Rapport de stage

Clement Lotteau

Juin 2020

Résumé

Certains systèmes quantiques ne peuvent être étudiés analytiquement par la résolution de l'équation de Schrödinger indépendante du temps. Pour cette raison, les méthodes numériques de résolution de cette équation sont centrales en physique moderne car elles nous permettent d'étudier des systèmes non idéaux. Ce rapport de stage présente les résultats de deux méthodes de résolution : un algorithme de diffusion Runge-Kutta d'ordre 4 (RK4), et une approche stochastique originale reposant sur les réseaux de neurones. La contrainte de 6 pages nous a poussé à mettre l'accent sur la seconde méthode et ses résultats, mais plus de détails sur les deux méthodes sont disponibles ici : [9]. Les caractéristiques de la machine utilisée pour la méthode stochastique sont disponibles en bibliographie [5] (J'ai mis un lien vers un google Colab pour le moment mais je dois détailler les caractéristiques quelque part au cas où celles de Colab changent). Des tutoriels d'initiation aux réseaux de neurones réalisés par Colin Bernet sont disponibles en bibliographie : [2], [3] et [4].

Table des matières

1 Introduction

Le but de ce stage était de calculer numériquement la fonction d'onde et l'énergie d'une particule piégée dans un potentiel harmonique. Pour ce faire, j'ai exploré deux méthodes : un algorithme Runge-Kutta d'ordre 4 permettant de trouver les états liés et leurs énergie, et une méthode de minimisation stochastique de l'énergie utilisant un réseau de neurones permettant de trouver l'état fondamental. Dans les deux cas, on cherche à résoudre l'équation de Schrödinger indépendante du temps [1] :

$$\left[\frac{-\hbar^2}{2m}\frac{d^2}{dx^2} + \frac{1}{2}m\omega^2 x^2\right]\psi(x) = E\psi(x) \qquad (1)$$

Pour trouver l'énergie de $\psi(x)$, on multiplie (1) à gauche par $\psi(x)$ et on intègre par partie :

$$E = \frac{\frac{-\hbar^2}{2m}\left(\left[\psi\psi'\right]_a^b - \int_a^b |\psi'|^2 dx\right) + \frac{1}{2}m\omega^2 \int_a^b x^2|\psi|^2 dx}{\int_a^b |\psi|^2 dx} \qquad (2)$$

On obtient les solutions analytiques en intégrant de $-\infty$ à $+\infty$:

$$E_c = \frac{\hbar\omega}{4} \quad ; \quad E_p = \frac{\hbar\omega}{4} \quad ; \quad E_{totale} = \frac{\hbar\omega}{2} \qquad (3)$$

2 Méthode Runge-Kutta

La méthode RK d'ordre 4 utilisée nous permet de trouver les énergies (table 1) et les fonctions d'ondes (figure 1) des états liés.

État	Énergie
0	-2.7184
1	-2.1545
2	-1.5849
3	-0.9935
4	-0.3488

TABLE 1: Énergies trouvées grâce à la propagation RK4

On observe que, contrairement à la théorie, l'écart entre les énergies n'est pas constant. La figure 1 montre aussi que les fonctions d'onde ne tendent pas vers 0 lorsque l'énergie augmente. Ces deux problèmes sont à priori dus à la distance de propagation qui est trop petite en dehors du potentiel. Des informations supplémentaires et mon utilisation de RK4 sur un puit carré sont disponibles sur mon GitHub [9].

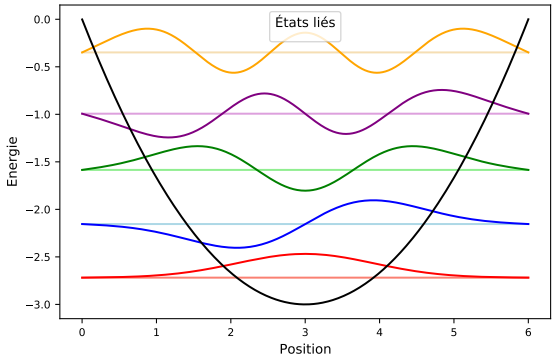

FIGURE 1: Tracer des états liés, les fonctions d'onde sont alignées sur leurs énergies.

3 Réseaux de neurones

3.1 Concept et vocabulaire

On cherche ici à ce que notre réseau reproduise la fonction discrète suivante :

$$f(x) = \begin{cases} 1 & quand \quad x = 0 \\ 7 & quand \quad x = 1 \\ 3 & quand \quad x = 2 \end{cases} \qquad (4)$$

On peut représenter cette fonction par deux listes de trois nombres : $x = [0, 1, 2]$ et $y = [1, 7, 3]$. On va entrainer notre réseau pour qu'il reproduise la liste y lorsqu'on lui demande de faire une prédiction à partir de la liste x. Pour ce faire, on commence par initialiser le réseau avec des paramètres aléatoires puis on lui fournit x et y.

Entrainement :

Epoch 1 : Le réseau fait une première prédiction $z = [32, -9, 64]$ éloignée de la liste attendue. La *fonction de coût* calcule ensuite le *loss*, l'écart quadratique moyen entre y et z. On calcule ensuite le gradient de la fonction de coût par rapport aux

paramètres du réseau pour savoir comment les modifier afin de minimiser le loss. On modifie les paramètres en conséquence, fin du premier epoch.

Epoch 2 : Le réseau prédit $z = [11, 3, 9]$. La prédiction est meilleure grâce au premier entrainement mais doit être améliorée. On calcule le loss, le gradient, on modifie les paramètres du réseau et on recommence le processus pour l'epoch 3.

Ces calculs sont gérés automatiquement par Keras [8], la librairie Python que nous utilisons ici.

Géométries :

Les réseaux utilisés ici prenennt une discrétisation de l'espace (x) en entrée et renvoient une fonction d'onde (y) en sortie. Entre l'entrée et la sortie, les données sont propagées à travers des "couches cachées" dont on choisit le nombre de neurones. Par la suite, j'utiliserai le mot "couche" pour désigner les couches cachées des réseaux. Aussi, un réseau 200x200 par exemple désignera un réseau à deux couches cachées, toutes deux composées de 200 neurones.

3.2 Fit d'une gaussienne

On cherche ici à faire reproduire l'état fondamental d'un oscillateur harmonique à une dimension. On dispose d'une liste x discrétisée de -5 à 5 sur 10.001 points, et d'une liste y calculée à partir de la solution analytique de l'équation (1) page 1 pour l'état fondamental [1]. m, $\hbar$ et ω sont tous égaux à 1 par la suite.

$$\psi(x) = \left(\frac{m\omega}{\pi\hbar}\right)^{\frac{1}{4}} e^{\frac{-m\omega x^2}{2\hbar}} \tag{5}$$

Une fonctionnalité de Keras nous permet de mesurer le loss à chaque epoch. On voit sur la figure 2 que le minimum du loss est atteint autour du 30^e epoch. Le réseau continue à modifier ses paramètres lors des epochs suivants et se déplace autour du minimum de la fonction de coût. Le loss augmente (les prédictions s'éloignent de la cible), c'est le sur-apprentissage.

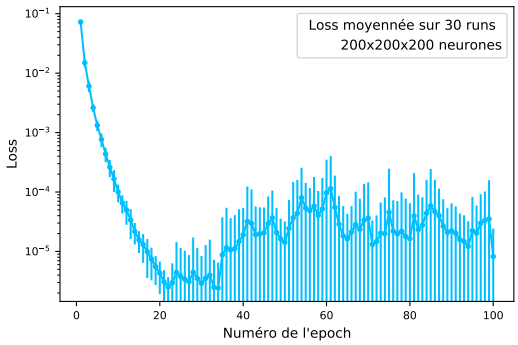

FIGURE 2: Moyenne sur 30 runs. Barres d'erreur avec l'écart-type des loss pour chaque run. Réseau : 200x200x200 neurones, 50 batch size.

Une étude du loss en fonction des epochs a été faite pour comparer l'impact de la géométrie sur la minimisation du loss [9]. On y voit que l'ajout de non linéairité (plusieurs couches) permet d'atteindre le minimum du loss en moins d'epochs que pour un réseau à une couche. On y voit aussi que l'augmentation du nombre de paramètres permet d'obtenir des loss plus bas mais que les oscillations autour du minimum de la fonction de coût ont une plus grande amplitude qu'avec des réseaux moins denses.

3.3 Calcul de l'énergie d'une prédiction du réseau

Une fois l'état fondamental fitté par le réseau, on peut en extraire une énergie avec l'équation (2) page 1. Une première tentative d'extraction nous avait posé problème du fait que les prédictions du réseau sont des morceaux de droites. Un bon apprentissage permet de lisser le résultat mais la dérivée de la prédiction restait néanmoins discontinue. Nous avons extrait l'énergie à partir de splines cubiques de la prédiction dont le calcul [7], la dérivation et l'intégration [6] sont gérés par la librairie SciPy. La figure 3 page 3 présente les résultats du calcul des énergies et de la norme de 50 prédictions effectuées par le réseau après entrainement. On ob-

serve que l'énergie totale des prédictions est toujours supérieure à l'énergie théorique de 0.5, celle-ci étant le minimum atteignable.

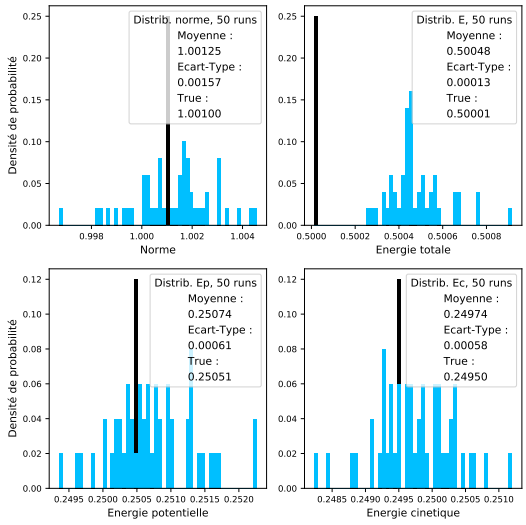

FIGURE 3: Les barres noires sont des repères visuels de la fonction à fitter.

4 Minimisation de l'énergie : utilisation stochastique des réseaux de neurones

4.1 Première version du programme

Jusqu'ici, nous nous sommes servis de la solution analytique pour entrainer le réseau puis nous avons extrait l'énergie de ses prédictions. Notre but maintenant est de trouver une méthode pour que le programme retrouve l'énergie et la fonction d'onde de l'état fondamental sans connaître la solution. Pour résoudre ce problème, j'ai eu l'idée du programme détaillé en figure 5 page 4 dans lequel le réseau cherche à reproduire une fonction d'onde constante. Les erreurs de fit qu'on peut voir sur la figure 2 page 2 nous servent à générer de petites variations dans les prédictions et de trouver des fonction ayant une énergie inférieure à la cible.

Dans la première version du programme la première cible était une fonction composée de nombres aléatoires entre 0 et 1 et l'étape 6 n'existait pas. De plus, je détruisais le réseau puis le reconstruisais à l'étape 8 en espérant générer de grandes erreurs de fit et ainsi accélérer le processus de minimisation. Cette méthode a donné le résultat visible sur la figure 4.

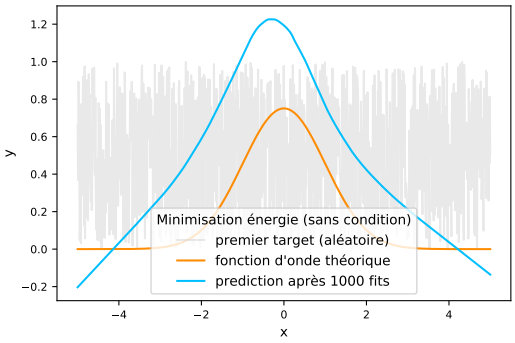

FIGURE 4: Première version du programme. 1000 itérations, discrétisation sur 1000 points.

On voit que les prédictions tendaient vers une gaussienne au bout de 1000 itérations mais le processus était particulièrement lent et les résultats trop peu précis. J'ai donc ajouté 3 conditions (figure 5, étape 6). Dorénavant, la fonction d'onde doit être : positive, symétrique et normée. Les deux premières conditions donnaient de meilleurs résultats que la figure 4 mais la précision restait faible. C'est la normalisation qui a permis d'atteindre des résultats similaires à la figure 6 page 4. La déconstruction-reconstrution du réseau avec des paramètres aléatoires avait permis d'obtenir les résultats de la figure 4, mais elle faisait perdre beaucoup de temps au programme lorsque les prédictions était déjà bonnes. Il ne restait plus qu'à affiner le résultat mais le programme continuait de reprendre son apprentissage à 0 à chaque fois qu'une meilleure cible était trouvée. Autrement dit, si on regarde la figure 2 page 2, on voit que le loss est grand au début de l'entrainement. Cela signifie que les prédictions du réseau sont très éloignées de la

cible alors qu'on cherche de petites variations pour affiner le résultat. De plus, les nouvelles conditions permettaient déjà d'orienter le programme vers une gaussienne.

4.2 Dernière version du programme

Dans cette version, j'ai abandonné la décons.-recons. du réseau et j'ai aussi changé la discrétisation en passant de 1000 à 100 points pour diminuer le temps de calcul. La figure 5 présente la dernière version en date du programme de minimisation dont le code commenté en anglais est disponible sur mon GitHub [9].

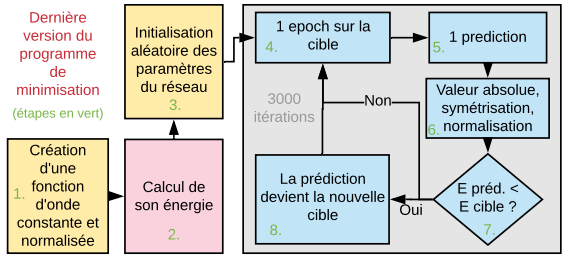

FIGURE 5: La première cible est une fonction constante. On garde la prédiction avec l'énergie la plus faible en sortie de boucle.

Un parallèle peut être fait entre le programme et la méthode du recuit simulé. Chaque itération du programme "réchauffe" le système (le réseau) via un epoch et ses paramètres changent légèrement. Néanmoins, à la différence du recuit, la sélection des prédictions nous permet de trouver la meilleure configuration des paramètres en vue de minimiser l'énergie du système quantique étudié et non d'un paramètre du réseau. On peut ainsi voir ce programme comme un hybride mélangeant le recuit et les réseaux de neurones à travers une méthode de *hill climbing*.

Pour étudier la convergence du programme, on calcule l'écart entre l'énergie trouvée et l'énergie théorique (équation (3) page 1). Les constantes m, $\hbar$ et ω étant égales à 1 dans notre programme, l'écart s'écrit :

$$E_{programme} - 0.5 \tag{6}$$

On trace ensuite cet écart en fonction du temps de calcul CPU pour observer la convergence. Le résultat est illustré par la figure 10 page 6 dans laquelle on a fait la moyenne des écarts et du temps CPU sur 30 runs du programme. On y voit notamment que la convergence dépend beaucoup du nombre de paramètres du réseau et de la géométrie choisie. Par exemple, les réseaux à une couche et les réseaux à 500 paramètres ne convergent pas. On remarque aussi que, pour les réseaux à 6 couches, 5000 paramètres semble être un meilleur choix que 30.600, contrairement aux réseaux à 3 couches.

La figure 6 illustre un résultat moyen de l'algorithme après 2000 itérations. On peut y voir que l'écart entre la fonction d'onde théorique et le résultat de l'algorithme semble être le plus important là où la courbure est forte. On voit que la partie linéaire est inférieure à la théorie alors que le sommet est supérieur. D'autres résultats semblaient comporter les mêmes problèmes. Le but du réseau étant de "se reproduire lui-même", il est possible qu'une droite soit plus simple à reproduire qu'une courbe pour le réseau. Cela rendrait le réseau "rigide", réticent à changer certains de ses paramètres qui permettraient de gagner en précision sur l'énergie, et éventuellement en temps de calcul.

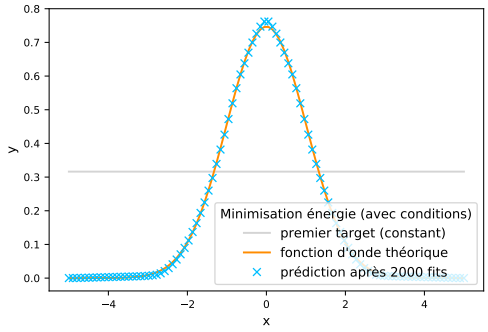

FIGURE 6: Version finale du programme. 2000 itérations, discrétisation sur 100 points.

Les potentiels étudiées pour les figures 7 et 8 sont respectivement :

$$V(x) = |x|$$
$$V(x) = \frac{1}{2}x^2 + 3e^{-4x^2} \tag{7}$$

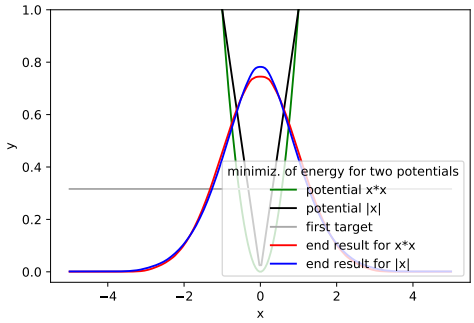

FIGURE 7: Discrétisation de l'espace sur 200 points et 20.000 itérations.

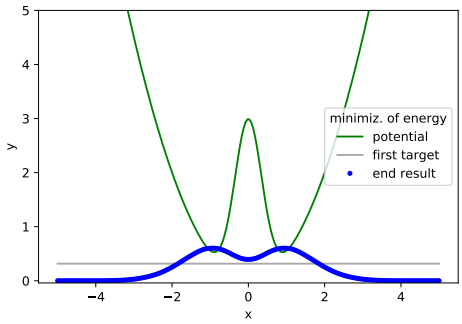

FIGURE 8: Discrétisation de l'espace sur 200 points et 20.000 itérations.

Pour finir, la figure 9 représente le temps de calcul CPU en fonction de la géométrie utilisée. On y voit que les réseaux à 500 et 5000 paramètres ont des temps CPU similaires pour la plupart des géométries. 30.600 est un peu au dessus et même très au dessus pour le réseau à 1 couche.

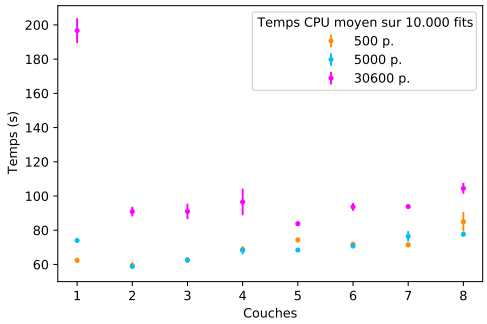

FIGURE 9: Temps CPU au bout de 10.000 itérations, moyenné sur 30 runs.

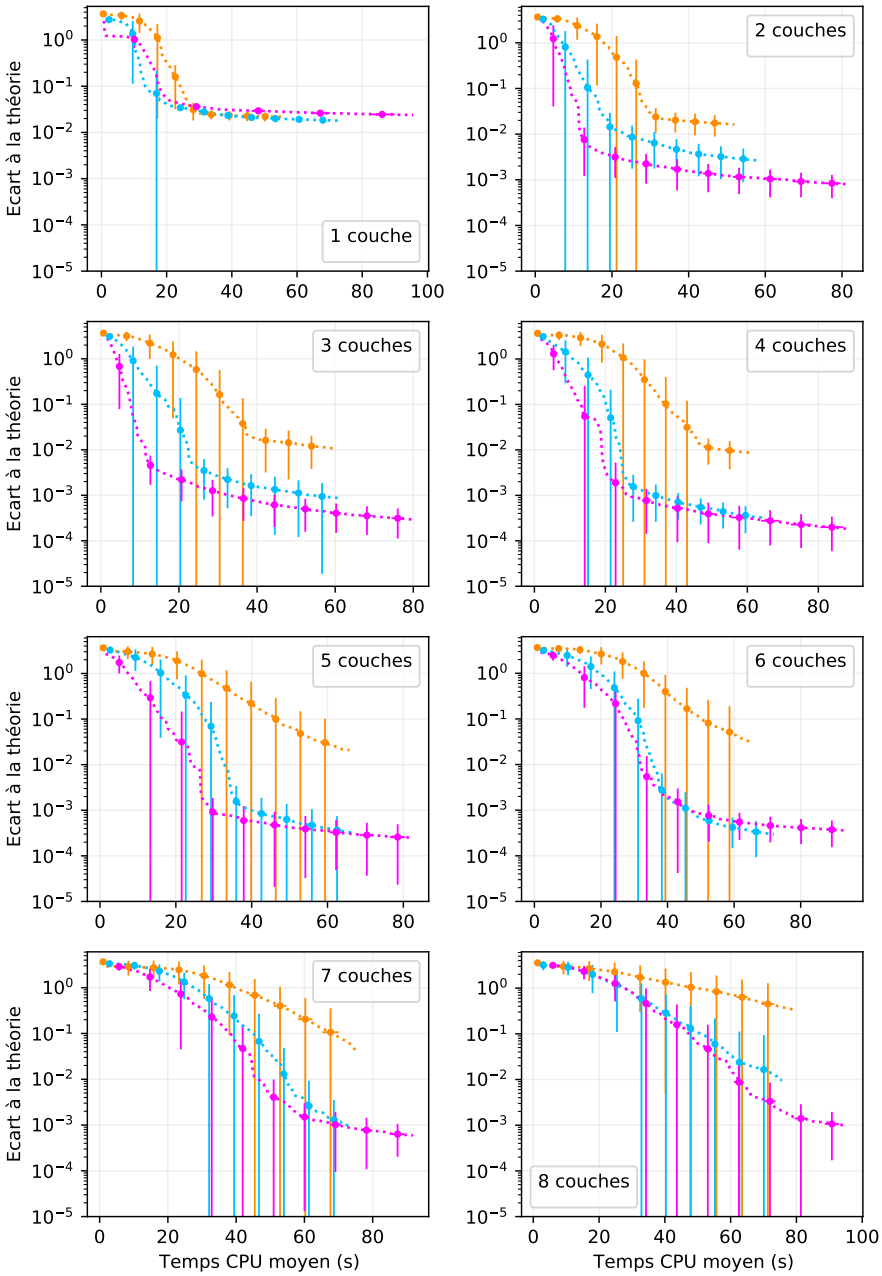

FIGURE 10: /!\les échelles de temps en abscisses sont différentes. Orange : 500 paramètres ; Bleu : 5000 ; Rose : 30600. Les courbes s'arrêtent lorsque le programme a effectué 10000 itérations, à l'exception de "1 couche - rose" qui est coupée à 5000 itérations pour des raisons de lisibilité (200 secondes au total). La précision et le temps sont moyennés sur 30 runs.

5 Conclusion

Nous avons étudié deux méthodes numériques de résolution de l'équation de Schrödinger indépendante du temps permettant de trouver la fonction d'onde et l'énergie de l'état fondamental d'un oscillateur harmonique. Contrairement à un algorithme de diffusion tel que RK4 (page 1), le programme de minimisation stochastique ne permet de trouver que le mode fondamental. Il est difficile de conclure sur les performances du programme, notamment du fait que les résultats de la figure 10 page 6 ne sont pas complètement convergés après 10.000 itérations pour la plupart des géométries étudiées. Une étude plus détaillée de l'impact du nombre de paramètres sur la convergence doit être faite sur des temps de calcul plus longs. Les figures 9 page 5 et 10 page 6 nous permettent néanmoins de dire que le temps de calcul est très long en comparaison d'une méthode numérique comme RK4 présentée page 1. On espère néanmoins que le programme sera efficace pour l'étude de systèmes à plusieurs dimensions, contrairement à un algorithme de diffusion. D'autres études doivent être faite, notamment sur le nombre optimal de points pour la discrétisation de l'espace, sur la "rigidité" du réseau et sa tendance à tracer des morceaux de droites plutôt que des courbes, ainsi qu'une étude comparative avec d'autres méthodes de résolution stochastique.

Bibliographie

[1] Claude Cohen-Tannoudji, Bernard Diu et Franck Laloë. *Mécanique quantique Tome 1*. EDP Sciences, 2018, p. 371-378. ISBN : 9782759822874.

[2] Colin Bernet. *Handwritten Digit Recognition with scikit-learn*. URL : https://thedatafrog.com/en/articles/handwritten-digit-recognition-scikit-learn/.

[3] Colin Bernet. *Le réseau à un neurone : régression logistique*. URL : https://thedatafrog.com/fr/articles/logistic-regression/.

[4] Colin Bernet. *Premier réseau de neurones avec keras*. URL : https://thedatafrog.com/fr/articles/first-neural-network-keras/.

[5] Google Colab. *Specs de la machine*. URL : https://colab.research.google.com/notebook#fileId=1_x67fw9y5aBW72a8aGePFLlkPvKLpnBl.

[6] SciPy community. *Integration avec SciPy*. URL : https://docs.scipy.org/doc/scipy/reference/tutorial/integrate.html.

[7] SciPy community. *Interpolation avec SciPy*. URL : https://docs.scipy.org/doc/scipy/reference/tutorial/interpolate.html.

[8] Keras. *Model training*. URL : https://keras.io/api/models/model_training_apis/.

[9] Clément Lotteau. *Mon GitHub avec les codes et des résultats plus détaillés*. URL : https://github.com/quadrivecteur?tab=repositories.