Baptiste Cojean & Gatien Auffret

ESIR 1 INFO

# PROJ-MATH Sujet n°2

*24/02/2025*

## Sommaire :

# I - Introduction

Ce travail a pour objectif d'implémenter un algorithme de descente de gradient pour la régression linéaire. L'étude commence par la génération de données selon un modèle linéaire avec du bruit. Ensuite, on utilise une fonction de perte pour mesurer la précision du modèle. On ajuste les paramètres du modèle et on minimise la perte grâce à l'algorithme de descente de gradient. Différentes expériences sont testées en modifiant certains paramètres comme le taux d'apprentissage ou le nombre d'itérations. Une variante stochastique de l'algorithme est aussi testée. Ce compte rendu présente la démarche suivie, les résultats obtenus et leur analyse.

# II - Generate data

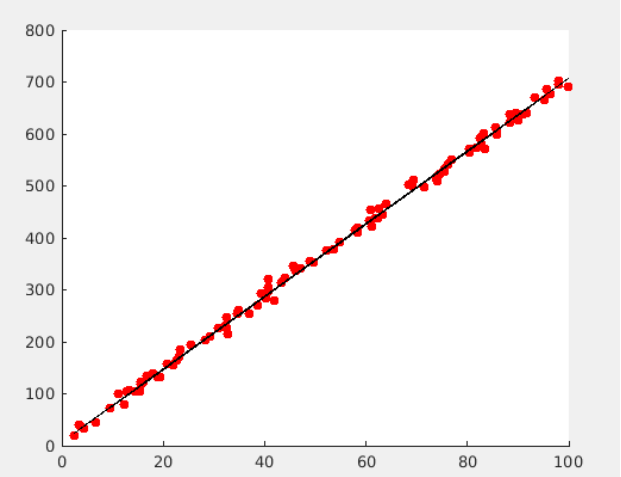
On génère un ensemble de points aléatoires x,y obtenu par le biais d'un modèle linéaire bruité :

$$f_w : x_i \rightarrow (w_0 x_i + w_1) + noise = y_i$$

où les valeurs du vecteur $w = (w_0, w_1)$ sont les paramètres du modèle linéaire et $noise$ est un bruit ajouté.

On obtient les données suivantes :

```
% Nombre de points de données
N = 100;

X = unifrnd(0, 100, [1, N]);

w0 = randi([-10, 10]);
w1 = randi([-10, 10]);

% Création de y = w0 * x + w1 + un bruit gaussien ou normal
Y = w0 * X + w1 + normrnd(0, 10, [1, N]);

% Tracer les données pour voir si elles semblent linéaires
scatter(X, Y, 'filled', 'r');
hold on;
plot(X, w0 * X + w1, 'k');
hold off;
```




Les paramètres d'un modèle linéaire sont $w_0$ (le coefficient directeur ou pente), $w_1$ (l'ordonnée à l'origine) et le bruit (noise)

## III - Loss function

Nous allons évaluer la performance d'un modèle linéaire $f_w$ grâce à des loss functions :

- $SSR(w,x,y) = \sum_{i=1}^{N} (\hat{y}_i - y_i)^2$, où $\hat{y}_i = f_w(x_i)$
- $MSE(w,x,y) = \frac{1}{N} SSR(w,x,y) = \frac{1}{N} \sum_{i=1}^{N} (\hat{y}_i - y_i)^2$, où $\hat{y}_i = f_w(x_i)$

.

On implémente une fonction de perte puis on évalue $f_w$ grâce à notre jeu de donnée :

```
%% SSR
yt = X * w0;
SSR = sum((yt - Y).^2);
disp("Erreur SSR : "+SSR);
%% MSE
MSE = SSR / length(Y);
disp("Erreur MSE : "+MSE);
```

```
Erreur SSR : 13802.1528
Erreur MSE : 138.0215
```

## IV - Gradient descent

Maintenant, on va chercher le modèle qui génère des valeurs $\hat{y}$ le plus proches des valeurs $y$ pour minimiser la loss. On cherche donc $grad(loss(w, x, y)) = 0$ . On implémente une fonction qui calcule le gradient de notre loss :

$$\nabla f(w_1, \cdots, w_n) = (\frac{\delta f(w_1, \cdots, w_n)}{w_1}, \cdots, \frac{\delta f(w_1, \cdots, w_n)}{w_n}) \text{ et } \nabla \sum_i f(x_i) = \sum_i \nabla f(x_i),$$

```
Y_pred = w0 * X + w1;
dL_dw0 = (-2/N) * sum(X .* (Y - Y_pred));
dL_dw1 = (-2/N) * sum(Y - Y_pred);
grad = [dL_dw0; dL_dw1];
disp("gradient : "+grad);
```

```
"gradient : 69.7944"
"gradient : 0.478382"
```

Ensuite, on implémente l'algorithme de descente de gradient avec comme paramètre le taux d'apprentissage μ et un nombre d'itérations n en sachant que :

L'idée de l'algorithme de descente de gradient tiens dans le fait que les fonctions de loss utilisée soient des fonctions convexes. Le calcul de gradient nous indique quelles sont les modifications à effectuer sur les paramètre $w$ du modèle $f_w$ évalué afin de minimiser la loss.

- si $\nabla[w_i] > 0$ alors j'aurais une meilleure prédictions si je diminue la valeur de $w_i$.
- si $\nabla[w_i] < 0$ alors j'aurais une meilleure prédictions si j'augmente la valeur de $w_i$.
- si $\nabla[w_i] = 0$ alors $w_i$ est la meilleure valeur possible pour le paramètre $w$ car il minimise la fonction de loss.

```
%% descente gradient
history = zeros(1000, 3);
mu = 0.000001;
for i = 1:1000
    Y_pred = w0 * X + w1;
    dL_dw0 = (-2/N) * sum(X .* (Y - Y_pred));
    dL_dw1 = (-2/N) * sum(Y - Y_pred);
    w0 = w0 - mu * dL_dw0;
    w1 = w1 - mu * dL_dw1;
    MSE = sum((Y_pred - Y).^2) / N;
    history(i, :) = [i, w0, w1];
    if mod(i, 100) == 0
        disp("Itération : "+ i);
        grad = [dL_dw0; dL_dw1];
        disp("gradient : "+grad);

    end
end
```

```
Itération : 800
    "gradient : 0.79666"
    "gradient : -0.18989"

Itération : 900
    "gradient : 0.44105"
    "gradient : -0.19549"

Itération : 1000
    "gradient : 0.24482"
    "gradient : -0.19857"
```

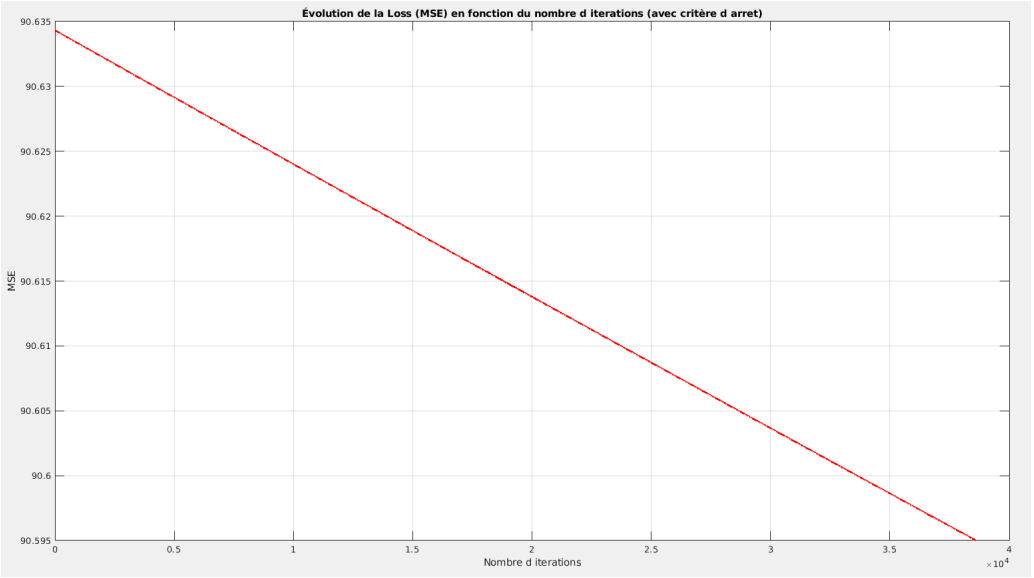
Désormais, on implémente une autre version de l'algorithme où les paramètres sont le taux d'apprentissage $\mu$ et une valeur $\epsilon$.

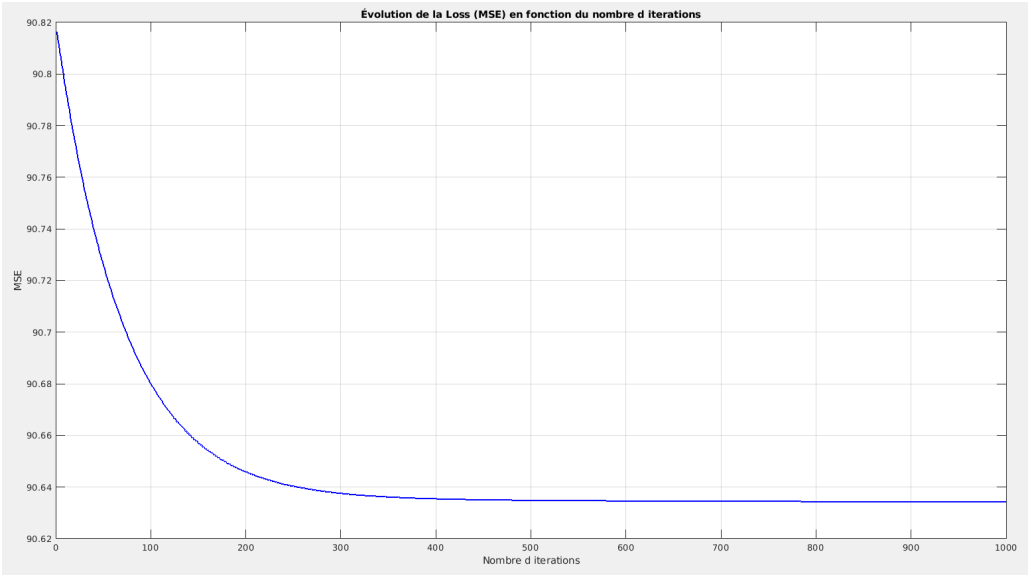
```
%% descente gradient epsilon


history2 = [];
prev_MSE = inf;
iteration = 0;
epsilon = 0.000001;

while true
    Y_pred = w0 * X + w1;

    dL_dw0 = (-2/N) * sum(X .* (Y - Y_pred));
    dL_dw1 = (-2/N) * sum(Y - Y_pred);

    w0 = w0 - mu * dL_dw0;
    w1 = w1 - mu * dL_dw1;

    MSE = sum((Y_pred - Y).^2) / N;

    history2 = [history2; iteration, w0, w1, MSE];

    if mod(i, 100) == 0
        disp("Itération : "+ iteration);
        grad = [dL_dw0; dL_dw1];
        disp("gradient : "+grad);

    end

    if abs(prev_MSE - MSE) < epsilon
        break;
    end
    prev_MSE = MSE;
    iteration = iteration + 1;
end
```

```
Itération : 0
    "gradient : 0.13531"
    "gradient : -0.1181"

Itération : 1
    "gradient : 0.13455"
    "gradient : -0.11812"
```

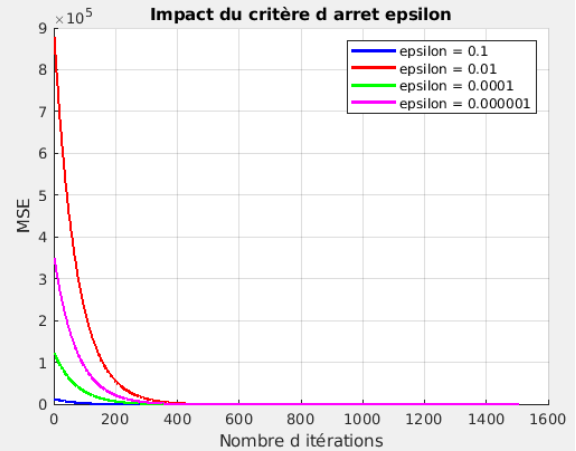
## V - Experiment

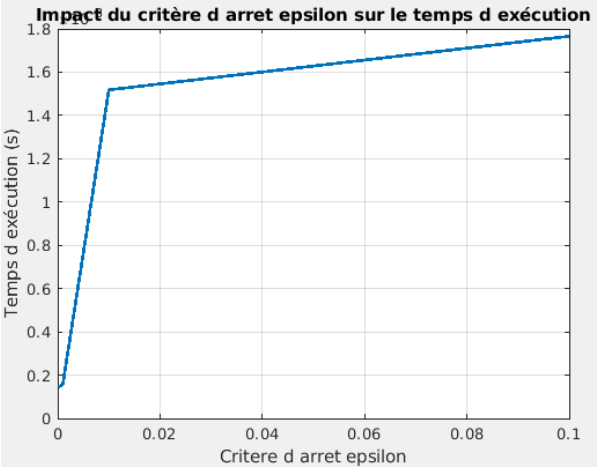
On visualise l’évolution de loss en fonction de nombre d'itérations.



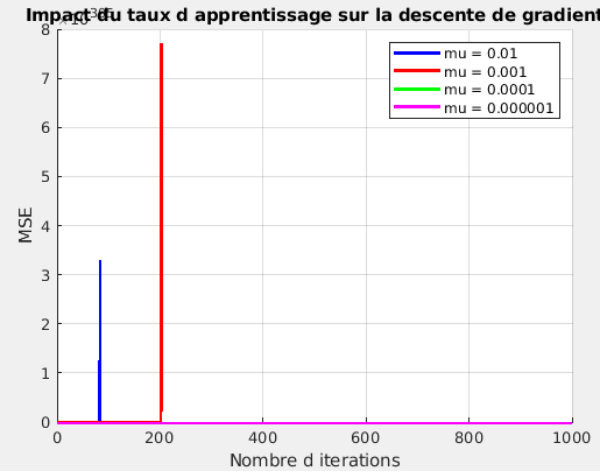



Plus il y a d'itérations moins la loss est élevée, mais il semble y avoir une limite.

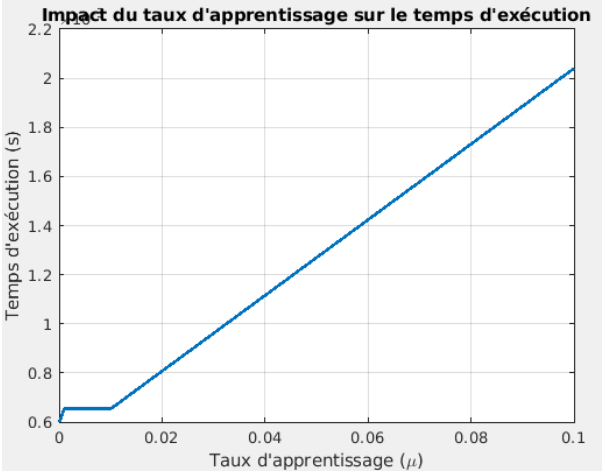



Plus $\epsilon$ est grand, plus l'algorithme s'arrête rapidement. Cependant, il risque de ne pas atteindre un minimum suffisamment bas de l'erreur. À l'inverse, plus $\epsilon$ est petit, plus l'algorithme effectue d'itérations avant de s'arrêter, garantissant une convergence plus précise mais au prix d'un temps d'exécution plus long.

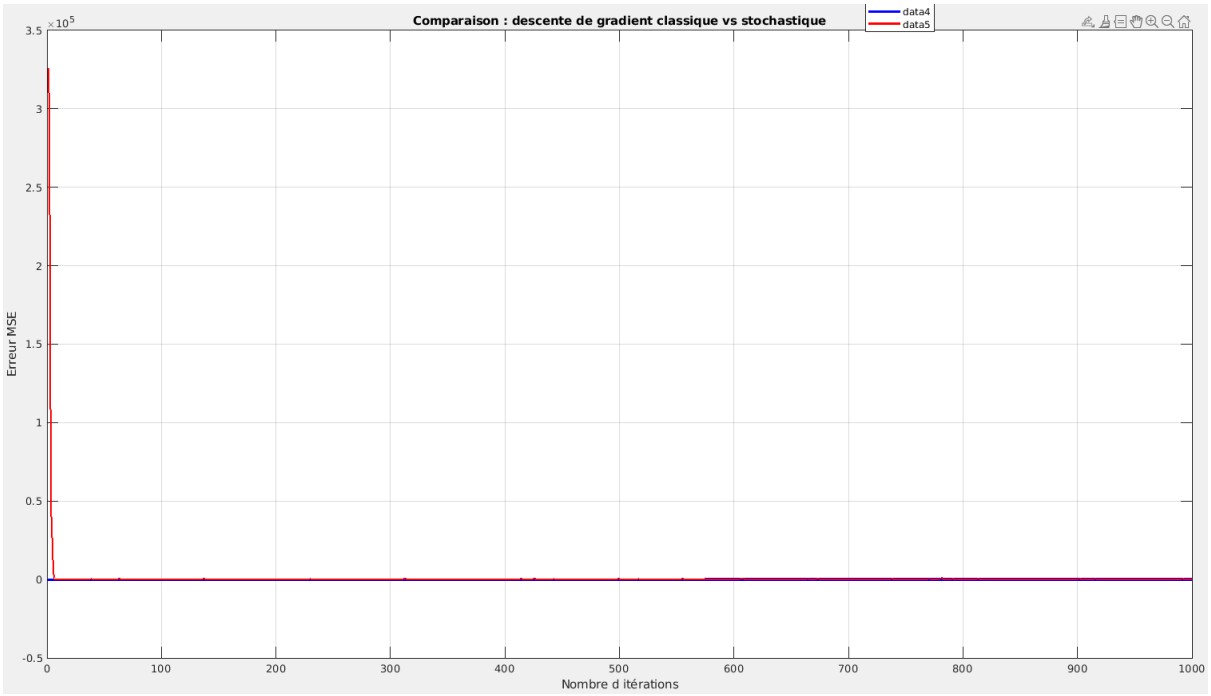


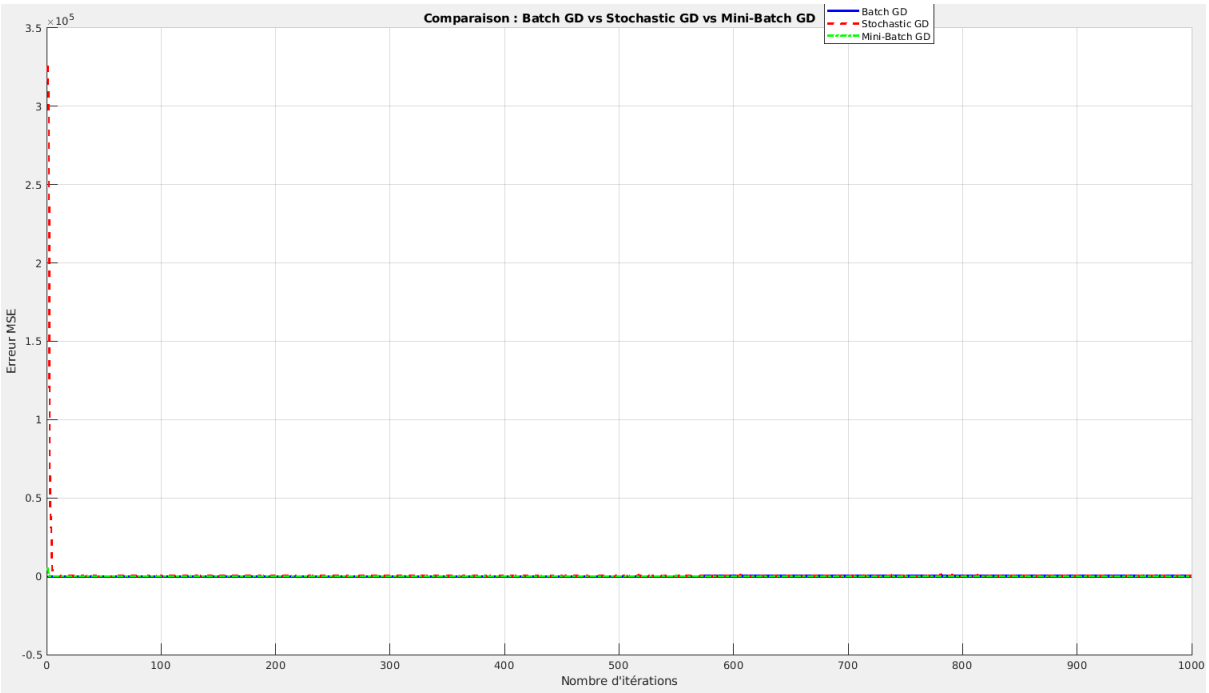

Un taux d'apprentissage trop élevé peut provoquer une divergence ou une convergence instable. Un taux trop faible conduit à une convergence très lente, nécessitant un grand nombre d'itérations pour atteindre un minimum satisfaisant.




Plus le taux d’apprentissage est élevé, plus le temps d'exécution est long.

## V - Stochastic gradient descent

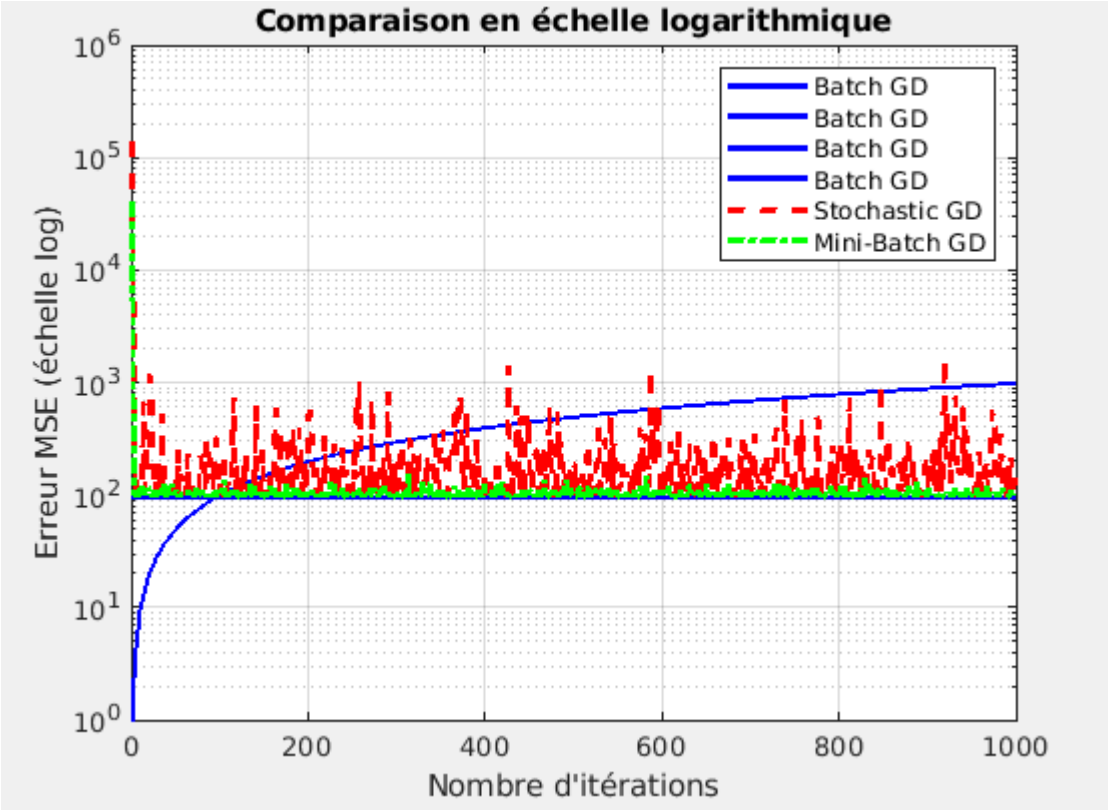


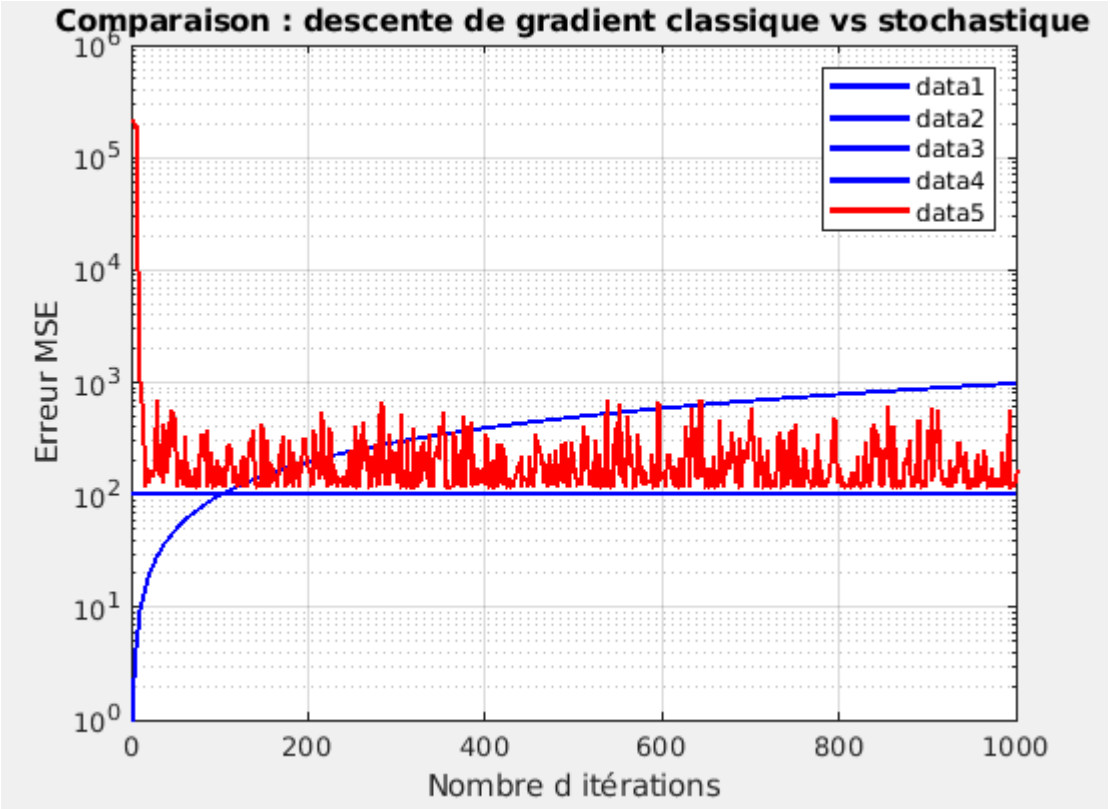



```
Temps d'exécution de SGD : 0.000811 secondes
```

```
Temps d'exécution de Mini-Batch GD : 0.00347 secondes
```

Les résultats n'étant pas très claire, on change d'échelle et on passe en logarithmique, on remarque des courbes plus lisibles :






## VII - Conclusion

Ce travail a permis de comprendre et d'implémenter l'algorithme de descente de gradient pour un modèle de régression linéaire. L'utilisation de fonctions de perte a permis l'évaluation des performances du modèle. Les expériences réalisées ont montré l'influence du taux d'apprentissage et du nombre d'itérations sur la convergence de l'algorithme. La version stochastique a offert une alternative plus rapide mais moins stable. Ces observations montrent l'importance du choix des paramètres et de la méthode d'optimisation pour obtenir un modèle efficace.