# Chapitre 2
# Introduction à Scilab

Cyrille Bertelle

UFRST
Université du Havre

M1 Math-Info - MD2PMS

# 2.1 Aperçu et environnement I

Scilab est un langage de programmation associé à une riche collection d'algorithmes numériques qui couvrent de nombreux aspects des problèmes de calcul scientifique. C'est un pseudo-clone libre de Matlab qui tourne sous Windows, Linux et Mac.

Développé par l'INRIA (aujourd'hui Consortium Scilab-Digiteo).

Plus précisément, c'est un langage interprété (du style LISP, CAML, ...). Cela permet de faire des développements rapides grâce à des fonctionnalités de haut niveau.

Interfaçage avec des langages comme Fortran ou C, mais aussi LabVIEW.

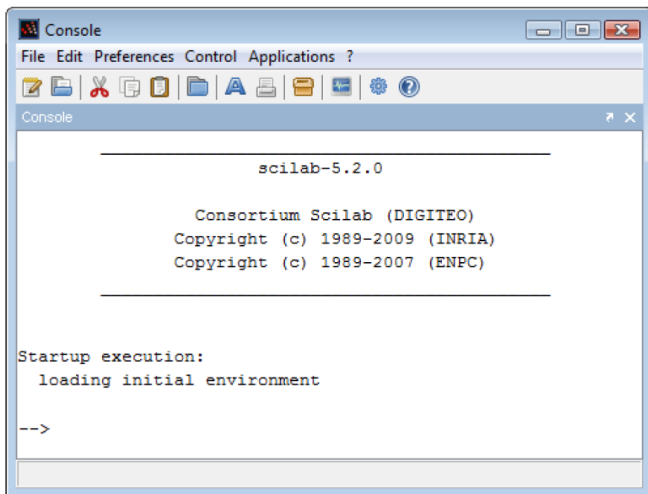
# 2.1 Aperçu et environnement II

Plus précisément, Scilab possède des méthodes usuelles de haut niveau traitant de :

- algèbre linéaire et matriciel
- polynômes et fonctions rationnelles
- interpolation et approximation
- optimisation linéaire, quadratique et non linéaire
- EDO / EDP
- Contrôle classique et robuste
- Traitement du signal
- Statistiques
- Graphisme 2D/3D
- ...

Le lancement de Scilab conduit à une fenêtre qui correspond (au gestionnaire de fenêtre près) à la figure suivante.

# 2.1 Aperçu et environnement III



Une des manières les plus simples d'obtenir de l'aide avec Scilab est d'utiliser la fonction `help` intégrée à l'interface Scilab. Pour cela, il suffit de taper "`help`" dans la console et de valider (touche entrée).

Une fenêtre ***Help Browser*** s'ouvre et il est possible de naviguer dans le menu de gauche.

# 2.1.1 Comment obtenir de l'aide II

# 2.1.1 Comment obtenir de l'aide III

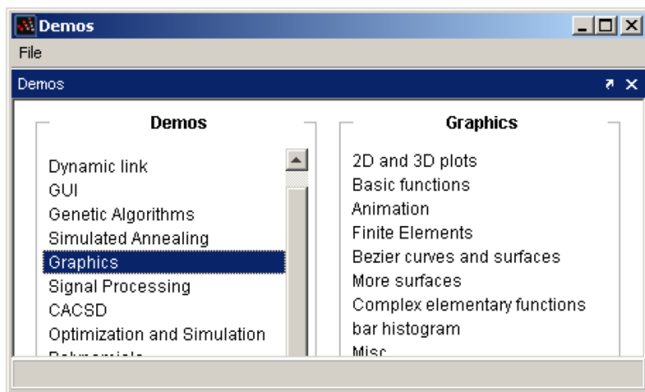
Supposons que l'on souhaite obtenir de l'aide sur la fonction ***optim***, on peut la chercher dans la fenêtre ***Help Browser*** ou simplement en tapant dans la console ***help optim***.

On peut aussi accéder à une importante ressource documentaire sur la page web de Scilab: `http://www.scilab.org/support/documentation`

Des mailing listes existent également.

Finalement, on pourra aussi accéder à des aides sous la forme de démonstrations dans le menu de la fenêtre console intitulé `?`.

# 2.1.1 Comment obtenir de l'aide IV



# 2.1.2 Mode d'utilisation - console, éditeur, batch processing I

Scilab s'utilise principalement par interaction avec le fenêtre console, au travers d'une boucle du type

*lecture-évaluation-résultat*

Voici une première interaction élémentaire :

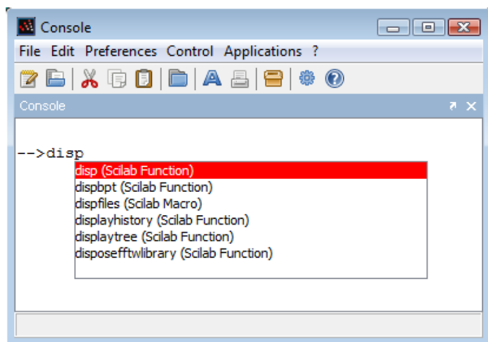
```
--> s="Hello World!"
s=
Hello World!
--> disp(s)
Hello World!
```

# 2.1.2 Mode d'utilisation - console, éditeur, batch processing II

Les flèches ↑ et ↓ permettent une navigation dans la liste des commandes précédemment saisies

La touche `<TAB>` lance un système de complétion facilitant la saisie.

# 2.1.2 Mode d'utilisation - console, éditeur, batch processing III



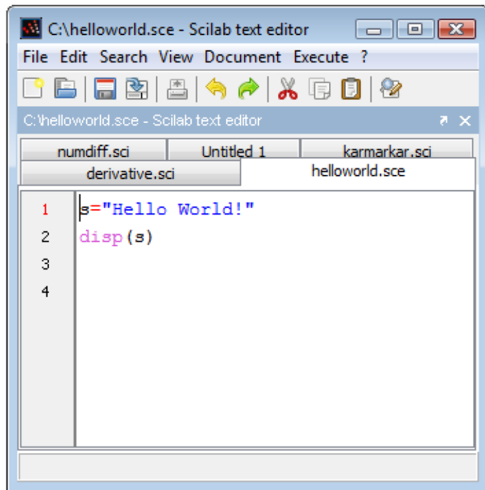
# 2.1.2 Mode d'utilisation - console, éditeur, batch processing IV

L'environnement contient depuis la version 5.2, un **éditeur intégré**. On y accède en tapant directement à la console :

```
-> editor()
```

On peut aussi y accéder en ouvrant le menu `Application > Editor`.

# 2.1.2 Mode d'utilisation - console, éditeur, batch processing V



# 2.1.2 Mode d'utilisation - console, éditeur, batch processing VI

On peut éditer plusieurs fichiers en simultané.

On s'intéressera au menu `Execute` ... voir la première session de TP pour l'expérimentation pratique.

**Exécution de fichiers de scripts / commandes**

A partir de l'éditeur intégré ou d'un autre éditeur, on peut construire un fichier de commandes qui sera introduit dans Scilab par la commande

```
exec nomfichier
```

- un fichier d'extension `.sci` contient des descriptions de fonctions Scilab que l'on peut charger dans l'environnement Scilab sans être exécutée.
- un fichier d'extension `.sce` contient des fonctions, mais aussi des commandes qui seront éxécutées au chargement.

# 2.1.2 Mode d'utilisation - console, éditeur, batch processing VIII

## Mode batch

On peut lancer Scilab en mode batch soit pour disposer d'un environnement simplifié (sans fenêtrage), soit pour lancer, en externe, des scripts et fichiers `.sce`

Suivant le système (Windows, Linux ou Mac), le nom de lancement différe :

- `Scilex` sous Windows
- `scilab` ou `scilab-cli` ou `scilab-adv-cli` sous Linux ou Mac

# 2.1.2 Mode d'utilisation - console, éditeur, batch processing IX

`Scilex -nw` et `scilab-adv-cli` lancent scilab en ligne de commande, sans fenêtrage.

`Scilex -f myscript.sce` lance scilab et le fichier de commande directement. Si ce ficheir se termine par `quit()`, scilab se referme alors directement après l'éxécution du fichier de commande.

# 2.2 Eléments de base I

Scilab est un langage interprété permettant des constructions dynamiques de programmes. Généralement, comme pour *Matlab*, on peut dire que presque tout dans Scilab est une matrice (sauf des structures de données moins usuelles ; ce sera notamment des listes, par exemple, mais que l'on sera amené à peu utiliser dans la suite).

Malgré cette remarque, nous allons commencer par décrire des types de données simples (des matrices 1x1) avant de passer aux aspects plus généraux de traitements de matrice.

## Variables réelles ou numériques

Voici une séquence d'instructions illustrant une manipulation très claasique des variables numériques. On notera l'opérateur d'affectation représenté par un seul signe = qu'il ne faut pas confondre avec le double symbole == qui désigne l'opérateur booléen de comparaison.

```
--> x=1
x=
1.
--> x=x*2
x=
2.
```

En ajoutant un point-virgule `;` à la fin d'une commande, on demande de ne pas générer d'affichage suite à l'exécution de la commande.

- $x$^2 ou $x**2$ désigne l'élévation au carré
- $x/y = xy^{-1}$
- $x\backslash y = x^{-1}y$

# 2.2 Eléments de base IV

**Noms de variables**

Seuls les 24 premiers caractères d'un nom de variable sont significatifs. Un nom de variable est constitué :

- des lettres de l'alphabet en minuscules ou majuscules, avec différenciation de ces deux types
- des nombres de 0 à 9
- des caractères % _ # ! $ ?

Une variable commençant par % a une signification spéciale (vue plus tard).

## Commentaires et lignes de continuation

Un commentaire est une ligne démarrant par `//`.
Une ligne se terminant par deux points successifs `..` sera prolongée par la ligne suivante.

```
--> // c'est un commentaire
--> x = 1 ..
--> + 2 ..
--> + 3
x=
6
```

## Fonctions mathématiques élémentaires

Scilab intégre les fonctions mathématiques courantes :

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| acos | acosd | acosh | acoshm | acosm | acot | acotd | acoth |
| acsc | acscd | acsch | asec | asecd | asech | asin | asind |
| asinh | asinhm | asinm | atan | atand | atanh | atanhm | atanm |
| cos | cosd | cosh | coshm | cosm | cotd | cotg | coth |
| cothm | csc | cscd | csch | sec | secd | sech | sin |
| sinc | sind | sinh | sinhm | sinm | tan | tand | tanh |
| tanhm | tanm | | | | | | |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| exp | expm | log | log10 | log1p | log2 | logm | max |
| maxi | min | mini | modulo | pmodulo | sign | signm | sqrt |
| sqrtm | | | | | | | |

# 2.2 Eléments de base VII

La plupart de ces fonctions prennent un argument et en retourne un autre. Elles sont “vectorisées”, c’est à dire qu’elles manipulent des matrices.

Exemple :

```
--> x = cos(2)
x=
-0.4161468
--> y = sin (2)
y=
0.9092974
--> x^2 + y^2
ans=
1.
--> ans*2
ans=
2.
```

# 2.2 Eléments de base VIII

On notera la variable `ans` qui évalue l'expression courante et qui peut être réutilisée.

### Variables mathématiques prédéfinies

Dans Scilab, plusieurs variables mathématiques sont prédéfinies ; elles commencent par % :

- % i : nombre complexe dont le carré vaut -1
- % e : constante d'Euler, base du mogarithme népérien
- % pi : constante $\pi$

### Booléens

Les booléens sont des variables qui peuvent prendre deux valeurs :

- %t ou %T pour “true”
- %f ou %F pour “false’

,

# 2.2 Eléments de base XI

Les opérateurs booléens sont listés ci-dessous

| | |
|---|---|
| a&b | logical and |
| a\|b | logical or |
| ~a | logical not |
| a==b | true if the two expressions are equal |
| a~=b or a<>b | true if the two expressions are different |
| a<b | true if a is lower than b |
| a>b | true if a is greater than b |
| a<=b | true if a is lower or equal to b |
| a>=b | true if a is greater or equal to b |

## Nombres complexes

Les nombres complexes s'écrivent sous leur forme mathématique habituelle en décomposant la partie réelle et la partie imaginaire avec le nombre % i, imaginaire pur dont le carré vaut -1.
Listes des fonctions sur les nombre complexes

| | |
|---|---|
| real | real part |
| imag | imaginary part |
| imult | multiplication by $i$, the imaginary unitary |
| isreal | returns true if the variable has no complex entry |

**Chaînes de caractères**

**Modification de variables**

# 2.4 Visualiser un graphe simple I

On va construire le graphe de la fonction

$$y = e^{-x}\sin(4x) \quad \text{pour } x \in [0, 2\pi]$$

On commence alors par créer un maillage de l'intervalle $[0, 2\pi]$ avec la fonction `linspace` :

```
-->x=linspace(0,2*%pi,101);
```

$x$ est alors un vecteur de valeurs.

On calcule ensuite les valeurs de la fonction pour chaque composante du vecteur $x$, ce qui se fait en une seule instruction puisque Scilab manipule "naturellement" des vecteurs et matrices :

```
-->y=exp(-x).*sin(4*x);
```

# 2.4 Visualiser un graphe simple II

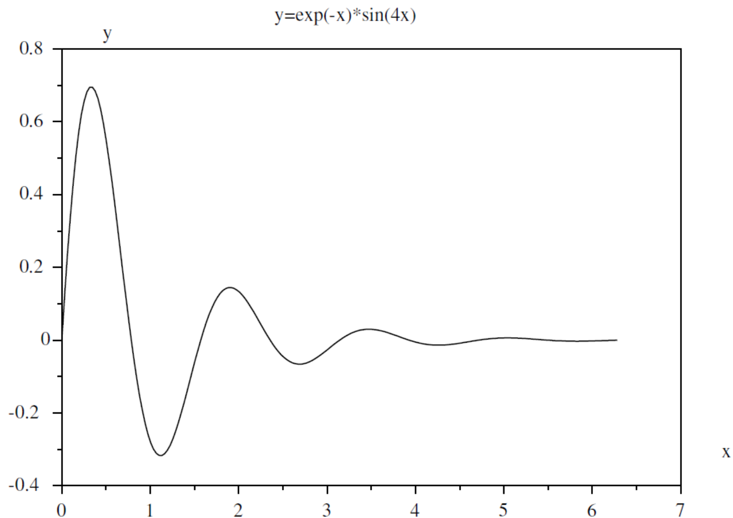
Le tracé du graphique se fait alors par l'instruction `plot` ou `plot2d`. Elle est suivie ici par l'instruction `xtitle` qui permet d'afficher des légendes :

```
-->plot(x,y)
-->xtitle("y=exp(-x)*sin(4x)", "x", "y")
```

L'instruction permet de tracer une courbe passant par les points dont les coordonnées sont données dans les vecteurs $x$ pour les abscisses et $y$ pour les ordonnées. Les points sont reliés par des segments de droites.

Remarque : on étudiera plus tard des possibilités évoluées de tracé graphique.

# 2.4 Visualiser un graphe simple III



# 2.5 Ecrire et exécuter un script I

On peut écrire dans un fichier `nomfich` une suite de commandes et les faire exécuter par l'instruction :

```
-->exec('nomfich') // ou encore exec("nomfich")
```

On peut aussi charger le fichier par l'intermédiaire du menu "fichier", puis l'éxécuter par l'item "Exécuter".

Voici un petit exemple que l'on nomera "script.sce" :

```
// mon premier script Scilab
a = input(" Rentrer la valeur de a : ");
b = input(" Rentrer la valeur de b : ");
n = input(" Nb d'intervalles n : ");
// calcul des abscisses
x = linspace(a,b,n+1);
// calcul des ordonnees
```

# 2.5 Ecrire et exécuter un script II

```
y = exp(-x).*sin(4*x);
// representation graphique
plot(x,y)
xtitle("y=exp(-x)*sin(4x)","x","y")
```

## 2.6.1 Lecture par le clavier et affichage I

Une saisie au clavier s'effectue comme cela a été illustré dans l'exemple de script donné dans la section précédente. On utilise la commande `input`.

## 2.6.1 Lecture par le clavier et affichage II

### Exemple

*Quelques saisies au clavier*

```
--> n=input("entrer la dimension n")
entrer la dimension n
--> 2
n =
  2.
--> a=input("entrer une matrice")
entrer une matrice
--> [1, 2; 3, 4]
a =
! 1.  2.!
! 3.  4.!
```

## 2.6.1 Lecture par le clavier et affichage III

L'entrée d'une chaîne de caractères se fait encore avec la fonction `input`, en ajoutant un second paramètre qui est la constante “string”.

### Exemple

*Saisie d'une chaîne de caractères*

```
--> nomfichier =
     input("donner le nom du fichier",
           "string")
donner le nom du fichier
--> resultats.txt
nomfichier =
   resultats.txt
```

## 2.6.1 Lecture par le clavier et affichage IV

Une des commandes les plus usuels pour afficher les variables est `disp` qui contient autant d'arguments d'entrée que de variables à afficher.

### Exemple

*Affichage de données avec “disp”*

```
--> a=[1, 2; 3, 4]; b="coucou"; c=2-sqrt(3);
--> disp(a,b,c)
 .2679492
 coucou
 ! 1.  2.!
 ! 3.  4.!
```

On remarquera l'ordre inversé d'affichage des données par rapport à l'ordre des arguments en entrée à la fonction “disp”.

On peut aussi utiliser l'instruction “printf” qui est en fait celle qui est héritée du langage C.

**Exemple**

*Affichage avec la fonction “printf”*

```
--> a=64; b=sin(a);
--> printf('le sinus de %f \n est %f', a, b)
le sinus de 64.00000
est .920026
```

On ne donnera pas plus de détails ici sur les spécifications de `printf`. Le lecteur non habitué au langage C trouvera les informations dans tout manuel ou tutorial sur ce langage.

**Exercice**

*Utiliser l'aide pour savoir ce que fait la fonction* `x_message`*.*
*Tester cette fonction en utilisant 1 ou 2 paramètres.*

# 2.6.2 Utilisation de fichiers I

Deux fonctions de base servent à écrire des données dans un fichier.

`print (nomf, a)` ou `write(nomf, a)` : écrit la variable `a` dans le fichier dénommé `nomf`, en utilisant un format par défaut.

**Exemple**

*Ecriture de données non formatées dans un fichier*

```
--> a=[1,2,3; 4,5,6]; b(1)="Université";
          b(2)="du Havre";
--> write("fic1",a); write("fic2",b);
```

**Exercice**

*Tester les instructions précédentes (et ensuite les suivantes) et éditer les deux fichiers générés pour observer leur contenu.*

# 2.6.2 Utilisation de fichiers II

On pourra relire ces fichiers par la commande `read`.

`a=read(nomf,m,n)` lits dans le fichier `nomf` les `m` premières lignes et les `n` premières colonnes et les affecte à la matrice `a`. Si on veut lire toutes les lignes du fichier mais que l'on n'en connait pas le nombre, on remplace le second argument, noté `m` précédemment, par -1.

## Exemple

*Lecture d'un fichier*

```
--> c=read(``fic1'',2,3);
--> d=read(``fic2'',-1,1,'(a)');
--> c, d
c =
! 1.   2.   3.!
! 4.   5.   6.!
d =
! Université !
!            !
! du Havre   !
```

Remarquons que nous avons ajouté un quatrième argument pour la lecture de “fic2”. Cet argument désigne un format, ici

# 2.6.2 Utilisation de fichiers IV

'(a)' qui indique que le fichier contient des chaînes de caractères. Cette indication est ici indispensable sinon ce sont des valeurs numériques qui sont attendues à la lecture du fichier et une erreur est alors générée.

Plus généralement, les 2 instructions `write` et `read` peuvent avoir un argument supplémentaire correspondant à un format. Les formats d'écriture correspondent à ceux du langage C. Le lecteur est invité à consulter l'aide en ligne pour plus de précisions.

**Exercice**

*Rechercher dans l'aide en ligne, la fonction* `file` *qui permet de gérer des fichiers et tester quelques exemples de manipulation.*

Scilab propose un langage de programmation complet qui permet d'aller plus loin que d'utiliser des primitives toutes prêtes. Il s'agit d'un langage beaucoup plus "simple" que les langages classiques (C, C++, Java, ...) car il intégre déjà des primitives sophistiquées, notamment pour manipuler les matrices". Dans un soucis de simplification, le langage ne nécessite pas de déclarations qui sont gérées automatiquement par l'interpréteur. Une autre facilité appréciable est l'intégration d'une bibliothèque graphique.

# 2.7 La programmation en Scilab II

L'inconvéniant majeur du langage Scilab par rapport aux langages classiques est qu'il n'est pas compilé et traduit directement en langage machine. Par ailleurs ses capacités de représentation de nombres cachent des routines soft de manipulation de grands nombre plutôt que des insctructions adhoc pour le processeur de la machine. Un programme Scilab nécessitera facilement 10 fois plus de temps qu'un programme compilé.

Scilab offre un interfaçage avec les langaes Fortran 77 et C, permettant de lancer des sous-programmes écrits et compilés dans l'un de ces langages.

# If - instructions conditionnelles I

## Exemple

*Test simple*

```
if (%t) then
   disp("Hello!")
end
```

# 2.7.1 Branchements et boucles
# If - instructions conditionnelles II

**Exemple**

*Test avec alternative*

```
if (%f) then
   disp("Hello!")
else
   disp("Bye!")
end
```

On peut enchaîner plusieurs tests avec “elseif”.
L’expression de sélection qui se trouve entre paranthèses, après “if”, est une expression booléenne. On se souviendra que si c’est une comparaison, on utilisera le doule symbole “==” et non “=” qui est une affectation.

## Exemple

*Instruction select*

```
i=2
select i
case 1 disp("One")
case 2 disp("Two")
case 3 disp("Three")
else disp("Beaucoup")
end
```

Seule l'instruction qui suit directement un case est exécutée (et pas les suivantes) si la comparaison est vraie. “Else” (qui est facultatif) sera effectuée si tout les tests précédents ont échoués.

# For - boucle I

Pour construire une boucle de type “for”, on utilise un index qui décrit un ensemble de valeurs. Ici, on va utiliser l’opérateur deux-points “:” que l’on manipuler dans le chapitre précédent.

**Exemple**

*Séquence d’exemples d’utilisation de l’opérateur “:”*

```
--> i=1:5
i =
  1.  2.  3.  4.  5.
--> i=1:2:5
i =
  1.  3.  5.
--> i=5:-1:1
i =
  5.  4.  3.  2.  1.
```

# For - boucle II

**Exemple**

*Construction de boucle “for” élémentaires*

```
for i=1:2:5
  disp(i)
end
```

En fait, l’indice de boucle peut parcourir tout type de séquences de données, comme des vecteurs, des lignes de matrices, des listes. Il faut que les éléments énumérés dans ces séquences soient de type entier, double, chaînes ou polynômes.

# For - boucle III

## Exemple

*Construction d'une boucle "for" parcourant une matrice/vecteur*

```
v=[1.5 exp(1) %pi];
for x=v
  disp(x)
end
```

## Remarque

*Un point important sur l'utilisation de la boucle "for" est de savoir si il n'est pas possible de la remplacer par une opération matricielle, c'est à dire une opération "vectorisée". En effet il peut y avoir un facteur de performance de l'ordre de 10 à 100 entre ces deux calculs au bénéfice des opérations vectorisées.*

# While - boucle I

Une boucle “while” conditionne la répétition d’une séquence d’instructions de manière conditionnée à la vérification d’une expression booléenne.

**Exemple**

*Une boucle “while” élémentaire calculant la somme des 10 premiers entiers*

```
s=0; i=1;
while (i<=10)
  s=s+i
  i=i+1
end
```

Pour ce calcul spécifique, on pourra utiliser plus efficacement la fonction prédéfinie “sum” :

## Exemple

*Utilisation de la fonction "sum"*

```
--> sum(1:10)
ans =
    55.
```

## Remarque

*La boucle "while" produit le même effet que la boucle "for" et il faut à nouveau se demander si une opération vectorielle ne peut pas la remplacer de manière plus efficace.*

# Break et continue - sorties I

Il s’agit de deux instructions qui sont à bannir pour les “puristes” de l’algorithmique (car traduisant une faiblesse d’analyse et des comportements invérifiables).

L’instruction “break” permet de sortir d’une boucle. Typiquement on vérifie une condition qui lorsqu’elle est satisfaite, ne nécessite plus de continuer la boucle (similaire à un traitement d’exception dans d’autres langages).

# Break et continue - sorties II

## Exemple

*Utilisation basique de "break"*

```
s=0; i=1;
while (%t)
  if (i>10) then
    break
  end
  s=s+i
  i=i+1
end
```

L'instruction "continue" permet de signaler à l'intérieur d'une boucle que l'on doit "sauter" la fin des instructions restant à exécuter dans le corps de la boucle pour passer directement à l'itération suivante.

## Exemple

*Utilisation basique de "continue"*

```
s=0; i=0;
while (i<=10)
  if (modulo(i,2)==0) then
    i=i+1
    continue
  end
  s=s+i
  i=i+1
end
```

**Remarque**

*La construction précédente, qui est un exemple de construction algorithmique à ne pas suivre, s'obtient plus simplement et efficacement avec la seule instruction :* `sum(1:2:10)`.

# 2.7.2 Les fonctions I

La démarche essentielle qui permet d'élaborer des programmes en Scilab, est basée sur la décomposition fonctionnelle, c'est à dire à élaborer un programme de manière incrémentale en définissant des briques de base fonctionnelles. On va ainsi construire des fonctions de base que l'on va assembler ou utiliser dans d'autres fonctions plus élaborées et abstraites.

Une fonction en Scilab s'utilise de la manière suivante :

```
outvar=myfunction(invar)
```

où

- `myfunction` est le nom de la fonction ;
- `invar` correspond aux arguments d'entrée qui doivent être fournis à la fonction pour qu'elle puisse s'exécuter ;
- `outvar` correspond aux arguments de sortie dont le but de l'exécution de la fonction est de les calculer.

# 2.7.2 Les fonctions III

Nous avons déjà utilisé beaucoup de fonctions qui sont prédéfinies dans Scilab, notamment les fonctions mathématiques usuelles telles que sin, cos, etc.

Les fonctions Scilab peuvent avoir un nombre arbitraire d'arguments d'entrée et d'arguments de sortie. On les distingue de la manière suivante :

```
[o1, o2, ..., on] =
          myfunction(i1, i2, ..., in)
```

# 2.7.2 Les fonctions IV

## Exemple

*Décomposition LU d'une matrice (exécuter les lignes d'instructions suivantes puis utiliser "help" pour comprendre ce qu'elles font)*

```
A=testmatrix("hilb",2)
[L,U]=lu(A)
[L,U,P]=lu(A)
```

# Définir une fonction I

## Exemple

*Construction d'une fonction élémentaires*

```
function y = myfunction(x)
  y=2*x
endfunction
```

Ecrire la définition d'une fonction peut être long suivant la complexité de celle-ci. On peut la saisir en ligne dans l'environnement Scilab qui attendra de lire “endfunction” pour l'évaluer et renvoyer un prompt.

Si cette description est longue, il peut être préférable d'utiliser l'éditeur proposé dans Scilab et d'utiliser dans le menu de celui-ci, “Exécuter/Charger dans Scilab” pour la charger dans l'environnement interactif.

# Définir une fonction II

Une troisième alternative consiste à sauvegarder le code de la fonction (ou de plusieurs fonctions) dans un fichier d'extension ".sci"(fichier ne contenant que des définitions de fonctions) ou ".sce" (fichier contenant en plus des instructions exécutables), puis de d'utiliser la fonction "exec" dans l'environnement interactif avec une instruction qui pourrait être

```
exec("myFonctionsFile.sce")
```

**Exercice**

*Définir la fonction précédente en utilisant successivement les 3 modes d'édition proposés.*

# Définir une fonction III

**Remarque**

*Il est essentiel dans le code de la fonction de trouver des instructions qui affectent des valeurs à chacun des arguments de sortie. Si ces affectations ne sont pas présentes, une erreur sera générée à l'appel de la fonction.*

# Gestion des arguments de sortie I

Voici un exemple de fonction renvoyant 2 arguments en sortie. Nous allons nous intéresser à différentes manière de récupérer ces arguments pour comprendre certaines spécificités de Scilab.

**Exemple**

*Fonction renvoyant deux arguments de sortie*

```
function [y1,y2]=simplef(x1,x2)
  y1=2*x1
  y2=2*x2
endfunction
```

Voici plusieurs résultats d'exécution :

# Gestion des arguments de sortie II

```
--> [z1,z2] = simplef(1,2)
z2 = 6.
z1 = 2.
--> simplef(1,2)
ans = 2.
--> z = simplef(1,2)
z = 2.
```

La première instruction permet de récupérer les deux arguments de sortie comme décrit dans le corps de la fonction. La seconde et troisième instruction ne permettent de ne récupérer qu'une valeur qui est alors la valeur du premier argument de sortie (le second est alors perdu).

Nous donnons ci-dessous un nouvel exemple permettant d'illustrer la construction de fonctions récursives, ne nécessitant en Scilab aucun signalement spécifique.

### Exemple

*Fonction récursive factorielle*

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

**Remarque**

*Compléments vus en TP : des fonctions particulières de manipulation de fonctions permettant notamment de servir d'aide au débogage (disp, pause, resume, setbpt, delbpt, error, warning, type, typeof, argn, exists).*

# Utiliser une fonction comme argument d'une autre fonction I

Une fonction est elle-même une variable du type "function" et elle peut donc être passée comme argument d'une autre fonction.

**Exemple**

*Fonction possédant en argument d'entrée une autre fonction*

```
function y=f(x)
  y=sin(x) .* exp(-abs(x))
endfunction
```

# Utiliser une fonction comme argument d'une autre fonction II

## Exemple

*Fonction possédant en argument d'entrée une autre fonction (suite)*

```
function graphef(a,b,fct,n)
// n est un argument optionnel,
// sa valeur par défaut est 61
  [lhs,rhs]=argn(0)
  if rhs==3 then n=61
  end
  x=linspace(a,b,n)
  y=fct(x)
  plot(x,y)
endfunction
```

# Utiliser une fonction comme argument d'une autre fonction III

On pourra alors tester la fonction `graphef` par les instructions suivantes :

```
--> graphef(-2*%pi,2*%pi,f)
--> graphef(-2*%pi,2*%pi,f,21)
```

# Construire des fonctions de manière dynamique avec “deff”, “evstr” et “execstr” I

Scilab propose de définir des fonctions “in-line”, grâce à une seule instruction grâce à la commande `deff` qui prend en argument d’entrée des chaînes de caractères où la fonction sera décrite.

**Exemple**

*Définition d’une fonction “in-line”*

```
--> deff('[s,d]=plusmoins(a,b)',
          ['s=a+b', 'd=a-b'])
-->[p,q]=plusmoins(2,3)
 q =
  -1.
 p =
   5.
```

# Construire des fonctions de manière dynamique avec “deff”, “evstr” et “execstr” II

Cette possibilité est vraiment intéressante car elle permet de définir du code de manière dynamique : une chaîne de caractères contenant la totalité ou des éléments d'une fonction, peut-être lue au cours de l'exécution d'un programme et être utilisée en temps que fonction dans la suite du déroulement du programme.

Scilab offre aussi la possibilité d'évaluer une expression qui est décrite dans une chaîne de caractères, grâce à la fonction `evstr`. On peut également exécuter une instruction décrite dans une chaîne de caractères, grâce à la fonction `execstr`.

# Construire des fonctions de manière dynamique avec “deff”, “evstr” et “execstr” III

## Exemple

*Conversion de chaîne de caractères en expression évaluable ou en instructions exécutables*

```
--> evstr("sqrt(3)/2")
ans = 0.8660254
--> a=1; evstr("2 + a")
ans = 3.
--> evstr(["a" "2"])
ans = ! 1.   2.!
--> execstr("A=rand(2,2)"); A
 A =
! 0.2113249  0.0002211 !
! 0.7560439  0.3303271 !
```