Mémento Python au service des mathématiques

Ce document ne prétend ni à l'exhaustivité ni à une parfaite consistance. Il ne se substitue pas aux documentations officielles Python, scipy/numpy ou sympy. Il s'agit seulement d'un aide-mémoire facilitant l'utilisation de Python comme outil pour la pratique des mathématiques. Le mode interactif de Python permet si nécessaire d'avoir des informations plus précises : type(*objet*) # donne de type de l'*objet*
dir(*objet_ou_fonction*), help(*objet_ou_fonction*), *objet_ou_fonction* ? # affiche les méthodes et attributs pouvant concerner l'*objet ou la fonction*.

Calculs dans $\mathbb{Z}$, $\mathbb{R}$, $\mathbb{Q}$, $\mathbb{C}$: types integer, float, Rational, complex.

	import numpy as np	from sympy import *
5//3 # quotient de la division euclidienne de 5 par 3 5%3 # reste de la division euclidienne de 5 par 3 int(-1.9) # troncature entière int(*booléen*) #fct caractéristique True $\mapsto$ 1 et False $\mapsto$ 0 round(1/6,4) # float arrondi du réel 1/6 à 10^{-4} près 1.2**3.4 # $(1,2)^{3,4}=e^{3,4\ln(1,2)}$ car $1,2>0$	np.pi # float approximation de π np.sin, np.cos, np.tan # fcts pouvant s'appliquer à une liste np.arcin, np.arccos, np.arctan np.sqrt, np.log, np.exp # peuvent s'appliquer à un complexe	floor(-1,9) # fonction partie entière Rational(1/2+3/4) # opérations dans $\mathbb{Q}$ numer(Rational(5/4)) # numérateur denom(Rational(5/4)) # dénominateur sin, cos, tan, asin, acos, atan, factorial binomial(3,2) # 2 parmi 3 Piecewise((*expr1*,*cond1*),(*expr2*,*con2*),...) # expression définie par morceaux

2+3j # nombre complexe $2+3i$
1j # nombre complexe i
(2+3j).real # partie réelle
(2+3j).imag # partie imaginaire
(2+3j).conjugate() # conjugué
abs(2+3j) # module
(1+1j)**(1/3) # donne UNE racine cubique de $1+i$
	# idem mais pouvant s'appliquer à une liste
	np.real()
	np.imag()
	np.conj()
	np.absolute()
	np.angle(2+3j) # approximation d'un argument de $2+3i$
		2+3*I # nombre complexe $2+3i$
		I # nombre complexe i
		re(2+3*I) # partie réelle
		im(2+3*I) # partie imaginaire
		conjugate(2+3*I) # conjugué
		abs(2+3*I) # module
		arg(2+3*I) # argument

Les booléens : True, False

Comparaisons : == ,!=, >, <, <=, >=, in	Opérations : and, or, not

Les chaînes de caractères et le calcul symbolique	from sympy import *

str(123)# conversion de l'integer 123 en chaîne de caractères '123'
'ABC'+'123' # concaténation de deux chaînes de caractères
3*'ABC' # concaténation donnant 'ABCABCABC'
ord('A') # code ascii du caractère A
chr(65) # chaîne constituée du caractère dont le code ascii est 65
chaîne.split() # liste des mots (séparés par des espaces dans *chaîne*)
chaîne.join(*liste_de_chaînes*) # concaténation de toutes les chaînes de caractères
chaîne.upper(), *chaîne*.lower() # conversion d'un texte en majuscule ou minuscules
chaîne.replace(*old*,*new*) # remplace les caractères dans une chaîne
eval(*chaîne*) # interprète la chaîne de caractères comme du code Python
	x,y,z=symbols('x y z') # x , y et z désignent alors des inconnues ou des paramètres
	f=2*x+3 # f est une expression, pas une fonction Python
	f.subs(x,4) # renvoie la valeur de f quand x vaut 4
	simplify(*expression*) # simplifie (si possible) l'expression
	E=Eq(*membre de gauche*,*membre de droite*) # E est une équation
	E.lhs , E.rhs #renvoie left hand side ou right hand side de l'équation E
	expression1.equals(*expression2*) # booléen
	pprint(*expression*) # pretty print, affichage amélioré d'une expression

Tableaux à une dimension : liste ou, pour les calculs dans $\mathbb{R}^n$, $\mathbb{C}^n$ (voire $\mathbb{R}^{\mathbb{N}}$ et $\mathbb{C}^{\mathbb{N}}$) : array et Matrix

	import numpy as np # le type array permet de manipuler des tableaux de nombres dont le cardinal est fixé lors de sa création	from sympy import * x,n,k = symbols('x n k')

list(...) # conversion en liste
list(range(*fin_exclue*)) # entiers POSITIFS OU NULS
⚠ list(range(0))=[]
list(range(*début* : *fin_exclue*)) # liste d'entiers consécutifs
list(range(*début* : *fin_exclue* : *pas*) # liste d'entiers
[f(i) for i in *liste*] # création de liste en compréhension
[f(i) for i in *liste* if g(i)] # g(i) est un booléen
[1,2]+[3,4] # concaténation
3*[1,2] # concaténation de 3 fois la même liste
	np.array([1,2,3]) # matrice ligne
	np.array([[1],[2],[3]]) # matrice colonne
	np.zeros(3),np.ones(3)# matrice ligne de trois 0 ou trois 1
	np.linspace(*début*,*fin_incluse*,*nombre d'éléments*)
	np.arange(*début*,*fin_incluse*,*pas*)
	np.array([1,2])+np.array([3,4]) # somme dans $M_{1,2}(\mathbb{R})$
	3*np.array([1,2]) # multiplication externe dans $M_{1,2}(\mathbb{R})$
		Matrix([1,2,3])# ⚠matrice colonne cf Matrix([[1],[2],[3]])
		Matrix([[1,2,3]]) # matrice ligne
		zeros(1,3), ones(1,3) # matrice ligne de trois 0 ou trois 1
		Matrix([[1,2]])+Matrix([[3,4]]) # somme dans $M_{1,2}(\mathbb{R})$
		3*Matrix([[1,2]]) # multiplication externe dans $M_{1,2}(\mathbb{R})$

len(L) # nombre d'éléments de L
max(L), min(L) # minimum et maximum de L
sum(L) # somme des éléments de L
élément in L # booléen égal à True ssi *élément* est dans L
map(*liste*,*fonction*) # liste des images des éléments de *liste* par *fonction*
set(L) # ensemble des valeurs de L (sans doublon)
	np.array([3,4])**2 # array des carrés des termes
	1/np.array([3,4]) # array des inverses des termes
	np.linalg.norm(...) # norme eucl. canonique
		v.norm() # norme eucl. canonique
		pprint(summation(k*x**k,(k,1,n))) # somme partielle
		pprint(summation(k*x**k,(k,1,oo))) #somme série entière

L[0] # premier terme de la liste L L[i] # terme d'index i la liste L L[-1] # dernier terme de la liste L L[*début* : *fin(exclue)* : *pas*] # sous-liste selon les index L[*début*:],L[:*fin(exclue)*],L[:],L[::*pas*] ⚠ si k ⩾ len(L) alors L[k:]=[] ⚠ si k<1 alors L[:k]=[] L[i],L[j]=L[j],L[i] #permutation de deux éléments	idem pour array	idem pour Matrix
L.count(*élément*) #nombre d'occurrence d'*élément* dans L L.sort() # modifie L en triant ses éléments L.pop() #enlève le dernier terme L et renvoie sa valeur L.pop(i) #enlève le terme d'index i de L et renvoie sa valeur L.append(*élément*) # ajoute l'*élément* à la fin de L L.insert(i,*élément*) # insert *élément* à l'index i dans L et décale les autres L.index(*élément*) # index de la première occurrence de *élément* dans la liste L	np.vdot([1,2],[3,4]) #produit scalaire canonique np.cross([1,2,3],[4,5,6]) # produit vectoriel np.transpose([1,2]) #matrice colonne	v1.dot(v2) # produit scalaire de matrices colonnes transpose(v1)*v2 # matrice ligne × matrice colonne v1.cross(v2) # produit vectoriel de matrices colonnes

Tableaux à 2 dimensions : liste de listes ou, pour des calculs dans $\mathrm{M}_{n,p}(\mathbb{R})$ ou $\mathrm{M}_{n,p}(\mathbb{C})$, array et Matrix :
Une matrice $\left(a_{i,j}\right)_{\substack{1\leqslant i\leqslant n\\1\leqslant j\leqslant p}}\in\mathrm{M}_{n,p}(\mathrm{K})$ est définie par la liste de ses n lignes (chacune des lignes étant une liste de p nombres) : ⚠ les index *ligne_i* et *colonne_j* commencent à 0.

	import numpy as np	from sympy import *

M=[[a(i+1,j+1) for j in range(p)] for i in range(n)]#liste des lignes
len(M) # nombre de lignes
len(M[0]) # nombre de colonnes
	np.array([[1,2,3],[4,5,6]]) # 2 lignes, 3 colonnes
	np.zeros((*nb_ de_lignes*,*nb_de_colonnes*)) # ⚠ couple
	np.ones((*nb_ de_lignes*,*nb_de_colonnes*)) # ⚠ couple
	np.eye(*taille*) # matrice identité
	np.diag([1,2,3]) # matrice diagonale
	np.concatenate((u,v,w),axis=1) # crée la matrice dont les colonnes sont les matrices colonnes u, v, w
		Matrix([[1,2,3],[4,5,6]]) # 2 lignes, 3 colonnes
		zeros(*nb_ de_lignes*,*nb_de_colonnes*)
		ones(*nb_ de_lignes*,*nb_de_colonnes*)
		zeros(3), ones(3) # matrices carrées
		eye(3)
		diag(1,2,3) # fonctionne aussi par blocs
		u.row_join(v).row_join(w) # crée la matrice dont les colonnes sont les matrices colonnes u, v, w

M[*ligne_i*] # renvoie la liste indexée par *ligne_i* M[*ligne_i*][*colonne_j*] # terme (M[*ligne_i*])[*colonne_j*] [[M[i][*colonne_j*]] for i in range(n)]#liste des lignes de *colonne_j*	M[*ligne_i*,*colonne_j*] # terme $a_{i+1;\,j+1}$ M[*ligne_i*] # array 1D M[*ligne_i*:*ligne_i*+1, :] # array 2D M[:,*colonne_j*] # array 1D M[:,*colonne_j*:*colonne_j*+1] # array 2D	M[*ligne_i*,*colonne_j*] # terme $a_{i+1;\,j+1}$ M[*ligne_i* × p+*colonne_j*] # terme $a_{i+1;\,j+1}$ M[*ligne_i*,:] # matrice ligne M[:;*colonne_j*] # matrice colonne
# les opérations dans $\mathrm{M}_{n,p}(\mathrm{K})$ ne sont pas obtenues par manipulations simples sur les listes de listes.	3*M+N # combinaisons linéaires de matrices M*N # ⚠ produit des termes 2 à 2 ≠ M×N np.dot(M,N) # produit matriciel M×N M.dot(N) # produit matriciel M×N M**3 # ⚠ cube de chaque terme ≠ M^3 np.linalg.matrix_power(M,3) # M^3 si M est carrée np.linalg.inv(M) # M^{-1} si M est inversible	3*M+N # combinaisons linéaires de matrices M*N # produit matriciel M×N M**3 # M^3 si M est carrée M**(-1) # M^{-1} si M est inversible
	np.transpose(M),np.trace(M), np.linalg.det(M), np.linalg.norm(M) np.poly(M) #polynôme caractéristique cf Polynomial np.linalg.eigvals(M) # spectre dans $\mathbb{C}$ np.linalg.eig(M) # couple : spectre, matrice de passage	transpose(M), trace(M), det(M), M.norm() M.charpoly() #polynôme caractéristique M.eigenvals() # val propres : ordre de multiplicité M.eigenvects() # val. propres, ordre, base des sous-espaces propres M.rref() # reduced row echelon form M.nullspace() # base du noyau d'une matrice M.columnspace() # base de l'image d'une matrice M.rank() # rang d'une matrice P,J=M.jordan_form() # $\mathrm{M}=\mathrm{P}\,\mathrm{J}\,\mathrm{P}^{-1}$ avec J matrice triangulaire voire diagonale si possible

Calculs dans $\mathbb{R}[X]$ ou $\mathbb{C}[X]$: évaluation, coefficients, racines, dérivation, intégration, division euclidienne

from numpy.polynomial import Polynomial	from sympy import * X=symbols('X')

P=Polynomial([-3,2,0,1]) # P est le polynôme X^3+2X-3
P(1+2j) # évaluation de $P(1+2i)$
P.coef # renvoie array([-3,2,0,1])
P.degree()
P.roots() # racines complexes
	P=X**3-2*X-3 # P est une expression
	P.subs(X,1+2*I) # évaluation de $P(1+2i)$
	expand(P) # expression de P dans la base canonique de K[X]
	degree(poly(P))
	poly(P).all_coeffs() # liste des coeffcients
	solve(P) # racines complexes de P

P.deriv() # polynôme dérivé
P.deriv(2) # dérivée seconde du polynôme P
P.integ()# primitive de P s'annulant en 0
Q=P.integ(*n*,[y_{n-1} ,..., y_0]) # $Q^{(n)}=P$, $Q(0)=y_0$,...., $Q^{(n-1)}(0)=y_{n-1}$
	diff(P,X) # dérivée de P
	diff(P,X,X) # dérivée seconde de P
	integrate(P) # primitive de P s'annulant en 0
	integrate(P,(X,2,X)) # primitive de P s'annulant en 2

Q=P//D # quotient dans la division euclidienne du polynôme P par le polynôme D
R=P%D # R est le reste dans la division euclidienne du polynôme P par le polynôme D
	poly(P).div(poly(D))# couple de Poly (quotient,reste) dans la division euclidienne du polynôme P par le polynôme D

Fonctions scalaires d'une variable réelle : représentation graphique, équation numérique, dérivation,intégration développements limités, séries entières, séries de Fourier

def f(x) : … return ...	from sympy import * x=symbols('x') f=... # f est une expression utilisant le symbole x

X=... # liste ou array des valeurs prises par x
Y=[f(x) for x in X] # liste des images
import matplotlib.pyplot as plt
plt.plot(X,Y,color='r',linestyle=':',marker='o')# plot(*liste_abscisses*,*liste_ordonnées*)
plt.axis('equal') # repère orthonomal
plt.axis(*x_min*,*x_max*,*y_min*,*y_max*) # bornes du repère
plt.show() #affichage de la courbe
	plot(f,(x,*x_min*,*x_max*)) # courbe représentative de f sur [*x_min*;*x_max*]
	
	plot(f,(x,*x_min*,*x_max*),ylim=(*y_min*,*y_max*))

from scipy.otpimize import fsolve fsolve(f, x_0) # approximation d'une solution de $f(x)=0$ par itérations avec valeur initiale x_0 . ⚠risque de ne pas converger ou de converger vers une autre racine que celle la plus proche de x_0 . Dans le doute une dichotomie est préférable.	solve(f) # solutions formelles de l'équation $f(x)=0$

[(Y[i+1]-Y[i])/(X[i+1]-X[i]) for i in range(len(X)-1)] # liste des taux d'accroissements	diff(f,x) # dérivée formelle de f par rapport à x
	series(f,x) # développement limité de f au voisinage de 0
	h=symbols('h')
	f.subs(x,2+h).series(h) # développement limité $x\to 2$ i.e $h\to 0$

from scipy.integrate import quad	integrate(f,x) # primitive de f s'annulant en 0

quad(f,0,1) # couple (valeur approchée de $\int_0^1 f(t)\,dt$, majoration de l'erreur)
quad(f,0,np.inf) # couple (valeur approchée de $\int_0^{+\infty} f(t)\,dt$, majoration de l'erreur)
	integrate(f,(x,0,1)) # valeur exacte de $\int_0^1 f(t)\,dt$
	integrate(f,(x,0,oo)) # valeur exacte de $\int_0^{+\infty} f(t)\,dt$

[sum(a_n *x**n for n in range(10)] for x in X] # somme partielle $\sum_{n=0}^{9} a_n x^n$ import numpy as np [a_0 +sum(a_n *np.cos(n* ω *x)+ b_n *np.sin(n* ω *x)] for n in range(1,10)] for x in X] # somme partielle de la série de Fourier	n=symbols('n') summation(a_n *x**n,(n,0,oo)) # somme de la série entière $\sum_{n\geqslant 0} a_n x^n$ a_0 +summation(a_n *cos(n* ω *x)+ b_n *sin(n* ω *x),(n,1,oo)) # somme de la série de Fourier

Fonctions vectorielles d'une variable réelle : courbe paramétrée (2D ou 3D), longueur d'une courbe paramétrée ; vecteur vitesse, point stationnaires, développements limités

def f(t) :
  …
  return (…,...) # couple pour la 2D, triplet pour la 3D
	from sympy import *
	t=symbols('t')
	x,y=... ;... # couple de deux expressions utilisant le symbole t

T=... # liste des valeurs prises par t
X=[f(t)[0] for t in T] # la première composante de $f(t)$ est une abscisse
Y=[f(t)[1] for t in T] # la seconde composante de $f(t)$ est une ordonnée
import matplotlib.pyplot as plt
plt.plot(X,Y) # ligne brisée passant par les points (x(t);y(t))
plt.show()
	plotting.plot_parametric(x,y,(t, t_0 , t_1)) # arc décrit par $f(t)$ pour $t\in[t_0;t_1]$

import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
ax=Axes3D(plt.figure())
ax.plot(X,Y,Z) # listes des abscisses, des ordonnées et des cotes
plt.show()
	plotting.plot3d_parametric_line(x,y,z)

sum([np.sqrt((X[i+1]-X[i])**2+(Y[i+1]-Y[i])**2) for i in range(len(T)-1)] # approximation de la longueur de la courbe paramétrée par sommation de longueur de segments	integrate(sqrt(diff(x,t)**2+diff(y,t)**2),(t, t_0 , t_1)) # longueur exacte de l'arc décrit par $f(t)$ pour $t\in[t_0;t_1]$

	Matrix([diff(x,t),diff(y,t)].subs(t,2)) # vecteur vitesse en t=2
	solve([diff(x,t),diff(y,t)],t) # recherche de points stationnaires
	Matrix([x.series(t),y.series(t)]) # développement limité au voisinage de 0
	h=symbols('h')
	Matrix([x.subs(t,2+h).series(h),y.subs(2+h).series(h)]) # DL $t\to 2$ i.e. $h\to 0$

Fonctions scalaires de plusieurs variables : courbe d'équation $f(x;y)=0$, surface d'équation $z=f(x;y)$, extremum, dérivées partielles

def f(x,y) :
  …
  return ... # float
import numpy as np
X = … # liste ou array des valeurs de x
Y = … # liste ou array des valeurs de y
Z = [[f(x,y) for x in X] for y in Y]
import matplotlib.pyplot as plt
	from sympy import *
	x,y =symbols('x y')
	f=... # expression dépendant de x et y

plt.contour(X,Y,Z,0) # courbe d'équation $f(x,y)=0$
plt.contour(X,Y,Z,[z_1 , z_2 , z_3]) # courbes d'équations $f(x,y)=z_1$ etc...
plt.show()
	plot_implicit(f,(x, x_{min} , x_{max}),(y, y_{min} , y_{max})) # courbe d'équation $f(x,y)=0$
	diff(f,x) # expression de $\frac{\partial f}{\partial x}(x;y)$
	Matrix([diff(f,x).subs(x,a).subs(y,b),diff(f,y).subs(x,a).subs(y,b)]) # $\nabla f(a;b)$

min([min(Z[i]) for i in range(len(Z))]) # recherche du minimum
X, Y = np.meshgrid(X, Y) # création du maillage
ax=Axes3D(plt.figure())
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
ax.plot_surface(X,Y,Z) # maillage abscisses × ordonnées et cotes
plt.show()
	plotting.plot3d(f,(x, x_{min} , x_{max}),(y, y_{min} , y_{max}),xlabel='x',ylabel='y',zlabel='z') # surface d'équation $z=f(x;y)$
	
	solve([diff(f,x),diff(f,y)],[x,y]) # recherche de point critique

Fonctions vectorielles de plusieurs variables : système

def f(v) : # v étant une liste ou un array , v[0], v[1] et v[2] correspondent à x, y et z
  …
  return … , ... # tuple de float
	from sympy import *
	x,y,z=symbols('x y z')
	f= (… , … , …) # tuple ou liste d'expressions dépendant de x, y et z

from scipy.optimize import root sol=root(f,[1,2,3]) # méthode itérative initialisée au point (1,2,3) sol.success # booléen pour la convergence de la méthode sol.x # dernier point lors des itérations	solve(f,[x,y,z]) # solutions formelles de l'équation vectorielle $f(x;y;z)=0_{\mathbb{R}^n}$

Équations différentielles scalaires du premier ordre, système d'équations différentielles, application aux équations différentielles scalaires linéaire homogène d'ordre n

from scipy.integrate import odeint X=... # liste ou array décrivant l'intervalle de résolution	from sympy import *

def f(y,x) :
  ...
  return … # pour résoudre $y'(x)=f(y,x)$
	x=symbols('x')
	y=Function('y')
	eq=... # expression utilisant y(x) et diff(y,x)

Ly=odeint(f, y_0 ,X) # approximation des y(X[i]) pour la solution vérifiant la condition initiale y(X[0])= y_0	dsolve(eq,y(x)) # ensemble des solutions de eq=0, ⚠l'intervalle de résolution peut poser problème

def f(Y,x) :
…
return np.array([... ,...]) # système différentiel $\begin{pmatrix} Y'[0] \\ \vdots \\ Y'[n-1] \end{pmatrix}=\begin{pmatrix} f(Y,x)[0] \\ \vdots \\ f(Y,x)[n-1] \end{pmatrix}$
LY=odeint(f,np.array[y_0 ,... , y_{n-1}],X) # approximation des arrays Y(X[i]) pour la solution vérifiant la condition initiale Y(X[0])=(y_0 ,... , y_{n-1})
	t=symbols('t')
	x, y = symbols('x y',cls=Function)
	eq1 = …# expression utilisant x(t),y(t),diff(x,t) et diff(y,t)
	eq2= …
	dsolve((eq1,eq2)) # ensemble des solutions du système différentiel $\begin{cases} eq1=0 \\ eq2=0 \end{cases}$

def f(Y,t) : # ici $Y=\left[y, y', \ldots, y^{(n-1)}\right]$ et on veut résoudre Y'=AY
n=... #odre de l'équation différentielle linéaire
A=np.zeros([n,n])
for i in range(n-1):
A[i,i+1]=1
A[n-1,:]=[a_0 ,..., a_{n-1}] # pour résoudre $y^{(n)}=a_0 y+\ldots+a_{n-1} y^{(n-1)}$
return A.np.dot(Y)
Ly=odeint(f,np.array[y_0 ,... , y_{n-1}],X)[0] # première composante de Y, approximation des $y(X[i])$ pour la solution vérifiant les conditions initiales $y^{(k)}(X[0])=y_k$
	x=symbols('x')
	y=Function('y')
	eq=... # expression utilisant y(x), diff(y,x), diff(y,x,x),...
	dsolve(eq,y(x)) # ensemble des solutions de eq=0, ⚠l'intervalle de résolution peut poser problème

Simulation d'expériences aléatoires : une variable aléatoire X, couple de variables aléatoires (X,Y)

import numpy.random as rd	

rd.randint(1,7) # variable aléatoire suivant une loi uniforme sur $[\![1;6]\!]$ (entiers)
rd.random # variable aléatoire suivant une loi uniforme sur l'intervalle [0;1[
rd.binomial(1,0.3) # variable aléatoire suivant une loi de Bernoulli de paramètre p=0,3
rd.binomial(10,0.3) # variable aléatoire suivant une loi binomiale n=10 et p=0,3
rd.geometric(0.5) # variable aléatoire suivant une loi géométrique de paramètre p=0,3
rd.poisson(4) # variable aléatoire suivant une loi de Poisson de paramètre $\lambda=4$
	X=... # liste des valeurs de la variable aléatoire X après plusieurs simulations
	E=sum(X)/len(X) # valeur moyenne empirique
	V=sum((X-E)**2)/len(X) # si X est un array, variance empirique
	[[x,X.count(x)/len(X)] for x in set(X)]#tableau à deux dimensions valeurs × fréquences

L=[]
for k in range(*n*) :
x=...
y=... # peut dépendre de x
L.op([x,y]) # création d'une liste de *n* couples empiriques (x;y)
	X=[couple[0] for couple in L] # valeurs de x
	EX=sum(X)/len(X) # moyenne empirique des x
	Y=[couple[1] for couple in L] # valeurs de y
	EY=sum(Y)/len(Y) # moyenne empirique des y
	COV=sum([(couple[0]-EX)*(couple[1]-EY) for couple in L])/len(L) # covariance empirique