Annexe : quelques fonctions scilab utiles

présentation

cette annexe donne une liste des fonctions *scilab* que vous pourrez employer lors des séances de TP. Les fonctions propres à scilab sont représentées en ***italique gras***, les fonctions écrites par l'enseignant sont représentées en *italique tout court*. Ayez le réflexe d'employer l'aide de scilab: ***help*** <nom de fonction> fournit de l'aide sur la fonction concernée, par exemple ***help poly*** fournit l'aide sur la fonction ***poly*** de *scilab*

les exemples correspondant a cette annexe sont regroupés dans le fichier ***annexe_scilab.sce****, et sont numérotés de la même façon que dans ce fascicule, pour que vous puissiez facilement les adapter à vos besoins(à coups de copier-coller).*

1 polynômes et fractions rationnelles

1.1 *saisie de polynômes :fonction poly*

1.1.1 saisie en indiquant les racines(choix par défaut)

p=***poly***(0,"p"); correspond à la saisie du polynôme
1-qui emploie comme variable représentative la chaîne de caractères "p"
2-qui a une seule racine en 0
soit encore au polynôme (p-0), il est donc logique de l'appeler *p*

R=***poly***([1,2],"p"); correspond à la saisie du polynôme R
1-qui emploie comme variable représentative la chaîne de caractères "p"
2-qui a deux racines en p=1 et en p=2
soit encore R=(p-1).(p-2)

1.1.2 saisie en indiquant les coefficients

p=***poly***([0,1],"p","coeffs"); correspond à la saisie du polynôme *p*
1-qui emploie comme variable représentative la chaîne de caractères "p"
2-dont les coefficients en puissance croissante sont *0* et *1*
soit encore au polynôme $0.p^0+1.p^1=p$, il est donc logique de l'appeler *p*

R=***poly***([1,2],"p","coeffs"); correspond à la saisie du polynôme *R*
1-qui emploie comme variable représentative la chaîne de caractères "p"
2-dont les coefficients en puissance croissante sont *1* et *2*
soit encore au polynôme $R=1.p^0+2.p^1=1+2.p$

1.2 calculs sur les polynômes et ou fractions rationnelles

scilab connaît l'arithmétique sur les polynômes et les fractions rationnelles. Vous pouvez employer des expressions du genre:
x=(p+1)*(p+7) / (p+8)^3

1.3 caractéristiques et exploitation des polynômes et fractions

numerateur et dénominateur d'une fraction rationnelle ***numer***, ***denom***

les instructions
n=***numer***(x),
d=***denom***(x)
renvoient les numérateur et dénominateur de la fraction rationnelle *x dans les* variables *n* et *d*

1.3.1 racines => fonction *roots*

R=***roots***(*p*); renvoie un vecteur *R* contenant les racines du polynôme *p*

1.3.2 coefficients => fonction coeffs

C=***coeff***(p) renvoie le vecteur *C* des coefficients de *p*, par puissance croissantes
C=***coeff***(p,3) renvoie le coeff en puissance 3 du polynôme *p*

1.3.3 évaluation=> fonction *horner,hornerij*

val_p=***horner***(p,2) évalue la valeur de p(2).
val_p=***horner***(p,[2,1]) évalue le vecteur des valeurs [p(2),p(1)].
la fonction ***horner*** *peut également être employée pour effectuer des changements de variables.* Je déconseille son emploi dans ce cas*: elle a tendance à effectuer des simplifications abusives, ou pas de simplifications quand il le faudrait.*

1.4 Exploitation des fonctions de transfert codées sous forme de listes

Les fonctions de transfert en z, codées directement sous forme de fractions rationnelles sont difficilement exploitables, dès que leur degré devient élevé.
Pour cette raison tous les systèmes qui ont pour but de coder des filtres numériques les codent sous forme de 'listes' de fonction de transfert d'ordre 1 ou 2, en cascade ou en parallèle.
Cette partie explique l'utilisation et les fonctions permettant de décrire et d'exploiter de telles listes.

1.4.1 Création d'une liste, la fonction list() de scilab

une liste scilab est un ensemble d'éléments qui peuvent être de nature différentes, indexés.
Pour créer une liste vide, on emploie la fonction ***list().***
Exemple :

```
l=list();  // crée une liste l, vide
on peut ensuite insérer des éléments de la façon suivante
l(3)=25; // la liste l contient un seul élément, d'indice 3, qui vaut 25
l(7)="salut les gars" // on ajoute un autre element, d'indice 7;
indice_max=length(l) // plus grand indice de l , indice_max vaut 7
indices_de_l=definedfields(l)// indices definis de l, indices_de_l vaut [3,7]
// boucle for en scilab, affichant les differents elements de l
for i=definedfields(l),
 element_i=l(i); // on recupere l'element d'indice i (d'abord 3, puis 7)
 disp("l("+string(i)+")="+string(element_i));// et on l'affiche a l ecran
end
```

1.4.2 création d'une liste de fonctions de transfert en z

pour créer une liste de fonctions de transfert, on pourra s'inspirer du code suivant, qui correspond à la description de $F(z)=\underbrace{\frac{(z+0.9)}{(z+0.91)}}_{cel\_z(1)}.\underbrace{\frac{(z-0.9)}{(z-0.91)}}_{cel\_z(2)}$ *, sous forme factorisée*

```
z=poly(0,"z");
F_de_z=list(); // creation d'une liste F_de_z contenant les expressions en z
F_de_z(1)= (z+0.9)/(z+0.91); // element d'indice 1
```

F_de_z(2)= (z-0.9)/(z-0.91); // element d'indice 2

1.4.3 changements de variable et evaluation: fonctions *hornerij, horner11_inv*

la fonction hornerij est équivalente à la fonction ***horner.*** *Elle s'applique sur des listes, ou des fractions rationnelles (d'ordre 1 à 4 uniquement). Contrairement à la fonction* ***horner*** *de scilab, hornerij n'effectue pas de simplifications abusives des résultats.*

1.4.3.1 Changement de variable a l'aide de hornerij, horner11_inv

la fonction *hornerij* permet de réaliser rapidement les changements de plan (z-> w-> z-1) sur des fractions rationnelles (ou des listes de fractions rationnelles d'ordre 1 à 2)

```
w=poly(0,"w");
z_de_w=(1+w)/(1-w); // expression z(w) de z en fonction de w
w_de_z=horner11_inv(z_de_w,"z"); // renvoie w(z), en inversant z(w)
```

1.4.3.2 normalisation , la fonction normalize

on peut 'normaliser' les fractions rationnelles renvoyées par *hornerij* de 2 façons
1-soit en appelant *hornerij* avec une 3ème entrée, *type_norm*
2- soit en appelant la fonction *normalize*(fraction,*type_norm*)
dans tous les cas type_norm est une chaîne de caractères indiquant le type de normalisation employé

type_norm="hd" (higher denom) <=> coeff de plus haut degré du dénom=1
type_norm="ld" (lower denom) <=> coeff de plus bas degré du dénom=1
type_norm="hn" (higher numer) <=> coeff de plus haut degré du numer=1
type_norm="ln" (lower numer) <=> coeff de plus bas degré du numer=1

classiquement,
les fractions rationnelles en *z,p,w* sont normalisées par rapport au coeff de plus haut degré du dénominateur("hd")
Les fractions rationnelles en z-1 sont normalisées par rapport au coeff de plus bas degré du dénominateur ("ld")
sous scilab, on écrira donc

```
z_de_w=normalize(z_de_w,"hd");// exp. of w=> normalize higher denom coeff
w_de_z=normalize(w_de_z,"hd")// exp. of z=> normalize higher denom coeff
```

soit à présent z_1 le polynôme représentant la variable z^{-1} , notée *q*
on pourra définir w_de_z_1 de la façon suivante :

```
z_1=poly(0,"q");
z_de_z_1=1/z_1;
w_de_z_1=hornerij(w_de_z,z_de_z_1,"ld"); // coeff de plus bas degré denom=1
z_1_de_w= horner11_inv( w_de_z_1,"w"); // calcul de z^-1(w)
z_1_de_w=normalize( z_1_de_w,"hd"); // fct de w=> coeff plus haut degré den.=1
```

1.4.3.3 changement de variables, évaluation, sur des listes

les fonctions *hornerij*, *normalize*,*horner11_inv* fonctionnent également sur des listes de fractions rationnelles, et permettent ainsi d'effectuer des changements de variables sur des expressions factorisées ...

exemple : calcul de la transformation bilinéaire $F(w)$ de la fonction de transfert $F(z)$, en gardant $F(w)$ sous forme factorisée

F_de_w=*hornerij*(F_de_z,z_de_w,"hd");// creation de la liste F(w) =Fz(z(w))
F_de_w(1) // affichage premier élément de F(w)
F_de_w(2) // affichage deuxième élément de F(w)
// evaluation de F(z) en z0=$i.\pi/4$, et de F(w0) sur la valeur correspondante de w
z0=***%i*** * ***%pi*** /4 ;// creation de z0=i.π/4
w0=*hornerij*(z0,w_de_z) // w0=w(z0)
F_de_z0=*hornerij*(F_de_z,z0); // evaluation de F(z), element par element
F_de_w0=*hornerij*(F_de_w,w0); // evaluation de F(w0), element par element

1.4.4 gestions des sommes et produits : *get_as_product, get_as_sum*

lorsque les listes ont plusieurs éléments, il est souvent nécessaire d'évaluer la somme ou le produit de tous leurs éléments, c'est le rôle des fonctions *get_as_sum* et *get_as_product*.
Si par exemple on cherche la fonction de transfert F(z) globale= produit de tous les élements, on écrira
F_de_z_globale=*get_as_product*(F_de_z)

noter que dans ce cas F_de_z_globale est une fraction rationnelle en z, ce que l'on peu vérifier sous scilab en écrivant
typeof(F_de_z)
typeof(F_de_z_globale)

on peut également évaluer la valeur de F(z) en z0 en écrivant :
F_de_z0_globale=***horner***(F_de_z_globale,z0)
noter que l'on a employé la fonction scilab ***horner***, à cause de l'ordre de F(z) qui peut être élevé (*hornerij* ne fonctionne que pour des ordres <=4)
de toute façon, cette technique est à proscrire pour des fonctions de transfert d'ordre élevé: les arrondis de coefficients dans F_de_z_globale font que cette fonction représente mal le produit de tous les termes!...
La bonne technique consiste à évaluer chacun des éléments en z=z0, puis à effectuer le produit des valeurs numériques obtenues :
F_de_z0=*hornerij*(F_de_z,z0); // evaluation de F(z), element par element
F_de_z0_globale=*get_as_product*(F_de_z0) // produit des valeurs numériques

1.4.5 évaluation de la réponse fréquentielle de listes de fraction rationnelles

1.4.5.1 création d'un vecteur de points : fonctions linspace, logspace

On suppose que l'on veut tracer la réponse fréquentielle de F(z) sur 100 points régulièrement répartis, entre f=0 et f= 400.
on pourra alors créer un vecteur fi des fréquences correspondantes, au moyen de la fonction ***linspace*** de scilab, de la façon suivante
f0=0;f1=400;nb_points=100;
fi=***linspace***(f0,f1,nb_points); // fi=vecteur ligne de 100 points entre f0 et f1

fi=fi.'; // on transpose fi, pour en faire un vecteur colonne, plus pratique pour les tracés

Si on avait préféré des points régulièrement répartis sur une échelle logarithmique,il aurait fallu employer la fonction **logspace** de scilab (attention, su une échelle logarithmique, on ne peut pas représenter f0=0=> on choisit f0=0.1 par exemple:
f0=0.1;f1=400;nb_points=100;
fi=***logspace***(***log10***(f0),***log10***(f1),nb_points); // fi=vecteur ligne de 100 points entre f0 //et f1, régulièrement répartis sur échelle logarithmique
fi=fi.'; // on transpose fi, pour en faire un vecteur colonne, plus pratique pour les tracés
size(fi) // on affiche la taille, par curiosité

1.4.6 réponse fréquentielle de F(z)

la réponse fréquentielle de F(z), à la fréquence fi, est donnée par :

$$F\left(z=\exp^{i2\pi\frac{fi}{f_e}}\right) \quad ,$$

on donne ci après un exemple d'évaluation de cette quantité, en supposant fe=1000;

fe=1000;
zi=***exp***(***%i****2****%pi****fi/fe); // points correspondants zi= $\exp^{i2\pi\frac{fi}{f_e}}$
F_de_zi=*hornerij*(F_de_z,zi); // evaluation de la liste F_de_zi= liste de vecteurs
F_de_zi_glob=*get_as_product*(F_de_zi); // F_de_zi_glob=vecteur $F\left(z=\exp^{i2\pi\frac{fi}{f_e}}\right)$

1.4.7 réponse fréquentielle de F(w)

la réponse fréquentielle de F(w), à la fréquence *fi*, est donnée par :

$$F\left(w=i.tan\left(\pi\frac{fi}{f_e}\right)\right) \quad ,$$

on donne ci après un exemple d'évaluation de cette quantité, en supposant fe=1000;
fe=1000;
vi=***tan***(***%pi****fi/fe); // pseudo pulsations vi
wi=i*vi; // valeurs wi correspondantes
F_de_wi=*hornerij*(F_de_w,wi); // evaluation de la liste F_de_wi= liste de vecteurs
F_de_wi_glob=*get_as_product*(F_de_wi); // F_de_wi_glob=vecteur $F(wi=i.vi)$

1.4.8 évaluation des module et argument

il existe une fonction scilab, ***dbphi***, qui permet d'évaluer les module en décibels, et argument en degrés, d'une réponse fréquentielle. Toutefois cette fonction est très désagréable à utiliser, pour les 2 raisons suivantes
1- elle calcule l'argument 'modulo 360°', sans gérer les sauts de phase
2- si le module réel est égal à zéro , elle génère une erreur
pour cette raison, on a écrit une fonction *get_module_arg*, qui a la même fonctionnalités, sans souffrir de ces 2 limitations
exemple :
module_reel=***abs***(F_de_zi_glob); // calcul du module reel de F(z)
[module_db,arg_degres]=*get_module_arg*(F_de_zi_glob); // calcul mod en db et arg en degres
on aurait pu écrire la même chose en raisonnant dans le plan w

1.4.9 calcul des réponses impulsionnelles : fonction *impulse_Fz*

la fonction *impulse_Fz* permet de calculer la(es) réponse(s) impulsionnelle(s) du système codé dans la liste F(z). On doit lui préciser

1 le nombre d'échantillons NB_ECH sur lequel on veut calculer cette réponse

2 le type de fonction de transfert représentée par F(z)

"cascade" ,si F(z) représente un produit de termes

"paralell", si F(z) représente une somme de termes

"matrix", si F(z) représente des termes disjoints,

dans ce dernier cas, la sortie sera une liste correspondant

à chacune des réponses impulsionnelles

exemple :

NB_ECHS=1000;

fn=*impulse_Fz*(F_de_z,NB_ECH,"cascade") // rep umpulsionnelle, vecteur ligne

calcule les 100 premiers échantillons de la réponse impulsionnelle de F(z), sous la forme d'un vecteur ligne. F(z) est suposée être le produit des cellules Fi ("cascade")

1.4.10 réponse à une entrée quelconque: fonction *filter_Fz*

la fonction *filter_Fz* permet de calculer la réponse de F(z) à une séquence d'entrée en quelconque. Sa syntaxe ressemble énormément à celle de impulse_Fz.

Exemple : réponse à l'échelon de F(z)

NB_ECHS=30;

en=***ones***(1,NB_ECHS); // en vecteur ligne ne contenant que des 1=> echelon

sn=*filter_Fz*(F_de_z,en,"cascade"); // fn =rep a l'echelon de F(z), vecteur ligne

1.4.1 calcul des normes de F(z): fonction *norme_Fz*

la fonction *norme_Fz* permet de calculer la norme (1,2, ou H_∞) d'une fonction codée sous forme de liste. Pour cela elle emploie les techniques approximatives décrites dans le travail préparatoire, à savoir l'approximation des normes sur un nombre très grand d'échantillons..

exemple :calcul des normes 1,2, et H_∞ de F(z)

type_norme=1;type_de_liste="cascade";NB_ECH=1000;

norme1_Fz=*norme_Fz*(F_de_z,type_de_liste,type_norme,NB_ECH);

type_norme=2;type_de_liste="cascade";NB_ECH=1000;

norme2_Fz=*norme_Fz*(F_de_z,type_de_liste,type_norme,NB_ECH);

type_norme=***%inf***;type_de_liste="cascade";NB_ECH=1000;

normeHinf_Fz=*norme_Fz*(F_de_z,type_de_liste,type_norme,NB_ECH);

2 graphiques scilab

2.1 code standard de lancement d'un graphique

le code standard de lancement d'un graphique sous scilab comprend 3 étapes

1- la création ou l'activation d'une fenêtre graphique, fonction ***xset()***

exemple création ou activation fenêtre de numéro 0

num_figure=0;***xset***("window",num_figure);

2- l'effacement (optionnel) des tracés précédemment effectués sur cette fenêtre

exemple: effacement des données sur la fenêtre de numéro *num_figure*

xbasc(num_figure*); // xbasc efface tout sur la fenêtre num_figure*

3- la sélection (optionnelle) d'une sous-zone de tracé, fonction ***subplot***

exemple :

nb_case_vert=2;nb_cases_hori=3;num_case_active=5;
subplot(nb_case_vert, nb_cases_hori, num_case_active);
cette fonction indique que l'on a partitionné (*mentalement*) la fenêtre graphique en 2 cases verticales, 3 cases horizontales, et que les graphiques seront tracés dans la case numéro 5, conformément au tableau suivant.

case 1	case 2	case 3
case 4	case 5(zone de tracé)	case 6

Par défaut (si on n'emploie pas ***subplot*** pour indiquer une zone de tracé) , la fenêtre graphique est partitionnée en une seule case <=> ***subplot***(1,1,1)

2.2 tracé de courbes : fonction *plot2d*

La fonction ***plot2d*** de scilab permet de tracer une (ou plusieurs) courbes dans la zone graphique courante (sélectionnée avec ***subplot***()). Elle offre de très nombreuses possibilités, et nous nous contenterons d'indiquer les plus usuelles... écrire ***help plot2d*** sous scilab pour aide plus complète et exemples

2.2.1 tracé d'un vecteur *y* en fonction de ses indices

pour tracer un vecteur *y (ligne ou colonne)* en fonction de ses indices, il suffit d'écrire : ***plot2d***(y)

2.2.2 tracé d'un vecteur y (colonne) en fonction d'un vecteur x colonne

=> écrire ***plot2d***(x,y)

2.2.3 tracé d'un ensemble de vecteurs colonne y1,yn en fonction du même vecteur colonne x

// former la matrice Y = juxtaposition des colonnes, puis tracer avec plot2d
Y=[y1,y2,...,yn];***plot2d***(x,Y)

2.2.4 spécification d'échelles: linéaire ou logarithmique

on peut envoyer en premier paramètres de ***plot2d*** une chaine de caractères spécifiant les échelles pour les axes x et y : logarithmiques => caractère *l*, normale => caractère *n*
exemple :
plot2d("nl",x,Y); // échelle normale en *x*, et logarithmique en *y*

2.3 personnalisation des graphiques

2.3.1 spécifier les couleurs

les couleurs utilisées par scilab pour le tracé sont codées sous forme de numéros, pas très clairs : 1-noir,2-bleu,3-je ne sais plus...
Par défaut, pour un tracé multi-colonnes, la colonne 1 est tracée en couleur 1, la colonne 2 en couleur 2, etc...
on peut spécifier les couleurs en ajoutant un vecteur ligne en 3 ème paramètre, spécifiant les couleurs pour chaque colonne
plot2d(x,[y1,y2,y3],[1,3,2]); // trace y1,y2,y3 en couleurs 1,3,2

2.3.1 *ajouter une légende* : *legends*

la fonction ***legends*** permet d'ajouter une légende , en spécifiant la couleur de chaque

légende
On peut également spécifier la position de la légende avec une chaîne de 2 caractères "ll" (lower-left),"ul"(up-left),"lr"(lower-right),"ur"(up-right)
legends(["salut','les","gars"],[3,1,2],"lr"); // legende en couleurs 3,1,2,en bas à droite

2.3.2 *quadrillage et titre* : *xgrid* et *xtitle*

xgrid permet d'ajouter un quadrillage en spécifiant la couleur
xgrid(2) => quadrillage en couleur 2 (bleu)
xtitle permet d'ajouter un titre + des titres en *x* et *y*
exemple :
xtitle("ceci est une courbe"," ceci est l axe des x"," ceci est l axe des y");

3 le guide de survie de scilab: syntaxe, principales fonctionnalités

ce petit guide a juste pour but de vous présenter les principales fonctionnalités de scilab, vous en apprendrez beaucoup plus en lançant les démos...

3.1 vecteurs et matrices

3.1.1 créer et concaténer des vecteurs et matrices

La virgule permet d'effectuer des juxtapositions colonne.
Le point-virgule permet d'effectuer des juxtapositions ligne
Les crochets [] indiquent une définition de matrice
exemples :

M=[1,2,3;4,5,6;7,8,9]; //crée la matrice $M=\begin{bmatrix}1 & 2 & 3\\ 4 & 5 & 6\end{bmatrix}$

x=M(2,3) // range la composante M $M_{23}=M_{\text{ligne 2, colonne 3}}$ *dans x*

X=[7,8,9]; //crée le vecteur ligne $X=\begin{bmatrix}7 & 8 & 9\end{bmatrix}$

Y=[7;8;9]; //crée le vecteur colonne $Y=\begin{bmatrix}7\\ 8\\ 9\end{bmatrix}$

N=M.' // range la transposée de M dans N => $N=\begin{bmatrix}1 & 4\\ 2 & 5\\ 3 & 6\end{bmatrix}$

V=[N,Y] // V=juxtaposition colonne de N et Y => $V=\begin{bmatrix}1 & 4 & 7\\ 2 & 5 & 8\\ 3 & 6 & 9\end{bmatrix}$

W=[M;X] // W=juxtaposition ligne de M et X => $W=\begin{bmatrix}1 & 2 & 3\\ 4 & 5 & 6\\ 7 & 8 & 9\end{bmatrix}$

*R=**ones**(M) // R = matrice de 1, de même taille que M :* $R=\begin{bmatrix}1 & 1 & 1\\ 1 & 1 & 1\end{bmatrix}$

*R=**zeros**(M) // R = matrice de 0, de même taille que M :* $R=\begin{bmatrix}0 & 0 & 0\\ 0 & 0 & 0\end{bmatrix}$

*R=**eye**(W) // R = matrice identité, de même taille que W :* $R=\begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{bmatrix}$

R=10:2:16 // vecteur ligne de 10 a 16 par pas de 2 : $R=\begin{bmatrix}10 & 12 & 14 & 16\end{bmatrix}$
*R=**linspace**(10,16,4)// vecteur ligne de 10 a 16 en 4 pas :* $R=\begin{bmatrix}10 & 12 & 14 & 16\end{bmatrix}$

3.1.2 extraire ou affecter des composantes ou des sous matrices

- la syntaxe pour extraire une partie d'une matrice *M* est du type
  R=M(indices lignes, indices colonnes)
-la même syntaxe peut être employée pour affecter une partie d'une matrice,
  M(indices lignes, indices colonnes)=*R*
-les dimensions de *M*(indices lignes, indices colonnes) et *R* doivent être compatibles
-le caractère ***:*** indique tous les indices possibles
-le caractère ***$*** indique le dernier indice possible
exemples

soit $M=\begin{bmatrix}1 & 2 & 3\\ 4 & 5 & 6\end{bmatrix}$

R=M(:,3) // R=M(toutes les lignes, colonne 3), $R=\begin{bmatrix}3\\ 6\end{bmatrix}$

R=M(2,[3,1]) // R=M(ligne 2, colonnes 3,1), $R=\begin{bmatrix}3 & 1\\ 6 & 4\end{bmatrix}$

M([1,2],2)=[25;26] // M(lignes 1 et 2,colonne 2)= $\begin{bmatrix}25\\ 26\end{bmatrix}$ <=> $M=\begin{bmatrix}1 & 25 & 3\\ 4 & 26 & 6\end{bmatrix}$

R=M($,:) // R= M(derniere ligne, toutes les colonnes) $R=\begin{bmatrix}4 & 5 & 6\end{bmatrix}$

R=M(1,2:$) // R=M(ligne 1, colonnes 2 à dernière) $R=\begin{bmatrix}2 & 3\end{bmatrix}$

M(2,[1,2])=[25;26]// M(ligne 2,colonne 1,2)= $\begin{bmatrix}25\\ 26\end{bmatrix}$ *// les dimensions sont non-compatibles=> scilab génère une erreur :* ***submatrix incorrectly defined!..***

3.1.3 stockage et affectation efficace des matrices

les matrices sont codées en interne par scilab sous la forme de tableaux à une seule dimension. Connaissant le nombre de lignes et de colonnes, scilab s'arrange ensuite à retrouver la composante (i,j). Les tableaux sont stockés colonne par colonne de la façon suivante :

$N=\begin{bmatrix}1 & 4\\ 2 & 5\\ 3 & 6\end{bmatrix}$ *est codée dans un tableau* $TN=\begin{bmatrix}1\\ 2\\ 3\\ 4\\ 5\\ 6\end{bmatrix}$ *(la variable TN n'existe pas sous scilab!.. On a inventé le nom TN pour expliquer le fonctionnement!...)*

-on peut accéder aux composantes de *N* en spécifiant les 2 indices ligne *i*, colonne *j*, comme vu précédemment.
- on peut aussi ne spécifier qu'un seul indice *k. Scilab* comprend alors qu'il s'agit de l'indice dans le tableau de stockage *TN à une dimension*

exemple :
N22=N(2,2) // N22 = *N* ligne 2, colonne 2
est équivalent à
N22=N(5) // N22 = composante numéro 5 du tableau de stockage *TN*
- cette syntaxe,employée conjointement avec la fonction ***find*** (paragraphe suivant), est redoutablement efficace pour traiter des parties de matrices...
Par exemple si on écrit
N(3:$)=***zeros***(N(3:$)), on annulera les composantes (3 à dernière), du tableau de

stockage *TN* => $TN=\begin{bmatrix}1\\2\\0\\0\\0\\0\end{bmatrix}$ => $N=\begin{bmatrix}1&0\\2&0\\0&0\end{bmatrix}$

d'une manière générale,
-si on indique un seul indice dans une matrice ou un vecteur, scilab comprend que l'on raisonne sur le tableau de stockage associé
TN = *N*(1:$) , ou *TN*=*N*(:) , permet d'accéder au tableau de stockage lié à *N*
- si on indique 2 indices *i* et *j*, scilab comprend que l'on raisonne sur les lignes et colonnes

3.1.4 *gestion efficace des matrices, size length et find*

[m,n]=***size***(*N*) renvoie
le nombre de lignes de *N* dans *m* (m=3)
le nombre de colonnes de *N* dans *n*.(n=2)
l=***length***(*N*) renvoie
le nombre total de composantes de *N* dans *l (l=6)*
- la fonction ***find*** permet de trouver l'ensemble des composantes d'une matrice qui vérifient une condition logique

exemple :soit $N=\begin{bmatrix}1&4\\2&5\\3&6\end{bmatrix}$ *et le tableau associé* $TN=\begin{bmatrix}1\\2\\3\\4\\5\\6\end{bmatrix}$

- pour trouver l'ensemble des indices *i* et *j* pour lesquels *N*(*i*,*j*)>=3, on écrira :
[i,j]=***find***(*N*>=3)
i et j sont alors les vecteurs (lignes) d'indices tels que N(i,j)>=3
donc, $i=\begin{bmatrix}3&1&2&3\end{bmatrix}$, car $N(3,1)\geq 3,\ N(1,2)\geq 3,\ N(2,2)\geq 3,\ N(3,2)\geq 3$
$j=\begin{bmatrix}1&2&2&2\end{bmatrix}$

- par contre cette syntaxe est peu pratique, lorsque l'objectif consiste à faire subir un traitement particulier aux composantes N(i,j) correspondantes...
si par exemple on veut ranger des zéros dans les composantes *N*(*i*,*j*) sélectionnées, on ne peut pas écrire :*N*(*i*,*j*)=0**N*(*i*,*j*) . cette instruction rangera des zéros pour chacune des lignes indiquées par *i*, dans toutes les colonnes indiquées par *j* => *on aura des*

zéros dans toutes les composantes de N!...

- ce problème est résolu en travaillant directement sur le tableau de stockage associé à N, en écrivant:
k=***find***(N>=3) // indices *k* tels que *TN*(*k*)>=3
N(k)=0*N(k) // on range des zéros dans les composantes correspondantes
- *k* est le vecteur d'indices tel que *TN*(*k*)>=3 => $k = \begin{bmatrix} 3 & 4 & 5 & 6 \end{bmatrix}$

- en sortie de la 2ème instruction, on aura donc $TN = \begin{bmatrix} 1 \\ 2 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$, soit $N = \begin{bmatrix} 1 & 0 \\ 2 & 0 \\ 0 & 0 \end{bmatrix}$

3.1.5 opérations sur les matrices, transposition, inversion

scilab est capable d'effectuer les opérations classiques sur les vecteurs et matrices +,-,*
- les opérations terme a terme sont précédées d'un point
exemples :
Z=M*N // multiplication matricielle : $Z_{ij} = \sum_k M_{ik} . N_{kj}$
Z=M.*M // multiplication terme à terme : $Z_{ij} = M_{ij} . M_{ij}$
l'opérateur **.'** transpose une matrice : T=M**.'** // T=transposée de M
l'opérateur **'** transpose et conjugue : T=M' // T =transposée conjuguée de M
l'inverse d'une matrice carrée est obtenue avec la fonction ***inv***
les fonctions : ***log***,***log10***,***cos***,***sin***,***asin***,***atan***, ***exp***,... sont appliquées terme à terme ..
T=cos(log(M)) // $T_{ij} = \cos(\ln(M_{ij}))$

4 les structures de programme, variables prédéfinies, fonctions

4.1 structures de programme

scilab dispose des structures classiques de programme :
if then else end,
for-end,
while end,
select-case-end
employer ***help if, help for*** , ***help while help select*** pour plus d'informations sur leur syntaxe...

4.1 *variables prédéfinies* : caractère %

Les variables prédéfinies (et protégées => on ne peut pas changer leur valeur) de scilab on un nom qui commence par le caractère ***%***
La plupart des variables classiquement employées en mathématiques (*i*, π , $e = \exp(1)$,etc...), sont prédéfinies sous scilab , (sous le nom ***%i***, ***%pi***, ***%e***)

4.2 les fonctions scilab : function endfunction

de même que tous les langages informatiques, scilab dispose de fonctions...

la syntaxe de déclaration d'une fonction est la suivante
function [sortie_1, sortie_2,...]=nom_de_la_fonction(entree_1, entree_2,...)
endfunction
-une fonction a pour rôle de calculer les sorties en fonction des entrées, ou d'exécuter des tâches qui dépendent des entrées.
- les fonctions scilab ont un nombre d'entrées et de sortie de sorties variable
que l'on peut retrouver avec l'instruction scilab ***argn***...
- les entrées sont transmises 'par valeur', si vous changez les entrées à l'intérieur d'une fonctions, les variables scilab correspondantes ne seront pas modifiées.
Exemple :
voici une fonction '*bidon*' qui effectue les tâches suivantes
- dans tous le cas elle affiche à l'écran le nombre d'entrées et de sorties
- si on a 0 entrees,0 sorties, on sort
- si on a 1 entree, 1 sortie => sortie=2*entree
-si on a 2 entrees, 2 sorties => sortie1 =2*entree_1, sortie 2= 2*entree_2
- dans les autres cas, elle génère un message d'erreur expliquant à l'utilisateur qu'il n'a pas respecté les conditions normales d'utilisation de la fonction

```
function [s1,s2]=bidon(e1,e2)
  [lhs,rhs]=argn(0) // lhs=nombre de sorties , rhs = nombre d'entrées, lors de l'appel
// explication de la notation lhs,rhs
// lhs signifie, left hand side , ou encore côté à gauche d'une expression
// rhs signifie, right hand side , ou encore côté à droite d'une expression
// cette notation vient des informaticiens, qui code les affectations sous la forme
// [lhs,cote gauche,variable que l'on modifie]<-[rhs,coté droit,expression à évaluer]
// lhs représente le côté 'sortie' d'une expression
// rhs représente le côté 'entrée' d'une expression
  disp("vous avez appele bidon avec "+string(lhs)+ " sorties , et "+string(rhs)+" entrées");
  lhs_rhs=10*lhs+rhs;  // juste pour clarté du programme
  if (lhs_rhs==00) then
  //  0 sorties, 0 entrees, on sort sans rien faire
    return
  end
  if (lhs_rhs==11) then
  //  1 sortie, 1 entree , sortie=2*entree
    s1=2*e1
    return
  end
  if (lhs_rhs==22) then
  // 2 sorties, 2 entrees,
   s1=2*e1
    s2=2*e2
    return
  end
// nombres d'entrées sorties non prévu
// => on arrete scilab, en generant un message d'erreur
  error("ca va pas mon gars, faut lire la doc!...")
endfunction
```

```
//-----------------------------------------------------------
// exemples d'appel de la fonction bidon dans un programme quelconque
//-----------------------------------------------------------
bidon(); // affichage du message 0 entrees, 0 sorties
y=bidon(3); // => y<- 2*3=6
[r,s]=bidon(y,4*y^2) //=> r=2*y ,s=8*y^2
[r,s,t]=bidon(y) // affichage d'erreur
```

5 les instructions de debuggage: *pause*, *abort*, *resume*, *whereami(),whos()*

scilab dispose d'instructions permettant de trouver les erreurs dans le programmes...
1-si, lors de l'exécution d'un programme, vous appuyez sur CTRL-C, scilab passe en mode pause. (normalement le prompt **>>** se modifie en **1-**, ce qui indique le mode debuggage)
dans ce mode vous pouvez
1.1- demander à quel endroit le programme s'est arrêté, en écrivant :
whereami() // mais ou suis je ?...
1.2- visualiser le nom des variables définies, en écrivant :
whos() // liste toutes les variables définies
whos -name *toto* ***// liste toutes les variables dont le nom commence par toto***
1.3- reprendre l'exécution normale, en écrivant :
resume // recommence l'execution normale
1.4- arrêter l'exécution, en écrivant :
abort // arrete tout
1.5 visualiser les valeurs des variables, modifier leur valeur, pour débugger
2- on peut également forcer le passage en mode **debug**, à un endroit précis, en écrivant :
pause // force le passage en mode debug a cette ligne de programme