1 – Étalonnage de l'axe dynamométrique

Des données sont enregistrées dans un fichier sous forme de chaînes de caractères. Il s'agit d'extraire ces données du fichier et de les préparer à un traitement numérique.

▶ Régression linéaire par la méthode des moindres carrés

▶ Q6. Écart entre tension mesurée et tension modélisée

La tension mesurée est u_i. D'après le modèle linéaire, la tension associée au poids P_i est (aP_i+b). Pour tout $0 \leqslant i < N$, l'écart est donc

$$\varepsilon_i = u_i - (aP_i + b).$$

▶ Q7. Variables S_{PP} et S_{Pu}

On reconnaît l'expression du produit scalaire.

```
SPP = np.dot(P, P)
SPu = np.dot(P, u)
```

▶ Q8. Variables S_P et S_u

On reconnaît cette fois le produit scalaire avec le vecteur $(1,\dots,1)$.

```
N   = len(P)
uns = np.ones(N)
SP  = np.dot(P, uns)
Su  = np.dot(u, uns)
```

▶ Q9. Coefficients de la droite de régression

Il suffit de coder les calculs indiqués par l'énoncé. La valeur de N a été définie à la question précédente.

```
a = N*(Spu - SP*Su)/(N*SPP - SP**2)
b = (SP*SPu - Su*SPP)/(SP**2 - N*SPP)
```

▶ **Pour les paresseux**

Il s'agit en fait de trouver l'unique solution

$$x = \begin{pmatrix} a \\ b \end{pmatrix}$$

au sens des moindres carrés de l'équation

$$A.x = B \quad \text{où} \quad A = \begin{pmatrix} P_0 & 1 \\ \vdots & \vdots \\ P_{N-1} & 1 \end{pmatrix} \quad \text{et} \quad B = \begin{pmatrix} U_0 \\ \vdots \\ U_{N-1} \end{pmatrix}.$$

Évidemment, tous ces calculs sont déjà programmés !

```
from scipy.linalg import lstsq    # pour least squares

a, b = lstsq(A, B)[0]
```

Plus de détails dans la documentation en ligne de `scipy.linalg`.

▶ **Q10. Tracé**

```
# les points expérimentaux
plt.plot(P, u, 'o')
# le modèle linéaire
absP = np.linspace(0, 600)
ordU = a*absP + b
plt.plot(absP, ordU)
```

2 – Filtrage de la tension de sortie

▶ Filtrage par moyenne glissante

▶ **Q11. Calcul pratique de moyenne glissante**

On calcule une moyenne glissante sur $n = 3$ données :
— pour $0 \leqslant i < n - 1 = 2$, on calcule la moyenne sur les $(i + 1)$ premières données ;
— pour $2 \leqslant i \leqslant 4$, on calcule la moyenne sur les $n = 3$ dernières données.

i	0	1	2	3	4
u_i	-2	0	-1	2,5	3
u_i^f	-2	-1	-1	0,5	1,5

▶ **Q12. Code du calcul précédent**

On reprend le même calcul en deux étapes : pour $0 \leqslant i < n - 1$ d'abord, puis pour $n - 1 \leqslant i$ ensuite.
Pour la seconde étape, il est commode de ne pas prendre les indices donnés par l'énoncé. On remplace $n - 1 \leqslant i < N$ par $j = i - n + 1 = i - (n - 1)$ et on a donc

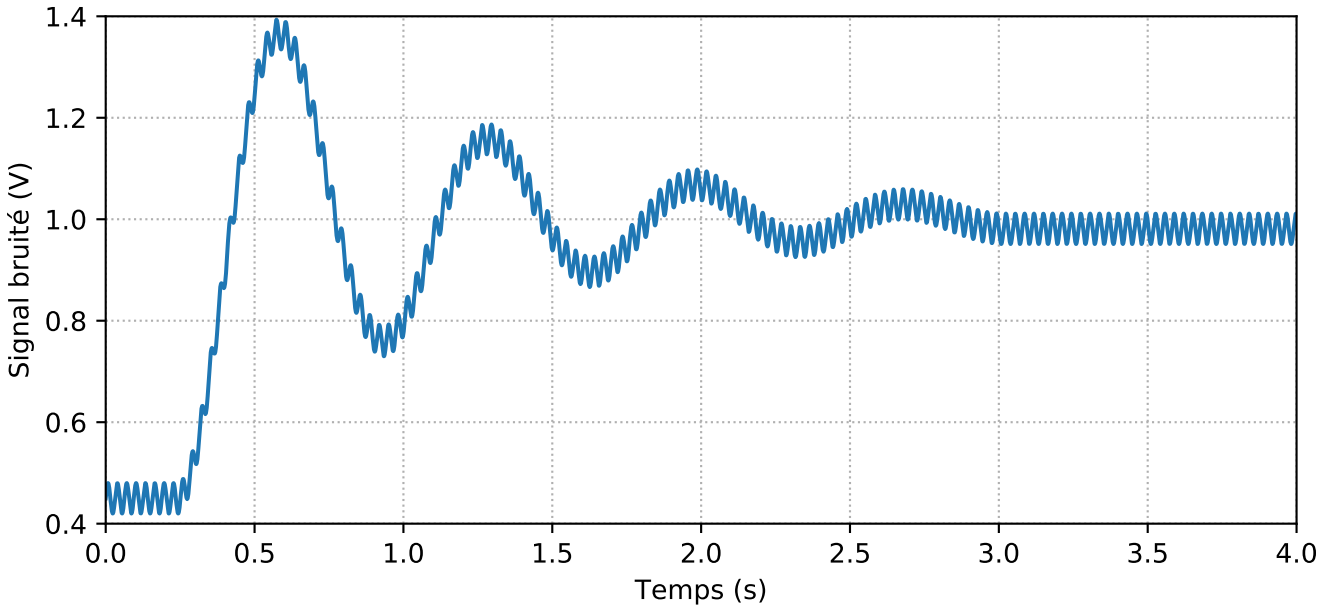
$$u_i^f = u_{j+n-1}^f = \frac{1}{n} \sum_{k=j}^{j+n-1} u_k$$

pour $0 \leqslant j < N - n + 1$.

```
def filtre_mg(u, n):
    N  = len(u)
    uf = np.ones(N)
    # D'abord (n-1) moyennes sur les (i+1) premières données
    for i in range(n-1):
        uf[i] = np.sum(u[:i+1])/(i+1)
    # puis N-(n-1)=N-n+1 moyennes sur n données
    for j in range(N-n+1):
        uf[j+n-1] = np.sum(u[j:j+n])/n
    return uf
```

► **Q13. Filtrages d'ordres différents**

On part d'un signal bruité dont la période d'échantillonnage est égale à 8.10^{-4}s (soit 1250 points par seconde).

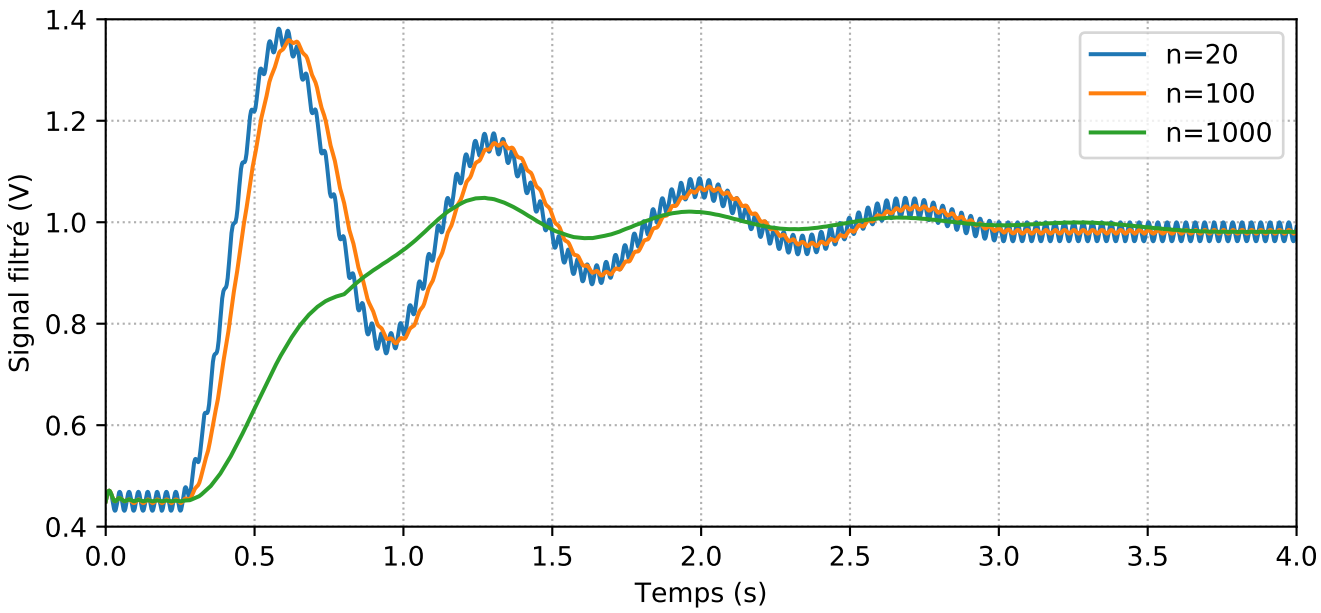


Plus l'ordre n de la moyenne glissante augmente, plus le signal filtré est lissé.

— Pour $n = 20$, on note une *atténuation* des oscillations à haute fréquence.

— Pour $n = 100$, les oscillations à haute fréquence ont *disparu*. On peut aussi remarquer que les maxima locaux sont *retardés*.

— Pour $n = 1000$, ce ne sont plus seulement les oscillations à haute fréquence qui disparaissent, on constate aussi une *perte du signal*.



REMARQUE.— Dans le dernier cas, on calcule une moyenne mobile sur une durée de près d'une seconde, ce qui est à peu près égal à la pseudo-période du signal...

▸ Filtrage par filtre passe-bas

▸ Q14. Schéma d'Euler explicite

❧ Le schéma d'Euler explicite consiste à approcher la dérivée $\frac{du_f(t)}{dt}$ par le taux d'accroissement $\frac{u_f(t_{i+1})-u_f(t_i)}{t_{i+1}-t_i}$. On en déduit que

$$\begin{aligned}\forall\, 0 \leqslant i < N, \quad u^f_{i+1} &= u^f_i + \frac{(t_{i+1}-t_i)(u_i-u^f_i)}{\tau}\\ &= \left(1-\frac{t_{i+1}-t_i}{\tau}\right)\cdot u^f_i + \frac{t_{i+1}-t_i}{\tau}\cdot u_i.\end{aligned}$$

❧ Pour appliquer la relation de récurrence précédente, il faut choisir une valeur initiale. Nous conviendrons donc que

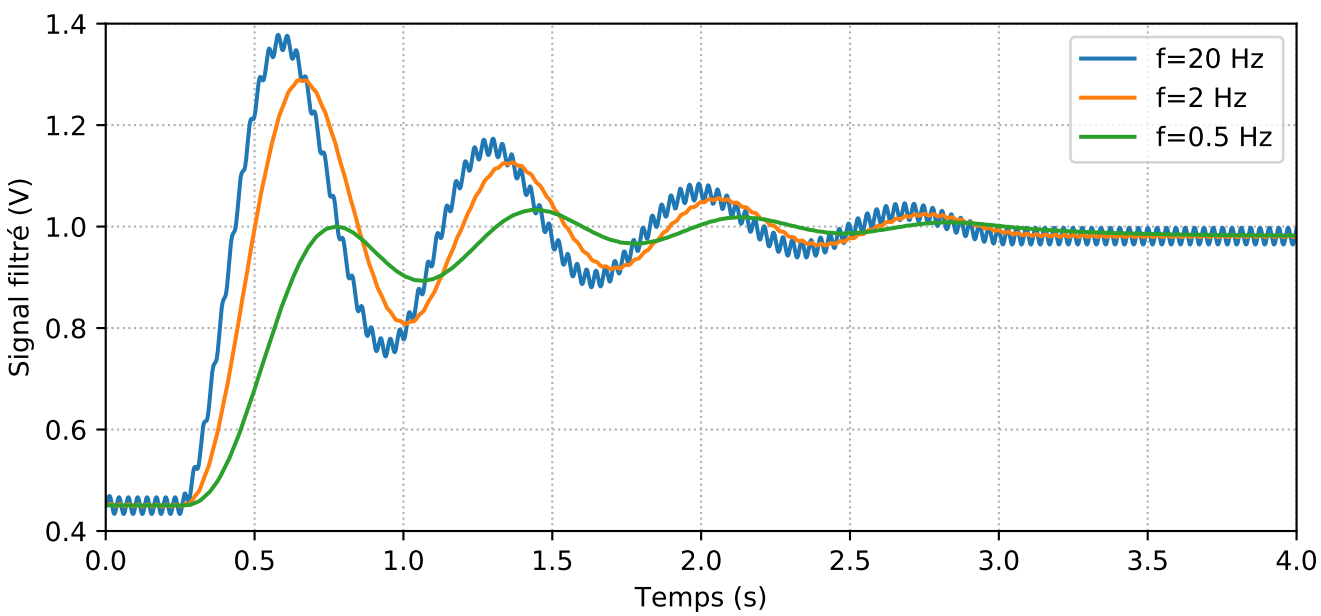
$$u^f_0 = u_0.$$

▸ Q15. Code du filtre passe-bas

Le code est sans mystère.
Plutôt que de diviser par $\tau = 1/2\pi f$, je préfère multiplier par $2\pi f$ (puisque f est un argument de la fonction).

```
def filtre_pb(u, temps, f):
  ppf = 2*np.pi*f
  N = len(u)
  uf = np.ones(N)
  uf[0] = u[0]
  for i in range(N-1):
      k = ppf*(temps[i+1]-temps[i])
      uf[i+1] = (1-k)*uf[i] + k*u[i]
  return uf
```

▸ Q16. Influence de la fréquence de coupure



Plus la fréquence de coupure est basse, plus le signal est lissé.
— Pour la fréquence de coupure $f_c = 20$ Hz, les oscillations sont atténuées.
— Pour $f_c = 2$ Hz, les oscillations sont devenues imperceptibles et on note que le signal est un peu retardé.
— Pour $f_c = 0,5$ Hz, on n'a pas seulement traité le bruit, on a aussi modifié le signal...

REMARQUE.— Le signal bruité considéré fait apparaître une pseudo-période de l'ordre de la seconde et un bruit périodique (ô oxymore!) dont la fréquence est environ 30 Hz. Dans ces conditions, un filtre passe-bas avec une fréquence de coupure supérieure à 100 Hz devrait laisser le signal bruité absolument intact et même une fréquence de coupure de 20 Hz ne devrait pas avoir d'impact notable sur le signal.

Je reste donc perplexe devant la légende de la figure qui se trouve dans l'énoncé.

▶ Comparaison des méthodes

▶ Q17. Complexités des méthodes

❧ Dans les deux cas, le calcul du signal filtré repose essentiellement sur une boucle `for`, avec un nombre d'itérations égal à la longueur du signal à traiter (complexité *linéaire*).

❧ La complexité du filtre passe-bas est indépendante de la fréquence de coupure : chaque itération effectue un petit nombre d'opérations arithmétiques.

❧ En revanche, la complexité du filtre par moyenne glissante augmente avec l'ordre du filtre et on pourrait estimer naïvement que la complexité de l'instruction `sum(S[j:j+n])` est proportionnelle à l'ordre n du filtre.

L'Astuce taupinale nous permet de limiter les calculs au minimum :

$$\frac{1}{n}\sum_{k=(j+1)}^{j+n} u_k = \frac{1}{n}\left(u_{j+n} + \sum_{k=j}^{j+n-1} u_k - u_j\right) = \frac{u_{j+n}-u_j}{n} + \frac{1}{n}\sum_{k=j}^{j+n-1} u_k.$$

Les deux méthodes sont donc de complexités temporelles (et spatiales !) équivalentes.

3 – Exploitation des données d'un étalonnage

▶ Q18. Première requête

La requête est simple, sa traduction en langage SQL est directe.

```
SELECT COUNT(*) AS "Nb_étalonnages"
FROM etalonnages
WHERE numero_capteur=3
```

REMARQUE.— Le renommage du résultat est facultatif (mais bien utile en pratique).

▶ Q19. Deuxième requête

Les dates d'étalonnage et les coefficients de corrélation se trouvent dans la table `etalonnages`, tandis que le nom de l'entreprise ayant acquis le matériel est dans la table `capteurs`, il faut donc effectuer une jointure sur les deux tables.

Cette jointure se fait sur le numéro de série de l'axe dynamométrique, c'est-à-dire l'attribut `num` de la table `capteurs` (clé primaire dans cette table) et l'attribut `numero_capteur` de la table `etalonnages`.

```
SELECT C.date_etalonnage AS DATE, E.R2 AS CoeffCorrélation
FROM capteurs AS C
JOIN etalonnages AS E
ON C.num=E.numero_capteur
WHERE C.entreprise="XXX"
```

REMARQUE.— Le renommage des deux tables permet d'alléger très sensiblement l'écriture de la requête.

▶ Q20. Troisième requête

Même remarque sur la nécessité d'une jointure.

En outre, il faut utiliser une fonction d'agrégation (`AVG() ... GROUP BY ...`) pour calculer l'information voulue à partir des données dont on dispose et une fonction de tri (`ORDER BY ...`) pour présenter les résultats.

```
SELECT C.Entreprise AS NomEntreprise, AVG(E.R2) AS R2_Moyen
FROM capteurs AS C
JOIN etalonnages AS E
ON C.num=E.numero_capteur
GROUP BY NomEntreprise
ORDER BY NomEntreprise
```