Leçon 3 : Graphiques

- **Exemples de graphiques simples**
- **Objet graphique**
- **Personnalisation des commandes graphiques**
- **Primitives graphiques**
- **Modifier les options par défaut standards**
- **Passer des options graphiques à une commande**
- **Difficultés usuelles en 2D**
- **Supplément sur les fonctions graphiques**

Mathematica permet de créer aisément des graphiques, puis de les transformer, les juxtaposer, les superposer, ou les animer en séquences. La création de graphiques à l'aide des commandes graphiques avec leurs paramètres par défaut est élémentaire. Toutefois, elle produit rarement des graphiques parfaits. Il faut donc personnaliser les graphiques, soit par l'utilisation des options des fonctions graphiques, soit par la superposition de primitives graphiques sur un objet graphique.

Il est possible avec *Mathematica* de contrôler dans le détail le tracé des graphiques et de générer des graphiques très complexes. Pourtant, il est fréquent de buter, même pour un graphique élémentaire, sur des difficultés coûteuses en temps. Un résultat final conforme au rendu désiré exige en effet l'emploi d'options ou de commandes annexes, éventuellement nombreuses, pour les commandes graphiques et aussi d'inévitables tâtonnements sur les valeurs adéquates des paramètres. Ces difficultés viennent essentiellement de ce que l'utilisateur se représente les graphiques comme des images et non comme des expressions symboliques de type *Mathematica*. Un des objectifs de cette leçon est en fait d'introduire le lecteur à la notion d'expression.

Des fonctions graphiques avancées sont disponibles soit dans le répertoire de fichiers de commandes Graphics livré en standard, soit dans des fichiers de commandes spécialisés (voir, par exemple, la disquette d'accompagnement du livre *Mathematica Graphics* de T. Wickham Jones).

Les principales catégories de graphiques sont les graphiques en deux dimensions ou en trois dimensions (en fait, en pseudo-trois dimensions), les graphiques paramétriques, les graphiques de contour et les graphiques de densité, enfin les graphiques en tableau *(array)*. Des commandes graphiques comme `Plot` ou `Plot3D` s'appliquent à des fonctions, d'autres, comme `ListPlot` ou `ListPlot3D`, à des listes de valeurs. Les commandes graphiques génèrent les différents types de graphiques suivants :

`Graphics`
`Graphics[primitives, options]` représente un graphique à deux dimensions

`Graphics3D`
`Graphics3D[primitives, options]` représente un objet graphique à trois dimensions

`SurfaceGraphics`
`SurfaceGraphics[liste]` est la représentation d'une surface tridimensionnelle

`DensityGraphics`
`DensityGraphics[liste]` représente un graphique de densité

`ContourGraphics`
`ContourGraphics[liste]` représente un graphique de courbes de niveaux

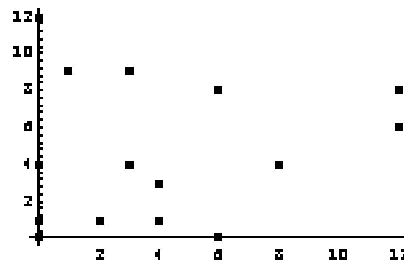
`GraphicsArray`
`GraphicsArray[{g1, g2,...}]` représente un tableau de graphiques sur une ligne
`GraphicsArray[{{g11, g12,...}, ...}]` représente un tableau à deux dimensions.

Types de graphiques et primitives graphiques.

■ Exemples de graphiques simples

Une liste de doublets est créée et nommée.
Elle est représentée par des points au moyen de **`ListPlot`** qui interprète les doublets comme des coordonnées {x, y}.
L'option **`Prolog`** est exécutée avant le tracé et permet ici de fixer la taille du dessin des points ; la valeur **`0.02`** attribuée est une valeur relative qui signifie que les points doivent faire 2 % de la taille totale du graphique.

```
points = {{0,1},{0,4},{1,9},{0,12},{6,8},{2,1},
{0,0},{4,3},{8,4},{0,1},{6,0},{3,9},{3,4},{4,1},
{0,4},{12,6},{12,8}};
ListPlot[points, Prolog->{PointSize[0.02]}]
```

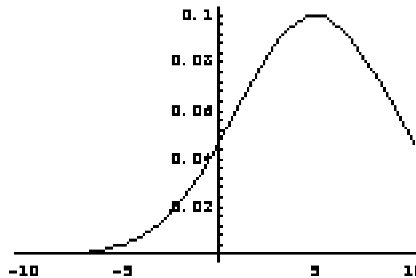


En plus du dessin de l'objet graphique, le type de l'objet est affiché, ici le type `Graphics`.

```
-Graphics-
```

Le fichier de commandes contenant la définition de la loi normale est chargé.
Le graphique est produit par trois fonctions imbriquées. La commande **Plot** est utilisée car il s'agit de représenter une fonction (en fait une fonction de fonction)
et non une liste de valeurs.
La fonction **PDF[Distribution, x]** est la fonction de densité de probabilité d'une distribution statistique évaluée au point **x**.

```
<<Statistics`NormalDistribution`
Plot[PDF[NormalDistribution[5,4], x], {x,-10,10}]
```

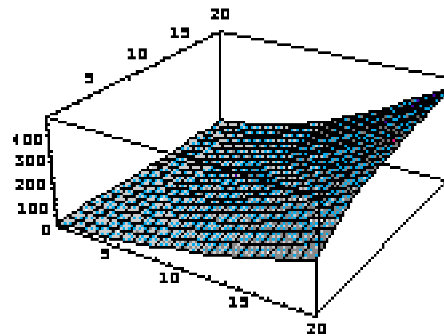


-Graphics-

Une liste de valeurs est créée et nommée.
Elle est passée à la fonction **ListPlot3D** qui l'interprète comme une liste de « hauteurs » et représente une surface dans l'espace.
Remarquez que la liste utilisée doit être « carrée ».

L'objet affiché est de type SurfaceGraphics.

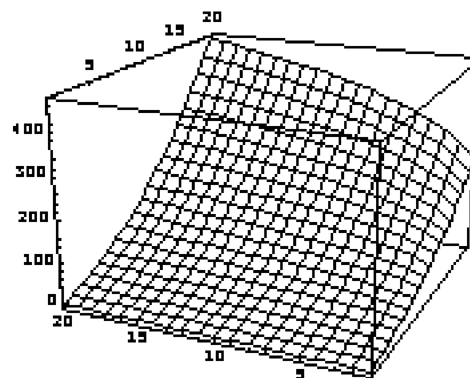
```
val = Table[i^(1/4) j^(9/5), {i,1,20}, {j,1,20}];
ListPlot3D[val]
```



-SurfaceGraphics-

La surface précédente est mal lisible. En modifiant les options par défaut concernant le point de vue et les proportions de la boîte, on peut obtenir un dessin plus explicite.
Les valeurs de **ViewPoint** ont été entrées par l'intermédiaire du sous-menu
3D ViewPoint Selector .
Notez l'emploi de **%** qui représente le dernier résultat obtenu.

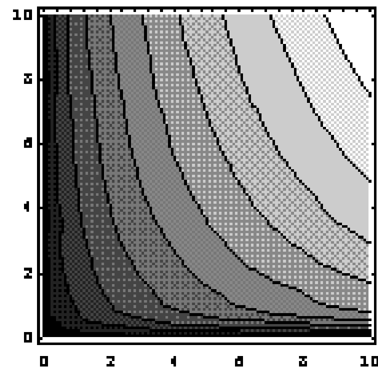
```
Show[%, ViewPoint->{-2.911,-1.325,1.104},
Shading->False, BoxRatios->{1,1,2/3}]
```



-SurfaceGraphics-

Une fonction de type Cobb-Douglas est représentée sous forme de graphique de contour, c'est-à-dire de courbes de niveaux.

```
cd = 10 k^(1/2) l^(1/4);
contourcd = ContourPlot[cd, {k,0,10}, {l,0,10}]
```

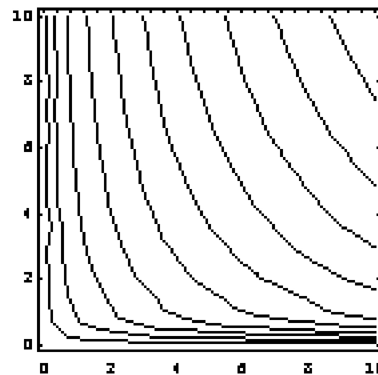


Le graphique est de type `ContourGraphics`.

```
-ContourGraphics-
```

En supprimant les couleurs, les courbes de niveaux deviennent plus lisibles et montrent alors des irrégularités anormales.

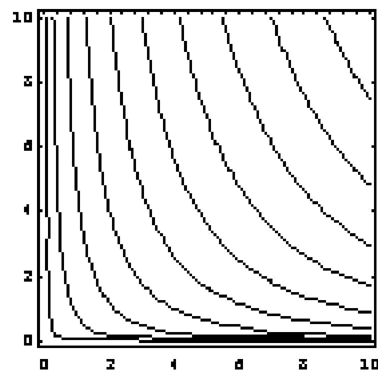
```
Show[contourcd, ContourShading->False]
```



```
-ContourGraphics-
```

Ces irrégularités sont résorbées en augmentant la valeur de l'option **PlotPoints** qui définit le nombre de points où la fonction à représenter est échantillonnée.

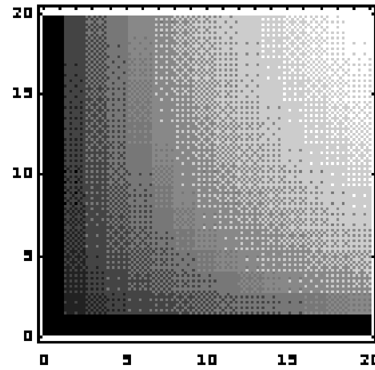
```
contourcd2 = ContourPlot[cd, {k,0,10}, {l,0,10},
ContourShading->False, PlotPoints->50]
```



```
-ContourGraphics-
```

La même fonction est maintenant représentée par un graphique de densité.

```
densitecd = DensityPlot[cd, {k,0,20}, {l,0,20},
Mesh->False]
```



Le type du graphique est `DensityGraphics`.

```
-DensityGraphics-
```

Le fichier de commandes contenant les définitions des distributions statistiques continues est chargé.
Quatre fonctions de densité de probabilité sont définies pour quatre distributions différentes.

```
<<Statistics`ContinuousDistributions`
mu = 0;sigma = 1;
pdf1 = PDF[NormalDistribution[mu,sigma], x];
pdf2 = PDF[LogisticDistribution[mu,sigma], x];
pdf3 = PDF[LaplaceDistribution[mu,sigma], x];
pdf4 = PDF[LogNormalDistribution[mu,sigma], x];
```

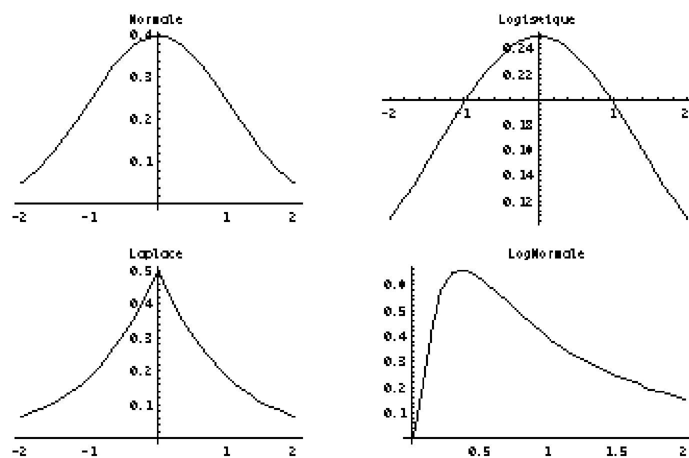
Quatre objets graphiques représentant les fonctions de densité de probabilité sont créés sans être affichés ; l'option **`DisplayFunction->Identity`** permet en effet d'éviter l'affichage d'un objet graphique.

```
gr1 = Plot[pdf1, {x,-2,2}, PlotLabel->"Normale",
DisplayFunction->Identity,
DefaultFont->{"Monaco",6}];
gr2 = Plot[pdf2, {x,-2,2}, PlotLabel->"Logistique",
DisplayFunction->Identity,
DefaultFont->{"Monaco",6}];
gr3 = Plot[pdf3, {x,-2,2}, PlotLabel->"Laplace",
DisplayFunction->Identity,
DefaultFont->{"Monaco",6}];
gr4 = Plot[pdf4, {x,0.01,2}, PlotLabel->
"LogNormale",
DisplayFunction->Identity,
DefaultFont->{"Monaco",6}];
```

Les quatre objets graphiques sont combinés dans un nouvel objet du type `GraphicsArray`. L'affichage de cet objet doit être demandé explicitement par la commande **`Show`**.

```
GraphicsArray[{{gr1,gr2}, {gr3,gr4}}];
Show[%]
```

Les quatre graphiques sont combinés en tableau. Chaque graphique est doté de son titre propre introduit antérieurement par l'option **`PlotLabel`**. L'option **`DefaultFont`** a permis de fixer la police de texte en Monaco 6 points.



L'objet créé est du type `GraphicsArray`.

```
-GraphicsArray-
```

Objet graphique

Avant d'être un dessin sur l'écran ou sur le papier d'une imprimante, un graphique est d'abord, pour le système, un objet particulier de type graphique. Il faut donc garder à l'esprit cette distinction entre l'objet et son image. L'objet est un ensemble structuré d'informations, dont l'image est la représentation visuelle. Cette dissociation permet la manipulation de l'objet graphique en tant que tel et de façon indépendante de tout support d'affichage.

Un graphique est obtenu en trois étapes : une commande graphique comme `Plot` traite ses arguments, elle renvoie comme résultat une expression *Mathematica* sous la forme interne au noyau, l'interface transforme l'expression *Mathematica* en fichier Postscript affichable ou imprimable sur un périphérique Postscript (écran ou imprimante).

Objet graphique comme expression *Mathematica*

Un objet graphique est une expression *Mathematica* d'une forme particulière. Cette expression comprend en général des primitives graphiques, appliquées à des valeurs numériques sous le contrôle d'options de tracé, le tout étant englobé dans un en-tête de type graphique. Plutôt que de définir ces termes d'une façon abstraite, examinons un exemple.

L'objet graphique **`graph`** est créé mais sans être affiché. Néanmoins, l'afficheur renvoie `Graphics` qui indique qu'un objet graphique a bien été créé.

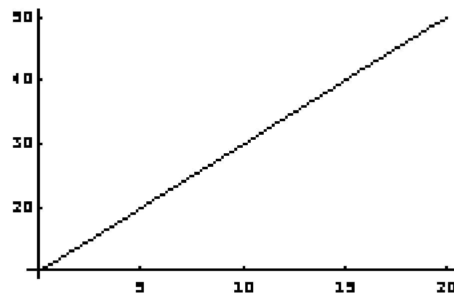
```
graph = Plot[2 x + 10, {x,0,20},
DisplayFunction->Identity]
-Graphics-
```

La commande **`Show`** ne suffit pas à faire s'afficher le graphique.

```
Show[graph]
-Graphics-
```

Pour restaurer l'affichage, il faut changer l'option **`DisplayFunction->Identity`** en **`DisplayFunction->$Display\ Function`**. Cette écriture sibylline signifie que l'option prend pour valeur la valeur de la variable système **`$DisplayFunction`**. La valeur par défaut de **`$DisplayFunction`** induit l'affichage à l'écran.

```
Show[graph,
DisplayFunction->$DisplayFunction]
```




```
-Graphics-
```

La différence entre l'objet graphique et son image est maintenant évidente. L'image est la représentation des composants ou données d'un objet de type graphique. Un tel objet est une expression *Mathematica* dotée d'une structure particulière. La commande `FullForm[graph]` nous montre la structure complète de `graph` en écriture interne, c'est-à-dire selon le code traité par le noyau : son en-tête, la liste des valeurs (liste de listes), la liste des options avec leurs valeurs. Pour une vision plus réduite, on peut préférer utiliser la commande `InputForm`.

FullForm permet de voir l'objet graphique sous la forme où il est connu par le noyau. C'est une longue expression présentée ici de façon tronquée.
On reconnaît diverses commandes, notamment : **Graphics**, **List**, **Line**, **Rule**.

On aurait pu limiter l'affichage des décimales en écrivant plutôt la commande **NumberForm[FullForm[graph], 5]**.

```
FullForm[graph]
Graphics[List[List[Line[List[List[0., 10.],
List[0.8333333333333333333, 11.66666666666666667],
List[1.666666666666666667, 13.33333333333333333],
List[2.5, 15.],
[..........................]
List[19.16666666666666667, 48.33333333333333333],
List[20., 50.]]]]],
List[Rule[PlotRange, Automatic],
Rule[AspectRatio, Power[GoldenRatio, -1]],
RuleDelayed[DisplayFunction, Identity],
Rule[ColorOutput, Automatic], Rule[Axes,
Automatic], Rule[AxesOrigin, Automatic],
[.......................]
 Rule[FrameTicks, Automatic], Rule[FrameLabel,
None], Rule[PlotRegion, Automatic]]]
```

Le même objet **graph** tel qu'il est représenté au format d'entrée (Input).
La vocation de ce format est de servir à l'utilisateur pour entrer ses commandes.
On voit que par rapport à l'écriture interne, List[a, b] est converti en {a, b}, tandis que Rule[a, b] devient a->b.

```
InputForm[graph]
Graphics[{{Line[{{0., 10.},
{0.8333333333333333333, 11.66666666666666667},
{1.666666666666666667, 13.33333333333333333},
{2.5, 15.},
[..........................]
{19.16666666666666667, 48.33333333333333333},
{20., 50.}}]}},
{PlotRange -> Automatic,
AspectRatio -> GoldenRatio^(-1),
DisplayFunction :> $DisplayFunction,
ColorOutput -> Automatic,
Axes -> Automatic, AxesOrigin -> Automatic,
[..........................]
FrameTicks -> Automatic, FrameLabel -> None,
PlotRegion -> Automatic}]
```

OutputForm renvoie le type du graphique.

```
OutputForm[graph]
-Graphics-
```

L'expression de graph est en fait constituée de trois morceaux dont le rôle respectif est assez évident. La forme graph[[n]] extraie la partie n de l'expression graph. Par suite : graph[[0]] détermine l'en-tête de l'objet, graph[[1]] est la partie qui contient les primitives graphiques et les coordonnées, graph[[2]] contient les options de tracé.

Vérification de l'en-tête soit comme partie de rang **0**, soit explicitement comme en-tête.
L'en-tête est celui d'un objet graphique.

```
graph[[0]]
Graphics
Head[graph]
Graphics
```

La partie **1** de **graph** est la primitive graphique **Line** qui trace une ligne entre les points dont les coordonnées lui sont passées sous forme de listes comme arguments.

```
graph[[1]]
{{Line[{{0., 10.}, {0.833333, 11.6667},
{1.66667, 13.3333}, {2.5, 15.},
[..........................]
{19.1667, 48.3333}, {20., 50.}}]}}
```

La partie **2** de **graph** est la liste des options de tracé avec les valeurs de ces options.

```
graph[[2]]
{PlotRange -> Automatic,
AspectRatio ->1/GoldenRatio,
DisplayFunction :> $DisplayFunction,
ColorOutput -> Automatic,
```

Par défaut, les options prennent souvent des valeurs comme `Automatic` ou `None`.

```
Axes -> Automatic, AxesOrigin -> Automatic,
[...........................]
FrameTicks -> Automatic, FrameLabel -> None,
PlotRegion -> Automatic}
```

Le rendu d'un objet graphique renvoie deux cellules : en premier, la cellule contenant le graphique, en second, une cellule qui spécifie le type de l'objet graphique créé (`Graphics`, `SurfaceGraphics`, `GraphicsArray`, `DensityGraphics` ou `ContourGraphics`). Ce type correspond en fait à l'en-tête de l'expression interne du graphique.

■ Manipulation d'un objet graphique sous forme interne

La forme interne d'un objet graphique est une expression *Mathematica*. Par définition, toute expression *Mathematica* est manipulable jusqu'au niveau des constituants élémentaires ou *atomes* (voir la Leçon 4). Les éléments que l'on souhaite manipuler sont accessibles en fonction de leur position ou de leur type. Connaître les positions des éléments visés peut sembler d'abord difficile, mais il suffit de demander la position d'un élément situé au niveau recherché pour obtenir une indication suffisante. Par exemple, supposons que l'on veuille extraire de l'objet `graph` la liste des coordonnées exactes des points représentés.

Un terme quelconque des coordonnées de **`graph`** est choisi. **`Position`** nous indique que ce terme est le vingt-cinquième d'une liste indicée `{1,1,1,1}`. L'extraction des valeurs est directe connaissant la position de la liste.

```
Position[graph, {20.,50.}]
{{1, 1, 1, 1, 25}}
```

Notez bien que le doublet **`{20.,50.}`** est écrit avec des points décimaux ; le doublet **`{20,50}`** n'existe pas dans la liste.

```
graph[[1,1,1,1]]
{{0., 10.}, {0.833333, 11.6667}, {1.66667,
13.3333}, {2.5, 15.},
[...........................]
{19.1667, 48.3333}, {20., 50.}}
```

Une solution plus directe est de transformer **`Line`** en **`List`** puis d'aplatir la liste de listes obtenue. La règle de substitution transforme **`Line`** en **`List`**. **`Flatten[%, 3]`** aplatit la liste au niveau adéquat.

```
graph[[1]]//Short
{{Line[{{0., 10.}, <<23>>, {20., 50.}}]}}
%/.Line->List//Short
{{{{{0., 10.}, <<23>>, {20., 50.}}}}}
Flatten[%, 3];
```

■ Objet graphique comme expression Postscript

Typiquement, un objet graphique sous forme interne (tel que montré par `FullForm`) sera converti en commandes Postscript avant d'être affiché ou imprimé et cette conversion sera automatique. Parfois, il est nécessaire de disposer d'un objet graphique sous forme Postscript, soit pour le modifier en manipulant les commandes Postscript, soit pour transmettre le graphique dans un fichier au format Postscript afin qu'il puisse être restitué sur une imprimante Postscript quelconque (il faudra alors utiliser l'utilitaire `psfix`, voir à ce sujet le manuel), soit pour insérer un graphique Postscript dans un logiciel autre que *Mathematica.*

Pour obtenir un objet graphique sous forme d'expression Postscript, plusieurs voies sont ouvertes. Première voie : sélectionner le graphique, le copier, puis, dans *Mathematica*, placer le curseur *dans une cellule vide*, faire `Paste` , les commandes Postscript du graphique apparaissent. Le texte peut alors être sélectionné et collé dans un traitement de texte. Remarquez que si on colle le graphique *comme cellule* (le curseur vertical étant remplacé par la ligne d'insertion horizontale), alors c'est l'image graphique qui apparaît. Deuxième voie : créer l'objet graphique en le nommant `graphps` par exemple, puis exécuter la commande `Display["fgraphps", graphps]`, l'objet `graphps` est alors enregistré sous forme

Postscript dans le fichier dénommé `fgraphps`. Ce fichier peut dès lors être transmis, relu et mis en image par un autre programme, relu et modifié dans un traitement de texte, etc. Si ce fichier est ouvert par *Mathematica*, les commandes Postscript apparaissent dans une cellule d'entrée (`Input`) dont l'exécution échoue. Pour voir l'image correspondant aux commandes, il faut sélectionner *la cellule* des commandes et lui donner le format `Graphics` par les menus `Style` , `Cell Style` et `Graphics` . Sous forme Postscript, les commandes peuvent être modifiées directement, pour autant que l'utilisateur connaisse ce langage.

■ Conversions entre types de graphiques

Un objet graphique est caractérisé par un type graphique particulier défini par la commande qui a créé cet objet. Les types graphiques sont : `Graphics`, `Graphics3D`, `SurfaceGraphics`, `DensityGraphics`, `ContourGraphics`, `GraphicsArray`. Ces différents types peuvent se convertir les uns dans les autres suivant la table ci-dessous. L'intérêt de ces conversions est soit de raccourcir les temps de calcul, puisque l'on passe d'une représentation à une autre en convertissant l'objet graphique initial sans qu'il soit recalculé, soit de combiner des graphiques de types différents sur une même image.

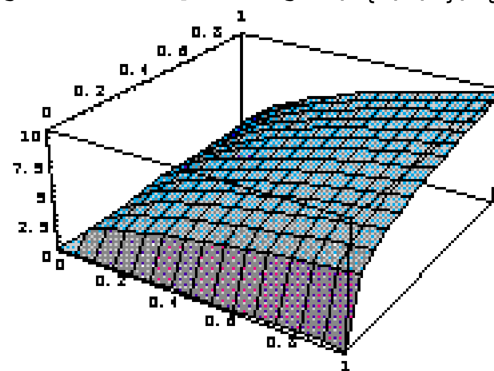
Graphics	
Graphics3D	Graphics
SurfaceGraphics	Graphics, Graphics3D, ContourGraphics, DensityGraphics
DensityGraphics	Graphics
ContourGraphics	Graphics, SurfaceGraphics, DensityGraphics
GraphicsArray	Graphics

Conversions graphiques (type initial et types possibles après conversion).

Montrons quelques exemples de conversion :

Une fonction de deux variables est représentée par une surface dans l'espace.
L'objet graphique correspondant est nommé **gr3D**.

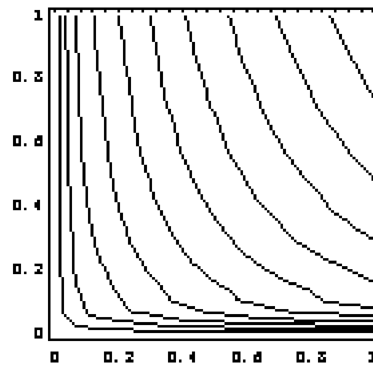
```
cobbdouglas = 10 k^(1/2) l^(1/4);
gr3D = Plot3D[cobbdouglas, {k,0,1}, {l,0,1}]
```



-SurfaceGraphics-

Le graphique de contour est obtenu directement par conversion de **gr3D** au moyen de la primitive graphique **ContourGraphics**.
L'avantage est un gain en temps de calcul par rapport à l'application de **ContourPlot** à **cobbdouglas**.

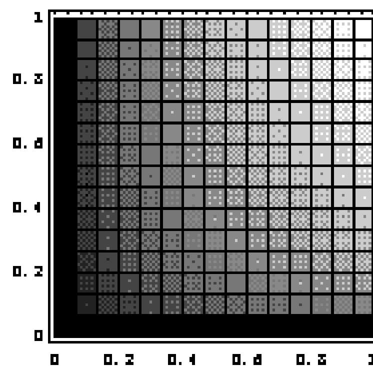
```
gr2D = Show[ContourGraphics[gr3D],
ContourShading->False,
ContourSmoothing->True]
```



-ContourGraphics-

De la même façon, un graphique de densité est obtenu directement par conversion de **gr3D** au moyen de la primitive graphique **DensityGraphics**.

```
Show[DensityGraphics[gr3D],
MeshStyle->{Thickness[0.001]}]
```

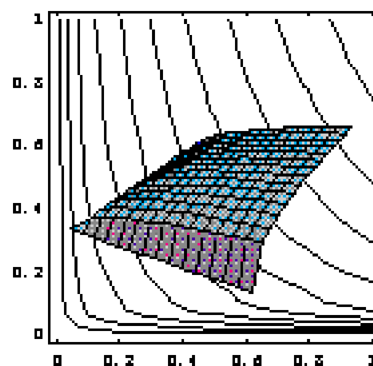


-DensityGraphics-

Il est possible de superposer des figures en deux dimensions et en pseudo-trois dimensions.

L'échelle relative des deux graphes s'avère ici satisfaisante car la conversion du type `SurfaceGraphics` se fait en ramenant les trois dimensions à l'échelle de 1, ce qui est aussi la taille de la représentation du graphique initial et de son dérivé de type `ContourGraphics`. Pour appréhender cette difficulté, essayez les mêmes commandes en remplaçant les bornes des itérateurs **k** et **l** par **10** dans la définition de **gr3D**.

```
Show[gr2D, gr3D]
```



-Graphics-

■ Personnalisation des commandes graphiques

■ Options, primitives, directives

Les *options* doivent être distinguées des *primitives* et des *directives*. Pour les différencier, observez que les options s'écrivent avec une flèche `->` ou `:>` (sous forme interne, ce sont des éléments avec `Rule` ou `RuleDelayed` comme en tête, voir ci-dessus), tandis que les primitives ou les directives sont des commandes qui tiennent leurs arguments entre crochets. En première approche, il est fréquent de confondre une option avec une primitive imbriquée ou d'ignorer l'usage des primitives et des directives en se limitant indûment aux possibilités de personnalisation offertes par les options.

□ Options

Les options des fonctions graphiques sont le moyen de base pour personnaliser les graphiques. Les options attribuent des valeurs à certaines variables de la procédure de tracé d'un graphique. Les options d'une commande graphique sont d'abord définies par défaut. Mais elles sont aussi personnalisables, pour cela il suffit d'introduire dans le corps de la commande graphique l'option avec la valeur souhaitée, par exemple : `PlotPoints->50` qui remplacera l'option par défaut `PlotPoints->25`.

□ Primitives

Les *primitives* comme `Line` ou `Point` sont des fonctions, mais des fonctions plus élémentaires que les commandes graphiques comme `Plot`. Typiquement, ce sont des composantes de la forme interne d'un fichier graphique (voir ci-dessus). Elles peuvent être combinées avec les fonctions graphiques soit par *juxtaposition*, comme dans `Show[Plot[…], Graphics[…]]` suivant le modèle `Show[FonctionGraphique[…], PrimitiveGraphique[…]]`, soit par *imbrication*, comme dans `Plot[expression, Plot\ Label->FontForm[…]]`, `FontForm` étant une primitive de texte, qui suit le modèle `Fonc\ tionGraphique[expression, … PrimitiveGraphique[…]]`.

Les primitives graphiques `Point`, `Line`, `Polygon`, `Cuboid`, `Circle`, `Disk`, tracent des formes élémentaires. Pour les chaînes de caractères, les primitives `Text` et `FontForm` permettent de placer du texte sur un graphique, d'en fixer la position, l'orientation, la police et la taille, la couleur du fond de texte et le type de formatage du texte.

□ Directives

Les directives ou directives de style *(style directives)* modifient le style sous lequel les primitives sont rendues. Elles affectent la taille des points (`PointSize`, `AbsolutePointSize`), l'épaisseur des traits (`Thickness`, `AbsoluteThickness`), la forme du pointillé (`Dashing`, `AbsoluteDashing`), la couleur (`GrayLevel`, `Hue`, `RGB`, `CMYKColor`).

■ Options et valeurs des options d'une commande

Les options des commandes graphiques sont nombreuses et elles diffèrent selon la commande considérée. Pour les connaître, on peut se reporter au manuel, à l'aide à l'écran ou faire la commande `Options` qui montre toutes les options d'une commande ainsi que leurs valeurs par défaut. Maîtriser les commandes graphiques signifie en réalité savoir utiliser leurs options et les combiner avec les primitives.

La commande **Options** renvoie toutes les options d'une commande quelconque. Les options de **Plot** forment une longue liste présentée ci-contre en entier. Les options sont classées par ordre alphabétique en commençant par les règles de substitution simples (->) et en terminant par les règles de substitution différées (:>).

```
Options[Plot]
{AspectRatio ->1/GoldenRatio, Axes -> Automatic,
AxesLabel -> None, AxesOrigin -> Automatic,
AxesStyle -> Automatic, Background -> Automatic,
ColorOutput -> Automatic, Compiled -> True,
DefaultColor -> Automatic, Epilog -> {},
Frame -> False, FrameLabel -> None,
FrameStyle -> Automatic, FrameTicks -> Automatic,
GridLines -> None, MaxBend -> 10.,
PlotDivision -> 20., PlotLabel -> None,
PlotPoints -> 25, PlotRange -> Automatic,
PlotRegion -> Automatic, PlotStyle -> Automatic,
Prolog -> {}, RotateLabel -> True,
Ticks -> Automatic, DefaultFont :> $DefaultFont,
DisplayFunction :> $DisplayFunction}
```

Pour voir les valeurs par défaut appliquées effectivement sur un graphique, il faut utiliser la commande `FullOptions`. Elle indique, par exemple, les valeurs effectives utilisées par `PlotRange` sur un graphique particulier quand `PlotRange` est paramétrée à `PlotRange-> Automatic`.

Les options et les valeurs effectivement appliquées à un graphique sont renvoyées par **FullOptions**. **Plot** avec **PlotRange-> Automatic** se traduira par un domaine de représentation effectif dont les dimensions sont données ici. On note que ce domaine ne commence pas exactement à zéro et déborde un peu sur le domaine des valeurs négatives des axes.

```
FullOptions[gr2D]
{AspectRatio -> 1., Axes -> False,
AxesLabel -> None, AxesOrigin -> Automatic,
[...........................]
PlotRange ->{{-0.02, 1.02}, {-0.02, 1.02},
{-0.25, 10.25}}, PlotRegion -> Automatic,
Prolog -> {}, RotateLabel -> True,
Ticks -> Automatic, DefaultFont -> {Courier, 6.5},
DisplayFunction -> (Display[$Display, #1] & )}
```

■ Options courantes en 2D

Dans la suite, nous allons nous intéresser aux options les plus souvent rencontrées. Pour cela, nous recherchons les options communes à tous les graphiques en 2D.

La liste des commandes créant des graphiques en 2D est définie.
La commande **Map** applique successivement **Options** à chaque élément de la liste, on obtient alors sept sous-listes.

```
liste2D = {ContourPlot, DensityPlot, Plot,
ListContourPlot, ParametricPlot, ListDensityPlot,
ListPlot};
listopt = Map[Options, liste2D];
```

Le résultat obtenu est une liste de sept listes formées d'éléments de la forme `a->b` ou bien `a:>b`, ou encore, en écriture interne, `Rule[a, b]` ou respectivement `RuleDelayed[a, b]`. Par la commande suivante, nous extrayons les noms des options en abandonnant leurs valeurs.

Les options et leurs valeurs sont réduites aux noms des options.
Chaque sous-liste de noms étant indicée `[[1,n]]`, l'intersection des sept sous-listes peut s'écrire comme ci-contre.

```
nomsopt = listopt/.{Rule[x_,y_]->x,
RuleDelayed[x_,y_]->x};
Intersection[nomsopt[[1,1]], nomsopt[[1,2]],
nomsopt[[1,3]], nomsopt[[1,4]], nomsopt[[1,5]],
nomsopt[[1,6]], nomsopt[[1,7]]];
```

Ou plus élégamment avec **Apply**.
Voici la liste des options communes aux commandes de graphiques en 2D.

```
Intersection[Apply[Sequence, nomsopt[[1]]]]
{AspectRatio, Axes, AxesLabel,
AxesStyle, Background, ColorOutput,
DefaultColor, DefaultFont, DisplayFunction,
Epilog, Frame, FrameLabel, FrameStyle,
FrameTicks, PlotLabel, PlotRange, PlotRegion,
Prolog, RotateLabel, Ticks}
```

Sous un aspect anodin, `Apply[Sequence, nomsopt]` est une forme d'écriture fondamentale qui peut nous épargner des heures de recherche infructueuse pour transformer une *liste* comme `{arg1, arg2, arg3,…}` en *séquence* d'arguments pour une commande ou une fonction comme dans `Intersection[arg1, arg2, arg3,…]` ou `Times[arg1, arg2, arg3,…]`. Rappelons qu'une liste est un ensemble d'éléments tenus en commun par des accolades qui s'écrit `List[element1, element2, element3,…]` en écriture interne tandis qu'une séquence est un ensemble d'éléments de forme `element1, element2, element3,…` en écriture ordinaire et qui s'écrit `Sequence[element1, element2, element3,…]` en écriture interne. Nous y reviendrons plus en détail dans la Leçon 4.

■ Effets des options courantes en 2D

En partant d'un graphique initial, les différentes options courantes sont introduites successivement de façon à mettre en évidence leur signification et leur incidence.

Changement de la police par défaut.
Une fonction simple est définie.
Sa représentation est demandée sur l'intervalle `{0.01, 5}` en invoquant la commande **Plot**. La borne basse de l'intervalle est choisie pour éviter les messages d'avertissement relatifs à la division par zéro.
Sur ce graphique, en l'absence de toute personnalisation, les options de la commande **Plot** ont pris leurs valeurs par défaut.

```
$DefaultFont = {"Courier", 6.5}
qdemande = 1/p;
graphdemande = Plot[qdemande,
{p,0.01,5}]
```

Le même graphique est demandé en modifiant l'option **AxesOrigin**. La valeur par défaut de l'origine des axes est `{0, 0}`. Ici, l'origine des axes est déplacée en `{-1, -1}`.

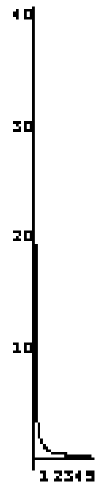
La nouvelle valeur de l'option vient supplanter la valeur antérieure.
Notez que les axes ne se rejoignent pas (voir plus loin la section Difficultés usuelles pour une solution).

```
Show[graphdemande,
AxesOrigin->{-1, -1}]
```

Le rapport de la hauteur à la largeur du graphique est défini par l'option **AspectRatio**. Par défaut, cette valeur est l'inverse du nombre d'or.
Quand l'option prend la valeur **Automatic**, alors les échelles sont identiques sur les deux axes. Pour toutes les autres valeurs, les échelles sont différentes.

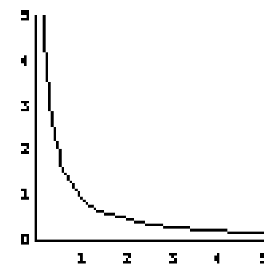
Le graphique obtenu est inadéquat, en raison de l'interaction entre l'option **AspectRatio** et l'intervalle {0.01,5} retenu pour les abscisses.

```
gd = Show[graphdemande,
AspectRatio->Automatic]
```



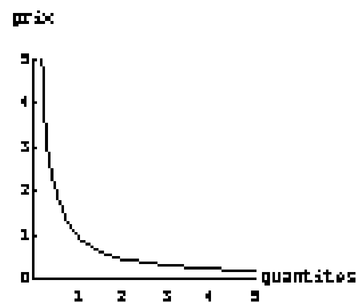
Une représentation plus équilibrée est obtenue en spécifiant par **PlotRange** les deux intervalles des valeurs en x et en y à retenir pour le tracé.
L'option **PlotRange** prend une valeur nouvelle pour le graphique **gd**.

```
gd0 = Show[gd,
PlotRange->{{0,5},{0,5}}]
```



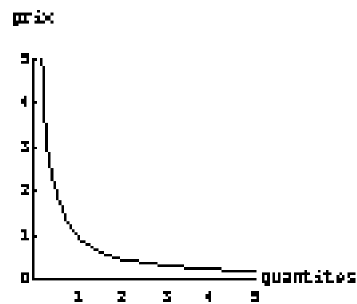
L'option **AxesLabel** permet d'attribuer des étiquettes aux axes.
Notez que les accents éventuels seront représentés à l'écran mais ne s'imprimeront pas sur une imprimante Postscript.

```
gd1 = Show[gd0,
AxesLabel->{quantites,prix}]
```



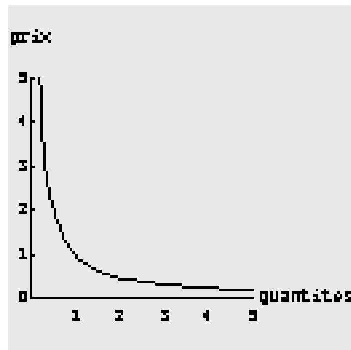
L'option **AxesStyle** permet de modifier le style de représentation des axes.
Ici, l'épaisseur des axes a été augmentée au moyen de la directive de style **Thickness** qui reçoit une valeur relative à la taille du graphique : le trait doit faire 1,5 % de la taille du graphique.

```
gd2 = Show[gd1,
AxesStyle->{Thickness[0.015], Thickness[0.015]}]
```



L'option **Background** détermine la couleur du fond de la boîte contenant le graphique.
Ici, la couleur est déterminée par la primitive **GrayLevel** qui définit des niveaux de gris de 0 (noir) à 1 (blanc).
D'autres primitives de couleurs (**Hue**, **RGB**, **CMYKColor**) sont utilisables.

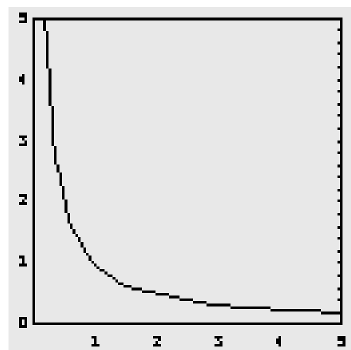
```
gd3 = Show[gd2,
Background->GrayLevel[0.9]]
```



L'option **Frame** avec la valeur **True** crée un cadre autour du graphique.

Les options relatives aux axes (**AxesLabel**, **AxesStyle**) se trouvent alors ignorées, le cadre remplaçant les axes.

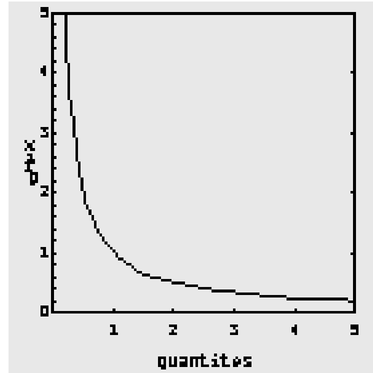
```
gd4 = Show[gd3,
Frame->True]
```



Pour rétablir des étiquettes, il faut maintenant utiliser l'option `FrameLabel`.

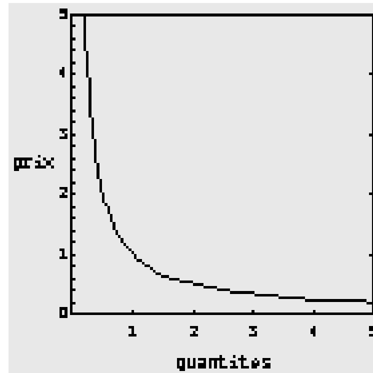
L'étiquette des ordonnées est inscrite verticalement, dans le sens ascendant, ce qui la rend peu lisible.

```
gd5 = Show[gd4, FrameLabel->{"quantites","prix"}]
```



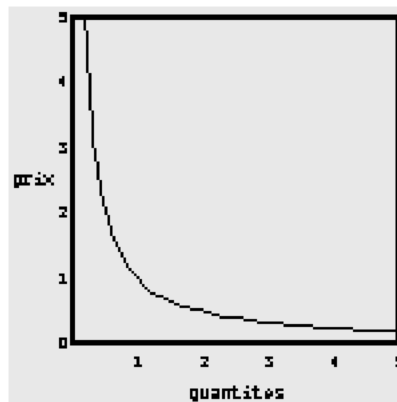
L'étiquette des ordonnées est rétablie à l'horizontale par `RotateLabel`.

```
gd6 = Show[gd5, RotateLabel->False]
```



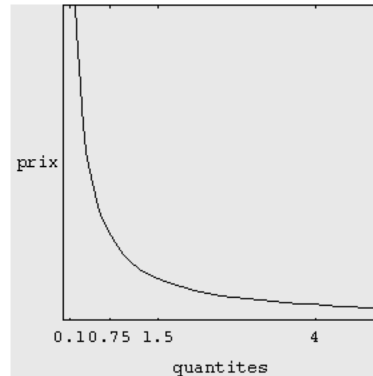
Le style de tracé du cadre est personnalisable par l'option `FrameStyle` associée à une ou plusieurs primitives graphiques. On utilise ici la primitive `Thickness` pour modifier l'option.

```
gd7 = Show[gd6, FrameStyle->{Thickness[0.015],
Thickness[0.015]}]
```



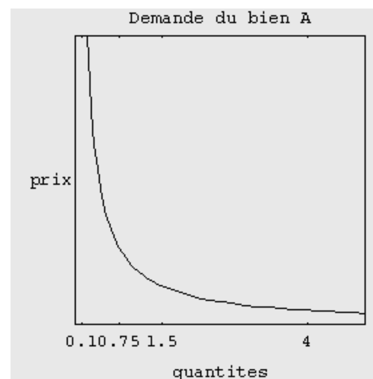
Des marques de graduation *(ticks)* personnalisées sont placées individuellement sur l'axe des abscisses par les valeurs de l'option **`FrameTicks`**.
Le style par défaut du cadre a été restauré par la valeur **`Automatic`** assignée à **`FrameStyle`** afin de rendre les marques mieux visibles.

```
gd8 = Show[gd7, FrameStyle->Automatic,
FrameTicks->{{0.1, 0.75, 1.5, 4}, None}]
```



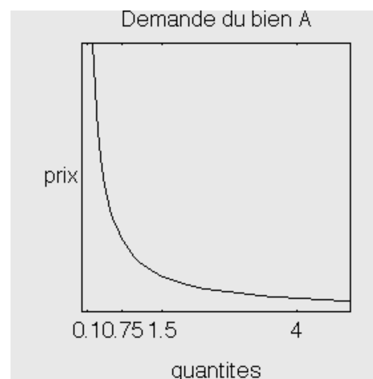
Un titre est attribué au graphique au moyen de l'option **`PlotLabel`**.
Le titre est écrit entre guillemets faute de quoi les termes composants seraient rendus par ordre alphabétique.

```
gd9 = Show[gd8, PlotLabel->"Demande du bien A"]
```



La police du graphique est modifiée.
Cette modification est locale, elle ne s'applique qu'au graphique considéré, contrairement à une commande comme
`$DefaultFont = {"nom",taille}`
qui modifie un paramètre général du système.

```
gd10 = Show[gd9, DefaultFont->{"Helvetica",8}]
```



Les options finales de gd10, vérifiables par Options[gd10], sont le cumul des options successives des graphiques consécutifs. En fait, les options des graphiques successifs sont accumulées dans l'ordre inverse de leur création, les plus récentes venant en premier. Certaines options, comme AspectRatio, Background ou DefaultFont, apparaissant plusieurs fois avec des valeurs différentes dans la définition de gd10, on constate que c'est la valeur d'une option figurant en premier dans la liste des options qui est mise en œuvre dans le résultat final. En effet, les options sont lues et interprétées en commençant par la fin de la liste, si bien que les valeurs des options d'un rang inférieur viennent écraser les valeurs des options d'un rang supérieur dans l'ordre de la liste. Il est aisé de voir les options qui ont été modifiées par rapport aux options par défaut en utilisant la commande Complement[Options[gd10], Options\ [Plot]].

Primitives graphiques

Les primitives graphiques (Point, Line, Rectangle, Cuboid, Polygon, Circle, Disk, Raster, RasterArray, Text) permettent de créer des objets élémentaires (et, pour Text, de mettre du texte en graphique). Toutefois, ces fonctions ne tracent pas directement les figures correspondantes. Elles ne le font que sous le contrôle d'autres commandes, comme Graphics ou Plot, dont la vocation spécifique est d'engendrer des représentations graphiques.

Par exemple, un objet graphique est créé sans être affiché.
Les données constitutives de l'objet graphique sont montrées par **InputForm**. Parmi ces données se trouvent plusieurs appels à la primitive **Point**. Notez que ces appels sont entourés par la commande **Graphics**.

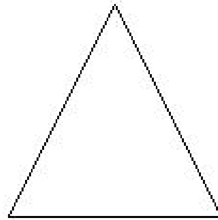
L'aide en ligne nous indique que **Point** est une primitive graphique qui représente le point dont les coordonnées sont fournies.
De même, **Line** représente une ligne joignant les points dont les coordonnées sont données.
Essayons de tracer une ligne en utilisant **Line**. Manifestement, la primitive n'affiche rien et même ne crée rien par elle-même malgré sa définition. Une solution pourrait être de faire **Show[Line[]]**.
Show n'accepte d'afficher qu'un objet graphique mais **Line** ne crée pas d'objet graphique. On doit donc utiliser **Graphics** pour créer un objet graphique à partir de **Line**.

```
In[1]:= points =
{{1,2}, {5,3}, {4,5}, {3,3}, {3.5,4}};
In[2]:= ListPlot[points, DisplayFunction->Identity]
Out[2]= -Graphics-
In[3]:= InputForm[%]
Out[3]//InputForm= Graphics[{Point[{1, 2}],
Point[{5, 3}], Point[{4, 5}], Point[{3, 3}],
Point[{3.5, 4}]}, {PlotRange -> Automatic,
AspectRatio -> GoldenRatio^(-1),
[.................],
 FrameLabel -> None, PlotRegion -> Automatic}]
In[4]:= ?Point
Point[coords] is a graphics primitive that
represents a point.
In[5]:= ?Line
Line[{pt1, pt2, ...}] is a graphics primitive which
represents a line joining a sequence of points.
In[6]:= Line[{{0,0}, {1,0}, {1/2,1}, {0,0}}]
Out[6]= Line[{{0, 0}, {1, 0}, {1/2, 1}, {0, 0}}]
In[7]:= Show[Line[{{0,0}, {1,0}, {1/2,1}, {0,0}}]]
Show::gtype: Line is not a type of graphics.
Out[7]= Show[Line[{{0, 0}, {1, 0}, {1/2, 1},
{0, 0}}]]

In[8]:= Graphics[Line[{{0,0}, {1,0}, {1/2,1},
{0,0}}]]
Out[8]= -Graphics-
```

Puis, finalement, on demande explicitement par **Show** l'affichage de cet objet créé par **Graphics**.

```
In[9]:= Show[%, AspectRatio->Automatic]
```



```
Out[9]= -Graphics-
```

Modifier les options par défaut standard

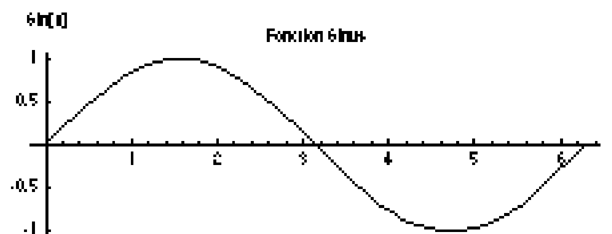
Les valeurs des options des commandes graphiques sont fixées par défaut. Mais elles ne sont pas nécessairement adéquates au travail particulier en cours. Comme il est fastidieux de changer toujours les mêmes options au niveau de chaque graphique, on peut souhaiter au cours d'une session *Mathematica* que les graphiques aient tous certaines options de base identiques. Pour cela, il faut entrer de nouvelles valeurs par défaut au moyen de la commande `SetOptions`. C'est surtout utile pour les options `AspectRatio`, `AxesOrigin`, `DefaultFont` et `Rotate\ Label`. Les modifications introduites par `SetOptions` vaudront pour toute la session et tant que les valeurs des options ne sont pas modifiées localement (par insertion de nouvelles valeurs d'option dans une commande graphique) ou globalement (par une nouvelle utilisation de `SetOptions`).

Certaines options par défaut de **Plot** sont modifiées.

```
SetOptions[Plot,
AspectRatio->Automatic,
AxesOrigin->{0,0}, RotateLabel->False,
DefaultFont->{"Helvetica",6}];
```

Un graphique est créé par **Plot**.
Son tracé prend en compte les valeurs d'option redéfinies au sein de **SetOptions**.

```
Plot[Sin[x], {x,0,2 Pi}, AxesLabel->{x,Sin[x]},
PlotLabel->"Fonction Sinus"]
```



Les valeurs exactes appliquées par **PlotRange** sont indiquées par **FullOptions**.
Les options par défaut d'origine sont restaurées.

```
FullOptions[%, PlotRange]
{{-0.15708, 6.44026}, {-1.05, 1.05}}
SetOptions[Plot,
AspectRatio->1/GoldenRatio,
AxesOrigin->Automatic,
DefaultFont->$DefaultFont, RotateLabel->True];
```

Pour une modification habituelle des options, il faut envisager de placer une ou plusieurs commandes `SetOptions` dans le fichier d'initialisation `init.m`. Alors, toutes les sessions de travail commenceront automatiquement avec cet ensemble d'options personnalisées. Cette technique est applicable à toute commande ayant des options.

Passer des options graphiques à une commande

Un utilisateur peut souhaiter regrouper ses options personnelles de fonctions graphiques dans un ou plusieurs ensembles, chacun nommé par un symbole, et ce afin de pouvoir passer un ensemble de ces options en bloc à une fonction graphique. Cela permet d'utiliser un code plus compact, surtout quand on crée de nombreux graphiques, et cela facilite un emploi uniforme des options et de leurs modifications. Cependant, le passage des options aux commandes graphiques par une variable s'avère délicat. Ce petit problème ne concerne pas seulement les options graphiques, il touche de façon très générale à la forme de la présentation des arguments des fonctions et à leur évaluation, c'est pourquoi nous y insistons.

La valeur d'une option est redéfinie, la règle relative à l'option est nommée par le symbole **optperso**.

```
optperso = GridLines->Automatic;
```

On tente, sans succès, de passer la nouvelle valeur de l'option au moyen du symbole contenant la règle.

```
Plot[Sin[x], {x,0,2 Pi}, optperso]
Plot[Sin[x], {x,0,2 Pi}, optperso]
```

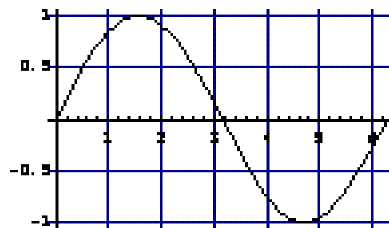
L'échec vient de ce que **Plot** n'évalue pas ses arguments, ce que montre **Attributes** en renvoyant HoldAll pour indiquer que la fonction exécute ses arguments sans les évaluer.

```
Attributes[Plot]
{HoldAll, Protected}
```

Donc, puisque Plot n'évalue pas ses arguments, il faut forcer l'évaluation des arguments de Plot si on veut lui passer des options par symbole. Quatre solutions sont considérées (mais il y en a d'autres !) : utiliser Evaluate, ou supprimer l'attribut HoldAll qui empêche Plot d'évaluer ses arguments, ou définir les options concernées comme des variables locales au sein de la commande Block[], ou modifier les options de Plot par SetOptions qui, elle, évalue ses arguments à condition que ceux-ci lui soient présentés *en séquence*.

Solution 1 : utiliser **Evaluate**.
La valeur d'*une option unique* est redéfinie, la règle relative à l'option est nommée par le symbole **optperso**. La nouvelle valeur de l'option est passée à **Plot** au moyen du symbole contenant la règle.
L'évaluation du symbole **optperso** est forcée par la commande **Evaluate**.
Le résultat attendu est obtenu.

```
optperso = GridLines->Automatic;
Plot[Sin[x], {x,0,2 Pi}, Evaluate[optperso]]
```



Le symbole **optperso** est vidé.
Les valeurs de *deux options* sont redéfinies, les règles relatives aux options, *mises en apparence sous forme de séquence*, sont nommées **optperso**.

```
optperso = .;
optperso = GridLines->Automatic,
Background->GrayLevel[0.9];
```

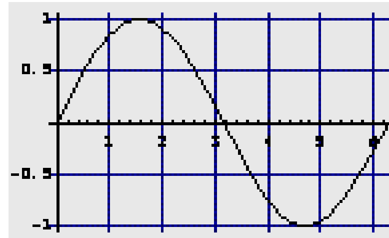
L'idée de mettre les options en séquence vient de ce qu'elles apparaissent en séquence dans **Plot**.
Mais une erreur de syntaxe apparaît, le symbole **optperso** ne peut recevoir directement comme contenu une séquence.

```
Plot[Sin[x], {x,0,2 Pi}, Evaluate[optperso]]
Syntax::sntxf: "optperso=GridLines->Automatic"
cannot be followed by
",Background->GrayLevel[0.9]".
```

Les règles relatives aux options, *mises sous forme de liste*, sont nommées **`optperso`**.
Les nouvelles valeurs des options sont passées à **`Plot`** au moyen du symbole.
L'évaluation du symbole **`optperso`** est forcée par la commande **`Evaluate`**.
Le résultat attendu est obtenu.

```
optperso = .;
optperso = {GridLines->Automatic,
Background->GrayLevel[0.9]};

Plot[Sin[x], {x,0,2 Pi}, Evaluate[optperso]]
```

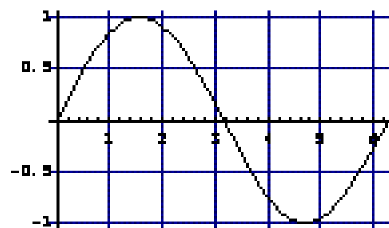


Lors du premier essai, pour entrer les options par une vraie séquence, il aurait fallu écrire :
`optperso = Sequence[ GridLines->Automatic, Background->GrayLevel[0.9]]`.

Solution 2 : supprimer de la fonction **`Plot`** l'attribut `HoldAll` à cause duquel elle n'évalue pas ses arguments. Pour être modifiée, la fonction qui est protégée doit d'abord être déprotégée.
L'attribut `HoldAll` est supprimé de **`Plot`**. La fonction n'a plus d'attributs.
L'option nouvelle est créée et nommée. L'option est passée par le symbole **`optperso`** qui, maintenant, est évalué par **`Plot`**, c'est-à-dire que **`GridLines->Automatic`** vient se substituer à **`optperso`**. Et la fonction s'exécute bien avec la nouvelle option.
Plusieurs options peuvent être passées en liste ou en séquence comme ci-dessus.
Attention, la déprotection d'une fonction interne étant susceptible de produire des résultats inattendus, les attributs d'origine sont restaurés.

```
Attributes[Plot]
{HoldAll, Protected}

Unprotect[Plot]
{Plot}
ClearAttributes[Plot, HoldAll]
Attributes[Plot]
{}
optperso = GridLines->Automatic;
Plot[Sin[x], {x,0,2 Pi}, optperso]
```



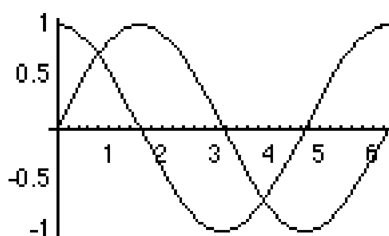

```
SetAttributes[Plot, {HoldAll, Protected}]
```

Solution 3 : écrire les options comme variables locales d'un bloc d'instructions.
Cela ne présente d'intérêt que pour modifier localement les valeurs des seules variables système **`$DisplayFunction`** et **`$DefaultFont`**.
La commande **`Block`** définit une procédure avec des variables locales qui figurent dans la liste de tête. Ici ce sont les variables systèmes **`$DisplayFunction`** et **`$DefaultFont`** qui reçoivent de nouvelles valeurs locales ; notamment, l'affichage à l'écran est supprimé.
Les graphiques sont créés avec ces valeurs d'option.
Ensuite, l'affichage standard, c'est-à-dire l'impression à l'écran, est restauré.
L'affichage des graphiques superposés est effectué par **`Show`**.

```
Block[
{$DisplayFunction = Identity,
$DefaultFont = {"Helvetica",9}},

g1 = Plot[Sin[x], {x,0,2 Pi}];
g2 = Plot[Cos[x], {x,0,2 Pi}];
$DisplayFunction = Display[$Display,#1]&;

Show[{g1,g2}]
]
```

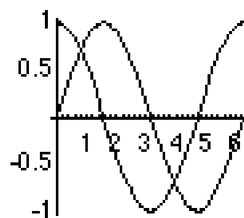


Solution 4 : passer les options par **SetOptions** qui modifie les options des commandes. **SetOptions** évalue ses arguments. L'inconvénient est que ces modifications sont globales, il faut donc trouver un moyen de leur donner une portée seulement locale. Les valeurs d'origine des options sont stockées dans **optsyst**. Puis, la fonction **optio** appelant **SetOptions** et lui passant une liste d'arguments *sous forme de séquence*, est créée. Les nouvelles valeurs d'option sont entrées dans **optperso1**. Ces valeurs sont passées comme arguments à **optio** qui appelle **SetOptions**. Le graphique est généré sur la base de ces valeurs personnalisées.
Le résultat recherché est obtenu.

```
optsyst = Options[Plot]
{AspectRatio -> 1/GoldenRatio, [...............],
DefaultFont :> $DefaultFont,
DisplayFunction :> $DisplayFunction}

optio[perso_]:=
SetOptions[Plot, Apply[Sequence, perso]]

optperso1 = {AspectRatio->1,
DefaultFont->{"Helvetica",9}};
optio[optperso1]
{AspectRatio -> 1, [...............],
DefaultFont -> {Helvetica, 9},
DisplayFunction :> $DisplayFunction}
Plot[Sin[x], {x,0,2 Pi}]
```



Les valeurs d'origine des options sont restaurées en rappelant **optsyst** dans **SetOptions**. La modification des options de **Plot** ne concernera donc que le graphique.

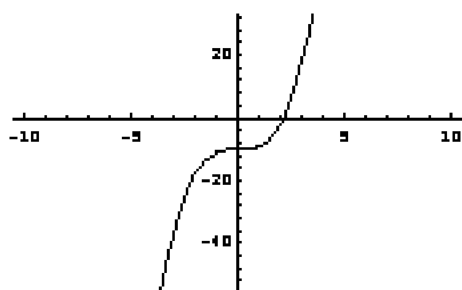
```
optio[optsyst]
{AspectRatio -> 1/GoldenRatio, [...............],
DefaultFont :> $DefaultFont,
DisplayFunction :> $DisplayFunction}
```

Difficultés usuelles en 2D

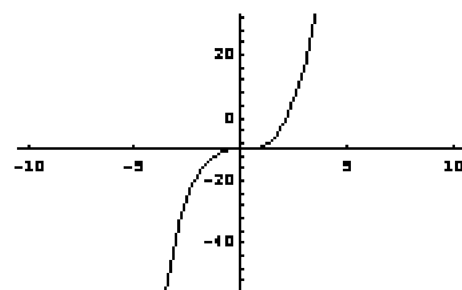
Les axes ne se coupent pas au point désiré

L'algorithme choisit le point d'intersection des axes censé être le meilleur mais ce n'est pas nécessairement le point désiré. Il arrive notamment que les axes ne se coupent pas en {0,0} comme on le souhaiterait. Pour déplacer le point d'intersection des axes, entrer l'option `AxesOrigin->{xdesire, ydesire}`.

```
Plot[(x^3) - 10, {x,-10,10}]
```



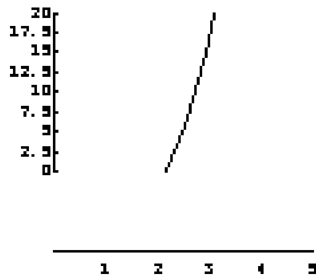

```
Plot[(x^3) - 10, {x,-10,10},
AxesOrigin->{0,-10}]
```



La combinaison de `AxesOrigin` et de `PlotRange` produit parfois des résultats inattendus. Il arrive qu'une partie des axes ne soit pas figurée. On constate alors que le tracé des axes et de la fonction est complètement défini par `PlotRange` même si les axes sont censés se couper en dehors de la zone de

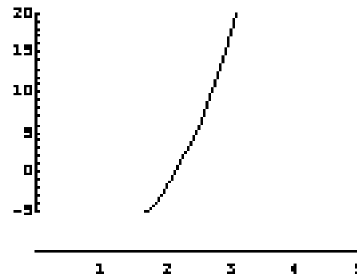
tracé. Les valeurs de l'option `AxesOrigin` sont prises en compte pour positionner l'origine des axes mais, si elles sortent du domaine défini par `PlotRange`, une partie des axes n'est pas tracée.

```
Plot[(x^3) - 10, {x,-10,10},
AxesOrigin->{0,-10},
PlotRange->{{0,5},{0,20}}]
```

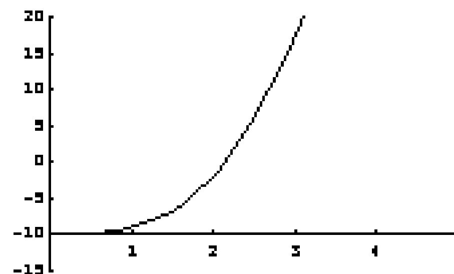


En modifiant les valeurs numériques de l'option **`PlotRange`**, on obtient le tracé souhaité pour les axes.

```
Plot[(x^3) - 10, {x,-10,10},
AxesOrigin->{0,-10},
PlotRange->{{0,5},{-5,20}}]
```




```
Plot[(x^3) - 10, {x,-10,10},
AxesOrigin->{0,-10},
PlotRange->{{0,5},{-15,20}}]
```



Il apparaît en définitive que l'axe des ordonnées est tracé uniquement selon les bornes définies dans `PlotRange`.

■ Les axes ne représentent pas les valeurs désirées

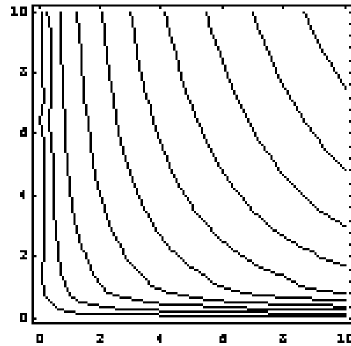
Utiliser l'option `PlotRange` pour modifier la zone de tracé. Utiliser l'option `Ticks` s'il s'agit des valeurs numériques portées sur les axes. Des commandes supplémentaires pour une gestion approfondie des marques de graduation sont proposées par T. Wickham-Jones dans *Mathematica Graphics*.

■ Les courbes présentent des anomalies

Les graphiques peuvent présenter des incohérences apparentes qui tiennent à une valeur insuffisante de l'option `PlotPoints`, valeur qui doit alors être augmentée. L'algorithme de calcul utilisé par `Plot` ou `ContourPlot` est un algorithme adaptatif qui échantillonne la fonction à représenter en une série de points régulièrement espacés, puis qui modifie cet espacement selon les variations décelées pour les valeurs échantillonnées de la fonction.

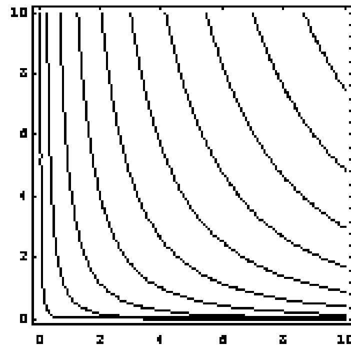
Cette représentation est défectueuse pour les parties des courbes de niveaux situées près des axes.
La cause en est le trop petit nombre de valeurs échantillonnées qui met en défaut l'interpolation.

```
cobbdouglas = 10 k^(1/2) l^(1/4);
ContourPlot[cobbdouglas, {k,0,10}, {l,0,10},
ContourShading->False]
```



Il suffit pour y remédier d'augmenter la valeur par défaut de l'option **PlotPoints** qui est **PlotPoints->15**, mais le temps de calcul augmente.

```
ContourPlot[cobbdouglas, {k,0,10}, {l,0,10},
ContourShading->False, PlotPoints->100]
```



L'option `ContourSmoothing` peut aussi être utilisée pour améliorer le produit de la fonction `ContourPlot`. L'option MaxBend qui définit l'angle maximal entre deux segments servant à tracer une courbe par approximation peut également être en cause pour les commandes `Plot` et `ParametricPlot`.

Plus gravement, il arrive que l'algorithme de tracé renvoie des graphiques erronés parce qu'il ne détecte pas la périodicité véritable d'une fonction.

Ci-contre, un exemple connu de cette erreur (le lecteur aura la surprise du résultat).
Mais celui-ci, plus original, me paraît encore meilleur !

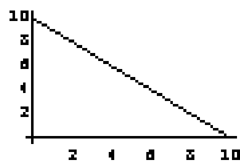
```
Plot[Sin[x], {x, 0, 96 Pi},
Axes->{False,True},DisplayFunction->Identity]
Plot[Sin[x], {x, 0.000001, 96 Pi}]
```

L'algorithme de calcul étant un algorithme adaptatif, sur le premier exemple ci-dessus, comme les mêmes valeurs sont trouvées successivement en plusieurs points d'estimation, l'algorithme en infère que la fonction est constante, les erreurs d'arrondis viennent finalement perturber l'exact espacement des points d'estimation, et alors, en présence de valeurs non constantes, l'algorithme augmente le nombre de points estimés et détecte l'oscillation. La solution élémentaire est d'augmenter le nombre de points où la fonction est estimée (essayez `PlotPoints->50` puis `PlotPoints->100`). Ces erreurs ne se produiront plus avec la version 3.

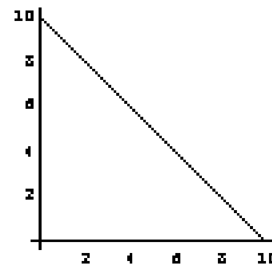
■ Les échelles ne sont pas identiques sur les deux axes

Pour avoir des échelles identiques sur les axes, il suffit d'entrer l'option `AspectRatio` avec la valeur `Automatic`.

```
ListPlot[{{0,0},{0,10},{10,0},{0,0}},
PlotJoined->True, Axes->True]
```



```
ListPlot[{{0,0},{0,10},{10,0},{0,0}},
PlotJoined->True, Axes->True,
AspectRatio->Automatic]
```



■ Les proportions du graphique sont inadéquates

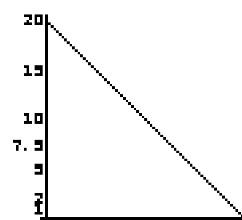
Utiliser `AspectRatio` pour les graphiques 2D et `BoxRatios` pour les diagrammes 3D. `AspectRatio` définit en fait le ratio hauteur/base de la boîte contenant le graphique. Le graphique occupe toute la boîte avec l'option par défaut `PlotRegion->Automatic` et la zone occupée par le graphique est modifiable par cette option. Pour `AspectRatio->1`, la boîte est un carré.

■ Les marques de graduation ne conviennent pas

La solution la plus simple est d'utiliser l'option `Ticks` ou `FrameTicks` pour poser directement les valeurs des marques à représenter.

Une liste de doublets est représentée.
Les marques de graduation sont posées explicitement sur l'axe des ordonnées par l'option **`Ticks`**.
Remarquez que, pour l'axe des abscisses, la valeur de l'option est **`None`**.

```
ListPlot[{{0,0},{0,20},{5,0},{0,0}}, PlotJoined->True,
Axes->True, AspectRatio->1,
Ticks->{None,{1,2,5,7.5,10,15,20}}]
```

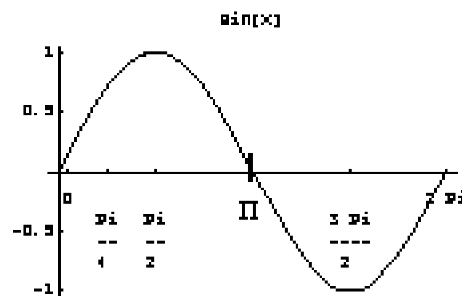


Le format d'entrée des valeurs des *ticks* peut suivre une des six formes ci-après, `longueur` se référant à la longueur du trait de la marque de graduation, `longueurhaut` à la longueur du trait vers le haut, `longueurbas` à la longueur du trait vers le bas et `style` au style de ce trait : `position` ou,
`{position, label}` ou,
`{position, label, longueur}` ou,
`{position, label, longueur, {style}}` ou,
`{position, label, {longueurhaut, longueurbas}}` ou,
`{position, label, {longueurhaut, longueurbas}, {style}}`.

La fonction **`Sin[x]`** est représentée.
Un titre est porté sur le graphique.
Ici, les valeurs sont d'abord entrées suivant le modèle `{position, label}`. Puis, en utilisant le modèle `{position, label, {longueurhaut, longueurbas}, {style}}`, le symbole est positionné avec un trait de `0.04` au-dessus de l'axe et de `0.02` au-dessous et avec le style d'épaisseur `0.01`.
D'autres valeurs sont positionnées en abscisses.
Les marques et valeurs en ordonnées sont spécifiées par **`Automatic`**.

Notez que l'on a posé les marques en combinant plusieurs modèles différents de format.

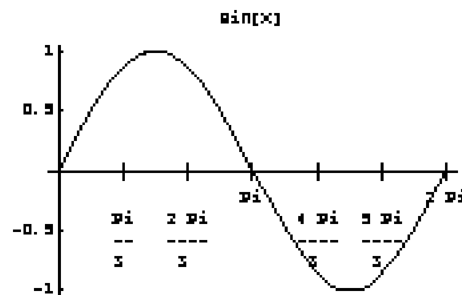
```
Plot[Sin[x], {x,0,2 Pi},
PlotLabel->"Sin[x]",
Ticks->{{{0.15,0},{Pi/4,Pi/4},{Pi/2,Pi/2},
{Pi,FontForm[P,{"Symbol",10}],{0.04,0.02},
{Thickness[0.01]}},
{3 Pi/2,3 Pi/2},{2 Pi,2 Pi}},
Automatic}]
```



L'entrée manuelle de ces valeurs des *ticks* est fastidieuse, surtout si ces valeurs doivent être calculées à chaque graphique. Ce travail peut être allégé par `Table` ou par une fonction *ad hoc*.

La commande **`Table`** produit une liste de termes comme **`{Pi/3,Pi/3,{0.02,0.02}}`** qui suivent le modèle `{label, position, {longueurhaut, longueurbas}}`.
La commande **`Table`** est évaluée en premier, puis son résultat est passé à **`Ticks`**.

```
Plot[Sin[x], {x,0,2 Pi},
PlotLabel->"Sin[x]",
Ticks->{Table[{x, x, {0.02, 0.02}},
{x, 0, 2 Pi, Pi/3}],
Automatic}]
```



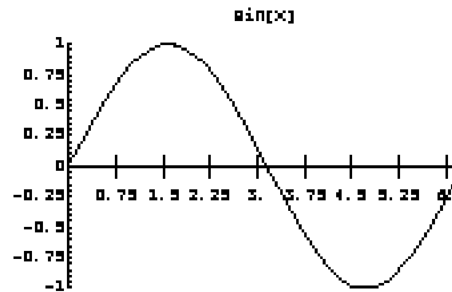
On remarque que le tracé excède légèrement 0 et 2 Pi sur l'axe des x. Ce qui rend nécessaire l'adjonction d'une option `PlotRange` dans le graphique suivant (regardez ce qui se passe si cette option est supprimée).

Une fonction **tickmark** générant une table de marques de graduation est créée.
Le graphique est tracé à l'aide de la fonction **tickmark** et sur l'intervalle exact **{0,2 Pi}** pour les abscisses.

Dans cette suite de commandes proposée par T. Wickham-Jones, la fonction **tickmark** incluse dans l'option **Ticks**, reçoit ses arguments d'une façon mystérieuse.
L'avantage est que **tickmark** récupère les valeurs des bornes automatiquement.

```
tickmark[min_,max_]:=
Table[{x,x,0.05}, {x,min,max,0.75}]

Plot[Sin[x], {x,0,2 Pi}, PlotLabel->"Sin[x]",
Ticks->{tickmark,Automatic},
PlotRange->{{0,2 Pi},{-1,1}}]
```



Pour des détails supplémentaires, voir *The Mathematica Journal,* vol. 5 n° 2, 1995, p. 42 et s. T. Wickham-Jones a conçu un fichier de commande pour la gestion des *ticks* (voir ci-après).

■ L'angle de vue du graphique en 3D est inadéquat

En choisissant Action puis Prepare Input puis 3DView Point Selector... (ou en faisant -V ou Shift-Ctrl-V), on accède à une fenêtre de dialogue qui permet de modifier le point de vue sur le graphique de façon interactive et de coller les coordonnées correspondant au choix retenu. Notez qu'il existe plusieurs options pour définir le nouveau point de vue. Signalons l'existence de *MathLive*, un programme complémentaire, qui permet la visualisation en temps réel des graphiques sans passer par 3DView Point Selector... .

■ Les polices utilisées par défaut ne conviennent pas

On peut changer la police de l'ensemble du graphique par l'option `DefaultFont`. Dans les graphiques, la valeur par défaut de la police est donnée par la valeur de la variable système `$DefaultFont`. On a donc le choix entre changer l'option du graphique, en écrivant dans la commande graphique, par exemple `DefaultFont->{"Geneva", 14}`, ou l'option du système, en écrivant avant la commande graphique, `$DefaultFont = {"Geneva", 14}` pour avoir la police Geneva en taille 14.

Si on veut agir sur la police d'un seul élément de texte, et non sur les caractères de l'ensemble du graphique, il faut alors utiliser la primitive `FontForm` avec la syntaxe `FontForm["texte", {"nomdepolice",taille}]`.

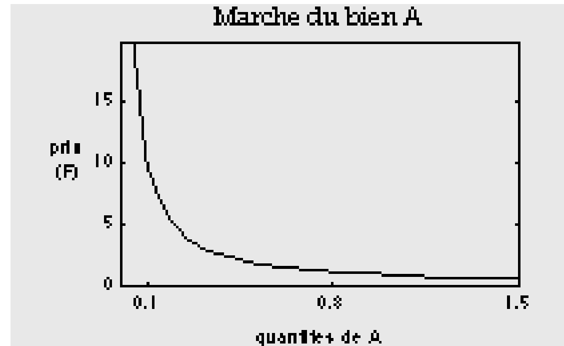
Un graphique est créé avec de nombreuses options personnalisées.

La police générale du graphique est **Helvetica** en **7** points.
Notez l'usage de **\n** pour obtenir un saut de ligne dans l'étiquette de l'axe vertical.

```
qdemande = 1/p;
graphdemande = Plot[qdemande, {p,0.001,5},
AxesLabel->{quantités, prix},
AxesStyle->{Thickness[0.01],Thickness[0.01]},
Background->GrayLevel[0.9],
DefaultFont->{"Helvetica",7}, Frame->True,
FrameLabel->{"quantites de A", "prix \n (F)"},
FrameStyle->{Thickness[0.005],Thickness[0.005]},
FrameTicks->{{0.1, 0.8, 1.5, 4}, {0, 5, 10, 15}},
```

La police du titre est spécifiée par l'intermédiaire de la primitive **FontForm** introduite dans l'option **PlotLabel**. De cette façon, le titre est dans une police différente de la police générale du graphique.

```
PlotLabel->FontForm["Marche du bien A",
{"Palatino-Bold",9}],
PlotRange->{{0, 1.5}, {0, 20}},
RotateLabel -> False]
```

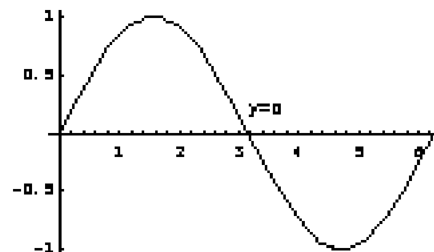


■ Comment introduire du texte ?

Si on veut entrer, sur un graphique `graphA`, du texte autre que le titre ou les étiquettes des axes pour lesquels il existe `AxesLabel` et `PlotLabel`, une solution logique mais lourde est de créer ce texte comme un objet graphique spécifique `graphtext` à l'aide des primitives `Graphics` et `Text`, puis de combiner `graphA` et `graphtext` à l'aide de `Show` !

L'objet graphique **graphA** est créé sans être affiché.
L'objet graphique **graphtext** est créé sans être affiché. La primitive **Text** place le coin inférieur gauche (spécifié par **{-1,-1}**) de la boîte contenant le texte **"y=o"**, au point **{3.2,0.1}**. L'objet créé par **Text** est transformé en objet graphique par **Graphics**. Le tracé de la fonction et le texte sont superposés par **Show**.

```
graphA = Plot[Sin[x], {x,0,2 Pi},
DisplayFunction->Identity]
graphtext = Graphics[Text["y=0",{3.2,0.1},{-1,-1}],
DisplayFunction->Identity]
Show[graphA, graphtext,
DisplayFunction->$DisplayFunction]
```

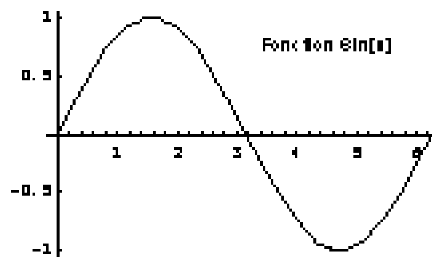


Une solution moins évidente mais plus directe est d'utiliser l'option `Epilog` (ou `Prolog`) dont la fonction est d'exécuter des primitives graphiques après (respectivement, avant) que le corps principal du graphique a été rendu.

Après l'exécution du tracé de la fonction, l'option **Epilog** est exécutée. Elle appelle la primitive **Text** qui inscrit le texte **Fonction Sin[x]** en reprenant les paramètres de **FontForm**, et en centrant le texte au point **{4.5,0.75}**.
Cette solution est plus compacte et plus lisible.

```
graphB = Plot[Sin[x], {x,0,2 Pi},
Epilog->Text[FontForm["Fonction Sin[x]",
{"Helvetica",7}],{4.5,0.75}]]
```

Le graphique apparaît avec un texte personnalisé qui fait fonction de titre.

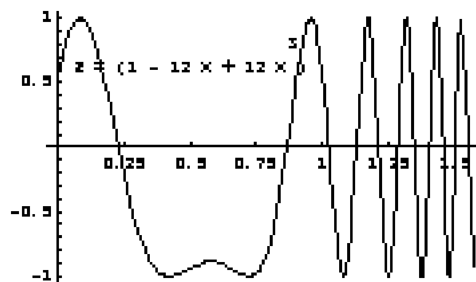


■ Comment reprendre sur le graphique des valeurs calculées ?

Il s'agit ici de passer des valeurs au texte du graphique, ces valeurs ayant été calculées avant la commande de tracé. Comme `Plot` n'évalue pas ses arguments, la question se pose de savoir si un tel transfert peut être réalisé et comment.

Une fonction est créée et le cosinus de cette fonction est représenté.
La définition de la fonction est reprise par **`Epilog`** et **`Text`**, la définition de la fonction étant passée simplement par le symbole **`fonction`**.

```
fonction = D[3 x^4, x] - D[2 x^3, {x,2}] + 1;
Plot[Cos[fonction], {x,0,Pi/2},
Epilog->Text["z ="fonction, {0.5,0.7}]]
```



Remarquez que la primitive **`Text`** évalue ses arguments, ce qui permet le passage de valeur par le symbole **`fonction`**, bien que **`Plot`** n'évalue pas ses arguments. Cela se vérifie soit en appliquant **`Attributes`** à **`Text`**, soit, ici, en utilisant **`Trace`** qui montre que **`fonction`** est évalué, puis le premier argument de **`Text`**, puis **`Text`**.

```
Trace[Text["z ="fonction, {0.5,0.7}]]
                                     3
{{{fonction, 1 - 12 x + 12 x },
                          3
z = (1 - 12 x + 12 x )},
                               3
Text[z = (1 - 12 x + 12 x ),{0.5, 0.7}]]}
```

Pour passer des valeurs au titre du graphique, on procède de même.

```
Plot[Cos[fonction], {x,0,Pi/2},
PlotLabel->"y ="fonction,
DisplayFunction->Identity]
```

L'exemple ci-dessous exige de former une chaîne de caractères par concaténation car il s'agit de combiner dans le titre noms de symboles (en lettres grecques) et valeurs numériques :

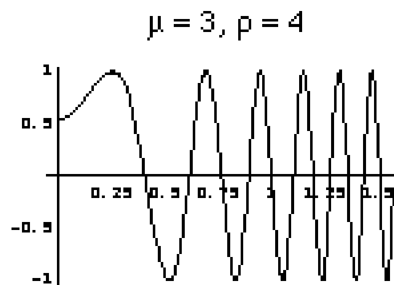
Deux paramètres et une fonction de ces deux paramètres sont définis.

```
m = 3; r = 4;
fonction = D[3 x^m, x] - D[2 x^r, {x,2}] + 1;
```

Une chaîne de caractères est créée. Les éléments en sont liés par l'opérateur de concaténation (**`<>`**) ; la *valeur numérique* des paramètres doit être convertie en *caractères de chaîne* par la commande **`ToString`**.

```
chaine = "m = " <> ToString[m] <> ", r = " <>
ToString[r];
Plot[Cos[fonction], {x,0,Pi/2},
PlotLabel->FontForm[chaine,{"Symbol",12}]]
```

La chaîne ainsi formée est transmise à **PlotLabel** et exécutée sous **FontForm** qui emploie la police **Symbol** transformant les lettres latines des symboles initiaux en lettres grecques.
Dans la version 3, l'usage direct des lettres grecques et autres symboles est possible.



■ Comment obtenir du texte accentué ?

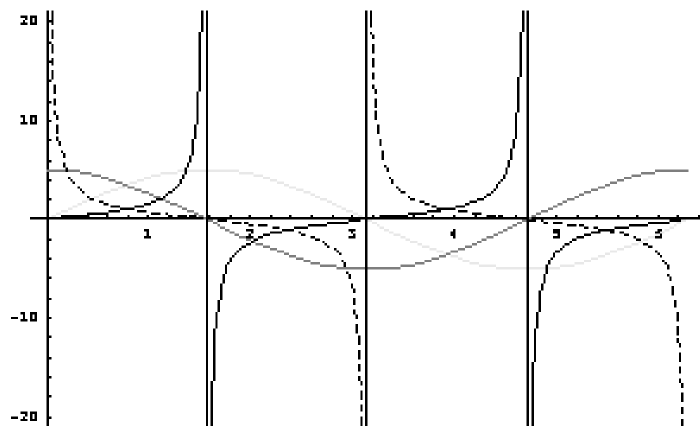
Avec les versions 2.2.2 ou antérieures de *Mathematica*, il n'existe pas de solution pour obtenir l'impression des caractères français accentués pour un graphique imprimé sur une imprimante Postscript. Les caractères accentués sont bien rendus à l'écran mais ils ne s'impriment pas en Postscript ; en revanche, ils s'impriment correctement sur une imprimante StyleWriter non Postscript. Interrogé sur ce problème, le support technique de *Mathematica* a confirmé que ces caractères ne sont pas envoyés correctement à l'imprimante et que ce problème doit être traité dans la prochaine mise à jour majeure, c'est-à-dire la version 3.

■ Les courbes ne se distinguent pas les unes des autres

Pour différencier les courbes, utiliser l'option `PlotStyle` en combinaison avec `Thickness`, `GrayLevel` et `Dashing`.

Les composantes de **PlotStyle** sont appliquées successivement aux différentes fonctions.
Thickness et **GrayLevel** permettent de varier épaisseur et nuance de gris.
La dernière courbe reçoit un pointillé par **Dashing**.

```
Plot[{5 Sin[x],5 Cos[x],Tan[x],Cot[x]},{x,0,2 Pi},
PlotStyle->{
{Thickness[0.005],GrayLevel[0.9]},
{Thickness[0.002],GrayLevel[0.5]},
{Thickness[0.003]},
{Thickness[0.003],Dashing[{0.01,0.01}]}}]
```



■ Comment faire des étiquettes sur plusieurs lignes ?

Les étiquettes *(labels)* peuvent faire plus d'une ligne. Il suffit de les couper à l'endroit voulu par `\n` (\ = anti-slash), qui commence une nouvelle ligne à l'impression.

■ Comment relever des coordonnées sur un graphique ?

Sélectionner le graphique, puis appuyer sur sous MacOS ou sur Ctrl sous Windows, le cur-seur devient un curseur en croix. La position du curseur dans les coordonnées du graphique est indiquée en bas à gauche de la fenêtre de travail. En cliquant sur chaque point dont on souhaite obtenir les coordonnées, celles-ci sont stockées dans un fichier temporaire. En faisant Copier puis Coller , ces données apparaissent dans une cellule sous forme de liste réutilisable pour un calcul.

■ Plot[{fct1, fct2,…},...] ne donne rien

La fonction `Plot` peut traiter plusieurs fonctions entrées en liste. Si toutefois `Plot[{fct1, fct2,…},…]` ne donne rien, essayer `Plot[Evaluate[{fct1, fct2,…}], …]`. Le problème, déjà rencontré, vient de ce que `Plot` n'évalue pas ses arguments.

■ Comment juxtaposer plusieurs graphiques ?

Utiliser la fonction `GraphicsArray` qui permet d'ordonner les graphiques en tableau à volonté. Chaque sous-liste de la commande `GraphicsArray` correspond à une ligne du tableau des graphiques. Ainsi, la commande `GraphicsArray[{{gr1}, {gr2,gr3}, {gr4,gr5, gr6}}]` crée un tableau avec `gr1` en première ligne, `gr2` et `gr3` en deuxième ligne, `gr4`, `gr5`, `gr6` en troisième ligne. Pour un exemple, voir la section introductive au début de cette Leçon.

■ Comment superposer des objets graphiques ?

Chaque objet est créé pour lui-même par une fonction comme `Plot` ou par la primitive `Graphics`, puis les objets sont combinés par `Show`.

Un quart de cercle est créé comme objet graphique ainsi que la bissectrice de l'angle droit du premier quadrant.

```
cercle = Plot[(1 - x^2)^(1/2), {x,0,1},
AspectRatio->Automatic, DisplayFunction->Identity]
bissectrice = Plot[x, {x,0,1},
DisplayFunction->Identity]
```

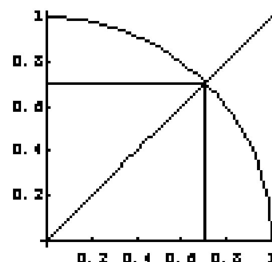
La solution de l'intersection entre droite et cercle est calculée.

```
Solve[{y == (1 - x^2)^(1/2), x == y}, {x,y}];
solution = Flatten[%]
          1              1
{y -> -------, x -> -------}
      Sqrt[2]        Sqrt[2]
```

Des lignes tracées depuis le point d'intersection jusqu'aux axes sont définies comme objet graphique en reprenant les valeurs obtenues au-dessus dans **`solution`**.

```
lignes =
Graphics[Line[{{0,y}, {x,y}, {x,0}}]/.solution,
DisplayFunction->Identity]
Show[cercle, bissectrice, lignes,
DisplayFunction->$DisplayFunction]
```

Les objets **`cercle`**, **`bissectrice`** et **`lignes`** sont superposés par **`Show`**.

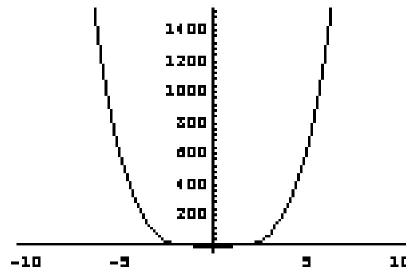


■ Comment insérer un graphique dans un graphique ?

Il s'agit toujours de superposer des objets créés indépendamment les uns des autres. On crée un premier objet puis, après l'avoir dessiné, on dessine par `Epilog` et avec `Rectangle` un objet supplémentaire qui s'inscrira dans la boîte du graphique principal.

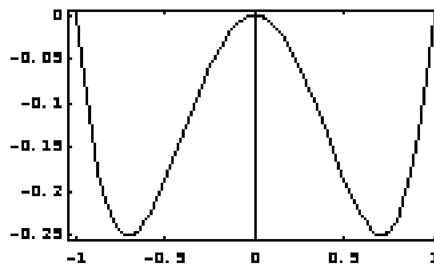
Une fonction est représentée.
À cette échelle, elle apparaît comme une parabole.

```
gr1 = Plot[x^4 - x^2,{x,-10,10}]
```



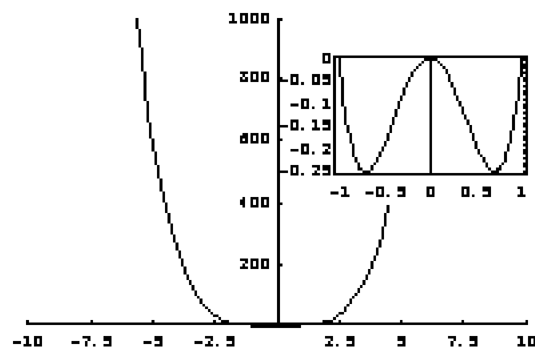
L'existence de deux minima et d'un maximum n'est décelée que dans une représentation des valeurs proches de l'origine.
Dans un tel cas, une double représentation de la fonction est utile.

```
gr2 = Plot[x^4 - x^2, {x,-1,1},
Frame->True]
```



Il est nécessaire de peaufiner les paramètres si on souhaite que les deux graphes ne se mélangent pas.
La représentation de la grande courbe est limitée par l'emploi de **`PlotRange`**. Les coordonnées du rectangle où se trace **`gr2`** sont `{0,200}` pour le coin en bas à gauche et `{10,1100}` pour le coin en haut à droite. Ces valeurs ont été obtenues par tâtonnement.
Notez que les coordonnées utilisées sont celles des axes du graphique principal.

```
gr1 = Plot[x^4 - x^2, {x,-10,4.5},
PlotRange->{{-10,10},{-1,1000}},
DisplayFunction->Identity]
Show[gr1,
Epilog->Rectangle[{0,200},{10,1100},gr2],
DisplayFunction->$DisplayFunction]
```



Supplément sur les fonctions graphiques

■ *Packages* graphiques supplémentaires

□ *Packages* graphiques standards

Il existe des fonctions graphiques supplémentaires fournies dans des fichiers de commandes. Bien que ces *packages* fassent partie de l'installation standard de *Mathematica*, leurs commandes n'appartiennent pas aux commandes de base du noyau et elles ne sont pas chargées au démarrage du programme. Pour pouvoir les utiliser, il faut activer explicitement les *packages* correspondants par `Needs` ou `<<` ou `Get`. Notamment, il y a parmi les *packages* standards, dans le répertoire Packages, les fichiers suivants :
- `Graphics`Graphics``, contient les définitions de plusieurs types de graphiques à échelle logarithmique, de graphiques à barres ou à secteurs, de la commande `DisplayTogether[gr1, gr2,…]` qui équivaut à `Show[gr1,gr2,…,DisplayFunction->$DisplayFunction]` et de différentes versions étendues de `ListPlot3D` ;
- `Graphics`Animation``, regroupe des commandes facilitant l'animation d'une séquence de graphiques ;
- `Graphics`Arrow``, contient la commande `Arrow` qui permet de créer des flèches personnalisées, voir à ce propos *The Mathematica Journal*, vol. 6 n° 2, 1996, p. 52 et s. ;
- `Graphics`ContourPlot3D`` définit des commandes créant des graphiques de contours (en fait des surfaces de niveau) dans l'espace ;
- `Graphics`FilledPlot``, permet de colorer ou griser des surfaces sur un graphique ;
- `Graphics`Graphics3D``, permet de créer des graphiques de *nuages de points* par `ScatterPlot3D`, (à la différence de la fonction standard `ListPlot3D` qui trace une *surface* à partir d'une liste de points « carrée » c'est-à-dire de forme `{{pt11,…,pt1n},…,{ptn1,…, ptnn}}`) ;
- `Graphics`Legend``, offre les deux fonctions `Legend` et `ShowLegend` qui permettent de placer et manipuler une légende ;
- `Graphics`MultipleListPlot``, permet de créer un graphique à partir de listes multiples par `MultipleListPlot[liste1,liste2]` sans passer par la solution standard `ListPlot\[liste1], ListPlot[liste2], Show[%,%%]`. Le manuel des *packages* standards explique aussi comment créer des symboles personnalisés pour chaque liste représentée.

□ *Packages* graphiques de T. Wickham-Jones

La disquette d'accompagnement du livre *Mathematica Graphics* de T. Wickham-Jones, le concepteur des programmes graphiques de *Mathematica*, contient des suppléments intéressants qui, le plus souvent, sont l'amélioration de commandes existantes dans la version standard de *Mathematica*. Parmi ceux-ci, on peut retenir :
- `ExtendGraphics`Plot``, qui étend la commande `Plot` usuelle de façon à mieux traiter les singularités ;
- `ExtendGraphics`Contour``, qui étend la commande `Contour` pour représenter les ensembles irréguliers de points (c'est-à-dire sous forme de liste « non carrée ») ;
- `ExtendGraphics`ConstrainedContour``, qui offre la possibilité de tracer un graphique de contour à partir d'une inégalité ;
- `ExtendGraphics`LabelContour``, qui offre différentes possibilités de mettre des étiquet-tes à des graphiques de contour ;

– `ExtendGraphics`Ticks``, qui présente un ensemble de commandes permettant de contrôler les marques d'axe *(ticks)* dans le détail. Le package standard `Graphics`Graphics`` permet de créer des échelles sophistiquées (`Log`, `LogLog`) en combinaison avec `Ticks`.

■ Exemples de *packages* graphiques supplémentaires

□ Gestion des *ticks*

Le fichier de commandes Ticks dans ExtendGraphics est chargé dans le noyau, le chemin de ExtendGraphics a été préalablement entré dans la variable système `$Path`. **`Names`** nous donne les commandes nouvelles disponibles.

```
Needs["ExtendGraphics`Ticks`"]
Names["ExtendGraphics`Ticks`*"]
{MajorLength, MajorStyle, MinorLength, MinorStyle,
TextFunction, TickFunction, TickLabels,
TickNumbers, TickPosition, TrimText}
```

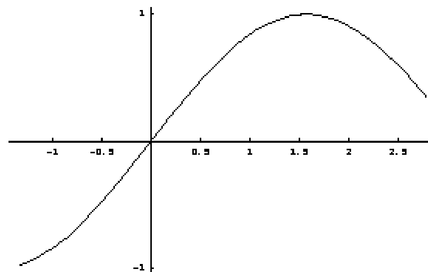
La commande **`TickPosition`** permet de calculer, sur un intervalle donné, des valeurs de marques bien arrondies pour un nombre de marques donné (par excès).

```
?TickPosition
TickPosition[ min, max, num] returns a list of at
most num nicely rounded positions between min and
max. These can be used for tick mark positions.
```

Par exemple, sur l'intervalle **`{-1.320,2.7777}`**, pour **`10`** marques au plus, les meilleures valeurs sont `{-1.,-0.5,0,0.5,1.,1.5,2.,2.5}`.

```
TickPosition[-1.320, 2.7777, 10]
{-1., -0.5, 0, 0.5, 1., 1.5, 2., 2.5}
Plot[Sin[x], {x,-1.320,2.7777},
Ticks->{TickPosition[#1,#2,10]&,
TickPosition[#1,#2,4]&}]
```

Le graphique est rendu avec les marques de graduation calculées par la commande **`TickPosition`** insérée dans l'option **`Ticks`**. Remarquez que **`TickPosition`** est utilisée dans une fonction pure.



□ Animation de graphiques

L'interface de *Mathematica* donne accès par le menu Graph à diverses commandes permet-tant de faire défiler en superposition des graphiques placés en séquence ; il s'ensuit une impres-sion de mouvement animé. Animation... permet de paramétrer la vitesse du mouvement et Animate Selected Graphics de lancer l'animation qui est ensuite contrôlable par boutons. Le fichier `Graphics`Animation`` est une aide appréciable pour créer des graphiques en série.

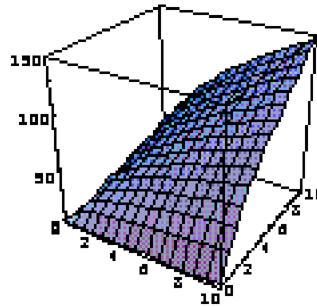
Le fichier de commandes Animation du répertoire Graphics est chargé. Il offre de nombreuses commandes supplémentaires.

```
<<Graphics`Animation`
Names["Graphics`Animation`*"]
{Animate, Animation, AnimationFunction, Closed,
Frames, MovieContourPlot, MovieDensityPlot,
MovieParametricPlot, MoviePlot, MoviePlot3D,
RasterFunction, RotateLights, ShowAnimation,
SpinDistance, SpinOrigin, SpinRange, SpinShow,
SpinTilt}
```

Une fonction à deux variables est représentée par une surface en 3D.

```
cobbdouglas[k_,l_]:= 10 k^(1/2) l^(2/3);
s3D = Plot3D[cobbdouglas[k,l], {k,0,10}, {l,0,10},
BoxRatios->{1,1,1}]
```

Pour mieux l'appréhender, on souhaite la faire pivoter.



La commande **SpinShow** crée **5** images de **s3D** (non reproduites ici), chaque image étant obtenue par une égale rotation.
Pour animer la séquence : sélectionner les cellules de la série puis choisir le menu Graph, puis Animate Selected Graphics, régler l'animation avec les boutons en bas à gauche de la fenêtre de travail.

```
SpinShow[s3D, Frames->5,
SpinRange->{0 Degree, 180 Degree}]
[....................]
{-SurfaceGraphics-, -SurfaceGraphics-,
-SurfaceGraphics-, -SurfaceGraphics-,
-SurfaceGraphics-}
```

□ Superposition de contours en 3D

Le fichier de commandes **Contour\ Plot3D`** est chargé. Une fonction de trois variables est définie.

```
<<Graphics`ContourPlot3D`
cobbdouglas1[k_,l_,z_]:= k^(1/4) l^(1/5) z^(1/3)
```

Un objet graphique contenant le contour pour lequel **cobbdouglas1** est de valeur **3** est calculé.

```
g1 = ContourPlot3D[cobbdouglas1[k,l,z],
{k,0,10}, {l,0,10}, {z,0,10},
Contours->{3},
PlotRange->{{0,10},{0,10},{0,10}},
BoxRatios->{1,1,1},
DisplayFunction->Identity]
-Graphics3D-
```

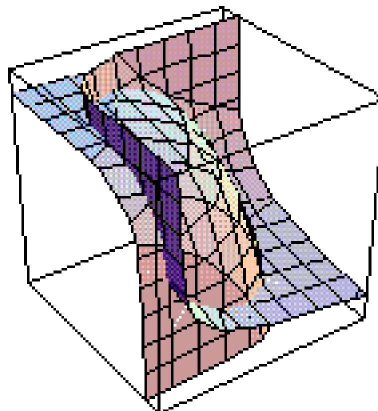
Une autre fonction à trois variables est définie. Son origine est à l'opposé de celle de la première fonction.

```
cobbdouglas2[k_,l_,z_]:=
(10-k)^(1/5) (10-l)^(1/3) (10-z)^(1/4)
```

Le contour pour lequel **cobbdouglas2** est de valeur **3** est calculé.

```
g2 = ContourPlot3D[cobbdouglas2[k,l,z],
{k,0,10}, {l,0,10}, {z,0,10},
Contours->{3},
PlotRange->{{0,10},{0,10},{0,10}},
BoxRatios->{1,1,1},
DisplayFunction->Identity]
-Graphics3D-
Show[g1, g2, DisplayFunction->$DisplayFunction,
ViewPoint->{3.408, -2.373, 2.601}]
```

Les deux objets graphiques sont combinés et affichés par **Show**, le point de vue étant ajusté.



□ Création d'une légende

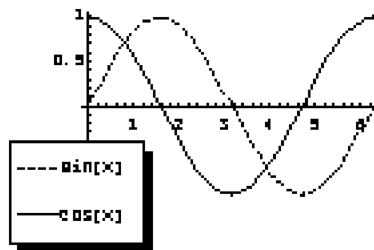
Nous avons déjà vu comment créer du texte simple à fonction de légende par Text. Mais une légende au sens propre reprend dans un cartouche les étiquettes avec leurs symboles (par exemple, les noms des courbes avec l'exemple de leur tracé). Le *package* standard contient des commandes qui permettent de gérer une légende. Pour donner une légende à un graphique créé par Plot, il faut utiliser dans la commande Plot l'option PlotLegend. Pour les autres graphiques, il faut recourir à ShowLegend.

Le fichier de commande **Legend`** est chargé. Il ajoute l'option **PlotLegend** aux options ordinaires. Cette option permet de passer du texte à une légende mais aussi des options relatives à la fonction **Legend** qui est invisible et qui contrôle la légende produite.

```
<<Graphics`Legend`
?PlotLegend
PlotLegend is an option for Plot, which assigns
text to lines in a 2D plot to create a legend for
that plot. PlotLegend->{txt1,txt2...} assigns text
to each line in the fashion of PlotStyle.
PlotLegend also enables Plot to accept options for
Legend, which will modify the legend produced.
```

La représentation de deux fonctions est demandée. Par **PlotStyle**, chaque courbe est dessinée suivant un style spécifique. **PlotLegend** est appelée comme une option supplémentaire de la commande **Plot**, option qui permet d'associer le style de la courbe et son nom en légende.

```
Plot[{Sin[x], Cos[x]}, {x,0,2 Pi},
PlotStyle->{{Thickness[0.01],
Dashing[{0.02,0.02}]},{Thickness[0.005]}},
PlotLegend->{Sin[x],Cos[x]}]
```



Ce n'est pas un succès, la légende écrase le graphique ! Toutefois, il est possible de faire mieux en modifiant diverses options de Legend.

Voici la liste des options utilisables pour une légende.
La plus importante est l'option **LegendPosition**.
LegendSize permet, par tâtonnement, de positionner les étiquettes par rapport aux symboles de la légende.

```
Options[Legend]
{LegendPosition -> {-1, -1},
LegendSize -> Automatic, LegendShadow -> Automatic,
LegendTextSpace -> Automatic,
LegendTextDirection -> Automatic,
LegendTextOffset -> Automatic, LegendLabel -> None,
LegendLabelSpace -> Automatic,
LegendOrientation -> Vertical,
LegendSpacing -> Automatic,
LegendBorder -> Automatic,
LegendBorderSpace -> Automatic,
LegendBackground -> Automatic}
```

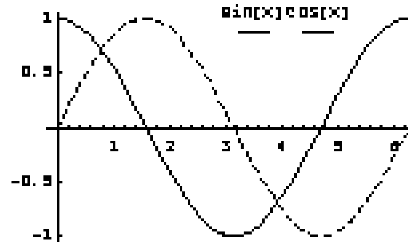
L'utilisateur attentif remarquera que **Legend**, qui n'est pas une commande directement utilisable, a des options qui sont précisément celles que l'on peut entrer dans **Plot** si on utilise **PlotLegend**. En sens inverse, faire **FullOptions[graph]** ne renvoie les valeurs d'aucune des options relatives à **Legend**.

```
?LegendPosition
LegendPosition is an option for Legend, which
specifies the exact location of a legend box (the
lower-left corner). If called from ShowLegend, the
position will be in the coordinate system, with
the graphic centered at {0,0} and scaled to fit
inside {{-1,-1},{1,1}}.
```

PlotLegend définit les étiquettes des symboles. **LegendPosition** positionne la légende en un point défini de la même façon pour **Plot** et pour **ShowLegend** (voir la définition de **LegendPosition**). La boîte contenant la légende a son coin droit positionné par défaut au coin gauche inférieur de la boîte du graphique référencé **{-1,-1}**, le centre étant en **{0,0}**. **LegendSize** utilise un système de mesure que je n'ai pu identifier.

Il est judicieux de s'exercer avec différentes valeurs des paramètres d'option en conservant le tracé de la boîte pour mieux
en apprécier les effets.

```
graph = Plot[{Sin[x],Cos[x]}, {x,0,2 Pi},
PlotStyle-> {{Thickness[0.01],
Dashing[{0.02,0.02}]}, {Thickness[0.005]}},
PlotLegend->{"Sin[x]","Cos[x]"},
LegendPosition->{0,0.4},
LegendShadow->None,
LegendOrientation->Horizontal,
LegendSize->{0.7,0.2}]
```



L'utilisation des autres options fut comme le parcours d'un labyrinthe… Pour des détails sur `PlotLegend`, voir *The Mathematica Journal*, vol. 5, n° 3, 1995, p. 79, et aussi le *Guide To Standard Mathematica Packages*.