Réaliser des graphiques avec Scilab

Philippe Roux

18 mai 2009
Version PDF : fiche_graphiques.pdf

Table des matières

Ce document est conçu comme un recueil commenté d’exemples “simples” de sorties graphiques obtenues avec le nouveau mode graphique de *Scilab*. Il permet de découvrir l’existence de certaines commandes *Scilab* et leur domaine d’utilisation, mais la lecture de ce document ne peut remplacer la lecture de l’aide en ligne pour une utilisation optimale des commandes décrites. En particulier, ce document décrit la plupart des entités graphiques (handle) mais seulement un petit nombre des propriétés graphiques associées à chacune de ces entités. Ces propriétés sont décrites de manières exhaustive dans l’aide en ligne de *Scilab*.

1 Le “nouveau” mode graphique

Depuis sa création *Scilab* possède un certain nombre de possibilité d’affichages graphiques dont la gestion a considérablement évoluée avec l’apparition d’un nouveau mode graphique (voir [3] et [1]). D’un point de vue chronologique, les graphiques de *Scilab* reposaient encore sur l’ancien mode graphique jusqu’aux versions 3.x, le nouveau mode graphique n’étant encore qu’optionnel. Ensuite dans les versions 4.x de *Scilab* le nouveau mode graphique est devenu le mode par défaut, mais une certaine compatibilité avec les fonctionnalités de l’ancien mode graphique avait été

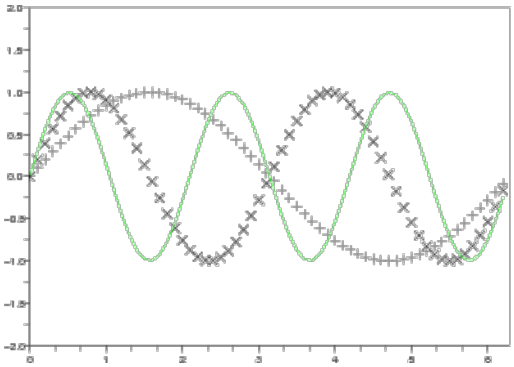
conservée. À partir de la version 5.0 l'ancien mode graphique est devenu totalement obsolète entraînant la disparition de certaines fonctions très utilisées comme `xset`, `xget`, `xdel` ou `xbasc()`.

1.1 Les variables de type handle

Le nouveau mode graphique gère maintenant la mémorisation des entités graphiques selon un modèle hiérarchique : chaque entité va posséder des propriétés ainsi que des parents ou des descendants, certaines propriétés des descendants étant directement hérités des propriétés du parent. Cette description est étroitement associée à un nouveau type de variable *Scilab* : le type `handle`. Ce type de variable permet de stocker les propriétés d'un objets graphique et de pointer vers ses descendants (littéralement c'est un "pointeur"). Il y a plusieurs types à considérer dans l'ordre hiérarchique :

- il y a d'abord le handle `Figure`, qui contient des informations sur la fenêtre graphique (taille positionnement, mode d'affichage . . .)
- il y a ensuite le handle `Axes`, qui contient des informations sur les axes (échelles et graduations qui apparaissent dans la fenêtre)
- viennent ensuite les objets graphiques de base qui composent la figure : `Polyline`, `Rectangle`, `Arc`, `Segs` . . .qui peuvent aussi être agrégés grâce au handle `Compound` pour former des objets plus complexes.

L'intérêt du nouveau mode graphique est de pouvoir modifier les propriétés de ces différents objets après leur création (et tant que la fenêtre graphique n'a pas été détruite ou effacée). Prenons l'exemple du graphique de démonstration de `plot2d` :



```
-->plot2d()

Demo of plot2d()
=========================================
x=(0:0.1:2*%pi)';
plot2d(x,[sin(x),sin(2*x),sin(3*x)],style=[-1,-2,3]);

-->get("current_figure")// handle figure courante (pas stocké)
 ans  =

Handle of type "Figure" with properties:
=========================================
children: "Axes"
figure_style = "new"
figure_position = [27,35]
figure_size = [610,461]
axes_size = [610,461]
```

```
auto_resize = "on"
figure_name = "Scilab Graphic (%d)"
figure_id = 0
color_map= matrix 32x3
pixmap = "off"
pixel_drawing_mode = "copy"
immediate_drawing = "on"
background =  -2
visible = "on"
rotation_style = "unary"
user_data = []
-->f=gcf();//figure courante (stocké dans f)
-->typeof(f)//type handle
 ans  =

 handle
```

Pour agir sur la figure il faut d'abord récupérer son handle, c'est ce permet de faire la commande `get("current_axes")`. Le contenu du handle s'affiche, ce qui donne une idée de la quantité d'informations stockés dans une variable de type `handle` ! Pour pouvoir manipuler le handle il est plus pratique de le stocker dans une variable, c'est ce qu'on fait avec l'instruction `f=gcf()`, où `gcf` est un des nombreux raccourcis de la fonction `get` :

- `gcf()` pour `get("current_figure")`
- `gca()` pour `get("current_axes")`
- `gce()` pour `get("current_entities")`
- `gdf()` pour `get("default_figure")`
- `gda()` pour `get("default_axes")`

Comme *Scilab* permet de gérer plusieurs fenêtres graphiques en même temps on peut avoir besoin de sélectionner ou de modifier la fenêtre graphique courante dont on veut récupérer le handle. On a pour cela plusieurs fonctions à notre disposition :

- `winsid` permet de lister les fenêtres graphiques existantes
- `xselect` permet de mettre en avant la fenêtre graphique courante
- `scf` permet de définir une fenêtre comme fenêtre graphique courante via, par exemple, son handle ou son numéro (récupéré par `winsid`) et, dans ce cas, on peut ensuite récupérer son handle.

on pourra étudier les exemples suivants pour comprendre la manipulation de ces fonctions :

```
-->clf()
-->plot2d();//fenêtre 0

Demo of plot2d()
=======================================
x=(0:0.1:2*%pi)';
plot2d(x,[sin(x),sin(2*x),sin(3*x)],style=[-1,-2,3]);

-->id=winsid()//liste des fenêtres
 id  =

    0.
-->old=id($)//numéro de la dernière fenêtre
```

```
 old  =

    0.
-->new=old+1//numéro de la prochaine fenêtre
 new  =

    1.
-->f=scf(new);//crée une nouvelle fenêtre (n=1)
-->plot3d();//s'affiche dans la nouvelle fenêtre courante (n=1)

Demo of plot3d()
=========================================
t=-%pi:0.3:%pi;plot3d(t,t,sin(t)'*cos(t),35,45,'X@Y@Z',[2,2,4]);

-->winsid()//nouvelle liste des fenêtres
 ans  =

    0.    1.
-->scf(old);//on change la fenêtre courante (n=0)
-->xselect()//mise en avant de la fenêtre courante (n=0)
-->delete(f)//destruction de la fenêtres de f (n=1)
```

1.2 Manipulations des propriétés d'un handle

Une variable de type handle est donc constituée de nombreuses propriétés dont il va falloir récupérer ou modifier les valeurs pour modifier l'aspect d'un graphique. Pour s'y retrouver *Scilab* possède une interface graphique pour naviguer dans l'arborescence des pointeurs graphiques. On y accède depuis le menu « `Editer`»de la fenêtre graphique, via l'onglet « `propriété de la figure` ». Reprenons l'exemple de la figure de démonstration de `plot2d`. En naviguant dans l'arborescence visible dans la partie gauche de l'éditeur (*cf*. FIG.1) on constate que la `Figure` courante possède pour descendant un objet `Axes` qui possède pour descendant une entité `compound` qui possède pour descendants trois objets `Polyline` qui sont en fait les trois courbes sin$x$, sin2x et sin3x qui constitue la figure.

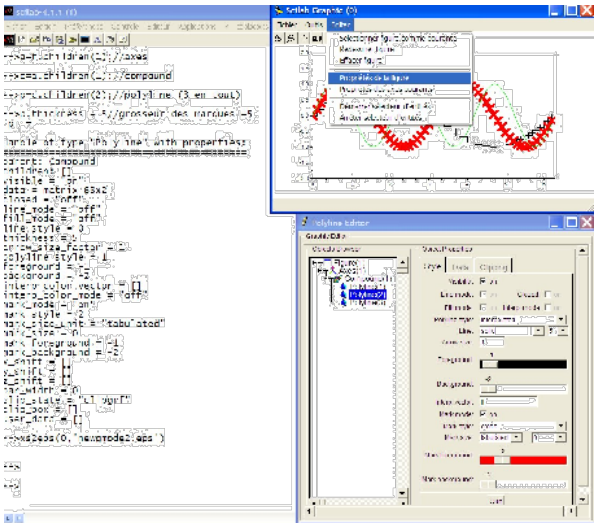


FIG. 1: l'éditeur d'entités graphiques

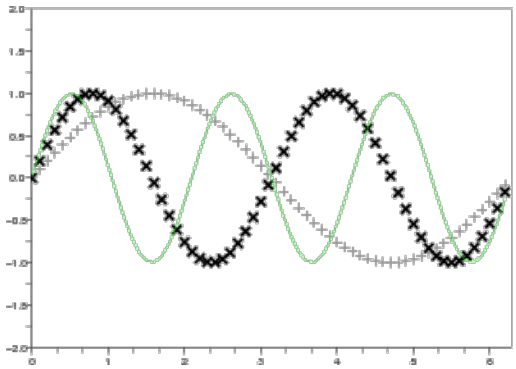
On peut retrouver cette liste de descendants d'une figure en utilisant la partie des handles qui permet de pointer vers les descendant d'une entité : la propriété `children` (qui est aussi de type `handle`) et ceci pour arriver jusqu'aux objets graphiques de base et en modifier l'apparence :

```
-->f=gcf();//handle de la figure
-->a=f.children(1);//axes idem a=gca()
-->e=a.children(1);//entité compound
-->p=e.children(2);//polyline (3 en tout)
```

L'accès aux autres propriétés graphiques de chaque handle se fait avec la même syntaxe `handle.propriété`. Pour modifier la valeur d'une propriété il suffit donc d'y affecter une valeur avec l'opérateur d'affectation ou la fonction `set` (qui remplace la fonction `xset`) :

```
-->p.thickness = 5;//on affecte 5 à la propriété tickness de p
-->set(p,"thickness",5);//affectation équivalente
-->f.children(1).children(1).children(2).thickness = 5;//de même
```

dans chacun des 3 cas l'épaisseur (propriété `thickness`) de la deuxième courbe (deuxième descendant `p=e.children(2)`) a été modifiée et prend maintenant la valeur 5. Il semble que la syntaxe alliant la plus grande simplicité et la plus grande lisibilité soit la première (avis personnel certainement discutable . . .). Dans tous les cas on obtient la figure :



La fonction `set` possède plusieurs raccourcis pour créer, initialiser ou remettre à zéro les propriétés d'une figure :

- `clf()` efface une figure
- `scf()` et `sdf()` créent ou réinitialisent une nouvelle figure
- `sda()` et `sca()` créent ou réinitialisent de nouveaux axes

Dans la suite on verra que les différentes fonctions graphiques de *Scilab* créent des entités graphiques de type handle et les assemblent pour former une figure. Ces fonctions reposent sur quelques fonctions de base capable de manipuler globalement les handle :

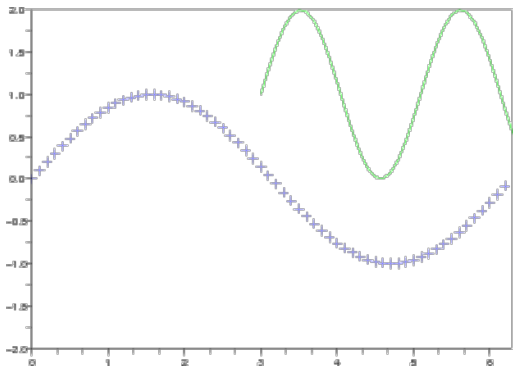
- `glue` permet d'agréger différentes entités pour en faire une nouvelle de type `Compound`
- `unglue` permet au contraire de décomposer une entité graphique de type `Compound` en ses descendants
- `copy` permet de copier une entité graphique (pour l'insérer dans une autre figure par exemple)
- `delete` permet de détruire une ou plusieurs entités graphiques (une fenêtre si l'entité est de type `Figure`).
- `move` permet de faire bouger un objet graphique, par rapport à son ascendant (ses descendants bougerons avec lui !)

Par exemple :

```
-->clf()//effacement fenêtre courante
```

```
-->plot2d();//graphique de départ

Demo of plot2d()
=========================================
x=(0:0.1:2*%pi)';
plot2d(x,[sin(x),sin(2*x),sin(3*x)],style=[-1,-2,3]);

-->e=gce()// dernière entité crée (compound)
 e  =

Handle of type "Compound" with properties:
==========================================
parent: Axes
children: ["Polyline";"Polyline";"Polyline"]
visible = "on"
user_data = []
-->p=e.children(3);//troisième graphe
-->move(e.children(1),[3,1]);//on bouge la première courbe
-->delete(e.children(2));//on détruit la deuxième
-->p.mark_foreground=2;//la troisième en bleu
-->e.children(3)//n'existe plus car on a détruit une des courbes!
 !--error 21
invalid index
at line      17 of function %h_e called by :
e.children(3)//n'existe plus car on a détruit une des courbes!
```

donnera :



1.3 Listes des entités graphiques

Pour finir on peut donner la liste des entités graphiques existantes et leur liens avec les primitives graphiques de *Scilab* qu'on étudiera plus loin. La liste des propriétés associées à chacune de ces entités est extrêmement longue, mais bien détaillée dans l'aide en ligne de *Scilab* ou dans le livre [2].

Handle	description
Figure	c'est le premier niveau graphique, un objet de type Figure contient les informations sur la fenêtre graphique. Elle a pour descendant une entité de type Axes. On récupère son pointeur par `f=get("current_figure")` ou `gcf()`. Voir `figure_properties` pour plus de détails.

Axes	c'est le deuxième niveau graphique, un objet de type Axes contient les informations sur les échelles les axes et les graduations de la figure. Elle a pour descendant des entités de type Compound. On récupère son pointeur par `a=get("current_axes")` ou `gca()`. Voir `axes_properties` pour plus de détails.
Compound	c'est le troisième niveau graphique. Cette entité sert à agréger les différents objets graphiques à l'aide des fonctions `glue` et `unglue`. Voir `Compound_properties` pour plus de détails.
Axis	permet de paramétrer les axes dessinés avec `drawaxis`. Voir `axis_properties` pour plus de détails.
Polyline	contient les propriétés des lignes tracées avec `plot2d` par exemple. Voir `polyline_properties` pour plus de détails.
Arc	propriété des (arcs d')ellipses obtenues avec les fonctions `xarc`, `xfarc`, `xarcs` et `xfarcs`. Voir `arc_properties` pour plus de détails.
Rectangle	propriété des rectangles obtenus avec les fonctions `xrect`, `xfrect` et `xrects`. Voir `rectangle_properties` pour plus de détails.
Surface	propriétés des surfaces définies par des facettes et dessinées en trois dimensions par `plot3d2`. Voir `surface_properties` pour plus de détails.
Fec	propriétés des graphes affichés sous forme de niveau de gris (ou de couleurs) avec `Sgrayplot`, et `Sfgrayplot`. Voir `fec_properties` pour plus de détails.
Grayplot	propriétés des graphes affichés sous forme de niveau de gris (ou de couleurs) avec `grayplot`, et `fgrayplot`. Voir `grayplot_properties` pour plus de détails.
Plot3d	graphiques réalisés avec `plot3d`, ils ne sont pas compatibles avec les "colormap" et ne sont pas documenté dans l'aide en ligne !
Matplot	matrices d'entiers affichées par `Matplot` ou `Matplot1`. Voir `Matplot_properties` pour plus de détails.
Segs	propriétés des segments dessinés par `xsegs` et des flèches dessinées avec `xarrows`. Voir `segs_properties` pour plus de détails.
Champ	affichage des champs de vecteurs avec les fonctions `champ`, `champ1` et `fchamp`. Voir `champ_properties` pour plus de détails.
Text	propriété du texte affiché avec `xstring` ou `xstringb`. Voir `text_properties` pour plus de détails.
Label	contient les propriétés des labels des 3 axes (x,y,z) et du titre de la figure. Voir `label_properties` pour plus de détails.
Legend	propriétés des légendes des courbes en 2 dimensions (obtenues avec `legends` ou `legend`). Voir `legend_properties` pour plus de détails.

Dans la suite on va voir un certain nombre de fonctions qui construisent directement certaines des entités ci-dessus (voir aussi [4]).

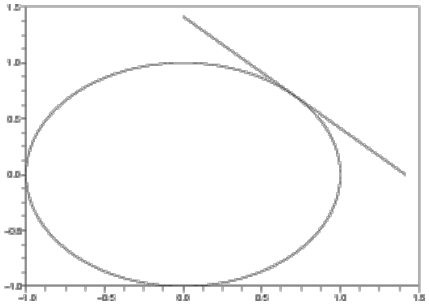
2 Courbes et surfaces

Le dessin avec *Scilab* repose sur le tracé de lignes brisées définies par une suite de points. La manière la plus simple de décrire une liste de points est de se donner la liste de leurs coordonnées x et y (et z en dimension 3) dans l'ordre de parcours de la ligne brisée. Ces listes de coordonnées peuvent être facilement stockées dans des matrices puis utilisées par les différentes fonctions graphiques de *Scilab*.

2.1 En deux dimensions

La commande de principale de *Scilab* pour dessiner des figures planes est l'instruction `plot2d(x,y)`. Cette commande permet de tracer une ligne brisée en reliant par des segments de droites les points de coordonnées `(x(i),y(i))`, qui sont des entités de type `Polyline`. Plusieurs tracés successifs sont affichés sur la même fenêtre graphique jusqu'à son effacement via `clf()` ou la destruction de la fenêtre :

```
//cercle
t=[0:0.02:2*%pi]';
x=cos(t);y=sin(t);
plot2d(x,y)
//tangente y=sqrt(2)-x
x=[0;sqrt(2)];y=[sqrt(2);0];
plot2d(x,y)
```



Sur cette figure le cercle apparaît ovale ! C'est normal puisqu'on a laissé *Scilab* décider de l'échelle en fonctions des données envoyées à `plot2d`. L'échelle choisie n'est donc pas forcément isométrique. On peut spécifier à *Scilab* le comportement à adopter en utilisant des paramètres optionnels de `plot2d`. Ces paramètres sont tous de la forme `clé=valeur`, les plus utiles sont :

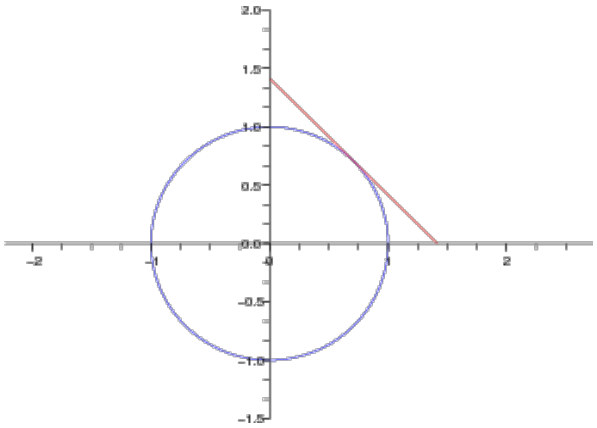
- `axesflag` pour gérer le positionnement des axes, par exemple `axesflag=5` pour avoir des axes qui se croisent en 0,0,
- `frameflag` pour gérer la taille de la fenêtre et les échelle, `frameflag=3 ou 4` pour une échelle isométrique, `frameflag= 5 ou 6` pour avoir une échelle avec des graduations « simples », `frameflag= 0` pour superposer un graphe à la figure précédente en conservant l'échelle
- `rect=[xmin,ymin,xmax,ymax]` permet de délimiter la fenêtre graphique par une boite,
- `style` permet de définir le style de chaque courbe, une couleur pour les valeurs positives ou une marque pour les valeurs négatives

Pour les couleurs on peut utiliser la table des couleurs par défaut de *Scilab*, que l'on peut visualiser avec `getcolor()` :

numéro	1	2	3	4	5	6	7	8	. . .
couleur	noir	bleu	vert	cyan	rouge	magenta	rouge	blanc	. . .

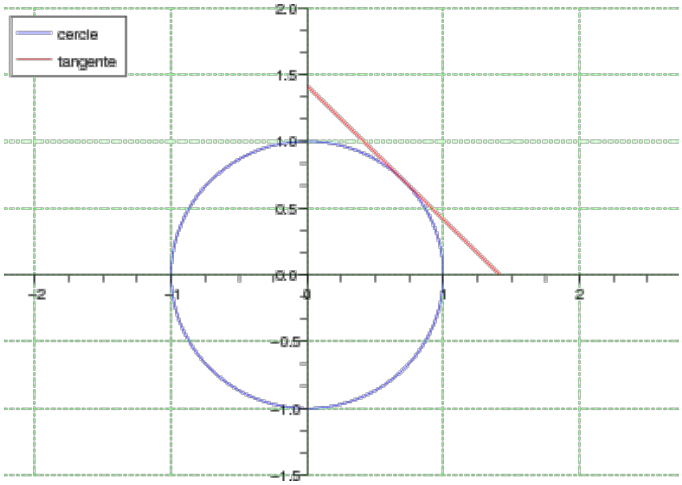
Si on reprend notre exemple on a :

```
clf()
t=[0:0.02:2*%pi]';
x=cos(t);y=sin(t);
plot2d(x,y,style=2,rect=[-1.5,-1.5,2,2],frameflag=3,axesflag=5)//bleu
x=[0;sqrt(2)];
plot2d(x,sqrt(2)-x,style=5,frameflag=0)//rouge
```



Quand on a plus besoin du graphe on peut détruire la fenêtre ou utiliser la commande `clf()` pour effacer le contenu de la fenêtre. Pour agrémenter notre figure on peut y ajouter une grille, pour repérer les coordonnées, avec `xgrid()` et une légende avec `legends`

```
xgrid(3)//grille verte
legends(['cercle' 'tangente'],[2,5],2)//2->position en haut à gauche
```

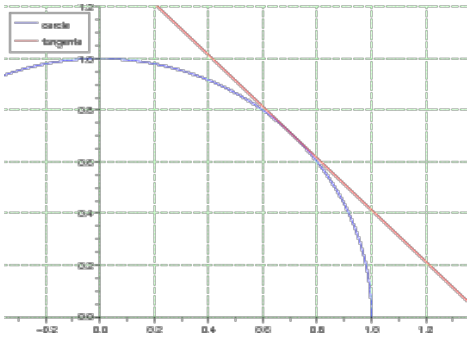


Il est possible de zoomer sur une partie de la fenêtre graphique en utilisant les icônes de la barre d'outils en haut à gauche de la fenêtre :

FIG. 2: barre d'outils de la fenêtre graphique

La première icône de cette barre permet de zoomer sur une partie de la figure, sélectionnée à la souris, la seconde icône permet de "dézoomer" en revenant à la figure de départ. Il est aussi d'effectuer ces opérations en lignes de commandes avec zoom_rect et unzoom :

```
zoom_rect([0 0 1 1])
unzoom()
```

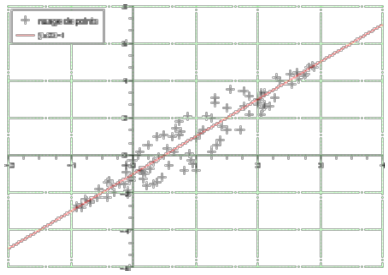


Il est aussi possible d'afficher la position de points avec une marque. Une marque s'obtient en donnant un numéro de couleur négatif. Il y a 15 marques disponibles qui sont définies par un numéro *k* :

k=	0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10	-11	-12	-13	-14
marque	•	+	×	⊕	♦	◇	△	▽	⊕	○	*	□	▷	◁	☆

La liste des marques est aussi accessible via la commande `getmark()` . Un exemple pour afficher un nuage de points et la droite de régression correspondante :

```
clf()
x=4*rand(100,1)-1;//abscisses
bruit=(2-abs(x-1)).*(2*rand(x)-1);
y=2*x-1+bruit//ordonnées
plot2d(x,y,style=-1,..
frameflag=6,axesflag=5)
X=[-2;4];Y=2*X-1;
plot2d(X,Y,style=5)
xgrid(3)
legends(['nuage de points'..
'y=2x-1'],[-1,5],2)
```



Pour dessiner le graphe d'une fonction sur un intervalle défini par une matrice `x`, il faudra faire attention au calcul des ordonnées *y fx*. En effet la commande `y=f(x)` appliqué à

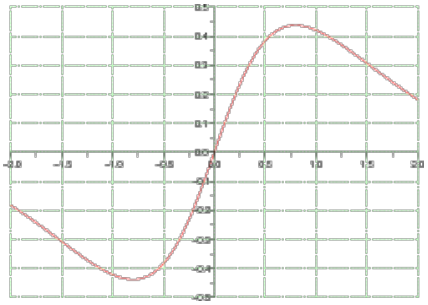
des matrices peut conduire à certaines confusions entres opérations terme à terme et opérations matricielles.

Il y a deux stratégies pour éviter ce genre de problèmes :

- coder la fonction `f` en remplaçant systématiquement les opérations matricielles (`* / ^`) par des opérations terme à terme (`.* ./ .^`)
- faire appel à la fonction `feval` pour l'évaluation au lieu de faire `y=f(x)`
- ne pas évaluer `y=f(x)` mais faire appel à `fplot2d`.

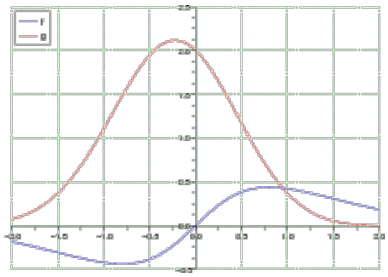
Cette dernière fonction accepte les même options que `plot2d` :

```
clf()
deff('y=f(x)','y=sin(x)/(1+x^2)')
x=[-2:0.02:2]';
fplot2d(x,f,5,..
frameflag=6,axesflag=5)
xgrid(3)
```



Par contre on peut afficher plusieurs courbes en un unique appel à `plot2d` comme ci-dessous :

```
clf()
deff('y=f(x)','y=sin(x)/(1+x^2)')
deff('y=g(x)','y=(2-x)*exp(-x^2)')
x=[-2:0.02:2]';
y1=feval(x,f);y2=feval(x,g);
plot2d([x x],[y1 y2],[2 5],..
frameflag=6,axesflag=5)
xgrid(3)
legends(['f' 'g'],[2,5],2)
```

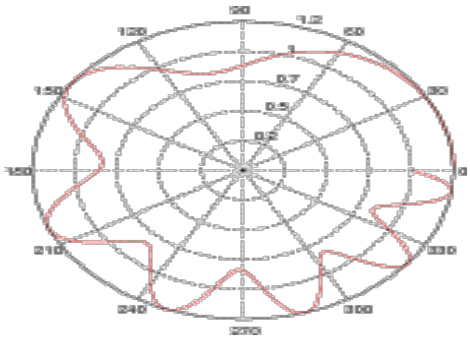


Pour pouvoir tracer plusieurs courbes avec un seul appel à `plot2d` il faut que les différentes matrices contenant les coordonnées (`x`, `y1`,`y2`) soient des matrices **colonnes** et de même taille. C'est nécessaire pour que les arguments de `plot2d` (`[x x]` et `[y1 y2]`) soient

des matrices correctement définies et avec autant de colonnes que de courbes.

si besoin on peut aussi tracer des courbes définies en coordonnées polaires avec `polarplot`. Par exemple pour la courbe définie par $\varrho\theta$ 1 0.2cosθ^2 :

```
clf()
theta=[0:0.02:2*%pi]';
rho=1+0.2*cos(theta.^2)
polarplot(theta,rho,style=5)
a=gca()//axes pour graduations
a.isoview='on'//isométrique
a.data_bounds=[-1.2,-1.2;1.2,01.2]
```



une échelle (graduée en degrés) permet alors de repérer les coordonnées polaires des points.

2.2 En trois dimensions

Scilab permet aussi d'afficher des figures en 3 dimensions. L'affichage d'une telle figure dépend du point de vue d'où on l'observe. Ce point peut être repéré par deux angles α,θ>0,1800,360 comme sur la figure FIG.3. C'est pour cette raison qu'on retrouve dans toutes les fonctions graphiques d'affichage en trois dimension deux paramètres optionnels `alpha` et `theta` qui permettent de fixer le point de vue. Par défaut ces deux valeurs sont fixées par *Scilab* à `alpha=45` et `theta=35`. On peut aussi modifier le point de vue depuis la fenêtre graphique en utilisant le bouton et la souris.

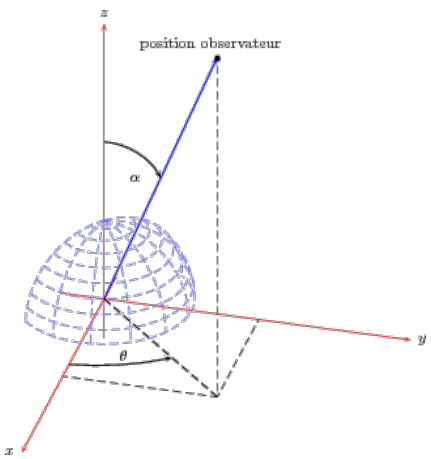


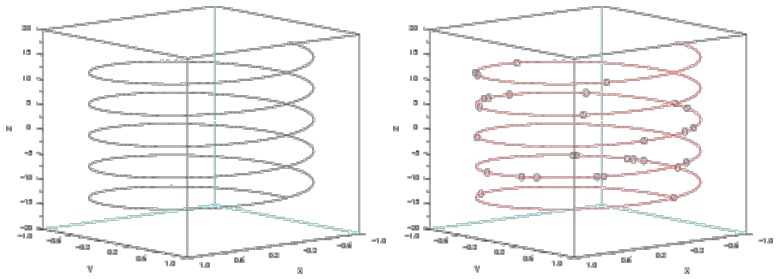
FIG. 3: angles de vue α,θ

Passons maintenant aux primitives *Scilab* pour l'affichage en 3 dimensions. Prenons le cas d'une courbe dans l'espace définie de manière paramétrique, c'est à dire décrite par trois fonctions *xt,yt,zt*. Il faut d'abord coder cette description en une fonction *Scilab* :

```
function [x,y,z]=helice(t)
x=cos(t)
y=sin(t)
z=t
endfunction
```

Ensuite on peut utiliser `param3d` ou `param3d1` pour afficher la courbe. Dans les deux cas on pourra afficher la courbe en couleur ou avec des marques soit directement avec `param3d1` soit en utilisant en accédant aux propriétés de la figure via les pointeurs graphiques (les handle décrits précédemment) :

```
t=[-5*%pi:0.02:5*%pi]';
[x,y,z]=helice(t);
param3d(x,y,z,alpha=15,theta=50)
//placer des points marqué par un "O"
t=10*%pi*rand(30,1)-5*%pi;
[x,y,z]=helice(t);
param3d1(x,y,list(z,-9),alpha=15,theta=50)
a=gca();//axes courants
p=a.children(2);//première courbe (celle en noir)
p.foreground=5;//maintenant en rouge
```



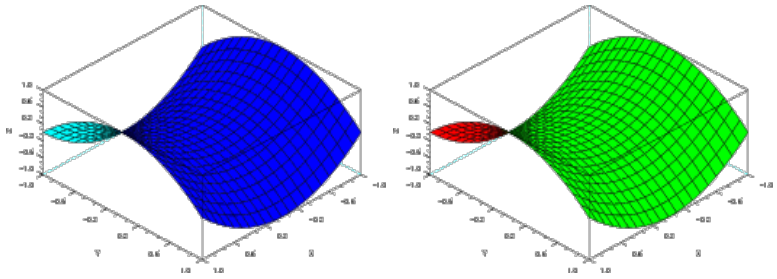
L'affichage des surfaces dépend de leur description mathématique. Les surfaces les plus simples à tracer sont celles qui sont le graphe d'une fonction à deux variables. Le principe est le même que pour `plot2d` :

- découper le domaine des *x,y* sur le quel on veut tracer la fonction
- évaluer la fonction sur ce domaine *z fx,y*
- lancer `plot3d` pour l'affichage avec les paramètres *x,y* et *z*

la seule différence ici est que le domaine est défini par deux matrice *x* et *y*, donc par des carrés dans le plan *xOy*, et l'évaluation de *f* sur ce découpage nécessite impérativement l'utilisation de `feval`.

```
function [z]=selle(x,y)
z=x^2-y^2
endfunction
x=[-1:0.1:1]';y=x;
z=feval(x,y,selle);
plot3d(x,y,z)

a=gca();//"axes" courant
p=a.children;//Plot3d
p.foreground=1;//grille noir
p.color_mode=3;//dessus vert
```

```
p.hiddencolor=5;//dessous rouge
```

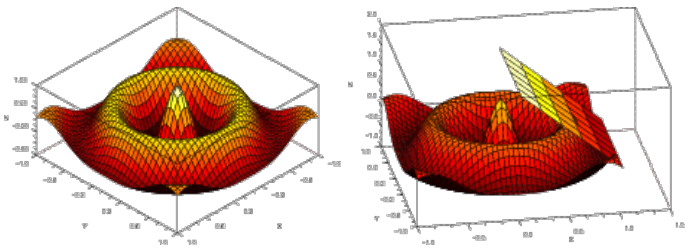


Les surfaces sont donc affichées avec 3 couleurs :

- `color_mode` pour le dessus de la courbe (bleu par défaut),
- `hiddencolor` pour le dessous de la courbe (cyan par défaut),
- `foreground` pour la grille délimitant les facettes (noir par défaut).

Si l'on veut plutôt utiliser un dégradé de couleurs fonction du niveau d'élévation de la surface, il faut alors utiliser la fonction `plot3d1` et la propriété `color_map` de la figure. La propriété `color_map` désigne la table des couleurs utilisée, c'est une matrice de taille 3 *n* où chaque ligne donne les niveaux RGB des *n* couleurs. Il existe plusieurs fonctions *Scilab* pour créer automatiquement des tables de couleur comme `hotcolormap` ou `jetcolormap`. On peut aussi une table de couleur ne contenant que des niveaux gris `graycolormap`.

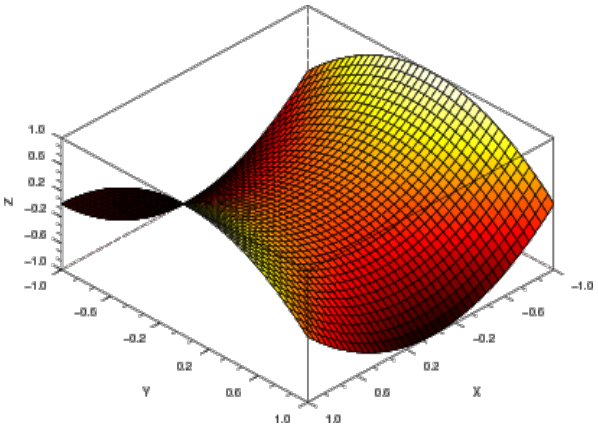
```
function z=f(x,y)
r=sqrt(x^2+y^2)
z=exp(-r)*cos(3*%pi*r)
endfunction

//plan tangent en x=0.7 y=0
function z=planT(x,y)
z=0.47228069-1.91854395*(x-0.7)
endfunction

x=[-1:0.05:1]';y=x;
z=feval(x,y,f);
plot3d1(x,y,z)
cmap=hotcolormap(64);
f=gcf();//figure courante
f.color_map=cmap;
x=[0.1:0.2:1.1]';
y=[-0.6:0.2:0.6]';
z=feval(x,y,planT);
plot3d1(x,y,z,theta=-100,alpha=45)
```



Contrairement aux surface obtenues avec `plot3d1` les entités de type `Plot3d` ne sont pas compatible avec les « colormap ». Si la propriété color_map de la figure à été changé alors les surfaces `Plot3d` s'afficherons en noir.

Comme pour les graphes de fonctions à une variable, il existe des fonctions `fplot3d` et `fplot3d1` qui évaluent directement la fonction sur la grille avant d'afficher la surface correspondante. Elles qui acceptent les mêmes options que `fplot3d` et `fplot3d1`. Par exemple on pourra vérifier l'équivalence entre les instructions `plot3d*` et `fplot3d*` suivantes :

```
function z=f(x,y)
z=x^2-y^2
endfunction
x=[-1:0.05:1]';
y=x;
z=feval(x,y,f);

plot3d(x,y,z)
plot3d1(x,y,z)

fplot3d(x,y,f)
fplot3d1(x,y,f)
```



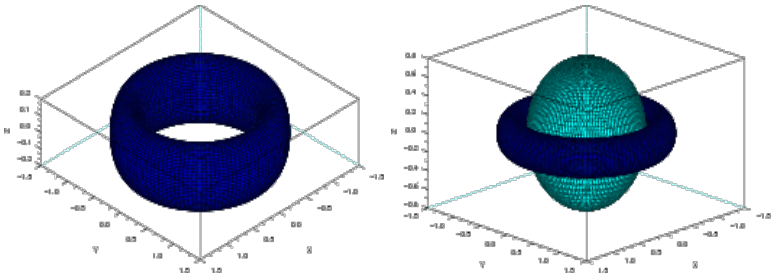
Les surfaces définies de manière paramétrique sont plus difficiles à tracer. Il faut d'abord coder la fonction décrivant les coordonnées *x,y,z* des points en fonction des paramètres :

```
function [x,y,z]=tore(theta,phi)
R=1,r=0.2
x=(R+r*cos(phi)).*cos(theta)
y=(R+r*cos(phi)).*sin(theta)
z=r*sin(phi)
endfunction
function [x,y,z]=sphere(theta,phi)
R=0.8
x=R*cos(phi).*cos(theta)
y=R*cos(phi).*sin(theta)
z=R*sin(phi)
endfunction
```

Ensuite, il faut appeler la fonction `eval3dp` pour calculer la liste des facettes à afficher, et c'est

seulement après qu'on peut appeler `plot3d` ou `plot3d1` :

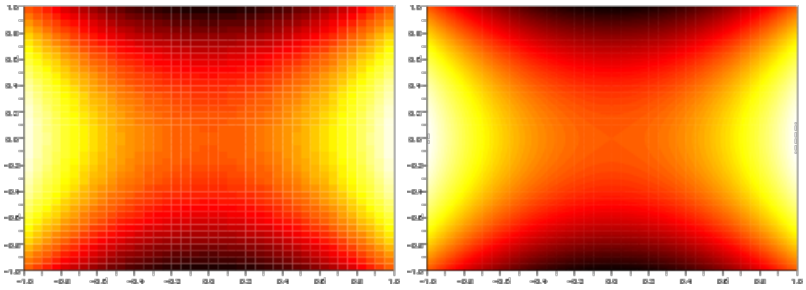
```
phi=[0:0.1:2*3.15];
theta=[2*3.15:-0.05:0];
[x,y,z]=eval3dp(tore,theta,phi);
plot3d(x,y,z,alpha=75,theta=45)
a=gca();//"axes" courant
p=a.children(1);//Plot3d
p.color_mode=10;//dessus bleu

//on ajoute une sphère au tore
[x,y,z]=eval3dp(sphere,theta,phi);
plot3d(x,y,z,alpha=75,theta=45)
p=a.children(1);//Plot3d
p.color_mode=4;//dessus cyan
```



Pour que `eval3dp` puisse évaluer correctement les facettes d'une surface paramétrique il faut absolument que les opérations qui peuvent être des opérations matricielles (`*, /, ^`) soient remplacées par les opérations terme à terme correspondantes (`.*, ./, .^`) dans le code de la fonction.

Pour revenir aux graphes de fonctions, il existe aussi d'autres manières de représenter le graphe d'une fonction à deux variables directement dans le plan. Une première approche consiste à colorier chaque point *x,y* du domaine, sur lequel on veut tracer le graphe, en fonction de l'altitude *fx,y* en ce point (exactement comme pour les colormap avec `plot3d1` mais à plat cette fois), le choix des couleurs étant fixé par la `colormap` de la fenêtre graphique (comme pour `plot3d1`). C'est ce que font les fonctions `grayplot` et `Sgrayplot` qui utilisent la même syntaxe que `plot3d1`. Techniquement ce n'est pas un point qui est colorié mais une cellule de la grille définie par les matrices `x` et `y` :

```
function z=selle(x,y)
z=x^2-y^2
endfunction
x=[-1:0.05:1]';
y=x;

z=feval(x,y,selle);
grayplot(x,y,z)
f=gcf()//handle figure
f.color_map=hotcolormap(64)
Sgrayplot(x,y,z)
```

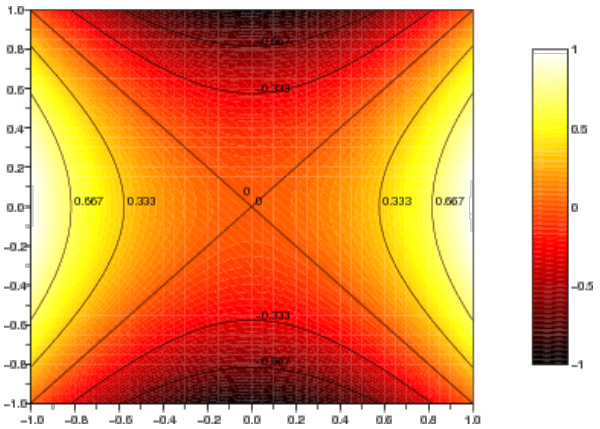


La différence entre les fonctions `grayplot` et `Sgrayplot` tient au mode de calcul de la couleur (valeur) assignée à chaque cellule. Pour `grayplot` la valeur est constante sur toute la cellule (carrée) et déterminée par la moyenne des valeurs de la fonction aux quatre coins de la cellule. Pour `Sgrayplot` la valeur varie "linéairement" sur chaque cellule (triangulaire) ce qui donne un affichage plus "lisse" (littéralement « *Smooth* grayplot ») c'est à dire avec une moindre pixelisation même avec une grille manquant de finesse. Il existe aussi des fonctions `fgrayplot` et `Sfgrayplot` pour effectuer directement l'évaluation de la fonction sur la grille avant l'affichage. Pour rendre plus lisible cet affichage on peut y ajouter :

- une échelle de couleurs, avec la fonction `colobar(amin,amax)`, indiquant le lien entre couleur et altitude pour l'intervalle *amin,amax*
- les lignes de niveaux de la surface (comme sur les cartes géographiques) avec les fonctions `contour`, `contour2d`, `fcontour` ou `fcontour2d`

les fonctions `contour` prennent en paramètre les données de la surface (*x,y,z* ou *x,y,f*) et le nombre de courbes de niveaux à tracer.

```
clf()
f=gcf()
f.color_map=hotcolormap(64)
colorbar(-1,1)
Sgrayplot(x,y,z)
contour(x,y,z,5)
```

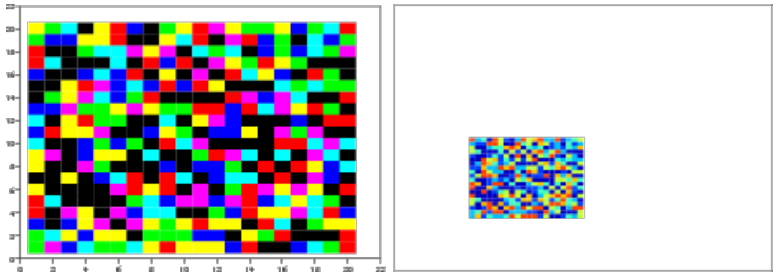


la fonction `colorbar` doit être appelée **avant** les instructions graphiques `plot3d1`, `grayplot3d ...`

Si on désire afficher sous forme d'image une matrice à coefficients entier on utilisera plutôt

`Matplot` et `Matplot1`. La première fonction prend en entrée une matrice d'entier `z` (ou `int(z)` si les valeurs de `z` ne sont pas entières) dans une fenêtres dont les axes sont gradués en pixels. La seconde permet en plus de spécifier le pacement dans la fenêtre graphique, dans l'échelle courante, préalablement spécifiée par un appel à `plot2d` ou directement en modifiant la propriété `data_bounds` des `Axes`.

```
//premier exemple avec
//colormap par défaut
f=gcf()
delete(f)
z=int(8*rand(20,20))
Matplot(z)

clf()
f=gcf()
f.color_map=jetcolormap(8)
ax=gca();//axes courants
ax.data_bounds=[0,0;10,10];
ax.box='on'; //boite
Matplot1(z,[2,2,5,5])
```



3 Dessiner des formes géométriques simples

Scilab possède quelques macros spécifiques pour dessiner directement des formes géométriques courantes dans le plan. Pour tester ces différentes fonctions il faut commencer par ouvrir une fenêtre graphique vide de dimensions fixées. Pour cela on peut utiliser `plot2d` avec l'option `rect` pour délimiter les dimensions et `frameflag=3` pour avoir une échelle isométrique. C'est ce qui est fait dans les exemples suivants avec en plus l'ajout d'une grille pour mieux visualiser les coordonnées :

```
plot2d(0,0,0,rect=[0,0,10,10],frameflag=3)
xgrid(1)//grille pour les coordonnées
```

Ces deux commandes sont utilisées avant chacun des exemples de cette partie, après éventuellement avoir effacé la fenêtre graphique avec `clf()`.

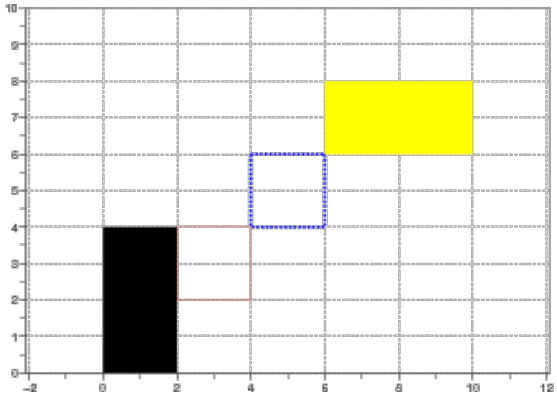
3.1 Les rectangles

Un rectangle est défini par quatre nombres réels *x,y,w,h* :

- *x,y* donnent les coordonnées du point en haut à gauche du rectangle,
- *w,h* donnent les dimensions (largeur et hauteur) du rectangle.

Les fonctions `xrect` et `xfrect` prennent en entré quatre paramètres *x,y,w,h* pour dessiner un rectangle. La fontion `xrect` dessine seulement le contour du rectangle alors que `xfrect` remplit le rectangle avec une couleur :

```
xfrect(0,4,2,4)//rectangle
xrect(2,4,2,2) //carré
e=gce()//dernière entité
e.foreground=5//rouge
xrect(4,6,2,2) //carré
e=gce()//dernière entité
e.foreground=2//bleu
e.line_style=3//pointillés
e.thickness=5//épaisseur
xfrect(6,8,4,2)
e=gce()//dernière entité
e.background=7//jaune
```

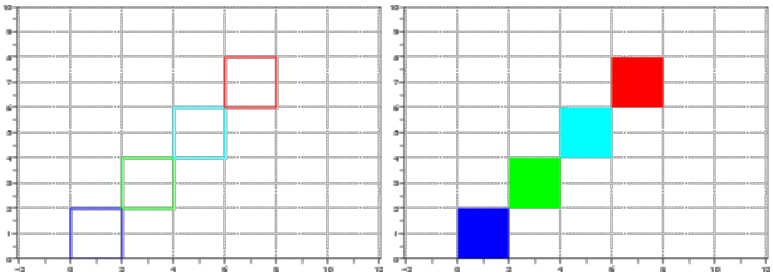


En fait, les deux fonctions `xrect` et `xfrect` créent une même entité `Rectangle` . La seule différence réside dans la propriété `fill_mode` qui est à “off” pour le contour et “on” pour un rectangle plein. Suivant qu’on a faire à un contour ou pas, la couleur utilisée sera dans la propriété `foreground` ou `background`. On a aussi la fonction `xrects` qui permet de tracer plusieurs rectangles en même temps. Cette fonction prend deux arguments en entrée :

- une matrice 4 *n* dont chaque colonne décrit un rectangle
- une liste de *n* éléments donnant la couleur de chaque rectangle

```
rects=[0,2,2,2;
2,4,2,2;
4,6,2,2
6,8,2,2]';


//pour les contours
xrects(rects,[-2:-1:-5])
e=gce()//entité compound
e.children.thickness=4//épaisseur
//pour des rectangles plein
xrects(rects,[2:5])
```

notez bien la transposition `'` pour que chaque colonne de `rects` corresponde à un rectangle !



le deuxième argument de `xrects` est une matrice `fill` à 1 ligne et *n* colonnes qui permet de définir la couleur ainsi que le mode utilisé pour colorier chacune des *n* courbe

- si *fill(i)*<0 alors seuls les contours du $i^{ième}$ rectangle sont dessinés avec la couleur `-fill(i)`
- si *fill(i)*>0 alors le $i^{ième}$ rectangle est rempli avec la couleur `fill(i)`

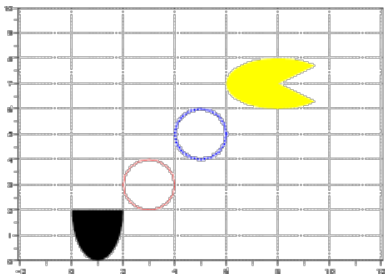
3.2 Les ellipses

De même que pour les rectangles, il faut une liste de six nombres (x,y,w,h,a_1,a_2)pour décrire une portion ou un secteur d'ellipse :

- x,y,w,h définissent les dimensions du rectangle contenant l'ellipse,
- a_1,a_2 délimitent la portion d'arc dessinée, dans un repère orthonormé d'origine le centre de l'ellipse, l'arc démarre à l'angle polaire a_1~64 et se termine à l'angle polaire a_1 a_2~64 en tournant dans le sens direct.

Là encore deux fonctions `xarc` et `xfarc` prennant en entrée six réels x,y,w,h,a_1,a_2 permettent, comme pour les rectangles, de décrire des ellipses. La différence entre les deux fonctions : `xarc` permet de tracer un arc d'ellipse alors que `xfarc` permet de remplir un secteur d'ellipse.

```
//ellipse
xfarc(0,4,2,4,64*180,64*180)
xarc(2,4,2,2,64*0,64*360) //cercle
e=gce()//dernière entité (Arc)
e.foreground=5//rouge



xarc(4,6,2,2,0,64*360) //cercle
e=gce()//dernière entité (Arc)
e.foreground=2//bleu
e.line_style=3//pointillés
e.thickness=5//épaisseur
xfarc(6,8,4,2,64*45,64*270)
e=gce()//dernière entité (Arc)
e.background=7//jaune
```

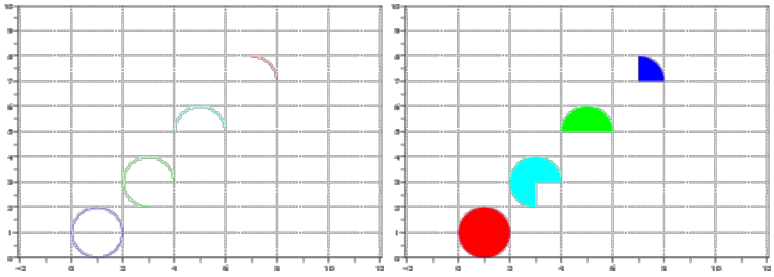


Les deux fonctions `xarc` et `xfarc` créent une même entité `Arc`. La seule différence réside dans la

propriété `fill_mode` qui est à “on” pour un secteur d’ellipse et “off” pour un arc. Suivant qu’on a faire à un arc ou un secteur, la couleur utilisée sera dans la propriété `foreground` ou `background`. On a aussi les deux fonctions `xarcs` et `xfarcs` permettent de tracer plusieurs ellipses en même temps. Ces fonctions prennent deux arguments en entrée :

- une matrice 6 *n* dont chaque colonne décrit une ellipse
- une liste de *n* éléments donnant la couleur de chaque ellipse

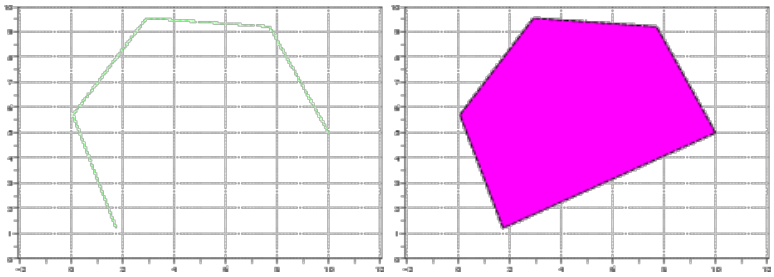
```
arcs=[0,2,2,2,0,64*360;
2,4,2,2,0,64*270;
4,6,2,2,0,64*180
6,8,2,2,0,64*90]';
//cercle
xarcs(arcs,[2:5])
//ellipse
xfarcs(arcs,[5:-1:2])
```



3.3 Les polygones

Plus généralement *Scilab* permet de tracer des lignes brisées de n’importe quelle longueur et des polygones, qui sont des lignes brisées “refermées” sur elle-même. Comme pour `plot2d` une ligne brisée est définie par deux listes de coordonnées définissant une suite de points du plan. C’est ce que fait la commande `xpoly` qui possède en plus un paramètre pour déterminer si la ligne est fermé (polygone) ou pas :

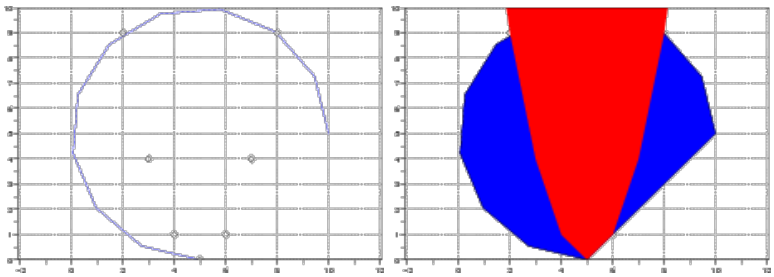
```
t=[0:3*%pi/2],
x=5+5*cos(t),
y=5+5*sin(t)
xpoly(x,y,"lines",0)//0= ouvert
e=gce()//entité courante
e.foreground=3//vert
e.line_style=2//pointillés

xfpoly(x,y,1)//1=fermé
e=gce()//entité courante
e.background=6//violet
e.line_style=2//pointillés
e.thickness=3//épaisseur
```



Encore une fois, les deux fonctions `xpoly` et `xfpoly` créent une même entité `Polyline`, la seule différence réside dans la propriété `fill_mode` qui est à “on” pour le contour et “off” pour remplir la surface. Suivant que la ligne soit fermée ou pas, la couleur utilisée sera dans la propriété `background` ou `foreground`. On a aussi les deux fonctions `xpolys` et `xfpolys` qui permettent de tracer plusieurs polygones en même temps. Ces fonctions prennent deux arguments en entrée :

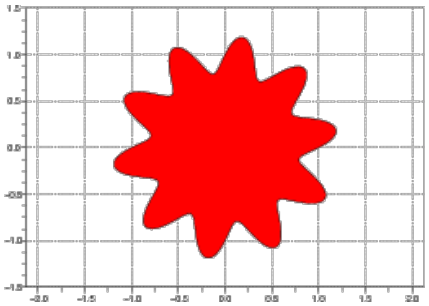
- une matrice *p n* dont chaque colonne décrit un polygone à *p* points
- une liste de *n* éléments donnant la couleur de chaque polygone

```
//premier polygone
t=linspace(0,3*%pi/2,11)'
x1=5+5*cos(t),
y1=5+5*sin(t)
//deuxième polygone
x2=[0:10]'
y2=(x2-5).^2

//assemblage des paramètres
X=[x1,x2];Y=[y1,y2]
//contours avec lignes ou marques
xpolys(X,Y,[2,-5])
//remplissage avec couleur
xfpolys(X,Y,[2,-5])
```



à partir de polygones on peut créer n’importe quelle forme géométrique, à condition de prendre en compte un nombre suffisant de cotés :

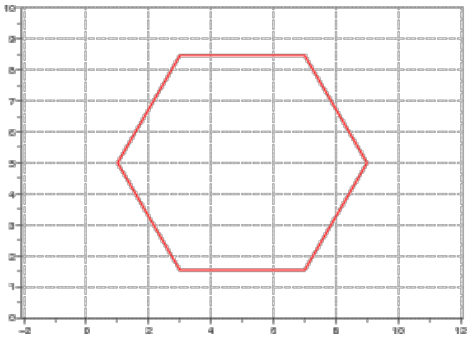
```
t=[0:0.02:2*%pi]';
r=1+0.2*sin(10*t);
x=r.*cos(t);
y=r.*sin(t);
xfpoly(x,y,5)
```



Il existe enfin une commande `xrpoly` qui permet de tracer un polygone régulier, en donnant :

- les coordonnées *x*,*y* de son centre,
- son nombre de cotés,
- son diamètre (celui du cercle dans lequel il est inscrit).

```
xrpoly([5,5],6,4)
e=gce()
e.foreground=5
e.thickness=3
```

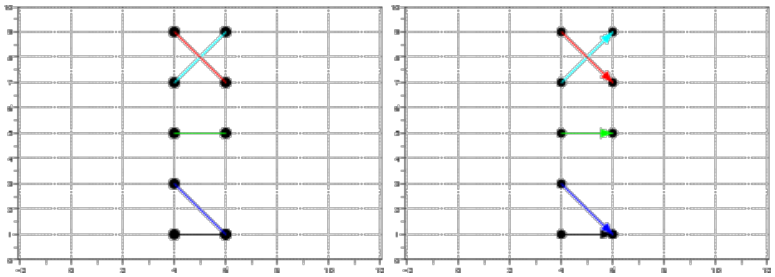


3.4 Segments et flèches

Il est aussi possible de dessiner des segments, déconnectés les uns des autres, avec la commande `xsegs`, chaque segment étant décrit par deux matrices *X* et *Y* de taille 2 *n* donnant les coordonnées *x*,*y* des points de départ (première ligne) et d'arrivée (deuxième ligne) de chacune des *n* flèches. On peut aussi donner une couleur à chaque segment avec le troisième argument de la fonction `xsegs`. Les segments sont des entités de type `Segs` pour lesquels les propriétés de type `mark_*` permettent de gérer les marques aux extrémités des segments. De même la propriété `arrow_size` permet d'ajouter un triangle à la deuxième extrémité du segment pour en faire une flèche. On peut aussi directement faire des flèches avec la fonctions `xarrows` :

```
x1=4*ones(1,5)
y1=[1:2:9];
x2=6*ones(1,5)
y2=[1 1 5 9 7];
X=[x1;x2];Y=[y1;y2]
xsegs(X,Y,[1:5])
e=gce()
e.thickness=3
e.mark_size=2
e.mark_mode="on"
xarrows(X,Y,8,[1:5])
e=gce()
```

```
e.thickness=3
e.mark_size=1
e.arrow_size=10
e.mark_mode="on"
```



3.5 Champs de vecteurs

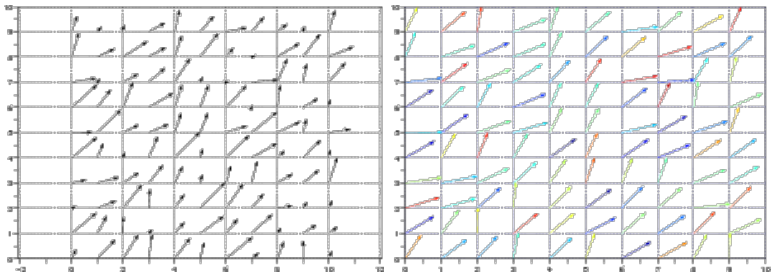
Un cas particulier de flèche est celui des champs de vecteurs. Un champ de vecteur du plan est une fonction qui à chaque point *x,y* du plan associe un vecteur *u,v*. Représenter graphiquement un champ de vecteur revient à dessiner la flèche correspondant au vecteur de coordonnées *u,v* au point *x,y* qui lui est associé. Un champ va être décrit par quatre matrices

- deux matrices colonnes *x* et *y* (tailles *p* et *n*) donnant la discrétisation du domaine sur lequel on va tracer le champ,
- deux matrices f_x et f_y de taille *pn* telles que le vecteur associé au point de coordonnées *xi,yj* ait pour coordonnées *u* f_x*i,j* et *v* f_y*i,j*

Pour *Scilab* les champs de vecteurs sont des entités spécifiques dénommées `Champ`. Les fonctions `champ` et `champ1` vont prendre en argument quatre matrices pour afficher le champ, plus éventuellement des options du même type que celles de `plot2d`. La différence entre ces deux fonctions réside dans le fait que `champ` affiche en chaque point *x,y* des flèches de longueur proportionnelle à celle celle du vecteur *u,v* alors que `champ1` va tracer des flèches de même longueur et indiquer la longueur de *u,v* via la couleur donnée à la flèche (en utilisant la table de couleurs courante) :

```
x=[0:10]';y=x;
u=rand(11,11);v=rand(11,11);
champ(x,y,u,v,rect=[0,0,10,10])
xgrid(1)//grille
a=gca()//affichage isométrique
a.isoview="on"
clf()
champ1(x,y,u,v,rect=[0,0,10,10])
xgrid(1)//grille
cmap=jetcolormap(64)
f=gcf()//choix table couleurs
f.color_map=cmap
```

le passage de l'affichage couleur à un affichage noir et blanc est contrôlé par la propriété `colored` du handle `Champ`. Pour obtenir un affichage isométrique (même échelle en *x* qu'en *y*) on ne peut pas utiliser le paramètre optionnel `frameflag` de `plot2d`c'est pourquoi on a modifié la propriété `isoview` des `Axes`" courant, ce qui donne :

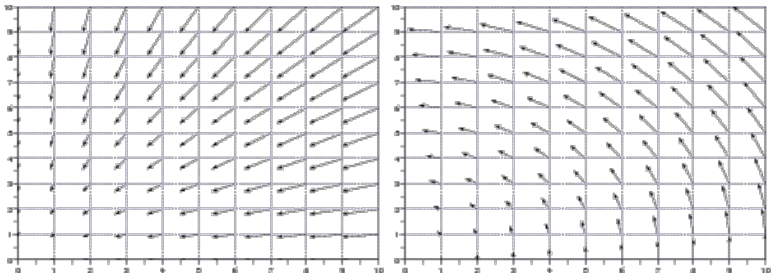


Dans la pratiques les champs de vecteurs sont utilisés pour représenter des systèmes d'équations différentielles de la forme

$$\frac{d}{dt}\begin{pmatrix}x\\y\end{pmatrix} = f(t,x,y) \quad \text{où} \quad \begin{aligned} f : \ \mathbb{R}\times\mathbb{R}^2 &\longrightarrow \mathbb{R}^2 \\ t,(x,y) &\longmapsto (u,v) \end{aligned}$$

La fonction *Scilab* `fchamp` permet de tracer le champ de vecteur associé à la fonction *f* en donnant seulement comme paramètres la valeur de *t* et bien sûr les matrices *x* et *y* (pour le découpage du domaine) :

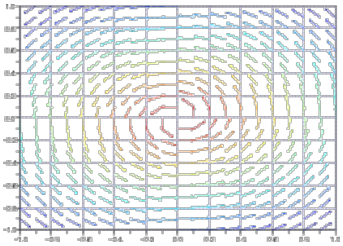
```
function [u]=converge(t,x)
u(1)=-x(1),u(2)=-x(2)
endfunction
function [u]=rotation(t,x)
u(1)=-x(2),u(2)=x(1)
endfunction
x=[0:10]';y=x;
rect=[0,0,10,10]
clf()
fchamp(converge,0,x,y,rect=rect)
xgrid(1)//grille
clf()
fchamp(rotation,0,x,y,rect=rect)
xgrid(1)//grille
```



il n'existe pas de fonction `fchamp1` pour tracer directement en couleurs un champ de vecteurs défini par une fonction *f*, mais il est très facile de définir une telle fonction à partir de la fonction `fchamp` :

```
function fchamp1(f,t,x,y)
rect=[min(x),min(y),max(x),max(y)]
fchamp(f,t,x,y,rect=rect)
xgrid(3)//grille
cmap=jetcolormap(64)
F=gcf()//charger colormap
```

```
F.color_map=cmap
e=gce()//entité champ
e.colored="on"//mode couleur
endfunction
clf()//utilisation
x=[-1:0.1:1]';y=x;
fchamp1(rotation,0,x,y)
```

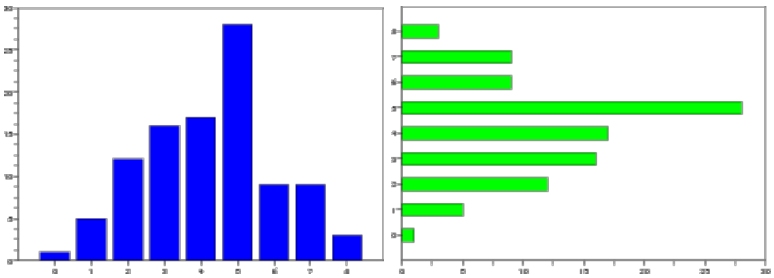


3.6 Statistiques

Les diagrammes utilisés en statistiques nécessitent l'utilisation de formes géométriques pour les diagrammes en bâtons, les « camemberts »*ect. . . Scilab* permet de tracer directement ces diagrammes sans efforts. Les fonction graphiques ci-après génèrent des objets graphiques de type `compound` plus ou moins complexes.

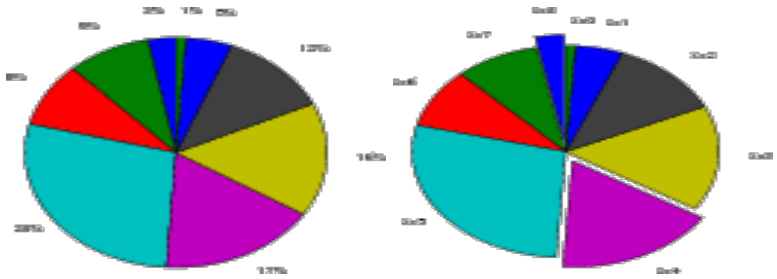
Pour représenter les effectifs d'une série statistique on utilise souvent des diagrammes en bâtons. On peut les tracer avec les commandes `bar` et `barh` suivant que l'on veuille des bâtons verticaux ou horizontaux.

```
X=grand(100,1,'bin',10,0.4);
m=tabul(X)//table des fréquences
x=m(:,1)//valeurs
n=m(:,2)//effectifs
clf()//barres verticales
bar(x,n)
clf()//barres horizontales
barh(x,n,0.5,'green')
```



Au contraire pour représenter les effectifs d'un caractère qualitatif on utilise plutôt des diagrammes appelés "camembert". Pour cela on peut faire appel à la fonction `pie` :

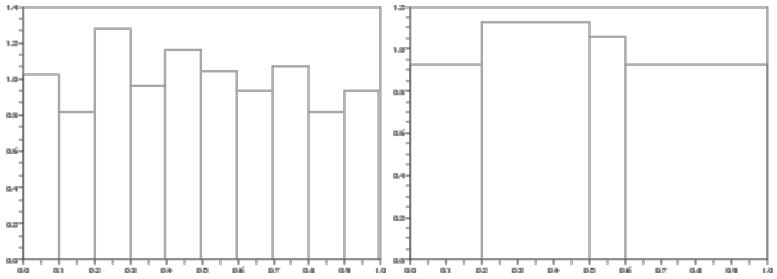
```
clf()
pie(n)
clf()//avec commentaires
pie(n,bool2s((x==4)|(x==8)),'x='+string(x))
```



Pour les séries statistiques regroupées par intervalles on utilisera plutôt la commande `histplot`, qui prend deux paramètres en entrée

- le premier donnant au choix le nombre de modalités ou un vecteur ligne avec les bornes des modalités
- second étant la série statistique elle même

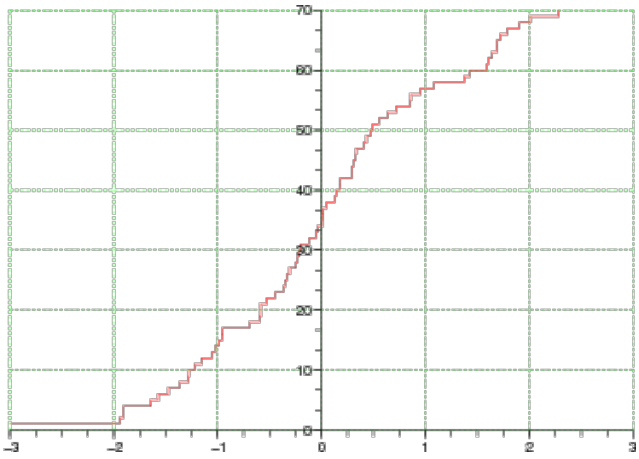
```
Y=rand(1000,1);
clf()
histplot(10,Y)
clf()
mod=[0 0.2 0.5 0.6 1]
histplot(mod,Y,normalization=%t)
```



L'option `normalization` peut prendre les valeurs `%f` ou `%t` selon que l'échelle en ordonnées correspond à l'effectif réel ou à l'effectif corrigé. Dans ce dernier cas la surface de chaque rectangle est proportionnelle à l'effectif de la modalité (et dépend donc de son amplitude) et l'échelle en ordonnées est calculée de telle sorte que la somme des surfaces des rectangles vaille 1. Cette option est très pratique si l'on veut vérifier sur l'histogramme la convergence de variables aléatoire vers une loi donnée.

Pour les fonctions de répartitions on peut utiliser la fonction `plot2d2` qui s'utilise comme `plot2d` mais au lieu de dessiner le graphe comme s'il s'agissait d'une fonction affine par morceau elle la dessinera comme si elle était constante par morceau :

```
//simulation d'une loi normale
clf()
Z=grand(70,1,'nor',0,1);
m=tabul(Z);

//répartition empirique
x=m($:-1:1,1);
y=cumsum(m($:-1:1,2));
plot2d2(x,y,5,axesflag=5)
xgrid(3)
```



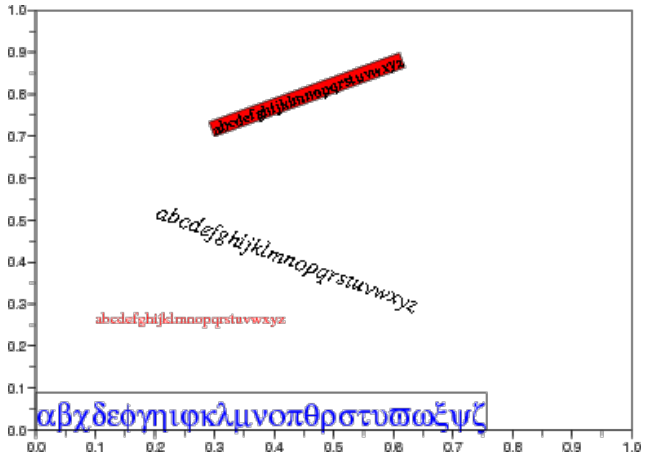
4 Pour aller plus loin

Les possibilités graphiques de *Scilab* ne se limitent pas à la création de figures, il est aussi possible d'interagir avec la fenêtre graphique, d'y faire apparaître des informations sous forme de texte, de sauvegarder les figure vers différents formats ou d'enchaîner les images pour faire des animations.

4.1 Titres et commentaires

Scilab permet d'ajouter du texte, sous différentes formes, dans les figures. La principale commande pour ajouter du texte est la commande `xstring`. Cette commande crée des entités de type `Text` qui contiennent les informations sur la chaîne de caractères à afficher : taille, fonte, couleur . . .

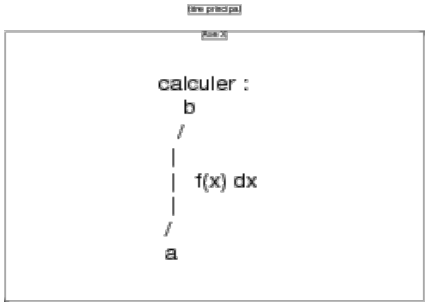
```
plot2d(0,0,0,rect=[0,0,1,1])
alpha="abcdefghijklmnopqrstuvwxyz"
//font courrier, noir
xstring(0.1,0.25,alpha)
e=gce()
e.font_size=1
e.font_style=2
e.font_foreground=5
//font symbol, bleu et encadrée
xstring(0,0,alpha,0,1)
e=gce()
e.font_size=4
e.font_style=1
e.font_foreground=2

//font times, noir, inclinée
xstring(0.2,0.5,alpha,20)
e=gce()
e.font_size=3
e.font_angle=20
e.font_style=3
e.foreground=2
//font helvetica sur fond rouge
xstring(0.3,0.7,alpha,340)
e=gce()
e.font_size=2
e.font_style=7
e.box="on"
e.fill_mode="on"
e.background=5
```



Les chaînes de caractères affichées avec `xstring` sont encapsulées dans une boite dont les dimensions sont calculées par la fonction `xstingl`. Cela peut être utile pour calculer le placement de la boite en fonction de la longueur de la chaîne. On a aussi la fonction `xstingb` pour centrer une chaîne dans une boite donnée.

Il est aussi possible de faire apparaître un titre en haut d'une figure ou de modifier les labels des axes *x* et *y* avec la commande `xtitle`. La commande `titlepage` permet elle d'afficher un titre centré dans l'image.

```
integrale=["calculer :"
"    b ";
"   /  ";
"  |   ";
"  |   f(x) dx  ";
"  |  ";
" /  ";
" a "];
clf()
titlepage(integrale)
xtitle('titre principal',..
'Axe X','Axe Y',boxed = 1)
```



Enfin il est aussi possible de faire apparaître des informations dans la fenêtre graphique en utilisant la ligne qui se trouve tout en bas de cette fenêtre (la barre de message) et la commande `xinfo` .

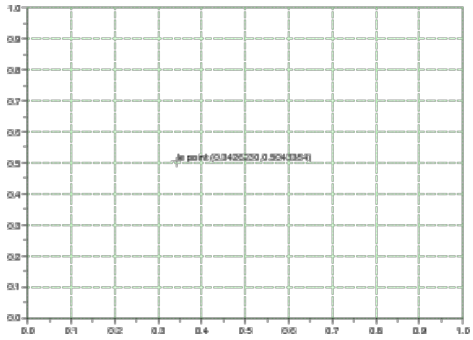
```
xinfo('on peut ajouter des informations supplémentaires ici')
```

4.2 Interactions

Si on a besoin de récupérer des informations avec un clic de souris (ou par une saisie au clavier) dans une des fenêtres graphiques existantes on peut, après avoir sélectionné cette fenêtre, utiliser `xclick`. Si une fenêtre graphique est ouverte (et pas iconifiée) la commande

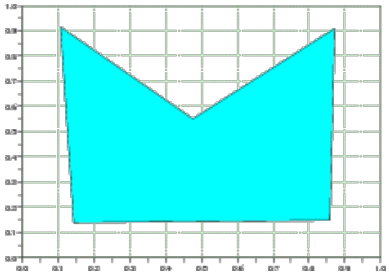
`[c_i,c_x,c_y,c_w,c_m]=xclick()` suivi permet de récupérer, après une action sur les boutons de la souris, la position du curseur `( c_x,c_y)` ainsi que le type d'action effectuée (pression, clic, double clic, sur quel bouton . . .) :

```
plot2d(0,0,0,rect=[0,0,1,1])
xgrid(3)
[c_i,c_x,c_y,c_w,c_m]=xclick() //cliquer dans la fenêtre
plot2d(c_x,c_y,-1)//placer une croix
xstring(c_x,c_y,'le point ('+string(c_x)+','+string(c_y)+')')
```



Dans le même style la fonction `locate` permet de récupérer une liste de points pour tracer un polygone par exemple :

```
clf()//fenêtre graphique 0
plot2d(0,0,0,rect=[0,0,1,1])
xgrid(3)
//cliquer 5 fois dans la fenêtre 0
X=locate(5)
xfpolys(X(1,:)',X(2,:)',4)
```



Si l'on veut repérer non seulement les clics mais aussi les mouvements de la souris il faut utiliser la fonction `xgetmouse` . Cette fonction va renvoyer les coordonnées du curseur (lorsqu'il se trouve dans la fenêtre graphique courante) non seulement lors d'un clic mais aussi à chaque mouvement de la souris. Voici un exemple pour comprendre :

```
function [x,y]=coordonnes()
rep=-ones(1,3)
//bouger la souris jusqu'à un clic
while rep(3)==-1
  rep=xgetmouse(0)
  x=rep(1);y=rep(2);
  //affichage des coordonnées
  txt='('+string(x)+','
  txt=txt+string(y)+')'
  info(txt)
end
//(x,y)=coordonnées finales
endfunction
```

```
function select_rectangle()
//coordonnées du premier coin
[x0,y0]=coordonnes()
//tracé du rectangle
xrect(x0,y0,0,0)
r=gce()//handle du rectangle
r.line_style=2//pointillé
r.thickness=3//épaisseur
rep=-ones(1,3)
//bouger la souris jusqu'à un clic
while rep(3)==-1
  rep=xgetmouse(0)
  x=rep(1);y=rep(2);
  x1=min(x0,x),y1=max(y0,y)
  //dimension du rectangle
  w=abs(x0-x),h=abs(y0-y)
  //modif du rectangle
  r.data=[x1,y1,w,h]
end
endfunction
```

il ne reste plus qu'à appeler la fonction `select_rectangle()` pour sélectionner à la souris un rectangle dans la fenêtre graphique courante :

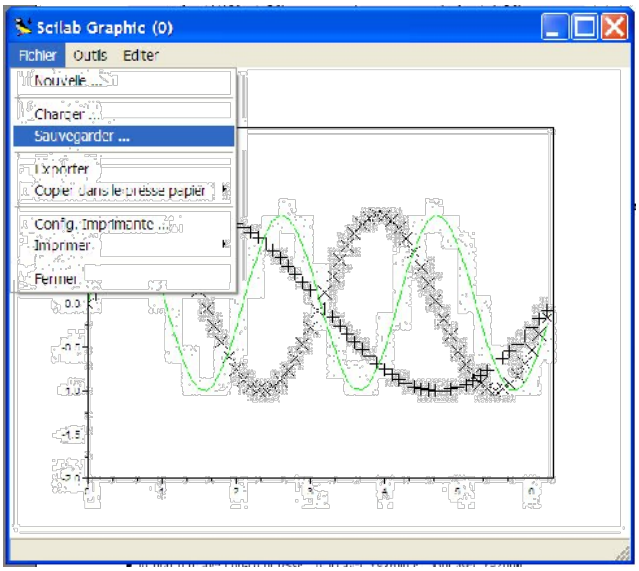
```
clf()
plot2d(0,0,0,rect=[0,0,1,1])
xgrid(3)
select_rectangle()
```

4.3 Exportation et sauvegarde des graphiques

Dans le nouveau mode graphique, les figures sont des variables de type handle qui n'existe que tant que la fenêtre graphique correspondante existe. Comme ce sont des variables, on peut les sauver dans un fichier binaire, comme n'importe quelle variable *Scilab*, en utilisant la fonction `save`, pour ensuite la recharger dans *Scilab* avec la commande `load` :

```
plot2d()//une figure
f=gcf()//figure courante
save('figure.sav',f)//sauver f dans un fichier
delete(f)//détruit la fenêtre courante
load('figure.sav')//charger la variable f
```

Après la commande `load` la figure sauvée dans le fichier `figure.sav` apparaîtra dans la fenêtre courante ou dans une nouvelle fenêtre. On peut aussi sauvegarder le contenu d'une fenêtre graphique dans un format spécifique `scg` à partir du menu `fichier`, onglet `sauvegarder` de la fenêtre graphique :



on pourra après recharger cette figure depuis l'onglet `charger` du même menu. Il est aussi possible d'exporter les figures obtenues vers des formats courants bitmap, jpeg, gif . . .soit depuis l'onglet `exporter` du menu `fichier` la fenêtre graphique, soit directement depuis la console en utilisant les fonctions :

- format d'image compressé :gif avec `xs2gif` jpeg avec `xs2jpg`
- format d'image non-compressé : bmp avec `xs2bmp` et ppm avec `xs2ppm`
- format d'image vectoriel : post-script (encapsulé) avec `xs2eps` mais aussi format Xfig avec `xs2fig`

ces fonctions prennent en argument le numéro de la fenêtre graphique à exporter et une chaîne de caractères donnant le nom du fichier à créer :

```
id=winsid()//liste des fenêtres
scf(id($))//dernière fenêtre devient la fenêtre courante
clf()//efface la fenêtre courante
plot2d()//nouvelle figure dans la fenêtre courante
xs2gif(id($),'figure.gif')//sauvegarde de la dernière fenêtre
```

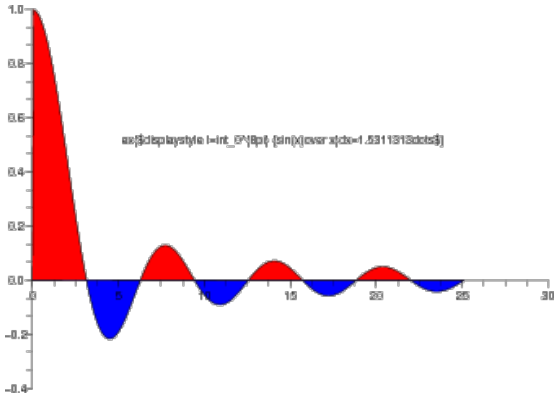
L'exportation au format `*.eps` sera particulièrement utile pour les utilisateurs de LaTeX (PDFTEXen l'occurence).

Dans certain cas, il arrive que la bounding box du fichier `*.eps` ne soit pas correctement calculée. Il faudra alors récalculer la bounding box avec `ps2epsi` :
`ps2epsi image1.eps image2.eps`
parfois cela s'avère insuffisant, des objets non-visibles dépassant de la bounding box, dans ce cas on utilisera en plus `convert` de Image Magick[5] pour réécrire le fichier post-script :
`convert image2.eps image3.eps`
mais on aura dans ce cas une perte de qualité au niveau des entités de type Text présentes dans l'image.

On peut aussi inclure du code LaTeX dans la figure avec la fonction `xstring`, mais il faut encapsuler le code LaTeX à l'intérieur d'une commande `\tex{ }` pour que le texte soit identifié comme du code LaTeX dans le fichier `*.eps`. Par exemple le code *Scilab* suivant :

```
clf()
rect=[0,-0.4,30,1]
plot2d(0,0,0,rect=rect,frameflag=5,axesflag=5)
//la  figure
ieee(2)
deff('y=f(x)','y=sin(x)./x')//fonction sinus cardinal
x=[0:0.05:8*%pi]'
y1=max(feval(x,f),zeros(x))
y2=min(feval(x,f),zeros(x))
xfpolys([x x; 0 0],[y1 y2; 0 0],[5,2])
I=integrate('f(x)','x',0,8*%pi)//valeur exacte
//code latex à insérer
latex='\tex{$\displaystyle I=\int_0^{8\pi}..
{sin(x)\over x}dx='+string(I)+'\dots$}'
xstring(5,0.5,latex)
```

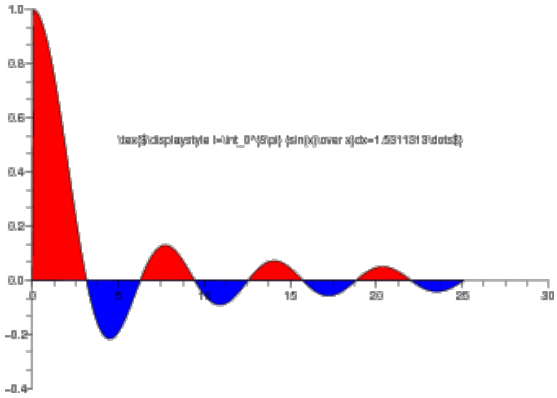
si vous obtenez la figure suivante où le caractère `\` a disparu :



c'est que `\` est un caractère « actif »lors de l'écriture du fichier `*.eps` il faut alors « échapper »le `\` pour qu'il apparaisse dans le fichier `*.eps` :

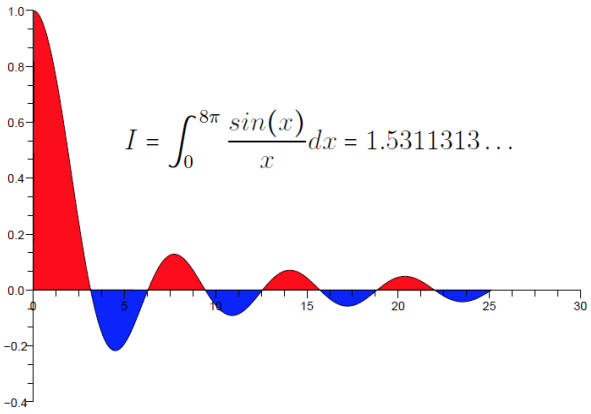
```
clf()
rect=[0,-0.4,30,1]
plot2d(0,0,0,rect=rect,frameflag=5,axesflag=5)
//la  figure
ieee(2)
deff('y=f(x)','y=sin(x)./x')//fonction sinus cardinal
x=[0:0.05:8*%pi]'
y1=max(feval(x,f),zeros(x))
y2=min(feval(x,f),zeros(x))
xfpolys([x x; 0 0],[y1 y2; 0 0],[5,2])
I=integrate('f(x)','x',0,8*%pi)//valeur exacte
//code latex à insérer
latex='\\tex{$\\displaystyle I=\\int_0^{8\\pi}..
{sin(x)\\over x}dx='+string(I)+'\\dots$}'
xstring(5,0.5,latex)
```

pour obtenir :



Il faudra après utiliser le package `psfrag` pour dire à LaTeX de compiler le code se trouvant dans la figure. Pratiquement il suffit d'ajouter `\usepackage{psfrag}` dans l'entête puis `\psfragscanon` avant la commande `\includegraphics`. On peut aussi produire l'image à partir d'un fichier LaTeX indépendant comme celui-ci :

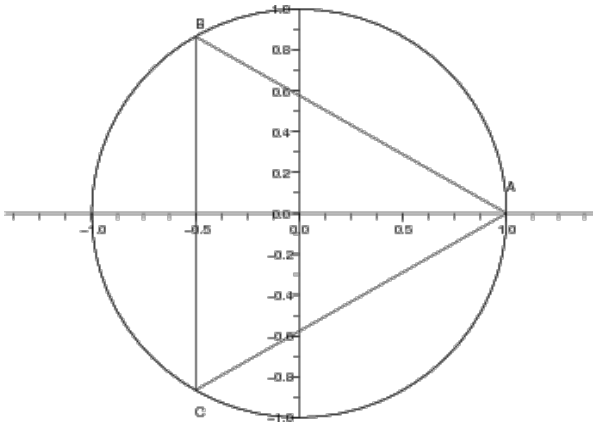
```
\documentclass{minimal}
\usepackage{amsmath,amssymb,graphicx,psfrag}
\begin{document}
\psfragscanon
\includegraphics[width=0.75\textwidth]{figure.eps}
\end{document}
```



Une autre manière de procéder avec le package `psfrag` consiste à n'inclure dans la figure qu'un label

```
clf()
rect=[-1,-1,1,1]
plot2d(0,0,0,rect=rect,frameflag=3,axesflag=5)
//la  figure
xrpoly([0,0],3,1)
xarc(-1,1,2,2,0,64*360)
xarc(-1,1,2,2,0,64*360)
xstring(1,0.1,'A')//label A
xstring(-0.5,0.9,'B')//label B
xstring(-0.5,-1,'C')//label C
xs2eps(0,'export4.eps')
```

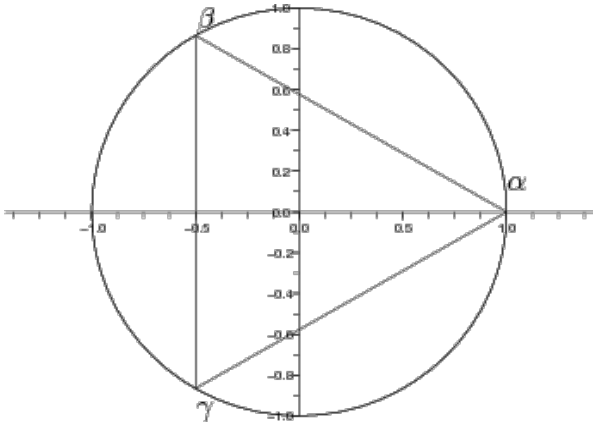
ce qui donne la figure :



ensuite il faut indiquer à LaTeX par quoi remplacer ce label lors de la compilation.

```
\begin{psfrags}
\psfrag{A}{$\alpha$}%%remplace A par $\alpha$
\psfrag{B}{$\beta$}%%remplace B par $\beta$
\psfrag{C}{$\gamma$}%%remplace C par $\gamma$
\includegraphics[width=0.7\textwidth]{export4}
\end{psfrags}
```

ce qui donnera après compilation :

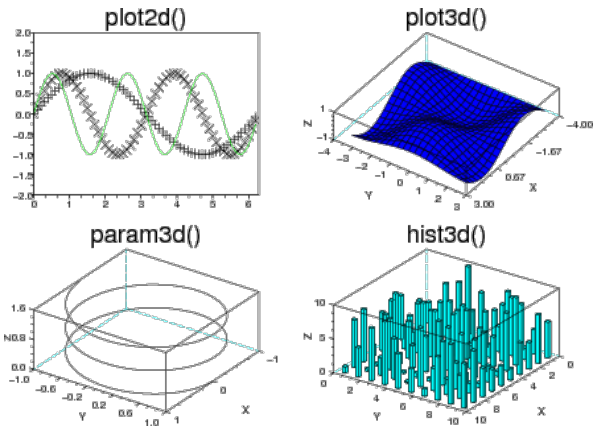


On peut finir en signalant que la fonction `subplot` permet de placer plusieurs figures dans une même figure. Cette fonction s'utilise avec la syntaxe `subplot(p,n,m)` ou `subplot(pnm)`. Dans ce cas la figure est considérée comme une « matrice de figures »à *p* lignes et *n* colonnes. Après l'instruction `subplot` les commandes graphiques sont envoyées dans la $m^{1ème}$ case de cette « matrice » :

```
clf()
f=gcf()
subplot(221)
plot2d()
xtitle('plot2d()')
subplot(222)
plot3d()
xtitle('plot3d()')
subplot(2,2,3)
param3d()
xtitle('param3d()')
subplot(2,2,4)
hist3d()
xtitle('hist3d()')
```

```
f.children.title.font_size=5;
```

En fait chaque sous-figure de *f* correspond à une entité Axes.



4.4 Animations

Scilab permet de générer des animations en enchaînant plusieurs figures générées avec les commandes vus dans cet article. Mais la réalisation d'une animation de bonne qualité nécessite encore de résoudre quelques problèmes. Pour comprendre nous allons étudier un exemple très simple : faire se balancer un pendule constitué d'une boule de diamètre 0.2 (crée avec `xfarc`) suspendue au bout d'un fil de longueur 1 (créé avec `plot2d`) accroché au point de coordonnées 0,0 :

```
//initialisation de la fenêtre graphique
clf()
rect=[-1.2,-1.2,1.2,0.2];//taille de la fenêtre
plot2d(0,0,1,rect=rect,frameflag=3,axesflag=5)
//paramètres de l'animation
w=2*%pi/50//fréquence de balancement
angle0=%pi/4//angle maximum
l=1//longueur du fil
d=0.2//diamètre de la boule
for t=0:200//la boucle de l'animation
    angle=angle0*cos(w*t)
    x=l*sin(angle),y=-l*cos(angle)
    clf()
    xfarc(x-d/2,y+d/2,d,d,0,64*360)//la boule
    plot2d([0;x],[0;y],1,rect=rect,frameflag=3,axesflag=5)//le fil
    xinfo('t='+string(t))//voir le temps
end
```

Cette animation n'est pas très esthétique car l'affichage semble saccadé en particulier par ce qu'on voit clignoter la barre d'outils (la barre contenant, entre autre, les icônes de zoom). On peut facilement résoudre ce problème avec la fonction `toolbar(num,etat)`, qui montre ou cache la barre d'outils de la fenêtre `num` suivant que `etat` est à `'on'` ou `'off'` :

```
//initialisation de la fenêtre graphique
clf()
f=gcf()
toolbar(f.figure_id,'off')
rect=[-1.2,-1.2,1.2,0.2];//taille de la fenêtre
```

```
plot2d(0,0,1,rect=rect,frameflag=3,axesflag=5)
//paramètres de l'animation
w=2*%pi/50//fréquence de balancement
angle0=%pi/4//angle maximum
l=1//longueur du fil
d=0.2//diamètre de la boule
for t=0:200//la boucle de l'animation
    angle=angle0*cos(w*t)
    x=l*sin(angle),y=-l*cos(angle)
    clf()
    xfarc(x-d/2,y+d/2,d,d,0,64*360)//la boule
    plot2d([0;x],[0;y],1,rect=rect,frameflag=3,axesflag=5)//le fil
    xinfo('t='+string(t))//voir le temps
end
```

Mais là encore l'animation semble saccadée. La raison vient du léger décalage qui existe entre le moment où la fenêtre est effacée (`clf()` dans la boucle `for`) et celui ou le pendule est affiché (exécution de `xfarc` puis de `plot2d`). La résolution de ce problème réside dans l'utilisation d'un mode d'affichage spécial : le mode Pixmap. Dans ce mode, les figures ne sont pas directement affichées dans la fenêtre graphique. Elles sont d'abord affichées dans une autre fenêtre, qu'on appelle le *buffer graphique*, avant de basculer l'affichage du buffer vers la fenêtre graphique courante. Le buffer est une fenêtre graphique virtuelle, c'est à dire qu'elle n'apparaît pas à l'écran. L'intérêt du buffer est de laisser le temps à *Scilab* de lire les instructions graphiques avant de les afficher à l'écran, ceci permet de supprimer l'aspect saccadé de l'affichage dans les animations. Il y a trois petites choses à retenir pour utiliser ce mode pixmap :

- le handle de chaque figure possède une propriété `pixmap` qui vaut `'on'` ou `'off'` suivant que la figure est en mode pixmap ou pas,
- la fonction `show_pixmap` permet de basculer le contenu du buffer graphique dans la fenêtre courante,
- en mode pixmap la fonction `clf()` efface le contenu du buffer graphique mais pas le contenu de la fenêtre courante, celui-ci ne sera effacé qu'au prochain appel de `show_pixmap`.

ceci nous permet de créer une animation vraiment fluide (suivant la fréquence `w` choisie !) :

```
//initialisation de la fenêtre graphique
clf()
f=gcf()//handle figure courante
toolbar(f.figure_id,'off')//supprime la barre d'outils
rect=[-1.2,-1.2,1.2,0.2];//taille de la fenêtre
plot2d(0,0,1,rect=rect,frameflag=3,axesflag=5)
//paramètres de l'animation
w=2*%pi/150//fréquence de balancement
angle0=%pi/4//angle maximum
l=1//longueur du fil
d=0.2//diamètre de la boule
f.pixmap='on'//démarre le mode pixmap
for t=0:200//la boucle de l'animation
    angle=angle0*cos(w*t)
    x=l*sin(angle),y=-l*cos(angle)
    clf()//efface le buffer
    xfarc(x-d/2,y+d/2,d,d,0,64*360)//la boule
    plot2d([0;x],[0;y],1,rect=rect,frameflag=3,axesflag=5)//le fil
```

```
    show_pixmap()//affichage du contenu du buffer dans la fenêtre
    xinfo('t='+string(t))//voir le temps
end
f.pixmap='off'//arrête le mode pixmap
```

il existe un autre mode d'affichage comparable au mode pixmap. Vous avez certainement remarqué que chaque modification d'un handle de figure provoque une mise à jour de l'affichage dans la fenêtre graphique. Ceci peut être la cause d'un ralentissement important du fonctionnement de *Scilab* si la figure contient beaucoup d'instructions graphiques. Il existe plusieurs manières de différer l'affichage d'un graphique le temps de faire certaines modifications. La méthode repose sur la propriété `immediate_drawing` de la figure. Cette propriété gère si l'affichage doit être immédiat ou différé suivant que sa valeur est à `'on'` ou à `'off'`. On a aussi les deux fonctions `drawlater()` et `drawnow()` qui modifient la valeur de la propriété `immediate_drawing` de la figure courante. On pourra utiliser de manières équivalentes les deux schémas suivants :

```
drawlater()
...//instructions graphiques
drawnow()
```

```
f=gcf()//figure courante
f.immediate_drawing='off'
...//instructions graphiques
f.immediate_drawing='on'
```

`drawlater()` et `drawnow()` agissent sur la fenêtre courante, il faut prendre garde à ne pas changer de figure courante entre les instructions `drawlater()` et `drawnow()` !

Pour en finir avec les animations, on peut avoir besoin d'exporter une animation réalisé avec *Scilab* pour l'insérer dans une page web ou une présentation power-point ou OOImpress. Dans ce cas la solution la plus simple consiste à créer un "gif animé", pour cela il suffit d'exporter au format gif chaque figure composant l'animation puis de fabriquer le gif animé à partir des différentes images sauvegardées. Pour sauver nos figures sous forme d'images gif on peut utiliser `xs2gif` :

```
//initialisation de la fenêtre graphique
clf()
f=gcf()//handle figure courante
toolbar(f.figure_id,'off')//supprime la barre d'outils
rect=[-1.2,-1.2,1.2,0.2];//taille de la fenêtre
plot2d(0,0,1,rect=rect,frameflag=3,axesflag=5)
//paramètres de l'animation
w=2*%pi/150//fréquence de balancement
angle0=%pi/4//angle maximum
l=1//longueur du fil
d=0.2//diamètre de la boule
f.pixmap='on'//démarrer le mode pixmap
for t=0:200//la boucle de l'animation
    angle=angle0*cos(w*t)
    x=l*sin(angle),y=-l*cos(angle)
    clf()//efface le buffer
    xfarc(x-d/2,y+d/2,d,d,0,64*360)//la boule
```

```
    plot2d([0;x],[0;y],1,rect=rect,frameflag=3,axesflag=5)//le fil
    show_pixmap()//affichage du contenu du buffer dans la fenêtre
    //chaîne de caractère donnant le nom du fichier gif
    fichier='imagesgif/animation'+string(1000+t)+'.gif';
    xs2gif(f.figure_id,fichier)//exportation de la figure
    xinfo('t='+string(t))//voir le temps
end
f.pixmap='off'//arrêter le mode pixmap
```

ici on suppose que le répertoire courant possède un répertoire `imagegif/` où seront stockées toutes les images gif. Ensuite on pourra utiliser divers logiciel pour créer un gif animé à partir des fichiers créés (de `animation1000.gif` jusqu'à `animation1200.gif`). Le plus simple consiste à utiliser l'utilitaire `convert` de ImageMagic [5] qui permet de créer un gif animé à partir d'une série d'images gif en une ligne de commande. Avec les images de l'exemple précédent il suffira de lancer la commande suivante depuis le répertoire `imagegif/` :

```
convert animation*.gif -delay 20 -loop 0 animation.gif
```

ce qui va créer, dans le répertoire `imagegif/`, un fichier `animation.gif` contenant chacune des images `animation*.gif` (soit de `animation1000.gif` jusqu'à animation1200.gif) affichées en boucle (`-loop 0`) pendant 20 centièmes de seconde chacune. On peut aussi lancer ImageMagic directement depuis scilab avec la commande `unix` :

```
unix('convert animation*.gif -delay 20 -loop 0 animation.gif')
```

Cela peut être pratique pour les utilisateurs de Windows™ peu habitués à lancer des opérations en ligne de commande. Pour que ça marche il faut que le répertoire courant de *Scilab* soit `imagegif/`. On trouvera d'autre exemples de créations de gif animés avec ImageMagic sur le site [6]

Références

[1] Bruno Pinçon *une introduction à Scilab* http://www.iecn.u-nancy.fr/~pincon/scilab/scilab.html

[2] J -P Chancelier, F Delebecque, C Gomez, M. Goursat, S. Steer, R. Nikoukhah, *Introduction à SCILAB*, Springer

[3] http://h0.web.u-psud.fr/orscilab/Spas402.html

[4] http://fr.wikibooks.org/wiki/Découvrir_Scilab/Graphiques_et_sons

[5] http://www.imagemagick.org/script/index.php

[6] http://www.imagemagick.org/Usage/anim_basics/