Initiation à Scilab

1 Introduction

Scilab est l'abréviation de "*Scientific laboratory*". Il s'agit d'un environnement informatique conçu spécifiquement pour le calcul matriciel. De même que son "grand frère" MATLAB (malheureusement, un produit commercial !), dont il reprend les principes, ce logiciel est parfaitement adapté à l'analyse de circuits, au traitement du signal et des images, à la conception de filtres, à l'analyse de systèmes de commande, et à bien d'autres domaines d'application. De plus, sa facilité d'emploi avec les nombres complexes et les tracés graphiques en fait un outil intéressant dans de nombreux problèmes de programmation. Scilab peut d'ailleurs être vu comme un langage de programmation de quatrième génération. Il peut être utilisé dans un mode interactif où les instructions sont exécutées immédiatement après qu'elles aient été tapées. Inversement, un programme peut être écrit à l'avance et sauvegardé sur le disque dans un fichier, à l'aide d'un éditeur, puis exécuté sous Scilab. Les deux modes sont utiles.

Signalons que le logiciel peut être téléchargé gratuitement à l'adresse `http://www.scilab.org`.

2 Aide

Scilab possède une aide (*help*) en ligne et toute une collection de démonstrations. Pour obtenir des détails sur les aides disponibles, tapez

```
--> help
```

Pour obtenir une aide sur une fonction spécifique, par exemple `sin`, tapez

```
--> help sin
```

Pour rechercher les fonctions relatives à un mot clef donné (en anglais), on peut utiliser la commande `apropos`. Par exemple, essayez

```
--> apropos('root')
```

Enfin, pour avoir un aperçu rapide des fonctionnalités de Scilab, vous pouvez aussi lancer quelques démos.

3 Une calculatrice sur ordinateur

Premiers exemples.
Scilab tout comme Matlab peut-être vu comme étant fondamentalement une calculatrice sur ordinateur. Au lancement de Scilab on voit une fenêtre vide à l'exception d'un petit texte nous donnant la version du logiciel et une invite du type : `-->`. Il suffit alors de taper une commande et d'appuyer sur la touche `Entrée` pour qu'elle s'exécute :

```
--> 1+3.1
   ans  =
    4.1
```

On remarque que la *virgule* française est remplacée par un point ".". Si on tape la même commande mais suivie d'un point virgule : ";", on constate qu'il n'y a aucun affichage. Pour autant Scilab a bien exécuté le calcul mais la présence du point virgule lui indique qu'on ne désire pas le voir affiché. Ci-dessus nous avons fait une addition ; les autres opérations classiques se font à l'aide des opérateurs : `-`, `*` et `/`. Par exemple, la formule $(1+2)\frac{5}{3}$ s'écrit :

```
--> (1+2)*5/3
```

Les fonctions classiques.
Si l'on désire calculer $\sqrt{2}$, il faut cette fois utiliser la fonction "racine carrée", en anglais *square root* ce qui explique le nom de la fonction que nous allons utiliser : `sqrt`.

```
--> sqrt(2)
```

Voici un tableau présentant la correspondance entre les opérations les plus courantes d'une calculatrice et les fonctions de Scilab :

Fonction classique :	Syntaxe dans Scilab	Exemple
Racine carrée : $\sqrt{x}$	`sqrt`	`sqrt(2)`
Carré : x^2	`^2`	`5^2`
Puissance : x^p	`^`	`5^7`
Exponentielle : e^x	`exp`	`exp(0)`
Logarithme : $\ln(x)$	`log`	`log(1)`
Logarithme en base 10 : $\log_{10}(x)$	`log10`	`log10(10)`
Cosinus : $\cos(x)$	`cos`	`cos(0)`
Sinus : $\sin(x)$	`sin`	`sin(0)`
Tangente : $\tan(x)$	`tan`	`tan(0)`
Valeur absolue : $\lvert x\rvert$	`abs`	`abs(-1)`
Partie entière inférieure : $\lfloor x\rfloor$	`floor`	`floor(1.4)`
Partie entière supérieure : $\lceil x\rceil$	`ceil`	`ceil(1.4)`
Arrondi entier le plus proche	`round`	`round(1.4)`

Les variables. Sur de nombreuses calculatrices on trouve une touche `M+` qui permet de mettre en mémoire le résultat d'un calcul. Sous Scilab on peut aussi sauvegarder nos calculs dans ce qu'on appellera des variables. Voyons un premier exemple :

```
--> 3+1
```

On observe ici que Scilab renvoie le résultat précédé de la mention `ans =`. Ce `ans`, de l'anglais *answer*, signifie que le résultat du calcul qui vient d'être effectué est stocké dans une variable appelée `ans`. Tout résultat d'un calcul dans Scilab est stocké dans une variable. Si l'on ne fournit pas de variable alors il enregistre le résultat dans la variable par défaut : `ans`. Si l'on tape une autre instruction comme celle ci-dessus alors le résultat sera lui aussi stocké dans `ans` et remplacera le résultat précédent. Pour éviter ce genre de problème, nous pouvons enregistrer le résultat d'un calcul dans une variable que nous choisissons. Pour cela il suffit d'écrire par exemple :

```
--> a=sin(12)^2+cos(12)^2
```

La variable `a` contient désormais la valeur 1. Pendant toute la durée de l'exécution de Scilab et tant que `a` n'est pas effacée ou réaffectée, `a` conservera cette valeur et pourra être rappelée à tout moment ; ainsi :

```
--> b=sqrt(a+15)-a;
--> b+log(a)
```

La première ligne est terminée par un point virgule, ce qui interdit à Scilab d'afficher le résultat. L'exécution de la deuxième ligne nous prouve que la variable `b` a pourtant bien été définie et a reçu la valeur 3.
Le choix des noms de variables est libre mais suit tout de même quelques règles :

- Une variable doit avoir un nom commençant par une lettre non accentuée (ou parfois par des symboles comme `%` ou `$`). Par exemples les noms `A` ou `f` sont corrects, par contre `ù`, `ç`, `£e` ou `1a` ne le sont pas.
- Les caractères alphanumériques peuvent être utilisés pour écrire la suite du nom de la variable. Par exemple, les noms `a1` ou `temp` sont corrects.
- Pour éviter les confusions, il est recommandé de ne pas choisir comme nom de variable le nom d'une fonction déjà existante. En cas de doute, rechercher dans l'aide si le nom est déjà utilisé.
- Scilab fait la distinction entre majuscule et minuscule pour les noms de variables. Par exemple :

```
--> a=3;
--> A=2;
--> a-A
```

Constantes.
Un certain nombre de constantes sont définies dans Scilab. En voici quelques unes qui peuvent être utiles :

- `%pi` : π, par exemple : `cos(%pi)`.
- `%e` : constante d'Euler.
- `%i` : le nombre imaginaire $\imath$ (par convention le nombre tel que $\imath^2 = -1$).
- `%inf` : cette constante sert à représenter l'infini. Par exemple : `exp(1000)`.
- `%nan` : nous sommes parfois confronté à des indéterminations en mathématiques ; cette constante sert à représenter ces cas. Cet acronyme signifie *not a number* (pas un nombre). Par exemple : `%inf-%inf`.

4 Vecteurs et matrices

Nous avons vu que Scilab peut être utilisé comme une simple calculatrice mais à l'origine Scilab est spécialement conçu pour manipuler des matrices et des vecteurs.

Les vecteurs.
Ce que nous appellerons ici un vecteur peut aussi bien être compris comme une suite finie de nombres. La syntaxe la plus simple pour construire un vecteur est la suivante :

```
--> V = [1 2 3 4 5 6 7 8 9 10]
```

Ce vecteur `V` contient donc tous les nombres entiers compris entre 1 et 10. On les a entrés à la main en écrivant chaque valeur séparée par un espace (on aurait pu aussi les séparer par des virgules) et le tout est encadré par des crochets `[ ]`. Cette liste de nombres peut-être constituée de n'importe quelle expression Scilab. Par exemple :

```
--> x = [-1.3 sqrt(3) (1+2+3)*4/5]
```

où l'on a saisi un nombre, le résultat d'une fonction et une expression arithmétique.
Un vecteur est comme nous l'avons dit une suite finie de nombres : $V = (v_1, v_2, \ldots, v_M)$. Formellement pour accéder au nombre en position n dans notre suite on écrit : v_n. Sous Scilab on peut faire de même : on accède au nombre en position n dans notre vecteur en tapant `V(n)`. Par exemple, pour accéder au sixième nombre du vecteur `V` ci-dessus on tape

```
--> V(6)
```

On peut aussi se servir de cette notation pour changer une valeur dans un vecteur :

```
--> V(6)=-3
```

On a remplacé dans le vecteur `V` la valeur en sixième position par la valeur -3.
Cette notation permet aussi d'augmenter la taille d'un vecteur :

```
--> V(11)=64
```

On a rajouté une onzième valeur à notre vecteur. Si l'on tape

```
--> V(13)=63
```

On n'a pas défini de douzième valeur à notre vecteur et pourtant Scilab n'a pas fait d'erreur. Il a complété automatiquement la valeur manquante en mettant un zéro en douzième position.
Enfin, pour connaître la longueur d'un vecteur il suffit d'utiliser la fonction `length` (*longueur* en anglais).

```
--> M=length(V)
```

Opérations et fonctions sur les vecteurs.
On peut facilement multiplier tous les éléments d'un vecteur par un scalaire :

```
--> U=6*V
```

Si l'on dispose de deux vecteurs `U` et `V` de même longueur, on peut les soustraire et les additionner de la même manière que les scalaires en utilisant les opérateurs `+` et `-`. Par exemple :

```
--> U-V
```

Cela revient à faire les opérations sur ces vecteurs coordonnée par coordonnée : $(u_1-v_1, ..., u_{13}-v_{13})$.
On pourrait souhaiter faire le même type d'opération avec la multiplication et la division mais si l'on essaye de taper le même type d'expression avec les opérateurs `*` et `/` on génère bien vite des erreurs. En effet, les multiplications de vecteurs en mathématiques sont régies par certaines règles issues de l'algèbre linéaire et donc ces opérateurs réalisent une autre opération de multiplication que nous ne détaillerons pas pour le moment. La volonté des concepteurs de Matlab et de Scilab étant de faire un logiciel de calcul numérique favorisant l'utilisation des matrices et des vecteurs, il était nécessaire que la multiplication et la division coordonnée par coordonnée soient définies. Pour cela on utilise les opérateurs `.*` et `./` (avec un point). Par exemple :

```
--> (U.*V)./(U-V)
```

On ajoute également un point pour prendre la puissance coordonnée par coordonnée. Exemple :

```
--> U.^3
```

On a vu quelques fonctions classiques comme le logarithme ou le sinus. Toujours dans un souci de favoriser la manipulation des matrices et des vecteurs, les fonctions classiques s'appliquent également sur les vecteurs, coordonnée par coordonnée. Ainsi :

```
--> sin(U)
```

calcule le sinus de chaque coordonnée de `U`. Une autre opération souvent utile est la concaténation de vecteurs (c'est à dire la mise bout à bout de deux vecteurs). La syntaxe est très simple puisqu'on se contente de créer un nouveau vecteur contenant les deux vecteurs à la suite. Par exemple :

```
--> [ U [1 2 3] ]
```

Les matrices.
Une matrice est un objet mathématique utile en algèbre linéaire mais qui peut être simplement vu comme un tableau de nombres. La manière la plus simple d'entrer une matrice est d'utiliser une ligne explicite d'éléments. Dans la liste, les éléments sont séparés par des espaces ou des virgules, et des points virgules (;) sont utilisés pour indiquer la fin de ligne. La liste est encadrée par des crochets `[ ]`. Par exemple :

```
--> A = [1 2 3;4 5 6;7 8 9]
```

La variable `A` est donc une matrice de dimension 3×3. Comme pour les vecteurs, les éléments d'une matrice peuvent être formés de n'importe quelle expression Scilab (nombre, expression algébrique, résultat de fonction...). On peut accéder à tout élément d'une matrice en utilisant simplement ses coordonnées : en tapant `A(2,1)` on obtient l'élément se situant à la deuxième ligne de la première colonne de la matrice. Pour connaître la taille d'une matrice on dispose de la fonction `size` (la fonction `length` renvoie quant à elle le nombre d'éléments dans la matrice)
Si l'on écrit une matrice n'ayant qu'une ligne, on retrouve la syntaxe décrite précédemment pour les vecteurs. En ce sens, on peut considérer que les matrices sont une généralisation des vecteurs. Un certain nombre de faits qu'on a vu pour les vecteurs restent vrais. On peut ainsi appliquer les fonctions classiques sur les matrices, on peut additionner ou soustraire deux matrices élément par élément, et on peut multiplier ou diviser deux matrices de même taille élément par élément en utilisant `.*` et `./`. Par exemple :

```
--> B= sin(A)
--> (A-6*B).*(cos(A).^2)
```

Enfin citons le caractère spécial (`'`) qui sert à désigner la transposée d'un matrice. La commande

```
--> A'
```

transforme les lignes de `A` en les colonnes de `A'`, et vice versa.

Quelques matrices utiles.

- Pour créer une matrice identiquement nulle on utilisera la fonction `zeros` avec comme paramètre le nombre de lignes et de colonnes. Par exemple, `Z=zeros(3,5)` crée une matrice de taille 3×5 contenant uniquement des 0.
- Pour créer une matrice constante on a la commande `ones` avec comme paramètre le nombre de lignes et de colonnes. Par exemple, `C=6*ones(3,5)` crée une matrice de taille 3×5 contenant uniquement des 6.
- Pour créer une matrice de nombres aléatoires uniformément distribués sur $[0, 1]$, on utilisera `rand` avec encore une fois comme paramètre le nombre de lignes et de colonnes. Par exemple, `R=rand(3,5)` crée une matrice de taille 3×5 contenant des nombres aléatoires compris entre 0 et 1.

Algèbre linéaire et matrices.
La multiplication de deux matrices n'a de sens que si leurs dimensions "internes" sont égales. En d'autres termes, `A*B` n'est valable que si le nombre de colonnes de `A` est égal au nombre de lignes de `B`. Si a_{ij} désigne l'élément situé sur la $i$ème ligne et la $j$ème colonne, alors la matrice `A*B` est formée des éléments

$$(AB)_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} \tag{1}$$

où n est le nombre de colonnes de `A` et aussi le nombre de lignes de `B`. Si l'on tape :

```
--> A = [1 2 3;4 5 6];
--> B = [7;8;9];
--> A*B
```

On obtient une matrice de dimension 2×1.
Le produit interne (produit scalaire) de deux vecteurs colonnes `x` et `y` (un vecteur colonne est une matrice n'ayant qu'une colonne) est le scalaire défini comme le produit `x'*y` ou, de manière équivalente `y'*x`. Par exemple :

```
--> x = [1;2],y = [3;4], x'*y
```

renvoie 11.
Enfin donnons quelques exemples de fonctions classiques d'algèbre linéaire sous Scilab : le calcul du rang d'une matrice se fait avec la commande `rank`, celui du déterminant avec `det` et celui de l'inverse avec `inv`. Les calculs de valeurs et vecteurs propres sont également possibles.

5 Le deux points

On peut utiliser le deux points de différentes manières dans Scilab (voir `help colon`). Il sert fondamentalement à construire un vecteur dont les valeurs des éléments sont incrémentées séquentiellement. Tapez par exemple

```
--> x = 3:9
```

pour obtenir

```
x =
   3 4 5 6 7 8 9
```

L'incrément est de 1 par défaut. Pour avoir un autre incrément, tapez des commandes telles que

```
--> x = 1:0.5:4
--> x = 6:-1:0
```

En examinant les résultats produits dans ces exemples, on comprend bien que l'incrément, autrement dit l'écart entre deux nombres consécutifs, est donné par le nombre du milieu. La syntaxe générale est donc la suivante : "Départ:Incrément:Arrivée".
Si l'on tape une expression de la forme `i:j:k` et qu'il n'existe pas $n \in \mathbb{N}$ tel que $k = i + jn$ alors Scilab renvoit la suite : i, $i + j$, $i + 2j$, ..., $i + nk$ où $n = \max\{m \in \mathbb{N} \,|\, i + mk \leq j\}$ autrement dit n est le plus grand entier tel que $i + nk$ soit inférieur à j.
Enfin il faut bien faire attention à la forme de distributivité de cet opérateur sur les opérations classiques. Par exemples :

```
--> 1+1:5
--> 1+(1:5)
```

ne renvoient pas le même résultat. Il faut donc s'efforcer de taper les commandes de manière non ambigüe. Si l'on a un doute, il vaut toujours mieux utiliser des parenthèses pour avoir la certitude que l'expression tapée correspond bien à ce que l'on recherche.

Extraction d'une sous-matrice ou d'un sous-vecteur. On peut aussi utiliser les deux points pour extraire une sous-matrice d'une matrice `A` ou d'un vecteur `V`.

`V(i:j:k)` extrait le sous vecteur de `V` composé des éléments situés en position : i, $i + j$, $i+2j$, ..., k. Par exemple, on peut extraire tous les nombres d'indices pairs dans un vecteur en procédant comme suit :

```
--> V=[10:-1:0];
--> V(1:2:length(V))
```

`A(i,:)` extrait la $i$ème ligne de `A`.

`A(:,j)` extrait la $j$ème colonne de `A`. On considère successivement toutes les lignes de `A` et on choisit le $j$ème élément de chaque ligne.

`A(:)` reforme la matrice `A` en un seul vecteur colonne en concaténant toutes les colonnes de `A`.

`A(:,j:k)` extrait la sous-matrice de `A` formée des colonnes j à k.

`A(j:k,:)` extrait la sous-matrice de `A` formée des lignes j à k.

`A(j:k,q:r)` extrait la sous-matrice de `A` formée des éléments situés dans les lignes j à k et dans les colonnes q à r.

Ces définitions peuvent s'étendre à des pas d'incrémentation des lignes et des colonnes différents de 1.

6 Scripts et fonctions

Scilab offre la possibilité d'exécuter les commandes les unes à la suite des autres dans la fenêtre principale, mais on peut également créer des programmes extérieurs que l'on exécute ensuite. Il existe deux types de programmes extérieurs : les scripts (fichiers *.sce*) et les fonctions (fichiers *.sci*).

Les scripts.
Un script est une suite d'instructions. Scilab lit le fichier ligne après ligne ce qui revient donc à exécuter les commandes à la suite, comme si elle étaient tapées. Par exemple :

```
--> v = [0:.01:10]*2*%pi;
--> fv = sin(v);
--> plot(fv)
```

peut être remplacer par un fichier *essai.sce* contenant les lignes :

```
v = [0:.01:10]*2*%pi;
fv = sin(v);
plot(fv)
```

et lancé grâce à :

```
--> exec('essai.sce');
```

Un fichier script utilise les variables stockées en mémoire par les instructions utilisées précédemment dans la fenêtre principale. Toute variable créée pendant l'exécution du script est encore accessible après l'exécution. Par exemple, dans le script ci-dessus la variable `s` est toujours utilisable.

Les fonctions.
Une fonction est une suite d'instructions prenant un ou plusieurs arguments en entrée et renvoyant un ou plusieurs arguments en sortie. On la définit à l'aide de l'instruction `function` et on l'enregistre dans un fichier *.sci*. Ainsi, si nous voulons écrire une fonction prenant un vecteur et élevant chacun de ses éléments au carré, on peut écrire dans un fichier dénommé *carre.sci* :

```
function resultat=carre(argument)
n=2;
resultat = argument.^n;
```

Pour utiliser la fonction, il faut d'abord la charger dans Scilab :

```
--> getf('carre.sci')
```

Puis, on s'en sert exactement comme d'une fonction classique de Scilab :

```
--> carre([1 2;3 4])
```

La variable `n` dans la fonction ci-dessus est créée pendant son exécution et est supprimée à la fin de l'appel de fonction.

Quel outil pour créer les fichiers ?
Puisqu'il ne s'agit que de fichiers contenant du texte, tous les éditeurs de texte peuvent être utilisés. Citons par exemple *notepad* sous Windows ou *emacs* sous Linux.
À partir de la version 3.0, Scilab possède un éditeur intégré qui présente certains avantages qui méritent d'être soulignés :

- reconnaissance de la syntaxe et des mots clefs
- insertion aisée de blocs de commentaires
- possibilité d'exécuter seulement une sélection de commandes dans le cas d'un script.

Pour lancer l'éditeur intégré il suffit de taper `scipad`.

7 Structures usuelle en programmation

Un peu de logique !
Dans la suite, une expression logique est simplement une formule qui une fois évaluée vaut 0 ou 1 ou encore vaut `F` (*false*) ou `T` (*true*).
Les opérateurs de comparaison sont souvent utiles pour définir des expressions logiques :
- `==` symbole d'égalité
- `>` et `>=` supérieur strictement et supérieur ou égal
- `<` et `<=` inférieur strictement et inférieur ou égal

Par exemple, si :

```
--> a=5;
--> b=5;
```

En tapant `a>b` on obtient `F` et en tapant `a==b`, on obtient alors `T`.
Il existe également de nombreuses fonction de tests renvoyant `F` et `T`. Leur nom commence généralement par "is". Par exemple, `isnan`, `isdef`, `isequal`, `isempty`...

Enfin sous Scilab plusieurs opérateurs sont définis permettant de manipuler des expressions logiques et de les combiner entre elles :
- `&` correspond à *et*
- `|` correspond à *ou*
- `~` correspond à *non*, c'est le symbole de négation

Par exemple :

```
--> (~isnan(a))&(b>=a)
```

est vrai si b est supérieur ou égal à `a` **et** si `a` est différent de `NAN`.

Structure "*si, alors, sinon*"
Il s'agit de la structure de test la plus classique. Elle se définit sous Scilab de la manière suivante :

```
if condition
    action1
else
    action2
end
```

`condition` est l'évaluation d'une phrase logique et `action1` et `action2` sont des suites de commandes. Par exemple :

```
if a<=0
    b=0
else
    b=log(a)
end
```

La syntaxe ci-dessus est valable dans un fichier script ou fonction. En ligne de commande, il faut tout mettre sur une seule ligne donc la formule équivalente serait : `if a<=0, b=0, else b=log(a), end`

Structure "*Pour* x *allant de* a *à* b"
Cette structure permet de répéter une certaine opération (dépendant d'un indice ou non), un certain nombre de fois. La syntaxe est la suivante :

```
for k=D:F
    action
end
```

Cette boucle exécute $F - D + 1$ fois les commandes représentées par `action`. Par exemple :

```
T(1)=1;
T(2)=1;
for k=3:25
    T(k)=T(k-1)+T(k-2);
end
```

calcule les 26 premiers termes de la suite de Fibonacci.

Structure "*Tant que...*"
Si l'on ne sait pas *a priori* combien d'itérations seront nécessaires dans une boucle, on peut aussi utiliser une boucle *tant que*. La syntaxe est la suivante :

```
while condition
    action
end
```

Où `condition` et `action` sont définis comme précédemment. Par exemple :

```
k=0;
while ~isinf(exp(k))
    k=k+1;
end
```

calcule le plus petit nombre entier dont l'exponentiel dans Scilab prend numériquement la valeur `%inf` (constante pour ∞).

Boucles et vectorisation
Comme nous l'avons déjà souligné, Scilab (tout comme Matlab) a été créé pour manipuler des vecteurs et des matrices. A ce titre, toutes les opérations sur les vecteurs sont très bien optimisées et assez rapides. Si l'on souhaite faire une boucle pour mener un calcul, il est toujours intéressant de se demander si l'on ne peut pas faire la même opération par de simples manipulations sur les vecteurs. Par exemple, si on souhaite calculer le vecteur des cosinus des dix millions premiers entiers. A l'aide d'une boucle on pourrait écrire :

```
tic;
for k=1:10000000
    cos(k);
end;
toc
```

Les instructions `tic` et `toc` permettent de mesurer le temps écoulé entre les deux instructions. De manière équivalente on pourrait également écrire l'opération vectorielle suivante :

```
tic;
cos(1:10000000)
toc
```

En exécutant ces deux groupes de commandes sur un *Pentium IV 2.60GHz* on observe que le premier groupe d'instructions prend 51 secondes pour s'exécuter contre 1 seconde pour le deuxième !

8 Variables complexes

Comme nous l'avons déjà vu, le nombre complexe $\sqrt{-1}$ est prédéfini dans Scilab et stocké dans la variable `%i`. Notez la manière dont est affichée une variable complexe. Entrez par exemple `z1 =1+2*%i`, `z2 = 2+1.5*%i`. Comme `%i` est considéré comme une variable, il faut utiliser le signe `*` de la multiplication, sinon apparaîtra un message d'erreur. Scilab ne fait pas de différence entre les variables réelles et une variable complexe (si ce n'est à la mise en mémoire). Les variables peuvent être ajoutées, soustraites, multipliées et même divisées. Tapez par exemple `x = 2, z = 4 + 5*%i` et `z/x`. Les parties réelle et imaginaire de `z` sont toutes deux divisées par `x`. Scilab traite simplement `x` comme une variable dont la partie imaginaire est nulle. Une variable complexe dont la partie

imaginaire est nulle est traitée comme une variable réelle. Soustrayez `2*%i` de `z1` et affichez le résultat.

Scilab contient plusieurs fonctions prédéfinies pour manipuler les nombres complexes. Par exemple, `real(z)` extrait la partie réelle du nombre complexe `z`. Tapez

```
--> z = 2+1.5*%i; real(z)
```

pour obtenir le résultat

```
ans =
     2
```

De même, `imag(z)` extrait la partie imaginaire du nombre complexe `z`. La fonction `abs(z)` calcule le module du nombre complexe `z`. Tapez par exemple

```
--> z = 2+2*%i;
--> r = abs(z)
--> theta = imag(log(z)) // calcul de l'argument du nombre complexe
--> z = r*exp(%i*theta)
```

La dernière commande montre comment retrouver le nombre complexe original à partir de son module et de sa phase.
Une autre fonction utile, `conj(z)`, retourne le complexe conjugué du nombre complexe `z`. Si `z = x+%i*y` où `x` et `y` sont réels, alors `conj(z)` est égal à `x-%i*y`. Vérifiez ceci pour différents nombres complexes en utilisant la fonction `conj(z)`.

Transformée de Fourier
La transformée de Fourier est une transformée complexe permettant de donner une représentation fréquentielle d'un signal. Elle est très populaire en traitement du signal en particulier en raison de la rapidité de calcul de l'algorithme FFT (Fast Fourier Transform). La fonction `fft(V,-1)` permet de calculer la transformée de Fourier du signal `V` et `fft(V,1)` sa transformée inverse.

```
--> fv = sin([0:.1:10]*2*%pi);
--> tv=fft(fv,-1);
--> plot(abs(tv))
```