Calcul Scientifique en Python

Licence 2 de Mathématiques à Distance

Fiche de TP

1 Arithmétique

Outils : syntaxe de base du langage, module `math`*.*

Exercice 1. (Logique) Écrire des structures logiques permettant de répondre par `True` ou `False` aux propositions suivantes.

1) L'entier n est divisible par 5.
2) Il existe un triangle de côtés a, b et c (positifs).
3) Les entiers m, n et p sont de même signe.
4) Le point de coordonnées (x, y) est à l'intérieur du cercle centré en O et de rayon r.
5) Le cosinus et le sinus de $\theta \in [0, 2\pi]$ sont positifs.

Exercice 2. (Euclide) On donne le code ci-dessous.

```
def euclide(a, b):
    while b != 0:
        a, b = b, a%b
    return a
```

1) Que calcule la fonction `euclide(a, b)` ?
2) Effectuer quelques tests et ajouter des commandes `print` afin de suivre son déroulement pas à pas.

Exercice 3. (Parfaits et amicaux) On dit que $d \in \mathbb{N}^*$ est un diviseur strict de n s'il divise n mais que $d \neq n$. Un nombre est *parfait* s'il est égal à la somme de ses diviseurs stricts (ex : $6 = 3 + 2 + 1$). Deux nombres sont *amicaux* si l'un est égal à la somme des diviseurs stricts de l'autre et réciproquement.

1) Écrire une fonction `sommediv(n)` qui renvoie le couple `(s, L)` où `L` est la liste des diviseurs stricts de `n` et `s` la somme de ses diviseurs stricts.
2) Écrire une fonction `nbparfaits(n)` qui renvoie la liste des nombres parfaits inférieurs ou égaux à `n`.
3) Écrire une fonction `nbamicaux(n)` qui renvoie la liste des couples de nombres amicaux (distincts) inférieurs ou égaux à `n`.

Exercice 4. (Premiers et jumeaux) On rappelle qu'un entier supérieur ou égal à 2 est *premier* si ses seuls diviseurs positifs sont 1 et lui-même. Deux nombres premiers impairs sont *jumeaux* si leur différence est égale à 2 (ex : (3, 5), (5, 7), ...).

1) Écrire une fonction `estpremier(n)` qui renvoie `True` ou `False` selon que `n` est premier ou non, par recherche exhaustive des diviseurs.

2) Écrire une fonction `eratosthene(n)` qui renvoie la liste `L` des nombres premiers inférieurs ou égaux à `n` par l'algorithme du crible d'Ératosthène.

3) Écrire une fonction `jumeaux(n)` qui renvoie la liste des couples de nombres premiers jumeaux inférieurs ou égaux à `n`.

Exercice 5. (Décomposition) Le *théorème fondamental de l'arithmétique* stipule que tout entier naturel non nul se décompose de manière unique en un produit de nombres premiers (à l'ordre près des facteurs).

1) Écrire une fonction `decomp(n)` renvoyant la liste `L` des nombres premiers de la décomposition de `n`.

2) Améliorer la fonction afin qu'elle puisse regrouper les occurrences identiques en notation puissance. Par exemple pour $1200 = 2^4 \times 3 \times 5^2$, on voudrait passer de la liste `[2, 2, 2, 2, 3, 5, 5]` à la chaîne de caractères `2[4] x 3 x 5[2]` en sortie de la fonction.

Exercice 6. (Partition) Une *partition* de $n \in \mathbb{N}^*$ est une écriture de la forme $n = a_0 + a_1 + \cdots + a_k$ telle que ses coefficients satisfont $a_j \geqslant a_{j+1} \geqslant 1$. On dit que deux partitions p et p' vérifient $p \prec p'$ si $k \leqslant k'$ et $a_j \leqslant a'_j$ pour tout $0 \leqslant j \leqslant k$. On code une partition par la liste décroissante $p = [a_0, \ldots, a_k]$.

1) Écrire une fonction `estpart(n, p)` renvoyant `True` ou `False` selon que `p` est une partition de `n` ou non.

2) Écrire une fonction `listepart(n, p)` qui retourne la liste `L` des partitions p' de $n+1$ telles que $p \prec p'$.

2 Algèbre linéaire

Outils : syntaxe de base du langage, module `numpy` (sous-module `numpy.linalg`).

Tutoriel 1. Suivre la présentation de `numpy`.

Exercice 7. (Déterminant) On cherche à effectuer manuellement le calcul du déterminant d'une matrice carrée par une approche récursive.

1) Écrire une fonction `det2d(M)` prenant une matrice `M` de taille 2×2 en entrée et renvoyant son déterminant.

2) Écrire une fonction `det3d(M)` prenant une matrice `M` de taille 3×3 en entrée et renvoyant son déterminant, qui utilise `det2d` pour faire ses calculs.

3) Généraliser le programme pour obtenir une fonction `detnd(M)` prenant une matrice `M` de taille $n \times n$ en entrée et renvoyant son déterminant par une approche récursive.

4) Comparer vos résultats avec la commande `det` du module `np.linalg`.

Exercice 8. (Spectre et inverse) La commande `eig` du module `np.linalg` permet de récupérer les valeurs propres d'une matrice carrée et les vecteurs propres qui leur sont associés.

1) Effectuer quelques tests pour en comprendre le fonctionnement.

2) Construire P et D dans l'écriture $M = PDP^{-1}$ (on prendra des exemples où M est diagonalisable). On aura besoin de chercher la commande de `np.linalg` permettant d'inverser une matrice.

3) Résoudre le système

$$\begin{cases} x & - & y & + & 2z & = & 3 \\ -x & + & 2y & + & 3z & = & -7 \\ & & -y & + & z & = & 1. \end{cases}$$

Exercice 9. (Rang) On rappelle que le *rang* d'une matrice est le nombre maximal de vecteurs lignes (ou colonnes) linéairement indépendants qu'elle contient. C'est aussi la dimension du sous-espace vectoriel qu'elle engendre.

1) Chercher une commande dans le module `np.linalg` permettant de déterminer le rang d'une matrice.
2) Effectuer quelques tests avec les matrices

$$M = \begin{pmatrix} 1 & -1 & 2 & 1 & 2 \\ -1 & 2 & 3 & -4 & 1 \\ 0 & -1 & 1 & 0 & 0 \end{pmatrix} \quad \text{et} \quad N = \begin{pmatrix} 1 & 2 & 3 & 16 & 0 & 17 \\ -2 & -1 & -3 & -14 & 0 & -16 \\ -1 & -2 & -3 & -16 & 0 & -17 \end{pmatrix}$$

ou toute autre matrice de votre choix.

Exercice 10. (Cayley-Hamilton) Le *théorème de Cayley-Hamilton* stipule que toute matrice carrée annule son polynôme caractéristique.

1) Écrire une fonction `puissmat(M, n)` qui renvoie la matrice M^n. On pourra comparer en termes de temps de calcul les performances de cette fonction avec celles de `np.linalg.matrix_power` pour des grandes valeurs de n.
2) Vérifier Cayley-Hamilton à l'aide de la fonction `puissmat` dans quelques cas simples où polynôme se calcule facilement à la main (en particulier dans les cas 2×2 et 3×3).

3 Suites numériques

Outils : syntaxe de base du langage, module ***numpy****, module* ***matplotlib*** *(sous-module* ***matplotlib.pyplot****).*

Tutoriel 2. Suivre la présentation de `matplotlib`.

Exercice 11. (Périodique) On définit la suite (u_n) par

$$\forall\, n \in \mathbb{N}, \qquad u_n = \operatorname{pgcd}\left(n^2 + 4n + 3, n^3 + 3n - 5\right).$$

1) Définir une fonction `u(n)` renvoyant le terme u_n.
2) Afficher les 30 premiers termes de la suite. Semble-t-elle périodique ?
3) Afficher beaucoup plus de termes. La suite semble-t-elle périodique ? Si oui, déterminer sa période.

Exercice 12. (Convergence et vitesse) On définit la suite (u_n) par

$$\forall\, n \in \mathbb{N}^*, \qquad u_n = \left(1 + \frac{1}{n}\right)^n.$$

1) Montrer qu'il existe un réel ℓ tel que $u_n \to \ell$ lorsque $n \to +\infty$.
2) Représenter sur un même graphique l'évolution de la suite ainsi que la limite ℓ.
3) Déterminer un équivalent asymptotique de la différence $|u_n - \ell|$.
4) Illustrer la vitesse de convergence de la suite par une représentation log-log.

Exercice 13. (Récurrence) Soit (u_n) la suite définie par $u_0 = -1$ et, pour tout $n \in \mathbb{N}$, par $u_{n+1} = f(u_n)$, où la fonction est définie sur $D_f = [-2, +\infty[$ par $f(x) = 2\sqrt{x+3}$.

1) Étudier les variations de f sur son domaine.
2) Montrer que la suite est croissante et majorée par 6.
3) Effectuer des simulations pour conjecturer que la suite converge vers une certaine valeur ℓ. Démontrer ce résultat de simulation.

Exercice 14. (Proies-prédateurs) On aimerait décrire l'évolution de deux espèces de poissons dans la Loire sous la forme d'un système de proies/prédateurs. Les proies sont les truites, les prédateurs sont les brochets. On désigne par $m_T(n)$ et $m_B(n)$ les masses (en kg) de truites et de brochets dans la zone considérée, à l'instant n. Entre les instants n et $n+1$, les populations évoluent de la manière suivante :

- Une masse $\alpha m_T(n)$ de nouvelles truites nait ($\alpha > 0$).
- Une masse $\beta m_T(n) m_B(n)$ de truites nourrit les brochets ($\beta > 0$) et sert à leur reproduction.
- Une masse γ de brochets nait pour chaque kg de truites consommé ($0 \leqslant \gamma \leqslant 1$).
- Une masse $\delta m_B(n)$ de brochets meurt ($0 \leqslant \delta \leqslant 1$).

1) Mettre le système en équations et écrire une fonction `simuTB(a, b, c, d, N, t0, b0)` simulant la population pour $1 \leqslant n \leqslant N$ avec $m_T(0) = t_0$ et $m_B(0) = b_0$. Faire les représentations graphiques adéquates.
2) Tester différents jeux de paramètres.

4 Analyse

Outils : syntaxe de base du langage, module `numpy` *(sous-module* `numpy.linalg`*), module* `matplotlib` *(sous-module* `matplotlib.pyplot`*), module* `scipy` *(sous-modules* `scipy.optimize` *et* `scipy.integrate`*).*

Exercice 15. (Dichotomie) On cherche à résoudre une équation de la forme $f(x) = 0$ sur un intervalle $I = [a, b]$ tel que $f(a)f(b) < 0$. On suppose l'existence d'une unique solution α sur I. Si f est continue, c'est toujours le cas (quitte à réduire l'intervalle I). La méthode de la dichotomie consiste à :

- Initialiser $a_0 = a$ et $b_0 = b$.
- Pour tout $n \geqslant 0$, calculer le milieu m_n de l'intervalle $[a_n, b_n]$. Ensuite :
  - Si $f(m_n) = 0$, alors $a_{n+1} = b_{n+1} = m_n$.
  - Sinon si $f(a_n)f(m_n) > 0$, alors $a_{n+1} = m_n$ et $b_{n+1} = b_n$.
  - Sinon, $a_{n+1} = a_n$ et $b_{n+1} = m_n$.

1) Montrer que la suite (m_n) converge vers α.
2) Écrire une fonction `dichotomie(f, a, b, err)` prenant une fonction `f` en entrée et cherchant par la méthode de la dichotomie la solution de l'équation $f(x) = 0$ sur $[a, b]$ avec une précision de `err`, c'est-à-dire que l'algorithme s'arrête quand l'intervalle d'étude est de longueur strictement inférieure à `err`.
3) Effectuer quelques tests, par exemple avec la fonction $g(x) = x\tan(x) - 1$ ou la fonction $h(x) = \mathrm{e}^x - 2$. On choisira une précision de 10^{-8}.

Exercice 16. (Méthode des rectangles) On se donne une fonction f définie sur un intervalle $[s, t]$ et on cherche à approcher numériquement la valeur de

$$I = \int_s^t f(x)\,\mathrm{d}x.$$

De nombreux algorithmes existent, on propose ici la *méthode des rectangles au point milieu.*

1) Écrire une fonction `rect(f, s, t, n)` qui prend en paramètres une fonction `f` ainsi que deux nombres `s` et `t` et un entier positif `n`, qui représente graphiquement la fonction f sur l'intervalle $[s, t]$ et, superposée, une série de n rectangles de même largeur, couvrant $[s, t]$ et tels que le rectangle de largeur $[x_i, x_{i+1}]$ est de hauteur $f(\frac{x_i + x_{i+1}}{2})$. On pourra commencer par faire un schéma pour bien saisir la problématique.
2) Enrichir la fonction précédente pour qu'elle renvoie `I`, une approximation de l'intégrale donnée dans l'énoncé, à l'aide des rectangles.
3) Effectuer des tests avec des fonctions faciles à intégrer. Tester aussi avec la fonction définie sur $[0, 1]$ par

$$f(x) = x\,(1 - x)\sin^2(200\,x\,(1 - x))$$

pour laquelle on pourra trouver une valeur approchée de I à l'aide des procédures de `scipy.integrate`.

Exercice 17. (Interpolation) Soit une fonction f définie sur un intervalle $[s,t]$. On cherche l'unique fonction polynomiale d'ordre 2 qui coïncide avec f en trois points : les deux bornes de l'intervalle et le point milieu. Concrètement, sur l'intervalle $[s,t]$ pour $s<t$, on cherche une fonction $P(x)=u\,x^2+v\,x+w$ qui vérifie $P(s)=f(s)$, $P(m)=f(m)$ et $P(t)=f(t)$, où $m=\frac{s+t}{2}$. Ces égalités donnent lieu à trois équations que l'on peut résumer par le système

$$\begin{pmatrix} s^2 & s & 1 \\ m^2 & m & 1 \\ t^2 & t & 1 \end{pmatrix} \begin{pmatrix} u \\ v \\ w \end{pmatrix} = \begin{pmatrix} f(s) \\ f(m) \\ f(t) \end{pmatrix}.$$

1) Écrire une fonction `poly2(f, s, t)` qui prend en paramètres une fonction `f` ainsi que deux nombres `s` et `t`, et qui renvoie le triplet (u,v,w) satisfaisant la relation ci-dessus.
2) Écrire une fonction `poly2graph(f, s, t)` prenant le même triplet en paramètre et qui représente graphiquement la fonction f sur l'intervalle $[s,t]$ et, superposé, le graphique de la fonction P.
3) Tester les programmes avec les fonctions définies sur $[0,1]$ par

$$f(x)=\frac{x\,\tan x}{\sqrt{1+x^2}} \quad \text{et} \quad g(x)=\begin{cases} 0 & \text{si } x\leqslant \frac{1}{2} \\ x-\frac{1}{2} & \text{si } x>\frac{1}{2}. \end{cases}$$

Exercice 18. (Suites de fonctions) On cherche à illustrer la convergence simple et la convergence uniforme (ou plutôt, l'absence de convergence uniforme) de certaines suites de fonctions.

1) On sait que la suite de fonctions définie par $f_n(x)=x\,(1-x^n)$ pour $n\geqslant 0$ converge simplement sur $[0,1]$. Identifier cette limite et proposer une illustration.
2) Faire de même avec la suite de fonctions définie par $f_n(x)=\sin^n(x)$ pour $n\geqslant 0$ sur $[0,\pi]$.
3) On sait que la suite de fonctions définie par $f_n(x)=n^2\,x\,\mathrm{e}^{-nx}$ pour $n\geqslant 0$ converge simplement vers la fonction nulle sur $[0,1]$ mais qu'elle converge uniformément sur $]0,1]$ uniquement. Illustrer ce phénomène.
4) Reprendre la question précédente avec $f_n(x)=n\,x\,\mathrm{e}^{-nx}$ puis avec $f_n(x)=n\,x\,\mathrm{e}^{-n^2x}$.

Exercice 19. (Zéros et extrema) Le module `scipy` propose un grand nombre d'outils pour la résolution d'équations (mais aussi l'intégration, la dérivation, les équations différentielles, etc.).

1) Commencer par montrer que la fonction polynomiale $p(x)=x^4-5x^2+4$ admet 4 racines sur l'intervalle $[-3,3]$ puis retrouver ces valeurs à l'aide de la commande `bisect` du module `scipy.optimize`.
2) Effectuer des recherches d'optima locaux à l'aide de la commande `fmin` (minimum et maximum). Reprendre la question en passant par les racines de p'.
3) On aura compris de par son nom que `bisect` utilise la méthode de la dichotomie. Chercher une fonction de `scipy.optimize` permettant d'utiliser la méthode de Newton.

5 Simulation

*Outils : syntaxe de base du langage, module **`numpy`** (sous-module **`numpy.random`**), module **`matplotlib`** (sous-module **`matplotlib.pyplot`**).*

Exercice 20. (Pseudo-aléatoire) Dans tout logiciel de calcul numérique, on trouve un générateur de loi uniforme sur $[0,1]$. Plus précisément, avant de simuler des variables aléatoires de lois diverses, il s'agit de savoir générer un hasard 'de base' qui correspond à une suite de variables aléatoires indépendantes de loi uniforme sur $[0,1]$. Celle-ci se fait à partir d'une initialisation réellement aléatoire (basée par exemple sur l'heure de démarrage du programme, la température, etc.) puis par l'intermédiaire d'une suite de nombres 'pseudo-aléatoires', c'est-à-dire une suite déterministe mais ayant un comportement chaotique. Cette simulation ne peut être parfaite : l'ensemble est non-dénombrable mais l'erreur commise est considérée comme négligeable. Sur Python, la génération de la loi uniforme peut se faire à l'aide de la commande `rand()` du module `numpy.random`.

1) Construire un échantillon de taille 500 de la loi uniforme sur $[0,1]$ et tracer l'histogramme des fréquences qui lui est associé avec `plt.hist`.
2) On se propose de recréer une suite de nombres pseudo-aléatoires. Écrire une fonction `unif(n, y0)` permettant de générer une suite de n nombres pseudo-aléatoires selon la formule

$$x_n = \frac{y_n}{N} \quad \text{avec} \quad y_{n+1} = (ay_n + b) \bmod N$$

pour $a = 16807$, $b = 0$, $N = 2^{31} - 1$ et $y_0 > 0$ passé en paramètre. Générer quelques échantillons et tracer les histogrammes associés.
3) Initialisez y_0 comme votre voisin(e) et comparez vos simulations. Est-ce vraiment aléatoire ?

Exercice 21. (Pile ou face) On considère le jeu de pile ou face entre deux joueurs A et B. Si A fait 'pile' alors B lui donne 1 euro et si A fait 'face' c'est lui qui donne 1 euro à B. Chacun dispose d'une mise initiale, n_A pour A et n_B pour B, et le jeu s'arrête lorsqu'un joueur est ruiné.
1) Écrire une fonction `ruine(nA, nB)` renvoyant les évolutions $(a_0, a_1, \ldots, a_n)$ et $(b_0, b_1, \ldots, b_n)$ des sommes à disposition de A et de B au cours du jeu, pour lequel on a appelé n le dernier lancer. On cherchera comment simuler des variables de loi de Bernoulli.
2) Écrire une fonction `trajectoire(nA, nB)` permettant de tracer l'évolution de ces deux suites pour une partie donnée. On veillera à générer le graphe sous la forme d'une fonction en escalier.
3) Reprendre les questions précédentes en faisant en sorte que la pièce n'est plus équilibrée mais qu'elle donne 'pile' avec probabilité p et 'face' avec probabilité $1 - p$.
4) Simuler un grand nombre de parties et calculer la fréquence avec laquelle A gagne. Comparer avec p.

Exercice 22. (Koh-Lanta) Dans cet exercice comme dans le suivant, on déduit par simulation une approximation (plus amusante qu'efficace) de π. On simule ici un tir à l'arc.
1) Commencer par écrire une fonction `unifrectangle(a, b, c, d)` qui renvoie un couple (u_1, u_2) simulé de manière uniforme et indépendante dans $[a, b] \times [c, d]$.
2) Par la suite, on choisit $a = c = -1$ et $b = d = 1$. Écrire une fonction `tracercible()` qui représente graphiquement le carré $[-1, 1] \times [-1, 1]$ ainsi que le cercle inscrit dans ce carré. Le disque délimité par ce cercle jouera le rôle de la cible. Pour ajuster l'échelle sur les deux axes, on pourra utiliser `plt.axis("scaled")`.
3) On suppose que le tireur à l'arc, qui n'est pas un expert, répartit ses flèches uniformément dans le carré $[-1, 1] \times [-1, 1]$. Proposer une fonction `simulerfleches(n)` qui simule `n` tirs et qui affiche sur le graphique uniquement les flèches ayant atteint la cible.
4) En vertu d'un célèbre résultat, la *loi des grands nombres*, on peut affirmer que la proportion de flèches qui atteignent la cible est une bonne approximation de la probabilité qu'à chaque flèche d'atteindre la cible (à condition que le nombre de flèches soit suffisamment grand). Vérifier expérimentalement ce résultat.
5) Pour aller plus loin, écrire une fonction `calculerscore(n)` qui, à l'issue de `n` flèches, détermine le score obtenu par le tireur (avec une convention à choisir : par exemple 0 à l'extérieur de la cible et ensuite de 1 à 5 selon des anneaux concentriques de même épaisseur, à représenter sur le graphique).

Exercice 23. (Aiguille de Buffon) L'aiguille de Buffon est une expérience statistique proposée dès le XVIIIème siècle par le mathématicien français Buffon. Le concept est assez simple : en répétant un grand nombre de fois une situation aléatoire bien particulière que l'on va décrire, on peut aboutir à une approximation de π. On considère un parquet composé de planches parallèles sur lequel une aiguille est lancée un grand nombre de fois. On compte le nombre de fois où l'aiguille touche au moins deux planches du parquet. Les lames du parquet sont de même largeur $\ell > 0$ et l'aiguille est modélisée par un segment de longueur $0 < a \leqslant \ell$. À chaque lancer, on note $0 \leqslant d \leqslant \frac{\ell}{2}$ la distance du centre de l'aiguille à la rainure la plus proche et $\theta \in [0, \frac{\pi}{2}]$ l'angle formé par l'aiguille avec les rainures, et ces deux quantités sont considérées comme générées uniformément et indépendamment.

1) Faire un schéma pour bien saisir le problème.
2) Montrer que la condition pour que l'aiguille coupe une rainure est $d \leqslant \frac{a}{2} \sin\theta$. Sans entrer dans les détails techniques, on peut montrer que la probabilité qu'une aiguille coupe une rainure vaut $\frac{2a}{\pi\ell}$.
3) Écrire une fonction `tracerparquet(N, l)` qui représente graphiquement un parquet composé de `N` lames parallèles de largeur `l`.
4) Écrire une fonction `lanceraiguilles(n, a, N, l)` qui simule le lancer de `n` aiguilles de longueur `a` et qui les représente graphiquement sur un parquet de `N` lames de largeur `l`. On pourra réutiliser la fonction `unifrectangle` de l'exercice précédent.
5) Étudier l'évolution de la suite $(\pi_n = \frac{2a}{\ell F_n})_{n \geqslant 1}$ où F_n est la fréquence avec laquelle les n aiguilles ont coupé une rainure du parquet.

Exercice 24. (Régression) Considérons un ensemble de points aléatoires $(x_i, y_i)_{1 \leqslant i \leqslant n}$ générés selon le schéma

$$\forall\, 1 \leqslant i \leqslant n \quad x_i \sim \mathcal{N}(2,2), \quad \varepsilon_i \sim \mathcal{N}(0,1) \quad \text{et} \quad y_i = a + b\, x_i + \varepsilon_i$$

où a et b sont deux paramètres réels.

1) Écrire une fonction `simulernuage(a, b, n)` renvoyant les vecteurs $(x_1, \ldots, x_n)$ et $(y_1, \ldots, y_n)$. On cherchera comment simuler des variables de loi normale.
2) On peut montrer que les valeurs des paramètres qui minimisent la somme des erreurs quadratiques $\sum_{i=1}^n (y_i - a - b\, x_i)^2 = \sum_{i=1}^n \varepsilon_i^2$ sont données par

$$b^* = \frac{\gamma_{xy}}{v_x^2} \quad \text{et} \quad a^* = m_y - b^*\, m_x$$

où

$$m_x = \frac{1}{n}\sum_{i=1}^n x_i, \quad m_y = \frac{1}{n}\sum_{i=1}^n y_i, \quad v_x^2 = \frac{1}{n}\sum_{i=1}^n (x_i - m_x)^2 \quad \text{et} \quad \gamma_{xy} = \frac{1}{n}\sum_{i=1}^n (x_i - m_x)(y_i - m_y).$$

La droite $y = a^* + b^* x$ est appelée *droite de régression*. Écrire une fonction `nuage(a, b, n)` simulant un nuage de `n` points, calculant l'équation de la droite de régression et effectuant les représentations graphiques associées.