In [1]:

```
from IPython.core.display import HTML
css_file = './custom.css'
HTML(open(css_file, "r").read())
```

Out[1]:

In [2]:

```
import sys #only needed to determine Python version number
import matplotlib #only needed to determine Matplotlib version
import scipy # idem
import sympy # idem

print('Python version ' + sys.version)
print('Matplotlib version ' + matplotlib.__version__ )
print('Scipy version ' + scipy.__version__ )
print('Sympy version ' + sympy.__version__ )

%reset -f
```

```
Python version 3.6.2 |Anaconda custom (64-bit)|
(default, Jul 20 2017, 13:51:32)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-1)]
Matplotlib version 2.1.0
Scipy version 1.0.0
Sympy version 1.1.1
```

M62_CM2 Calcul approché (package SciPy) VS formel (package SymPy).

Table of Contents

- 1 Calcul approché avec le module SciPy
  - 1.1 fsolve
  - 1.2 integrate.quad
  - 1.3 odeint
    - 1.3.1 Exemple de résolution approchée d’une équation différentielle
    - 1.3.2 Exemple de résolution approchée d’un système d'équations d
    - 1.3.3 Exemple de résolution approchée d’une équation différentielle
- 2 Calcul formel avec le module sympy

- 2.1 Manipulations algèbriques
  - 2.1.1 `simplify` simplifie des expressions
  - 2.1.2 `cancel` simplifie des fractions de polynômes
  - 2.1.3 `expand` dévéloppe une expression:
  - 2.1.4 `expr.subs(variable,valeur)` remplacer dans `expression` la `variable` par `valeur`
  - 2.1.5 `apart(expr,x)` décompose une fraction en éléments simples
  - 2.1.6 Pour remettre tout ensemble, on utilise `together(expr, x)` :
  - 2.1.7 `factor` factorise un polynôme
- 2.2 `solve` pour la résolution d'(in)équations et systèmes
  - 2.2.1 Solution d'une équation
  - 2.2.2 Solution d'une inéquation
  - 2.2.3 Solution d'un système
- 2.3 Algèbre linéaire
- 2.4 Calcul de limites
- 2.5 Graphe d'une fonction $x \mapsto f(x)$
- 2.6 Calcul de dérivées
- 2.7 Calcul de primitives et intégrales
- 2.8 Développements en série
- 2.9 Interpolation
- 2.10 EDO

3 Références

1 Calcul approché avec le module `SciPy`

Cette librairie est un ensemble très complet de modules d’algèbre linéaire, statistiques et autres algorithmes numériques. Le site de la documentation en fournit la liste : http://docs.scipy.org/doc/scipy/reference (http://docs.scipy.org/doc/scipy/reference)

1.1 `fsolve`

Si on ne peut pas calculer analytiquement la solution d'une équation, on peut l'approcher numériquement comme suit:

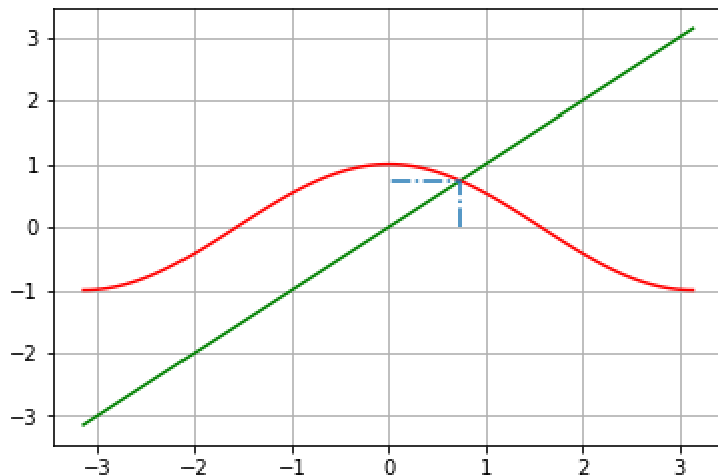
In [3]:

```
from math import cos
from scipy.optimize import fsolve

sol=fsolve(lambda x: x-cos(x), 1)[0]
print(sol)
```

```
0.7390851332151607
```

In [4]:

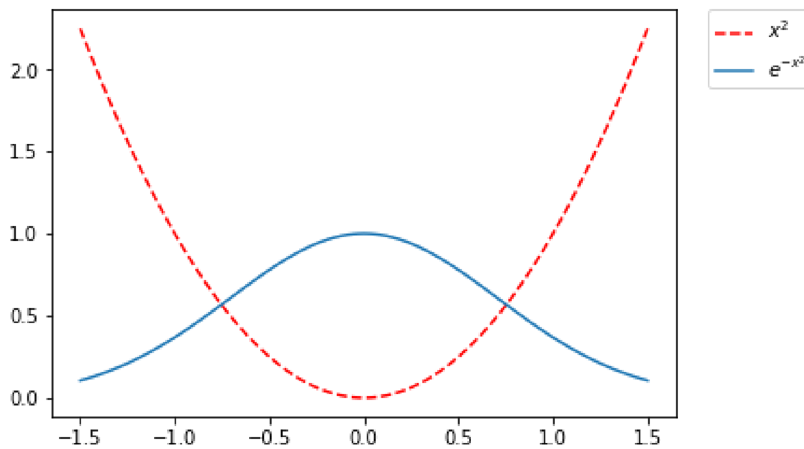
```
%matplotlib inline
from matplotlib.pylab import *
x=linspace(-pi,pi,101)
y=[cos(xi) for xi in x ]
plot(x,x,'g-',x,y,'r-')
plot( [sol,sol,0],[0,sol,sol] ,'-.')
grid(True)
```



In [5]:

```
from math import exp
from matplotlib.pylab import *
from scipy.optimize import fsolve

sol=fsolve(lambda x: x**2-exp(-x*x), -1)
print(sol)

sol=fsolve(lambda x: x**2-exp(-x*x), 0.2)
print(sol)

x=linspace(-1.5,1.5,101)
plot(x,[xi**2 for xi in x],'r--',label=r'$x^2$')
plot(x,[exp(-xi*xi) for xi in x],label=r'$e^{-x^2}$')
legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.);

# figure()
# plot(x,[xi**2-exp(-xi*xi) for xi in x],x,[0 for xi in x])
```

```
[-0.75308916]
[0.75308916]
```



1.2 *integrate.quad*

Pour approcher la valeur numérique d'une intégrale on peut utiliser `integrate.quad`
https://docs.scipy.org/doc/scipy/reference/tutorial/integrate.html (https://docs.scipy.org/doc/scipy/reference/tutorial/integrate.html)

In [6]:

```
from math import *
from scipy import integrate

a=0
b=1
f = lambda x:x**2
integr = integrate.quad(f,a,b)
print("Integrale =",integr[0], " Erreur =",integr[1] )

f = lambda x: exp(-x*x)
print(integrate.quad(f, 0,inf)[0])
print(sqrt(pi)/2)
```

```
Integrale = 0.33333333333333337  Erreur = 3.70074
3415417189e-15
0.8862269254527579
0.8862269254527579
```

In [7]:

```
help(integrate)
```

```
Help on package scipy.integrate in scipy:

NAME
    scipy.integrate

DESCRIPTION
    ==========================================
===
    Integration and ODEs (:mod:`scipy.integrat
e`)
    ==========================================
===

    .. currentmodule:: scipy.integrate

    Integrating functions, given function obje
ct
    ==========================================
==
```

1.3 odeint

L'ECUE M62 n'est qu'une introduction aux méthodes numériques d'approximation d'EDO en utilisant Python comme langage commun de programmation. Cependant une grande partie des méthodes numériques dont vous avez besoin en tant que scientifique (y compris l'intégration numérique d'équations différentielles) se trouve déjà dans le module `SciPy`.
Bien sûr vous devez toujours faire un peu de programmation avec Python

et une compréhension des fondements de la méthode numérique que vous utilisez est toujours indispensables mais avouons que c'est commode quand quelque chose est déjà programmé.

Voyons sur deux exemples comment utiliser la fonction `odeint` du module `SciPy` pour approcher la solution d'abord d'une EDO ensuite d'un système d'EDO (ce qui inclut les équations différentielles d'ordre 2 ou plus).

In [8]:

```
from matplotlib.pylab import *
from scipy.integrate import odeint
```

Le principe d'utilisation de `odeint` (pour intégrer numériquement des équations différentielles) est le suivant: pour avoir une estimation numérique de la solution du problème de

$$\begin{cases} \mathbf{y}'(t) = \boldsymbol{\varphi}(t, \mathbf{y}(t)), \\ \mathbf{y}(t_0) = \mathbf{y}_0 \end{cases}$$

avec $\mathbf{y}(t) = (y_1(t), y_2(t), \dots, y_n(t))$ le vecteur des fonctions recherchées, dépendant de la variable t et $\boldsymbol{\varphi} = (\varphi_1, \varphi_2, \dots, \varphi_n)$ une fonction de forme quelconque, on donne comme argument la fonction $\boldsymbol{\varphi}$ (qui doit avoir deux paramètres, même dans le cas autonome, avec t comme deuxième paramètre), la condition initiale $\mathbf{y}_0$ et le domaine de temps qui nous intéresse (qui commence à t_0). Elle retourne un tableau (même si t était une liste).

Notons que la résolution de $y''(t) = F(t, y(t), y'(t))$ passera par celle du système différentiel

$$\begin{cases} y_1'(t) = y_2(t), \\ y_2'(t) = F(t, y_1(t), y_2(t)) \end{cases}$$

avec $y(t) = y_1(t)$, $y'(t) = y_1'(t) = y_2(t)$ et $\varphi_1(t, y_1(t), y_2(t)) = y_2(t)$, $\varphi_2(t, y_1(t), y_2(t)) = F(t, y_1(t), y_2(t))$.

1.3.1 Exemple de résolution approchée d'une équation différentielle

En guise d'exemple, considérons une équation logistique simple de la forme

$$y'(t) = ry(t)\left(1 - \frac{y(t)}{K}\right).$$

Pour les simulations on prendra $r = 1.5$, $K = 6$ et $y_0 = 1$. On crée alors la fonction φ

In [9]:

```
# def phi(y,t):
#     return 1.5*y*(1.-y/6.)
# ou
phi = lambda y,t : 1.5*y*(1.-y/6.)
```

La fonction `odeint` peut être appelée avec au minimum trois arguments: la fonction φ, la condition initiale $y(t_0) = y_0$ et le temps t comme variable principale:

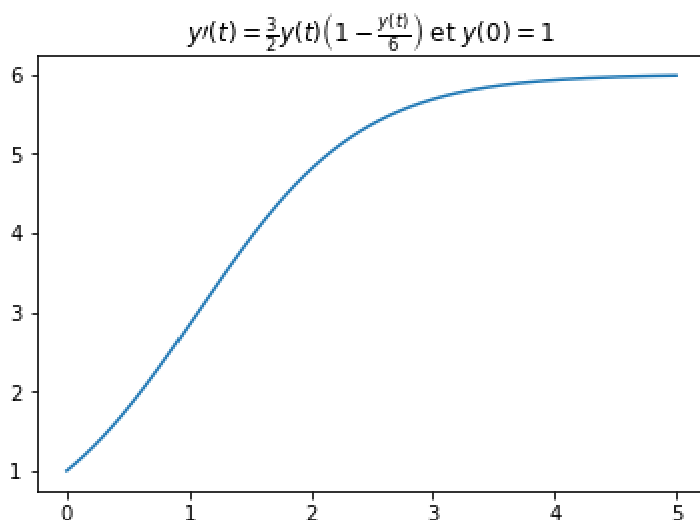
In [10]:

```
y0  = 1.0
t   = linspace(0,5,201)
sol = odeint(phi,y0,t)
```

La solution peut alors être tracée simplement

In [11]:

```
plot(t,sol)
title(r'$y\prime(t)=\frac{3}{2}y(t)\left(1-\frac{y(t)}{6}\righ
show()
```

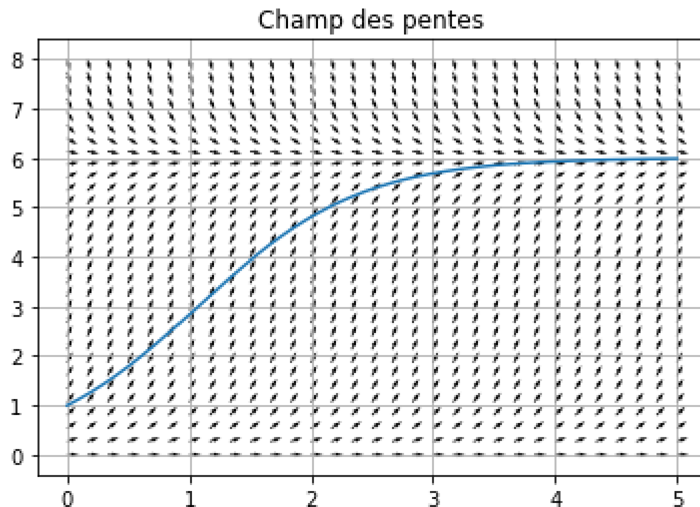


NB: le nombre de points en lesquels les résultats sont évalués n'est pas (du moins directement) relié à la précision des calculs internes (ne pas imaginer que cela fixe le pas de la méthode, en particulier).

Notons que la fonction `quiver` du module `matplotlib` permet de tracer un champ de vecteurs. Utilisons-la pour obtenir celui associé à notre équation différentielle. La courbe bleu est la solution déterminée par `odeint` .

In [12]:

```
T,Y = np.meshgrid(linspace(0,5,31),linspace(0,8,31))
U = 1.
V = phi(Y,T)
r=sqrt(U**2+V**2)
quiver(T, Y, U/r, V/r)
plot(t,sol)
grid(True)
title(r'Champ des pentes');
```



1.3.2 Exemple de résolution approchée d’un système d'équations différentielles

Proposées indépendamment par Alfred James Lotka en 1925 et Vito Volterra en 1926, les équations dites de Lotka-Volterra ou modèle proie-prédateur, peuvent servir à reproduire les évolutions temporelles de deux populations animales, l’une étant le prédateur de l'autre.

Si ces deux populations sont représentées par des variables continues $y_1(t)$ et $y_2(t)$, le système différentiel en question est alors

$$\begin{pmatrix} y_1 \\ y_2 \end{pmatrix}'(t) = \begin{pmatrix} \varphi_1(t, y_1(t), y_2(t)) \\ \varphi_2(t, y_1(t), y_2(t)) \end{pmatrix} = \begin{pmatrix} y_1(t)(a - by_2(t)) \\ -y_2(t)(c - dy_1(t)) \end{pmatrix}.$$

Pour les simulations on prendra $a = 2$, $b = 1$, $c = 1$ et $d = 0.3$. On crée alors la fonction vectorielle $\boldsymbol{\varphi} = (\varphi_1, \varphi_2)$:

In [13]:

```
# def phi(y,t):
#     return [ y[0]*(2.-y[1]) , -y[1]*(1.-0.3*y[0]) ]
# ou
phi = lambda y,t : [ y[0]*(2.-y[1]) , -y[1]*(1.-0.3*y[0]) ]
```

En faisant varier le temps sur l'intervalle $[0; 10]$ et en prenant comme condition initiale le vecteur $\mathbf{y}_0 = (2, 1)$ on écrit

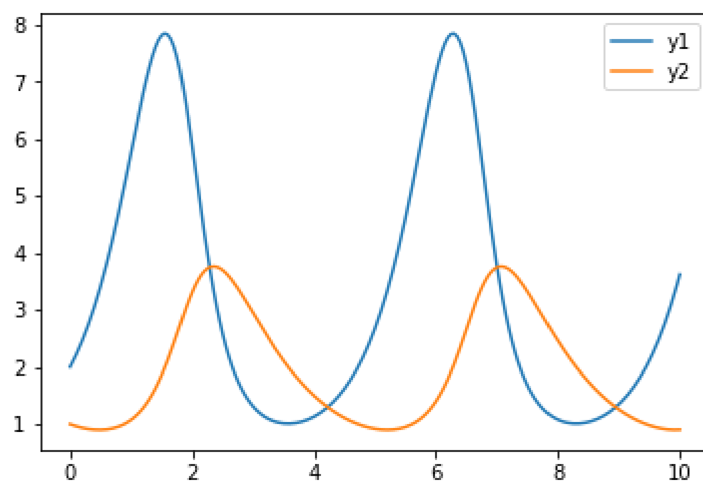
In [14]:

```
y0  = [2.0,1.0]
t   = linspace(0,10,201)
sol = odeint(phi,y0,t)
```

Le tracé des évolutions de y_1 et y_2 en fonction du temps t peut être obtenu par

In [15]:

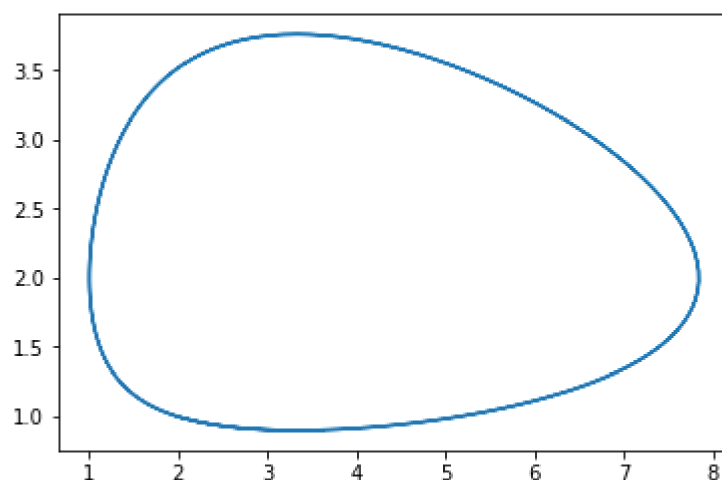
```
plot(t,sol)
legend(["y1","y2"]);
```



Le tracé des évolutions de y_2 en fonction de y_1 peut être obtenu par

In [16]:

```
plot(sol[:,0],sol[:,1]);
```



1.3.3 Exemple de résolution approchée d'une équation différentielle d'ordre 2

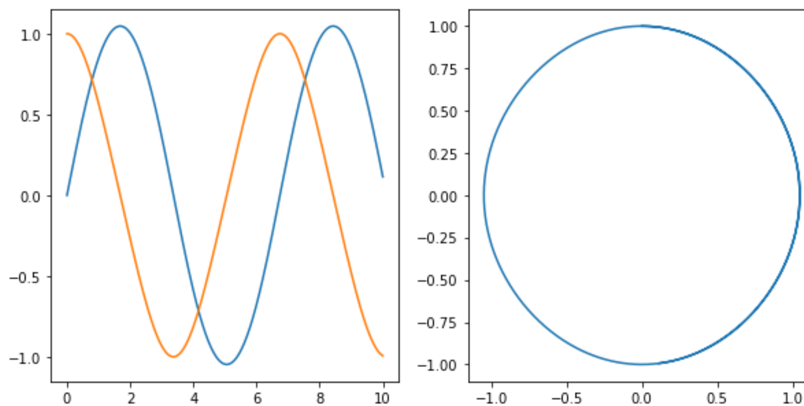
Considérons l'EDO $y''(t) = -\sin(y(t))$ qui décrit le mouvement d'un pendule non amorti, équivalente au système différentiel

$$\begin{cases} y_1'(t) = y_2(t), \\ y_2'(t) = -\sin(y_1(t)) \end{cases}$$

avec $y(0) = 0$ et $y'(0) = 1$.

In [17]:

```
phi = lambda y,t : [ y[1], -math.sin(y[0])  ]
y0  = [0,1.0]
t   = linspace(0,10,1001)
sol = odeint(phi,y0,t)
figure(1, figsize=(10, 5))
subplot(1,2,1)
plot(t,sol)
subplot(1,2,2)
plot(sol[:,0],sol[:,1]);
```



2 Calcul formel avec le module `sympy`

`SymPy` est une bibliothèque Python pour les mathématiques symboliques. Elle prévoit devenir un système complet de calcul formel ("CAS" en anglais : "Computer Algebra System") tout en gardant le code aussi simple que possible afin qu'il soit compréhensible et facilement extensible.

Quelques commandes : https://www.sympygamma.com/ (https://www.sympygamma.com/)

Pour tester en ligne: https://live.sympy.org/ (https://live.sympy.org/)

In [18]:

```
%reset -f
from sympy import *
```

Pour un meilleur aspect des résultats affichées on utilisera :

In [19]:

```
init_printing()
```

Le symbole `oo` est utilisé pour une classe définissant l'infini mathématique :

In [20]:

```
oo+1
```

Out[20]:

$$\infty$$

Contrairement à d'autres systèmes de calcul formel, SymPy vous oblige à déclarer les variables symboliques explicitement :

In [21]:

```
#  premiere methode pour declarer des variables
x = Symbol('x')
y = Symbol('y')
#  deuxieme methode pour declarer des variables
h,k = symbols('u,v')
#  troisieme methode pour declarer des variables
var('u,v');
```

2.1 Manipulations algèbriques

In [22]:

```
(x+y)+(x-y)
```

Out[22]:

$$2x$$

2.1.1 `simplify` simplifie des expressions

In [23]:

```
expr=sin(x)**2 + cos(x)**2
expr
```

Out[23]:

$$\sin^2(x) + \cos^2(x)$$

In [24]:

```
simplify(expr)
```

Out[24]:

$$1$$

In [25]:

```
expr=(x**3 + x**2 - x - 1)/(x**2 + 2*x + 1)
expr
```

Out[25]:

$$\frac{x^3 + x^2 - x - 1}{x^2 + 2x + 1}$$

In [26]:

```
simplify(expr) # ou expr.simplify()
```

Out[26]:

$$x - 1$$

2.1.2 `cancel` simplifie des fractions de polynômes

In [27]:

```
expr=(x**3-y**3)/(x**2-y**2)
expr
```

Out[27]:

$$\frac{x^3 - y^3}{x^2 - y^2}$$

In [28]:

```
cancel(expr)
```

Out[28]:

$$\frac{x^2 + xy + y^2}{x + y}$$

2.1.3 `expand` dévéloppe une expression:

In [29]:

```
expr=(x+y)**2
expr
```

Out[29]:

$$(x + y)^2$$

In [30]:

```
expand(expr) # ou ((x+y)**2).expand()
```

Out[30]:

$$x^2 + 2xy + y^2$$

2.1.4 `expr.subs(variable,valeur)` remplacer dans `expression` la `variable` par `valeur`

In [31]:

```
(expr).subs(x,1)
```

Out[31]:

$$(y + 1)^2$$

In [32]:

```
expr = x*x + x*y + x*y + y*y
expr.subs({x:1, y:2})
```

Out[32]:

$$9$$

In [33]:

```
expr.subs({x:1-y})
_.simplify()
```

Out[33]:

$$9$$

2.1.5 `apart(expr,x)` décompose une fraction en éléments simples

In [34]:

```
expr=(x**2+8*x+4)/(x**2-4)
expr
```

Out[34]:

$$\frac{x^2 + 8x + 4}{x^2 - 4}$$

In [35]:

```
apart( expr ,x )
```

Out[35]:

$$1 + \frac{2}{x+2} + \frac{6}{x-2}$$

2.1.6 Pour remettre tout ensemble, on utilise `together(expr, x)` :

In [36]:

```
expr=-1/(x+2)+1/(x+1)
expr
```

Out[36]:

$$-\frac{1}{x+2} + \frac{1}{x+1}$$

In [37]:

```
together(expr ,x)
```

Out[37]:

$$\frac{1}{(x+1)(x+2)}$$

2.1.7 `factor` factorise un polynôme

In [38]:

```
expr=x**4/2+5*x**3/12-x**2/3
expr
```

Out[38]:

$$\frac{x^4}{2}+\frac{5x^3}{12}-\frac{x^2}{3}$$

In [39]:

```
factor(expr)
```

Out[39]:

$$\frac{x^2}{12}(2x-1)(3x+4)$$

2.2 `solve` pour la résolution d'(in)équations et systèmes

2.2.1 Solution d'une équation

On utilise la commande `solve` pour résoudre une équation:

In [40]:

```
eq=x**4-1
eq
```

Out[40]:

$$x^4-1$$

In [41]:

```
solve(eq,x)
```

Out[41]:

$$[-1, \quad 1, \quad -i, \quad i]$$

In [42]:

```
eq=log(x)-x
eq
```

Out[42]:

$$-x + \log(x)$$

In [43]:

```
solve(eq)
```

Out[43]:

$$[-\operatorname{LambertW}(-1)]$$

In [44]:

```
var('a,b,c')
expr = a*x**2 + b*x + c
sol=solve(expr, x)
sol
```

Out[44]:

$$\left[\frac{1}{2a}\left(-b + \sqrt{-4ac + b^2}\right), \quad -\frac{1}{2a}\left(b + \sqrt{-4ac + b^2}\right)\right]$$

2.2.2 Solution d'une inéquation

Pour resoudre une inéquation on utilise `solve_univariate_inequality`

In [45]:

```
solve_univariate_inequality(x**2 > 4, x)
```

Out[45]:

$$(-\infty < x \wedge x < -2) \vee (2 < x \wedge x < \infty)$$

2.2.3 Solution d'un système

On peut résoudre des systèmes (linéaires ou non linéaires):

In [46]:

```
x, y, z = symbols('x, y, z')
eqs=[3*x-z+2, y-3*z-5, x+y-z]
eqs
```

Out[46]:

$$[3x - z + 2, \quad y - 3z - 5, \quad x + y - z]$$

In [47]:

```
solve(eqs,[x,y,z])
```

Out[47]:

$$\left\{x : -\frac{9}{7}, \quad y : -\frac{4}{7}, \quad z : -\frac{13}{7}\right\}$$

In [48]:

```
eqs=[x**2 + y**2 - 1, x**2 - y**2 - S(1) / 2]
eqs
```

Out[48]:

$$\left[x^2 + y^2 - 1, \quad x^2 - y^2 - \frac{1}{2}\right]$$

In [49]:

```
solve(eqs, x, y)
```

Out[49]:

$$\left[\left(-\frac{\sqrt{3}}{2}, \quad -\frac{1}{2}\right), \quad \left(-\frac{\sqrt{3}}{2}, \quad \frac{1}{2}\right), \quad \left(\frac{\sqrt{3}}{2}, \quad -\frac{1}{2}\right)\right]$$

Notons l'utilisation de `S()` dans la definition de l'équation. En effet, si on écrit $\frac{1}{2}$ directement comme `1/2`, Python traduira cette écriture comme 0.5. Pour que Python considère vraiment la fraction, on peut par exemple forcer le numerateur a être un entier en utilisant la fonction `S()`.

NB Il faut toujours avoir un expris critique même lors de calculs formels ! Voici un exemple. Soit le système linéaire (avec paramètre):

$$\begin{cases} x - \alpha y = 1, \\ \alpha x - y = 1. \end{cases}$$

Par la méthode de Gauss ($L_2 \leftarrow L_2 - \alpha L_1$) on obtient

$$\begin{cases} x - \alpha y = 1, \\ (\alpha^2 - 1)y = 1 - \alpha. \end{cases}$$

Comme $\alpha^2 - 1 = (\alpha - 1)(\alpha + 1)$ on conclut que

- si $\alpha = 1$ (la dernière équation correspond à $0 = 0$) alors le système possède une infinité de solutions,
- si $\alpha = -1$ (la dernière équation correspond à $0 = 2$) alors le système ne possède aucune solution,
- si $\alpha \notin \{-1; 1\}$ alors le système possède une solution unique $x = \frac{1}{\alpha+1}$ et $y = -\frac{1}{\alpha+1}$.

Que nous dit `sympy` ?

In [50]:

```
x,y,alpha=symbols('x,y,alpha')
solve([x-alpha*y-1,alpha*x-y-1],[x,y])
```

Out[50]:

$$\left\{x : \frac{1}{\alpha + 1}, \quad y : -\frac{1}{\alpha + 1}\right\}$$

In [51]:

```
expr1 = 2*x + 3*y - 6
expr2 = 3*x + 2*y - 12
solnliste=solve((expr1, expr2), dict=True)
solnliste
# `(dict=True)` specifies that we want the result to be return
```

Out[51]:

$$\left[\left\{x : \frac{24}{5}, \quad y : -\frac{6}{5}\right\}\right]$$

In [52]:

```
soln = solnliste[0] # Le dictionnaire est dans une liste !
expr1.subs({x:soln[x], y:soln[y]})
```

Out[52]:

$$0$$

2.3 Algèbre linéaire

On définit une matrice

In [53]:

```
M = Matrix(((1,1), (2,-1)))
M
```

Out[53]:

$$\begin{bmatrix} 1 & 1 \\ 2 & -1 \end{bmatrix}$$

In [54]:

```
det(M)
```

Out[54]:

$$-3$$

On calcule les valeurs propres et vecteurs propres

In [55]:

```
P, D = M.diagonalize()
```

In [56]:

```
P
```

Out[56]:

$$\begin{bmatrix} -\frac{\sqrt{3}}{2} + \frac{1}{2} & \frac{1}{2} + \frac{\sqrt{3}}{2} \\ 1 & 1 \end{bmatrix}$$

In [57]:

```
D
```

Out[57]:

$$\begin{bmatrix} -\sqrt{3} & 0 \\ 0 & \sqrt{3} \end{bmatrix}$$

In [58]:

```
simplify(P*D*P**-1)
```

Out[58]:

$$\begin{bmatrix} 1 & 1 \\ 2 & -1 \end{bmatrix}$$

Polynôme caractéristique

In [59]:

```
var('t')
p=M.charpoly(t)
factor(p)
```

Out[59]:

$t^2 - 3$

2.4 Calcul de limites

Pour calculer une limite, la syntaxe est `limit(function, variable, point)`. Par exemple, pour calculer $\lim_{x \to 0} f(x)$ il suffit d'entrer `limit(f, x, 0)` (ou `f.limit(x,0)`):

In [60]:

```
limit(cos(x),x,0)
```

Out[60]:

1

In [61]:

```
limit(x,x,oo)
```

Out[61]:

∞

In [62]:

```
limit(1/x,x,oo)
```

Out[62]:

$$0$$

In [63]:

```
limit(x**x,x,0)
```

Out[63]:

$$1$$

On peut créer une limite sans l'évaluer avec `Limit` puis l'valuer avec `doit()` :

In [64]:

```
lim=Limit(1/x,x,oo)
lim
```

Out[64]:

$$\lim_{x \to \infty} \frac{1}{x}$$

In [65]:

```
lim.doit()
```

Out[65]:

$$0$$

On peut aussi indiquer la direction (`dir="-"` pour la gauche, `dir="+"` pour la droite)

In [66]:

```
limit(1/x,x,0,dir="-")
```

Out[66]:

$$-\infty$$

In [67]:

```
limit(1/x,x,0,dir="+")
```

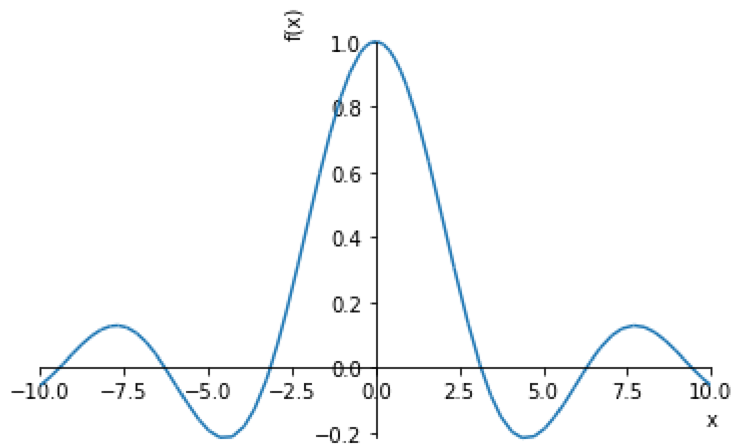
Out[67]:

∞

2.5 Graphe d'une fonction $x \mapsto f(x)$

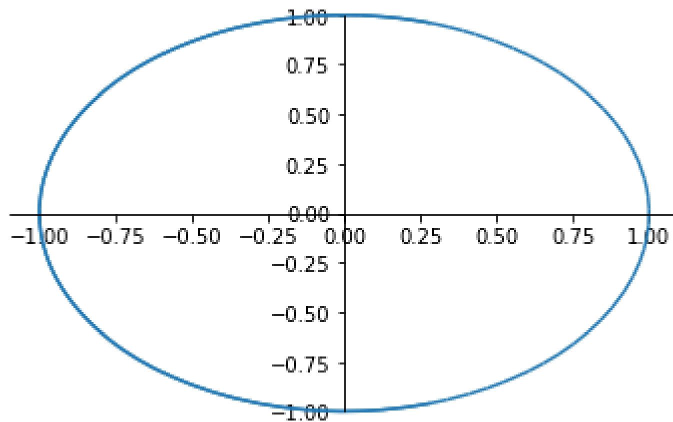
In [68]:

```
plot(sin(x)/x);
```



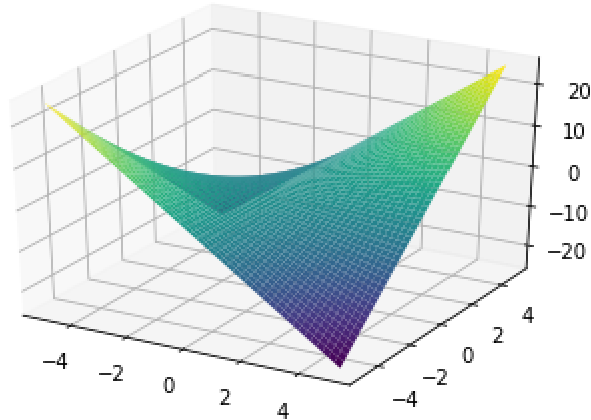
In [69]:

```
from sympy.plotting import plot_parametric
plot_parametric(cos(u), sin(u), (u, -5, 5));
```



In [70]:

```
from sympy.plotting import plot3d
plot3d(x*y, (x, -5, 5), (y, -5, 5));
```



2.6 Calcul de dérivées

Pour calculer des dérivées, la syntaxe est `diff(f,x)` :

In [71]:

```
diff(sin(x),x)
```

Out[71]:

$\cos(x)$

In [72]:

```
diff(sin(2*x),x)
```

Out[72]:

$2\cos(2x)$

Vérifions ce résultat :

In [73]:

```
display(Limit((sin(2*(x+h))-sin(2*x))/h,h,0))
limit((sin(2*(x+h))-sin(2*x))/h,h,0)
```

$$\lim_{u \to 0^+}\left(\frac{1}{u}(-\sin(2x) + \sin(2u + 2x))\right)$$

Out[73]:

$$2\cos(2x)$$

Dérivée partielle

In [74]:

```
diff(x*y**2,y)
```

Out[74]:

$$2xy$$

Dérivée d'ordre n:

In [75]:

```
diff(x**4,x,3)
```

Out[75]:

$$24x$$

On peut créer un dérivée sans l'évaluer avec `Derivative` puis l'évaluer avec `doit()` :

In [76]:

```
deriv=Derivative(y*x**4,x,3)
display(deriv)
deriv.doit()
```

$$\frac{\partial^3}{\partial x^3}\left(x^4 y\right)$$

Out[76]:

$$24xy$$

2.7 Calcul de primitives et intégrales

Pour calculer une primitive on utilise la synatxe `integrate(f,x)` :

In [77]:

```
integrate(sin(x),x)
```

Out[77]:

$$-\cos(x)$$

In [78]:

```
integrate(exp(-x**2), x)
```

Out[78]:

$$\frac{\sqrt{\pi}}{2}\operatorname{erf}(x)$$

On peut créer une primitive sans l'évaluer avec `Integral` puis l'évaluer avec `doit()` :

In [79]:

```
primit=Integral(sin(x),x)
display(primit)
primit.doit()
```

$$\int \sin(x)\, dx$$

Out[79]:

$$-\cos(x)$$

Pour calculer l'intégrale définie $\int_a^b f(x)\mathrm{d}x$ on utilise la syntaxe `integrate(f,(x,a,b))`

In [80]:

```
integrate(sin(x),(x,0,pi/2))
```

Out[80]:

$$1$$

On peut aussi calculer une intégrale impropre :

In [81]:

```
integrate(exp(-x),(x,0,oo))
```

Out[81]:

$$1$$

2.8 Développements en série

Pour calculer un dévéloppement en série la syntaxe est `series(fonction,variable,point,ordre)` :

In [82]:

```
series(cos(x),x,0,5)
```

Out[82]:

$$1 - \frac{x^2}{2} + \frac{x^4}{24} + \mathcal{O}\left(x^5\right)$$

In [83]:

```
series(sqrt(1+x),x,0,5)
```

Out[83]:

$$1 + \frac{x}{2} - \frac{x^2}{8} + \frac{x^3}{16} - \frac{5x^4}{128} + \mathcal{O}\left(x^5\right)$$

In [84]:

```
series(2*x+sqrt(x**2-1),x,oo,1)
```

Out[84]:

$$3x + \mathcal{O}\left(\frac{1}{x}; x \to \infty\right)$$

2.9 Interpolation

Pour calculer le polynôme d'interpolation d'une liste de point on utilise la commande `interpolate(liste,variable)` :

In [85]:

```
var('a,y_a,b,y_b,x')
pts=[(a,y_a),(b,y_b)]
interpolate(pts,x)
```

Out[85]:

$$-\frac{ay_b}{-a+b}-\frac{by_a}{a-b}+\frac{xy_a}{a-b}+\frac{xy_b}{-a+b}$$

In [86]:

```
pts=[(0,1),(1,2),(2,0)]
interpolate(pts,x)
```

Out[86]:

$$-\frac{3x^2}{2}+\frac{5x}{2}+1$$

In [87]:

```
pts=[(-a,0),(0,y_b),(a,0)]
interpolate(pts,x).collect(y_b)
```

Out[87]:

$$y_b\left(1-\frac{x^2}{a^2}\right)$$

2.10 EDO

[https://docs.sympy.org/latest/modules/solvers/ode.html (https://docs.sympy.org/latest/modules/solvers/ode.html)](https://docs.sympy.org/latest/modules/solvers/ode.html)

Considérons le problème de Cauchy

$$\begin{cases} u'(x) = -3x^2u(x) + 6x^2, \\ u(0) = 2. \end{cases}$$

In [88]:

```
x=Symbol('x')
C1=Symbol('C1')
u=Function('u')
f=6*x**2-3*x**2*u(x)
edo=Eq(diff(u(x),x),f)
edo
```

Out[88]:

$$\frac{d}{dx}u(x) = -3x^2u(x) + 6x^2$$

In [89]:

```
classify_ode(edo)
```

Out[89]:

```
('separable',
 '1st_exact',
 '1st_linear',
 'Bernoulli',
 'almost_linear',
 '1st_power_series',
 'lie_group',
 'separable_Integral',
 '1st_exact_Integral',
 '1st_linear_Integral',
 'Bernoulli_Integral',
 'almost_linear_Integral')
```

In [90]:

```
solgen=dsolve(edo,u(x))
solgen
```

Out[90]:

$$u(x) = C_1e^{-x^3} + 2$$

Prise en compte des conditions initiales:

In [91]:

```
x0=0
u0=4
consts = solve( [solgen.rhs.subs(x,x0)-u0 ], dict=True)[0]
consts
```

Out[91]:

$\{C_1 : 2\}$

In [92]:

```
solpar=solgen.subs(consts)
solpar
```

Out[92]:

$u(x) = 2 + 2e^{-x^3}$

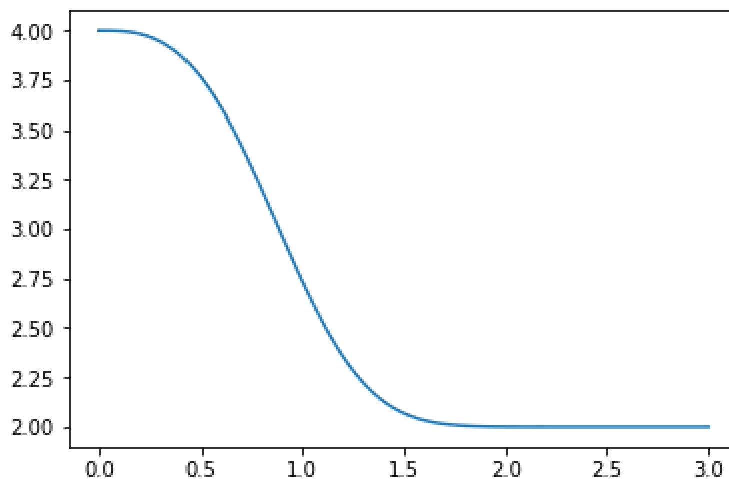
On transforme le résultat en une fonction:

In [93]:

```
func = lambdify(x,solpar.rhs,'numpy')
```

In [94]:

```
from matplotlib.pylab import *
xx=linspace(0,3,101)
yy=func(xx)
plot(xx,yy);
```



3 Références

- Numerical Python, A Practical Techniques Approach for Industry (http://jrjohansson.github.io/numericalpython.html)
- IPython Interactive Computing and Visualization Cookbook, Second Edition (2018) (https://ipython-books.github.io/), by Cyrille Rossant, contains over 100 hands-on recipes on high-performance numerical computing and data science in the Jupyter Notebook.
- Python Data Science Handbook: full text in Jupyter Notebooks (https://jakevdp.github.io/PythonDataScienceHandbook/)

In [ ]: