Université of Science et Technologies HOUARI BOUMEDIENE
Département d'informatique

APPRENTISSAGE AUTOMATIQUE ET RESEAUX DE NEURONNES

« RAPPORT TP 2 »

NOM	PRENOM	MATRICULLE	GROUPE
BELIMI	***Ibrahim Sabri***	***161631074255***	***A1 TP1***
ARAB	***MAHER***	***171731045353***	***A1 TP1***
ZAIT	***Fouad***	***181831072145***	***A1 TP1***
ASMA	***SRAOUIA***	***161631102642***	***A1 TP1***

OBJECTIFS

- Dans ce TP, dans la 1ere partie nous aimerions prédire le bénéfice d'une entreprise dans plusieurs ville en nous basant sur les habitants de cette ville uniquement (1 caractéristique) puis dans la 2eme partie nous allons prédire le prix d'une maison à partir de la superficie et le nombre de chambres (plusieurs caractéristiques).

- Pour ce faire, nous étudierons un ensemble de données avec le bénéfice d'une entreprise d'une maison (y) et les caractéristiques des habitants (X) pour la 1ère partie et nous étudierons un ensemble de données avec le prix d'une maison (y) et les caractéristiques la superficie et le nombre de chambres (X_0,X_1,X_2) que nous avons généraliser (X_0,X_1,X_2,X_4 …..X_n) dans l'implémentation de notre algorithme .

- La prédiction se fera avec l'algorithme de descente du gradient.

Importation des librairies nécessaires pour le travail

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

numpy : pour la manipulation des tableaux (array)
matplotlib : pour la manipulation des graphs (2d, 3d)
time : pour calculer le temps d'exécution

Lecture des fichiers de données pour les classifier

Code :

```
# données
data = np.genfromtxt('data.csv', delimiter=',', dtype=int)
data.shape
```

Résultat :

===> (97, 2)

97 : signifie le nombre de données (ou exemples) dans le fichier (***data.csv***) c.à.d. il traite ***97 villes***
2 : représente 2 colonnes dans le fichier
(colonne 1 X: la population d'une ville
et
colonne 2 Y: le bénéficie d'une ville)

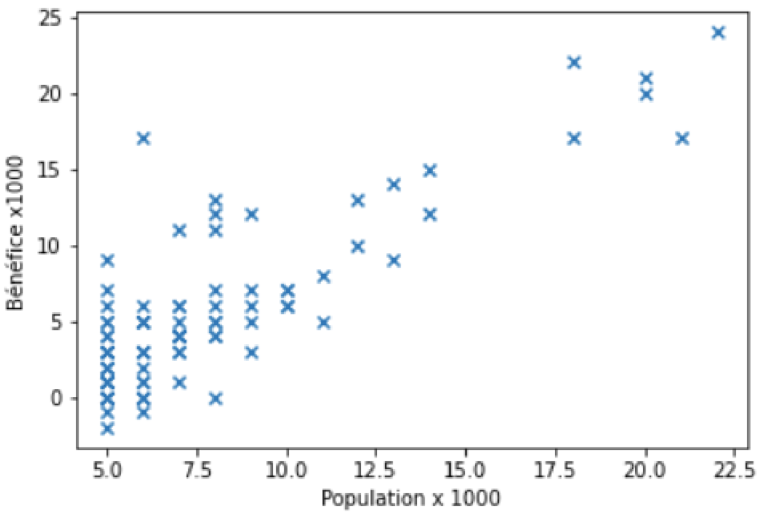
Représentation des données sous graph :

Code :

```
# rajoutons l'ordonnée à l'origine theta 0
intercept=np.ones((data.shape[0],1))
X=np.column_stack((intercept,data[:,0]))
y = data[:, 1];
```

```
# traçons ces données pour visualisation
plt.scatter(X[:,1],y,marker ='x')
plt.xlabel('Population x 1000')
plt.ylabel('Bénéfice x1000')
```

Résultat :



L'axis (X) : représente la 1 ère colonne du fichier (data.csv): **(la population d'une ville)**
L'axis (Y) : représente la 2 ème colonne du fichier (data.csv): **(le bénéficie d'une ville)**

Descente du Gradient

Préparation des fonctions:

1ere Partie : une seule variable

1. La Fonction de Cout (Sans vectorisation) :

Code:

Descente du Gradient : Préparation des fonctions

1- Calcul du coût

Cette fonction servira à calculer le cout $J(\theta_0, \theta_1)$

Elle prendra l'ensemble de données d'apprentissage en entrée ainsi que les paramètres définis initialement

```
def computeCostNonVect(X, y, theta):
    # idéalement, tracer le coût à chaque itération pour s'assurer que la descente du gradient est correcte

    # calculer le coût avec et sans vectorisation,
    # comparer le temps de traitement
    j=0
    for i in range(len(X)):
        j=j+(((theta[0]*X[i,0])+(theta[1]*X[i,1]))-y[i])**2
    j=j/(2*(len(X)))
    return j
print(computeCostNonVect(X, y, theta))
```

```
[29.25773196]
```

Cette fonction nous permettra de calculer le cout vu en cours on a implémenté la fonction de cout ci-dessous :

$$J(\theta_0, \ \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta (x^{(i)}) - y^{(i)})^2$$

Avec :

hθ(x)= θ0X0 + θ1X1

X0 =1

On a modéliser notre θ_0 et θ_1 par un vecteur à 2 lignes et 1 colonne la 1ere ligne représente θ_0 et la 2eme ligne représente θ_1 **theta[1,0] = $\boldsymbol{\theta_1}$, theta[0,0] = $\boldsymbol{\theta_0}$**.nous allons d'abord parcourir tous les $\mathbf{X^i}$ et calculer le cout grâce à la formule et les sommer pour avoir le cout **J(θ0 , θ1).**

2. La Fonction de Cout (Avec vectorisation) :

```
def computeCost(X, y, theta):
    # idéalement, tracer le coût à chaque itération pour s'assurer que la descente du gradient est correcte

    # calculer le coût avec et sans vectorisation,
    # comparer le temps de traitement
    cost=(1/(2*len(y)))*np.sum(np.square(X.dot(theta)-y.reshape(97,1)))
    return cost
print(computeCost(X, y, theta))
```

```
29.257731958762886
```

Cette fonction nous permettra de calculer le cout vu en cours, comme la précédente juste que cette fois pour raccourcir les calculs et **gagner en temps d'exécution** au lieu de faire une boucle pour comparer à chaque fois on calcule le cout **en une itération** grâce à **numpy .**

3. Fonction de la descente du gradient :

2- Fonction de la descente du gradient

Cette fonction mettra à jour les paramètres θ_0, θ_1 jusqu'à convergence: atteinte du nombre d'itérations max, ou dérivée assez petite.

```
all_t0=[]
all_t1=[]
all_costs=[]
def j(t0,t1,X,y):
    j1=0
    j0=0
    j=[]
    for i in range(len(X)):
        j1=j1+(((t0*X[i,0])+(t1*X[i,1]))-y[i])*X[i,1]
        j0=j0+(((t0*X[i,0])+(t1*X[i,1]))-y[i])
    j1=j1/(len(X))
    j0=j0/(len(X))
    j.append(j0)
    j.append(j1)
    return j

def gradientDescent(X, y, theta, alpha, iterations):
    # garder aussi le cout à chaque itération
    # pour afficher le coût en fonction de theta0 et theta1
    i=0
    while(i<iterations):
        jd=[]
        t0=theta[0]
        t1=theta[1]
        jd=j(t0,t1,X,y)
        print(jd)
        theta[0][0]=theta[0][0]-alpha*jd[0]
        all_t0.append(theta[0][0])
        theta[1][0]=theta[1][0]-alpha*jd[1]
        all_t1.append(theta[1][0])
        all_costs.append(computeCost(X, y, theta))
        i=i+1
        print("t0=",theta[0])
        print("t1=",theta[1])
    return theta
print(gradientDescent(X, y, theta, alpha, iterations))
print(all_t0)
print(all_t1)
print(all_costs)
```

Dans cette fonction nous avons implémenter **la descente du gradient** ci-dessous vu en cours

Algorithme de la descente du gradient:

Répéter

$$\theta_j \leftarrow \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1) \quad \text{(pour j=0 et j=1 } \textbf{simultanément}\text{)}$$

Jusqu'à convergence

Nous lui donnerons en entrée **X ,y** , **alpha (taux d'apprentissage)**,et **itérations** qui est le nombre d'itérations qu'on fixe pour **la convergence** ,tant qu'on n'a pas atteint le nombre d'itérations fixer ou la dérivée est assez petite on mettra à jour à chaque fois θ_0 et θ_1:

theta[0,0] = theta[0,0] - alpha * jd[0]

theta[1,0] = theta[1,0] -alpha * jd[1]

jd est un tableau à 2 cases qui contiendra les dérivés de :**jd[0]** (dérivé de **J(theta0,theta 1**) par rapport **à theta 0** qu'on calcul grâce à un appelle à la fonction **j**) et : **jd[1]** (dérivé de **J(theta0,theta 1)** par rapport **à theta 1** qu'on calcul grâce à un appelle à la fonction **j**) .

on l'applique jusqu'à divergence (atteindre **le minimum local**) .

Initialisation de θ_0 et θ_1 ,calcul du cout initial et appel de la fonction de calcul du gradient :

Descente du Gradient : Appel des fonctions

Initialisation de θ_0 et θ_1

```
theta = np.zeros((2, 1))
print(theta)
```

```
[[0.]
 [0.]]
```

Calculer le cout initial

```
initialCost=computeCost(X, y, theta)
```

Appel des la fonction de calcul du gradient

```
# paramètres
iterations = 1500;
alpha = 0.01;
# Appel
theta = gradientDescent(X, y, theta, alpha, iterations);
```

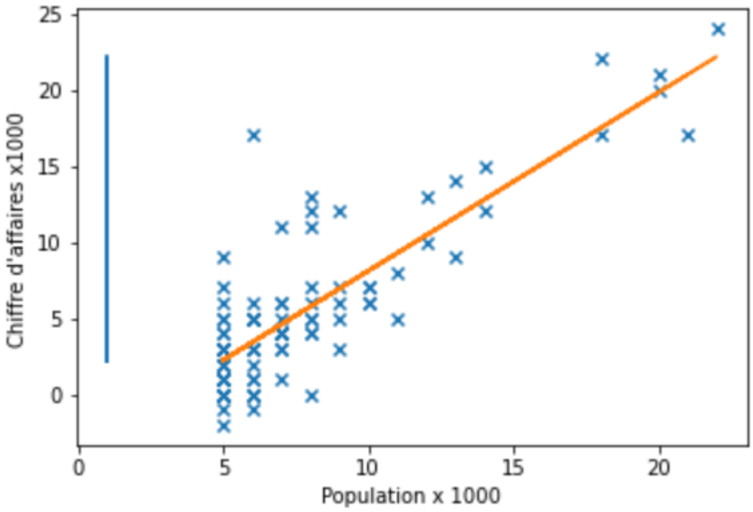
Theta est un vecteur de **2 lignes** et une colonne qui contiendra nos θ_0 et θ_1 , **initialCost** pour calculer le cout initial , on fixe le **nombre d'iterations** à **1500** et **le taux d'apprentissage alpha** à **0.01** ensuite on appelle la fonction de **calcul du gradient** qu'on a défini précédemment et regardons le résultat :

On observe à chaque fois le changement des valeurs de nos **θ_0** et **θ_1** jusqu'à trouvé la valeur du **minimum local .**

```
t1= [1.19167148]
[array([0.0022128]), array([-0.00023077])]
t0= [-3.79741695]
t1= [1.19167379]
[array([0.00220849]), array([-0.00023032])]
t0= [-3.79743903]
t1= [1.19167609]
[array([0.00220419]), array([-0.00022987])]
t0= [-3.79746108]
t1= [1.19167839]
[array([0.0021999]), array([-0.00022942])]
t0= [-3.79748307]
t1= [1.19168068]
[array([0.00219562]), array([-0.00022897])]
t0= [-3.79750503]
t1= [1.19168297]
[array([0.00219134]), array([-0.00022853])]
t0= [-3.79752694]
t1= [1.19168526]
```

Traçage de la fonction du coût :

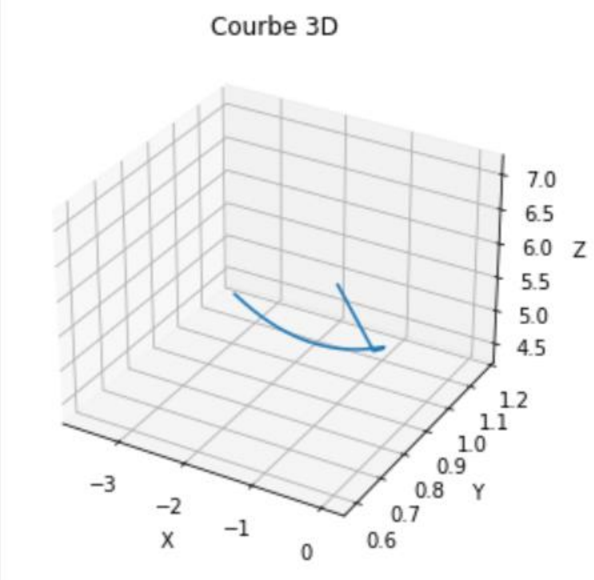
$h_\theta(x) = x_0\theta_0 + \theta_1 x_1$



Traçage de la fonction du coût en fonction de theta0 et theta1 :

Traçage du coût en fonction de theta0 et theta1

```
fig = plt.figure()
ax = fig.gca(projection='3d')  # Affichage en 3D
ax.plot(all_t0, all_t1, all_costs, label='Courbe')  # Tracé de la courbe 3D
plt.title("Courbe 3D")
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
plt.tight_layout()
plt.show()
```



Prédire les valeurs de y :

Prédire des valeurs de y

```
# Predire pour une opulation = 35,000 et 70,000
predict1 = np.matmul([1, 3.5],theta);
predict1
```

```
array([0.36662291])
```

```
predict2 = np.matmul([1, 7],theta);
predict2
```

```
array([4.54140048])
```

Partie 2 : Régression linéaire à plusieurs variables

Dans cette partie nous allons prédire le prix d'une maison à partir de la superficie et le nombre de chambres (plusieurs caractéristiques).
nous allons étudier un ensemble de données avec le prix d'une maison **(y)** et les caractéristiques(futures) : la superficie et le nombre de chambres **(X_0,X_1,X_2)** que nous avons généraliser **(X_0,X_1,X_2,X_4 .....X_n)** dans l'implémentation de notre algorithme.
Nous allons changer de **data set** :

```
# données
dataMulti = np.genfromtxt('dataMulti.csv', delimiter=',', dtype=int)
dataMulti.shape
```

```
(47, 3)
```

---

47 : signifie le nombre de données (ou exemples) dans le fichier (***datMultia.csv***) c.à.d. il traite ***47 maisons***
3: représente 3 colonnes **(2 caractéristiques)** dans le fichier
(colonne 1 : la superficie X_1
et
colonne 2: le nombre de chambres X_2
et
colonne 3 :prix de la maison y)

Créer X et y :

X est un tableau à **3 colonnes** qui contiendra tous les caractéristiques des maisons de notre fichier **dataMulti** (**1 ere colonne** X0=1,

2eme colonne X1=superficie,

3eme colonne X2=nombre de chambres) et **y** est **un tableau** qui contient toutes les valeurs y (prix de la maison)

Nous allons afficher ci-dessous X et y :

```python
# d'abord créer X et y
intercept=np.ones((dataMulti.shape[0],1))
X=np.column_stack((intercept,dataMulti[:,1],dataMulti[:,2]))
print(X)
y = dataMulti[:, 2]
print(y)
```

```
[[1.000e+00 3.000e+00 3.999e+05]
 [1.000e+00 3.000e+00 3.299e+05]
 [1.000e+00 3.000e+00 3.690e+05]
 [1.000e+00 2.000e+00 2.320e+05]
 [1.000e+00 4.000e+00 5.399e+05]
 [1.000e+00 4.000e+00 2.999e+05]
 [1.000e+00 3.000e+00 3.149e+05]
 [1.000e+00 3.000e+00 1.990e+05]
 [1.000e+00 3.000e+00 2.120e+05]
 [1.000e+00 3.000e+00 2.425e+05]
 [1.000e+00 4.000e+00 2.400e+05]
 [1.000e+00 3.000e+00 3.470e+05]
 [1.000e+00 3.000e+00 3.300e+05]
 [1.000e+00 5.000e+00 6.999e+05]
 [1.000e+00 3.000e+00 2.599e+05]
 [1.000e+00 4.000e+00 4.499e+05]
 [1.000e+00 2.000e+00 2.999e+05]
 [1.000e+00 3.000e+00 1.999e+05]
 [1.000e+00 4.000e+00 5.000e+05]
```

```
 [1.000e+00 3.000e+00 5.799e+05]
 [1.000e+00 4.000e+00 2.859e+05]
 [1.000e+00 3.000e+00 2.499e+05]
 [1.000e+00 3.000e+00 2.299e+05]
 [1.000e+00 4.000e+00 3.450e+05]
 [1.000e+00 4.000e+00 5.490e+05]
 [1.000e+00 4.000e+00 2.870e+05]
 [1.000e+00 2.000e+00 3.685e+05]
 [1.000e+00 3.000e+00 3.299e+05]
 [1.000e+00 4.000e+00 3.140e+05]
 [1.000e+00 3.000e+00 2.990e+05]
 [1.000e+00 2.000e+00 1.799e+05]
 [1.000e+00 4.000e+00 2.999e+05]
 [1.000e+00 3.000e+00 2.395e+05]]
[399900 329900 369000 232000 539900 299900 314900 199000 212000 242500
 240000 347000 330000 699900 259900 449900 299900 199900 500000 599000
 252900 255000 242900 259900 573900 249900 464500 469000 475000 299900
 349900 169900 314900 579900 285900 249900 229900 345000 549000 287000
 368500 329900 314000 299000 179900 299900 239500]
```

Redéfinissions de la fonction de cout :

1-Non vectorisée :

```
theta = np.zeros((X.shape[1], 1))
# redéfinissez vos fonctions de coût si cela est necessaire
def computeCostNonVectt(X, y, theta):
    jt=0
    for i in range(len(X)):
        j=0
        for k in range(X.shape[1]):
            j=j+theta[k][0]*X[i,k]
        j=j-y[i]
        j=j**2
        jt=jt+j
    jt=jt/(2*(len(X)))
    return jt
print(computeCostNonVectt(X, y, theta))
```

```
65591585744.680855
```

Cette fonction nous permettra de calculer **le cout** vu en cours on a implémenté la fonction de cout pour plusieurs futures ci-dessous :

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)})\right) - y^{(i)})^2$$

Avec :

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \ldots.. + \theta_n x_n$$

$X_0 = 1$

On a modélisé nos **θ_0 θ_1 θ_2 θ_3.... θ_n** par un vecteur à **k lignes** la 1ere ligne représente θ_0 et la 2eme ligne θ_1 …. **theta[0,0] = θ_0 theta[1,0] = θ_1 theta[2,0] = θ_2 …. theta[n,0] = θ_n** .nous allons d'abord parcourir tous les **X^i** et calculer le cout grâce à la formule et les sommer pour avoir **le cout J(θ).**

Si on applique l'algorithme générale sur notre exemple on a utiliser 2 futures donc on aura pour les **X :X_0,X_1,X_2** et pour les **θ : θ_0 θ_1 θ_2 .**

Mise à l'échelle des données :

Code :

```
print(X)
def mis(X):
    xms=[]
    for j in range(1,(X.shape[1])):
        print(j)
        for i in range(len(X)):
            X[i,j]=(X[i,j]-np.amin(X[:,j]))/(np.amax(X[:,j])-np.amin(X[:,j]))
mis(X)
print(X)
```

Résultat :les X mis à l'echelle.

```
[[1.         0.5        0.43396226]
 [1.         0.55555556 0.47135272]
 [1.         0.55555556 0.52721788]
 [1.         0.33333333 0.33147551]
 [1.         0.78571429 0.77139581]
 [1.         0.78571429 0.42848951]
 [1.         0.57142857 0.44992116]
 [1.         0.57142857 0.28432599]
 [1.         0.57142857 0.30290013]
 [1.         0.57142857 0.3464778 ]
 [1.         0.78571429 0.34290586]
 [1.         0.57142857 0.49578491]
 [1.         0.57142857 0.47149571]
 [1.         1.         1.        ]
 [1.         0.72727273 0.43388955]
 [1.         1.         0.75108502]
```

Nous avons appliqué la formule :
Pour $\mathbf{X_1}$ nous avons pour chaque $\mathbf{X^i_1}$ on lui affecte $\mathbf{X^i_1}$ **-le minimum des X^i_1 / (le minimum des X^i_1 + le maximum des X^i_1)**
Pour $\mathbf{X_2}$ nous avons pour chaque $\mathbf{X^i_2}$ on lui affecte $\mathbf{X^i_2}$ **-le minimum des X^i_2 / (le minimum des X^i_2 + le maximum des X^i_2)**

Normalisation des données :
Code :

```
print(X)
def normalisation(X):
    xms=[]
    for j in range(1,(X.shape[1])):
        print(j)
        for i in range(len(X)):
            X[i,j]=(X[i,j])/(np.amax(X[:,j])-np.amin(X[:,j]))
normalisation(X)
print(X)
```

Résultat :

```
[[1.         0.75       0.7545283 ]
 [1.         0.70588235 0.47135356]
 [1.         0.69863014 0.52721853]
 [1.         0.46496815 0.33147615]
 [1.         0.88202247 0.77139628]
 [1.         0.88202247 0.42848999]
 [1.         0.66151685 0.44992163]
 [1.         0.66151685 0.28432647]
 [1.         0.66151685 0.30290054]
 [1.         0.66151685 0.34647821]
 [1.         0.88202247 0.34290627]
 [1.         0.66151685 0.49578531]
 [1.         0.66151685 0.47149612]
 [1.         1.10252809 1.00000041]
 [1.         0.84864865 0.43389002]
```

Nous avons appliqué la formule : avec moyenne =0

Pour $\mathbf{X_1}$ nous avons pour chaque $\mathbf{X^i_1}$ on lui affecte X^i_1 -**la moyenne des $\mathbf{X^i_1}$ / (le minimum des $\mathbf{X^i_1}$ + le maximum des $\mathbf{X^i_1}$)**

Pour $\mathbf{X_2}$ nous avons pour chaque X^i_2on lui affecte $\mathbf{X^i_2}$ -**le moyenne des $\mathbf{X^i_2}$ / (le minimum des $\mathbf{X^i_2}$ + le maximum des $\mathbf{X^i_2}$)**

Descente du gradient :

Code :

```python
theta = np.zeros((3, 1))
def j(theta,X,y):
    theta_new=[]
    for l in range(X.shape[1]):
        print(l)
        j=0
        for i in range(len(X)):
            t=0
            for k in range(X.shape[1]):
                t=t+theta[k][0]*X[i,k]
            t=t-y[i]
            j=j+t*X[i,l]
        j=j/len(X)
        theta_new.append(j)
    print("n",theta_new)
    return theta_new
# theta=j(theta,X,y)
print(theta)
def gradientDescent(X, y, theta, alpha, iterations):
    i=0
    while(i<iterations):
        jd=[]
        jd=j(theta,X,y)
        if((np.isneginf(jd).any())|(np.isinf(jd).any())):
            print("hello")
            break
        print("jd",jd)
        for k in range(len(jd)):
            theta[k][0]=theta[k][0]-alpha*jd[k]
            all_t0.append(theta[k][0])
        all_costs.append(computeCostNonVectt(X, y, theta))
        i=i+1
        print("t0=",theta[0][0])
        print("t1=",theta[1][0])
        print("t2=",theta[2][0])
        print(theta)
    print(theta)
start = time.time()
print(gradientDescent(X, y, theta, alpha, iterations))
end = time.time()
print(end - start)
```

Dans cette fonction nous avons implémenter **la descente du gradient** ci-dessous vu en cours

L'algorithme de descente du gradient devient donc:

$\Theta_j \leftarrow \Theta_j - \alpha \frac{\partial}{\partial \Theta_j} J(\Theta)$ (pour tous les j **simultanément**)

Jusqu'à convergence

Nous lui donnerons en entrée **X** ,**y** , **alpha (taux d'apprentissage),**et **itérations** qui est le nombre d'itérations qu'on fixe pour **la convergence** ,tant qu'on n'a pas atteint le nombre d'itérations fixer ou la dérivée est assez petite on mettra à jour à chaque fois θ_0 θ_1 θ_2 …θ_n:

theta[0,0] = theta[0,0] - alpha * jd[0]
theta[1,0] = theta[1,0] -alpha * jd[1]
.
.
theta[n,0] = theta[n,0] -alpha * jd[n]
dans notre exemple on aura θ_0 θ_1 θ_2:
theta[0,0] = theta[0,0] - alpha * jd[0]
theta[1,0] = theta[1,0] -alpha * jd[1]
theta[3,0] = theta[3,0] - alpha * jd[3]

jd est un tableau à **n** cases qui contiendra les dérivés de :**jd[0]** (dérivé de **J(theta0,theta 1)** par rapport à **theta 0** qu'on calcul grâce à un appelle à la fonction **j**) et : **jd[1]** (dérivé de **J(theta0,theta 1)** par rapport à **theta 1** qu'on calcul grâce à un appelle à la fonction **j**)
jd[2] (dérivé de **J(theta0,theta 1,theta2)** par rapport à **theta 2** qu'on calcul grâce à un appelle à la fonction **j** ….)

on l’applique jusqu’à **divergence** (atteindre **le minimum local**) .
on observe les valeurs de **theta** jusqu’à divergence;

```
n [-340412.7659574468, -1120368.085106383, -131183171489.36171]
jd [-340412.7659574468, -1120368.085106383, -131183171489.36171]
t0= 3404.1276595744685
t1= 11203.680851063831
t2= 1311831714.8936172
[[3.40412766e+03]
 [1.12036809e+04]
 [1.31183171e+09]]
0
1
2
n [446564262236146.4, 1469734385406557.2, 1.7209024470260072e+20]
jd [446564262236146.4, 1469734385406557.2, 1.7209024470260072e+20]
t0= -4465642618957.336
t1= -14697343842861.89
t2= -1.7209024457141755e+18
[[-4.46564262e+12]
 [-1.46973438e+13]
```

Temps d’exécution avec normalisation :

```
n [2008.4454396580907, 1316.3468210755511, -4732.784057076915]
jd [2008.4454396580907, 1316.3468210755511, -4732.784057076915]
t0= 116610.22941658067
t1= 115103.38031014006
t2= 205339.26009478178
[[116610.22941658]
 [115103.38031014]
 [205339.26009478]]
[[116610.22941658]
 [115103.38031014]
 [205339.26009478]]
```

```
8.043315649032593
```

Temps d’exécution avant normalisation :

```
38.734883069992065
```

Vérification de l'implementation

Comparer vos algorithmes à ceux de scikitlearn

```
]: from sklearn.linear_model import SGDRegressor

   sgd_reg = SGDRegressor(max_iter=1500,alpha = 0.01)

   sgd_reg.fit(X,y.ravel()) #ravel flattens the array. Similar to reshape(-1)
   print(sgd_reg.coef_)
```

```
[-1.19522536e+10 -8.91478918e+10 -1.95276361e+14]
```

Temps d’execution avec scikitlearn :

```
0.024851560592651367
```

FIN