d1(x, y, z, 0, 90)

subplot(2, 2, 4)
plot3d1(x, y, z, 70, 70)
```



Exemple de graphique obtenu avec Scilab

Tracé évolutif

Lors de calculs longs, il peut être intéressant de tracer le résultat au fur et à mesure. Pour cela, on crée un polygone *(polyline)* vide (ne contenant aucun point), on récupère son pointeur, et l'on ajoute les points un par un dans la propriété `data`. Par exemple[3] :

```
clf;
a = gca();
a.data_bounds = [0 -1;10 1];
a.axes_visible = "on";
xpoly([],[]); p = gce(); // polygone vide
for x = 0:0.1:10
    p.data=[p.data ; x, sin(x)]; // ajout d'un point
end
```

Si c'est la courbe complète qui évolue, on peut changer la totalité de la propriété data, par exemple[3]

```
clf; a = gca();
a.data_bounds = [0 -1;10 1];
a.axes_visible = "on";
t = (0:0.01:10)';
w = 1:0.1:10;
xpoly([],[]); p = gce(); // polygone vide
realtimeinit(0.01) // unité de temps = 1/100 sec.
```

```
for k = 1:length(w)
    realtime(k) // attend l'instant donné (k centièmes de sec) pour afficher
    // permet de ralentir l'affichage
    p.data = [t, sin(w(k)*t)]; // affichage de la courbe pour le paramètre w(k)
end
```

Animations

Si vous utilisez une fonction paramétrique (en fait, une fonction de deux variables), il est alors possible de considérer la deuxième variable comme le temps et d'afficher l'évolution de la courbe en fonction du temps. Ceci se fait avec la fonction `paramfplot2d`.

Pour cela, vous devez définir :

- l'ensemble des valeurs que prend la première variable, sous la forme d'un vecteur ;
- l'ensemble des valeurs que prend le paramètre, sous la forme d'un vecteur.

Si `f` est la fonction, `x` le vecteur pour la variable et `t` le vecteur pour les paramètres, alors la syntaxe est :

```
paramfplot2d(f,x,t)
```

Vous pouvez aussi faire générer plusieurs fichier images, puis ensuite générer un GIF animé avec ces images — par exemple avec JASC Animation Shop ou encore le programme Convert de la suite ImageMagick, mais nous préférerons la solution libre avec The Gimp, voir *Création d'animation .gif* de l'Atelier graphique de Wikipédia. Vous pouvez générer des noms de fichier numérotés en utilisant la fonction `string`.

Exemple

```
for i = 0:9
  for j = 1:9
    clf;
    y = feval(x, t(i*10 + j), f); // f est une fonction de x ayant un paramètre t allant
    plot2d(x,y)
    nom = "test"+string(i)+string(j)+".png";
    xs2png(0,nom,1);
  end
end
```

(Voir aussi *Gestion des fichiers* > *Enregistrer un graphique*.)

Si l'on veut afficher une animation en temps réel, on a intérêt à retarder le tracé pour tout tracer d'un coup. Cela se fait avec les commandes `drawlater()` (qui suspend le tracé) et `drawnow()` (qui déclenche les tracés suspendus).

Sons

Scilab peut générer ou analyser des sons. Il peut même lire et écrire des fichiers `.wav` (voir *Gestion des fichiers*).

Un son est en fait un vecteur dont les valeurs sont comprises entre -1 et 1 ; les valeurs hors de cet interval sont ignorées (phénomène d'écrêtage).

Si f_s est la fréquence d'échantillonage en hertz, alors le pas d'échantillonage est $1/f_s$. L'instruction `soundsec` permet de générer un vecteur correspondant au pas d'échantillonnage :

```
t = soundsec(n, fs)
```

où *n* est le nombre de secondes et f_s est la fréquence. Si l'on omet f_s, Scilab prend la valeur 22 050.

Si l'on a défini la fonction de son *s* (pression relative de l'air/tension d'alimentation normée du haut-parleur en fonction du temps), alors on peut générer un vecteur de valeurs *u* :

```
u = s(t)
```

et jouer le son

```
sound(u, fs)
```

L'instruction `analyze` détermine le spectre sonore du son et trace le graphique.

```
analyze(s)
```

Voir aussi Découvrir Scilab/Gestion des fichiers#Enregistrer et lire des fichiers de sons.

Annexes : fonctions de la bibliothèque graphique

- `plot2d` : représentation de courbes dans le plan
  - `plot2d2` : représentation d'une fonction constante par morceaux
  - `plot2d3` : représentation de courbes dans le plan sous forme de barres verticales
  - `plot2d4` : représentation de courbes dans le plan sous forme de flèches
  - `fplot2d` : représentation d'une courbe définie par une fonction
- `champ` : champ de vecteur 2D
  - `champ1` : champ de vecteur 2D, flèches colorées
  - `fchamp` : champ défini par une équation différentielle du premier ordre
- `contour2d` : courbes de niveau d'une surface
  - `fcontour2d` : courbes de niveau d'une surface (définie par une fonction)
- `vgrayplot` : représentation d'une surface en 2D sous forme de couleurs
  - `fgrayplot` : représentation d'une surface (définie par une fonction) en 2D sous forme de couleurs
- `Sgrayplot` : représentation d'une surface en 2D sous forme de couleurs interpolées
  - `Sfgrayplot` : représentation d'une surface (définie par une fonction) en 2D sous forme de couleurs interpolées
- `xgrid` : ajoute une grille sur un dessin 2D
- `errbar` : ajoute des barres d'erreur sur un dessin 2D
- `histplot` : dessine un histogramme
- `Matplot` : dessin en 2D d'une matrice en pseudo-couleurs

Dessins 3D

- `plot3d` : représentation en couleurs d'une surface
  - `plot3d1` : représentation en couleurs d'une surface
  - `fplot3d` : représente une surface non paramétrique définie par une fonction
  - `fplot3d1` : représente une surface non paramétrique définie par une fonction

- `param3d` : représente une courbe paramétrique en 3D
  - `param3d1` : représente des courbes paramétriques en 3D
- `contour` : courbes de niveau sur une surface en 3D
  - `fcontour` : courbes de niveau sur une surface en 3D définie par une fonction
- `hist3d` : représentation d'un histogramme en 3D
- `genfac3d` : calcule les facettes d'une surface non paramétrique
- `eval3dp` : calcule les facettes d'une surface paramétrique
- `geom3d` : projection 3D vers 2D après un dessin 3D

Dessins de polygones et de lignes

- `xpoly` : dessine une ligne brisée ou un polygone
- `xpolys` : dessine un ensemble de lignes brisées ou de polygones
- `xrpoly` : dessine un polygone régulier
- `xsegs` : dessine des segments déconnectés
- `xfpoly` : remplit un polygone
- `xfpolys` : remplit un ensemble de polygones

Dessins de rectangles

- `xrect` : dessine un rectangle
- `xfrect` : remplit un rectangle
- `xrects` : dessine ou remplit un ensemble de rectangles

Dessins des arcs

- `xarc` : dessine un arc d'ellipse
- `xarcs` : dessine des arcs d'ellipses
- `xfarc` : remplit un secteur angulaire d'ellipse
- `xfarcs` : remplit des secteurs angulaires d'ellipses

Dessins de flèches

- `xarrows` : dessine un ensemble de flèches

Dessins de textes

- `xstring` : dessine des chaînes de caractères
- `xstringl` : calcule une boîte contenant des chaînes de caractères
- `xstringb` : dessine des chaînes de caractères dans une boîte
- `xtitle` : ajoute des titres et légendes sur une fenêtre graphique
- `titlepage` : ajoute un titre au milieu d'une fenêtre graphique
- `xinfo` : affiche une chaîne de caractères dans la sous-fenêtre des messages

Cadres et axes

- `xaxis` : dessine un axe
- `graduate` : graduations simplifiées
- `plotframe` : dessine un cadre avec axes et mise à l'échelle

Transformations de coordonnées

- `isoview` : échelle isométrique (fonction obsolète)
- `square` : échelle isométrique (par changement de la taille de la fenêtre)
- `scaling` : transformation affine d'un ensemble de points
- `rotate` : rotation d'un ensemble de points
- `xsetech` : choix d'une sous-fenêtre graphique
- `subplot` : divise la fenêtre graphique en sous-fenêtres
- `xgetech` : donne l'échelle graphique courante
- `xchange` : transformation de coordonnées réelles vers pixels

Couleurs

- `colormap` : utilisation des tables de couleurs
- `getcolor` : boîte de dialogue pour sélectionner des couleurs dans la table des couleurs
- `addcolor` : ajout de nouvelles couleurs à la tables des couleurs
- `graycolormap` : table de couleurs du noir au blanc
- `hotcolormap` : table de couleurs « chaude » du rouge au jaune

Contexte

- `xset` : change des valeurs du contexte graphique ; cette fonction est obsolète et a été remplacée par `set()` ;
- `xget` : récupère des valeurs du contexte graphique
- `xlfont` : charge une police dans le contexte graphique ou donne la liste des polices
- `getsymbol` : boîte de dialogue pour choisir un marqueur (ou symbole)

Sauvegarde et chargement

- `xsave` : sauve les graphiques dans un fichier
- `xload` : charge des graphiques sauvés
- `xbasimp` : envoie des graphiques à une imprimante Postscript ou un fichier
- `xs2…` : envoie des graphiques dans un fichier dans un format donné (voir *Gestion des fichiers* > *Enregistrer un graphique*) :
  - `xs2bmp` : BMP (bitmap),
  - `xs2emf` : EMF (enhanced metafile),
  - `xs2eps` : EPS (encapsulated PostScript),
  - `xs2fig` : XFig,
  - `xs2gif` : GIF (graphics interchange format),
  - `xs2jpg` : JPEG (joint photographic expert group),
  - `xs2pdf` : PDF (portable document format),
  - `xs2png` : PNG (portable network graphics),
  - `xs2ppm` : PPM (portable pixmap),
  - `xs2ps` : PS (PostScript).

Fonctions internes graphiques

- `xbasc` : efface une fenêtre graphique et les graphiques enregistrés
- `xclear` : efface une fenêtre graphique
- `driver` : choisit un pilote graphique
- `xinit` : initialisation d'un pilote graphique

- `xend` : termine une session graphique
- `xbasr` : redessine une fenêtre graphique
- `replot` : redessine une fenêtre graphique avec de nouvelles bornes
- `xpause` : suspend l'exécution de Scilab
- `xselect` : met une fenêtre graphique au premier plan
- `xclea` : effacement d'un rectangle
- `xclip` : définit de la zone dans laquelle s'effectue l'affichage, ce qui sort de la zone étant tronqué *(clipping)*
- `xdel` : supprime une fenêtre graphique
- `winsid` : renvoie la liste des fenêtre graphiques
- `xname` : change le nom d'une fenêtre graphique

Position de la souris

- `xclick` : attend un clic de souris
- `locate` : sélection d'un ensemble de points
- `xgetmouse` : renvoie la position courante du pointeur de la souris

Éditeur interactif

- `edit_curv` : éditeur graphique interactif
- `gr_menu` : éditeur graphique élémentaire
- `sd2sci` : conversion d'objets gr_menu en instructions Scilab
- `ged` : éditeur interactif des propriétés des entités graphiques

Fonctions graphiques en automatique

- `bode` : diagramme de Bode
- `gainplot` : diagramme de magnitude
- `nyquist` : diagramme de Nyquist
- `m_circle` : M-circle plot
- `chart` : diagramme de Nichols
- `black` : diagramme de Black
- `evans` : lieu des pôles d'Evans
- `sgrid` : tracé de la transformée de Laplace, dans le plans des *s*
- `plzr` : tracé de pôles zéro
- `zgrid` : grille de la transformée en *z*

Voir aussi

Dans Wikipédia

- Angle
  - Angles d'Euler
- Arobase
- Coordonnées polaires
- Diagramme de Bode
- Diagramme de Nyquist
- Échelle logarithmique
- Ellipse
- Perspective isométrique

- Police d'écriture
  - Arial
  - Empattement
  - Times
- Polygone
- RVB
- Son
- Théorie du contrôle
- Transformée en Z
- WAVEform audio format

Notes

1. cela n'est pas documenté ; on peut identifier l'objet référencé par le pointeur *i* en regardant celui qui disparaît et réapparaît avec les commandes

```
leg.children(i).visible = "off"
leg.children(i).visible = "on"
```

2. grâce à la bibliothèque Java JLaTeXMath
3. [anglais Serge Steer, [Scilab-users] Still running script (17avril 2013, 18h10) (http://mailinglists.scilab.org/Scilab-users-Still-running-script-tp4026563p4026566.html)]

Matrices creuses < ↑ > *Programmation*

Programmation

Table des matières — Index

---

6. Programmation

Scilab est un langage de programmation, il accepte un certain nombre d'instructions autres que mathématiques, permettant l'exécution d'algorithmes.

L'écriture de programmes se fait idéalement avec l'éditeur de texte SciNotes ; celui-ci met en exergue les instructions en couleurs, les parenthésages (correspondance entre les paires de parenthèses et de crochets), et surligne les lignes continuées avec un fond jaune. On peut aussi utiliser un autre éditeur de texte en sauvegardant le fichier avec l'extension `.sce` ou `.sci`. Lorsque l'environnement le permet, on peut faire du copier-coller depuis l'éditeur de texte externe vers SciNotes ou bien l'éditeur de ligne de commande.

Différence entre les extensions `.sce` et `.sci`

Les extensions de nom de fichier `.sce` et `.sci` sont toutes deux utilisées pour désigner des fichiers texte contenant des scripts Scilab. D'un point de vue de l'utilisateur, il y a peu de différences fonctionnelles entre les deux. Toutefois, la logique voulue par les concepteurs est la suivante[2] (http://www.equalis.com/forums/posts.asp?group=&topic=160605&DGPCrPg=1&hhSearchTerms=&#Post160909) :

- les fichiers dont le nom porte l'extension `.sci` ne devrait contenir que des définitions de fonctions ; ce sont des bibliothèques ; en particulier, seuls les fichiers `.sci` sont chargés par la commande `getd()` (voir *Syntaxe pour définir une fonction > Fonction définie dans un fichier extérieur*) ;
- les fichiers dont le nom porte l'extension `.sce` devrait contenir le reste du script, et quelques fonctions locales.

Voir les sections suivantes :

- *Bibliothèque de fonctions* ;
- *Recommandations*.

Interaction avec l'utilisateur

Interagir avec l'utilisateur, c'est :

- lui permettre d'entrer des données, par exemple des valeurs, paramètres, faire des choix, indiquer un fichier de données, … ce qui évite d'avoir à modifier le code à chaque utilisation spécifique et rend le script plus « universel » ;
- lui permettre de récupérer les résultats.

Entrées

Nous nous contentons dans cette section des entrées au clavier par le biais de la console Scilab. Les interactions plus complexes sont présentées dans le chapitre suivant *Créer une interface graphique GUI*.

La fonction `input` permet à l'utilisateur de rentrer une valeur. la syntaxe est :

```
x = input("message")
```

où *message* est une chaîne de caractères qui s'affiche et *x* est la variable dans laquelle sera mise la valeur (ou la matrice) entrée par l'utilisateur. Si l'utilisateur doit entrer une chaîne de caractères, il faut écrire (deux possibilités) :

```
x = input("message", "string")
x = input("message", "s")
```

Lorsqu'une variable *nom_de_variable* existe, alors

```
editvar nom_de_variable
```

ouvre une fenêtre permettant de modifier la contenu de cette variable. On peut également définir la valeur d'une variable en affichant une boîte de dialogue

```
a = x_dialog("message", "a0")
```

où `a0` est la valeur initiale de *a*. Si l'on veut un message sur plusieurs lignes, on utilise une matrice de chaînes de caractères, par exemple

```
a = x_dialog(("Entrez un" ; "nombre entier"], "1")
```

On peut aussi faire cliquer l'utilisateur et récupérer les coordonnées de l'endroit cliqué :

```
[bouton, x, y] = xclick
```

la valeur de *bouton* correspond à l'action menée par l'utilisateur.

La fonction `halt` arrête l'exécution du programme jusqu'à ce que l'utilisateur appuie sur une touche.

Sorties

Nous avons vu jusqu'ici l'affichage graphique et la génération de fichiers d'image et de son (voir le chapitre précédent *Graphiques et sons*). Nous présentons ici l'affichage dans la console ; d'autres méthodes plus avancées sont présentées dans le chapitre suivant *Créer une interface graphique GUI*.

La fonction

```
print(%io(2),a)
```

affiche le contenu de la variable *a* à l'écran. On peut aussi utiliser

```
write(%io(2),a)
```

La fonction

```
disp(a)
```

affiche le contenu de *a* sans faire figurer « `a =` » devant. On peut afficher un message dans la barre d'information située en bas de la fenêtre graphique courante, avec

```
xinfo("message")
```

La fonction `warning(`*`message`*`)` affiche la chaîne de caractère *message* sous la forme d'un avertissement, c'est-à-dire précédé de « `WARNING:` ». La fonction `error(`*`message`*`)` affiche la chaîne de caractère *message* sous la forme d'un message d'erreur, c'est-à-dire précédé de « `!--error 9999` ».

Pour lancer une impression, on peut utiliser la fonction `toprint()`, qui accepte comme paramètre :

- pour imprimer le contenu d'un fichier texte : un nom de fichier (et éventuellement son chemin d'accès) sous la forme d'une chaîne de caractères ;
- pour imprimer des lignes de texte : une matrice de chaînes de caractères (une entrée par ligne de texte), avec éventuellement une chaîne additionnelle pour l'en-tête ;
- pour imprimer une figure : le numéro de la figure (défini par `scf()` ou récupéré par `get(gcf(), "figure_id")`[1]) ;

la fonction retourne un booléen indiquant si l'impression a réussi ou pas.

Par exemple :

```
status = toprint("monfichiertexte.txt"); // impression du contenu du fichier

scf(0);
plot2d();
toprint(0); // impression de la fenêtre graphique

toprint(["ligne 1", "ligne 2", "ligne 3"], "en-tête");
```

Pour les figures, on peut aussi utiliser

```
scf(0);
plot2d();
printfigure(0); // impression de la fenêtre graphique
```

qui, en outre, ouvre la boîte de dialogue de configuration de l'impression.

On peut ouvrir cette boîte de configuration pour modifier les paramètres par défaut, avec la commande

```
printsetupbox
```

Interaction avec le système d'exploitation

Il est possible de demander au système d'exploitation (SE) d'effectuer des actions.

Déterminer le SE

La commande

```
SE = getos()
```

permet de connaître le système d'exploitation courant.

Sous Microsoft Windows

La commande

```
winopen("nom_de_fichier")
```

demande au SE d'ouvrir un fichier. Pour cela, l'extension de nom de fichier doit être associée à un programme extérieur. On peut connaître ce programme avec la commande `findfileassociation()`, par exemple

```
findfileassociation(".txt")
```

Sous les Unix et Microsoft Windows

La commande `unix()` permet de faire effectuer une commande au SE, que ce soit un unix (dont un Linux, BSD ou MacOS X) ou bien Microsoft Windows. La commande renvoie un entier : si l'exécution est possible, le résultat est le code de l'interpréteur de commande, et si elle est impossible, le résultat est -1.

On peut rediriger le contenu de la sortie standard vers la fenêtre Scilab avec `unix_w()`, ou bien vers une variable avec la commande `unix_g()`, par exemple

```
if getos() == "Windows" then
    A = unix_g("dir "+SCIHOME);
else
    A = unix_g("ls "+SCIHOME);
end
```

Structures de contrôle (boucles et branchements conditionnés)

Exécution conditionnelle

L'exécution conditionnelle se fait de manière classique avec les commandes `if` *`condition`* `then, …, else, …, end`

```
if condition1 then
  instructions
  ...
elseif condition2 then
  instructions
  ...
else
  instructions
  ...
end
```

Si une variable peut prendre plusieurs valeurs, on peut utiliser la structure `select … case … else … end` :

```
select nom_variable
  case valeur1 then
    instructions
  case valeur2 then
    instructions
    ...
  case valeurn then
    instructions
else
  instructions
end
```

Boucle itérative

La syntaxe d'une boucle `for` (http://help.scilab.org/docs/5.3.0/fr_FR/for.html) est la suivante :

```
for variable=expression
do
  instructions
  …
end
```

Le mot clef `do` est optionel. Il est possible d'écrire la boucle sur une seule ligne :

```
for variable=expression, instruction, …, instruction, end
```

L'`expression` est un vecteur ou une liste contenant les différentes valeurs qui seront successivement prises par `variable`.

Exemples :

- ```
  for i=1:10 ...
  ```

  la variable `i` prend successivement les valeurs 1, 2, ..., 10.

- ```
  for i=list (1, 2, "a") ...
  ```

  la variable `i` prend successivement les valeurs 1, 2 et "a".

Boucle itérative antéconditionnée

```
while condition
  instructions1
  …
[else
  instructions2
  …]
end
while condition, instructions1, … [, else instructions2, …], end
```

Le bloc d'instructions `instructions1` est exécuté tant que `condition` est vraie. Le bloc d'instructions `instruction2` n'est exécuté qu'une seule fois dès que `condition` est fausse.

Interruption d'une boucle

Lorsque le mot clé `break` [3] (http://help.scilab.org/docs/5.3.0/fr_FR/break.html) est rencontré dans une boucle `for` ou `while`, l'exécution de la boucle est interrompue, et l'exécution du programme se poursuit par les instructions suivants immédiatement le mot clé `end` terminant la boucle.

Tri, recherche et sélection

Dans un certain nombre de cas, il faut trier les éléments d'un ensemble, ou bien rechercher des éléments répondant à certaines conditions au sein d'un ensemble. Par « ensemble », nous entendons ici un vecteur ou une matrice.

Tri

La fonction `gsort` permet de trier ; elle utilise l'algorithme de tri rapide (*quick sort*, diviser pour régner). Si V est un vecteur, alors

```
A = gsort(V)
```

trie les éléments par ordre décroissant. Pour trier par ordre croissant, il faut ajouter l'option `"i"` *(increasing ordre)* : `A = gsort(V, "i")` Si M est une matrice, alors `gsort(M)` trie la matrice en classant les éléments de haut en bas puis de gauche à droite. Si M est une matrice $m \times n$ (m lignes, n colonnes), alors l'élément M(i, j) (i-ème ligne, j-ème colonne) donnera l'indice de classement

$$I = i + (j - 1) \times m$$

Si l'on veut trier les colonnes, c'est-à-dire que les éléments restent dans la même colonne mais changent de ligne *(row)*, on utilise l'option `"r"`. Et si l'on veut trier les lignes, on utilise l'option `"c"` (*column*, les éléments changent de colonne, mais pas de ligne). Par exemple

```
M = rand(3,3)
A = gsort(M, "r", "i")
```

On peut aussi vouloir conserver les lignes intègres, ou les colonnes intègres. Pour cela, on utilise le tri « lexicographique » (tri par le premier élément, puis en cas d'égalité par le deuxième éléments, …), avec les options `"lr"` ou `"lc"` :

```
A = gsort(M, "lr", "i")
```

Recherche

La recherche se fait avec la fonction `find`.

Par exemple, pour cherche une valeur a au sein d'un vecteur V :

```
I = find(V == a);
```

la variable I retournée est alors un vecteur contenant les numéros d'indices *i* tels que V(*i*) = *v*. On peut ensuite extraire les valeurs

```
A = V(I);
```

voire les supprimer du vecteur initial

```
V(I) = [];
```

Pour être un peu plus précis : `V == a` est un vecteur de booléens. On peut fabriquer un vecteur de booléens avec tous types de conditions, par exemple `(V > a & V ~= b)`…

Avec une matrice booléenne M, la commande

```
I = find(M);
```

renvoit un vecteur d'indice obtenu en parcourant la matrice par colonnes, de haut en bas, puis de gauche à droite. Comme pour le tri, si M est une matrice *m*×*n* (*m* lignes, *n* colonnes), alors l'élément M(*i*, *j*) (*i*-ème ligne, *j*-ème colonne) donnera l'indice

$$I = i + (j - 1) \times m$$

Si l'on veut des indices « classiques », il faut utiliser la syntaxe

```
[I, J] = find(M);
```

alors I(*k*), J(*k*) sont les indices du *k*-ième élément trouvé. La valeur de l'élément est extraite par `M(I(k), J(k))`.

Dans l'exemple suivant, les deux matrices A et B sont identiques

```
M = rand(10, 10);
M > 0.5
[I, J] = find(M > 0.5);
for k = 1:size(I, "*")
    A(k) = M(I(k), J(k));
end

L = find(M > 0.5);
B = M(L);
[A, B]
```

La commande `find()` travaille uniquement sur des matrices de booléens. Cela explique que la commande ne marche pas avec les listes (voir *Structures de données avancées*).

La commande `dsearch()` regarde si chaque élément d'une matrice est dans un intervalle donné ; l'intervalle est donné sous forme de matrice. Par exemple

```
dsearch([1 3 5 7], [2 4])
```

va regarder quels sont les éléments du vecteur (1 ; 3 ; 5 ; 7) qui sont dans l'intervalle [2 ; 4]. La réponse est

```
0 1 0 0
```

car seul le second élément est dans l'intervalle. On peut indiquer des intervalles conjoints, par exemple `[2 4 6]` correspond aux intervalles I_1 = [2 ; 4] et I_2 =]2 ; 6], et l'on a

```
-->dsearch([1 3 5 7], [2 4 6])
 ans  =

    0.    1.    2.    0.
```

indiquant que le deuxième élément est dans l'intervalle 1, et le troisième éléments est dans l'intervalle 2.

Avec l'option `"d"`, `dsearch()` considère que le deuxième argument est un ensemble de nombres disjoints. Il cherche donc si chaque élément du premier argument est un élément du second :

```
-->dsearch([1 3 5 7], [3 7], "d")
 ans  =

    0.    1.    0.    2.
```

indique que le premier élément de (1 ; 3 ; 5 ; 7) est le premier élément de (3 ; 7), et que le quatrième élément de (1 ; 3 ; 5 ; 7) est le second élément de (3 ; 7).

On peut aussi localiser un vecteur-ligne ou un vecteur-colonne V au sein d'une matrice M :

```
vectorfind(M, V, "r") // vecteur-ligne
vectorfind(M, V, "c") // vecteur-colonne
```

la fonction retourne un vecteur d'indices correspondant aux lignes ou colonnes trouvées.

La version 5.5 (encore en test en octobre 2013) propose la fonction `members()` pour chercher une « aiguille dans un botte de foin » *(needle in a haystack)*, c'est-à-dire une hypermatrice dans une hypermatrice d'hypermatrices. Bien sûr, on peut s'en servir pour chercher un scalaire dans ne matrice de scalaires, mais aussi une ligne ou une colonne particulière :

```
nb = members(A, BF)
[nb, pos] = members(A, BF)
```

cherche l'aiguille A dans la botte de foin BF, qui doit être une matrice ou hypermatrice d'éléments du même type que A ; la variable `nb` est le nombre d'occurrences et `pos` les indices de la première position trouvée. On peut ajouter l'option `"last"` pour avoir la dernière occurrence :

```
[nb, pos] = members(A, BF, "last")
```

Si l'on veut chercher une ligne, alors A doit être une matrice ligne, et l'on utilise

```
members(A, BF, "rows")
```

on peut aussi chercher les permutations de A avec l'option `"shuffle"` :

```
members(A, BF, "rows", "shuffle")
```

On peut chercher des colonnes avec l'option `"cols"`.

Opérations sur les ensembles

Certaines matrices contiennent des éléments dupliqués. La commande `unique()` permet de supprimer les doublons, ou bien, vu d'une autre manière, détermine l'ensemble des valeurs dont est constituée la matrice.

Dans sa syntaxe la plus simple

```
unique(M)
```

renvoie une matrice colonne contenant chacun des éléments, classés par ordre croissant. On peut obtenir l'indice de la première occurrence de chaque éléments avec la syntaxe

```
[N, k] = unique(M)
```

Par exemple

```
A = [0, 0, 1, 1;
     0, 1, 1, 1;
     2, 0, 1, 1;
     0, 2, 2, 2;
     2, 0, 1, 1;
     0, 0, 1, 1];
-->unique(A)
 ans  =

    0.
    1.
    2.

-->[N, k] = unique(A)
 k  =

    1.
    8.
    3.
 N  =

    0.
    1.
    2.
```

Nous voyons que la matrice A, de dimension 6×4, comporte les valeurs 0, 1 et 2 — donc qu'elle est construite sur l'ensemble {0 ; 1 ; 2} —, et que la première occurrence de 0 est à l'indice 1, la première

occurrence de 1 est à l'indice 8, et la première occurrence de 2 est à l'indice 3 (comptés comme d'habitude de haut en bas, et de gauche à droite).

On peut aussi considérer les lignes uniques, avec l'option `"r"` *(rows)*, ou les colonnes uniques, avec l'option `"c"`.

```
-->unique(A, "r")
ans  =

   0.    0.    1.    1.
   0.    1.    1.    1.
   0.    2.    2.    2.
   2.    0.    1.    1.
```

La matrice A comporte donc quatre lignes originales, et deux lignes qui sont des répétitions.

La commande `union()` fait la même opération, mais sur deux matrices : elle détermine la réunion des ensembles des valeurs des deux matrices, et le cas échéant indique l'indice de première occurrence pour chacune des matrices.

Par exemple :

```
A = [0 1 2
     0 2 1];
B = [3 2 4;
     4 3 2];
-->union(A, B)
 ans  =

   0.    1.    2.    3.    4.
-->[N, ka, kb] = union(A, B)
 kb  =

   1.    2.
 ka  =

   1.    3.    4.
 N  =

   0.    1.    2.    3.    4.
```

On voit donc que la réunion des ensembles est {0 ; 1 ; 2 ; 3 ; 4}. On peut de même utiliser les options `"c"` ou `"r"`.

Temporisation

Dans certains cas, il est nécessaire de suspendre l'exécution du script, par exemple pour laisser des opérations extérieures (appels système, …) s'exécuter. On peut utiliser les commandes suivantes :

- `sleep(n)` : suspend le processus en cours pendant *n* millisecondes ;
- `xpause(n)` : suspend Scilab pendant *n* millisecondes ;
- `realtimeinit(u) ; realtime(t0) ; realtime(t1)` : la première commande, `realtimeinit(u)`, indique une unité en nombre de secondes *u* ;
  la deuxième commande, `realtime(t0)`, indique que l'instant actuel est t_0, selon l'unité précédemment définie (habituellement, $t_0 = 0$) ;

la troisième commande, `realtime(t1)`, met Scilab en attente jusqu'à ce que la date t_1 soit atteinte, selon l'unité précédemment définie .

Analyse et construction de chaînes de caractères

Les chaînes de caractères sont un moyen pratique et robuste d'échanger des informations, entre Scilab et l'utilisateur (clavier, écran) ou entre logiciels (fichiers de texte) : les codes ASCII ou UTF-8 sont normalisés et reconnus sous toutes les plateformes par la plupart des logiciels.

Nous avons déjà vu précdemment l'utilisation des commandes :

- `ascii()` (et `char()`), pour créer une chaîne de caractères à partir de codes ASCII et *vice versa* ;
- `string()`, pour transformer des nombre ou matrices de nombres en chaînes de caractères ;
- `length()`, pour déterminer la longueur d'une chaîne ;
- `+` et `strcat()`, pour concaténer des chaînes ;
- `strsubst()` qui permet de « rechercher-remplacer » ;
- `evstr()` qui permet d'exécuter une commande Scilab contenue dans une chaîne de caractères.

Pour plus de détails voir : ***Découvrir Scilab/Calculs élémentaires#Chaînes de caractères***.

Localiser une sous-chaîne

La commande `strindex()` permet de localiser la position (le *n*-ième caractère) où se trouve une sous-chaîne :

```
-->strindex("bonjour Marcel", "r M")
 ans  =

    7.
```

car la sous-chaîne « r M » commence au 7e caractère. La sous-chaîne peut également être décrite par une expression régulière (voir ci-après), il faut alors rajouter le fanion `"r"` ; on peut également utiliser la commande `regexp()` dans ce contexte. Par exemple

```
-->strindex("bonjour Marcel", "/r\ M/", "r")
 ans  =

    7.

-->regexp("bonjour Marcel", "/r\ M/")
 ans  =

    7.
```

Couper à des positions données

La commande `strsplit()` découpe une chaîne et crée un vecteur-colonne des sous-chaînes. Sans autre argument que la chaîne elle-même, elle la découpe en caractère individuels :

```
-->strsplit("abcd")
```

```
 ans  =

!a  !
!   !
!b  !
!   !
!c  !
!   !
!d  !
```

Si l'on indique un vecteur de valeurs, il coupe après les endroits indiqués :

```
-->A = strsplit("aei ou y", [4, 7])
 A  =

!aei   !
!      !
!ou    !
!      !
!y     !

-->length(A(1))
 ans  =

    4.
```

La première sous-chaîne fait 4 caractères (l'espace en position 4 est inclus) ; la deuxième sous-chaîne fait trois caractères (l'espace en position 7 est inclus).

Couper à des délimiteurs

Enfin, on peut indiquer des points de découpe soit sous la forme d'une chaîne de caractères, dite « délimiteur » ; il s'agit typiquement de virgules (le délimiteur décimal étant le point), points-virgules, d'espaces, de tabulations ou d'un carcatère qui n'est pas utilisé dans les expressions (point d'exclamation, arobase, …). Les parties de la chaîne de caractère entre les délimiteurs sont appelés « jetons » *(tokens)*.

Les délimiteurs par défaut sont l'espace et la tabulation, mais on peut définir soi-même le délimiteur. La commande `tokens()` cherche les délimiteurs dans une chaîne de caractères, découpe la chaîne en *jeton1 — délimiteur1 — jeton2 — délimiteur2…* et crée une matrice colonne avec chaque sous-chaîne.

Par exemple, la commande

```
t = tokens("a = 5", "=")
```

crée la matrice `["a " ; "=" ; " 5"]`. La commande `tokenpos()` indique la position des jetons, sous la forme d'une matrice : la première colonne indique la position du premier caractère des jetons, la seconde la position du dernier caractère.

```
-->tokenpos("a = 5", "=")
 ans  =

    1.    2.
    4.    5.

-->tokenpos("a=5", "=")
```

```
 ans  =

    1.    1.
    3.    3.
```

La commande `strplit()` fait le même travail de découpe, mais avec plus de possibilités ; le point de découpe peut être une chaîne de caractères — on coupe là où se trouve la chaîne —, mais aussi un vecteur de chaînes — on coupe là où se trouve *un des* éléments du vecteur —, ou bien une expression régulière (voir ci-après).

```
-->strsplit("aei ou y", " ")
 ans  =

!aei  !
!     !
!ou   !
!     !
!y    !
```

Ici, les caractères de séparation — les espaces — sont supprimés. La première chaîne fait 3 caractères et la deuxième en fait 2.

Les expressions régulières permettent une plus grande souplesse. Par exemple, sil l'on veut couper un nombre indéterminé d'espaces :

```
-->strsplit("aei    ou y", "/\ +/")
 ans  =

!aei  !
!     !
!ou   !
!     !
!y    !
```

Là encore, les caractères de séparation sont supprimés.

Expressions régulières

Une expression régulière est une description d'une chaîne de caractères[2]. C'est elle-même une chaîne, qui commence et finit par une barre de fraction `/`. Par exemple, l'expression régulière qui décrit « blabla » est `/blabla/`.

Il existe un certain nombre de caractères réservés, outre la barre de fraction :

```
{}[]()^$.|*+?\
```

On les appelle « métacaractères ». Si l'on veut utiliser un de ces caractères, il faut le faire précéder d'une oblique inversée. Par exemple, la chaîne « 2+2 » est décrite par `/2\+2/`, et « /slash » par `/\/slash/`.

L'oblique inversée permet également de représenter des caractères d'échappement. Par exemple, une tabulation est représentée par `\t`, une nouvelle ligne par `\n` et un retour-chariot par `\r`.

Si un caractère est répété, on peut le faire suivre par :

- un point d'interrogation `?` : il est présent au plus une fois (0 ou 1 fois) ; `/a?/` décrit «» ou bien « a » ;
- un astérisque `*` : il est absent, ou présent un nombre indéterminé de fois ; `/a*/` décrit «», « a », « aa », « aaa », … ;
- un plus `+` : il est présent au moins une fois : `/a+/` décrit « a », « aa », « aaa », … ;
- une paire d'accolades : `/a{5}/` décrit « aaaaa », `/a{2,4}/` décrit « aa », « aaa » ou « aaaa », `/a{3,}` décrit « aaa », « aaaa », « aaaaa », …

Si une portion de chaîne doit se trouver en début de ligne, l'expression régulière commence par `^` ; si elle doit se trouver en fin de ligne, elle se termine par `$`. Par exemple, `/^bon/` et `/jour$/` sont tous les deux vrais pour « bonjour ».

On peut placer plusieurs caractères substituables entre crochets ; on parle de classe de caractères. Par exemple, `/[bt]éton/` décrit les chaînes de caractère « béton » et « téton ». L'expression `/[0-9a-fA-F]/</code correspond à un chiffre hexadécimal (chifre décimal de à 9, et letre de a à z, en minuscule et en majuscule). On peut aussi interdire une classe en la commençant par l'accent circonflexe : /[^0-9]/ est tout sauf un chiffre décimal.`

Enfin, il existe des classes toutes faites :

- `/\d/` est un chiffre, l'équivalent de `/[0-9]/` ;
- `/\D/` est tout sauf un chiffre, l'équivalent de `/[^0-9]/` ;
- `/\s/` est un caractère d'espacement : blanc, tabulation, nouvelle ligne ou retour-chariot ;
- `\S` est la négation de `\s` ;
- `\w` est un caractère alphanumérique (lettre ou chiffre, ainsi que le tiret de soulignement _) ;
- `\W` est la négation de `\w` ;
- le point `.` correspond à n'importe quel caractère, sauf une nouvelle ligne `\n`.

Ainsi, `/\s+` décrit un ou plusieurs espaces.

On peut avoir des chaînes alternatives : elles sont séparées par un tube `|`, et en cas d'ambiguïté, elles sont entre parenthèses : `/(bonjour|au\ revoir)\ Marcel/` décrit les chaînes « bonjour Marcel » et « au revoir Marcel ».

Fonctions extérieures

Syntaxe pour définir une fonction

Comme indiqué dans le chapitre *Calcul numérique*, la définition d'une fonction commence par le mot clé `function` et se termine par le mot clé `endfunction`, selon la syntaxe suivante :

```
function [<arguments de sortie>] = <nom de la fonction>(<arguments d'entrée>)
  <instructions>
endfunction
```

Les instructions pouvant comporter des branchements conditionnels (tests), des boucles, … ce qui recoupe la notion de sous-programme.

Par exemple, la fonction calculant la puissance n-ième peut s'écrire :

```
function [y] = puissance(x,n)
// calcule la puissance n-ieme
```

```
  y = 1 // valeur de y^0, et valeur initiale si n>0
  if n > 0 then
    for i = 1:n
      y = y*x // on multiplie n fois x par lui-même
    end
  end
endfunction
```

Cette fonction peut ensuite être utilisée en mode interactif ou à l'intérieur d'un programme :

```
-->puissance(2,3)
 ans  =

    8.
```

Le résultat d'un calcul à l'intérieur d'une fonction n'est jamais affiché, l'utilisation de virgules ou de points-virgules est donc indifférente.

Fonction définie dans un fichier extérieur

Lorsque la fonction est définie dans un fichier texte, celui-ci peut être chargé au moyen de la commande `exec()` :

```
exec("fichier.sci")
```

Un fichier peut contenir plusieurs fonctions. Le nom de fichier peut contenir un chemin d'accès si celui-ci n'est pas dans le répertoire (dossier) courant.

La commande `getd()` permet de charger tous les fichiers `.sci` d'un répertoire, avec la syntaxe :

```
getd("chemin/")
```

où `"chemin/"` est le chemin d'accès au répertoire.

Les commandes permettant de manipuler les répertoires sont décrites au chapitre *Gestion des fichiers* > *Manipulation des répertoires*.

Fonction définie en ligne de commande

On peut définir une fonction en mode interactif, en ligne de commande. La syntaxe est identique, il suffit d'entrer les différentes lignes dans l'interpréteur. Toutes les instructions avant le mot clef `endfunction` sont considérées comme appartenant à la fonction et ne sont pas interprétées immédiatement.

Passage d'une fonction en paramètre

d'un point de vue de la représentation des données, une fonction est une variable. On peut donc la passer en paramètre. Par exemple :

```
function [y] = ce_que_je_veux(action, parametre)
   y = action(parametre)
endfunction
```

peut ensuite s'utiliser ainsi :

```
-->ce_que_je_veux(sqrt, 2)
 ans  =

    1.4142136

-->ce_que_je_veux(max, [2 -1 3 0])
 ans  =

    3.
```

Et l'on peut ainsi renommer des fonctions, par exemple :

```
-->racine_carree = sqrt
 racine_carree  =


-->racine_carree(2)
ans  =

    1.4142136
```

On voit là l'importance des parenthèses vide lorsque l'on utilise une fonction qui n'a pas de paramètre. Par exemple

```
a = gce
```

crée une copie de la fonction `gce()`, tandis que

```
a = gce()
```

met dans la variable *a* le résultat de la fonction `gce()` (voir *Graphiques et sons > Organisation des entités graphiques*).

Définition compacte d'une fonction

Une fonction peut aussi se définir en une ligne à l'aide de l'instruction `deff` :

```
deff ("[y] = f(x)", "y=2*x")
```

Cette définition est équivalente à :

```
function [y] = f(x)
  y = 2*x
endfunction
```

Cette syntaxe permet également de définir des fonctions comportant plusieurs instructions :

```
deff ("[y] = puissance(x, n)", "y = 1, if n > 0 then, for i = 1:n, y = y*x, end, end")
```

Cette écriture, bien que compacte, rend difficile la compréhension de la fonction, et ne devrait être réservé qu'aux fonctions simples.

De même qu'avec le mot clef `function`, le résultat des calculs n'est pas affiché à l'intérieur d'une fonction.

Gérer le nombre de paramètres transmis

Lorsqu'une fonction est créée pour un seul programme, on peut facilement gérer le nombre de paramètres : le programmeur sait ce qu'il doit transmettre lors de l'appel de la fonction. Mais une fonction peut être amenée à être utilisée dans d'autres programmes, par d'autre programmeurs, par exemple en étant intégrée à une bibliothèque (voir *Bibliothèque de fonctions* et *Recommandations*. Il faut alors envisager

- que certains paramètres soient optionnels ;
- que le programmeur oublie de transmettre certains paramètres (voir *Gestion des erreurs*).

Si un paramètre `p` peut avoir une valeur par défaut, on peut tester son existence avec `exists()` :

```
function [y] = monproduit(x, p)
  // paramètre optionnel
  if exists("p", "local") == 0...
    then p = 1; end // valeur par défaut
  y = x*p;
endfunction
```

On peut définir une fonction avec un nombre variable d'arguments. Le dernier argument doit alors être `varagin` *(variable arguments input)* ; dans la fonction, `varagin` est une liste, `varagin(i)` donne donc le i-ème argument optionnel. Le nombre total d'arguments est donné par la commande `argn(2)`(le « 2 » est un paramètre imposé). Par exemple

```
function [y] = produit_multiple(x, varargin)
  // paramètre optionnel
  n = argn(2);
  y = x;
  for i = 1:n - 1
    y = y*varargin(i);
  end
endfunction
```

que l'on peut tester avec

```
produit_multiple(5)
produit_multiple(5, 4)
produit_multiple(5, 4, 2)
```

Si l'on veut utiliser la totalité des paramètres, il faut utiliser `varagin(:)`. Par exemple, si l'on ne désire tracer que les dix premiers points d'un jeu de données mais en permettant de passer des paramètres pour modifier le tracé :

```
function [y] = trace_10_premiers(x, y, varargin)
  plot(x(1:10), y(1:10), varargin(:))
endfunction
```

que l'on peut tester par exemple avec

```
x = rand(1, 100)
y = rand(1, 100)
scf(0); plot(x, y)
scf(1); trace_10_premiers(x, y, "r", "thickness", 2)
```

De la même manière, on peut utiliser `varargout` *(variable arguments output)* dans les paramètres de sortie, et « remplir » la liste `varargout` dans la fonction. Le nombre total de paramètres de sortie est donné par `argn(1)`.

Informations sur une fonction

La commande `macrovar(f)` donne la liste des paramètres d'entrée et sortie de la fonction `f`, sous la forme d'un vecteur :

- première composante : paramètres d'entrée ;
- deuxième composante : paramètres de sortie ;
- troisième composante : variables utilisées mais qui ne sont pas définies à l'intérieur de la fonction ;
- quatrième composante : fonctions appelées à l'intérieur de cette fonction ;
- cinquième composante : variables locales.

La commande `fun2string(f)` renvoie un vecteur de chaînes de caractères contenant la définition de la fonction.

La commande `head_comments f` renvoie les commentaires situés au début de la fonction `f`.

On peut aussi vouloir caractériser un script complet. Une manière de faire consiste à mettre le script dans une fonction et à utiliser la commande `macrovar()`. On peut aussi comparer, avec `setdiff()`, la liste des variables, obtenue avec `who()`, avant et après le chargement du script. Par exemple

```
variables_avant = who("local");
exec("monscript.sce");
variables_du_script = setdiff(variables_avant, who("local"));
```

Variables locales ou globales

Par défaut, toutes les variables sont locales, c'est-à-dire qu'elle sont définie dans un environnement d'exécution *(workspace)*. Ainsi, une variable dans une fonction peut avoir le même nom qu'une variable dans le programme principal ou dans une autre fonction, ce seront des variables distinctes ; modifier la valeur d'une variable dans une fonction ne modifie pas la valeur de cette variable dans le programme principal.

On peut modifier ce comportement avec la commande `global()` ; c'est une manière inélégante — voire risquée, car on ne maîtrise pas totalement la valeur de la variable — de gérer une valeur sans avoir à la passer en paramètre à une fonction. Cela peut être utile lorsqu'une valeur est utilisée et modifiée par de nombreuses fonctions — cela allège la syntaxe des fonctions —, mais cela rend le code moins portable (voir la section *Du code… Général*), et il faut être sûr de quelle fonction modifie la valeur et quand, voir par exemple sur Wikipédia *Parallélisme (informatique)* > *Dépendance des données*.

Lorsque l'on veut partager une variable entre plusieurs environnements, il faut déclarer la variable comme

globale dans chaque environnement. Par exemple, le script suivant :

```
clear()

// fonctions
function modifie()
   global("a") // déclare la variable comme globale dans l'environnement de la fonction
   a = 5
endfunction

// programme principal

global("a") // déclare la variable comme globale dans l'environnement de base (programme p
a = 1;
disp("Valeur initiale : "+string(a)) // affiche la valeur initiale

modifie();

disp("Valeur finale : "+string(a)) // affiche la nouvelle valeur
```

donne

```
 Valeur initiale : 1

 Valeur finale : 5
```

On peut partager une variable entre fonctions sans qu'elle soit partagée avec l'environnement du programme principal, par exemple le script

```
clear()

// fonctions

function initialisation()
   global("a")
   a = 1
   disp("a = "+string(a))
endfunction

function utilisation()
   global("a")
   disp("a = "+string(a))
endfunction

// programme principal

errcatch(4, "continue") // n'interrompt pas le programme lorsqu'une variable
// n'est pas définie

initialisation()

disp("a = "+string(a))
errclear(4) // apurement des erreurs de type 4

utilisation()
```

donne

```
 a = 1
```

```
disp("a = "+string(a))
                      !--error 4
Variable non définie : a


 a = 1
```

ce qui montre que la variable existe dans les environnements d'exécution des fonctions, mais pas dans l'environnement d'exécution du programme principal.

On peut tester si une variable *a* est globale avec la commande

```
isglobal(a)
```

Gestion des erreurs

L'exécution d'une fonction, et en particulier un calcul, peut générer une erreur, ce qui va interrompre le déroulement du programme. On peut éviter ceci de plusieurs manières :

1. « Blinder » le programme : faire des vérifications de la validité des données à certains instants critiques.
2. Prévoir les exceptions : si l'on doit diviser par un nombre, soit on s'assure que ce nombre ne *peut pas* être nul, soit on teste la nullité et le programme agit en conséquence.
3. Intercepter les erreurs : on peut laisser se produire les erreurs, et éviter l'interruption du programme, avec les risques que cela comporte : résultat incohérent, voire pire, résultat faux mais paraissant cohérent.
4. Renvoyer des valeurs particulières : la norme IEEE 754 définit la valeur « NaN » (*not a number*, « ceci n'est pas un nombre ») et « infini », représenté dans Scilab par `%nan` et `%inf`.

Blinder un programme, cela consiste essentiellement à vérifier que le contenu des variables est conforme à leur utilisation. C'est en particulier important lorsqu'il y a une entrée de données : lecture d'un fichier ou bien saisie au clavier. Par exemple, lorsque l'on demande à l'utilisateur d'entrer un nombre entier, on vérifie qu'il s'agit bien d'un entier, et si ce n'est pas le cas, on le signale et on redemande une nouvelle saisie.

On peut donc vérifier

1. Si la variable existe.
2. Si la variable est au bon format.
3. Si le contenu est dans une plage correcte (est dans l'ensemble de définitoin des fonctions qui s'en servent).

Vérifier l'existence de la variable

La commande

```
exists(nom_de_variable)
```

où `nom_de_variable` est une chaîne de caractères (par exemple `exists("X")`), renvoie `1` si la variable existe, `0` sinon. Si l'on précise le paramètre `"l"`, la commande vérifie alors si la variable existe *en local*. Par exemple :

```
exists("X", "l")
```

placé au sein d'une fonction permet de vérifier si la variable `X` existe au sein de cette fonction ; s'il s'agit d'une variable passée en paramètre, cela permet de vérifier que l'appel de la fonction contient bien cette variable (faute de quoi la fonction risquerait d'utiliser la valeur d'une variable globale de même nom).

On peut aussi utiliser la fonction

```
isdef("X")
isdef("X", "l")
```

où `X` est la variable ; cette fonction renvoie un booléen.

Dans une fonction, on peut récupérer le nombre d'arguments d'entrée avec la commande

```
nb_arg_entree = argn(2)
```

puis à tester la valeur de `nb_arg_entree`. On pourra par exemple voir la définition de la fonction `sind` en tapant `edit sind` dans la ligne de commande.

Une autre manière de faire consiste à initialiser la variable avec un contenu vide — `a = []` — ou bien particulier — `a = %nan` —, puis à tester la valeur de la variable avant de l'utiliser (voir plus loin).

Vérifier le format de la variable

Scilab propose des fonctions de test `is...` (« est-ce… ? »), c'est-à-dire des fonctions renvoyant des valeurs booléennes `%t` (vrai) ou `%f` (faux) :

- `isglobal(X)` : vrai si la variable X est une variable globale ;
- `isvector(X)` : vrai si la variable X est un vecteur (faux si c'est un scalaire ou une matrice de plus d'une ligne et une colonne) ;
- `issparse(X)` : vrai si X est une matrice creuse ;
- `isnum(X)` : vrai si la chaîne de caractères X représente un nombre ;
- `isalphanum(X)` : renvoit un vecteur de booléens : la *i*-ème composante est vraie si le *i*-ème caractère de la chaîne X est une lettre ou un chiffre ;
- `isascii(X)` : *idem*, mais vérifie s'il s'agit d'un caractère ASCII 7 bits ;
- `isdigit(X)` : *idem*, mais vérifie s'il s'agit d'un chiffre ;
- `isletter(X)` : *idem*, mais vérifie s'il s'agit d'une lettre ;
- `isreal(X)` : vrai si la matrice X ne contient que des nombres réels ;
- `isdir(X)` : vrai si la chaîne de caractères X est un chemin d'accès à un répertoire existant ;
- `isfile(X)` : vrai si la chaîne de caractères X est un nom de fichier existant ;
- `is_handle_valid(h)` : vrai si *h* est un pointeur vers un objet graphique existant (par exemple obtenu par `unget`, ou bien généré par un `scf`) ;
- `iscell(X)`, `iscellstr(X)`, `isstruct(X)` : vrai si la variable X est respectivement une cellule, une cellule de chaîne de caractères ou une structure (voir *Structures de données avancées*) ; `isfield(S, nomduchamp)` vérifie si le champ *nomduchamp* (chaîne de caractères) existe dans la structure S, `isparam(nomliste, nomparam)` vérifie si le paramètre *nomparam* (chaîne de caractères) existe dans la liste *nomliste*.

Vérifier le contenu de la variable

Rappelons les opérations de comparaison de valeurs : ==, <, <=, >, >=. La commande `isequal(X1, X2, …, Xn)` est vraie si les *n* valeurs sont toutes égales.

Scilabe propose également les commandes :

- `isempty(X)` : vrai si la matrice ou la liste X est vide ;
- `isinf(X)`, `isnan(X)` : vérifie si la matrice X a des composantes infinies ou NaN.

Générer un message d'erreur

Lorsque l'on détecte un problème, on peut générer soi-même une erreur, avec la commande `error` (déjà présentée), ou bien un message d'avertissement, avec `warning('message')`.

On peut personnaliser le message en indiquant la valeur de la variable illégale. Pour cela, il faut soit composer une chaîne de caractères contenant la variable, par exemple

```
texte_erreur = "fonction f : valeur de l'argument "+string(X)+" illégale.";
error(texte)
```

ou bien utiliser les commandes de type « langage C », mais il faut alors être sûr du format de la variable :

```
texte_erreur = "fonction f : valeur de l'argument %d illégale.";
error(msprintf(texte, X))
```

Notons qu'un certain nombre de termes standard peuvent être traduits automatiquement selon la langue définie sur le système (paramètres de régionalisation, *localisation*) ; la commande `getext()` permet de traduire automatiquement les termes, et d'adapter la typographie (par exemple, mettre une espace[3] devant les deux-points et points-virgules en français).

Dérouter une erreur

Pour « dérouter » une erreur, c'est-à-dire éviter que le programme ne s'interrompe, on utilise la fonctions `errcatch(-1)`. La fonction `iserror(-1)` permet ensuite de tester si une erreur est survenue. On peut ne s'intéresser qu'à un type d'erreur : `errcatch(n)` et `iserror(n)` ($n > 0$) pour l'erreur *n* (voir la *table des erreurs (http://help.scilab.org/docs/5.4.0/fr_FR/error_table.html)*).

On peut indiquer à `errcatch` une action faire en cas d'erreur :

```
errcatch(n, "action")
```

où *action* vaut :

- `pause` : s'arrête et reste dans le contexte courant ; on peut utiliser `whereami()` pour connaître le contexte ; c'est le comportement le plus utile en phas ede développement/déverminage ;
- `continue` : l'erreur est ignorée et le programme continue ; il faut effacer l'erreur avec `errclear` le plus tôt possible (après `iserror`) ;
- `kill` : arrête l'exécution du programme et revient au niveau 0 (comportement normal) ;
- `stop` : quitte Scilab.

Enfin, on peut indiquer ce que doit faire Scilab en cas d'erreur de calcul, avec la commande `ieee` :

- `ieee(0)` : génère une erreur (comportement par défaut) ;
- `ieee(1)` : renvoit une valeur Inf ou Nan, et génère un avertissement ;
- `ieee(2)` : renvoit une valeur Inf ou Nan sans générer d'avertissement.

Encapsuler le code problématique

Certaines partie de code sont plus susceptibles que d'autre de produire des erreurs. C'est par exemple le cas d'une entrée-sortie hasardeuse — saisie au clavier, lecture d'un fichier volumineux —, ou bien de l'utilisation d'une opération instable — valeur pouvant provoquer un dépassement ou un soupassement… Pour gérer cette situation, on peut utiliser une structure `try-catch-end` :

- Scilab exécute le code entre les commandes `try` et `catch` ; si aucune erreur ne se produit, il va tout de suite après le `end` ;
- si une erreur se produit entre le `try` et le `catch`, il exécute directement les instructions entre le `catch`.

```
try
  <instructions "normales">
catch
  <instructions exécutées en cas d'erreur>
end
[message_erreur, numero_erreur] = lasterror(%t) // enregistrement de l'erreur
<suite du script>
```

Si le code problématique est court, on peut également l'encapsuler dans une comande `execstr()>` avec l'option `"errcatch"`.

```
numero_erreur = execstr("<instructions normales>", "errcatch", "message d.erreur")
```

S'il n'y a pas d'erreur, le numéro d'erreur est 0.

Bibliothèque de fonctions

Il est possible de créer une bibliothèque de fonctions *(library)*. Cela permet à plusieurs programmes de partager les mêmes fonctions. L'avantage est double :

- on n'écrit qu'une seule fois les fonctions, ce qui réduit la taille du code source, et donc facilite sa lecture et sa maintenance ;
- lorsque l'on corrige une erreur ou que l'on améliore la fonction, cela profite à tous les programmes.

L'inconvénient est qu'une modification a un impact sur tous les programmes, ce qui peut obliger à une gestion des différentes versions d'une fonction. Cet inconvénient est aussi un avantage : une erreur dans une fonction va avoir un impact sur tous les programmes l'utilisant, donc va être détectée plus rapidement.

La bibliothèque peut aussi contenir des variables.

On peut simplement définir un fichier `.sci` et l'appeler avec `exec()` ou `getd()` comme indiqué précédemment. Mais Scilab fournit également une gestion spécifique des bibliothèques.

Le fait de recourir à une bibliothèque plutôt qu'à un fichier `.sci` permet entre autres de gérer des fonctions ayant le même nom, par exemple des fonctions réalisant la même chose mais avec un algorithme différent

— on peut raisonnablement se demander l'intérêt de la chose, et souligner les problèmes que cela pose (est-on sûr de savoir exactement quelle fonction l'on utilise ?), mais c'est une possibilité. Par exemple, si l'on a une fonction `f()` définie dans deux bibliothèques `bib1` et `bib2`, on peut alors charger les deux bibliothèques et utiliser `bib1.f()` ou `bib2.f()` ; si l'on utilise `f()` tout court, c'est alors *a priori* la dernière bibliothèque chargée qui est utilisée.

Structure d'une bibliothèque

Pour Scilab, une bibliothèque est :

- un répertoire (ou dossier, *directory*), contenant :
- un fichier `names` (sans extension de nom de fichier), qui est un fichier texte (ASCII) contenant le nom des différentes fonctions ; c'est en quelques sortes l'index de la bibliothèque ; ce fichier peut être créé par un éditeur de texte, ou bien par Scilab avec la fonction `mputl` ;
- un fichier pour chaque fonction ; le fichier porte le nom de la fonction auquel on ajoute l'extension `.bin` ; c'est un fichier binaire (compilé) créé par la fonction `save` de Scilab.

Création d'une bibliothèque

Le programme suivant :

- se place dans le répertoire `persolib` (déjà existant), les trois petits points désignant le début du chemin d'accès ;
- définit les fonctions `gauss` et `lorentz`, ainsi que la variable `unsurpi` ;
- crée les fichiers `gauss.bin`, `lorentz.bin` et `unsurpi.bin` dans ce répertoire, à partir des fonctions définies ;
- crée le fichier d'index `names`.

```
// Création de la bibliothèque persolib

clear;

// **********
// Paramètres de la bibliothèque
// **********

cheminbiblio = ".../persolib/";

// **********
// Fonctions
// **********

function [y] = gauss(mu, sigma, x)
    // fonction gaussienne
    // mu : espérance
    // sigma : écart type
    y = 1/(sigma * sqrt(2*%pi))*exp(-1/(2*sigma^2)*(x - mu).^2)
endfunction

function [y] = lorentz(x0, Gamma, x)
    // fonction lorentzienne
    // x0 : position du sommet
    // mu : gamma : largeur à mi-hauteur (FWHM)
    HWHM = Gamma/2; // facilite l'expression
    y = ((HWHM)/%pi)*((x - x0).^2 + HWHM^2).^(-1)
endfunction
```

```
// **********
// Constantes
// **********

unsurpi = 1/%pi;

// **********
// Création de la bibliothèque
// **********

chdir(cheminbiblio)

// compilation et enregistrement des fonctions
save("gauss.bin", gauss)
save("lorentz.bin", lorentz)
save("unsurpi.bin", unsurpi)

// creation de l'index

mputl(["gauss" ; "lorentz" ; "unsurpi"], "names")
```

Utilisation d'une bibliothèque

Le programme suivant :

- s'assure que toutes les variables sont effacées (`clear`) ;
- déclare l'utilisation de la bibliothèque `.../persolib/`, et l'affecte à la variable `mabibliotheque` (fonction `lib`) ;
- utilise les fonctions et variables de la bibliothèque.

```
// Utilisation de la bibliothèque persolib

clear;

// **********
// Paramètres de la bibliothèque
// **********

cheminbiblio = ".../persolib/";

// **********
// Déclaration et chargement de la bibliothèque
// **********

mabibliotheque = lib(cheminbiblio);

// **********
// Utilisation des fonctions
// **********

X = -3:0.1:3;
Yg = gauss(1,1,X);
Yl = lorentz(1,1,X);
plot(X,Yg)
plot(X,Yl)
disp(unsurpi)
```

Commandes sur les bibliothèques

Si l'on tape simplement le nom de la variable dans laquelle on a déclarée la bibliothèque (ici

`mabibliotheque`), Scilab retourne :

- le chemin d'accès ;
- la liste des fonctions.

La fonction `libraryinfo` a le même effet, mais permet de mettre les valeurs retournées dans des variables, par exemple avec

```
[fcts, chem] = libraryinfo(mabibliotheque)
```

la variable `fcts` est une matrice de chaînes de caractères contenant les noms des fonctions, et la variable `chem` est une chaîne de caractères contenant le chemin d'accès.

La fonction `isdef(mabibliotheque)` teste si la bibliothèque est déclarée.

La fonction `clear mabibliotheque` efface les fonctions et variables définies par la bibliothèque.

La fonction `whereis` indique dans quelle bibliothèque se trouve une fonction (mais pas une variable). Par exemple, `whereis(gauss)` retourne `mabibliotheque`.

La fonction `librarieslist` indique la liste des bibliothèques chargées.

Lancement d'un programme

Un programme (fichier `.sce`) peut être lancé depuis Scinotes, en cliquant sur le menu **Execute** puis en choisissant **Load into Scilab**. On peut aussi utiliser la combinaison de touches `CTRL+l`.

On peut aussi l'exécuter à partir de l'interpréteur de commandes, avec `exec('nomprogramme.sce', -1)`. Le paramètre « -1 » permet de ne pas afficher les lignes exécutées.

Optimisation d'un programme

Si un programme est destiné à traiter des données volumineuses, il devient important de s'intéresser à la durée d'exécution. Plusieurs outils permettent de comparer les durées d'exécution et ainsi de faire des choix entre plusieurs solutions possibles pour effectuer une tâche (réaliser une fonction, au sens large de fonctionnalité).

Tout d'abord, Scilab peut lire l'horloge du système. La commande `clock` retourne un vecteur ligne de 6 éléments indiquant :

1. L'année.
2. Le mois (1–12).
3. Le jour du mois (1—31).
4. L'heure (1–24).
5. Les minutes dans l'heure (0–59).
6. Les secondes dans la minute (0–59)[4].

Mais cela ne permet d'avoir une précision que de la seconde. On peut avoir les millisecondes avec :

```
instant = getdate();
```

```
instant(9) + instant(10)/1000
```

en effet, la commande `getdate()` renvoie un vecteur dont la neuvième composante est la seconde (nombre entier entre 0 et 59) et la dixième composante est la milliseconde (nombre entier entre 0 et 999).

Scilab dispose également d'un chronomètre :

- la commande `tic()` déclenche le chronomètre ;
- la commande `toc()` indique la valeur du chronomètre.

L'exécution d'une fonction externe peut être caractérisé par son « profil ». Le profil est une matrice ayant le même nombre de lignes que la fonction (la 1re ligne est celle contenant la commande `function`). Pour chaque ligne, le profil contient 3 colonnes :

1. Nombre de fois que la ligne exécutée.
2. Durée d'exécution de la ligne, en seconde.
3. « Effort » requis par l'interpréteur pour exécuter la ligne.

La commande `add_profiling("`*`nom_fonction`*`")` « prépare » la fonction : cette commande est nécessaire pour extraire ensuite le profil. La commande `profile(`*`nom_fonction`*`)` indique le « profile de la fonction ».

La commande `showprofile(`*`nom_fonction`*`)` met en correspondance ces données et les instructions de la fonctions. La commande `plotprofile(`*`nom_fonction`*`)` synthétise le profil sous la forme d'un histogramme.

Conseils

Un programme rapide est un programme qui effectue peu d'opérations. Le premier point est donc d'utiliser un algorithme « économe ». Par exemple, pour le calcul de la valeur d'un polynôme, on utilisera la méthode de Horner. L'algorithme le plus économe peut dépendre de la structure des données, c'est le cas notamment des algorithmes de tri — voir par exemple le tri par tas et le tri rapide.

Certaines opérations sont plus gourmandes que d'autres ; pour de gros volumes de données, il peut être intéressant de ne pas utiliser l'écriture « naïve » de ce que l'on veut calculer. Par exemple, la division utilise plus d'opérations que la multiplication, ainsi, si l'on remplace[5]

$$\begin{cases} e = a/b \\ f = c/d \end{cases}$$

par

$$\begin{cases} dn = 1/(b \times d) \\ e = a \times d \times dn \\ f = c \times b \times dn \end{cases}$$

on remplace deux divisions par une division et cinq multiplications. Ceci est favorable pour de gros volumes de données, comme le montre l'exemple suivant

```
// Génération des données
n = 9e5;
a = rand(n, 1);
b = rand(n, 1)+1;
```

```
c = rand(n, 1);
d = rand(n, 1)+1;

// Calcul naïf
tic();
e = a./b;
f = c./d;
t1 = toc();

// Calcul optimisé
tic();
dn = 1 ./(b.*d);
E = a.*d.*dn;
F = c.*b.*dn;
t2 = toc();

// Comparaison
disp(t1, t2);
```

Par contre, les erreurs d'arrondis ne sont pas les même, les résultats peuvent donc être légèrement différents, et il faut vérifier que cela ne cause pas d'erreur de dépassement ou de soupassement.

Lorsque Scilab effectue un calcul, il commence par vérifier le type de données et les dimensions de la matrice sur laquelle il travaille — rappelons qu'un nombre est une matrice 1×1 —, afin de savoir comment il va mener le calcul (arbre de décision). Par exemple, pour effectuer une addition `A+B`, Scilab effectue une dizaine d'opérations préalables[6] :

1. Allocation de mémoire pour `A`.
2. Allocation de mémoire pour `B`.
3. Détermination du type de données de `A` : matrice réelle ou complexe (type 1), matrice polynomiale (2), matrice booléenne (3), matrice creuse (4), matrice creuse booléenne (5), matrice de chaînes de caractères (10), …
4. Détermination du type de données de `B`.
5. Détermination de la dimension de la matrice `A`.
6. Détermination de la dimension de la matrice `B`.
7. Si `A` est une matrice réelle ou complexe, s'agit-i-il de nombre réels ou de nombres complexes ?
8. *Idem* pour `B`.
9. La mémoire libre est-elle suffisante pour contenir le résultat ?

Si l'on a des calculs répétitifs à faire, il vaut donc mieux mettre les données dans une matrice et effectuer le calcul sur la matrice : ainsi, les opérations de contrôle et de décision ne seront faites qu'une seule fois.

De manière générale, un calcul est plus long qu'une opération de lecture/écriture en mémoire. Ainsi, si un calcul est répété, on a intérêt à stocker son résultat dans une variable, en particulier s'il s'agit d'un calcul long (par exemple sur sur une matrice). Si par exemple on utilise plusieurs fois $\sin(\pi/3)$, on a intérêt à créer la variable

```
sinpt = sin(%pi/3)
```

et à utiliser cette variable, plutôt que de calculer à chaque fois la valeur.

Dans un tableau (matrice), si l'on s'attend à ce que des valeurs volumineuses (par exemple des nombre à virgule flottante à double précision) soient déplacées plusieurs fois, on peut créer une matrice de correspondance entre indices (donc ne contenant que des entiers), et faire les transferts uniquement à la fin ; ainsi, les opérations de lecture/écriture fréquentes se font sur des données peu volumineuses.

On voit que l'optimisation consiste fréquemment à trouver un équilibre entre volume de données et durée des opérations. Si l'on génère trop de données intermédiaires (dépassant la capacité de la mémoire vive), cela oblige le système d'exploitation à avoir recours à de la mémoire virtuelle *(swap)* et donc à faire des opérations de lecture et écriture sur disque dur (ou mémoire flash) qui ralentissent l'exécution.

Déverminage

Pour déverminer (« débugger ») un programme, on peut :

- mettre des fonctions `halt` dans le code, afin d'interrompre l'exécution à certain endroit pour effectuer des vérifications ; on relance le programme en appuyant sur la touche `[entrée]` dans l'invite de commande ;
- lancer le programme en mode 4 : `exec('nomprogramme.sce', 4)` ; le programme s'arrête à chaque ligne ; on relance le programme en appuyant sur la touche `[entrée]` dans l'invite de commande.

Recommandations

Les « bonnes pratiques de programmation » ont pour but de faciliter la conception et la maintenance des programmes. Cela inclue la possibilité de reprendre des programmes plusieurs années après, de les adapter aux nouvelles versions des logiciels, de réutiliser des bouts de code dans d'autres programme, que d'autres personnes réutilisent le code, l'adapte, … bref créer du code directement réutilisable sans modification *(boilerplate code)*. Rappelons que 80 % du coût d'un bout de programme réside dans la maintenance et qu'au bout d'un certain temps, cette maintenance n'est en général plus faite par l'auteur original du code[7].

Scilab est un langage de calcul ; ces calculs s'appliquent à divers domaines : mécanique, optique, thermique, électronique, traitement du signal, biologie, géophysique, économie, … En général, les utilisateurs sont des spécialistes du domaine d'application (physiciens, biologistes, économistes, …) mais ne sont *pas* des informaticiens. Cette section a donc pour but de leur apporter quelques outils simples facilitant la programmation.

Pour plus de détails voir : ***Conseils de codage en C***.

Démarche projet

Un programme est un projet. On doit donc lui appliquer les outils classiques d'un projet. Cela ne signifie pas nécessairement qu'il faille produire une documentation — notons que les commentaires d'un programme (voir ci après) *sont* une documentation, et que certains outils permettent de les extraire —, mais simplement qu'il faut se poser des questions et y répondre avant de commencer.

Un programme a un but, un objectif : il doit répondre aux attentes d'un utilisateur, qui est souvent le programmeur lui-même. Cette identité utilisateur = programmeur évite les incompréhensions liée à la transmissison d'informations, mais peut mener à un manque de rigueur et donc à des difficultés.

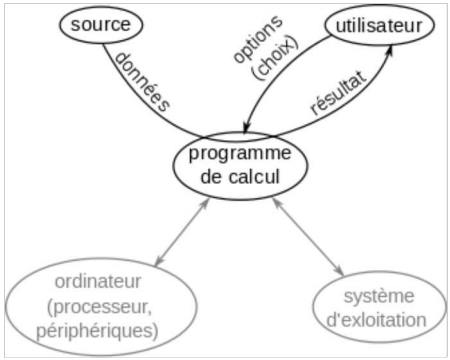
La première étape consiste donc à recenser (par écrit ou dans sa tête) les besoins et l'objectif à atteindre. Pour un projet de calcul, c'est essentiellement :

- quelles sont les données d'entrée ;
- quels sont les options possibles ;
- quel est le résultat attendu.

Cette étape doit inclure des données de test permettant de valider le programme : on a des données pour lesquelles on connaît le résultat, ce qui permettra de vérifier que le programme fonctionne bien.

Dans des cas limites — par exemple gros volumes de données, ou bien si le programme doit communiquer avec d'autres dispositifs (comme des capteurs) —, il est nécessaire de prendre en compte l'environnement du programme :

- ordinateur : par exemple parallélisation sur plusieurs processeurs, gestion des limitations de la mémoire ;
- système d'exploitation : par exemple lecture et écriture de fichiers, appel à des fonctions système.



Définition des fonctions (besoins de l'utilisateur) pour un programme de calcul à l'aide d'un diagramme APTE : le programme doit calculer des résultats à partir de données et avec des options fixées par l'utilisateur. Pour certaines applications, il faut prendre en compte l'environnement (ordinateur, système d'exploitation).

La deuxième étape consiste à rechercher les solutions. Cela peut être une recherche bibliographique : trouver un algorithme adapté, ou un programme déjà existant et faisant globalement la même chose. Cela peut aussi être le développement de solutions originales. À l'issue de cette étape, il faut choisir la solution appliquée.

Une méthode consiste à partir du besoin exprimé sous la forme de fonctions, une fonction étant un verbe à la forme active avec un sujet et un complément. Cette fonction peut être découpée en sous-fonctions, et l'on continue ce découpage jusqu'à arriver à des fonctions élémentaires faciles à coder (que l'on sait réaliser d'une ou plusieurs manières). Voir *Analyse fonctionnelle*.

Le choix de cette solution est critique car il conditionne tout le reste. Il ne peut être pertinent que si les objectifs sont clairs. Notons que l'on peut dans un premier temps retenir une solution peu performante mais facile à programmer, ce qui permet d'avoir rapidement un résultat et avec moins de risques d'erreur ; puis, par la suite, développer une autre solution plus performante, la première solution servant de référence. La programmation modulaire (fractionnement en fonctions) facilite cette démarche : il suffit d'écrire une nouvelle fonction (sous-programme) ayant le même nom que l'ancienne, et de renommer l'ancienne fonction pour se laisser la possibilité de revenir en arrière.

La troisième étape consiste à déterminer la structure du programme, son architecture. L'organisation des modules (fonctions) est obtenue à partir de l'analyse fonctionnelle. Il faut également choisir la structure des données (vecteur, matrice, liste, …).

Les modules doivent être *hiérarchisés*, ainsi :

- on sait quoi développer en premier, ce qui permet de fournir un programme partiel mais fonctionnel (délais courts, version de test, mode dégradé) ;
- on sait sur quels modules on doit concentrer ses moyens (temps passé en programmation et tests).

Typiquement, l'interface avec l'utilisateur peut être importante, en particulier si le programme doit être utilisé par quelqu'un « n'y connaissant rien », mais doit être développée à la fin, lorsque le programme fonctionne.

Rappelons-le, ces trois étapes ne sont pas nécessairement formalisées. Elles ne se font pas non plus nécessairement d'un bloc avant de commencer à programmer : les besoins exprimés peuvent changer en cours de projet, on peut découvrir de nouvelles solutions, … C'est donc un processus dynamique.

La quatrième étape consiste à faire du codage unitaire : on code chaque fonction, chaque module. On vérifie que chaque fonction remplit bien son rôle, avec des données de test unitaire.

La cinquième étape consiste à intégrer les fonctions, à les mettre dans un même programme. On vérifie que l'on transmet bien les bonnes données aux bonnes fonctions.

La sixième et dernière étape consiste à vérifier que le programme fini fonctionne bien, grâce au données de test établies à la première étape, et qu'il *remplit bien les besoins* de l'utilisateur (qu'il suit le cahier des charges).

Là encore, ces trois étapes se font souvent de manière dynamique voire simultanée : au sein du programme, on écrit une nouvelle fonction, on la teste (test unitaire et test d'intégration), puis on passe à la fonction suivante. Mais on n'oubliera pas la hiérarchisation : les premières fonctions développées ne sont pas les premières à être utilisées chronologiquement dans le programme, mais les fonctions les plus importantes (le moteur avant l'interface).

Dernière étape ? En fait non, il reste l'étape la plus longue et la plus difficile : la maintenance, la correction des erreurs qui se révèlent lors des utilisations (déverminage). Cela commence par le nettoyage du code : on retire les morceaux de code obsolètes (« bouts d'essai »).

La démarche globale est donc :

1. Définition des besoins de l'utilisateur (cahier des charges) : données d'entrée, option, résultat attendu. Choix de données de test.
2. Analyse fonctionnelle : découpage en fonctions. Recherche de solutions (sélection d'algorithmes).
3. Organisation des modules du programme. Hiérarchisation. Choix de la structure de données.
4. Codage et test unitaire des fonctions, en suivant l'ordre hiérarchique.
5. Intégration des fonctions au sein du programme et tests d'intégration (échange de données entre les fonctions)
6. Test du programme fini avec les données de test. Vérification que le programme répond bien aux besoins (cahier des charges).
7. Nettoyage du code. Maintenance.

Pour des projets importants et devant être formalisés, on pourra par exemple s'inétresser au modèle en V.

Structure générale du script

On recommande de mettre la plupart du code dans des fonctions ; le « programme principal », c'est-à-dire ce qui n'est pas dans les fonctions, devrait essentiellement se réduire à la définition des données de base et à l'appel des différentes fonctions. Un script a typiquement la forme suivante

```
Bloc de commentaires :
    nom du script
    versions du script, dates, auteurs
    version de Scilab utilisée, modules Atoms utilisés
    Objectif du script : fonctions qu'il rempli, son but, son rôle
    entrées et sorties : saisies clavier, affichages, fichiers, formats

Commandes préliminaires
    préparation de l'environnement
    p. ex. clear, clc, imports (input) …

Constantes
    définition de variables constantes

Fonctions
    définition des fonctions

Programme principal
    appel des fonctions
```

Par ailleurs, les fonctions ne devraient pas être trop longues, typiquement une page d'écran maximum

(environ 80 lignes) ; au-delà, il convient de scinder la fonction en plusieurs sous-fonctions. Elles ne devraient pas utiliser plus de 5 ou 10 variables locales[8] ; au-delà, il faut repenser la fonction, par exemple la scinder.

Le code doit être

- explicite ;
- aéré ;
- en alinéa ;
- abondamment commenté ;
- fractionné ;
- général.

Voir *Analyse statique de programmes*.

Du code explicite …

Il faut utiliser les noms de fonctions et de variable voulant dire quelque chose. On peut utiliser des noms « bidons » (comme `toto`, `foo`) pour des variables utilisées sur de petites portions de programme. On utilise aussi fréquemment des noms « standard » : `x` (abscisse, inconnue d'une équaiton, variable d'une fonction), `i` (nombre entier évolutif : indice, compteur), `n` (nombre entier constant : quantité), …

Si l'on travaille en équipe, il est recommandé d'utiliser une nomenclature systématique (voir *Convention de nommage*). D'un point de vue de la forme, on peut par exemple recommander de former les noms de variable à partir de mots ; chaque changement de mot est indiqué par une capitale, les autres lettres étant en minuscules (`maVariableAvecUnNomExplicite`). Concernant la composition du nom, on peut si nécessaire spécifier le type de variable (« matrice de réels », « vecteur ligne de chaînes de caractères », …) en début *ou* en fin de nom (mais toujours au même endroit), selon une convention commune (voir *Notation hongroise*).

… Aéré …

Il faut se limiter à une seule commande par ligne, voire plusieurs lignes pour une commande si la syntaxe est longue (utilisation de trois points `...`), la coupure se faisant après une virgule ou avant un opérateur. La longueur d'une ligne ne devrait pas dépasser 80 caractères (l'éditeur SciNotes matérialise d'ailleurs cette limite). Mais l'on peut à l'inverse envisager de mettre deux commandes courtes et s'articulant sur une même ligne.

Par ailleurs, on laisse un ligne vide au sein d'une fonction pour séparer les grands blocs, et on laisse deux lignes vides entre les fonctions.

… En alinéa (« indenté ») …

Des blancs en début de ligne marquent l'imbrication des commandes, en particulier à l'intérieurs des délimiteurs de fonctions (`function … endfunction`), de boucles (`for … end`, `while … end`) et des exécutions conditionelles (`if … then … else … end`). Par défaut, l'éditeur de Scilab (SciNotes) crée des alinéas de quatre espaces.

… Abondamment commenté …

Chaque fonction doit comprendre une introduction (commentaires) expliquant ce qu'elle fait, la nature des variables qui lui sont passées et la nature des variable en sortie — ce qu'elles signifient et le type qu'elles doivent avoir (précondition à l'exécution de la fonction, voir *Programmation par contrat). Les grandes*

étapes de la fonctions doivent être indiquées par des commentaires.

De manière générale, les commentaires doivent donner un aperçu du code, les informations nécessaire à l'utilisation du code ainsi que celle relatives à la lecture et à la compréhension du code, et les raisons du choix d'une solution si son choix n'est pas trivial. Par contre, on évite de commenter ce qui est évident dans le code, ainsi que ce qui risque de devenir obsolète lorsque le code évolue.

… Fractionné …

Comme signalé plus haut, le code doit être découpé en fonction de moins de 80 lignes. Le programmeur doit tant que faire se peut utiliser des variables locales. À l'exception de quelques variables très générales, essentiellement des constantes, toutes les variables utilisées par une fonction doivent être définies dans la fonction, ou bien transmises comme paramètre.

Outre la facilité de maintenance, le fait de mettre du code dans des fonctions accélère l'exécution d'un script. En effet, les fonctions sont compilée en code intermédiaire *(bytecode)* ; cette opération — automatique — de compilation consiste d'une part à vérifier que la syntaxe est correcte, et d'autre part réorganise les commandes en notation polonaise inverse, mieux adaptée à l'exécution (qui passe par une pile)[6].

Si certaines fonctions sont très générales, elle méritent de figurer dans un fichier `.sci` à part, appelé par la commande `exec`, voire dans une bibliothèque (voir ci-dessus).

… Et général

De manière plus spécifique, pour Scilab, on peut recommander de faire des fonctions les plus générales possibles. Par exemple, une fonction numérique devrait être capable de traiter un nombre, mais aussi une matrice ; on peut ainsi définir un ensemble de définition de type `X = 0:0.01:1;` et appliquer directement `f(X)`. Si l'on a une fonction de deux variables X et Y, on peut générer deux matrices « grille » et ainsi avoir une carte de résultats f(X, Y), avec la fonction `ndgrid()` (grille à *n* dimensions) :

```
X = vx1:pasx:vx2;
Y = vy1:pasy:vy2;

[X_matrice, Y_matrice] = ndgrid(X, Y);


f(X_matrice, Y_matrice)
```

On a ainsi

```
X_matrice = [X(1) X(1) … X(1)
             X(2) X(2) … X(2)
              …    …   …  …
             X(m) X(m) …  X(m)]
```

```
Y_matrice = [Y(1) Y(2) … Y(n)
             Y(1) Y(2) … Y(n)
              …    …   … Y(n)
             Y(1) Y(2) … Y(n)]
```

```
f(X_matrice, Y_matrice) = [f(X(1),Y(1)) f(X(1),Y(2)) … f(X(1),Y(n))
                           f(X(2),Y(1)) f(X(2),Y(2)) … f(X(2),Y(n))
```

```
                                    …          …       …       …
                    f(X(m),Y(1)) f(X(m),Y(2)) … f(X(m),Y(n))]
```

À titre d'exercice, on peut chercher comment soi-même cette fonction.

Génération de grilles

Considérons que l'on a deux vecteurs ligne X = [X(1), X(2), …] et Y = [Y(1), Y(2), …], obtenus avec la commande `linspace()` ou la syntaxe `v1:p:v2`. On peut utiliser le programme suivant :

```
taille_X = size(X, "c");
taille_Y = size(Y, "c");

X_matrice = X'*ones(taille_Y, 1);
Y_matrice = ones(1, taille_X)*Y;

f(X_matrice, Y_matrice)
```

S'il n'est pas possible de créer une fonction gérant les matrices (ou si l'on n'y arrive pas), on peut toujours utiliser « l'évaluation de groupe » `feval(X, f)` à la place de `f(X)`.

La manière la plus simple de créer une fonction pouvant agir sur une matrice consiste à faire défiler les indices, par exemple :

```
function [y]=carre(x)
    // calcule le carré des composantes d'une matrice
    // entrées : x : matrice de nombres
    // sorties : y : matrice de nombres
    [imax, jmax] = size(x);
    for i = 1:imax
        for j = 1:jmax
            y(i,j) = x(i,j)^2;
        end
    end
endfunction
```

ainsi, si l'on transmet un vecteur, `imax` ou `jmax` vaudra 1, et si l'on transmet un nombre unique (scalaire), les deux vaudront 1. Mais l'utilisation de boucle ralentit considérablement l'exécution du programme. On peut souvent utiliser le fait que Scilab est fait pour travailler avec des matrices, par exemple utiliser les versions « pointées » des opérateurs :

```
function [y]=carre(x)
    // calcule le carré des composantes d'une matrice
    // entrées : x : matrice de nombres
    // sorties : y : matrice de nombres
    y = x.^2;
endfunction
```

Rappelons de manière utile que l'inverse peut s'obtenir par `.^(-1)` ou par `1 ./` (bien respecter l'espace entre le 1 et le point), la racine carrée par `.^(0.5)` ; les racines n-ièmes par `.^(1/n)` ou `nthroot(x, n)`. D'autre part, la plupart des fonctions primitives s'appliquent aux matrices (par exemple `exp()`, `sqrt()`, …). Par ailleurs, nous avons vu précédemment qu'un calcul effectué sur une matrice est plus rapide qu'un calcul

effectué successivement sur chacun de ses éléments (en raison des opérations préliminaires au calcul). Pour une question de rapidité, il vaut donc mieux favoriser les calculs globaux sur des matrices que des calculs en faisant défiler les indices.

On peut utiliser de manière « intelligente » les raccourcis d'indice (`:`, `$`), ainsi que la possibilité d'extraire certains indice en utilisant une matrice booléenne. Dans l'exemple suivant[9], nous définissons une fonction ayant une formule différente à gauche et à droite d'une valeur x_0 ; il s'agit d'un pic dissymétrique, formé de deux demies gaussienne de même moyenne et même hauteur, mais de largeurs différentes.

```
function [y] = gauss_dissym(A, x)
    // génère un pic gaussien dissymétrique
    // entrées : A(1) : position du pic
    //     A(2) : hauteur de la courbe
    //     A(3) : largeur de la courbe à gauche
    //     A(4) : largeur de la courbe à droite
    //     x : vecteur de nombres
    // sorties : y : vecteur de nombres
    indice = (x < A(1)); // vecteur de %t à gauche, %f à droite
    y = zeros(x); // initialisation
    y(indice) = A(2)*exp(-(x(indice) - A(1)).^2/A(3)); // profil gauche
    y(~indice) = A(2)*exp(-(x(~indice) - A(1)).^2/A(4)); // profil droit
endfunction
```

On peut ainsi l'utiliser avec un nombre unique

```
A = [3, 2, 1.5, 2.5];
y = gauss_dissym(A, 2.5)
```

ou bien avec un vecteur

```
A = [3, 2, 1.5, 2.5];
X = 0:0.1:6;
plot(X, gauss_dissym(A, X))
```

De manière plus générale, considérons une fonction de deux variables, définie de manière différente à gauche et à droite pour x ; donc concrètement deux fonctions $f_g(x, y)$ et $f_d(x, y)$. Nous voulons l'évaluer sur deux vecteurs à partir desquels on crée deux matrices grille (voir plus haut). On peut utiliser la solution suivante :

```
function [z] = f(A, x, y)

indice = (x < A); // vecteur de %t à gauche, %f à droite
z = zeros(x, y); // initialisation
z = fg(x, y)*bool2s(indice)
z = z + fd(x, y)*bool2s(~indice)
```

Enfin, on peut transmettre une fonction en paramètre à une autre fonction. Dans l'exemple ci-dessous, nous appliquons la fonction `derivation` à deux fonctions différentes (il existe une fonction `derivative()` qui remplit parfaitement ce rôle ; le code suivant n'est donné qu'à titre d'illustration).

```
//=============================================================================
// nom : carre_inverse_derivee.sce
// auteur : Christophe Dang Ngoc Chan
```

```
// date de création : 2012-10-20
// dates de modification :
//   2013-07-08 : ajout de xtitle dans les tracés
//      guillemets ' -> "
//-------------------------------------------------------------------------
// version de Scilab : 5.3.1
// module Atoms requis : aucun
//-------------------------------------------------------------------------
// Objectif : Tracer les fonction carré et inverse, ainsi que leurs dérivées
// Entrées : aucune (paramètres en dur)
// Sorties : fenêtre graphique avec 4 tracés
//=========================================================================

// **********
// Initialisation
// **********

clear;
clf;


// **********
// Constantes
// **********

deltax = 1e-6; // précision pour la dérivation


// **********
// Fonctions
// **********

function [y]=carre(x)
    // calcule le carré des composantes d'une matrice
    // entrées : x : matrice de nombres
    // sorties : y : matrice de nombres
    y = x.^2;
endfunction


function [y]=inverse(x)
    // calcule l'inverse des composantes d'une matrice
    // entrées : x : matrice de nombres
    // sorties : y : matrice de nombres
    y = x.^(-1);
endfunction


function [yprime] = derivation (f, x, epsilon)
    // calcule la dérivée aprochée de la fonction f
    // entrées : f : fonction
    //     x : ensemble de définition (matrice de nombres)
    //     epsilon : nombre positif, normalement petit,
    //         mais pas trop pour éviter les erreurs de soupassement
    // sorties : yprime : matrice de nombres
    yprime = 1/(2*epsilon)*(f(x+epsilon) - f(x-epsilon));
endfunction


// **********
// Programme principal
// **********

// Génération des données

X = [1:0.1:10];
```

```
Y1 = carre(X);
Yprime1 = derivation(carre, X, deltax);

Y2 = inverse(X);
Yprime2 = derivation(inverse, X, deltax);

// Tracé

subplot(2,2,1)
plot(X, Y1, "b")
xtitle("$f(x) = x^2$", "x", "y")
subplot(2,2,3)
plot(X, Yprime1, "b")
xtitle("$f''(x)$", "x", "y")

subplot(2,2,2)
plot(X, Y2, "r")
xtitle("$y = g(x) = 1/x$", "x", "y")
subplot(2,2,4)
plot(X, Yprime2, "r")
xtitle("$y = g''(x)$", "x", "y")
```

Relecture de code

Une manière simple de s'assurer de la rigueur dans l'application des règles consiste à relire le code, voire à le faire relire par un tiers (voir *Revue de code*). La relecture de code consiste à s'assurer que les règles énoncées — découpage en fonctions de moins de 80 lignes et ayant moins de 10 variables locales, non recours à des variables globales, respect identations, commentation du code, … — sont bien vérifiées *et c'est tout* (mais c'est déjà beaucoup) !

En particulier, il serait illusoire, et contre-productif car décourageant, d'espérer comprendre *rapidement* un code qu'un auteur a mis plusieurs jours voire semaines à écrire. Par contre, on peut relever les endroits où l'auteur a été peu rigoureux — typiquement, des endroits où il y a eu beaucoup d'essais-erreurs, des modifications vites faites (correction de petites erreurs, petits ajouts).

Distribuer un exécutable

Un auteur peut vouloir distribuer un exécutable, par exemple pour avoir une utilisaiton transparente de Scilab, ou encore pour masquer un script afin que l'on ne puisse pas le lire, pour garder confidentiel un algorithme. On peut pour cela utiliser la fonction `SendScilabJob` au sein d'un programme C/C++. Cela fait appel à l'API `call_scilab`.

Au sein d'un programme C/C++, le code Scilab est alors mis dans une chaîne de caractères, puis envoyé au moteur Scilab par la commande `SendScilabJob`. Le programme C/C++ peut ensuite être compilé.

Le module Atoms `scetoexe` fournit une commande `scetoexe_generate()` permettant d'encapsuler un script Scilab dans un exécutable Microsoft Windows. Cependant, l'exécution fait appel à Scilab, il ne s'agit pas d'un programme indépendant.

Le module Atoms `Scilab 2 C` permet de convertir le script Scialb en C. Ce code peut ensuite être compilé.

La société Equalis distribue l'extension (payante) WildCruncher pour son logiciel (payant) Pro Plus, qui permet de générer du code C à partir du script Scilab puis de le compiler pour obtenir un exécutable [4] (http://www.equalis.com/?page=CoderAndCompiler).

Exemples

Voici quelques exemples de programmes complets mettant en œuvre des algorithmes simples.

Crible d'Ératosthène

Le programme suivant détermine la liste des nombres premiers inférieurs à une valeur donnée, appelée ici `maximum`, en utilisant l'algorithme du crible d'Ératosthène.

```
//===========================================================================
// nom : crible_eratosthene.sce
// auteur : Christophe Dang Ngoc Chan
// date de création : 2012-10-16
// dates de modification :
//   2013-07-08 : floor -> uint32 (pour diminuer la taille nécessaire)
//      guillemets ' -> "
//   2014-05-14 : uint8 (pour diminuer la taille nécessaire)
//---------------------------------------------------------------------------
// version de Scilab : 5.3.1
// module Atoms requis : aucun
//---------------------------------------------------------------------------
// Objectif : détermine la liste des nombres premiers inférieurs à un nombre
//   entier donné par le crible d'Ératosthène
// Entrées : nombre entier (scalaire)
// Sorties : vecteur de nombres entiers
//===========================================================================

// ***** fonctions et procédures *****

function [liste] = eratosthene(limite)
    // renvoit la liste (vecteur) des nombres premiers inférieurs à la limite
    indices = uint8(ones(limite, 1));
    // indices(i) = 1 si i est premier, 0 sinon
    imax = uint32(sqrt(limite));
    for i=2:imax
        if indices(i)=1 then
           jmax = uint32(limite/i);
           for j=2:jmax
               indices(i*j)=0; // élimination des multiples de i
           end
        end
    end
    // constitution de la liste
    j=1;
    for i = 1:limite
        if indices(i)=1 then
            liste(j)=uint32(i);
            j=j+1;
        end
    end
endfunction

// ***** programme principal *****

// accueil
disp("Crible d''Ératosthène")

// entrer le plus grand nombre que l'on veut tester
maximum=input("nombre entier maximum : ");

// exécuter l'algorithme
lst = eratosthene(maximum);

// afficher le résultat
disp(lst')
```

Algorithme d'Euclide

Le programme suivant permet de déterminer le PGCD et le PPCM de deux nombres entiers, en utilisant l'algorithme d'Euclide.

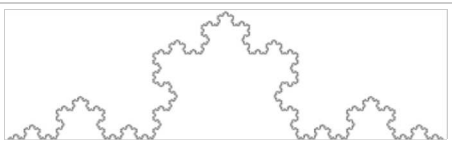
```
//===========================================================================
// nom : pgcd_ppcm_euclide.sce
// auteur : Christophe Dang Ngoc Chan
// date de création : 2012-10-16
// dates de modification :
//   2013-07-08 : guillemets ' -> "
//---------------------------------------------------------------------------
// version de Scilab : 5.3.1
// module Atoms requis : aucun
//---------------------------------------------------------------------------
// Objectif : détermine le PGCD et le PPCM par l'algorithme d'Euclide
// Entrées : deux nombre entier (scalaires)
// Sorties : deux nombres entiers (scalaires)
//===========================================================================

// ***** Fonctions et procédures *****

function [resultat] = euclide(a, b)
    // détermine le PGCD de a et de b (deux entiers)
    r = modulo(a,b); // initialisation du reste
    while r <> 0
        a = b;
        b = r;
        r = modulo(a,b); // calcul du reste
    end
    resultat = b;
endfunction

// ***** Programme principal *****

// accueil

disp("Calcul du PGCD et du PPCM de deux nombres par l''algorithme d''Euclide")

// valeurs

n1 = input("Premier nombre : ");
n2 = input("Second nombre : ");

// exécution de l'algorithme

pgcd = euclide(n1, n2);
ppcm = n1*n2/pgcd;

// affichage du résultat

print(%io(2),pgcd)
print(%io(2),ppcm)
```

Courbe de Koch

Le programme suivant trace la courbe du flocon de Koch de manière récursive.

```
//===========================================================================
// nom : von_koch.sce
```

```
// auteur : Christophe Dang Ngoc Chan
// date de création : 2012-10-23
// dates de modification :
//    2013-07-08 : guillemets ' -> "
//    2013-07-2 : vectorisation du calcul
//-----------------------------------------------------------
// version de Scilab : 5.3.1
// module Atoms requis : aucun
//-----------------------------------------------------------
// Objectif : trace la courbe du "flocon de neige" de von
// Entrées : aucun (paramètres codés en dur)
// Sorties : fenêtre graphique avec une courbe ; fichier
//===========================================================

clear;
clf;

// **************
// * constantes *
// **************

n = 6; // nombre d'étapes ;
// limité à 9 (262 145 points), sinon il faut changer la
// 6 etapes (4 097 points) sont suffisantes pour un bon r
sin_soixante = sqrt(3)/2; // sin(60°)
l = 1; // longueur du segment initial (unité arbitraire)

// ******************
// * initialisation *
// ******************



// *************
// * fonctions *
// *************

function []=vonkoch(A, B, i)
    u = (B - A)/3 ; // tiers du vecteur AB
    v = [0.5*u(1) - sin_soixante*u(2) ; sin_soixante*u(1)
    // vecteur tourné de +60°
    C = A + u ;
    D = C + v ;
    E = B - u ;
    if i == 1 then
        x = [A(1) ; C(1) ; D(1) ; E(1) ; B(1)] ;
        y = [A(2) ; C(2) ; D(2) ; E(2) ; B(2)] ;
        xpoly(x, y, "lines")
    else
        j = i - 1 ;
        vonkoch(A, C, j);
        vonkoch(C, D, j);
        vonkoch(D, E, j);
        vonkoch(E, B, j);
    end
endfunction

// ***********************
// * programme principal *
// ***********************

debut = [0;0] ;
fin = [l;0];

vonkoch(debut, fin, n)
```



Courbe de Koch

```
isoview(0, l, 0, l*sin_soixante/3)

// enregistrement du fichier

nom = "von_koch_"+string(n)+"_etapes.svg" ;
xs2svg(0, nom)
```

Lancer un programme sans ouvrir directement Scilab

Il est possible de lancer un programme Scilab sans ouvrir l'interface graphique Scilab. Cela se comporte comme s'il s'agissait d'un exécutable.

Pour cela, il faut exécuter la commande suivante dans la ligne de commande du système d'exploitation *(shell)* :

```
scilab -nw -f monprogramme.sce
```

Si le système d'exploitation ne trouve pas lui-même Scilab, il faut lui indiquer l'emplacement. Par exemple sous Unix

```
/bin/scilab -nw -f monprogramme.sce
```

ou bien sous Microsoft Windows :

```
"C:\Program Files\scilab-5.3.3\bin\Scilex.exe" -nw -f monprogramme.sce
```

Si l'on veut rediriger la sortie de Scilab vers la sortie standard (fenêtre système), on peut « tuber » Scilab, par exemple sous Unix

```
/bin/scilab -nwni < monprogramme.sce
echo "disp('Hello world')" |scilab -nwni
```

On peut mettre ces commandes de système d'exploitation dans un script.

On peut enfin appeler le moteur Scilab à partir d'un programme écrit dans un autre langage, voir les rubriques d'aide :

- `call_scilab` (http://help.scilab.org/docs/5.4.0/fr_FR/call_scilab.html) : C/C++ ;
- `javasci` (http://help.scilab.org/docs/5.4.0/fr_FR/javasci.html) : Java.

Voir aussi

Dans Wikipédia

- Algorithmique
- Structure de contrôle

Sur le Web

- (anglais) Code Conventions for the Scilab Programming Language (http://wiki.scilab.org

/Code%20Conventions%20for%20the%20Scilab%20Programming%20Language), Scilab Wiki
- (anglais) Interfacing C or Fortran programs with Scilab (http://www.snv.jussieu.fr/~wensgen/Doc/scilab-2.6/Intro-html/intro-html006.html), Scilab Group

Notes

1. ou bien `fig = gcf() ; fig.figure_id`
2. voir anglais Perl regular expressions quick start (http://perldoc.perl.org/perlrequick.html)
3. En typographie, le terme « espace » est au féminin lorsqu'il désigne un caractère « blanc ».
4. les secondes sont un nombre entier, mais en raison de la représentation utilisée, il est possible que Scilab affiche un nombre décimal, avec des 9 ou des 0 jusqu'à la cinquième décimale.
5. voir le chapitre « Optimisation » (transparents 16 et suivants) de Patrick Bousquet-Mélou, « Utilisation du cluster Antares (http://www.crihan.fr/calcul/tech/documentation-ibm-cluster-idataplex-antares/formations/Formation_Antares.pdf) » sur *CRIHAN*, novembre 2014
6. voir le anglais message de Serge Steer (INRIA) (http://mailinglists.scilab.org/Scilab-users-Regarding-simple-numerical-operations-result-display-tp4026274p4026307.html), 20 mars 2013 à 14h38, sur la discussion *Regarding simple numerical operations result display* de la liste de diffusion *Scilab users*.
7. Code Conventions for the Java Programming Language (http://www.oracle.com/technetwork/java/javase/documentation/codeconventions-139411.html#16712), Oracle
8. le cerveau humain est capable d'appréhender 5 à 9 objets en même temps, selon l'état de fatigue, de concentration, … voir anglais George Armitage Miller, « The magical number seven, plus or minus two: Some limits on our capacity for processing information », dans *Psychological Review*, vol. 63, n° 2, 1956, p. 81–97 [texte intégral (http://psychclassics.yorku.ca/Miller/)]
9. le code a été amélioré grâce à Stefan Du Rietz (http://mailinglists.scilab.org/Scilab-users-Conditional-assignments-tp4026572p4026573.html)

Graphiques et sons < ↑ > *Créer une interface graphique GUI*

Créer une interface graphique GUI

Table des matières — Index

---

7. Créer une interface graphique GUI

Jusqu'ici, nous avons interagit avec notre programme Scilab *via* la ligne de commande. Il est également possible d'utiliser une interface graphique utilisateur (UI : *user interface*, ou GUI : *graphical user interface*).

Rappelons-le : le fait de disposer d'une interface graphique est peut-être une fonctionnalité importante du script, notamment si celui-ci doit être utilisé par un « non-spécialiste », mais la priorité — en terme de délais et de moyens — est de faire un programme qui calcule juste ! On commence donc par réaliser un programme avec une interface peu attractive, et l'interface graphique est réalisée en dernier, lorsque tout le reste fonctionne.

Interface graphique préprogrammée

Scilab propose quelques fonctions permettant une interaction *via* une interface graphique utilisateur :

- `nom_chemin = uigetdir()` : ouvre une boîte de dialogue permettant de sélectionner un répertoire (dossier, chemin d'accès) ;
- `nom_fichier = uigetfile()` : ouvre une boîte de dialogue permettant de sélectionner un fichier ;
- `nom_fichier = uiputfile()` : ouvre une boîte de dialogue permettant de définir un nom fichier (en vue de le créer ou le modifier) ;
- `editvar nom_de_variable` : permet de modifier une variable.

Création d'une interface graphique utilisateur

Il est possible de créer soi-même une interface graphique utilisateur. Cela se fait avec la commande `uicontrol`.

Pointeur d'un objet

Chaque composant de l'interface graphique — *widget* : texte, bouton, curseur, liste déroulante, … — est désigné par un pointeur (*handle*, littéralement « poignée »). Le pointeur est une variable créée automatiquement par Scilab lors de la création du composant.

La fenêtre graphique (figure) elle-même est désignée par un pointeur. Le nom du pointeur peut être défini lors de la création de la fenêtre, par exemple :

- `f = figure(1)` : crée la fenêtre graphique n°1, la variable `f` pointant sur cette figure ;
- `f = scf(0)` : indique que la figure courante est la figure 0, la variable `f` pointant sur cette figure.

On peut aussi récupérer le pointeur de la fenêtre courante par la commande `f = gcf()` — raccourci de `f = get('current_figure')`.

Les objets de l'interface graphique ont eux aussi leur pointeur, dont le nom est défini à la création :

```
pointeur = uicontrol(...)
```

Le premier argument de la commande `uicontrol` est normalement le pointeur de la fenêtre graphique concernée. Les propriétés de l'objet peuvent être définies à la création dans la commande sous la forme d'un couple « `propriété, valeur` ». Elles peuvent aussi être définies *a posteriori* avec la syntaxe classique des pointeurs : « `pointeur.propriété = valeur` ».

Écrire dans une figure

Pour écrire un texte statique, on utilise la propriété `"style"` avec la valeur `"text"`.

Pour le texte en lui-même, on utilise la propriété `"string"`, la valeur étant le texte à insérer. On peut utiliser du code HTML pour la mise en forme.

Pour définir la taille et l'emplacement de la zone de texte, on utilise la propriété `"position"`, la valeur étant une vecteur contenant les coordonnées (*x*, *y*) du point en bas à gauche, et les dimensions (*l*, *h*) (largeur et hauteur) : `[x y l h]`.

Par exemple :

```
h = figure(0);
t = uicontrol(h, "style", "text", ...
    "string", "Ceci est une figure", ...
    "position", [50 70 100 100]);
```

Ce script crée une zone de texte, située en (50, 70), et ayant pour dimension 100×100, et contenant le texte « Ceci est une figure ».

On peut définir la fonte utilisée :

- police : propriété `"fontname"`, la valeur étant le nom de la police ;
- graisse : propriété `"fontweight"`, avec les valeurs `"light"`, `"normal"` (valeur par défaut), `"demi"` ou `"bold"` (gras) ;
- corps : propriété `"fontsize"`, avec la taille de corps en points (scalaire), ou en unité définie par `fontunit` ;
- unité pour le corps : propriété `"fontunit"`, avec les valeurs `"points"` (par défaut), `"pixel"` ou `"normalized"` ;
- inclinaison : propriété `"fontangle"`, avec les valeurs `"normal"` (par défaut), `"italic"` ou `"oblic"` ;
- alignement :
  - propriété `"HorizontalAlignment"`, avec les valeurs `"left"`, `"center"` (par défaut) ou `"right"`,
  - propriété `"VerticalAlignment"`, avec les valeurs `"top"`, `"middle"` (par défaut) ou `"bottom"`.

Par exemple :

```
h = gcf();
t = uicontrol(h, "style", "text", ...
    "string", "Ceci est <code>une</code> figure", ...
    "fontsize", 12, ...
    "fontweight", "bold", ...
    "fontangle", "italic", ...
```

```
    "position", [50 70 120 100]);
```

ou bien

```
h = gcf();
t = uicontrol(h, "style", "text", ...
    "string", "Ceci est <code>une</code> figure", ...
    "position", [50 70 120 100]);
set(t, "fontsize", 12);
set(t, "fontweight", "bold");
t.fontangle = "italic";
```

Créer un bouton

Les boutons sont créés avec la propriété `"style"` et la valeur `"pushbutton"`. C'est par ailleurs le style par défaut, on peut donc omettre de déclarer la propriété.

Les propriétés du bouton sont :

- `"string", "..."` : texte dans le bouton ;
- `"position", "[...]"` : position et taille du bouton ;
- `"callback", "code scilab"` : fonction de rappel (instructions à exécuter lorsque l'on clique).

Exemple :

```
b = uicontrol("style", "pushbutton",...
    "string", "$x^2$",...
    "callback", "X = [0:0.1:5]; plot(X, X.^2);");
```

Modifier les propriétés d'un élément

Les pointeurs permettent de définir les propriétés d'un élément graphique. Ainsi, une fonction de rappel *(callback)* peut modifier un autre objet.

Dans l'exemple ci-dessous, nous traçons une courbe en vert ; le pointeur de cette courbe est appelé `courbe`. Puis, nous créons des boutons pour modifier la couleur de la courbe, c'est-à-dire la propriété `courbe.foreground`.

Notons que nous travaillons avec des unités normalisées (propriétés `"units", "normalized"`) : dans ce système d'unités, la largeur et la hauteur de la fenêtre valent 1. Pour la fonction `xsetech()`, le point (0, 0) se trouve en haut à gauche, le point (1, 1) en bas à droite (sens de lecture latin). Pour la définition des autre objets, et notamment des boutons que l'on crée, le point (0, 0) est en bas à gauche et le point (1 ,1) en haut à droite (convention mathématique).

```
f = figure(0);
clf;

// la fenêtre de traçage occupe 90 % de la hauteur
// réserve 10 % de la fenêtre en bas pour les boutons
xsetech([0, 0, 1, 0.9])

// tracé de la courbe
X=[0:0.1:5];
```

```
plot(X, X.^2, "g"); // courbe verte

composant = gce();
courbe = composant.children;

// bouton changeant la couleur en noir
bnoir = uicontrol("style", "pushbutton",...
    "units", "normalized",...
    "position", [0, 0, 0.1, 0.05],...
    "BackgroundColor", [0,0,0],...
    "callback", "courbe.foreground=1");

// bouton changeant la couleur en rouge
brouge = uicontrol("style", "pushbutton",...
    "units", "normalized",...
    "position", [0.11, 0, 0.1, 0.05],...
    "BackgroundColor", [1,0,0],...
    "callback", "courbe.foreground=5");
```

Notons également que les couleurs sont définies de trois manières :

- par une « chaîne de caractères-raccourci », le paramètre `"g"` de la commande `plot()` ;
- par un vecteur RVB, dans la propriété `"BackgroundColor"` des boutons ;
- par le numéro de la couleur dans carte de couleurs *(colormap)*, pour la fonction de rappel `"callback"` des boutons.

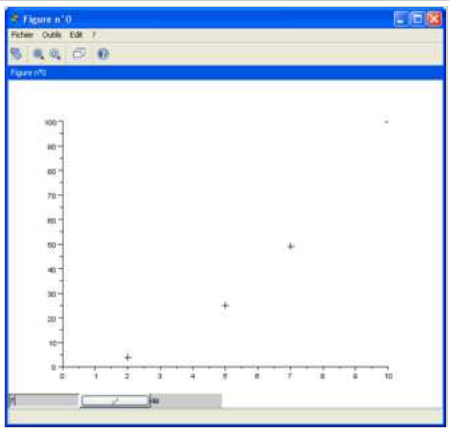
Le seul point ambigu est le dernier, puisque la carte de coleurs peut changer. Au sein d'un programme, il faut donc s'attacher à maîtriser la carte de couleurs (voir *Graphiques et sons > Cartographie couleur).*

Créer une zone d'édition

Une zone d'édition est créée avec la propriété `"style"` et la valeur `"edit"`. On peut indiquer le texte initial qu'elle contient avec la propriété `"string"`. C'est également par ce biais-là que l'on récupère sa valeur.

Exemple : calcul et tracé de la fonction carré

```
f = scf(0);
e = uicontrol(f, "style", "edit", ...
    "position", [0 0 100 20]);
t = uicontrol(f, "style", "text", ...
    "position", [200 0 100 20], ...
    "string", "...");
b = uicontrol(f, "style", "pushbutton", ...
    "string", "$x^2$",...
    "position", [100 0 100 20], ...
    "callback", "x = evstr(e.string);...
        y = x^2;...
        t.string = string(y);...
        plot2d(x, y, style = -1);");
```



Interface graphique utilisateur : calcul et tracé de la fonction carré.

Essayer l'exemple en prenant des valeurs successives sans interrompre le programme. Par exemple commencer par des valeurs extrêmes — `0` puis `10` — pour fixer l'échelle graphique, et « remplir » par des valeurs intermédiaires (`2`, `5`, `7`, …).

Analysons la fonction de rappel :

- `x = evstr(e.string)` : récupère la chaîne de caractères de la zone d'édition (pointeur `e`), la transforme en nombre (« évaluation » de la chaîne) et la place dans la variable `x` ;
- `y = x^2` : calcule le carré de `x` et place le résultat dans la variable `y` ;
- `t.string = string(y)` : transforme la valeur en chaîne de caractère, et place le résultat dans la chaîne de carcatères de la zone de texte (pointeur `t`) ; cela provoque l'affichage de la valeur ;
- `plot2d(x, y, style = -1)` : place le point sur le graphique.

Encapsulation dans une fonction

Si l'interface graphique est définie dans une fonction (`function … endfunction`), alors la fonction crée les composants de l'interface grapique *(wigets)* puis se termine. Lorsque l'on utilise les éléments de l'interface graphique, nous ne sommes donc plus dans l'environnement de la fonction, ce qui pose des problèmes ; en particulier, les pointeurs n'existent plus.

Pour gérer ces problèmes, on peut utiliser la propriété `userdata` pour stocker des données, puis y faire appel avec la variable `gcbo` — *get callback object*, c'est une variable crée automatiquement par Scilab lors de l'exécution da la fonction de rappel *callback*).

Par exemple, le petit programme suivant affiche « foo », et lorsque l'on clique sur le bouton `[OK]`, il affiche « 0 » à la place :

```
function []=test(h)
    texte = uicontrol(h, "style", "edit", ...
        "string", "foo",...
        "fontsize", 14, ...
        "position", [50 100 100 20]);
    bouton = uicontrol(h, "string", "OK",...
        "position", [50 50 50 20], ...
        "userdata", texte, ...
        "callback", "gcbo.userdata.string = string(0);");
endfunction

fig = figure(0);
test(fig);
```

Lorqque l'on définit le bouton, on enregistre donc le pointeur `texte` dans la propriété `userdata` du bouton. On peut ensuite faire appel à ce pointeur grâce à la variable `gcbo.userdata` : `gcbo.userdata.string` est l'équivalent de `texte.string`..

Exemple

Le script suivant reprend l'exemple du calcul du PGCD et du PPCM par l'algorithme d'Euclide. Comme nous avons voulu encapsuler la création de la fenêtre graphique dans une fonction (bonne pratique), il nous faut sauvegarder les pointeurs contenant les données dynamiques (qui varient lors de l'exécution). Comme il y a quatre pointeurs — deux variables d'entrée (n_1 et n_2), et deux variables de sortie (PGCD et PPCM) —, le champ `userdata` du bouton est un vecteur de quatre pointeurs : la définition du bouton d'exécution (composant dont le pointeur est `bouton`) contient

```
"userdata", [N1, N2, aff_pgcd, aff_ppcm]
```

avec :

- `N1` : pointeur vers la première zone d'édition, contenant la valeur n_1 ;
- `N2` : pointeur vers la seconde zone d'édition, contenant la valeur n_2 ;
- `aff_pgcd` : pointeur vers la zone de texte destinée à afficher PGCD(n_1, n_2) ;
- `aff_ppcm` : pointeur vers la zone de texte destinée à afficher PPCM(n_1, n_2).

La fonction d'appel du bouton contient :

- `n1 = eval(gcbo.userdata(1).string)` : récupération de n_1, car `gcbo.userdata(1)` est l'équivalent de `N1` ;
- `n2 = eval(gcbo.userdata(2).string)` : *idem* pour n_2 ;
- `gcbo.userdata(3).string = string(pgcd)` : écrit le contenu de la variable `pgcd` dans la chaîne du pointeur `gcbo.userdata(3)` qui n'est autre que `aff_pgcd` ;
- `gcbo.userdata(4).string = string(ppcm)` : *idem* avec le PPCM.

Ce qui nous donne le code source suivant.

```
//=============================================================================
// nom : pgcd_ppcm_euclide_gui.sce
// auteur : Christophe Dang Ngoc Chan
// date de création : 2013-01-02
// dates de modification : ...
//-----------------------------------------------------------------------------
// version de Scilab : 5.4.1
// module Atoms requis : aucun
//-----------------------------------------------------------------------------
// Objectif : détermine le PGCD et le PPCM par l'algorithme d'Euclide
// Entrées : deux nombre entier (scalaires)
// Sorties : deux nombres entiers (scalaires)
//=============================================================================

// ***** Fonctions et procédures *****

function [resultat]=euclide(a, b)
    r=modulo(a,b); // initialisation du reste
    while r<>0
        a = b;
        b = r;
        r=modulo(a,b); // calcul du reste
    end
    resultat = b;
endfunction

function []=fenetre_graphique(h)
    // titre de la boîte
    titre = uicontrol(h, "style", "text", ...
        "string", "PGCD et PPCM", ...
        "fontweight", "bold", ...
        "fontsize", 14, ...
        "position", [50 400 120 20]);
    // consignes
    texte_expl = uicontrol(h, "style", "text", ...
        "string", "Entrez deux nombres entiers et cliquez sur [OK]", ...
        "fontsize", 14, ...
        "position", [50 370 300 20]);
    // saisie du 1er nombre
    texte_b1 = uicontrol(h, "style", "text", ...
        "string", "N1 = ", ...
        "fontsize", 14, ...
        "position", [50 340 50 20]);
    N1 = uicontrol(h, "style", "edit", ...
```

```
        "string", "1",...
        "fontsize", 14, ...
        "position", [85 340 100 20]);
    // saisie du 2nd nombre
    texte_b2 = uicontrol(h, "style", "text", ...
        "string", "N2 = ", ...
        "fontsize", 14, ...
        "position", [50 310 50 20]);
    N2 = uicontrol(h, "style", "edit", ...
        "string", "1", ...
        "fontsize", 14, ...
        "position", [85 310 100 20]);
    // affichage du PGCD
    texte_res1 = uicontrol(h, "style", "text", ...
        "string", "PGCD = ", ...
        "fontsize", 14, ...
        "position", [50 250 50 40]);
    aff_pgcd = uicontrol(h, "style", "text", ...
        "string", "...", ...
        "fontsize", 14, ...
        "position", [110 250 50 40]);
    // affichage du PPCM
    texte_res2 = uicontrol(h, "style", "text", ...
        "string", "PPCM = ", ...
        "fontsize", 14, ...
        "position", [50 220 50 40]);
    aff_ppcm = uicontrol(h, "style", "text", ...
        "string", "...", ...
        "fontsize", 14, ...
        "position", [110 220 50 40]);
    // bouton d'exécution et fonction de rappel
    bouton = uicontrol(h, "string", "OK",...
        "position", [50 280 50 20], ...
        "userdata", [N1, N2, aff_pgcd, aff_ppcm], ...
        "callback", "n1 = eval(gcbo.userdata(1).string);...
            n2 = eval(gcbo.userdata(2).string);...
            pgcd = euclide(n1, n2);...
            ppcm = n1*n2/pgcd;...
            gcbo.userdata(3).string = string(pgcd);...
            gcbo.userdata(4).string = string(ppcm);");
endfunction

// ***** Programme principal *****

fig = figure(0); // création de la fenêre
clf;
fenetre_graphique(fig);
```

Voir aussi

Notes

Programmation < ↑ > *Environnement*

Environnement

Table des matières — Index

---

8. Environnement

Lorsque Scilab s'ouvre, il charge un certain nombre de fonctions internes et de variables prédéfinies. Ceci constitue l'environnement. Chaque fonction et variable que vous définissez s'ajoute à l'environnement.

Lorsque Scilab exécute une fonction, il ouvre un nouvel environnement qui est une copie conforme de l'environnement d'origine. La fonction peut ainsi utiliser toutes les variables et fonctions définies auparavant. Cependant, si l'on modifie la variable dans la fonction, cela ne les modifie que dans l'environnement de la fonction mais pas dans l'environnement initial, sauf si c'est expressément mentionné dans la fonction. Toutes les variables sont donc locales, même s'il n'est pas nécessaire de les transmettre en paramètre.

On peut interrompre l'exécution d'une fonction :

- par la combinaison de touches `[Ctrl]+[C]` ;
- en mettant la fonction `pause` dans le code de la fonction.

Dans ce cas-là, on reste dans l'environnement de la fonction. Le niveau d'environnement est indiqué dans l'invite de Scilab qui devient `-1->`.

On peut par exemple afficher les valeurs des variables dans cet environnement. Pour reprendre l'exécution de la fonction, il faut taper `resume` ou `return`. Pour interrompre définitivement l'exécution de la fonction et revenir à l'environnement précédent, il faut taper `quit` ; pour revenir à l'environnement initial, taper `abort`.

Voir aussi

Dans Wikipédia

- Environnement d'exécution
- Récursivité

Notes

Créer une interface graphique GUI < ↑ > *Structures de données avancées*

Structures de données avancées

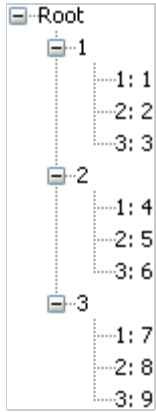
Table des matières — Index

9. Structures de données avancées

Listes

Une liste est un ensemble ordonné d'objets Scilab (nombre, chaîne de caractère, vecteur, matrice, liste, …). Les opérations standard sur les listes sont :

- `L = list(`a_1`, `a_2`, …, `a_n`)` : crée la liste avec les objets a_i et la met dans le vecteur L ;
- `L(0) =` *a* : insère *a* en début de liste (L(1) = *a*) et décale les autres éléments
- `L(`*i*`) =` *a* : remplace l'élément *i* par *a* ;
- `L($+1) =` *a* : place *a* en queue de liste ;
- `L(`*i*`) = null` : supprime l'élément *i* ;
- `size(L)` ou `length(L)` : taille de la liste ;
- `L3 = lstcat(L1, L2)` : concaténation des listes.



Affichage d'une liste sous la forme d'un arbre.

On peut décrire un arbre par des listes imbriquées. La commande `tree_show` permet d'afficher la liste sous forme arborescente, par exemple :

```
a = list(list(1, 2, 3), list(4,5,6),list(7,8,9));
tree_show(a);
```

Structures

Les structures sont des objets similaires aux vecteurs, mais dont les champs on un nom de type `structure.champ`. Elles sont crées par la commande `struct("`*champ1*`", `*valeur1*`, "`*champ2*`", `*valeur2*`, …)` :

```
-->aujourdhui = struct("jour", 23, "mois", "octobre", "annee", 2012)
 aujourdhui  =

   jour: 23
   mois: "octobre"
   annee: 2012

-->disp(aujourdhui.mois)

 octobre

-->aujourdhui('jour')
 ans  =

    23.

-->aujourdhui.mois = "novembre"
 aujourdhui  =
```

```
   jour: 23
   mois: "novembre"
   annee: 2012
```

On peut ajouter un champ à la volée :

```
-->aujourdhui.semaine = 47
 aujourdhui  =

   jour: 23
   mois: "novembre"
   annee: 2012
   semaine: 47
```

de fait, on peut créer une structure directement en assignant la valeur d'un champ :

```
-->demain.jour = 24
 demain  =

   jour: 24

-->type(demain)
 ans  =

    17.

-->typeof(demain)
 ans  =

 st
```

Notons que selon son contenu, la commande `type()` sur une structure retourne 16 (tlist) ou 17 (mlist).

On peut aussi créer une structure avec une liste typée :

```
hier = tlist(["nom", "jour", "mois", "annee"], 22, "octobre", 2012)
```

Dans le cas d'une liste typée, les appels suivants sont équivalents : `hier(2)`, `aujourdhui('jour')`,`hier.jour`. La syntaxe générale de `tlist` est :

```
tlist(type, a1, a2, …)
```

où

- *type* est un vecteur de chaînes de caractères dont
  - le premier élément est le nom que l'on donne au type ; c'est ce qui est retourné par la fonction `typeof`,
  - les éléments suivants sont les noms des champs ;
- *a1*, *a2*, … sont les valeurs des champs.

Si l'objet se comporte de la même manière qu'une structure, en revanche, la commande `typeof()` ne retourne donc pas `st`.

Enfin, on peut créer une matrice de structures :

```
-->foo(1, 1).nombre = %e
 foo  =

   nombre: list

-->foo(2, 2).texte = "e"
 foo  =

2x2 struct array with fields:
   nombre
   texte

-->foo(1, 1).nombre
 ans  =

    2.7182818

-->foo(1, 1).texte
 ans  =

     []

-->type(foo)
 ans  =

    17.

-->typeof(foo)
 ans  =

 st
```

Tableaux de cellules

On peut voir un tableau de cellules *(cell array)* comme une « hypermatrice » pouvant contenir des objets hétéroclites ; par hypermatrice, nous entendons un tenseur à *n* indices. Un tableau de cellules est créée par la commande `cell`. Par exemple

```
-->cell(4,3)
 ans  =


!{}  {}  {}  !
!            !
!{}  {}  {}  !
!            !
!{}  {}  {}  !
!            !
!{}  {}  {}  !
```

crée une matrice (tenseur à deux indices) 4×3. Cette matrice s'affiche encadrée de points d'exclamation. Les éléments vides sont représentés par des paires d'accolades `{}`.

Notons que `cell(4)` crée un matrice 4×4 ; pour créer une liste (tenseur à un indice, vecteur ligne ou colonne), il faut utiliser `cell(4,1)` ou `cell(1,4)`. La commande

```
cell(4, 3, 2)
```

crée un tenseur à trois indices 4×3×2, représenté comme deux couches de matrices 4×3 :

```
-->cell(4,3,2)
 ans  =


(:,:,1)

!{}  {}  {}  !
!            !
!{}  {}  {}  !
!            !
!{}  {}  {}  !
!            !
!{}  {}  {}  !
(:,:,2)

!{}  {}  {}  !
!            !
!{}  {}  {}  !
!            !
!{}  {}  {}  !
!            !
!{}  {}  {}  !
```

Les dimensions peuvent être indiquées par un vecteur de nombre. Par exemple

```
V = [4, 3, 2];
A = cell(V);
```

Pour accéder à un terme, c'est-à-dire lui assigner une valeur ou bien afficher cette valeur, il faut utiliser le pointeur `.entries`. Par exemple,

```
A(1, 1, 1).entries = 'vecteur';
A(1, 1, 2).entries = 1:5;
```

met la chaîne de caractères « vecteur » dans le terme A(1, 1, 1), et le vecteur [1, 2, 3, 4, 5] dans le terme terme A(1, 1, 2), ce qui donne

```
 A  =


(:,:,1)

!"vecteur"  {}  {}  !
!                   !
!{}         {}  {}  !
!                   !
!{}         {}  {}  !
!                   !
!{}         {}  {}  !
(:,:,2)

![1,2,3,4,5]  {}  {}  !
!                     !
!{}           {}  {}  !
!                     !
```

```
!{}           {}  {}  !
!                     !
!{}           {}  {}  !
```

Le pointeur `.dims` donne les dimensions du tableau de cellules.

```
-->A.dims
 ans  =

  4  3  2
```

Environnement < ↑ > *Gestion des fichiers*

Gestion des fichiers

Table des matières — Index

---

10. Gestion des fichiers

Nous avons vu précédemment que l'on pouvait définir une fonction ou un programme dans un fichier texte, puis appeler ce fichier texte avec l'instruction `exec()`, ce qui avait le même effet que taper la fonction ou le programme (voir *Programmation*).

Mais Scilab possède aussi d'autre possibilités de gestion de fichier.

Informations sur les fichiers

La commande `fileinfo('nom_de_fichier')` donne des informations sur les fichiers (au sens large d'Unix, c'est-à-dire y compris les unités de stockage et les répertoires)[1]. Elle renvoit un vecteur de 13 nombres :

1. Taille du fichier, en nombre de blocs (un bloc = 512 octets).
2. Mode du fichier : autorisations d'accès actuel (en lecture, écriture, modification), sous la forme d'un nombre entier.
3. L'identifiant du propriétaire du fichier.
4. L'identifiant du groupe du fichier.
5. Le numéro de lecteur.
6. La date de la dernière modification des données.
7. La date de la dernière modification de status.
8. La date de la dernier accès.
9. Le type de lecteur, si le fichier est un périphérique (fichier spécial).
10. La taille de blocs utilsée par le système d'exploitation pour les entrées-sortie.
11. Le nombre de blocs alloués.
12. Le numéro de nœud d'index *(i-number, inode number)*.
13. Le nombre de liens matériels (noms de fichiers différents pointant sur le même fichier).

Si le fichier n'existe pas, tous les nombres sont des NaN *(not a number)* ; la fonction retourne donc la matrice vide.

Par exemple, pour vérifier si un fichier existe (ou si un répertoire existe, ou si un lecteur est monté), on peut utiliser

```
fileinfo("nom_de_fichier")~= [ ]
```

Manipulation des répertoires

Scilab dispose d'un certain nombre de commandes de type Unix permettant de manipuler les chemins d'accès aux fichiers, les répertoires (dossiers) :

- `pwd` *(print working directory)* : affiche le répertoire courant ;
- `chdir(`*`chemin`*`)`, `cd` *`chemin`* *(change directory)* : change le répertoire courant ;

- `mkdir(chemin)` *(make directory)*, `createdir(chemin)` : crée un répertoire ;
- `ls` *(list)*, `listfile` : affiche la liste des fichiers dans le répertoire courant ;
- `dir` *(directory)* : *idem*, mais avec un affichage différent.

Par ailleurs, il existe des variables prédéfinies relatives aux répertoires :

- `home` : répertoire où les fichiers sont lu ou enregistrés par défaut ;
- `SCIHOME` : répertoire où sont les fichiers de configuration personnalisés ;
- `WSCI` : répertoire d'installation de Scilab.

Lecture/écriture d'une matrice en format texte

Les fonctions `read)` (http://help.scilab.org/docs/5.3.0/fr_FR/read.html) et `write()` (http://help.scilab.org/docs/5.3.0/fr_FR/write.html) permettent de lire et d'écrire les valeurs d'une matrice ou d'un vecteur dans un fichier au format texte. Cela permet entre autres d'échanger les données avec d'autres logiciels, comme par exemple un tableur.

Il est possible d'appliquer un format aux données dans un format proche du Fortran.

Syntaxe :

```
    write (fichier, a [, format])
[x]=read (fichier ,m ,n [, format])
```

Paramètres :

- `fichier` : Chaîne de caractère correspondant au nom du fichier.
- `a` : matrice, vecteur ou chaîne de caractères ;
- `m` : entier, nombre de ligne à lire ; le paramètre `m=-1` permet de lire toutes les lignes du fichier ;
- `n` : entier, nombre de colonnes à lire ;
- `format` : chaîne de caractère décrivant le format des données selon la syntaxe Fortran.

La commande `csvRead()` permet de lire des fichiers au format CSV. Cela permet par exemple de lire des fichiers exportés depuis un tableur, comme Microsoft Office Excel ou bien LibreOffice Calc. De même, la commande `csvWrite()` permet de créer des fichiers CSV, avec par défaut une virgule *(comma)* comme séparateur de champ ; le format de la commande est

```
csvWrite(A, fichier)
```

Exemples

```
x=linspace(0, 2*%pi, 100)';
y=sin(x);
write("sinus.txt", [x y], "(F6.3,1X,F6.3)");
```

Cet exemple écrit la table de la fonction sinus dans le fichier `sinus.txt`, chaque ligne représente un point.

```
sinus.txt:
 0.000  0.000
```

```
 0.063  0.063
 0.127  0.127
 0.190  0.189
 0.254  0.251
...
```

Si le format est omis, Scilab décide d'un format approprié. La commande :

```
write('sinus.txt', [x y]);
```

écrira le fichier suivant :

```
sinus.txt:
   0.0000000000000000         0.0000000000000000
  6.34665182543392575E-002  6.34239196565645064E-002
  0.12693303650867852       0.12659245357374926
  0.19039955476301776       0.18925124436041019
  0.25386607301735703       0.25114798718107922
  0.31733259127169627       0.31203344569848712
...
```

Inversement le fichier `temperature.txt` contient l'heure, la temperature et l'humidité relevé.

```
temperature.txt:
 18    -1 86
 17    -1 84
 16  -0.7 84
```

Sa lecture se fera par la commande `read()` :

```
a=read ("temperature.txt", -1, 3);
```

Il est plus simple ici de laisser scilab décider du format de lecture. Seul le nombre de colonnes est indiqué, le nombre de ligne est mis a -1 pour indiquer à scilab de lire l'intégralité du fichier. La matrice a de type flottant contient :

```
 a  =

    18.   - 1.     86.
    17.   - 1.     84.
    16.   - 0.7    84.
```

Syntaxe Fortran

- A*n* : chaîne de caractères de taille *n* ; des blancs sont éventuellement ajoutés à gauche ; si *n* n'est pas précisé, Scilab écrit simplement la chaîne de caractères sans blanc ;
- E*m*.*n* : notation exponentielle (réel à virgule flottante) de *n* caractères (y compris les signes +/-, le séparateur décimal « . » et le symbole de l'exposant « e »), dont *m* chiffres après la virgule (donc *m* + 1 chiffres significatifs) ; on doit avoir $n \geq m + 7$;
- F*m*.*n* : notation décimale (réel à virgule flottante) de *n* caractères (y compris le signe +/- et le séparateur

décimal « . »), dont *m* chiffres après la virgule ;
- `In` : nombre entier *(integer)* de *n* chiffres ; des blancs sont éventuellement ajoutés à gauche ;
- `Tc` : tabulation ;
- *n*`X` : place *n* espaces horizontaux.

Lecture d'un fichier Microsoft Office Excel

Scilab peut lire des fichiers Excel. La version 5.4 de Scilab peut lire les fichier de Microsoft Office Excel 2003 (format BIFF8), portant l'extension `.xls`, mais pas les fichiers `.xlsx`.

La première étape consiste à ouvrir le fichier Excel, avec la commande `xls_open()` ; puis, on lit les données avec la commande `xls_read()`.

Prétraitement d'un fichier

Dans certains cas, le fichier contient des données inutiles qu'il convient d'éliminer avant de l'ouvrir dans Scilab. Rappelons que des commandes du système d'exploitation peuvent être lancées depuis Scilab grâce à la commande `unix_g()`.

Par exemple, avec un système unix, il est possible de nettoyer un fichier texte en exécutant des commandes de type `sed`, `awk` et `grep`.

> *Pour plus de détails voir :* ***Programmation Bash/Commandes shell***.

Sauvegarde de variables en binaire

Il est possible d'exporter le résultat d'un calcul dans un fichier, ce qui permet de le réutiliser pour une autre fois. Ceci se fait avec la commande `save` :

```
save (fichier, a, b, c, ...)
```

Cela enregistre les variables `a`, `b`, `c`, ..., dans le fichier de nom `fichier`. Si aucune variable n'est spécifiée, toutes les variables de l'environnement de travail sont enregistrées.

Ce résultat peut ensuite être réimporté avec la commande `load` :

```
load (fichier, a, b, c, ...)
```

Cela lit les variables `a`, `b`, `c`, …, présentes dans le fichier de nom `fichier`. Il s'agit du nom des variables tel qu'il a été spécifié lors de la commande `save`. De même que précédemment, si aucun variable n'est spécifiée, toutes les variables présentes dans le fichier sont chargées.

Le fichier produit est un fichier binaire. Chaque variable est représentée par :

- le nom de la variable, sur 24 octets :
  - chaque caractère correspond à un code : 10 à 35 pour les minuscules de « a » à « z », -35 à -10 pour les capitales de « Z » à « A » ; taper `str2code('`*`chaine`*`')` pour avoir la suite de codes correspondant à la chaîne *chaine* ;
  - le nom se termine par le code 40 ;

- un code représentant le type de la variable : de 1 pour une matrice de réels en virgule flottante, 2 pour un polynôme, 10 pour un chaîne de caractères… taper `type(X)` pour avoir le code du type de la variable *X* ;
- puis suit la description de la variable, notamment le nombre d'éléments pour les matrices, chaînes… cette description dépend du type de la variable ;
- enfin vient la valeur de la variable.

L'instruction `listvarinfile('nom_de_fichier')` permet d'avoir la liste des variable contenues dans un fichier et leur description.

Enregistrer un graphique

Pour la création de graphiques, voir *Graphiques et sons*.

Scilab permet de sauvegarder des graphiques sous plusieurs formats. Il faut au préalable donner un numéro à la fenêtre graphique, en indiquant `scf(numéro)` avant les instructions de tracé.

La syntaxe générale des fonctions d'export est la suivante :

```
xs2format (id, fichier)
```

où

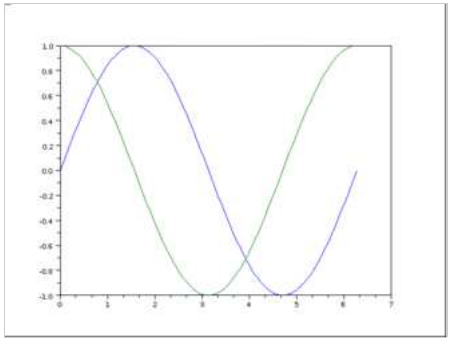
- `id` : entier, l'identifiant du graphique à exporter. Cet identifiant peut être obtenu par la fonction `gcf` ;
- `fichier` : chaîne de caractère, le nom du fichier, sans oublier l'extension, dans lequel le graphique sera enregistré.

Les fonctions sont :

- `xs2bmp` : format BMP (*bitmap*, Microsoft Windows);
- `xs2emf` : format EMF (*enhanced metafile*, Microsoft Windows) ;
- `xs2eps` : format EPS *(encapsulated PostScript)* ;
- `xs2fig` : format Xfig ;
- `xs2gif` : format GIF *(Graphics Interchange Format)* ;
- `xs2ppm` : format PPM *(portable pixmap)* ;
- `xs2ps` : format PS *(PostScript)*.

Exemple :

```
f = gcf ();
x = linspace (0, 2*%pi, 200)';
plot (x, [sin(x), cos(x)]);
xs2gif (f, "exemple_scilab_sin_cos.gif");
```



Courbe sinus et cosinus.

L'export de graphiques sous scilab semble se faire par copie d'écran, cela signifie qu'il peut y avoir des problèmes si lors de la commande `xs2`…, la fenêtre contenant le graphique n'est pas clairement visible à l'écran.

Lire un fichier d'image

Scilab ne dispose pas en natif d'instruction permettant de lire un fichier d'image. Il existe toutefois des

modules externes de traitement d'image que l'on peut charger grâce au gestionnaire de modules Atoms[2]. Citons :

- SIP (Scilab Image Processing toolbox)[3] ;
- SIVP (Scilab Image and Video Processing toolbox)[4] ;
- IPD (Image Processing Design toolbox)[5].

Avec SIP et SIVP[6][7], on charge une image dans une variable avec l'instruction

```
image = imread("nom_de_ficher")
```

et on l'affiche avec l'instruction

```
imshow(image)
```

Avec IPD[8], on charge une image dans une variable avec l'instruction

```
image = ReadImage("nom_de_fichier")
```

et on l'affiche avec l'instruction

```
ShowImage(image)
```

Enregistrer et lire des fichiers de sons

Lorsque la variable *x* est un vecteur, on peut le sauvegarder sous la forme d'un son, dans un fichier de type `.wav`. Le son correspond à des varations de pression de l'air, ou des variations de la tension d'alimentation du haut-parleur, *s* de la forme

$$s(t_i) = x(i)$$

avec

$$t_i = i/f$$

f étant la fréquence d'échantillonnage. Par défaut, f vaut 22 050 hertz (une valeur de *x* correspond à 1/22 050 seconde).

Pour plus de détails voir : ***Découvrir Scilab/Graphiques et sons#Sons***.

La création du fichier se fait avec la fonction `savewave`

```
savewave('nom_de_fichier', x)
```

Il faut bien s'assurer qu'il s'agit d'une matrice colonne.

On peut aussi définir une autre fréquence d'échantillonnage entière :

```
savewave('nom_de_fichier', x, f)
```

Un fichier `.wav` peut être mis dans une matrice avec la fonction `loadwave` :

```
x = loadwave('nom_de_fichier')
```

chaque ligne de la matrice correspondant alors à un canal. On peut ensuite jouer ce son dans Scilab avec l'instruction `playsnd()`

```
playsnd(x)
```

Voir aussi Graphiques et sons.

Gestion avancée des fichiers

Lecture/Ecriture « à la sauce » C

Différentes fonctions sont directement inspirées de la syntaxe du langage C.

- ```
  [fd, err] = mopen (fichier, mode)
  ```

  Ouverture du fichier de nom `fichier`. Le mode d'ouverture est `"r"` pour lecture, `"w"` pour écriture et `"a"` pour ajout.
  La fonction retourne un scalaire « descripteur » `fd` qui sert à identifier le fichier dans les fonctions suivantes. Le code d'erreur peut être récupéré par la variable `err`, il vaut 0 si l'ouverture s'est déroulée correctement.

- ```
  mfprintf (fd, format [, x1 [, x2, …]])
  ```

  Écriture dans le fichier d'identifiant `fd`. Le paramètre `format` est une chaîne de caractère décrivant le formatage des données. Cette fonction est équivalente à la fonction `fprintf` du langage C (voir *Programmation C-C++/Entrée / sortie de base#La fonction printf*).

- ```
  y(1,1:3) = mfscanf(fd, '%f %f %f\n')
  ```

  balaie *(scan)* le fichier numéro `fd` de manière séquentielle, et récupère les trois premières données et les met dans la première ligne d'une matrice de trois colonne appelée *y* ;

  ```
  y(2,1:3) = mfscanf(fd, '%f %f %f\n')
  ```

  continue l'opération de rapatriement et met les données dans la deuxième ligne de la matrice ;

- ```
  mclose (fd)
  ```

Ferme le fichier d'identifiant `fd`

Exploitation de fichiers texte

Le format texte (ASCII ou Unicode), du fait de son caractère universel, sert fréquemment de format d'échange entre logiciels. Il est donc intéressant de se focaliser sur les spécificités de ce format. Les fonctions `read()` et `csvRead()` sont simples d'utilisation, mais ne sont plus applicables pour les fichiers très grands.

Lire une matrice de nombres

La commande `fscanfMat` permet de lire directement une matrice dans un fichier texte (ASCII ou UTF-8). Dans sa syntaxe de base

```
M = fscanfMat('nomfichier');
```

la fonction crée une matrice de nombres M à partir du fichier texte `nomfichier`, en éliminant l'en-tête (les premières lignes qui contiennent des caractères non numériques). On peut lui demander de mettre le texte de l'en-tête dans une variable :

```
[M, entete] = fscanfMat('nomfichier');
```

Il est possible d'indiquer le format des données :

```
M = fscanfMat('nomfichier', 'format');
```

ou *format* est une chaîne de caractère qui peut être de la forme `%type`, par exemple :

- `%e` : réel en notation exponentielle ;
- `%f` : réel en notation décimale (point flottant) ;
- `%g` : réel en notation exponentielle si l'exposant est inférieur à -4, notation décimale sinon ;
- `%lg` : réel long (double précision) ;
- `%d` ou `%i` : nombre entier ;
- `%x` : entier hexadécimal.

La chaîne peut également contenir la largeur et la précision du nombre : `%largeur.précisionf` par exemple. La largeur est le nombre minimum de caractères que comporte un nombre. La signification de la précision dépend du type :

- pour un entier (`%d`, `%i`, `%x`) : nombre minimum de chiffre ;
- pour un réel en notation exponentielle ou décimale (`%e`, `%f`) : nombre de chiffres après la ivrgule ;
- pour un réel en notation « adaptative » (`%g`) : nombre de chiffres significatifs.

Lire les lignes une par une

La fonction `mgetl` lit une ou plusieurs lignes d'un fichier texte

```
txt = mgetl ('nomfichier', m)
```

```
txt = mgetl (fd, m)
```

Si le fichier a été ouvert (avec `mopen`, il dispose d'un descripteur `fd` qui permet d'y faire référence ; sinon, on peut utiliser le nom du fichier. Si *m* est omis ou vaut -1, la fonction lit toutes les lignes. Si *m* est un entier strictement positif, la fonction lit *m* lignes (accès séquentiel). La variable `txt` est une matrice colonne, chaque ligne correspondant à une ligne du fichier.

La commande `stripblanks(txt)` enlève les espaces situés au début et à la fin des lignes.

Découper une ligne en colonnes

Si l'on connaît les positions (index) des débuts de colonne, les nombres c1, c2, …, cn, on peut utiliser la commande `part()` pour extraire une colonne. Par exemple, pour extraite la colonne 5 :

```
p = part(txt, [c5, c6 - 1])
```

On peut également séparer les différentes colonnes pour les mettre dans un vecteur avec

```
cellules = strsplit(txt, [c1, c2, …, cn])
```

Une ligne — une chaîne de caractères — contient souvent des délimiteurs. S'il s'agit d'une chaîne de caractères fixes, on peut utiliser la commande `tokens()` :

```
t = tokens('a = 5', '=')
```

crée la matrice `['a ' ; '=' ; ' 5']`.

Si les séparateurs sont variables, on a intérêt à faire appel aux expressions régulières (voir *Programmation* > *Expressions régulières*). Par exemple, si les colonnes sont complétées par des caractères d'espacement (espaces, tabulations) en nombre indéterminé, on peut utiliser

```
cellules = strsplit(txt, '/\s+/)
```

Considérons par exemple le fichier `roisdefrance.txt`

```
Clovis Ier      466-511
Clodomir        495-524
Thierry Ier     485-534
Théodebert Ier  504-548
```

La séparation entre le nom du roi et la date est constituée de deux espaces ou plus. On veut créer une matrice dont la première colonne contient le nom du roi, la deuxième sa date de naissance et la troisième sa date de mort. On peut par exemple écrire

```
// lecture du fichier
fd = mopen('roisdefrance.txt');
texte = mgetl(fd, -1); // texte(i) contient la i-ème ligne
mclose(fd);
```

```
n = size(texte, 'r');

cellules = [];

// séparation des colonnes
for i = 1:n
    cellules(i,:) = strsplit(texte(i), '/\s{2,}|-/')'
end

disp(cellules');
```

Fonctions de gestion des fichiers

- listfiles (http://help.scilab.org/docs/5.3.0/fr_FR/listfiles.html) : Equivalent de ls sous unix

Notes et références

1. c'est une interface de la fonction C `stat()`
2. http://atoms.scilab.org/categories/Image_Processing
3. http://atoms.scilab.org/toolboxes/SIP
   http://siptoolbox.sourceforge.net/
4. http://atoms.scilab.org/toolboxes/SIVP
   http://sivp.sourceforge.net/
5. http://atoms.scilab.org/toolboxes/IPD
6. http://siptoolbox.sourceforge.net/doc/sip-0.7.0-reference/
7. [anglais Documentation SIVP (http://sivp.sourceforge.net/sivp-doc-2006.en.pdf)]
8. [anglais Didacticiel IPD (http://atoms.scilab.org/toolboxes/IPD/8.3.1/files/Tutorial.pdf)]

Voir aussi

Dans Wikipédia

- Encapsulated PostScript
- Fichier binaire
- Graphics Interchange Format
- Portable pixmap (en anglais)
- PostScript
- WAVEform audio format
- Windows bitmap (BMP)
- Windows Metafile (pour EMF)
- Xfig

Structures de données avancées < ↑ > *Équivalences Matlab-Scilab*

Équivalences Matlab-Scilab

Table des matières — Index

---

11. Équivalences Matlab-Scilab

Fonctions

Matlab	Scilab
`cputime`	`timer`
`nullspace`	`kernel`
`spdiags`	`sparse`

Constantes

	Matlab	Scilab
∞ (infini)	`Inf`	`%inf`
π ≃ 3,14…	`pi`	`%pi`
i (imaginaire pur)	`i, j`	`%i`
NaN *(not a number)*	`NaN`	`%nan`
précision relative de la représentation des nombres[1]	`eps`	`%eps`

Notes

1. 2^{-m} pour une mantisse à *m* bits ; 2^{-52} dans le cas des réel à double précision ;
   dans le cas de Matlab, `eps()` est une fonction qui donne la précision relative pour une représentation en simple précision (`eps('single')`) ou en double précision (`eps('double')`, ou `eps` simplement) ;
   dans le cas de Scilab, `%eps` est une constante

Voir aussi

- **`[pdf]`** (en) Eike Rietsch, *An Introduction to Scilab from a Matlab User's Point of View* (http://wiki.scilab.org/Tutorials?action=AttachFile&do=view&target=Scilab4Matlab.pdf), version 5.2, 2010

Gestion des fichiers < ↑ > *Xcos*

Xcos

Table des matières — Index

---

12. Xcos

Xcos est un module permettant de travailler sur les schémas-bloc (ou schémas fonctionnels), par exemple pour résoudre des problèmes d'automatisme ou de traitement du signal.

Prise en main

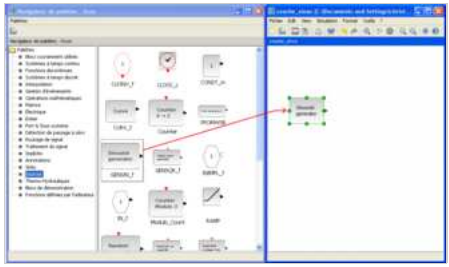
Pour ouvrir l'interface graphique de Xcos, il suffit de cliquer sur le bouton **Xcos** (représentant un écran avec une courbe), ou bien de taper la commande `xcos`.

L'interface comporte :

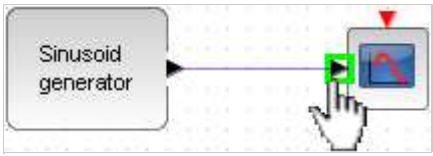
- l'espace de travail : c'est une fenêtre comportant un quadrillage ;
- un navigateur de palette : cette fenêtre propose des icônes.

Commençons par afficher une courbe sinusoïdale.

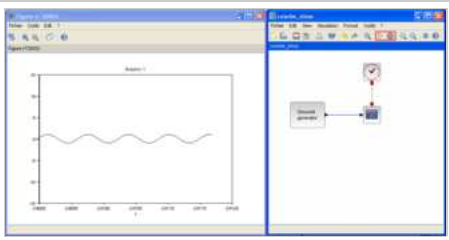
1. Dans le navigateur de palette, sélectionner la palette **Source** ; cliquer sur l'icône *Sinusoid generator* `GENSIN_f` (générateur de signal sinusoïdal), et faire un glisser-lâcher dans l'espace de travail.
2. Dans le navigateur de palette, sélectionner la palette **Sinks** (litt. puits) ; cliquer sur l'icône `CSCOPE` (oscilloscope), et faire un glisser-lâcher dans l'espace de travail.
3. Dans l'espace de travail, relier la sortie du générateur à l'entrée noir de l'oscilloscope :
   - pointer le curseur de la souris sur la sortie du générateur (triangle noir) ; un petit carré vert apparaît ;
   - cliquer et maintenir le bouton enfoncé ;
   - faire glisser le pointeur de la souris — cela tire un trait — sur le triangle d'entrée noir de l'oscilloscope ; un petit carré vert apparaît ;
   - lâcher le bouton de la souris ; les deux éléments sont reliés.
4. Dans le navigateur de palette, sélectionner la palette **Source** ; cliquer sur l'icône `CLOCK_c` (horloge), et faire un glisser-lâcher dans l'espace de travail.
5. Relier la sortie de l'horloge sur l'entrée rouge de l'oscilloscope.
6. Enregistrer le projet (avec le bouton **Enregistrer** ou bien le menu **Fichier | Enregistrer**).
7. Cliquer sur le bouton **Démarrer** : cela ouvre une fenêtre graphique qui affiche une sinusoïde. Cliquer sur le bouton **Arrêter**.



Ajout d'un générateur de signal par glisser-lâcher.



Création d'une liaison.

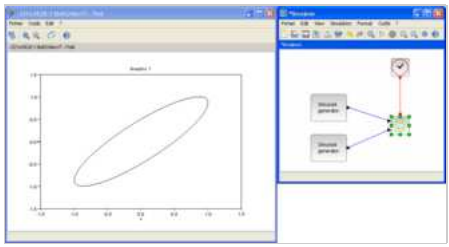


Tracer la courbe : mise en évidence des boutons **Démarrer/Arrêter**.

Pour modifier les paramètres de la source (amplitude, période, phase

à l'origine), de l'oscilloscope ou de l'horloge, il suffit de cliquer sur l'icône avec le bouton secondaire (« clic droit » pour les souris configurées pour droitiers) et de sélectionner **Paramètres du bloc**. On peut aussi utiliser la combinaison de touches `[Ctrl]+[B]`.

Nous allons maintenant afficher une courbe de Lissajous. Pour cela, on supprime l'oscilloscope et on le remplace par un `CSCOPXY` — afficheur fixe XY, un oscilloscope à double entrée — et l'on ajoute un deuxième générateur de signal sinusoïdal. Chaque générateur est relié à une entrée noire de l'afficheur, et l'horloge est reliée à l'entrée rouge.


Ellipse de Lissajous.

On utilise par exemple les paramètres suivants :

- premier générateur : amplitude = `1`, fréquence = `2*%pi`, phase = `0` ;
- second générateur : amplitude = `1`, fréquence = `2*%pi`, phase = `%pi/6` ;
- horloge : période = `0.01` ;
- afficheur : color or mark = `1`, Xmin = `-1.5`, Xmax = `1.5`, Ymin = `-1.5`, Ymax = `1.5`.

On obtient une ellipse. En faisant varier les paramètres des générateurs, on modifie la courbe. On peut par exemple mettre la période du second générateur à `%pi`.

On peut remplacer l'afficheur fixe par un afficheur d'animation `CANIMXY`.

Voir aussi

Notes

Équivalences Matlab-Scilab < ↑ > *Atoms*

Atoms

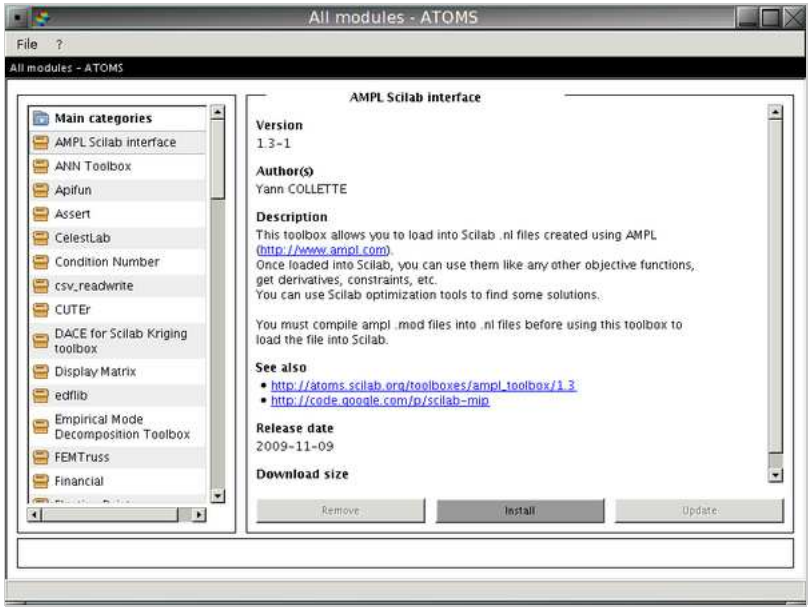
Table des matières — Index

---

13. Atoms

Atoms désigne le système de gestion des modules externes par Scilab ; c'est l'acronyme de *Automatic modules management for Scilab*. Les modules externes sont des paquets de fonctions, développés et mis à disposition par des utilisateurs.

Il suffit de cliquer sur le bouton **Gestionnaire de module — ATOMS** pour ouvrir une interface graphique permetant de sélectionner et d'installer un module. On peut alors utiliser les fonctions de ce module.



gestionnaire de modules Atoms.

Voir aussi

Notes

Xcos < ↑ > *Index*

GFDL

Vous avez la permission de copier, distribuer et/ou modifier ce document selon les termes de la **licence de documentation libre GNU**, version 1.2 ou plus récente publiée par la Free Software Foundation ; sans sections inaltérables, sans texte de première page de couverture et sans texte de dernière page de couverture.

Récupérée de « https://fr.wikibooks.org/w/index.php?title=Découvrir_Scilab/Version_imprimable&oldid=485475 »

Dernière modification de cette page le 23 juillet 2015 à 07:18.
Les textes sont disponibles sous licence Creative Commons attribution partage à l'identique ; d'autres termes peuvent s'appliquer.

Voyez les termes d’utilisation pour plus de détails.
Développeurs