Une très courte introduction à SymPy (Python)

Vincent Jalby

Université de Limoges

25 août 2023

1 Introduction

Le langage de programmation **Python** est particulièrement adapté aux mathématiques. De nombreuses bibliothèques (*libraries*) permettent d'étendre ses fonctions à de nombreux domaines d'application. Nous nous intéressons ici plus particulièrement à **Sympy** permettant d'effectuer des calculs formels (symboliques) « comme en cours ».
D'autres bibliothèques seront utilisées explicitement ou implicitement : **matplotlib**, **jupyterlab**.

2 Installation

Les plus aguerris pourront installer **Python 3.x** en le téléchargeant directement depuis le site python.org puis en installant les bibliothèques citées précédemment (il faudra pour cela utiliser la commande `pip` ou `pip3`).

Mais il est sans aucun doute beaucoup plus facile d'installer la distribution **Anaconda** à partir du site anaconda.com. Elle permet d'installer **Python** ainsi que toutes les bibliothèques nécessaires (et beaucoup plus encore) en un simple clic. Inconvénient, elle va utiliser environ 7 gigaoctets d'espace sur votre disque dur.

La dernière méthode consiste à ne rien installer, mais à utiliser **Anaconda** dans le *cloud* (en étant donc connecté à Internet). L'utilisation d'un compte gratuit sera largement suffisante à notre niveau. Pour cela, il suffit de remplir le formulaire **Sign Up** sur le site anaconda.cloud.

3 Utilisation

Il existe (au moins) trois façons d'utiliser **python**.

Le mode console, par exemple avec l'application IDLE inclus dans l'installation standard de **Python**, consiste à taper des instructions et à obtenir immédiatement le résultat correspondant. L'affichage des résultats est basique (ASCII) et il y a peu de possibilité d'enregistrer son travail si ce n'est le copier-coller.

On peut aussi créer un programme complet (fichier `.py`) puis l'executer avec **Python**, par exemple avec IDLE, mais l'affichage restera basique, et nécessitera d'utiliser la fonction `print()` pour afficher le moindre résultat.

La méthode *moderne* adaptée aux sciences est l'utilisation d'un **notebook** de type **Jupyter**. On retrouve l'interactivité du mode console, avec un affichage amélioré à travers un navigateur web. L'enregistrement est facilité, toujours à travers une interface web. C'est cette méthode que nous allons privilégier par la suite.

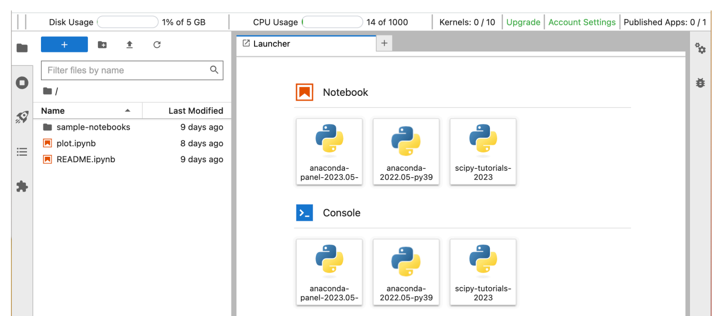
4 Utilisation de Jupyter

Si vous avez tenté une installation standard de **Python** et de ses bibliothèques, vous lancez Jupyter (et donc **Python**) à l'aide de l'instruction `jupyter-lab`. Mais peu de chance que cela fonctionne au premier coup !

Si vous avez installé la distribution de **Anaconda**, lancez l'application **Anaconda-Navigator** puis cliquez sur le bouton **Launch** de la case **JupyterLab**.

Si vous utilisez la version **cloud** de **anaconda**, connectez-vous sur anaconda.cloud puis cliquez sur le menu **Notebooks**.

Quelque soit la méthode utilisée, vous obtiendrez dans votre navigateur un écran proche du suivant :



La partie de gauche permet de naviguer et de gérer les fichiers (*notebooks*) **Python** créés. Celle de droite affiche les notebooks.
Pour créer votre premier notebook, il suffit de cliquer sur le premier icône de la liste dans la zone **Notebooks**.

5 Notions de base

Un **notebook** est composé d'une suite de lignes de commandes (dans la suite en gris) et de résultats (dans la suite en bleu) :

```
[1]: 1+1
[1]: 2
```

Chaque ligne de commande peut contenir une ou plusieurs instructions. Dans ce cas, il faudra les séparer par des points-virgules (;). Pour exécuter une ligne de commande, il suffit de placer le curseur dans la ligne (inutile de se positionner à la fin de la ligne !) contenant la ou les commandes et d'appuyer sur les touches Majuscule et Retour ou Contrôle et Entrée.

Pour rappeler le dernier résultat (c'est-à-dire, le dernier calcul effectué par **Python** lors de la session courante), on utilise l'instruction (_) (tiret bas ou underscore) :

```
_ + 3
```

5

On peut enregistrer une valeur (nombre) ou une expression (fonction, équation, etc) à l'aide de l'opérateur d'affectation (=) :

```
x0 = 12
```

puis l'utiliser comme un symbole mathématique :

```
x0 + x0 + 3
```

27

Attention de n'utiliser que des caractères alphanumériques ASCII (A-Z, 0-9) dans les noms d'affectation.

6 SymPy

La bibliothèque **Sympy** permet d'effectuer de nombreux calculs formels dans **Python** tels que calculs de dérivées, de limites, d'intégrales ou encore résolutions d'équations.

Il faut d'abord commencer par *importer* la bibliothèque à l'aide de l'instruction suivante :

```
from sympy import *
```

On déclare ensuite les variables (au sens mathématique) que l'on souhaite utiliser, par exemple x et y :

```
x, y = symbols('x y')
```

Il est alors possible d'effectuer des calculs avec ces variables, par exemple :

```
3*x - x + 1 + x*y/x
```

$2x + y + 1$

Lors de la déclaration des variables, il est parfois utile de préciser leur type (réel ou entier) :

```
x, y = symbols('x y', real=True)
```

```
n = symbols('n', integer=True)
```

7 Calcul dans ℝ

On peut utiliser les opérations standards de ℝ : somme (`+`), différence (`-`), produit (`*`), puissance (`**`) :

```
1+2*3+2**4
```

23

La division (`/`) donne la valeur numérique du résultat :

```
3/9
```

0.3333333333333333

Pour obtenir la version rationnelle, on utilise la fonction `Rational()` :

```
Rational(3,9)
```

$\frac{1}{3}$

8 Fonctions usuelles

SymPy définit les fonctions usuelles avec les notations mathématiques standards : `sin()`, `cos()`, `tan()`, `ln ()`, `exp()`, etc.

```
sin(pi/3)
```

$\frac{\sqrt{3}}{2}$

La constante $\pi = 3.14\ldots$ s'obtient avec l'instruction `pi`. De même, $e = 2.71\ldots$ s'obtient avec l'instruction `E`. Racine carrée et valeur absolue sont obtenues via les fonctions `sqrt()` (SQuare RooT) et `abs()` :

```
sqrt(8)
```

$2\sqrt{2}$

On peut forcer l'évaluation numérique (floating-point) d'un résultat avec la propriété `evalf()` :

```
sqrt(8).evalf()
```

2.82842712474619

La factorielle (!) est obtenue avec la fonction `factorial()` et les coefficients binomiaux (combinaisons) avec `binomial(n,k)`.

9 Simplification

Les instructions `expand()`, `factor()`, `simplify()` permettent de développer, factoriser, simplifier des expressions mathématiques :

```
expand((x-1)**2)
```

$x^2 - 2x + 1$

```
factor(x**2-1)
```

$(x-1)(x+1)$

```
simplify(x**2/(x+x**3))
```

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

10 Résolution d'équations

On utilise l'instruction **solve()** pour trouver les solutions exactes d'une équation, en indiquant l'expression devant s'annuler sans le « $= 0$ » (ici $x^2 + x - 2$) et la variable (ici x) :

```
solve(x**2+x-2, x)
```

$[-2, 1]$

Cela fonctionne de même pour les systèmes d'équations, en regroupant les équations (et variables) entre crochets :

```
solve([x+2*y-7,x-y-1],[x,y])
```

$\{x: 3, \ y: 2\}$

Pour obtenir les racines d'un polynôme avec leur ordre de multiplicité, on utilise **roots()** :

```
roots(x**3-3*x**2-9*x+27,x)
```

$\{-3 : 1, \ 3 : 2\}$

Lorsqu'il n'est pas possible de résoudre formellement l'équation (ou le système), on peut utiliser la fonction **nsolve()** pour trouver *une* solution numérique dans un intervalle (ici, $[0, 1]$) :

```
nsolve(cos(x)-x,x,(0,1))
```

0.739085133215161

11 Inégalités

On procède de même pour résoudre les inéquations :

```
reduce_inequalities(x**2-1 >= 0, x)
```

$1 \leqslant x \vee x \leqslant -1$

La lecture du résultat est moins aisée. Le symbole $\vee$ signifie « ou » (et $\wedge$ signifie « et »). Le résultat est donc $1 \leqslant x$ ou $x \leqslant -1$. Soit $x \in]-\infty, -1] \cup [1, +\infty[$.

12 Etude d'une fonction

On peut facilement définir une fonction (mathématique) comme une fonction **Python** standard :

```
def f(x):
    return 2*x**2-3
```

On peut ensuite utiliser les notations mathématiques standards :

```
f(3)
```

15

```
f(x+1)
```

$2(x+1)^2 - 3$

Les limites sont obtenues avec l'instruction **limit** :

```
limit(f(x),x,0)
```

-3

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

∞

Noter l'utilisation de « oo » (deux « o » minuscules) pour représenter le symbole ∞.

Les limites à droite et à gauche s'obtiennent en rajoutant un argument `'+'` ou `'-'` à la limites :

```
limit(1/x,x,0,'-')
```

$-\infty$

L'instruction **diff()** permet de calculer la dérivée d'une fonction :

```
diff(f(x),x)
```

$4x$

Les dérivées successives s'obtiennent en précisant l'ordre la dérivée :

```
diff(f(x),x,2)
```

4

On peut tester la convexité d'une fonction sur $\mathbb{R}$ ou un intervalle avec **is_convex()** :

```
is_convex(x**2,x)
```

True

```
is_convex(x**3,x)
```

False

```
is_convex(x**3,x,
    domain = Interval(0,oo))
```

True

L'instruction **integrate()** permet de calculer des primitives ou des intégrales :

```
integrate(f(x),x)
```

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

```
integrate(f(x),(x,0,1))
```

$-\frac{7}{3}$

Pour obtenir le développement limité d'une fonction $f(x)$ à l'ordre n au voisinage de x_0, on utilise l'instruction **series(f(x),x,x0,n+1)** :

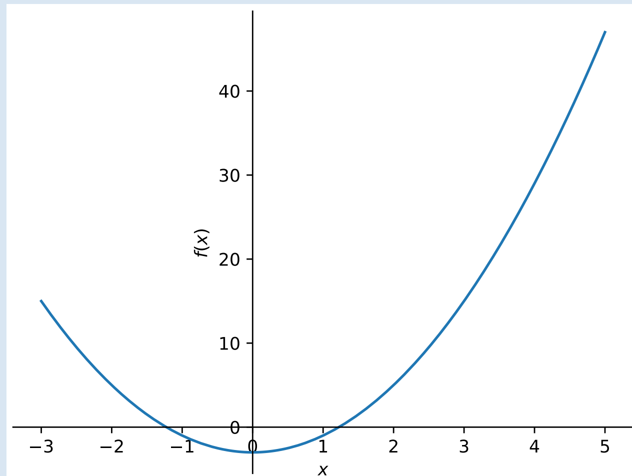
```
series(exp(x),x,0,4)
```

$1 + x + \frac{x^2}{2} + \frac{x^3}{6} + O(x^4)$

13 Représentations graphiques

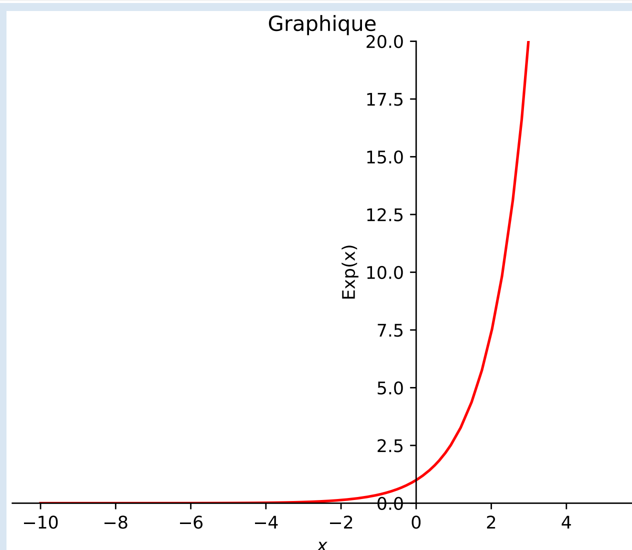
L'instruction **plot()** permet d'obtenir une représentation graphique d'une fonction sur un intervalle :

```
plot(f(x),(x,-3,5))
```



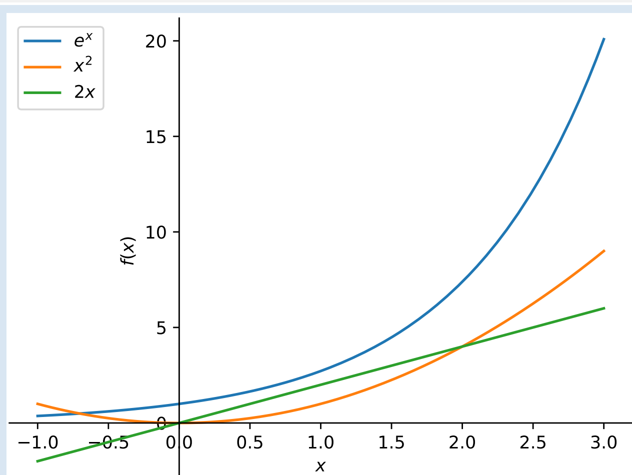
On peut aussi préciser les valeurs extrêmes de l'axe y et même la couleur de la courbe :

```
plot(exp(x),(x,-10,5),ylim=(0,20),
line_color='red',ylabel='Exp(x)',
title="Graphique")
```



Il est possible de représenter plusieurs fonctions sur le même graphique :

```
plot(exp(x),x**2,2*x,(x,-1,3),
legend=True)
```



14 Fonctions de plusieurs variables

Comme pour les fonctions d'une variable, on utilise une fonction **Python** pour définir une fonction mathématique de plusieurs variables :

```
def f(x,y):
    return x*y-x**2
```

On utilise alors la fonction avec les notations naturelles :

```
f(2,3)
```

2

```
f(x,2*x)
```

x^2

Les dérivées partielles premières et secondes s'obtiennent avec :

```
diff(f(x,y),x)
```

$-2x+y$

```
diff(f(x,y),y)
```

x

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

-2

```
diff(f(x,y),x,y)
```

1

Le gradient s'obtient de manière *indirecte* :

```
Matrix(derive_by_array(f(x,y),[x,y]))
```

$$\begin{bmatrix} -2x+y \\ x \end{bmatrix}$$

C'est plus simple pour la matrice hessienne et le hessien :

```
hessian(f(x,y),(x,y))
```

$$\begin{bmatrix} -2 & 1 \\ 1 & 0 \end{bmatrix}$$

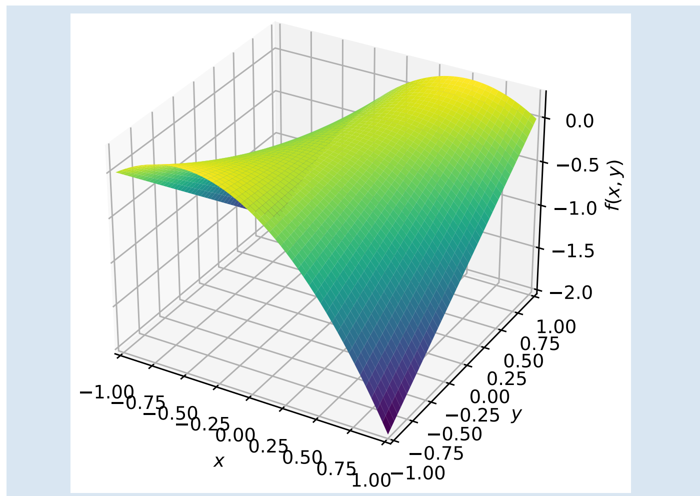
```
det( hessian(f(x,y),(x,y)) )
```

-1

Il est possible de représenter f graphiquement en 3 dimensions. Pour cela, il est nécessaire d'importer la fonction `plot3d()` :

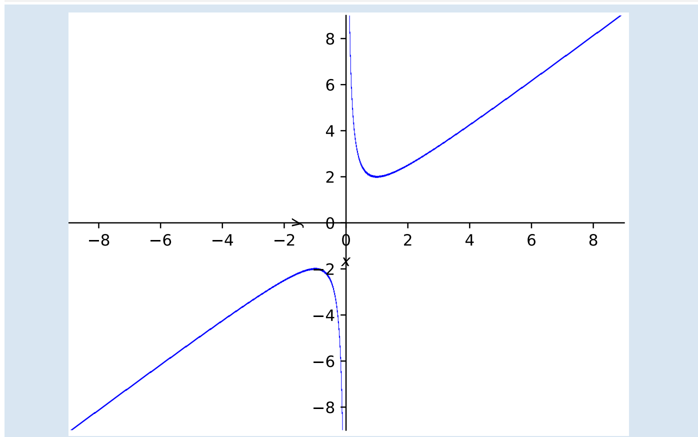
```
from sympy.plotting import plot3d
```

```
plot3d(f(x,y),(x,-1,1),(y,-1,1))
```



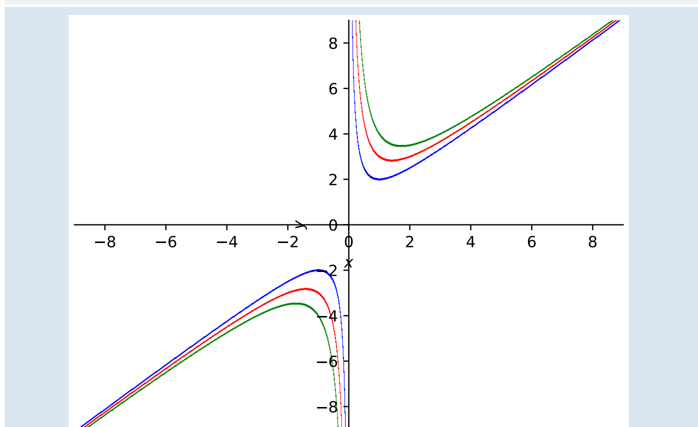
On peut également tracer les courbes de niveau :

```
plot_implicit(f(x,y)-1, (x,-9,9),
(y,-9,9) )
```



Pour représenter plusieurs lignes de niveau, il faut travailler un peu plus :

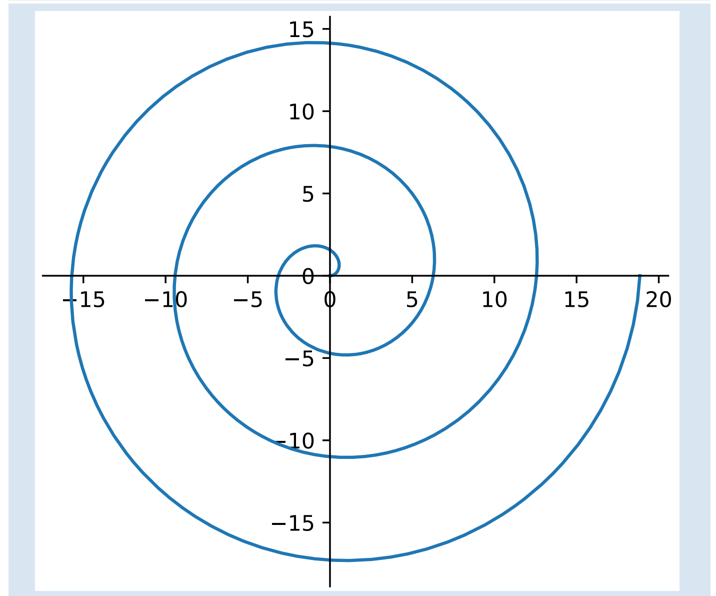
```
plot1 = plot_implicit(f(x,y)-1,
  (x,-9,9), (y,-9,9), show=False)
plot2 = plot_implicit(f(x,y)-2,
  (x,-9,9),(y,-9,9),
  line_color='red', show=False)
plot3 = plot_implicit(f(x,y)-3,
  (x,-9,9),(y,-9,9),
  line_color='green', show=False)
plot1.append(plot2[0])
plot1.append(plot3[0])
plot1.show()
```



15 Courbes paramétrées

L'instruction **plot_parametric** permet de représenter facilement des courbes paramétrées :

```
t = symbols('t')
plot_parametric((t*cos(t),t*sin(t)),
    (t,0,6*pi), aspect_ratio=(1,1))
```



16 Suites et séries

Une suite $(u_n)_n$ se définit de la même façon qu'une fonction :

```
def u(n):
    return 1/n**2
```

On peut alors calculer les termes successifs de la suite et déterminer sa limite :

```
u(2)
```

0.25

```
limit(u(n),n=oo)
```

0

Pour représenter graphiquement une suite (les couples (n, u_n)), il est nécessaire de charger la bibliothèque **matplotlib** :

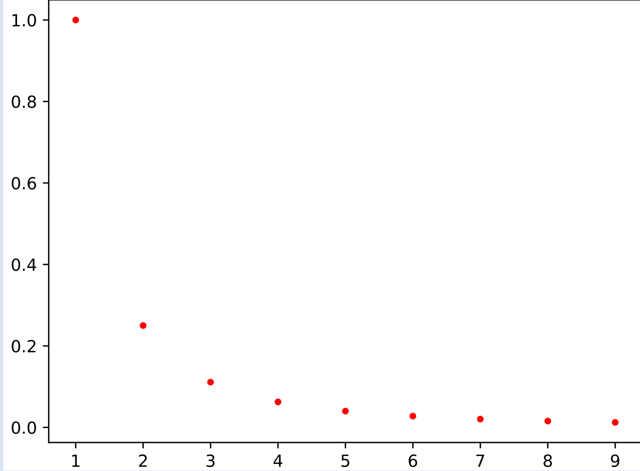
```
import matplotlib.pyplot as plt
```

On utilise ensuite une des deux méthodes suivantes :

```
for k in range(1,10):
    plt.plot(n,u(k),'r.')
```

ou

```
plt.plot(range(1,10),
[u(k) for k in range(1,10)],'r.')
```



La somme de la série (Σu_n) s'obtient avec **summation()** :

```
summation(u(n),(n,1,oo))
```

$$\frac{\pi^2}{6}$$

17 Equations récurrentes linéaires

Pour résoudre l'équation $u_{n+1} = 3u_n + n^2$, on commence par définir la suite inconnue (u_n) avec l'instruction **Function()** (attention au « F » majuscule) :

```
u = Function('u')
```

On peut alors résoudre l'équation en l'écrivant sous la forme $u_{n+1} - 3u_n - n^2$ (sans le $= 0$) :

```
rsolve(u(n+1)-3*u(n)-n**2, u(n))
```

$$3^n C_0 - \frac{n^2}{2} - \frac{n}{2} - \frac{1}{2}$$

Des conditions initiales peuvent être indiquées, en utilisant la syntaxe particulière **{u(n0):u0,u(n1):u1,…}** :

```
rsolve(u(n+1)-3*u(n)-n**2, u(n),
{u(0):2})
```

$$\frac{5 \cdot 3^n}{2} - \frac{n^2}{2} - \frac{n}{2} - \frac{1}{2}$$

18 Equations différentielles linéaires

La syntaxe pour résoudre une équation différentielle est très proche de celle des équations récurrentes.

Pour résoudre l'équation $x'(t) - tx(t) = t$, on définit d'abord la variable et la fonction inconnue :

```
t = symbols('t')
x = Function('x')
```

puis on utilise l'instruction **dsolve()** :

```
dsolve(diff(x(t),t)-t*x(t)-t,x(t))
```

$$x(t) = C_1 e^{\frac{t^2}{2}} - 1$$

La condition initiale $x(0) = 1$ est précisée via l'option `ics` :

```
dsolve(diff(x(t),t)-t*x(t)-t,x(t),
ics={x(0):1})
```

$$x(t) = 2e^{\frac{t^2}{2}} - 1$$

On résout de même les équations différentielles d'ordre 2, par exemple $x''(t) + 2x'(t) + x(t) = 1 + t, x(0) = 1, x'(0) = 2$:

```
dsolve(diff(x(t),t,2)+2*diff(x(t),t)
+x(t)-(1+t),x(t), ics={x(0):1,
diff(x(t),t).subs(t,0):2})
```

$$x(t) = t + (3t + 2)e^{-t} - 1$$

19 Vecteurs et matrices

Un vecteur est défini comme une matrice à une dimension de la façon suivante :

```
X = Matrix([1,2,3]); X
```

$$\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

On peut alors calculer sa norme avec **norm()** :

```
X.norm()
```

$$\sqrt{14}$$

Le calculs basiques se font assez naturellement :

```
Y = Matrix([4,5,6]); X + 2*Y
```

$$\begin{bmatrix} 9 \\ 12 \\ 15 \end{bmatrix}$$

Le produit scalaire utilise une syntaxe un peu particulière :

```
X.dot(Y)
```

32

Plus généralement, une matrice se définit par :

```
A = Matrix([[1,2,3],[4,5,6]]); A
```

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

Les instructions **eye(n)** et **diag()** permettent d'obtenir la matrice identité d'ordre n et une matrice diagonale quelconque :

```
eye(3)
```

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

```
diag(3,4,5)
```

$$\begin{bmatrix}3 & 0 & 0\\0 & 4 & 0\\0 & 0 & 5\end{bmatrix}$$

La transposée est obtenue par **tranpose()** et le déterminant (d'une matrice carrée) par **det()** :

```
transpose(A)
```

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

```
B = Matrix([[2,5],[1,3]]);B
```

$$\begin{bmatrix}2 & 5\\1 & 3\end{bmatrix}$$

```
det(B)
```

1

Evidemment, le produit de deux matrices est obtenu avec la multiplication standard **`*`** :

```
B*A
```

$$\begin{bmatrix}22 & 29 & 36\\13 & 17 & 21\end{bmatrix}$$

De même, l'inverse d'une matrice (inversible !) est obtenu en la mettant à la puissance -1 :

```
B**(-1)
```

$$\begin{bmatrix}3 & -5\\-1 & 2\end{bmatrix}$$

Pour diagonaliser une matrice

```
A = Matrix([[2,0,1],[1,3,-
1],[0,0,3]]);A
```

$$\begin{bmatrix}2 & 0 & 1\\1 & 3 & -1\\0 & 0 & 3\end{bmatrix}$$

on peut calculer le polynôme caractéristique :

```
lamda = symbols('lamda', real=True)
A.charpoly(lamda).as_expr()
```

$\lambda^3 - 8\lambda^2 + 21\lambda - 18$

puis les valeurs propres :

```
A.eigenvals()
```

$\{3 : 2, 2 : 1\}$

et les vecteurs propres :

```
A.eigenvects()
```

```
[(2,1,[Matrix([[-1],[1],[0]])]),
(3,2,[Matrix([[0],[1],[0]]),
Matrix([[1],[0],[1]])])]
```

ou directement en utilisant l'instruction **diagonalize()** :

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

```
D
```

$$\begin{bmatrix}2 & 0 & 0\\0 & 3 & 0\\0 & 0 & 3\end{bmatrix}$$

```
P
```

$$\begin{bmatrix}-1 & 0 & 1\\1 & 1 & 0\\0 & 0 & 1\end{bmatrix}$$

La puissance n-ième de la matrice s'obtient alors facilement :

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

$$\begin{bmatrix}2^n & 0 & -2^n + 3^n\\-2^n + 3^n & 3^n & 2^n - 3^n\\0 & 0 & 3^n\end{bmatrix}$$

Mais on aurait pu juste utiliser

```
A**n
```

20 Nombres complexes

Un nombre complexe est noté avec la notation naturelle (mais en utilisant un « I » majuscule).

```
z = 3+4*I; z
```

$3 + 4i$

Les instructions **re()**, **im()**, **abs()**, **arg()** renvoient, respectivement, la partie réelle et imaginaire, le module et l'argument du nombre complexe.

```
abs(z)
```

```
5
```

Le conjugué d'un nombre est obtenu avec l'instruction **conjugate()** :

```
conjugate(z)
```

$3 - 4i$

21 Fonctions (encore)

Pour définir une fonction mathématique f, outre l'utilisation d'une fonction **Python** vue plus haut, il est possible d'utiliser l'instruction **lambdify()** de la manière suivante :

```
f = lambdify(x,1+x**3)
```

```
f(2)
```

9

```
diff(f(x),x)
```

$3x^2$

Cela est aussi possible avec les fonctions de plusieurs variables :

```
f = lambdify((x,y),1+x*y)
```

```
f(3,4)
```

13

Parfois, il est juste nécessaire de nommer une expression, sans en faire véritablement une fonction :

```
f=3*x**2
```

On utilise alors la propriété `subs()` pour donner des valeurs à la variable :

```
f.subs(x,2)
```

12

Cela, y compris pour des expressions de plusieurs variables :

```
f = x*y
```

```
f.subs([(x,3),(y,4)])
```

12

22 Exemple : Optimisation 1 variable

On souhaite optimiser la fonction $f(x) = x^3 - 3x$ sur $\mathbb{R}$. On commence par définir la fonction :

```
def f(x):
    return x**3 - 3*x
```

Les limites en $\pm\infty$ indiquent que f n'admet pas d'extremum global :

```
limit(f(x),x,+oo)
```

∞

```
limit(f(x),x,-oo)
```

$-\infty$

En utilisant la dérivée de f :

```
diff(f(x),x)
```

$3x^2 - 2$

on détermine les points critiques (candidats) :

```
solve(diff(f(x),x),x)
```

$[-1, \ 1]$

A l'aide de la dérivée seconde de f :

```
diff(f(x),x,2)
```

$6x$

on détermine la nature des points critiques :

```
diff(f(x),x,2).subs(x,-1)
```

-6

```
diff(f(x),x,2).subs(x,1)
```

6

La fonction f admet donc un maximum local en $x = -1$ et un minimum local en $x = +1$. Ces extremums sont

```
f(-1)
```

2

```
f(1)
```

-2

Alternative pour les conditions suffisantes :

```
candidats = solve(diff(f(x),x),x)
```

```
for x0 in candidats:
  print("f''(", x0, ") = ",
    diff(f(x),x,2).subs(x,x0))
```

```
f''( -1 ) = -6
f''( 1 ) = 6
```

Finalement, on peut faire une représentation graphique pour illustrer le résultat :

```
plot(f(x),(x,-2.5,2.5))
```