Compte-rendu TP Projets Maths

Tom Chauvel

Table of contents

1 : Partie 1 : Décourverte Python

1.1 : Exercice 1 : Découverte Python

Dans cette partie, on va se familiariser avec numpy et python.

```
import numpy as np
```

```
print( 2+3 )
```

```
5
```

```
print( 2**10 )
```

```
1024
```

```
print( np.sin(2 * np.pi) )
```

```
-2.4492935982947064e-16
```

environ 0, python approxime très mal les nombres flottants

```
print( np.exp(1j * np.pi) + 1 )
```

```
1.2246467991473532e-16j
```

$= \cos(\pi) - 1 + i \times \sin(\pi) = 0$, même problème

```
A = np.matrix([[1,2],[3,4]])
B = np.matrix([[2,3],[4,5]])

print( A*B )
```

```
[[10 13]
 [22 29]]
```

```
A = np.matrix([[1,7],[4,2]])

print( np.linalg.det(A) )
```

```
-25.99999999999999
```

1.2 : Exercice 2 : Eigenvalues, eigenvectors

Ici on va chercher les valeurs et vecteurs propres de la matrice A.

```
A =1/2*np.matrix([
  [np.sqrt(3)+1,-2],
  [1,np.sqrt(3)-1]
])

res = np.linalg.eig(A)

for i in range(len(res.eigenvalues)):
  print(f'Valeur  Propre {i} : {res.eigenvalues[i]}')
  print(f'Vecteur Propre {i} : {res.eigenvectors[i]}')
```

```
Valeur  Propre 0 : (0.8660254037844384+0.49999999999999994j)
Vecteur Propre 0 : [[0.81649658+0.j 0.81649658-0.j]]
Valeur  Propre 1 : (0.8660254037844384-0.49999999999999994j)
Vecteur Propre 1 : [[0.40824829-0.40824829j 0.40824829+0.40824829j]]
```

Maintenant calculons le produit $A^x \times v$

```
A =1/2* np.matrix([[np.sqrt(3)+1,-2],[1,np.sqrt(3)-1]])

v = np.matrix([[1],[2]])

for i in range(1,14):
  print(f'A**{i}*V : {np.dot(A**i,v)}')
```

```
A**1*V : [[-0.6339746 ]
 [ 1.23205081]]
A**2*V : [[-2.09807621]
 [ 0.1339746 ]]
A**3*V : [[-3.]
 [-1.]]
A**4*V : [[-3.09807621]
 [-1.8660254 ]]
A**5*V : [[-2.3660254 ]
 [-2.23205081]]
A**6*V : [[-1.]
 [-2.]]
A**7*V : [[ 0.6339746 ]
 [-1.23205081]]
A**8*V : [[ 2.09807621]
 [-0.1339746 ]]
A**9*V : [[3.]
 [1.]]
```

```
A**10*V : [[3.09807621]
 [1.8660254 ]]
A**11*V : [[2.3660254 ]
 [2.23205081]]
A**12*V : [[1.]
 [2.]]
A**13*V : [[-0.6339746 ]
 [ 1.23205081]]
```

On remarque que $A^{13} \times v = A \times v$, donc que l'on obtient un cycle qui se répète toutes les 13 fois. Mais il y a aussi une étape intermédiaire au milieu ou le résulat est l'opposé de celui de départ : $A^7 \times v = -A \times v$

Faisons la même chose pour B et C.

```python
B =1/np.sqrt(2)* np.matrix([[np.sqrt(3)+1,-2],[1,np.sqrt(3)-1]])

v = np.matrix([[1],[2]])

for i in range(1,13):
  print(f'B**{i}*V : {np.dot(B**i,v)}')
```

```
B**1*V : [[-0.89657547]
 [ 1.74238296]]
B**2*V : [[-4.19615242]
 [ 0.26794919]]
B**3*V : [[-8.48528137]
 [-2.82842712]]
B**4*V : [[-12.39230485]
 [ -7.46410162]]
B**5*V : [[-13.38426086]
 [-12.6263861 ]]
B**6*V : [[ -8.]
 [-16.]]
B**7*V : [[  7.17260378]
 [-13.93906369]]
B**8*V : [[33.56921938]
 [-2.14359354]]
B**9*V : [[67.88225099]
 [22.627417  ]]
B**10*V : [[99.13843876]
 [59.71281292]]
B**11*V : [[107.07408688]
 [101.01108877]]
B**12*V : [[ 64.]
 [128.]]
```

On remarque aussi le cycle, sauf qu'ici les nombres sont multipliés par -8 toutes les 6 fois, (on le remarque pour n=6 et n=12).

```
C =1/2/np.sqrt(2)* np.matrix([[np.sqrt(3)+1,-2],[1,np.sqrt(3)-1]])

v = np.matrix([[1],[2]])

for i in range(1,13):
  print(f'C**{i}*V : {np.dot(C**i,v)}')
```

```
C**1*V : [[-0.44828774]
 [ 0.87119148]]
C**2*V : [[-1.04903811]
 [ 0.0669873 ]]
C**3*V : [[-1.06066017]
 [-0.35355339]]
C**4*V : [[-0.77451905]
 [-0.46650635]]
C**5*V : [[-0.41825815]
 [-0.39457457]]
C**6*V : [[-0.125]
 [-0.25 ]]
C**7*V : [[ 0.05603597]
 [-0.10889894]]
C**8*V : [[ 0.13112976]
 [-0.00837341]]
C**9*V : [[0.13258252]
 [0.04419417]]
C**10*V : [[0.09681488]
 [0.05831329]]
C**11*V : [[0.05228227]
 [0.04932182]]
C**12*V : [[0.015625]
 [0.03125 ]]
```

De même, On remarque le cycle, sauf qu'ici les nombres sont divisés par -8 toutes les 6 fois, (on le remarque pour n=6 et n=12).

1.3 : Exercice 3 : Visualiser des functions

Dans cette partie on va visualiser des fonctions. Puis les analyser.

```
import matplotlib.pyplot as plt

def plotfunc(f):
  x = np.arange(-2,2,0.001)
  y = f(x)
  plt.plot(x,y)
  plt.show()
```

```
plotfunc(lambda x : x**3)
```

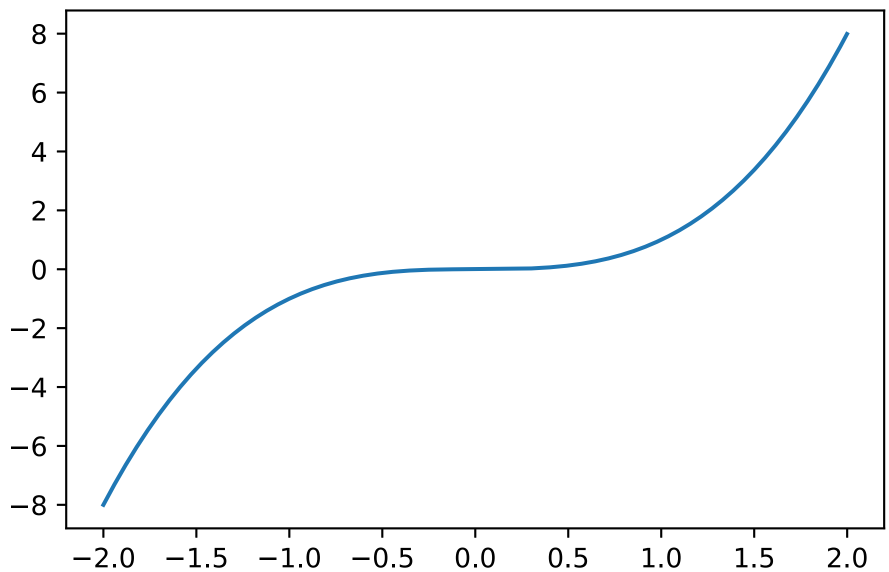


Figure 1: Une fonction cubique, rien de plus banal

```
plotfunc(lambda x : x**2*np.sin(1/x))
```

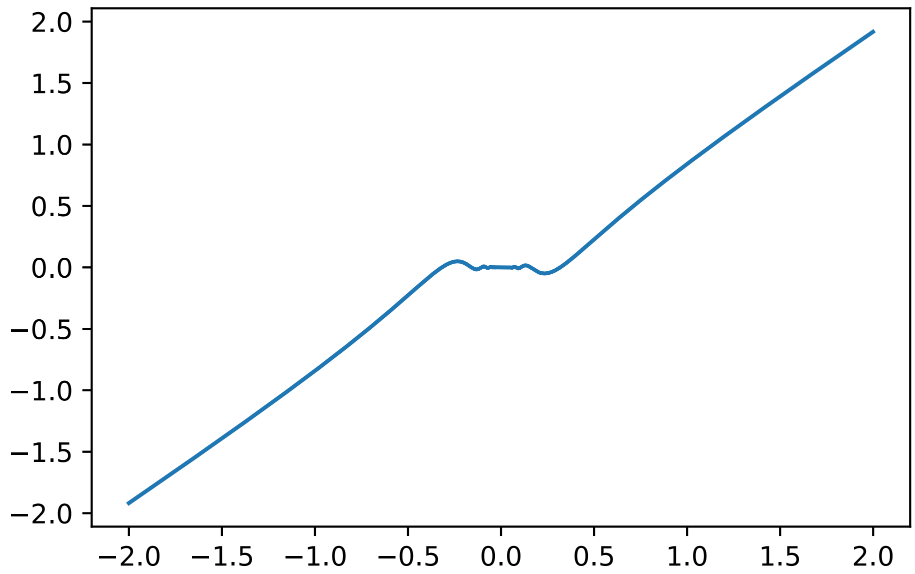


Figure 2: x**2*np.sin(1/x)

au voisinage de zéro, la fonction à des petites variations qui sont dû au sinus. La fonction semble être impaire.

```
plotfunc(lambda x : x**2*np.sin(1/(x**2)))
```

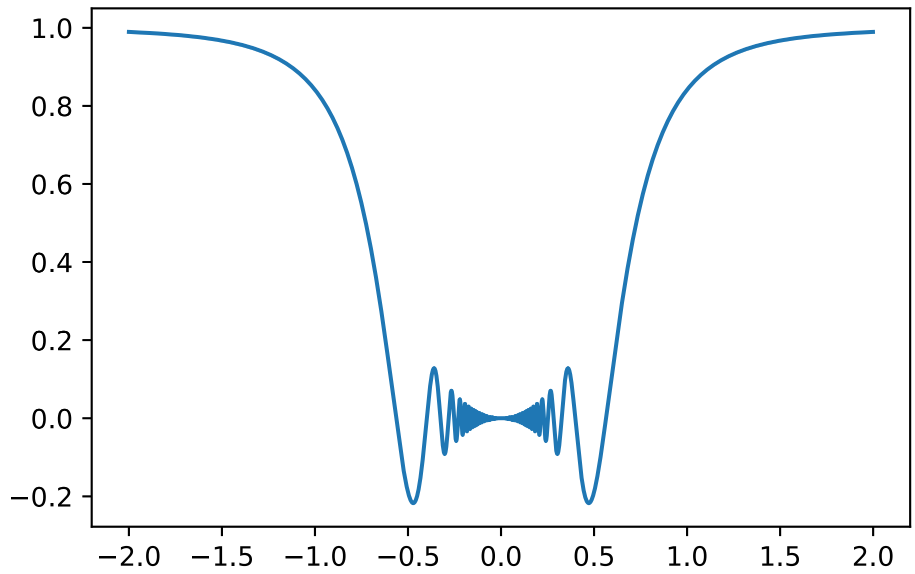


Figure 3: x**2*np.sin(1/(x**2))

de même; au voisinnage de zéro, le sinus fait quelque chose de spécial. On dirait qu'elle fait l'inverse d'un sinus cardinal. Sinon la fonction à l'air d'être borné entre $-1/3$ et 1 et elle semble être paire.

```
plotfunc(lambda x : 1/(1+np.exp(-x)))
```

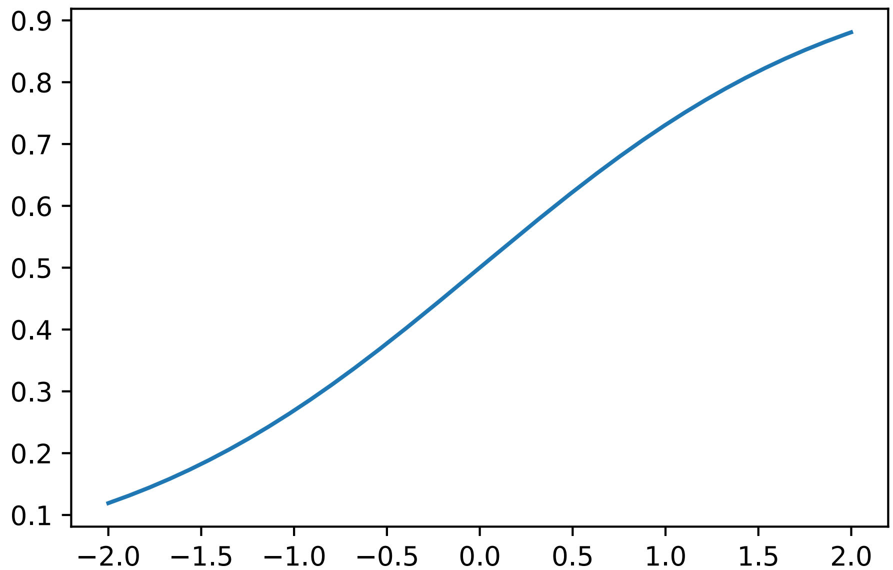


Figure 4: 1/(1+np.exp(-x))

La fameuse fonction sigmoïde. Elle est beaucoup utilisé en IA car elle est bornée entre 0 et 1 et elle est super simple à dérivé, ce qui est génial pour recalculer les poids des neuronnes.

```
plotfunc(lambda x : 1/(1+np.exp(-10*x)))
```

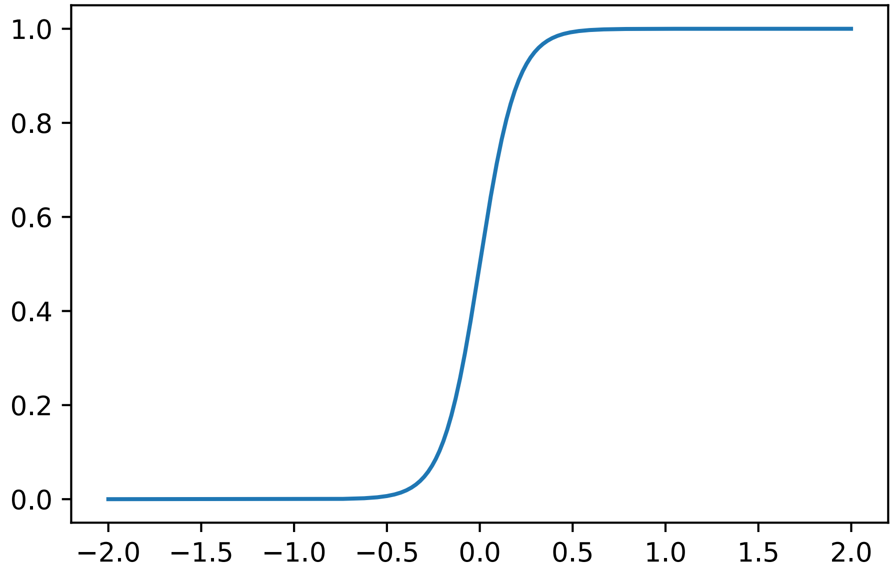


Figure 5: 1/(1+np.exp(−10x))

Ici on a toujours une fonction sigmoïde, mais avec un coefficient plus grand, ce qui modifie ça forme et la valeur de sa pente.

```
plotfunc(lambda x : 1/(1+np.exp(-100*x)))
```

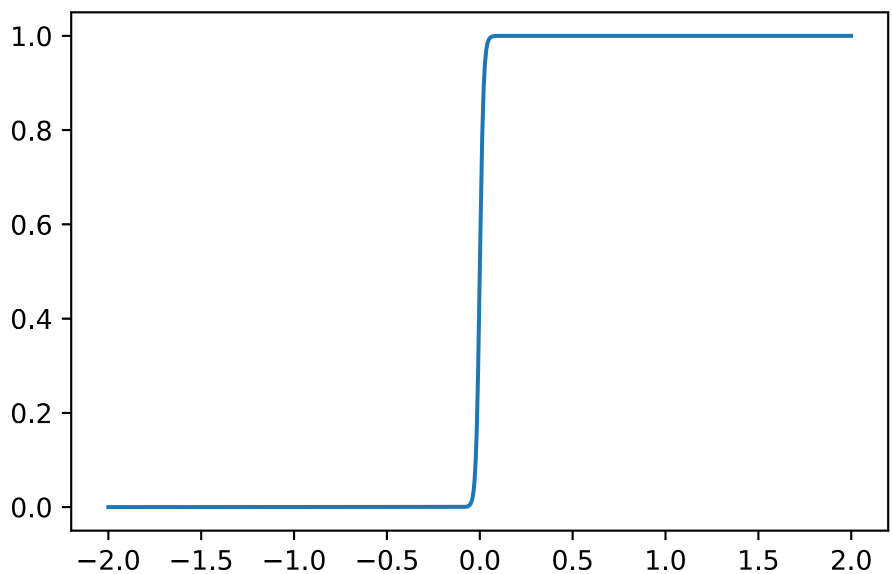


Figure 6: 1/(1+np.exp(−100x))

Pareil, mais avec une pente encore plus verticale.

```
plotfunc(lambda x : (np.exp(x)-np.exp(-x))/(np.exp(x)+np.exp(-x)))
```

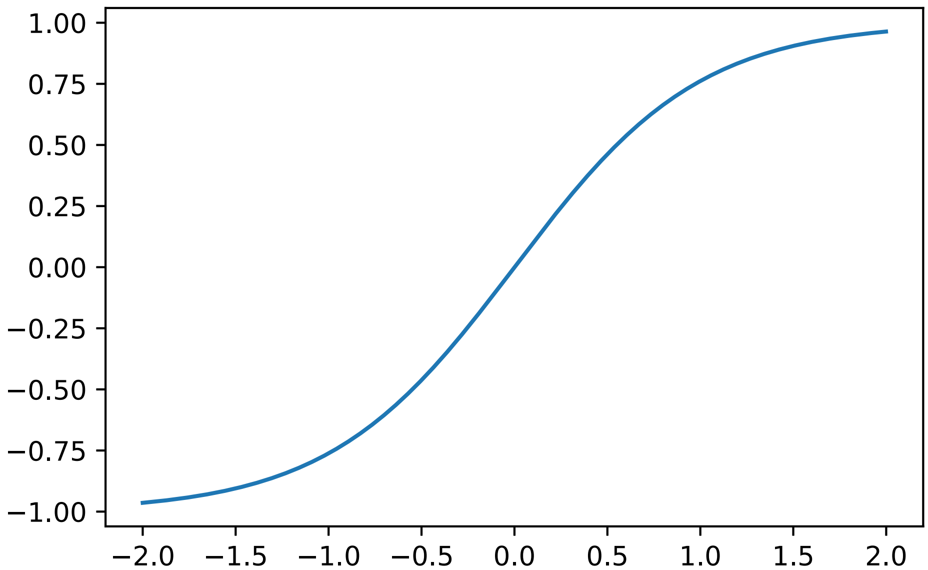


Figure 7: (np.exp(x)-np.exp(-x))/(np.exp(x)+np.exp(-x))

De premier abord, on dirait une fonction sigmoïde, sauf que celle-ci est compris entre -1 et 1, ce qui rends cette fonction impaire. Elle est aussi un peu plus courbée que la sigmoïde à coefficient à 1.

```
plotfunc(lambda x : (np.exp(100*x)-np.exp(-100*x))/(np.exp(100*x)
+np.exp(-100*x)))
```

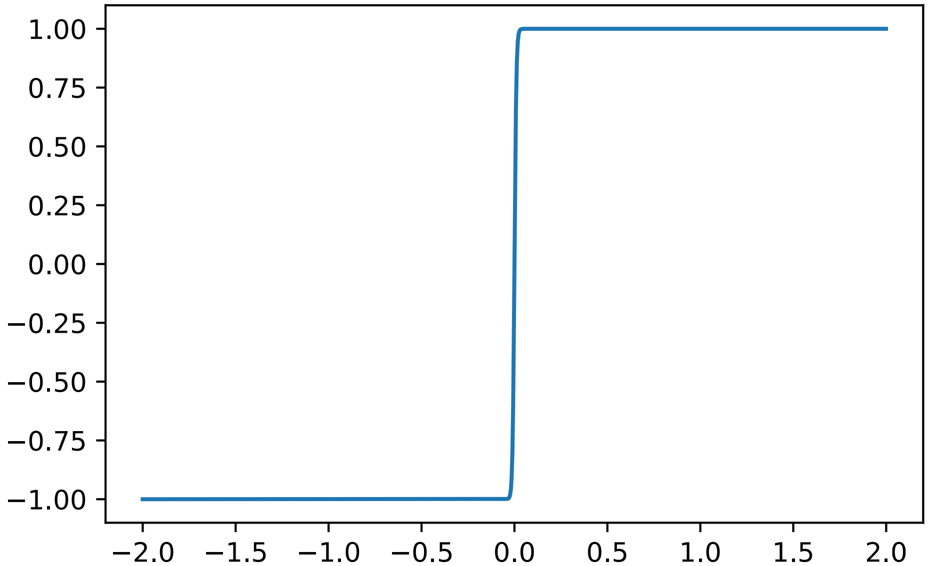


Figure 8: (np.exp(100*x)-np.exp(−100*x))/(np.exp(100*x)+np.exp(−100*x)))

On prend $\frac{1}{1+e^{-100x}}$, on l'allonge un peu pour qu'elle soit bornée entre −1 et 1, et on a la même courbe.

1.4 : Exercice 4 : Algèbre et géometrie

1.4.1 : Points Alignés

comment déterminé si trois points sont alignés ? C'est très simple, on calcule l'équation de droite défini par deux points et on regarde si le troisième points suit l'équation.

```
def areAligned(v1,v2,v3):
  sub = v1-v2
  a = sub[1]/sub[0]
  b = v1[1] - a * v1[0]

  return a*v3[0] + b == v3[1]

v1 = np.array([1,2])
v2 = np.array([2,3])
v3 = np.array([3,4])

print( areAligned(v1,v2,v3) )
```

```
True
```

1.4.2 : Point dans un triangle

Comment définir si un point est un triangle ? Une méthode qui existe est de calculé la somme des angles entre le point d (qui ne définit pas le triangle) et les sommets du triangle. Si cette somme est égale à 2pi oui 360°, c'est ok.

C'est ce que j'ai fait dans la méthode ci-dessous, sauf que j'ai dû arrondir le résultat car les nombres flottants en informatique, c'est pas précis.

```
def isInTriangle(p1,p2,p3,p4):
  # soustraction des points pour avoir les vecteurs, puis on calcule la norme
du vecteur
  a1 = p1-p4
  n1 = np.linalg.norm(a1)
  a2 = p2-p4
  n2 = np.linalg.norm(a2)
  a3 = p3-p4
  n3 = np.linalg.norm(a3)

  # on calcule les cos avec le produit scalaire
  cos1 = np.dot(a1,a2)/n1/n2
  cos2 = np.dot(a2,a3)/n2/n3
  cos3 = np.dot(a3,a1)/n3/n1

  # on somme les angles et on les arrondis
  return np.round( np.arccos(cos1) + np.arccos(cos2) + np.arccos(cos3), 3 ) ==
np.round(2*np.pi,3)


p1 = np.array([0,2])
p2 = np.array([2,0])
p3 = np.array([0,0])
p4 = np.array([0.5,0.5])

print( isInTriangle(p1,p2,p3,p4) )
```

```
True
```

1.4.3 : Point dans un triangle rotaté

C'est la même chose, sauf qu'ici on a va rotater le triangle d'un angle pi/2 sur son barycentre.

Mise à part la rotation avec la matrice, rien de bien méchant.

```
def centreMasse(p1,p2,p3):
  # calcul du barycentre
  return (p1+p2+p3)/3

def rotate(p1,angle,c):
  # on déplace le point pour que le barycentre soit à l'origine
  p1p = p1 - c

  # matrice de rotation
     matrixrot   =   np.array([[np.cos(angle),np.sin(angle)],[-
np.sin(angle),np.cos(angle)]])

  # on rotate et on réajoute le barycentre
  return matrixrot.dot(p1p) + c

def isInTriangleRotate(p1,p2,p3,p4,angle):
  c = centreMasse(p1,p2,p3)

  p1r = rotate(p1,angle,c)
  p2r = rotate(p2,angle,c)
  p3r = rotate(p3,angle,c)

  return isInTriangle(p1r,p2r,p3r,p4)

p1 = np.array([0,2])
p2 = np.array([2,0])
p3 = np.array([0,0])
p4 = np.array([0.5,0.5])

print( isInTriangleRotate(p1,p2,p3,p4,-np.pi/2) )
```

```
True
```

1.4.4 : Point incident

Plus compliqué à comprendre. On a un point, on veut savoir si ce point projeté sur un plan, appartient au triangle qui défini le plan.

Le plus dur, c'est le projeté orthogonal, sinon c'est la même méthode.

L'avantage, une formule existe :

Soit C un point du triangle, $\vec{n} = \begin{pmatrix} a \\ b \\ c \end{pmatrix}$ le vecteur normal du plan, et A le point à projeter.

$$p(x) = a \times x + b \times y + c \times z + d = 0$$

pour trouver d on prend un point du plan.

$$d = -(\vec{n} \cdot C)$$

Pour le projeté :

$$\lambda = \frac{a \times X_A + b \times Y_A + c \times Z_A + d}{a^2 + b^2 + c^2} = \frac{\vec{n} \cdot A}{\vec{n} \cdot \vec{n}}$$

$$A_p = \begin{pmatrix} X_A - a \times \lambda \\ Y_A - b \times \lambda \\ Z_A - c \times \lambda \end{pmatrix}$$

Implémentons cela :

```python
def isIncident(p1,p2,p3,p4,vn):
  d = - np.dot(vn,p1)
  lamb =  (np.dot(vn,p4)+d)/np.dot(vn,vn)
  ph = p4 - vn*lamb
  return isInTriangle(p1,p2,p3,ph)

p1 = np.array([4,2,-1])
p2 = np.array([1,3,1])
p3 = np.array([-3,0,3])
p4 = np.array([0,1.5,0])
vn = np.array([8,-2,13])

print( isIncident(p1,p2,p3,p4,vn) )
```

```
True
```

1.4.5 : Calcul de l'aire d'un triangle

D'un point de vue extérieur c'est simple à faire. Sauf que c'est long de calculer une hauteur sur des triangles qui ne sont pas toujours de la même forme. Donc j'ai chercher un autre méthode, la formule d'Héron.

Soit p le demi-périmètre, a b et c les côtés du triangle

$$p = \frac{a+b+c}{2}$$

$$S = \sqrt{p(p-a)(p-b)(p-c)}$$

```
def aireTriangle(p1,p2,p3):
  a = np.linalg.norm(p1-p2)
  b = np.linalg.norm(p1-p3)
  c = np.linalg.norm(p2-p3)

  p = (a+b+c)/2

  return (p*(p-a)*(p-b)*(p-c))**(1/2)

p1 = np.array([0,2])
p2 = np.array([2,0])
p3 = np.array([0,0])

print( aireTriangle(p1,p2,p3) )
```

```
1.9999999999999993
```

1.4.6 : Polygône convexe

Dernier algorithme, déterminer si un quadrilatère est convexe. Pour ce faire, on doit vérifier si tous les angles intérieurs sont inférieurs à 180°. L'équivalent, c'est de faire à chaque sommet, le produit vectorielle des arrêtes adjacentes à ce sommet, et vérifié que tous les produits vectorielle sont dans le même sens, le produit vectorielle prend l'angle minimal entre les deux arrêtes et l'oriente en fonction.

```
def est_convexe(p1,p2,p3,p4):
  v1 = p2-p1
  v2 = p3-p2
  v3 = p4-p3
  v4 = p1-p4

  a = np.cross(v1,v2)
  b = np.cross(v2,v3)
  c = np.cross(v3,v4)
  d = np.cross(v4,v1)

  cas1 = a <= 0 and b <= 0 and c <= 0 and d <= 0
  cas2 = a >= 0 and b >= 0 and c >= 0 and d >= 0

  return cas1 or cas2

est_convexe( np.array([0, 0]), np.array([1, 1]), np.array([2, 0]), np.array([1,
-1]) )
```

```
True
```

2 : Partie 2 : Algèbre linéaire et apprentissage par la machine

2.1 : Exercice 1 : Résolution de systèmes d'équations linéaires

On doit résoudre $Ax = b$

Ce qui revient à faire $x = A^{-1}b$

```
import numpy as np
```

```
A = np.matrix([
  [ 0, 2, 0, 1],
  [ 2, 2, 3, 2],
  [ 4,-3, 0, 1],
  [ 6, 1,-6,-5]
])

B = np.matrix([
  [ 0],
  [-2],
  [-7],
  [ 6],
])

print(np.linalg.inv(A).dot(B))
```

```
[[-0.5       ]
 [ 1.       ]
 [ 0.33333333]
 [-2.       ]]
```

2.2 : Exercice 2

```
A = np.matrix([
  [ 5, 6 ],
  [ 6, 7 ],
  [ 1, 1 ]
])

B = np.matrix([
  [ 3],
  [ 1],
  [-5],
])


def f(A,B,x,y):
  ax = np.matrix([[x],[y]])
  return np.linalg.norm( A.dot(ax) - B )
```

Ce système ne possède pas de solution car il est surdeterminé (et il n'y pas de lignes similaires).

$$\begin{pmatrix} 5 & 6 \\ 6 & 7 \\ 1 & 1 \end{pmatrix} \times \begin{pmatrix} x \\ y \end{pmatrix} - \begin{pmatrix} 3 \\ 1 \\ 5 \end{pmatrix}$$

on a donc

$$\begin{cases} 5x + 6y - 3 = 0 \\ 6x + 7y - 1 = 0 \\ 1x + 1y - 5 = 0 \end{cases}$$

pour calculer l'erreur, on doit calculer :

$$e = \sqrt{(5x + 6y - 3)^2 + (6x + 7y - 1)^2 + (1x + 1y - 5)^2}$$

après avoir développé tous les termes on trouve

$$e = \sqrt{62x^2 + 146xy - 32x - 40y + 86y^2 + 44}$$

et on fait quoi après ? Et bien, on cherche à minimiser cette erreur, ce qui revient à déterminer quand est-ce que la fonction ci-dessus a ses dérivés qui s'annulent

$$\begin{cases} \frac{\partial e}{\partial x} = 124x + 146y - 32 \\ \frac{\partial e}{\partial y} = 146x + 172y - 40 \end{cases}$$

On doit trouver ces deux différentielles égale à 0

$$\begin{cases} 0 = 124x + 146y - 32 \\ 0 = 146x + 172y - 40 \end{cases}$$

$$\begin{cases} 146y = 32 - 124x \\ 172y = 40 - 146x \end{cases}$$

$$\begin{cases} y = \frac{32-124x}{146} \\ y = \frac{40-146x}{172} \end{cases}$$

$$\frac{32 - 124x}{146} = \frac{40 - 146x}{172}$$

$$(32 - 124x) \times 172 = (40 - 146x) \times 146$$

$$5504 - 21328x = 5840 - 21316x$$

$$336 = -12x$$

$$x = -28$$

$$y = \frac{40 - 146 \times (-28)}{172} = \frac{40 + 4088}{172} = \frac{4128}{172} = 24$$

donc la solution qui minimise l'erreur est $x = \begin{pmatrix} -28 \\ 24 \end{pmatrix}$

```
f(A,B,-28,24)
```

```
1.7320508075688772
```

2.2.1 : Moore-Penrose pseudo-inverse

Maintenant on va vérifier si le résultat précédent est bon avec la Moore-Penrose pseudo-inverse.

```
import time

def solveMoore(A,B):
  A_plus = np.linalg.inv(A.T.dot(A)).dot(A.T)
  return A_plus.dot(B)

t = time.time()

res = solveMoore(A,B)

print("temps" , time.time()-t)
print(res)
```

```
temps 0.0
[[-28.]
 [ 24.]]
```

On trouve la même chose, mais le calcul est tellement simple qu'il est instantané.

2.2.2 : passage à l'échelle

Maintenant on va pousser les limites de cette méthode

```
import numpy.random as rd

def surdeter(row,col):
  A = np.random.randint(1024, size=(row, col)) - 512
  B = np.random.randint(1024, size=(row, 1)) - 512
  return A,B

A,B = surdeter(40000,200)

t = time.time()

res = solveMoore(A,B)

print("temps" , time.time()-t)
```

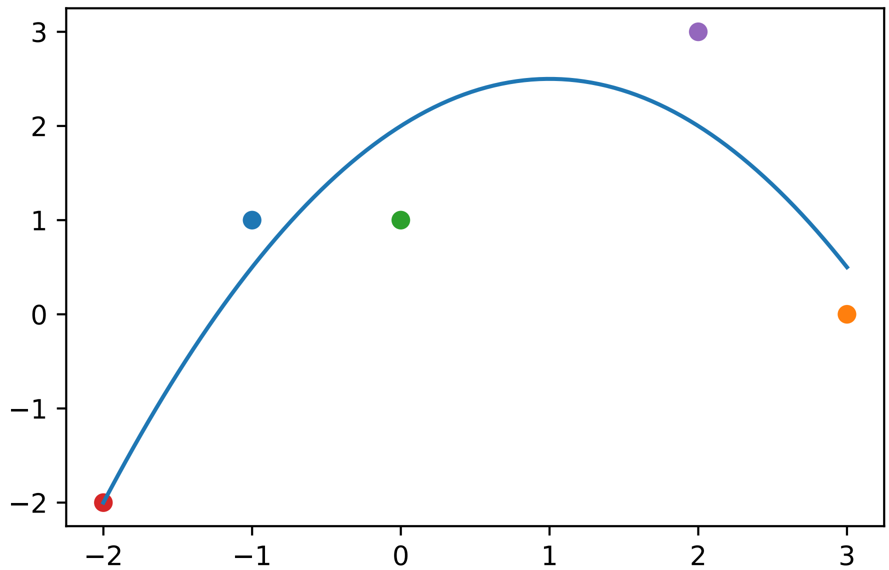
```
temps 3.055260181427002
```

Même si mon ordinateur est puissant, je ne vais pas allez plus loin que 40000 lignes et 200 colonnes. 3 secondes pour une matrice de cette taille c'est plûtot rapide.

2.2.3 : Regression polynomiale

Dans cette nouvelle partie, on veut approximer un nuage de points par un polynome du second degré.

```
import matplotlib.pyplot as plt

pts = [(-1,1),(3,0),(0,1),(-2,-2),(2,3)]

for x,y in pts:
  plt.scatter(x, y)

xs = np.linspace(-2,3,100)

A = np.matrix([[pt[0]**i for i in range(3)] for pt in pts])
B = np.matrix([[pt[1]] for pt in pts])

res = solveMoore(A,B)

res2 = [r[0,0] for r in res]
res2.reverse()

plt.plot( xs,np.polyval(np.poly1d(res2),xs) )

plt.show()


RSS = 0
for pt in pts:
  RSS += ( pt[1] - np.poly1d(pt[0]) ) ** 2

print( np.sqrt( RSS / 6 )[0] )
print( np.linalg.norm(A.dot(res)-B) )
```




```
1.5811388300841898
1.5811388300841898
```

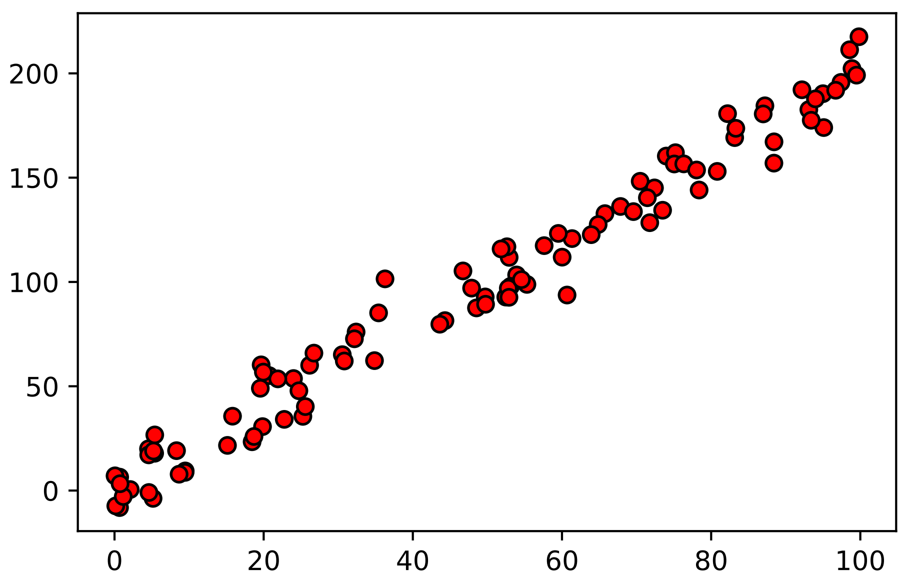
D'après ce que j'en déduis :

erreur $= \sqrt{\frac{RSS}{N+1}}$

3 : Partie 3 : Regession lineaire, descente de gradient

3.1 : Generate data

```
import numpy as np
import matplotlib.pyplot as plt

# Number of data points
N = 100

X = np.random.uniform(low=0, high=100, size=N)

# Making y = 2x + 1 + some gaussian or normal noise (assumption of linear
regression itself)
Y = 2 * X + 1 + np.random.normal(scale=10, size=N)

# Plotting the data to see if it looks linear
plt.scatter(X, Y, edgecolors='black', color="red")
plt.show()
```



les paramètres d'un modèle linéaire sont la pente (w0) et l'ordonné à l'origine (w1).

3.2 : Loss function

```
def SSR(w,x,y):
    N = len(y)
    s = 0

    lw = len(w)
    fw = lambda x : sum( w[i]*x**(lw-i-1) for i in range(lw) )

    for i in range(0,N):
        s += ( fw( x[i] ) - y[i] )**2
    return s

def MSE(w,x,y):
    N = len(y)
    return SSR(w,x,y)/N
```

```
print(SSR([2,1],X,Y))
print(MSE([2,1],X,Y))
```

```
10555.887405720425
105.55887405720425
```

3.3 : Gradient descent

```python
def gradient(w,x,y):
    lw = len(w)
    fw = lambda x : sum( w[i]*x**(lw-i-1) for i in range(lw) )

    l = ( fw(x) - y )

    grad_w = np.array([ sum(2 * x ** (lw-i-1) * l) for i in range(lw) ])

    return grad_w

gradient([2,1],X,Y)
```

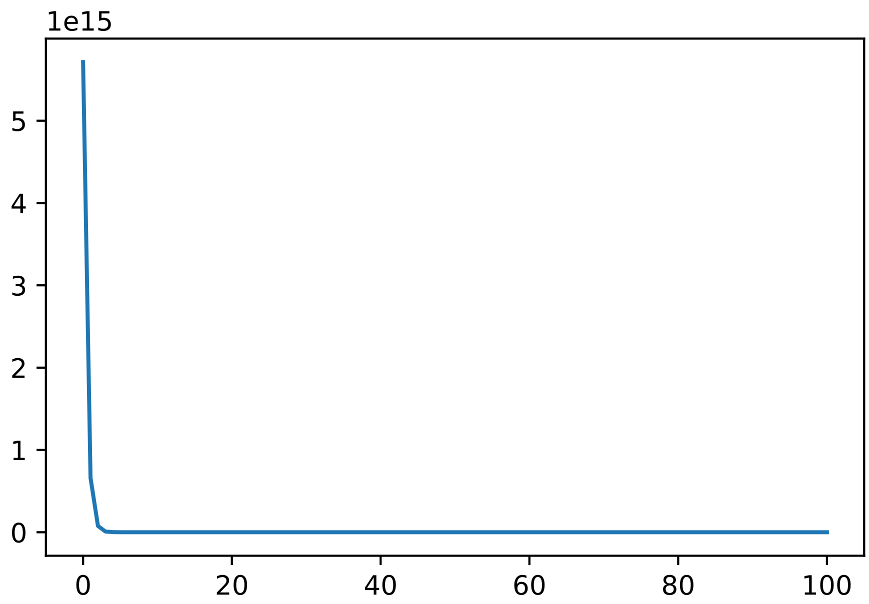
```
array([13580.33664556,   248.47071687])
```

```python
def descenteGradient(w,x,y,µ,n):
    for _ in range(n):
        grad_w = gradient(w,x,y)
        w -= µ * grad_w
    return w

def descenteGradient2(w,x,y,µ,ϵ):
    normeE = np.inf
    while normeE > ϵ:
        grad_w = gradient(w,x,y)
        w -= µ * grad_w
        normeE = np.abs(np.max(grad_w))
    return w

print( descenteGradient([0,0],X,Y,10**(-6),100) )
print( descenteGradient2([0,0],X,Y,10**(-6),10**(-1)) )
```

```
[1.99358798 0.02986075]
[1.99359696 0.02924155]
```

3.4 : Experiment

```
def descenteGradient3(w,x,y,µ,n):
    l = []
    for _ in range(n):
        grad_w = gradient(w,x,y)
        l.append(grad_w)
        w -= µ * grad_w
    l.append(w)
    return l

n=100
plt.plot(np.arange(0,n+1,1),[MSE( w, X, Y ) for w in descenteGradient3([0,0],X,Y,
10**(-6),n)])

plt.show()
```



ça converge vite.

Maintenant modifions un peu les paramètres. Enlevons l'aléatoire, diminuons la tolérance de l'epsilon et augmentons le pas (µ).

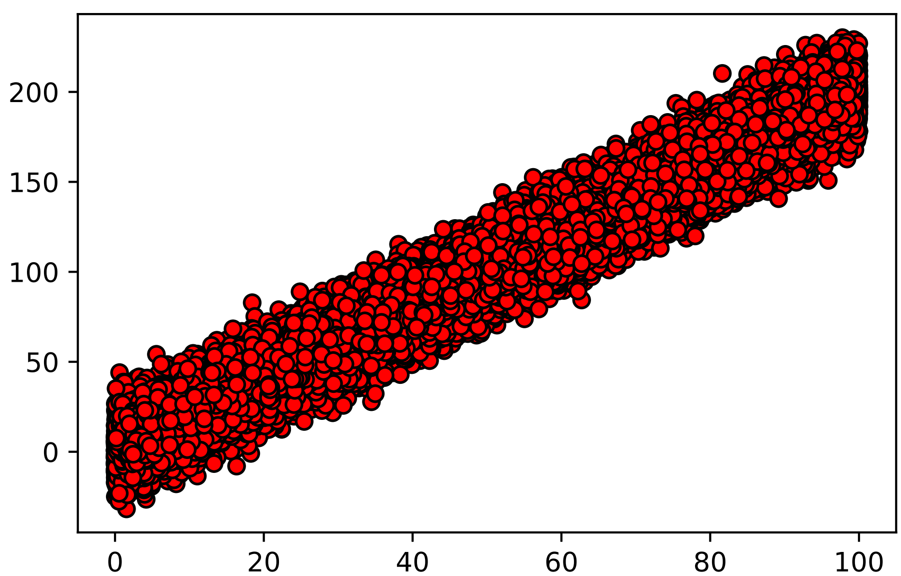
```
X = np.random.uniform(low=0, high=100, size=N)
Y = 2 * X + 1 # + np.random.normal(scale=10, size=N)

print( descenteGradient([0,0],X,Y,10**(-3),100) )
print( descenteGradient2([0,0],X,Y,10**(-3),10**(-1)) )
```

```
[-2.88604565e+283 -4.25621633e+281]
[nan nan]
```

```
C:\Users\tomch\AppData\Local\Temp\ipykernel_9524\3115440749.py:7:
RuntimeWarning: overflow encountered in scalar add
  grad_w = np.array([ sum(2 * x ** (lw-i-1) * l) for i in range(lw) ])
C:\Users\tomch\AppData\Local\Temp\ipykernel_9524\3171002804.py:11:
RuntimeWarning: invalid value encountered in subtract
  w -= μ * grad_w
```

Bizarre, on ne converge pas. C'est totalement normal, car comme on a augmenté μ, dès que l'on va vouloir récalculé le gradient, celui-ci n'aura pas un coefficient trop grand pour descendre et va à chaque itération remonter.

```
N = 100000


X = np.random.uniform(low=0, high=100, size=N)
Y = 2 * X + 1 + np.random.normal(scale=10, size=N)

plt.scatter(X, Y, edgecolors='black', color="red")
plt.show()

print( descenteGradient([0,0],X,Y,10**(-10),100) )
```




```
[2.01239356 0.03069089]
```

Avec un ensemble de 10000 élements, j'ai du diminué μ à 10^{-10} pour converger.