IPSA

Ma324 - Optimisation différenciable 2

Aéro 3

Analyse des modèles aux moindres carrés et algorithme de ransac

Étudiants :
Guillaume Faivre
Emma Filippone
Titouan Millet
Camille Pieussergues

Professeurs :
Y. Tendero
C. Peschard
L. Bletzacker

8 juin 2023

Cette page est laissée intentionnellement vierge.

— Date de remise du rapport : 08/06/2023

— Référence du projet : Ma324/S6/ 2022 – 2023

— Intitulé du projet : Analyse des modèles aux moindres carrés et algorithme de ransac

— Type de projet : expérimental, simulation

— Objectifs du projet :

L'objectif de cette étude est de simuler des modèles linéaires et non-linéaires, sans ou avec bruit selon une loi uniforme ou normale. Sera étudié aussi l'effet des outliers puis la réestimation du modèle en ne prenant en compte que les inliers à l'aide de l'agorithme de ransac. Enfin, deux toolbox seront abordées : scipy.optimize.least_square et numpy.linalg.svd.

— Mots-clefs du projet :

1. Méthode des moindres carrés
2. Échantillons
3. Outliers / Inliers
4. Algorithme de ransac

Table des matières

Notations, acronymes

n	nombre d'échantillons
MC	moindres carrés
ϵ	bruit
C_{XY}	covariance de X et Y
σ_X^2	variance de X
$\bar{x}$	moyenne de x
$\bar{y}$	moyenne de y
β_i	estimateur
U	matrice orthogonale de taille $m \times m$
Σ	matrice diagonale de taille $m \times n$
U	matrice orthogonale de taille $n \times n$

1 Méthodologie et organisation du travail

1.1 Répartition des tâches

Afin de réaliser ce projet, nous avons choisi de travailler au nombre de 4. Le choix de la méthode de travail a été de partager toutes nos ressources en permanence. Nous faisions des sessions de travail où nous pouvions travailler sur un même document en temps réel. Pour ce faire, nous avons utilisé l'IDE Visual Studio Code et son extension Live-Share. En complément nous avons créé un repesitory GitHub. Avec ces deux outils nous avons pu travailler de façon efficace et rigoureuse.

1.2 Arborescence de travail

Le projet se base sur les différents TP que nous avons pu réaliser durant le semestre. Afin de rester rigoureux, nous avons choisit de segmenter notre code selon les TP. Voici notre arborescence :

- TP1
  - Rapport
    - DOC_library_TP1.html
  - Doc
    - Sujet_TP1.pdf
  - Scripts
    - library_TP1.py
    - TP1.py
- TP2
- TP3

Nous avons donc trois dossiers, un pour chaque TP. Dans chaque dossier, nous retrouvons le dossier Rapport qui contient la documentation sous un format html. Dans le dossier Scripts nous retrouvons la bibliothèque qui contient toutes les déinitions de fonctions utilisées dans le script principale : [TP°].py

Lien vers le GitHub : Projet Ma324

2 Motivation et modélisation mathématique

Les ingénieurs font couramment appel à des modèles mathématiques afin de décrire une observation ou des jeux de valeurs. L'utilisation de ces modèles permettent de donner des explications, faire des prédictions, des simulations. Dans tous les cas de figure, l'objectif est de comprendre un phénomène ou de le simplifier pour l'étudier. Cependant, il faut que le modèle soit le plus proche de la réalité. Afin de pouvoir l'ajuster au mieux, nous pouvons utiliser des méthodes d'optimisation différentiable afin de trouver les meilleurs coefficients pour notre modèle.

C'est dans ce cadre que ce projet intervient. À travers ce rapport, nous étudierons différents jeux de données et nous découvrirons toutes les notions d'optimisation de modèle. Ce projet est découpé selon différents travaux qui contiennent eux-même différents exercices.

Plus précisément, nous allons simuler des modèles linéaires et non-linéaires, sans ou avec bruit selon une loi uniforme ou normale. Sera étudié aussi l'effet des outliers puis la réestimation du modèle en ne prenant en compte que les inliers à l'aide de l'agorithme de ransac. Enfin, deux toolbox seront abordées : scipy.optimize.least_square et numpy.linalg.svd.

3 TP1

Dans cette partie, nous allons commencer par nous familiariser avec les outils d'optimisation. Dans un premier temps, nous allons simuler le jeux de données que nous étudierons puis nous calculerons les premiers estimateurs β_1 et β_2. Nous ferons une estimation de notre modèle puis nous étudierons les effets des outliers sur notre modèle.

3.1 Simulation des données

Nous considérons que notre échantillon de données contient 100 valeurs comprises entre 0 et 1. Pour générer ces données, nous utilisons l'outil linspace de la bibliothèque numpy :

```
1 n = 100
2 x = np.linspace(0,1,n)
```

Afin d'étudier notre échantillon de données, nous nous proposons de créer un modèle. Nous commençons par définir notre modèle comme une combinaison linéaire :

$$b_1 f(x) + \cdots + b_k f(x) \tag{1}$$

L'objectif maintenant va être de définir les différents coefficients de l'équation (1). Prenons un exemple simple avec $k = 2$, nous obtenons alors $f_1(x) = 1$ et $f_2(x) = x$. Nous considérons notre modèle comme parfait, nous pouvons alors écrire notre modèle :

$$\begin{aligned} & b_1 f_1(x) + b_2 f_2(x) = b_1 + b_2 x \\ \Leftrightarrow\ & b_1 f_1(x) + b_2 f_2(x) = y \\ \Leftrightarrow\ & y_1 = b_1 + b_2 x_1 \\ & \vdots \\ & y_n = b_1 + b_2 x_n \end{aligned}$$

Dans cette première approche, nous allons considérer $b_1 = 10$ et $b_2 = 2$ de plus nous allons considérer qu'un terme de bruit est ajouté pour détériorer le modèle. Le bruit est défini uniformément de façon aléatoire sur l'intervalle [-1,1]. Nous obtenons alors le modèle suivant :

$$\forall x \in [0,1] : \quad y = 10 + 2x + \epsilon \tag{2}$$

où ϵ est notre terme de bruit calculé comme suit avec python :

```
1 def eps_aleatoire(n):
2     liste_eps = []
3     for i in range(n):
4         eps = random.uniform(-1,1)
5         liste_eps.append(eps)
6     return liste_eps
```

Maintenant, nous pouvons représenter notre modèle en traçant un nuage de point :

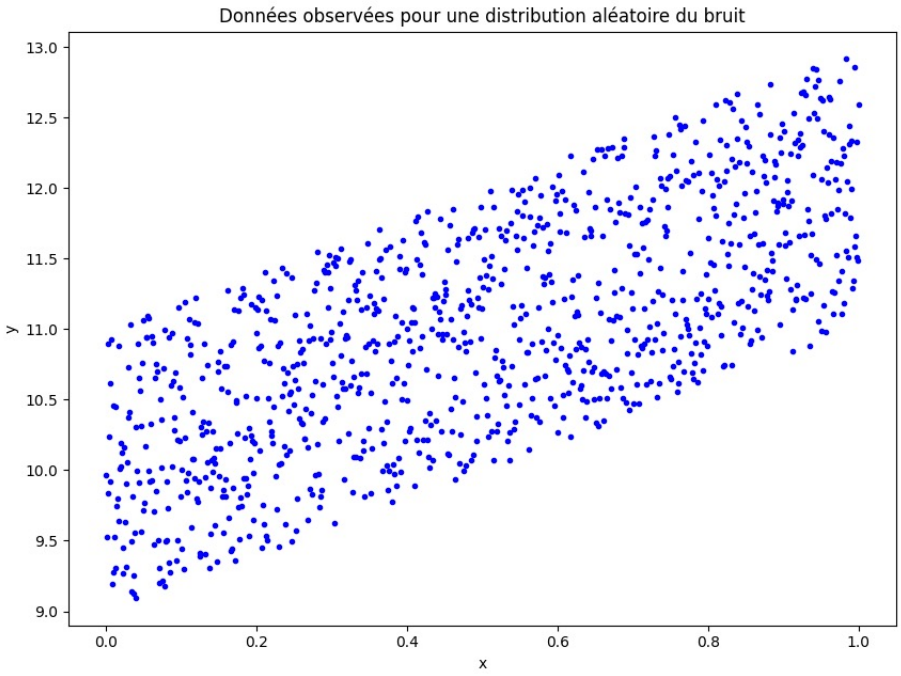


FIGURE 1 – Représentation des données observées pour une distribution aléatoire et uniforme du bruit

Sur cette figure, nous constatons que nous avons bien une combinaison linéaire et les valeurs de x varient bien de 0 à 1. Remarquons aussi que les valeurs de $y(x)$ sont comprises entre 9 et 13. Cette courbe est donc très cohérente avec nos attentes.

3.2 Calcul aux moindres carrés des estimateurs β_1 et β_2

Maintenant que nous avons notre simulation de données, nous créons notre modèle aux moindres carrés et vérifions s'il colle à notre simulation de données. Dans un premier temps, nous calculons C_{XY} et σ_X^2 :

$$\forall x, y \in \mathbb{R}^2, \sigma_X^2 = \frac{1}{n}\sum_{i=1]}^{n}(x_i - \bar{x})^2$$

$$C_{XY} = \frac{1}{n}\sum_{i=1]}^{n}(x_i - \bar{x})(y_i - \bar{y})$$

Nous les implémentons sous python de la manière suivante :

```
1  def variance(x,n):
2      sigma2_x = 0
3      for i in range(1,n):
4          sigma2_x += (x[i] - np.mean(x))**2
5      return (1/n)*sigma2_x
6
7  def covariance(x, y, n):
8      Cxy = 0
9      for i in range(1,n):
10         Cxy += np.dot((x[i] - np.mean(x)),(y[i] - np.mean(y)))
11     Cxy = (1/n)*Cxy
12     return Cxy
```

Le calcul des β_i se fait avec les formules suivantes qui ont été démontrées dans la lecture 1 du cours :

$$\beta_2 = \frac{C_{XY}}{\sigma_X^2}$$

$$\beta_1 = \bar{y} - \beta_2\bar{x}$$

Implémentation python :

```
1  def beta2(var, cov):
2      return cov / var
3
4  def beta1(x, y, beta2):
5      return np.mean(y) - beta2*np.mean(x)
```

Nous traçons alors notre modèle des moindres carrés défini par $\beta_1 + \beta_2 x$:

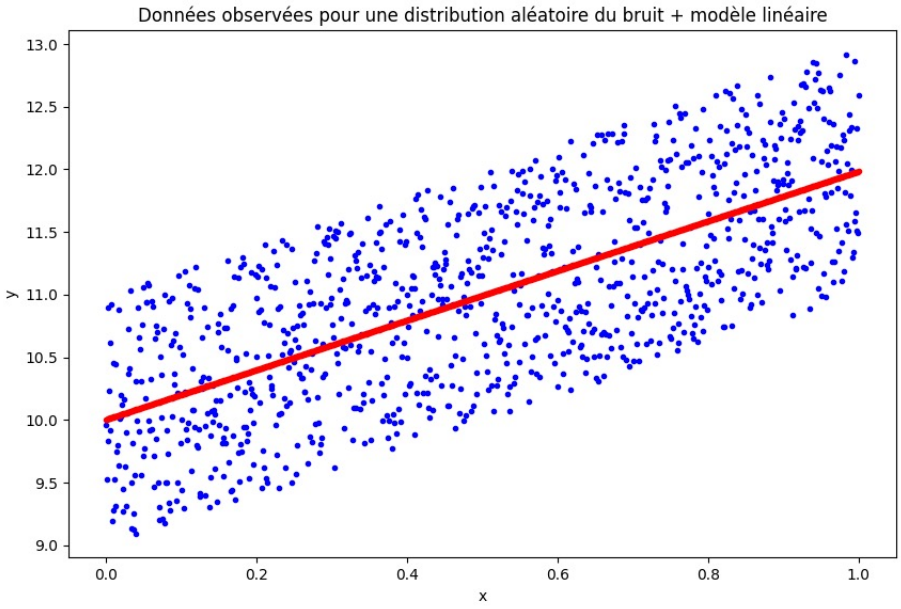


FIGURE 2 – Représentation des données observées pour une distribution aléatoire et uniforme du bruit (bleu) + modèle MC (rouge)

Nous constatons que ce modèle colle plutôt bien à notre simulation de donnée. Nous pouvons faire une première estimation de notre modèle en prenant 3 valeurs de x et nous les comparons avec les valeurs du vrai modèle sous-jacent défini en (2) avec $\epsilon = 0$ (sans bruit) :

Valeur prédite par le modèle MC :

$$x := 0 \to y = 9,98$$
$$x := \frac{1}{5} \to y = 10,99$$
$$x := 1 \to y = 11,99$$

Valeur du vrai modèle :

$$x := 0 \to y = 10$$
$$x := \frac{1}{5} \to y = 11$$
$$x := 1 \to y = 12$$

Nous constatons que notre modèle est très fidèle au vrai modèle avec une erreur d'environ $0,01$. Lors de cette simulation, le modèle MC a défini $\beta_1 = 2,007$ et $\beta_2 = 9,987$. Nous constatons qu'elles sont très proches de celles du vrai modèle (2) qui sont $b_1 = 2$ et $b_2 = 10$.

Cependant nous constatons qu'avec moins de valeurs observables, notre erreur augmente mais plus le nombre de valeurs observables augmente plus l'erreur diminue.

Nous nous essayons à tracer un modèle non-linaire défini par :

$$\forall x \in [0,1], \qquad y = 10 + 2x^2 + \epsilon \qquad ; \epsilon \in [-1,1]$$

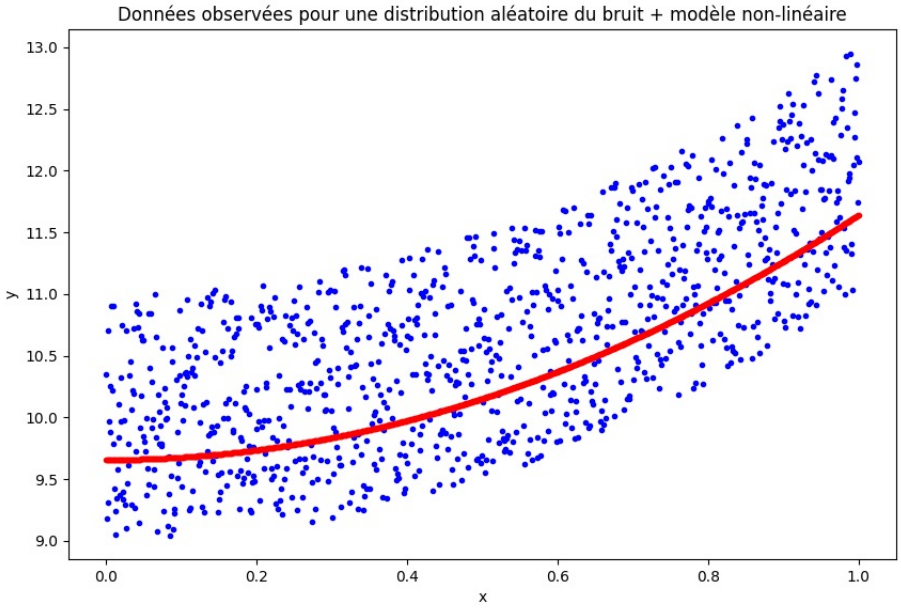


FIGURE 3 – Représentation des données observées pour une distribution non-linaire aléatoire et uniforme du bruit (bleu) + modèle non-linéaire (rouge)

Nous constatons que le modèle MC à réussi à calculer les valeurs de β même si le modèle est non-linéaire. Si nous comparons à $x = 0$ les valeurs du modèle MC avec le modèle vrai (2) dans le cas d'un modèle non-linaire, nous constatons que l'erreur est faible. Ici, nous avons pour 1000 observations. $x : 0 \rightarrow y = 9,68$. Ce qui reste proche de la valeur de référence qui est de 10. Par contre, en diminuant le nombre d'observations à 10, nous relevons $x : 0 \rightarrow y = 9,26$. Donc l'estimation est moins bonne.

En conclusion, nous pouvons affirmer qu'il est possible de faire le calcul des β_i avec ce modèle MC sur une simulation de données non-linéaires. L'estimation sera moins bonne mais reste plausible.

3.3 Estimation du modèle

L'estimation du modèle est un point clé pour la validation de notre modèle. Afin de l'estimer, nous observons si l'estimateur β_2 est biaisé. Lorsqu'on dit qu'un estimateur est biaisé, cela signifie que l'estimation obtenue est majoritairement loin de la vraie valeur du paramètre que nous cherchons à estimer. Pour ce faire nous allons créer un histogramme de β_2. Pour créer cet histogramme, nous allons faire plusieurs simulations en modifiant la valeur du bruit. Le bruit qui avait été défini à l'équation (2) va maintenant suivre une loi normale de 0 à 1. Nous le redéfinissons dans le code de la manière suivante :

```
1  def esp_aleatoire_gaussian(n, mu, sigma):
2      liste_eps = []
3      for i in range(n):
4          liste_eps.append(np.random.normal(mu,sigma))
5
6      return liste_eps
```

Nous refaisons une simulation des données :

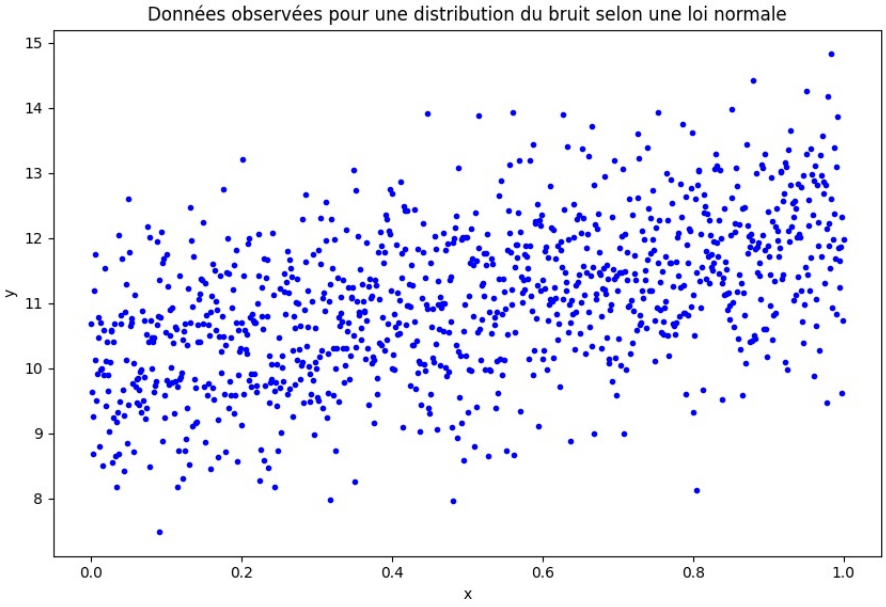


FIGURE 4 – Représentation de la données observées pour une distribution aléatoire du bruit

Nous remarquons que les points occupent beaucoup plus de place et sont plus éloignés les un des autres. Ce comportement était attendu car le bruit est plus présent.

Après 1000 simulations, nous récupérons une liste de β_2 que nous arrangeons sous forme d'histogramme :

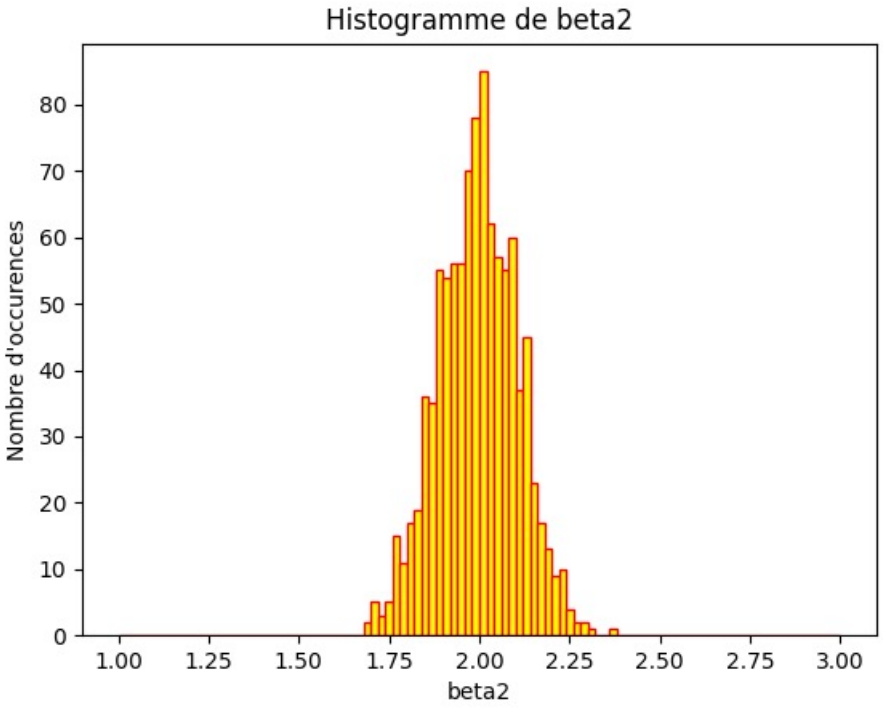


FIGURE 5 – Représentation de la données observées pour une distribution aléatoire du bruit

Un pic flagrant se forme autour de la valeur 2. Ce pic nous indique que β_2 est centré autour de la vraie valeur donc il n'est pas biaisé. Cette conclusion, nous rassure quant à l'utilisation de notre modèle. Le modèle aura tendance à converger vers la vraie valeur du coefficient et donc sera fiable.

3.4 Effet des outliers

Les outilers sont les points qui se démarquent du reste du peloton. Nous nous intéressons alors à leur impact sur notre modèle. Afin de faire cette expérience, nous reprenons les données de la partie 3.1 mais en les corrompant.

Nous utilisons le code python suivant :

```
1  def outliers(y, n, n_outliers, min, max):
2      for o in range(n_outliers):
3          i_random = random.randint(0,n)
4          y[i_random] = random.uniform(min,max)
5
6      return y
```

Enfin nous traçons le modèle ainsi que le nuage de point :

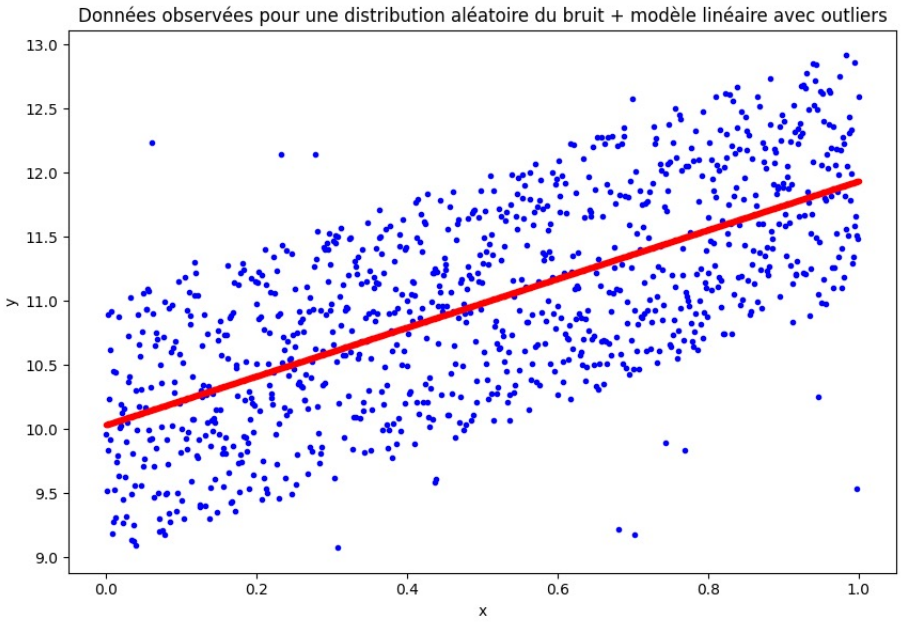


FIGURE 6 – Représentation des données observées pour une distribution aléatoire du bruit selon une loi normale

Nous constatons que le modèle est affaibli mais semble rester cohérent. Les outliers sont modifiés intentionnellement car ils ne sont pas présents de base dans notre modèle. Nous faisons cette modification afin de tester la robustesse de notre modèle. Les outliers nous aident également à comprendre le comportement de notre modèle et ainsi faire de meilleures estimations.

4 TP2

4.1 Courbes d'ajustement

Comme dans le TP1, nous considérons que notre échantillon de données contient 100 valeurs comprises entre 0 et 1. Pour générer ces données, nous utilisons l'outil linspace la bibliothèque numpy :

```
n = 100
x = np.linspace(0,1,n)
```

Nous venons à nous intéresser au modèle suivant :

$$\forall x \in [0,1], \quad y = 10 + 2x + 8x^2 + \epsilon \tag{3}$$

où ϵ est notre terme de bruit et suit la loi normale centrée avec variance égale à 2, nous pouvons le calculer de la manière suivante en python :

```
    def esp_aleatoire_gaussian(n, mu, sigma):
        liste_eps = []
        for i in range(n):
            liste_eps.append(np.random.normal(mu,sigma))

        return liste_eps
```

Pour estimer les paramètres du modèle, nous venons former la matrice X et le vecteur Y de la manière suivante :

$$X = \begin{pmatrix} 1 & x_1 & (x_1)^2 \\ 1 & x_2 & (x_2)^2 \\ \vdots & \vdots & \vdots \\ 1 & x_n & (x_n)^2 \end{pmatrix} \quad et \quad Y = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}$$

Que nous implémentons en python :

```
def mat_X_exo1(x):
    X = np.ones((n,3))
    for i in range(n):
        X[i,1] = x[i]
        X[i,2] = x[i]**2
    return X
def mat_Y_exo1(y):
    Y = np.ones((n,1))
    for i in range(n):
        Y[i] = y[i]
    return Y
```

Puis, à l'aide de la formule suivante :

$$\beta^* = (X^T X)^{-1} X^T Y \tag{4}$$

où

$$\beta^* = \begin{pmatrix} \beta_1 \\ \beta_2 \\ \beta_3 \end{pmatrix}$$

Nous pouvons ainsi estimer les paramètres β_i du modèle grâce au code suivant :

```
1       def calcul_beta(X, Y):
2           return np.dot(np.dot(np.linalg.inv(np.dot(X.T,X)),X.T),Y)
```

Ainsi nous obtenons la représentation du modèle et des données suivante :

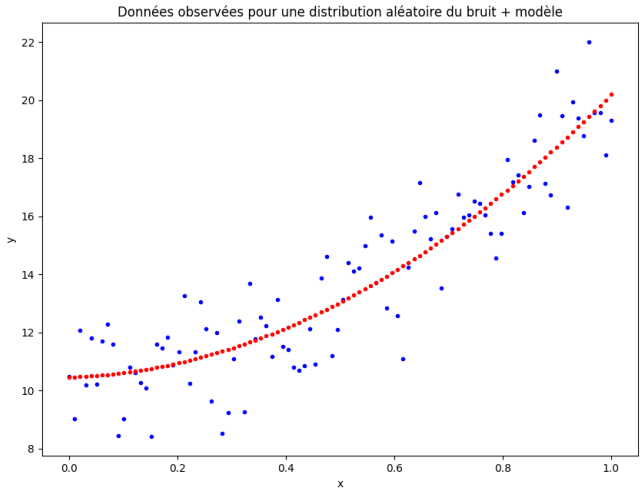


FIGURE 7 – Représentation des données observées pour une distribution aléatoire du bruit selon une loi normale

Nous effectuons les mêmes étapes pour le modèle suivant :

$$\forall x \in [0,1] : \quad y = 10 + 2x + 8x^2 + 10\sin(\pi x) + \epsilon \tag{5}$$

Nous gardons un epsilon aléatoire géréré par la fonction **esp_aleatoire_gaussian** que nous avons défini pour le modèle précédent.

Pour estimer les paramètres de ce modèle, nous devons former la matrice X et le vecteur Y de la manière suivante :

$$X = \begin{pmatrix} 1 & x_1 & (x_1)^2 & sin(\pi x_1) \\ 1 & x_2 & (x_2)^2 & sin(\pi x_2) \\ \vdots & \vdots & \vdots & \vdots \\ 1 & x_n & (x_n)^2 & sin(\pi x_n) \end{pmatrix} \quad et \quad Y = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}$$

Que nous implémentons en python :

```
def mat_X_sin(x):
    X_sin = np.ones((n,4))
    for i in range(n):
        X_sin[i,1] = x[i]
        X_sin[i,2] = x[i]**2
        X_sin[i,3] = np.sin(np.pi*x[i])
    return X_sin
def mat_Y_exo1(y):
    Y = np.ones((n,1))
    for i in range(n):
        Y[i] = y[i]
    return Y
```

Puis nous utilisons la fonction **calcul_beta** codée précédemment. Ainsi, nous obtenons la représentation du modèle et des données suivante :

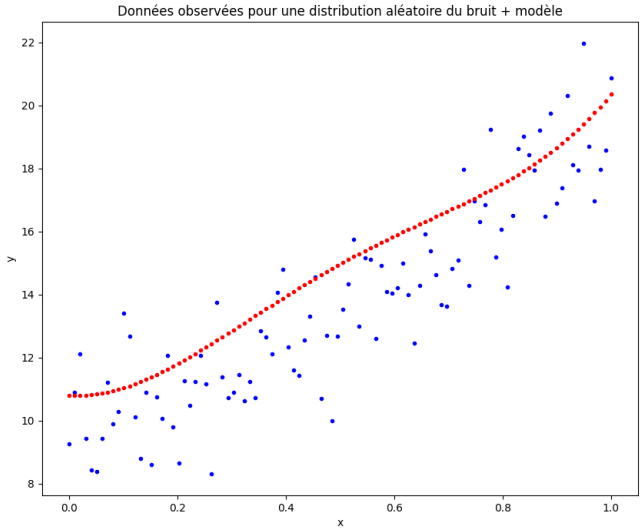


FIGURE 8 – Représentation des données observées pour une distribution aléatoire du bruit selon une loi normale

Nous remarquons dans ce cas-ci que le modèle estimé n'est pas très fidèle au vrai modèle. Nous pouvons mettre en cause la présence du sinus dans le modèle qui le rend encore moins linéaire.

4.2 Paramètre d'orbite

Dans cette partie nous nous intéressons à des paramètres d'orbite qui sont rangés dans le tableau suivant :

Angles en dégrés	43	45	52	93	108	126
Valeurs obervées	4.7126	4.5542	4.0419	2.2187	1.8910	1.7599

Nous considérons le modèle suivant :

$$f(\theta) = \frac{p}{1 - e\cos(\theta)} \tag{6}$$

Nous travaillons l'équation (6) afin de faire apparaître un potentiel modèle linéaire :

$$\begin{aligned} f(\theta) &= \frac{p}{1 - e\cos(\theta)} \\ \Leftrightarrow \frac{1}{f(\theta)} &= \frac{1}{p} + \frac{e}{p}(-\cos(\theta)) \end{aligned}$$

Par identification, nous posons :

$$y = \frac{1}{f(\theta)} \qquad x = -\cos(\theta)$$
$$\beta_1 = \frac{1}{p} \qquad \beta_2 = \frac{e}{p}$$

Au format matriciel nous obtenons :

$$X = \begin{pmatrix} 1 & -\cos(\theta_1) \\ 1 & -\cos(\theta_2) \\ \vdots & \vdots \\ 1 & -\cos(\theta_n) \end{pmatrix} \qquad Y = \begin{pmatrix} \frac{1}{f(\theta_1)} \\ \frac{1}{f(\theta_2)} \\ \vdots \\ \frac{1}{f(\theta_n)} \end{pmatrix} \qquad \beta = \begin{pmatrix} \beta_1 \\ \beta_2 \end{pmatrix} \tag{7}$$

Afin de calculer β nous utilisons la formule vue dans la lecture 2 :

$$\beta = \left(X^T X\right)^{-1} X^T Y \tag{8}$$

D'un point de vu python nous écrivons les matrices définies en (7) comme suit :

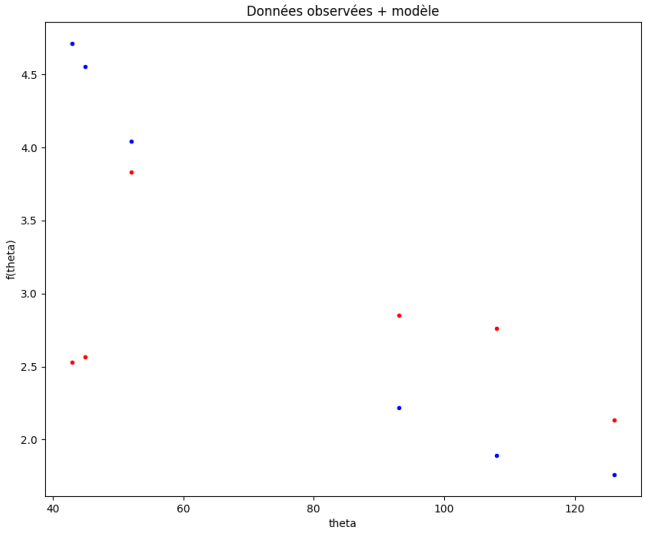
```
for i in range(n_theta):
    X_theta[i,1] = -np.cos(theta[i])
    Y_theta[i] = 1/f_theta[i]
```

L'équation de β définie en (8) :

```
beta_theta = np.dot(np.dot(np.linalg.inv(np.dot(X_theta.T,X_theta)),X_theta.T),
    Y_theta)
```

Enfin nous calculons les paramètres $p = \frac{1}{\beta_1}$ et $e = \frac{p}{\beta_2}$:

```
p = 1/beta_theta[0]
e = p*beta_theta[1]
```



Après exécution du code, nous obtenons les valeurs pour p et e :

$$p = 3.43$$
$$e = -0.6420$$

La figure si contre est une représentation de notre modèle. Notre conjecture est que nous n'avons pas suffisamment de valeurs pour obtenir une courbe optimale.

FIGURE 9 – Données observées + modèle

4.3 Gestion des outliers : algorithme ransac (simplifié)

Nous considérons que notre échantillon de données contient 100 valeurs comprises entre 0 et 1, et nous construisons le modèle $y = 10 + 2x$. Nous utilisons le code python suivant :

```
n = 100
def y_exo3(n, x, eps):
    y = []
    for i in range(n):
        yi = 10 + 2*x[i]
        y.append(yi)
    return y
```

Puis, nous venons modifier 10 valeurs de y en générant des valeurs aléatoires à partir d'une distribution gaussienne centrée sur zéro et avec une variance de un. Pour cela, nous utilisons le code suivant :

```
    n_outliers = 10

for o in range(n_outliers):
    i_random = random.randint(0,n-1)
    y_out[i_random] = np.random.normal(11, 0.5)
```

Ainsi, nous obtenons la représentation suivante :

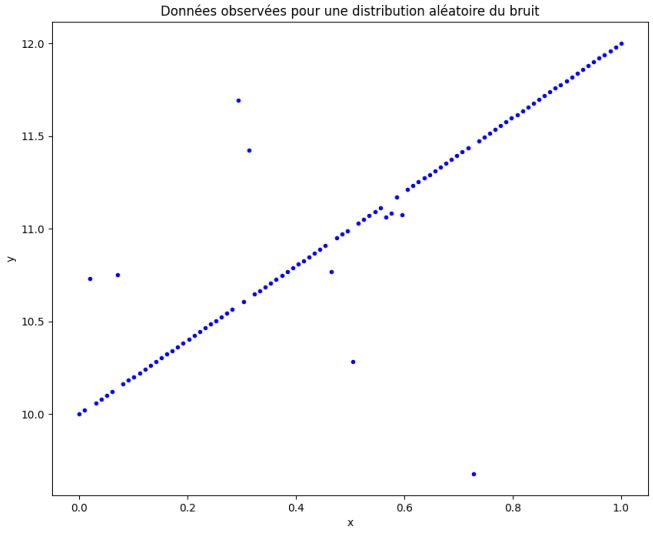


FIGURE 10 – Représentation des données observées pour une distribution aléatoire du bruit selon une loi normale

Maintenant, nous écrivons une fonction qui, pour deux modèles, nous donnera celui qui a le plus d'inliers que l'autre. Pour cela, nous allons vérifier, pour chaque couple (x,y), si la condition $\beta_1 x + \beta_2 - y < Treshold$ est remplie. $Treshold$ étant une valeur seuil donnée. Ainsi, les couples (x,y) qui valident cette condition sont des inliers.

Maintenant, nous créons une fonction qui nous permet de générer 2 échantillons de manière aléatoire puis de les associer à un modèle :

```
def calcul_model(n):
    eps = eps_aleatoire(n)

    y_m = y_exo3(n, x, eps)

    beta2_m = beta2(variance(x,n),covariance(x,y_m,n))
    beta1_m = beta1(x, y_m, beta2_m)

    m = [beta1_m,beta2_m]

    return m
```

Afin de trouver le meilleur modèle possible nous venons comparer 10 fois deux modèles. Pour cela, à chaque comparaison nous gardons le meilleur des 2 afin de le comparer avec un nouveau modèle aléatoire. Nous utilisons la fonction python suivante :

```
def inliers(m1,m2, Treshold):

    for _ in range(10):
        beta1_m1 = m1[0]
        beta2_m1 = m1[1]
        beta1_m2 = m2[0]
        beta2_m2 = m2[1]

        n_m1 = 0
        n_m2 = 0

        for i in range(n):
            if abs(beta1_m1*x[i] + beta2_m1 - y3[i]) < Treshold:
                n_m1 += 1

            if abs(beta1_m2*x[i] + beta2_m2 - y3[i]) < Treshold:
                n_m2 += 1

        if n_m1 >= n_m2:
            m2 = calcul_model(n)
            best_model = m1

        else:
            m1 = calcul_model(n)
            best_model = m2

    return best_model
```

Nous venons maintenant comparer la stratégie utilisée à la fin du TP1 et celle que nous venons de voir. Pour cela, nous utilisons le code suivant :

```
def compare(m1,m2, Treshold):

    beta1_m1 = m1[0]
    beta2_m1 = m1[1]
    beta1_m2 = m2[0]
    beta2_m2 = m2[1]

    n_m1 = 0
    n_m2 = 0

    for i in range(n):
        if abs(beta1_m1*x[i] + beta2_m1 - y3[i]) < Treshold:
            n_m1 += 1

        if abs(beta1_m2*x[i] + beta2_m2 - y3[i]) < Treshold:
            n_m2 += 1

    if n_m1 >= n_m2:
        m2 = calcul_model(n)
        return "TP1"

    else:
        m1 = calcul_model(n)
        return "TP2"
```

Avec :

```
m_TP1 = [10.037398645699664, 1.9031078680985294]
m_TP2 = inliers(m1, m2, 1)
```

Les paramètres de m_TP1 ayant été obtenu lors de l'excécution du fichier python TP1.

Nous obtenons le même résultat à chaque exécution, la stratégie du TP2, le modèle estimé via l'algorithme simplifié de ransac, est la meilleure, ce qui paraît logique car le modèle a été affiné en 10 répétitions.

5 TP3

Dans cette partie, nous allons approfondir l'estimation de notre modèle en utilisant cette fois l'algorithme ransac complet. En effet, la version simplifiée fonctionne mais elle ne prend pas en compte le bruit. La version complète peut fonctionner avec des échantillons contenant du bruit, ce qui est plus réaliste.

5.1 Gérer les outliers : algorithme ransac complet

Nous considérons que notre échantillon de données contient 100 valeurs comprises entre 0 et 1, et nous construisons le modèle $y = 10+2x+\epsilon$, où ϵ suit la loi normale centrée avec une variance de 2.

Nous utilisons le code python suivant :

```
def y_exo1_3(n, x, eps):
    y = []
    for i in range(n):
        yi = 10 + 2*x[i] + eps[i] # modèle linéaire
        y.append(yi)
    return y
```

Nous venons maintenant écrire une fonction qui d'après un modèle et la valeur seuille renvoie l'ensemble des inliers :

```
def inliers(m, Treshold):
    beta1_m = m[0]
    beta2_m = m[1]

    Liste_n_m = []

    for i in range(n):
        if abs(beta1_m + beta2_m*x[i] - y1[i]) < Treshold:
            Liste_n_m.append([x[i],y1[i]])

    return Liste_n_m
```

Puis nous écrivons une fonction qui calcule le modèle à partir d'un ensemble de données observées :

```
def calcul_model(n):
    beta2_m = beta2(variance(x,n),covariance(x,y1,n))
    beta1_m = beta1(x, y1, beta2_m)

    m = [beta1_m,beta2_m]

    return m
```

Nous souhaitons maintenant réestimer le modèle sur les inliers donné par le modèle actuel, pour cela nous modifions la boucle principale de la manière suivante :

```
    def reestimated(Liste_n_m):
        n_in = len(Liste_n_m)

        x_in = []
        y_in = []

        for i in range(n_in):
            x_in.append(Liste_n_m[i][0])
            y_in.append(Liste_n_m[i][1])

        beta2_m = beta2(variance(x_in,n_in),covariance(x_in,y_in,n_in))
        beta1_m = beta1(x_in, y_in, beta2_m)

        m = [beta1_m,beta2_m]

        return m
```

Nous pouvons maintenant comparer la version simplifiée vu dans le TP2 et la version complète que nous venons de voir :

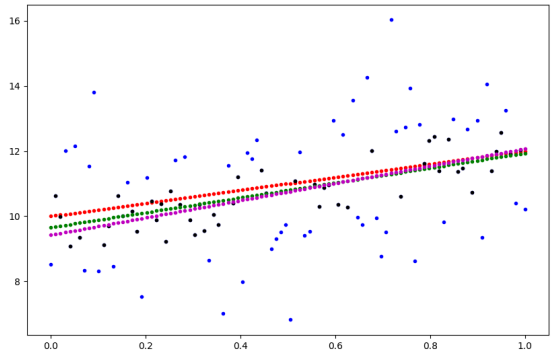
```
    def compare(m1,m2, Treshold):

        beta1_m1 = m1[0]
        beta2_m1 = m1[1]
        beta1_m2 = m2[0]
        beta2_m2 = m2[1]

        n_m1 = 0
        n_m2 = 0

        for i in range(n):
            if abs(beta1_m1 + beta2_m1*x[i] - y1[i]) < Treshold:
                n_m1 += 1

            if abs(beta1_m2 + beta2_m2*x[i] - y1[i]) < Treshold:
                n_m2 += 1

        if n_m1 >= n_m2:
            m2 = calcul_model(n)
            return "TP2"

        else:
            m1 = calcul_model(n)
            return "TP3"
```

Avec

```
m_TP2 = [10.000000000000002, 2.000000000000001]
m_TP3_re = reestimated(Liste_n_m)
```

Les paramètres de m_TP2 ayant été obtenu lors de l'excécution du fichier python TP2.

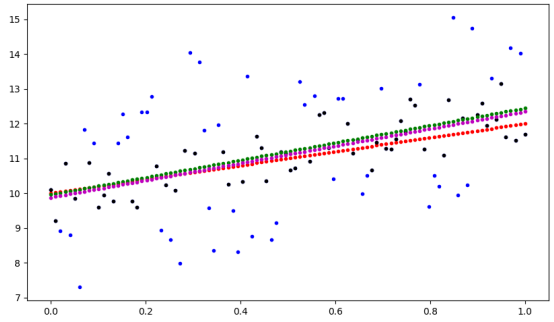
Les résultats quant à la meilleure stratégie ne sont pas toujours les mêmes, la génération des modèles étant aléatoire. Nous pouvons voir un exemple pour chaque cas :



Modèle rouge : algorithme ransac simplifié
Modèle vert : algorithme ransac complet
Modèle violet : algorithme ransac complet réestimé

Dans ce cas, la meilleure stratégie est celle de la version simplifiée

FIGURE 11 – Représentation des données observées



Modèle rouge : algorithme ransac simplifié
Modèle vert : algorithme ransac complet
Modèle violet : algorithme ransac complet réestimé

Dans ce cas, la meilleure stratégie est celle de la version complète

FIGURE 12 – Représentation des données observées

5.2 Moindres carrés non linéaires avec une toolbox

Dans cette partie, nous reprenons les valeurs suivantes déjà utilisées dans le TP2 :

Angles en dégrées	43	45	52	93	108	126
Valeurs obervées	4.7126	4.5542	4.0419	2.2187	1.8910	1.7599

Pour estimer une solution de moindres carrés non linéaires pour ce modèle, nous utilisons scipy.optimize.least_squares, ce qui nous donne :

```
1       param = [0,0] #initialisation des paramètres
2       def fun(parameters, theta, f_theta):
3           return parameters[0] / (1-parameters[1]*np.cos(theta))-f_theta
4
5       solution = optimize.least_squares(fun, param, args = (theta, f_theta))
6
7       beta = solution.x
8
9       print("beta : ", beta)
```

Ainsi, nous obtenons les valeurs suivantes : beta = [3.64013199 -0.35446369] ce qui, par identification, donne les valeurs des paramètres p et e :

$$p = 3.64$$
$$e = -0.35$$

Et pour comparaison, au TP2 nous avions trouvé :

$$p = 3.43$$
$$e = -0.6420$$

Les valeurs trouvées par ces deux méthodes sont très proches. Néanmoins, l'intérêt de cette méthode en utilisant scipy.optimize.least_squares est d'estimer les solutions d'un modèle non-linéaire sans avoir à le linéariser. La difficulté résidait dans la compréhension de cette librairie dont nous n'étions pas familiés.

5.3 Utilisation de la toolbox SVD

Enfin, nous allons aborder la Singular Value Decomposition (SVD) à l'aide de la toolbox svd qui se trouve dans la libraire numpy/linalg de python. Là aussi nous ne connaissions pas cette fonction, néanmoins elle restait assez simple de prise en main.

Pour cela, nous partons d'une matrice assez simple :

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}$$

que nous définissons donc comme ceci sur python :

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

Nous appliquons la méthode SVD à cette matrice en python :

```
1 U,S,V = np.linalg.svd(A)
```

Le code nous renvoie les valeurs de U, S et V :

$$U = \begin{pmatrix} -0.2298477 & 0.88346102 & 0.40824829 \\ -0.52474482 & 0.24078249 & -0.81649658 \\ -0.81964194 & -0.40189603 & 0.40824829 \end{pmatrix}$$

$$S = \begin{pmatrix} 9.52551809 & 0.51430058 \end{pmatrix}$$

$$V = \begin{pmatrix} -0.61962948 & -0.78489445 \\ -0.78489445 & 0.61962948 \end{pmatrix}$$

Nous définissons ensuite la matrice Σ comme suit, à partir de la matrice S :

$$\Sigma = \begin{pmatrix} S_{11} & 0 \\ 0 & S_{12} \\ 0 & 0 \end{pmatrix} = \begin{pmatrix} 9.52551809 & 0 \\ 0 & 0.51430058 \\ 0 & 0 \end{pmatrix}$$

Ce qui donne en python :

```
1 S = np.array([[sigma1,0],[0,sigma2],[0,0]])
```

Enfin, nous pouvons recomposer la matrice A pour vérifier que la décomposition SVD s'est bien faite :

```
A_recompose = U@S@V
```

et nous obtenons bien la matrice A.

Finalement, nous avons vu comment décomposer une matrice selon la méthode SVD à l'aide de la toolbox svd sur python. Cette décomposition permet de sous-tirer 3 matrices de celle de base, à partir desquelles peuvent être estimer notre solution β.

6 Conclusion

Dans ce projet, nous avons pu étudier l'optimisation de modèles mathématiques à travers des méthodes d'optimisation différentiable. L'étude des outliers et des inliners, nous ont permis de mieux comprendre les notions qui sont sous-jacentes aux méthodes d'optimisation. L'optimisation différentiable est une approche puissante pour ajuster les paramètres d'un modèle aux données disponibles. Nous avons aussi examiné les étapes générales de ce processus, notamment la définition d'un modèle mathématique, le calcul des estimateurs, l'estimation du modèle, le choix d'un algorithme d'optimisation approprié et la mise à jour des paramètres du modèle.

Avec ce projet, nous avons pu approfondir notre compréhension des notions vues dans le cours et ainsi mieux comprendre leurs applications dans la modélisation et l'ajustement de données. Cela nous permet de générer des modèles plus précis et plus robustes, capables de représenter au mieux les données étudiées. Bien sûr, l'optimisation de modèles est un domaine vaste et complexe, et il existe de nombreuses autres méthodes non abordées dans ce projet. Nous pouvons par exemple citer les méthodes de Gauss-Newton et de Levenberg-Marquardt qui sont des extensions de l'algorithme du gradient descendant et qui sont plus adaptées à l'ajustement de modèles non-linéaires.