Introduction

Le but de ce second TD est de découvrir les fonctionnalités de numpy et d'utiliser les fonctions d'ajustements disponible puis de les comparer à des approches analytiques.

Github

J'utilise pour le maintien de mon programme un service en ligne qui s'appelle Github, me permettant d'héberger mes différents fichiers de codes dans des dossiers en ligne facilement disponibles. Je laisse à la disposition dans le bas de page le lien vers le dossier en ligne de mon compte. Les différentes informations sont disponibles dans le fichier README.md du dossier Cours_Modelisation.

Organisation du code

Mes fichiers de code se séparent en trois parties distinctes, la première comprend l'entête avec l'appel des différents modules et de leurs alias. Elle contient également les différentes informations liées à la création du fichier, comme son auteur et la date de création.

```
1 # -*- coding: utf-8 -*-
2 """
3 Created on Tue Jan 28 13:36:26 2020
4 @author: MOLLIER Yoann
5 """
6 import numpy as np
7 from matplotlib import pyplot as plt
8 from scipy.optimize import curve_fit
9
```

La seconde partie contient les informations et les définitions liées aux différentes fonctions qui seront appelées tout au long du programme, elles sont définies dès le début du programme et après l'appel des modules. Les définitions sont placées dès le début du programme pour faciliter leurs maintenance et augmenter la lisibilité du programme

```
10 #%% Definition des fonctions utilisées
11 #Fonction de la loi de malus
12 def malus(angle, amplitude, phase, shift):
13     Intensité = amplitude * np.cos((angle + phase)*np.pi/180)**2 + shift
14     return Intensité
15
16 #Fonction de droite ax+b
17 def lineaire(x, a, b):
18     y = a*x + b
19     return y
20
```

La troisième grande partie contient le programme à proprement parler et contient la logique d'appel des fonctions et l'ensemble des appels permettant la résolution de l'exercice, les appels de variables sont effectués le plus groupé possible et respecte la règle dites du camelBack, qui demande à mettre une majuscule dès le début de chaque mot secondaire dans la variable. Les variables

doivent être le plus claire possible et j'ai également essayé de documenter à l'aide de commentaire le plus possible tant que cela est nécessaire.

1ère partie : Loi de Malus

Le but de cette première partie est de créer une loi d'ajustement pour une fonction cosinus, dans le cadre de la loi de Malus.

Pour cela on récupère les données expérimentales retenues sur un fichier externe, en utilisant les fonctions : *loadtxt* de *numpy*.

En utilisant les données du TD, on utilise les données des angles et des résistances fournies par le fichier externe pour former sur le graphique les points expérimentaux en suivant la fonction de l'intensité.

On veut maintenant comparer ces résultats avec une courbe théorique, que l'on génère, selon la loi de malus, cette fonction devrait s'apparenter à une fonction cosinus dont on ne connait pas les différents paramètres que l'on a appelé, la phase (angle en degrés), l'amplitude, et le shift (ou l'intensité à l'origine que l'on appelle également offset). En manipulant un peu cette fonction on trouve une fonction approchée dont les paramètres ne semblent pas entièrement satisfaisant.

Pour confirmer ou non ces paramètres on utilise la fonction *curve_fit* qui va calculer les paramètres optimaux de notre fonction malus.

```
68
69 #Determine les variables par fitting de notre fonction Malus()
70 pfit, pcov = curve_fit(malus, angleExp, IntensiteRes)
71
```

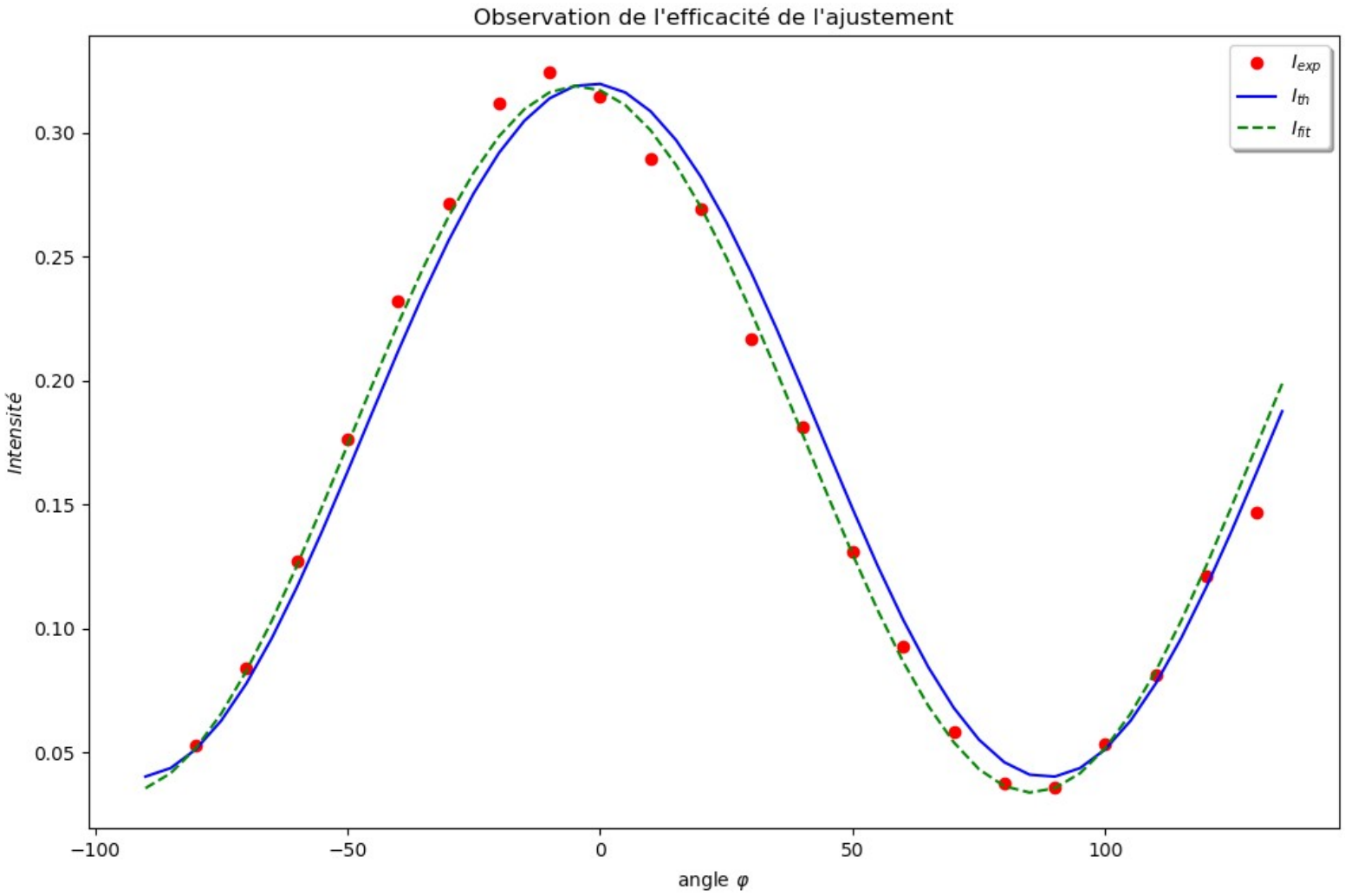
On introduit deux variables, *pfit* et *pcov* qui sont renvoyées par la fonction. Les paramètres sont renvoyés par l'array *pfit*, qui nous permettra de définir l'allure de la courbe approximée sur le graphique. L'array *pcov* nous permettra par sa diagonalisation de déterminer les incertitudes liées aux paramètres déterminés.

```
90 #Méthode de détermination  des incertitudes par la diagonalisation de pcov
91 S = np.sqrt(np.diag(pcov))
92
```

S est la liste contenant les incertitudes liées aux paramètres contenus dans pfit.

Après toutes ces opérations on peut donc récupérer les paramètres ainsi que leurs incertitudes.

I_0	0.285 A ± 0.006 A
Phase ϕ (°)	4.527° ± 0.529°
Offset	0.034 A ± 0.003 A



En rouge, les points expérimentaux, I_{exp}.
En bleu les points théoriques avec une fonction cosinus intuitée I_{th}.
En pointillé vert les points obtenus par l'approximation de curve fit.

On remarque que les points obtenus par le fitting est beaucoup plus proche de nos points expérimentaux.

2ème partie : Fonction affine et distribution aléatoire

Le but de cette deuxième partie est de créer par la méthode de l'écart type une comparaison avec la méthode des moindres carrés effectué par la fonction curve_fit pour effectuer une fonction de régression linéaire le long d'une fonction affine avec une simulation de bruit aléatoire.

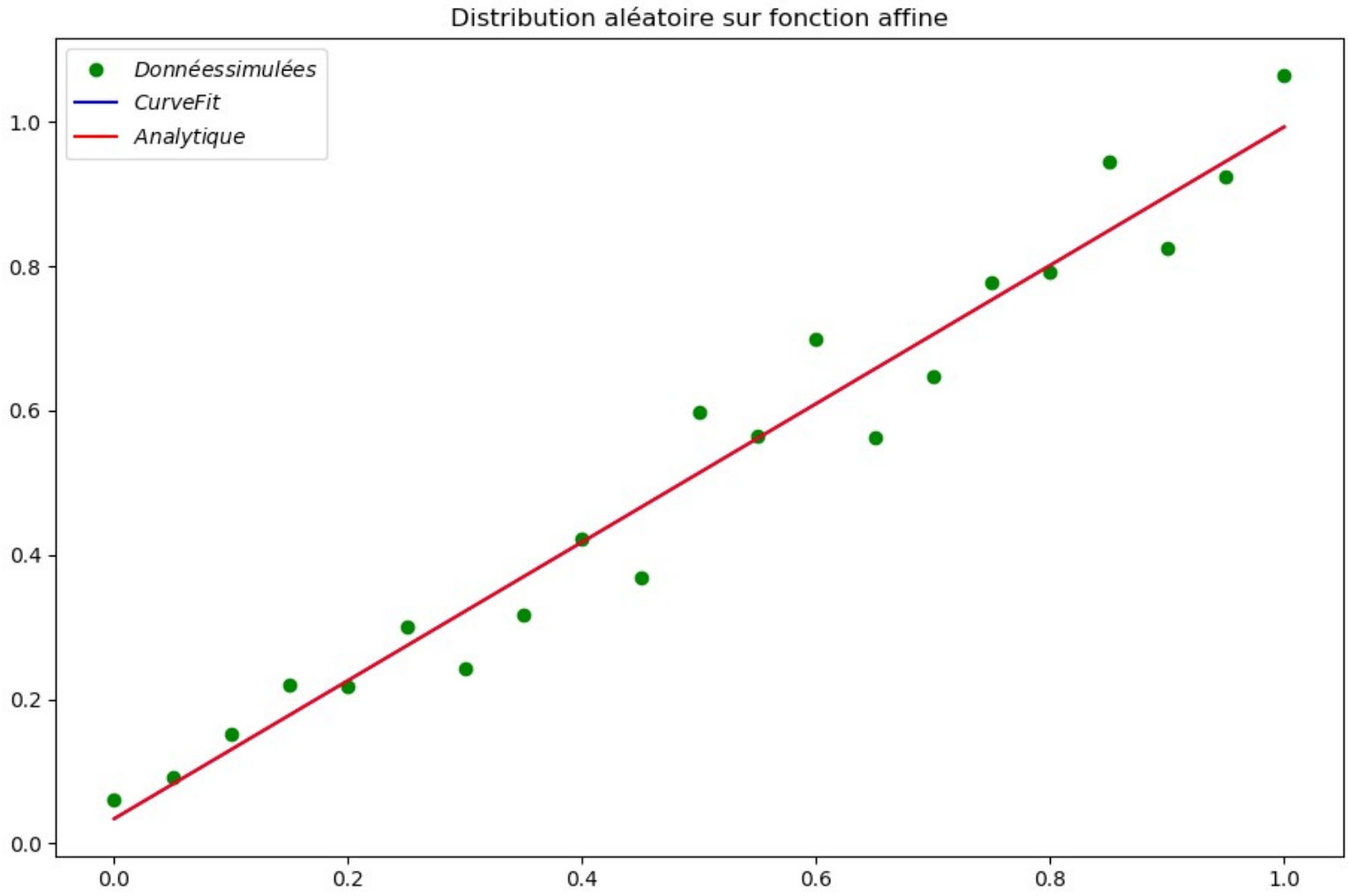
Fonction *rand()*, cette fonction renvoie un nombre aléatoire compris entre 0 et 1. Dans le cadre de cet exercice, j'ai voulu que mes données se rapprochent de la valeur du coefficient directeur que j'ai moi-même choisi (ici on gardera 1) pour cela je dois donc égaliser ma fonction rand() pour

que la moyenne des valeurs soit égale à mon coefficient directeur et à l'ordonnée à l'origine. il faut donc que la moyenne de rand() = 0. Pour cela il suffit de soustraire 0.5 à rand() pour chaque itération de la fonction.

Ainsi la moyenne de (rand()-0.5) = 0.

Pour utiliser la fonction curve_fit, on fera une fonction "linéaire" qui permettra de créer une fonction affine à l'aide des paramètres de x (les abscisses), a le coefficient directeur et b l'ordonnée à la l'origine.

Pour ce qui est de la solution analytique on privilégiera la solution générale, pour cela on définira 4 nouvelles fonctions : *coeff()*, *origine()*, *sigmaCoeff() et sigmaOrigine()*, qui permettront respectivement par la méthode analytique de redéfinir le coefficient directeur, son origine à l'ordonnée et leurs incertitudes.



En vert les données générées par la distribution aléatoire.

J'ai choisi de prendre les paramètres suivants pour cette distribution : a = 1 et b = 0.

On remarque que la droite bleu et rouge sont indiscernables l'une de l'autre. En augmentant l'effectif, les deux droites restent indiscernables, on peut se demander à première vue si les deux

droites sont bien dessinées, peut être qu'en augmentant la portée de la fonction qui ici reste compris entre [0,1] qui a été définie par la fonction linspace(). Mais lorsque l'on regarde les paramètres on s'aperçoit que les valeurs trouvées sont très proches, identiques même.

Méthode	Curve_fit	Analytique
Coefficient directeur	1.04738 ± 0.04494	1.04738 ± 0.04494
Ordonnée à l'origine	-0.02488 ± 0.02627	-0.02488 ± 0.02627

```
124 print('\nValeurs de a et b obtenue par la méthode curve_fit')
125 print('a = {:.5f}+-{:.5f}'.format(pfit[0],S[0]))
126 print('b = {:.5f}+-{:.5f}'.format(pfit[1],S[1]))
127
128 print('Valeurs de a et b obtenue par la méthode analytique')
129 print('a = {:.5f}+-{:.5f}'.format(coeff(x,distrib),sigmaCoeff(x,distrib)))
130 print('b = {:.5f}+-{:.5f}'.format(origine(x, distrib),sigmaOrigine(x,distrib)))
```