

## 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 *
<code>5//3</code> # quotient de la division euclidienne de 5 par 3 <code>5%3</code> # reste de la division euclidienne de 5 par 3 <code>int(-1.9)</code> # troncature entière <code>int(booléen)</code> #fct caractéristique True $\mapsto$ 1 et False $\mapsto$ 0 <code>round(1/6,4)</code> # float arrondi du réel 1/6 à $10^{-4}$ près <code>1.2**3.4</code> # $(1,2)^{3,4} = e^{3,4 \ln(1,2)}$ car $1,2 > 0$	<code>np.pi</code> # float approximation de $\pi$ <code>np.sin</code> , <code>np.cos</code> , <code>np.tan</code> # fcts pouvant s'appliquer à une liste <code>np.arcsin</code> , <code>np.arccos</code> , <code>np.arctan</code> <code>np.sqrt</code> , <code>np.log</code> , <code>np.exp</code> # peuvent s'appliquer à un complexe	<code>floor(-1,9)</code> # fonction partie entière <code>Rational(1/2+3/4)</code> # opérations dans $\mathbb{Q}$ <code>numer(Rational(5/4))</code> # numérateur <code>denom(Rational(5/4))</code> # dénominateur <code>sin</code> , <code>cos</code> , <code>tan</code> , <code>asin</code> , <code>acos</code> , <code>atan</code> , <code>factorial</code> <code>binomial(3,2)</code> # 2 parmi 3 <code>Piecewise((expr1,cond1),(expr2,con2),...)</code> # expression définie par morceaux
<code>2+3j</code> # nombre complexe $2+3i$ <code>1j</code> # nombre complexe $i$ <code>(2+3j).real</code> # partie réelle <code>(2+3j).imag</code> # partie imaginaire <code>(2+3j).conjugate()</code> # conjugué <code>abs(2+3j)</code> # module <code>(1+1j)**(1/3)</code> # donne UNE racine cubique de $1+i$	# idem mais pouvant s'appliquer à une liste <code>np.real()</code> <code>np.imag()</code> <code>np.conj()</code> <code>np.absolute()</code> <code>np.angle(2+3j)</code> # approximation d'un argument de $2+3i$	<code>2+3*I</code> # nombre complexe $2+3i$ <code>I</code> # nombre complexe $i$ <code>re(2+3*I)</code> # partie réelle <code>im(2+3*I)</code> # partie imaginaire <code>conjugate(2+3*I)</code> # conjugué <code>abs(2+3*I)</code> # module <code>arg(2+3*I)</code> # argument

Les booléens : True, False

Comparaisons : <code>==</code> , <code>!=</code> , <code>&gt;</code> , <code>&lt;</code> , <code>&lt;=</code> , <code>&gt;=</code> , <code>in</code>	Opérations : <code>and</code> , <code>or</code> , <code>not</code>
--	--

Les chaînes de caractères et le calcul symbolique

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

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

	<pre>import numpy as np # le type array permet de manipuler des tableaux de # nombres dont le cardinal est fixé lors de sa création</pre>	<pre>from sympy import * x,n,k = symbols('x n k')</pre>
<pre>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</pre>	<pre>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 <math>M_{1,2}(\mathbb{R})</math> 3*np.array([1,2]) # multiplication externe dans <math>M_{1,2}(\mathbb{R})</math></pre>	<pre>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 <math>M_{1,2}(\mathbb{R})</math> 3*Matrix([[1,2]]) # multiplication externe dans <math>M_{1,2}(\mathbb{R})</math></pre>
<pre>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)</pre>	<pre>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</pre>	<pre>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</pre>
<pre>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 <math>k \geq \text{len}(L)</math> alors <math>L[k:] = []</math> △ si <math>k &lt; 1</math> alors <math>L[:k] = []</math> L[i],L[j]=L[j],L[i] #permutation de deux éléments</pre>	<pre>idem pour array</pre>	<pre>idem pour Matrix</pre>
<pre>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</pre>	<pre>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</pre>	<pre>v1.dot(v2) # produit scalaire de matrices colonnes transpose(v1)*v2 # matrice ligne × matrice colonne v1.cross(v2) # produit vectoriel de matrices colonnes</pre>

Tableaux à 2 dimensions : liste de listes ou, pour des calculs dans  $M_{n,p}(\mathbb{R})$  ou  $M_{n,p}(\mathbb{C})$ , array et Matrix :

Une matrice  $(a_{i,j})_{\substack{1 \leq i \leq n \\ 1 \leq j \leq p}} \in M_{n,p}(\mathbb{K})$  est définie par la liste de ses  $n$  lignes (chacune des lignes étant une liste de  $p$  nombres) :  $\triangle$  les index *ligne\_i* et *colonne\_j* commencent à 0.

	import numpy as np	from sympy import *
<p>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</p>	<p>np.array([[1,2,3],[4,5,6]]) # 2 lignes, 3 colonnes            np.zeros((nb_de_lignes,nb_de_colonnes)) # <math>\triangle</math> couple            np.ones((nb_de_lignes,nb_de_colonnes)) # <math>\triangle</math> 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</p>	<p>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</p>
<p>M[ligne_i] # renvoie la liste indexée par <i>ligne_i</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 <i>colonne_j</i></p>	<p>M[ligne_i,colonne_j] # terme <math>a_{i+1;j+1}</math>            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</p>	<p>M[ligne_i,colonne_j] # terme <math>a_{i+1;j+1}</math>            M[ligne_i <math>\times</math> p+colonne_j] # terme <math>a_{i+1;j+1}</math>            M[ligne_i,:] # matrice ligne            M[:,colonne_j] # matrice colonne</p>
<p># les opérations dans <math>M_{n,p}(\mathbb{K})</math> ne sont pas obtenues par manipulations simples sur les listes de listes.</p>	<p>3*M+N # combinaisons linéaires de matrices            M*N # <math>\triangle</math> produit des termes 2 à 2 <math>\neq M \times N</math>            np.dot(M,N) # produit matriciel <math>M \times N</math>            M.dot(N) # produit matriciel <math>M \times N</math>            M**3 # <math>\triangle</math> cube de chaque terme <math>\neq M^3</math>            np.linalg.matrix_power(M,3) # <math>M^3</math> si M est carrée            np.linalg.inv(M) # <math>M^{-1}</math> si M est inversible</p>	<p>3*M+N # combinaisons linéaires de matrices            M*N # produit matriciel <math>M \times N</math>            M**3 # <math>M^3</math> si M est carrée            M**(-1) # <math>M^{-1}</math> si M est inversible</p>
	<p>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 <math>\mathbb{C}</math>            np.linalg.eig(M) # couple : spectre, matrice de passage</p>	<p>transpose(M), trace(M), det(M), M.norm()            M.charpoly() #polynôme caractéristique            M.eigenvals() # val propres : ordre de multiplicité            M.eigenvecs() # 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>

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 coefficients 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,[ $y_{n-1}, \dots, y_0$ ]) # $Q^{(n)}=P$ , $Q(0)=y_0, \dots, Q^{(n-1)}(0)=y_{n-1}$	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,listes_ordonnées) plt.axis('equal') # repère orthonormal 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.optimize import fsolve fsolve(f, $x_0$ ) # approximation d'une solution de $f(x)=0$ par itérations avec valeur initiale $x_0$ . $\triangle$ 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 $x \rightarrow 2$ i.e $h \rightarrow 0$
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)	integrate(f,(x,0,1)) # valeur exacte de $\int_0^1 f(t) dt$
quad(f,0,np.inf) # couple (valeur approchée de $\int_0^{+\infty} f(t) dt$ , majoration de l'erreur)	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 $\sum_{n=0}^9 a_n x^n$	n=symbols('n')
import numpy as np [ $a_0 + \sum [ a_n * np.cos(n * \omega * x) + b_n * np.sin(n * \omega * x) ]$ for n in range(1,10)] for x in X # somme partielle de la série de Fourier	summation( $a_n * x^{**n},(n,0,oo)$ ) # somme de la série entière $\sum_{n \geq 0} a_n x^n$ $a_0 + \text{summation}( a_n * \cos(n * \omega * x) + b_n * \sin(n * \omega * x), (n,1,oo) )$ # 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 <sub>0</sub> , t <sub>1</sub> )) # 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 <sub>0</sub> , t <sub>1</sub> )) # 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 $t \rightarrow 2$ i.e. $h \rightarrow 0$

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

<pre>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</pre>	<pre>from sympy import * x,y =symbols('x y') f=... # expression dépendant de x et y</pre>
<pre>plt.contour(X,Y,Z,0) # courbe d'équation <math>f(x,y)=0</math> plt.contour(X,Y,Z,[z1, z2, z3]) # courbes d'équations <math>f(x,y)=z_1</math> etc... plt.show()</pre>	<pre>plot_implicit(f,(x, x_min, x_max),(y, y_min, y_max)) # courbe d'équation <math>f(x,y)=0</math> diff(f,x) # expression de <math>\frac{\partial f}{\partial x}(x;y)</math> Matrix([diff(f,x).subs(x,a).subs(y,b),diff(f,y).subs(x,a).subs(y,b)]) # <math>\nabla f(a;b)</math></pre>
<pre>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()</pre>	<pre>plotting.plot3d(f,(x, x_min, x_max),(y, y_min, y_max),xlabel='x',ylabel='y',zlabel='z') # surface d'équation <math>z=f(x;y)</math>  solve([diff(f,x),diff(f,y)],[x,y]) # recherche de point critique</pre>

Fonctions vectorielles de plusieurs variables : système

<pre>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</pre>	<pre>from sympy import * x,y,z=symbols('x y z') f= ( ... , ... , ... ) # tuple ou liste d'expressions dépendant de x, y et z</pre>
<pre>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</pre>	<pre>solve(f,[x,y,z]) # solutions formelles de l'équation vectorielle <math>f(x;y;z)=0_{\mathbb{R}^n}</math></pre>

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

<pre>from scipy.integrate import odeint X=... # liste ou array décrivant l'intervalle de résolution</pre>	<pre>from sympy import *</pre>
<pre>def f(y,x) :     ...     return ... # pour résoudre <math>y'(x)=f(y,x)</math></pre>	<pre>x=symbols('x') y=Function('y') eq=... # expression utilisant y(x) et diff(y,x)</pre>

<pre>Ly=odeint(f, y0 ,X) # approximation des y(X[i]) pour la solution vérifiant la condition initiale y(X[0])= y0</pre>	<pre>dsolve(eq,y(x)) # ensemble des solutions de eq=0, Δ l'intervalle de résolution peut poser problème</pre>
<pre>def f(Y,x) : ... return np.array([... ,...]) # système différentiel <math>\begin{pmatrix} Y'[0] \\ \vdots \\ Y'[n-1] \end{pmatrix} = \begin{pmatrix} f(Y,x)[0] \\ \vdots \\ f(Y,x)[n-1] \end{pmatrix}</math></pre> <pre>LY=odeint(f,np.array[ y0 ,... , yn-1 ],X) # approximation des arrays Y(X[i]) pour la solution vérifiant la condition initiale Y(X[0])=( y0 ,... , yn-1 )</pre>	<pre>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 <math>\begin{cases} eq1=0 \\ eq2=0 \end{cases}</math></pre>
<pre>def f(Y,t) : # ici Y=[y, y', ..., y<sup>(n-1)</sup>] et on veut résoudre Y'=AY n=... #ordre de l'équation différentielle linéaire A=np.zeros([n,n]) for i in range(n-1): A[i,i+1]=1 A[n-1,:]=[ a0 ,..., an-1 ] # pour résoudre y<sup>(n)</sup>=a0y+...+an-1y<sup>(n-1)</sup> return A.dot(Y) Ly=odeint(f,np.array[ y0 ,... , yn-1 ],X)[0] # première composante de Y, approximation des y(X[i]) pour la solution vérifiant les conditions initiales y<sup>(k)</sup>(X[0])=yk</pre>	<pre>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, Δ l'intervalle de résolution peut poser problème</pre>

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

<pre>import numpy.random as rd</pre>	
<pre>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 λ=4</pre>	<pre>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</pre>
<pre>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)</pre>	<pre>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</pre>