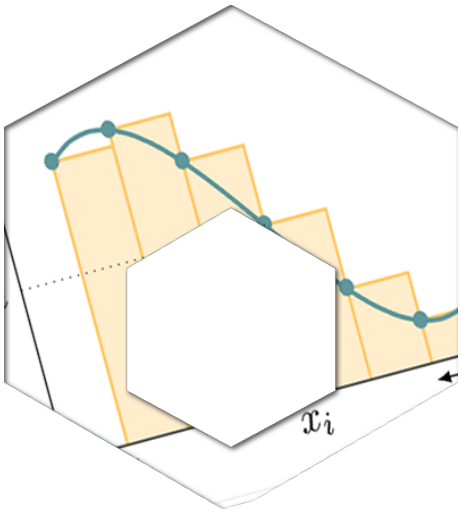
CPGE PTSI-PT Lycée Jean Zay - Thiers

Ingénierie Numérique

Cours

v1.0

Lycée Jean Zay – 21 rue Jean Zay – 63300 Thiers – Académie de Clermont-Ferrand



Compétences visées :

A3-07	Analyser un algorithme.
C1-03	Choisir une démarche de résolution d'un problème d'ingénierie numérique ou d'intelligence artificielle.
C3-02	Résoudre numériquement une équation ou un système d'équations.
C3-03	Résoudre un problème en utilisant une solution d'intelligence artificielle.
D3-03	Effectuer des traitements à partir de données.

Table des matières

1 Introduction

1.1 Objectif

Ce cours a pour objectif de lister les différentes méthodes d'analyse numérique évoquées dans le programme de Sciences de l'Ingénieur en PTSI-PT. Elles formeront un premier étage dans la construction des connaissances en ce domaine, connaissances importantes pour la plupart des ingénieurs qui travaillent dans le secteur de la simulation numérique notamment. L'ensemble des algorithmes présentés seront écrit en Python, conformément aux recommandations du programme.

Ce cours n'a en revanche pas vocation à mener une étude approfondie des concepts mathématiques sous-jacents et des limites de chacune des méthodes abordées. Il pourra donc être utile de se référer à la littérature spécialisée pour les étudier plus finement.

Attention

Pour faire le lien avec le cours d'Informatique du Tronc Commun, la plupart des fonctions Python proposées dans ce cours seront codées en Python pur. Seuls quelques exemples feront appels à des bibliothèques particulières telles que `numpy` (notamment au §3).

Cependant, en règle générale, il est bien plus simple et plus efficace d'utiliser les bibliothèques spécialisées (`numpy`, `scipy`, etc).

1.2 Enjeux de la simulation numérique

Lors de la phase de conception d'un système, il est long, complexe et coûteux de réaliser un prototype afin de le soumettre à des tests pour valider les choix de conception.

La simulation numérique, qui est une représentation de phénomènes physiques complexes à l'aide de modèles mathématiques simulés par des opérations faites sur un ordinateur, est une solution plus économique. En effet, le prototype est virtuel et facilement modifiable. La procédure de simulation est :

- rapide à mettre en place (quelques heures) ;
- peut être faite n'importe où et n'importe quand ;
- permet d'avoir une plus-value en offrant une meilleure compréhension et interprétation du comportement du produit dans des situations parfois difficiles voire impossibles à réaliser expérimentalement (par exemple la connaissance de la température à l'intérieur d'un matériau).

1.3 Mise en œuvre pratique de la simulation numérique

La plupart des modèles mathématiques font intervenir des équations différentielles linéaires ou non linéaires : il s'agit donc d'être capable de déterminer l'évolution d'une grandeur physique au cours du temps mais également en fonction de l'espace.

On procède alors à une discrétisation du problème qui consiste à découper les signaux et le temps en petits intervalles de manière à simplifier les études et on notera pour la suite de ce cours :

- $t = [t_i]$, $i \in [1, n]$ la suite des valeurs discrètes du temps en partant de zéro ($t_0 = 0$) ;
- $y = [y_i = y(t_i)]$ la suite des évaluations à chaque instant discret t_i d'une grandeur physique $y(t)$.

À partir de ces définitions, il sera possible de résoudre des équations différentielles linéaires selon différentes méthodes sachant que les opérations numériques élémentaires sont la dérivation et l'intégration.

Remarque

Dans la suite de ce cours, nous utiliserons dans les exemples la fonction polynômiale suivante (voir FIGURE 1 ci-contre) :

$$f(x) = \frac{1}{8}x^3 - \frac{3}{2}x^2 + 4x + 4$$

Dans les différents exemples de code Python, on considèrera comme prédéfinie la fonction `f` définie comme suit :

```
1 def f(x):
2     return 1/8*x**3 - 3/2*x**2 + 4*x + 4
```

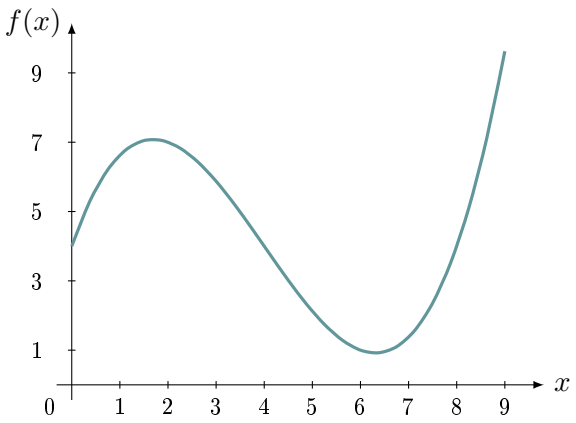


FIGURE 1 – Fonction polynômiale exemple

2 Intégration et dérivation numériques

2.1 Intégration numérique

2.1.1 Principe

L'intégration numérique consiste à intégrer (de façon approchée) une fonction $f(x)$ sur un intervalle borné $[a, b]$, c'est-à-dire à calculer l'aire I sous la courbe représentant la fonction, à partir d'un calcul ou d'une mesure en un nombre fini de points.

$$I = \int_a^b f(x)dx = \sum_{i=0}^{n-1} \int_{x_i}^{x_{i+1}} f(x)dx$$

L'approche numérique pour l'intégration est nécessaire pour le traitement de données expérimentales ou de résultats de simulation complexes, ou encore pour intégrer des fonctions n'ayant pas de primitive connue analytiquement. Elle ne permet néanmoins pas d'obtenir des résultats corrects pour n'importe quelle fonction : en particulier, la fonction ne doit pas présenter de branche infinie dans l'intervalle d'intégration.

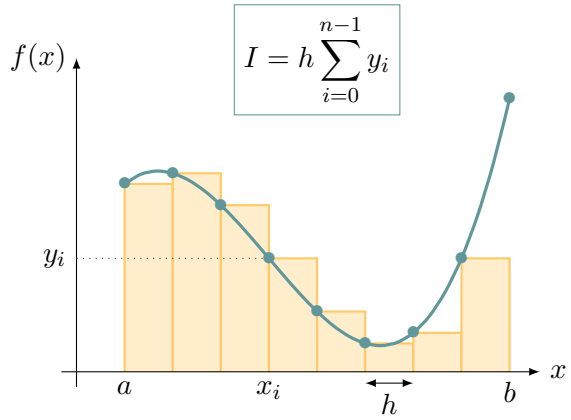
La répartition des points en abscisse est généralement uniforme (*pas* d'échantillonnage constant h), mais il existe des méthodes à pas variable, ou encore à pas adaptatif.

La précision de l'intégration numérique peut s'améliorer en augmentant le nombre de points n (en diminuant le *pas* d'échantillonnage h) ou en augmentant le degré de l'interpolation polynomiale (sous réserve de bonnes propriétés de continuité de la courbe).

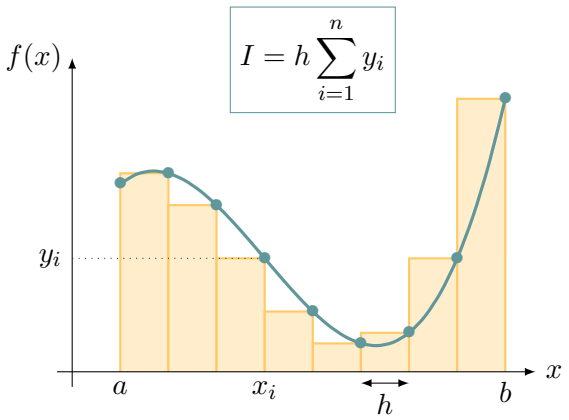
2.1.2 Méthode des rectangles

Dans cette méthode, on calcule l'intégrale numérique en réalisant une somme de surfaces de rectangles. Le domaine d'intégration est découpé en intervalles et on fait comme si la fonction restait constante sur chaque intervalle.

On a alors 2 approximations possibles :



(a) Méthode des rectangles à gauche



(b) Méthode des rectangles à droite

FIGURE 2 – Méthodes des rectangles

Cette méthode est très utilisée pour les intégrations en temps réel. L'intégrale calculée à l'instant t est en réalité légèrement retardée de $\frac{h}{2}$ (en moyenne), ce qui n'est pas gênant lorsque h est très faible.

La programmation classique de l'intégration numérique d'une courbe définie par deux tableaux d'abscisses `x` et d'ordonnées `y` s'écrit alors (pour un échantillonnage constant ou non) :

```
# Création des listes, avec a et b les bornes d'intégration et n le nombre de points calculés
# On peut aussi obtenir x et y en lisant un fichier de mesures
x = np.linspace(a,b,n)          # Création d'un vecteur abcisses
y = f(x)                        # Calcul de l'image de x par la fonction f
```

```
def integration_rectangles_gauche(x,y):
    """
    Paramètres :
    - x : liste des abscisses
    - y : liste des ordonnées
    """
    I = 0
    for i in range(len(x)-1):
        I += y[i]*(x[i+1]-x[i])
    return I
```

```
def integration_rectangles_droite(x,y):
    """
    Paramètres :
    - x : liste des abscisses
    - y : liste des ordonnées
    """
    I = 0
    for i in range(1,len(x)):
        I += y[i]*(x[i]-x[i-1])
    return I
```

2.1.3 Méthode du point milieu

La méthode du point milieu (hors programme) consiste à considérer la fonction constante sur chaque intervalle de largeur $2h$ et égale à la valeur prise par le point au milieu de l'intervalle (FIGURE 3).

La valeur approchée de l'intégrale s'écrit alors (pour n impair) :

$$I = 2h\sum_{i=1}^{\frac{n-1}{2}} y_{2i-1}$$

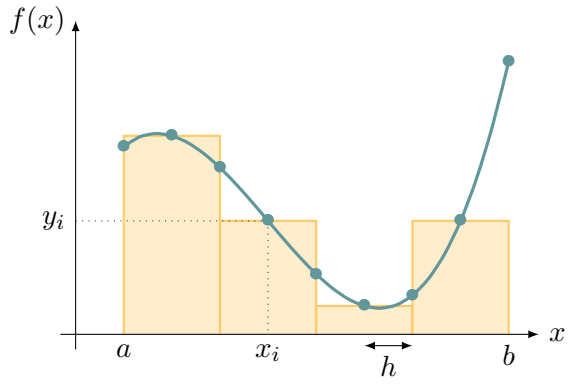


FIGURE 3 – Méthode du point milieu

La programmation classique de la méthode du point milieu pour une courbe définie par deux tableaux d'abscisses `x` et d'ordonnées `y` s'écrit alors (pour un échantillonnage constant ou non, avec un nombre impair de points) :

```
1  def integration_point_milieu(x,y):
2      """
3      Paramètres :
4      - x : liste des abscisses
5      - y : liste des ordonnées
6      """
7      I = 0
8      for i in range(1,len(x)-1,2):
9          I += y[i]*(x[i+1]-x[i-1])
10     return I
```

2.1.4 Méthode des trapèzes

Comme son nom l'indique, cette méthode d'intégration utilise une somme de surfaces de trapèzes, de base h.

La valeur approchée de l'intégrale s'écrit alors :

$$I = h \sum_{i=0}^{n-1} \frac{y_{i+1} + y_i}{2}$$

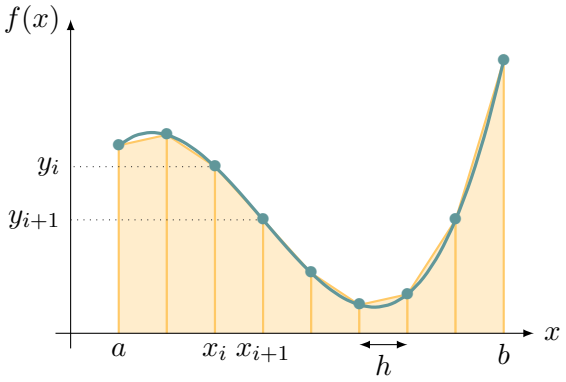


FIGURE 4 – Méthode des trapèzes

La programmation classique de la méthode des trapèzes pour une courbe définie par deux tableaux d'abscisses `x` et d'ordonnées `y` s'écrit alors (pour un échantillonnage constant ou non) :

```
1  def integration_trapezes(x,y):
2      """
3      Paramètres :
4      - x : liste des abscisses
5      - y : liste des ordonnées
6      """
7      I = 0
8      for i in range(len(x)-1):
9          I += (y[i]+y[i+1])*(x[i+1]-x[i])/2
10     return I
```

2.1.5 Comparaison des différentes méthodes

La fonction utilisée comme exemple pour ce cours étant un polynôme, il est aisé de calculer la valeur exacte de son intégrale entre 2 bornes définies. Dès lors, on peut tracer le graphique de la FIGURE 5, représentant l'écart obtenu (en %) en fonction du nombre de points pour chaque méthode numérique.

On constate alors que le choix de la valeur du pas est important et peut avoir de sérieuses répercussions sur la précision du calcul numérique :

- si le pas est trop grand, la FIGURE 5 indique que l'erreur croît rapidement à mesure que le pas augmente ;

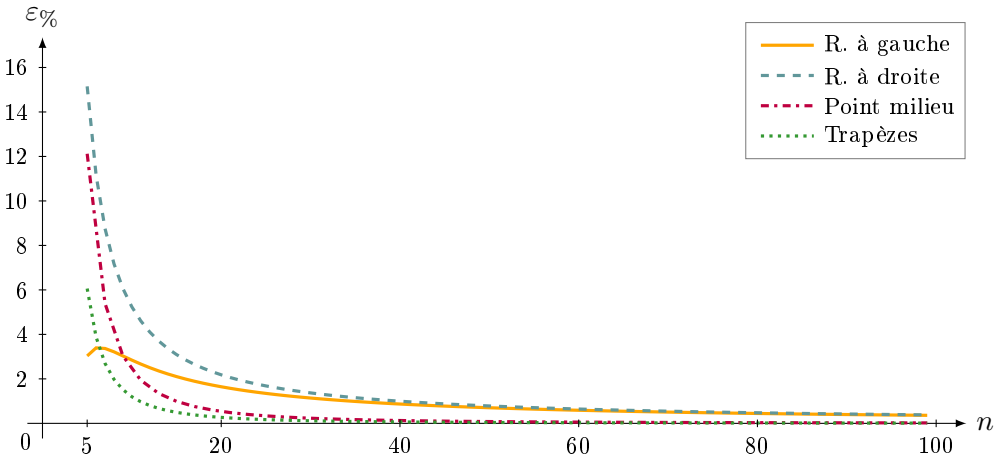


FIGURE 5 – Comparaison des méthodes de calcul d'intégrale

- si le pas est trop petit, on risque d'augmenter les erreurs numériques liées à la représentation des flottants en Python.

Le choix du pas de calcul se fera donc à l'aune de ces deux remarques et sera le plus souvent guidé par l'expérience, tant qu'il n'est pas contraint par la nature des données recueillies lors d'une acquisition par exemple (ou le choix sera limité par le nombre de points mesurés).

2.2 Dérivation numérique

2.2.1 Principe

La dérivation numérique permet de poser les bases utiles pour le paragraphe suivant sur l'intégration des équations différentielles. Elle s'avère par ailleurs très utile pour le traitement de données expérimentales.

La dérivation numérique consiste à dériver (de façon approchée) une fonction sur un intervalle borné $[a, b]$, c'est-à-dire à déterminer la pente de la courbe représentant la fonction, à partir d'un calcul ou d'une mesure en un nombre fini de points.

La répartition des points en abscisse est généralement uniforme (*pas* d'échantillonnage constant h), mais il existe des méthodes à pas variable, ou encore à pas adaptatif.

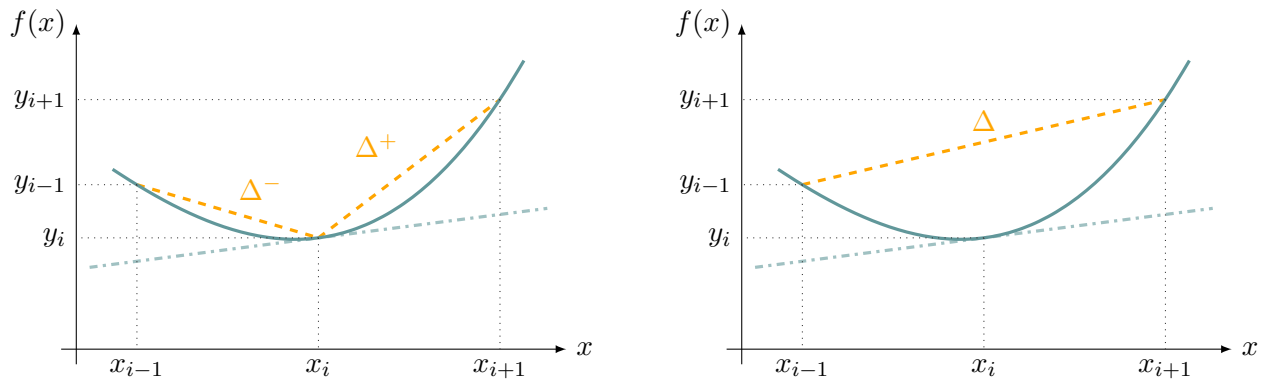
2.2.2 Méthode à 1 pas

L'estimation de la dérivée la plus simple consiste à calculer la pente à partir du point courant et du point précédent ou suivant (voir FIGURE 6a). L'estimation de la dérivée au point `i` peut alors se calculer par :

- différence avant : $D_i = \frac{1}{h}(y_{i+1} - y_i)$ (pente de la droite Δ^+) ;
- différence arrière : $D_i = \frac{1}{h}(y_i - y_{i-1})$ (pente de la droite Δ^-).

Bien entendu, lorsqu'il s'agit de dériver une fonction « en temps réel », le point (x_{i+1}, y_{i+1}) n'est pas connu. La seule méthode possible est donc celle de la différence arrière.

Notons aussi que le calcul de la dérivée conduit à un tableau de valeurs de dimension `n-1`.



(a) Méthodes à 1 pas

(b) Méthode à 2 pas

FIGURE 6 – Dérivation numérique

La programmation classique des deux méthodes de dérivation à 1 pas d'une courbe définie par deux tableaux d'abscisses `x` et d'ordonnées `y` s'écrit alors (pour un échantillonnage constant ou non) :

```
1  def derivation_1_pas_arriere(x,y):
2      """
3      Paramètres :
4      - x : liste des abscisses
5      - y : liste des ordonnées
6      """
7      deriv = []
8      for i in range(1,len(x)):
9          deriv.append((y[i]-y[i-1])/(x[i
       ]-x[i-1]))
10     return deriv
```

```
1  def derivation_1_pas_avant(x,y):
2      """
3      Paramètres :
4      - x : liste des abscisses
5      - y : liste des ordonnées
6      """
7      deriv = []
8      for i in range(len(x)-1):
9          deriv.append((y[i+1]-y[i])/(x[i
       +1]-x[i]))
10     return deriv
```

2.2.3 Méthode à 2 pas

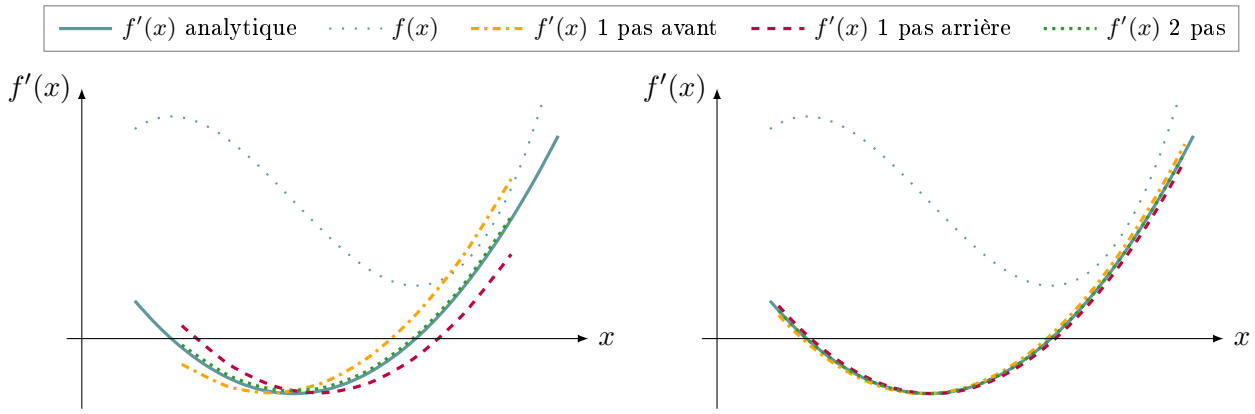
On peut aussi approximer la pente du segment reliant le point précédent au point suivant (voir FIGURE 6b). L'estimation de la dérivée au point `i` peut alors se calculer par : $D_i = \frac{1}{h}(y_{i+1} - y_{i-1})$ (pente de la droite Δ).

Cette méthode faisant appel au point (x_{i+1}, y_{i+1}) , elle ne peut être utilisée pour un traitement de données en temps réel.

La programmation classique de la méthode de dérivation à 2 pas d'une courbe définie par deux tableaux d'abscisses `x` et d'ordonnées `y` s'écrit alors (pour un échantillonnage constant ou non) :

```
1  def derivation_2_pas(x,y):
2      """
3      Paramètres :
4      - x : liste des abscisses
5      - y : liste des ordonnées
6      """
7      deriv = []
8      for i in range(1,len(x)-1):
9          deriv.append((y[i+1]-y[i-1])/(x[i+1]-x[i-1]))
10     return deriv
```

La FIGURE 7 montre les erreurs constatées entre la valeur exacte de la dérivée et les valeurs obtenues par les 3 méthodes en chaque point de l'intervalle $[1, 9]$, en fonction du nombre de points de calcul.



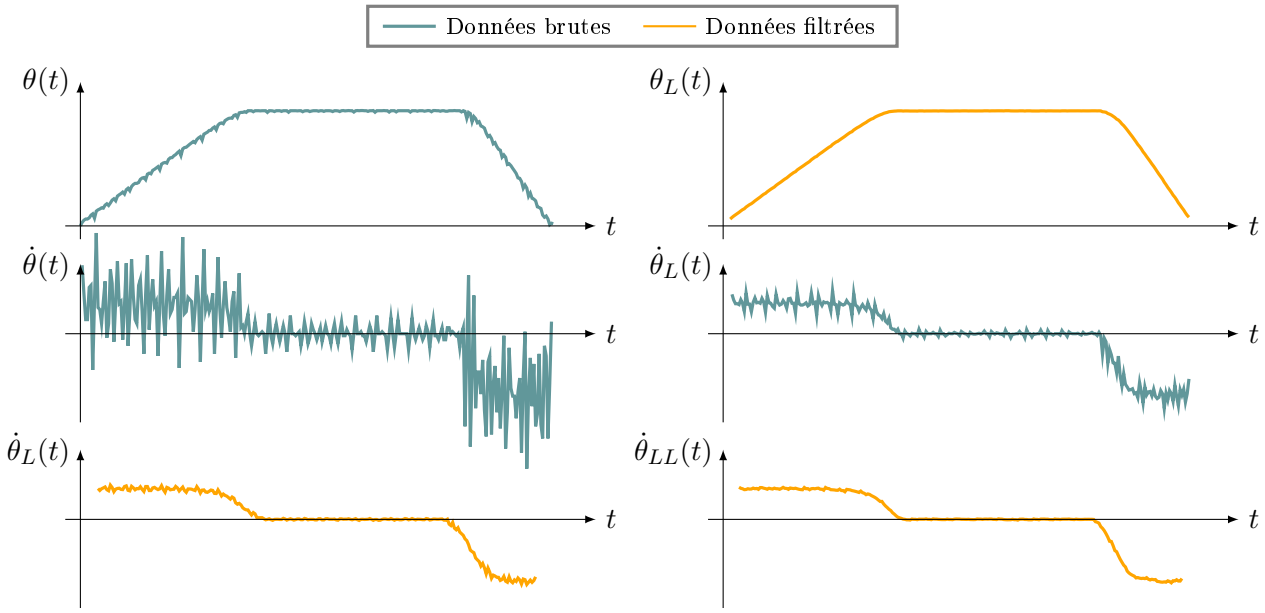
(a) Points calculés : $n = 10$ (b) Points calculés : $n = 50$

FIGURE 7 – Influence du pas de calcul sur la précision de la dérivée numérique

On constate immédiatement sur la FIGURE 7a que la précision du calcul numérique de la dérivée en un point est directement liée aux nombres de points choisis ou disponibles dans le cas d'une mesure. Ces résultats peuvent donc être très éloignés de la réalité quand les données sont trop peu nombreuses.

2.3 Influence du bruit de mesure

Lorsque la courbe est issue d'une mesure, elle est généralement entachée d'un léger bruit, qui peut devenir catastrophique pour l'évaluation de la dérivée (voir FIGURE 8).



(a) Scénario 1 (b) Scénario 2

FIGURE 8 – Mesure d'un angle $\theta(t)$ au cours du temps par un capteur potentiométrique (un lissage donne $\theta_L(t)$). Calcul numérique de la dérivée ($\dot{\theta}(t)$ et $\dot{\theta}_L(t)$).

En effet, si les points de mesure restent « en moyenne » au voisinage de la valeur mesurée, il existe des fluctuations rapides, à la fréquence d'échantillonnage, entre les points successifs.

Le calcul de la dérivée conduit à déterminer la pente entre deux points successifs, ce qui perturbe fortement le signal dérivé et cache les évolutions « lentes » du signal (lentes devant la période d'échantillonnage).

Deux solutions sont possibles :

- calculer la dérivée sur un temps plus long que le temps d'échantillonnage, par exemple avec une méthode à 2 pas. Solution adoptée sur la FIGURE 8a ;
- filtrer (ou lisser) le signal d'origine pour supprimer l'essentiel du bruit (voir §6.3), puis dériver (et filtrer/lisser une nouvelle fois le résultat), comme montré sur la FIGURE 8b.

Dans les deux cas, le signal dérivé sera entaché d'un retard sur le signal d'origine. Pour les traitements différés, le retard ne pose pas de problème mais, pour les systèmes de commande en temps réel, il est nécessaire de trouver un compromis entre la qualité du signal dérivé et le retard.

3 Résolution d'équations différentielles

3.1 Introduction

Une grande part des problèmes scientifiques se modélisent par une équation différentielle dont on cherche la solution pour dimensionner ou comprendre le phénomène.

Quand les équations sont simples (linéaires d'ordre 1 ou 2) la résolution analytique est aisée, mais nombre de modélisations conduisent à des équations non linéaires. Ce cours a pour objectif de proposer des méthodes pour déterminer une solution approchée.

Remarque

Dans ce chapitre, les courbes et tableaux de données correspondent à un problème de mécanique classique : la chute libre d'une bille dans l'huile. En appliquant le PFD à la bille, soumise à la pesanteur et à du frottement visqueux, on aboutit à l'équation différentielle suivante, qui régit l'évolution de la vitesse de la bille :

$$m\frac{dv(t)}{dt} = -fv(t) + mg$$

Sous forme adimensionnée, cette équation peut s'écrire :

$$\frac{dy(t)}{dt} = -y(t)$$

Cette équation présente une solution analytique $y(t) = y_0 e^{-t}$ où y_0 est la condition initiale $y(0) = y_0$.

3.2 Problème de Cauchy

Le problème de Cauchy consiste à trouver les fonctions Y de $[0,T] \to \mathbb{R}^N$, telles que :

$$\begin{cases} \dfrac{dY}{dt} = F(t,Y) \\ Y(t_0) = Y_0 \end{cases} \quad \text{où } t_0 \in [0,T] \text{ et } Y_0 \in \mathbb{R}^N \text{ sont des données.}$$

La plupart des systèmes d'équations différentielles de tout ordre peuvent se mettre sous cette forme de système d'équations différentielles du premier ordre (un exemple sera traité plus loin). Or, le théorème de Cauchy-Lipschitz montre que pour les équations classiquement abordées en SI, le problème de Cauchy admet une solution unique définie sur $[0, T]$.

3.3 Résolution numérique

La résolution numérique consiste à retrouver la courbe d'évolution de la solution $Y(t)$, en calculant un certain nombre de points sur un intervalle de temps donné. Il y a donc échantillonnage du temps.

La forme de l'équation différentielle traduit la loi d'évolution de la grandeur Y : à l'instant t , $\frac{dY}{dt}$ est la pente de $Y(t)$, donc la valeur de $F(t, Y)$ permet d'estimer $Y(t + \Delta t)$, pour Δt petit.

D'un point de vue plus mathématique, l'intégration de l'équation différentielle entre deux pas de temps de l'échantillonnage t_i et t_{i+1} conduit à :

$$\int_{t_i}^{t_{i+1}} \frac{dY}{dt} dt = \int_{t_i}^{t_{i+1}} F(t, Y) dt \qquad \text{on en déduit que : } Y_{i+1} = Y_i + \int_{t_i}^{t_{i+1}} F(t, Y) dt$$

L'intégrale sur un pas de temps $\frac{1}{\Delta t} \int_{t_i}^{t_{i+1}} F(t, Y) dt$ correspond au calcul de la pente moyenne de $Y(t)$ sur le petit intervalle Δt. Le principe des divers schémas numériques consiste à évaluer le plus précisément possible cette valeur de la pente moyenne pour estimer la valeur de Y_{i+1} lorsque Y_i est connu.

Connaissant les conditions initiales Y_0, l'intégration numérique devient le calcul d'une suite, où, à chaque temps t_i, un calcul approché permet de calculer une nouvelle valeur de Y en t_{i+1}.

3.4 Schémas numériques classiques

3.4.1 Le schéma d'Euler explicite

Le schéma d'Euler explicite utilise comme valeur approchée de la pente moyenne sur $[t_i, t_{i+1}]$, la valeur de F en t_i (en notant $\Delta t = t_{i+1} - t_i$). Ainsi :

$$\frac{dY}{dt} = F(t, Y) \quad \text{est approché par} \quad \frac{Y_{i+1} - Y_i}{\Delta t} = F(t_i, Y_i) \quad \Rightarrow \quad Y_{i+1} = Y_i + \Delta t F(t_i, Y_i)$$

La résolution numérique consiste à calculer la suite :

$$\begin{cases} Y_0 & \text{(condition initiale donnée)} \\ Y_{i+1} = Y_i + \Delta t F(t_i, Y_i) \end{cases}$$

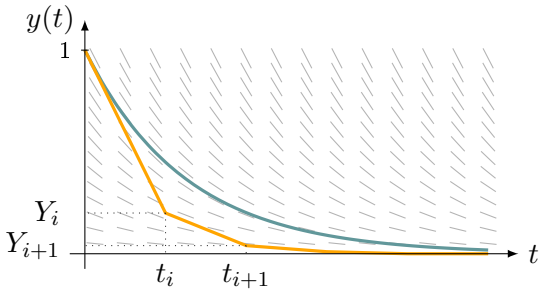


FIGURE 9 – Schéma d'Euler explicite

La FIGURE 9 montre la solution exacte ainsi que la solution approchée par le schéma d'Euler explicite. La fonction $F(t, Y)$, traduisant la pente imposée à la solution, est représentée par des segments inclinés selon une certaine pente. Ainsi, avec le schéma d'Euler explicite, la pente du segment $[t_i, t_{i+1}]$ est imposée par la valeur de F au point approché (t_i, Y_i).

La programmation du schéma d'Euler conduit à définir la fonction `F(t,y)` renvoyant la valeur de $F(t,y)$ et la fonction `euler_explicite(F,y0,t)` renvoyant le tableau des $y(t)$ (`y`) pour chaque valeur du tableau de temps `t`, avec pour condition initiale `y0`. Le code suivant est l'application pour le cas d'étude adimensionnel (voir §3.1) :

```
import numpy as np

# Définition de la fonction F
def F(t,y):
    return -y

# Schéma d'Euler explicite
def euler_explicite(F,y0,t):
    y = np.zeros([t.shape[0],y0.shape[0]])
    y[0,:]= y0
    for i in range(t.shape[0]-1):
        y[i+1,:]= y[i,:]+(t[i+1]-t[i])*F(t[i],y[i,:])
    return y

# Résolution
t = np.linspace(0,6,100)     # Temps
y0 = np.array([1])           # Conditions initiales
y = euler_explicite(F,y0,t)
```

La fonction `euler_explicite` est traitée vectoriellement (grâce à la bibliothèque `numpy`) de façon à pouvoir l'utiliser aussi bien sur une équation simple que sur un système de plusieurs équations différentielles (voir §3.5).

Analyse de la convergence du schéma L'évolution de la solution exacte et des points calculés par le schéma d'Euler explicite pour différents pas de temps sont fournis Figure 10.

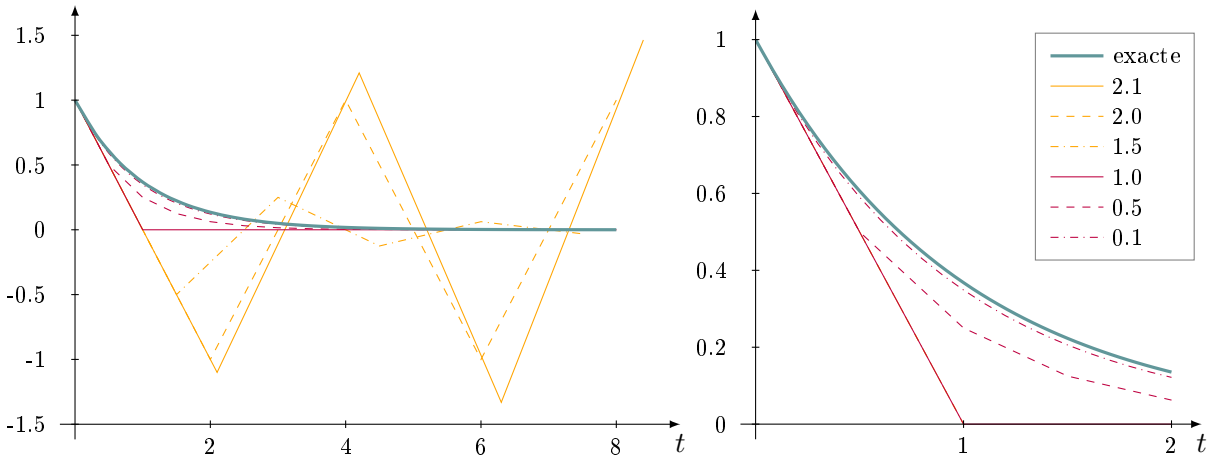


Figure 10 – Solution exacte et numérique pour le schéma d'Euler explicite

Les courbes montrent que le résultat est d'autant plus précis que le pas de temps est petit et que le schéma diverge au-delà d'un pas de 2 s.

L'équation de la chute libre dans l'huile est bien connue en physique et en sciences de l'ingénieur : elle présente une constante de temps de 1 s. Il est donc raisonnable de penser qu'une intégration numérique ne peut traduire correctement le comportement de l'équation différentielle que pour des pas de temps petits devant cette constante de temps, donc pour $\delta t \ll 1\,\text{s}$.

Si $1\,\text{s} < \delta t < 2\,\text{s}$, le schéma converge mais conduit à des oscillations qui n'ont pas lieu d'être. Pour $\delta t \gg 2\,\text{s}$, le schéma ne converge pas.

Cette observation se généralise pour les équations différentielles quelconques : le schéma d'Euler explicite présente des conditions de convergence liées aux constantes de temps du modèle.

Performances du schéma Le tableau de la FIGURE 11 donne l'erreur comme étant le maximum de l'écart entre la solution exacte et la solution approchée ainsi que le temps de calcul pour différentes valeurs du pas de temps.

h	10^{-1}	10^{-2}	10^{-3}	10^{-4}	10^{-5}	10^{-6}
N pas de temps	100	1000	10 000	100 000	1 000 000	10 000 000
Erreur	$4{,}1 \cdot 10^{-1}$	$4{,}9 \cdot 10^{-2}$	$5{,}0 \cdot 10^{-3}$	$5{,}0 \cdot 10^{-4}$	$5{,}0 \cdot 10^{-5}$	$5{,}0 \cdot 10^{-6}$
Temps (s)	$5{,}2 \cdot 10^{-3}$	$1{,}6 \cdot 10^{-2}$	$7{,}8 \cdot 10^{-2}$	$5{,}7 \cdot 10^{-1}$	5,0	50

FIGURE 11 – Performances du schéma d'Euler explicite

L'erreur évolue linéairement en fonction du pas de temps : le schéma est dit « d'ordre 1 ». Il faut descendre à un pas de temps très faible avant d'obtenir un niveau d'erreur satisfaisant.

Le temps de calcul évolue lui aussi linéairement en fonction du pas de temps (complexité en $O(N)$, car une seule boucle).

Pour améliorer la précision d'un facteur 10000, il faut diminuer d'un facteur 10000 le pas de temps et donc augmenter d'un facteur 10000 le temps de calcul.

3.4.2 Le schéma d'Euler implicite

Le schéma d'Euler implicite utilise comme valeur approchée de la pente moyenne sur $[t_i, t_{i+1}]$, la valeur de F en t_{i+1} (en notant $\Delta t = t_{i+1} - t_i$). Ainsi :

$$\frac{dY}{dt} = F(t,Y) \quad \text{est approché par} \quad \frac{Y_{i+1} - Y_i}{\Delta t} = F(t_{i+1}, Y_{i+1}) \quad \Rightarrow \quad Y_{i+1} = Y_i + \Delta t F(t_{i+1}, Y_{i+1})$$

La résolution numérique consiste à calculer la suite :

$$\begin{cases} Y_0 \quad \text{(condition initiale donnée)} \\ Y_{i+1} = Y_i + \Delta t F(t_{i+1}, Y_{i+1}) \end{cases}$$

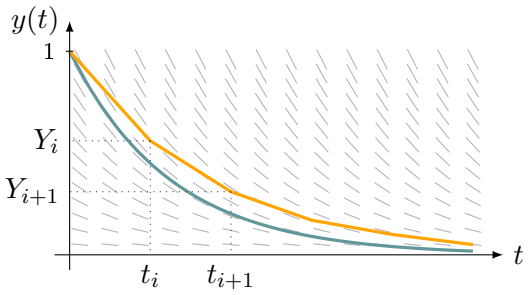


FIGURE 12 – Schéma d'Euler implicite

La FIGURE 12 montre la solution exacte ainsi que la solution approchée par le schéma d'Euler implicite. La fonction $F(t,Y)$, traduisant la pente imposée à la solution, est représentée par des segments inclinés selon une certaine pente. Ainsi, avec le schéma d'Euler explicite, la pente du segment $[t_i, t_{i+1}]$ est imposée par la valeur de F au point approché (t_{i+1}, Y_{i+1}).

À la différence du schéma d'Euler explicite, l'expression de Y_{i+1} devient implicite, c'est-à-dire qu'il se calcule comme la solution d'une équation. Il convient alors de distinguer trois cas :

1. Pour une équation simple, il est parfois possible d'exprimer Y_{i+1} en fonction des données. Le calcul est alors aussi simple que pour le schéma d'Euler explicite.
2. Pour un système d'équations linéaires de grande dimension de type $F(t_{i+1}, Yi+1) = AY_{i+1} + B$, où A et B sont une matrice et un vecteur de grande dimension, le calcul de Y_{i+1} est explicite mais nécessite de résoudre un système à chaque pas de temps.
3. Pour une équation quelconque, non linéaire, il est nécessaire de faire appel à un algorithme de recherche de zéro à chaque pas de temps (voir §4). Le programme Python proposé ci-après couvre ce cas.

Dans la plupart des cas, une méthode implicite est plus coûteuse en temps de calcul qu'une méthode explicite, à pas de temps égal. Cependant, lorsque les conditions de convergence du schéma explicite obligent à un pas très petit, il peut être avantageux d'employer une méthode implicite avec un pas de temps raisonnable.

La programmation du schéma d'Euler dans le troisième cas (le plus général) conduit à définir la fonction `equation(t,y)` renvoyant la valeur de l'équation $Y_{i+1} - Y_i - \Delta t F(t_{i+1}, Y_{i+1})$ à résoudre, la fonction `resoudre(equation,y0,eps)` renvoyant la solution de l'équation, et la fonction `euler_implicite(F,y0,t)` renvoyant le tableau des $y(t)$ pour chaque valeur du tableau de temps `t`, avec pour condition initiale `y0`.

```
import numpy as np

# Définition de la fonction F
def F(t,y):
    return -y

# Equation implicite à résoudre pour déterminer Yi +1
def equation(F,t,y,dt,yi):
    return y-yi-dt*F(t,y)

# Fonction de recherche de zéro par la méthode de la sécante
def resoudre(F,t,dt,yi,eps):
    y = [yi]                    # La recherche débute à yi
    y.append(yi+10*eps)         # Second point proche de yi
    i=0
    while (abs(y[i+1]-y[i])>eps):
        f1 = equation(F,t,y[i],dt,yi)
        f2 = equation(F,t,y[i+1],dt,yi)
        a = (f2-f1)/(y[i+1]-y[i])
        b = f1-a*y[i]
        y.append(-b/a)
        i=i+1
    return y[-1]

# Schéma d'Euler implicite
def euler_implicite(F,y0,t):
    y = np.zeros([t.shape[0],y0.shape[0]])
    y[0,:]= y0
    for i in range(t.shape[0]-1):
        y[i+1,:] = resoudre(F,t[i+1],(t[i+1]-t[i]),y[i,:],1e -4)
    return y

# Résolution
t = np.linspace(0,6,100)    # Temps
y0 = np.array([1])          # Conditions initiales
y = euler_implicite(F,y0,t)
```

Attention

Contrairement à l'intégration par le schéma d'Euler explicite, cet algorithme n'est pas utilisable pour un système d'équations différentielles, car la recherche de zéro dans ce cas n'est valable que pour une équation scalaire.

Analyse de la convergence du schéma L'évolution de la solution exacte et des points calculés par le schéma d'Euler implicite pour différents pas de temps est fournie FIGURE 13.

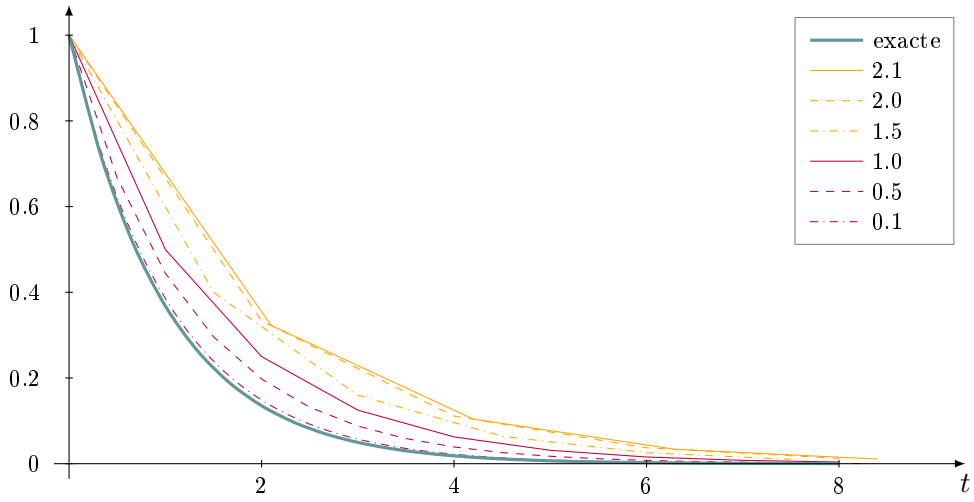


FIGURE 13 – Solution exacte et numérique pour le schéma d'Euler implicite

Les courbes montrent que le résultat est d'autant plus précis que le pas de temps est petit et que le schéma converge pour tous les pas de temps.

Cette observation se généralise pour les équations différentielles quelconques : le schéma d'Euler implicite ne présente pas de condition de convergence.

Performances du schéma Le tableau de la FIGURE 14 donne l'erreur comme étant le maximum de l'écart entre la solution exacte et la solution approchée ainsi que le temps de calcul pour différentes valeurs du pas de temps.

h	10^{-1}	10^{-2}	10^{-3}	10^{-4}	10^{-5}	10^{-6}
N pas de temps	100	1000	10 000	100 000	1 000 000	10 000 000
Erreur	$5{,}9 \cdot 10^{-1}$	$5{,}0 \cdot 10^{-2}$	$5{,}0 \cdot 10^{-3}$	$5{,}0 \cdot 10^{-4}$	$5{,}0 \cdot 10^{-5}$	$5{,}0 \cdot 10^{-6}$
Temps (s)	$5{,}2 \cdot 10^{-3}$	$3{,}1 \cdot 10^{-2}$	$3{,}7 \cdot 10^{-1}$	3,7	37	370

FIGURE 14 – Performances du schéma d'Euler implicite

L'erreur évolue linéairement en fonction du pas de temps : le schéma est dit « d'ordre 1 ». Il faut descendre à un pas de temps très faible avant d'obtenir un niveau d'erreur satisfaisant.

Le temps de calcul évolue lui aussi linéairement en fonction du pas de temps (complexité en $O(N)$, car une seule boucle) et s'avère plus long que dans le cas explicite du fait de la résolution de l'équation implicite.

Pour améliorer la précision d'un facteur 10000, il faut diminuer d'un facteur 10000 le pas de temps et donc augmenter d'un facteur 10000 le temps de calcul.

3.5 Cas des équations différentielles d'ordre supérieur à 1

Les équations différentielles mises en œuvre en physique ou en ingénierie sont généralement d'ordre supérieur à 1 (souvent d'ordre 2). Il est cependant possible d'exprimer une équation différentielle scalaire d'ordre n sous la forme d'un système de n équations différentielles d'ordre 1.

Soit l'équation différentielle :

$$\frac{d^n y}{dt^n} = f\left(t, y, \frac{dy}{dt}, \ldots, \frac{d^{n-1}y}{dt^{n-1}}\right)$$

En posant un vecteur Y (appelé vecteur d'état) contenant $y_1 = y$ et toutes les dérivées contenues dans f : $y_2 = \frac{dy_1}{dt}$, $y_3 = \frac{d^2y}{dt^2}$ et ainsi de suite jusqu'à $y_n = \frac{d^{n-1}y}{dt^{n-1}}$, l'équation différentielle s'exprime sous la forme d'un système différentiel :

$$\begin{cases} \frac{dy_1}{dt} = y_2 \\ \frac{dy_2}{dt} = y_3 \\ \vdots \\ \frac{dy_n}{dt} = f(t, y_1, y_2, \ldots, y_n) \end{cases} \quad \text{soit} \quad \frac{dY}{dt} = F(t, Y)$$

Avec :

$$Y = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_{n-1} \\ y_n \end{pmatrix} \quad \text{et} \quad F(t, Y) = \begin{pmatrix} y_2 \\ y_3 \\ \vdots \\ y_n \\ f(t, y_1, y_2, \ldots, y_n) \end{pmatrix}$$

Autre solution Une autre solution classique permettant de résoudre cette équation différentielle avec la méthode d'Euler explicite est d'appliquer deux fois la définition.

On note Yp la dérivée première. On a $Yp_i = \dfrac{Y_{i+1} - Y_i}{h}$

On note Ypp la dérivée seconde. On a $Ypp_i = \dfrac{Yp_{i+1} - Yp_i}{h} = \dfrac{Y_{i+2} - 2Y_{i+1} + Y_i}{h^2}$.

On obtient alors une relation de récurrence directe qui peut être programmée. Ce schéma est un schéma d'intégration à deux pas qui pourrait s'inscrire dans un cadre plus général des méthodes à plusieurs pas.

Exemples de mise en équation La FIGURE 15 présente 3 exemples de mise en équation de problèmes classiques :

1. charge d'un condensateur : évolution de la tension $u(t)$ dans un circuit RC ;
2. chute libre d'une bille dans l'huile : évolution de l'altitude $z(t)$;
3. pendule simple sans frottement : évolution de l'angle $\theta(t)$;
4. système masse-ressort-amortisseur : évolution de l'abscisse $x(t)$.

Ex.	Equa. diff.	$Y(t)$	$Y'(t)$	F
1	$\dot{u}(t) = \frac{1}{RC}(E - u(t))$	-	-	$f(u,t) = \frac{1}{RC}(E - u(t))$
2	$\ddot{z}(t) = g - \frac{f}{m}\dot{z}(t)$	$Z(t) = \begin{pmatrix} z(t) \\ \dot{z}(t) \end{pmatrix}$	$Z'(t) = \begin{pmatrix} \dot{z}(t) \\ F(Z,t) \end{pmatrix}$	$F(Z,t) = g - \frac{f}{m}\dot{z}(t)$
3	$\ddot{\theta}(t) = -\frac{g}{l}\sin\theta(t)$	$\Theta(t) = \begin{pmatrix} \theta(t) \\ \dot{\theta}(t) \end{pmatrix}$	$\Theta'(t) = \begin{pmatrix} \dot{\theta}(t) \\ F(\Theta,t) \end{pmatrix}$	$F(\Theta,t) = -\frac{g}{l}\sin\theta(t)$
4	$m\ddot{x}(t) + c\dot{x}(t) + kx(t) = f(t)$	$X(t) = \begin{pmatrix} x(t) \\ \dot{x}(t) \end{pmatrix}$	$X'(t) = \begin{pmatrix} \dot{x}(t) \\ F(X,t) \end{pmatrix}$	$F(X,t) = \frac{1}{m}\left(f(t) - c\dot{x}(t) - kx(t)\right)$

FIGURE 15 – Quelques exemples de mise en équation

3.6 Bibliothèques de calcul numérique

Python dispose de bibliothèques de calcul numérique optimisées : il n'est donc pas nécessaire de refaire le programme d'intégration numérique à chaque problème. Sous Python, la bibliothèque à charger est `scipy.integrate`. Il est nécessaire de définir une fonction `F` admettant comme arguments le temps `t` et le vecteur d'état `Y`, ainsi que les temps de début et de fin et un vecteur de conditions initiales. On peut ajouter explicitement le vecteur de temps pour lequel on souhaite récupérer les résultats.

Les 2 exemple ci-dessous traitent les cas 1 et 4 de la FIGURE 15. Ainsi, vous pourrez voir décrit le code pour des équations différentielles du premier et du deuxième ordre.

```
"""
Charge d'un condensateur
"""
import scipy.integrate as integrate
import numpy as np

# Définition de la fonction f
def f(t,u):
    R = 10      # Résistance
    C = 0.01    # Capacité
    E = 5       # Echelon de tension
    return (E-u)/(R*C)

# Résolution
T = 0.8             # Durée de l'étude
u0 = np.array([0])  # Conditions initiales
t = np.linspace(0,T,100) # Temps
u = integrate.solve_ivp(f,[0,T],u0)
```

```
"""
Masse-ressort-amortisseur
"""
import scipy.integrate as integrate
import numpy as np

# Définition de la fonction F
def F(t,Y):
    m = 1       # Masse
    c = 0.4     # Amortissement
    k = 1       # Raideur
    F0 = 10     # Echelon de force
    f1 = Y[1]
    f2 = 1/m*(F0-c*Y[1]-k*Y[0])
    return array([f1,f2])

# Résolution
T = 20              # Durée de l'étude
Y0 = np.array([0,0]) # Conditions initiales
t = np.linspace(0,T,100) # Temps
z = integrate.solve_ivp(F,[0,T],Y0)
```

4 Résolution d'équations de type $f(x) = 0$

4.1 Introduction

Un grand nombre de problèmes de physique ou d'ingénierie débouchent sur des équations non linéaires qu'il est impossible de résoudre analytiquement. L'outil numérique est alors indispensable

pour trouver des solutions approchées de ces équations. Ce chapitre est dédié à la présentation de deux méthodes classiques pour résoudre des équations (linéaires ou non) à une inconnue, c'est à dire déterminer x solution d'une équation $f(x) = 0$, où f est une fonction quelconque.

Remarque

Pour ce chapitre 4, nous utiliserons dans les exemples la fonction polynômiale du début de ce cours, et on cherchera à résoudre l'équation suivante :

$$f(x) = 3 \quad \Leftrightarrow \quad \frac{1}{8}x^3 - \frac{3}{2}x^2 + 4x + 4 = 3 \quad \Leftrightarrow \quad \frac{1}{8}x^3 - \frac{3}{2}x^2 + 4x + 1 = 0$$

On cherchera une solution sur l'intervalle $[2, 6]$.

On considère donc pour la suite de ce chapitre 4 que la fonction g est définie :

```
1 # Fonction g
2 def g(x):
3     return 1/8*x**3 - 3/2*x**2 + 4*x + 1
```

4.2 Méthode de la dichotomie

La méthode de la dichotomie est la plus simple et convient poiur les fonctions f continues pour lesquelles est *a priori* connu un intervalle $[a, b]$ tel que les signes de $f(a)$ et $f(b)$ soient opposés. Par conséquent, il existe au moins une solution de l'équation $f(x) = 0$ sur l'intervalle $[a, b]$. On suppose par la suite qu'il n'y a qu'une seule solution sur cet intervalle.

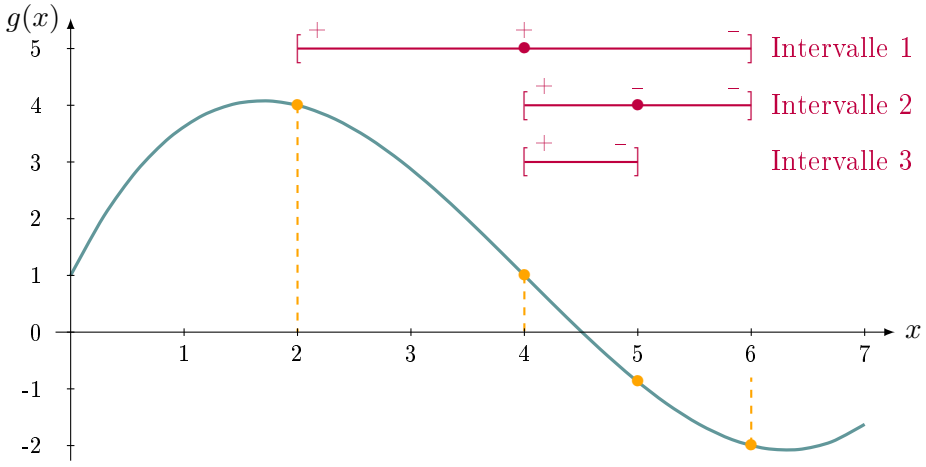


FIGURE 16 – Recherche du zéro par dichotomie

Le principe de la méthode est simple (voir FIGURE 16 : l'image par f du milieu c de l'intervalle est calculée, puis le signe est comparé aux signes de $f(a)$ et $f(b)$ pour déterminer si le zéro est sur l'intervalle $[a, c]$ ou $[c, b]$. Seul l'intervalle contenant la racine est conservé et le processus est répété jusqu'à obtenir une précision suffisante sur la valeur approchée de la solution.

```python
def dichotomie(f,a,b,epsilon):
    """
    Paramètres :
    - f : fonction
    - a,b : bornes de l'intervalle
    - epsilon : précision recherchée
    """
    while (abs(b-a)) > epsilon:
        c = (a+b)/2
        if f(a)*f(c) <= 0:
            b = c   # [a,c] sera le prochain intervalle
        else :
            a = c   # [c,b] sera le prochain intervalle
    return (a+b)/2
```

4.3 Méthode de Newton

La méthode de Newton s'appuie sur la dérivée de la fonction f pour s'approcher progressivement de la solution. Elle convient pour les fonctions f continues et dérivables. À partir d'une estimation x_i de la solution, la courbe est assimilée à sa tangente en x_i pour déterminer une estimation de la solution x_{i+1} et ainsi de suite (voir FIGURE 17).

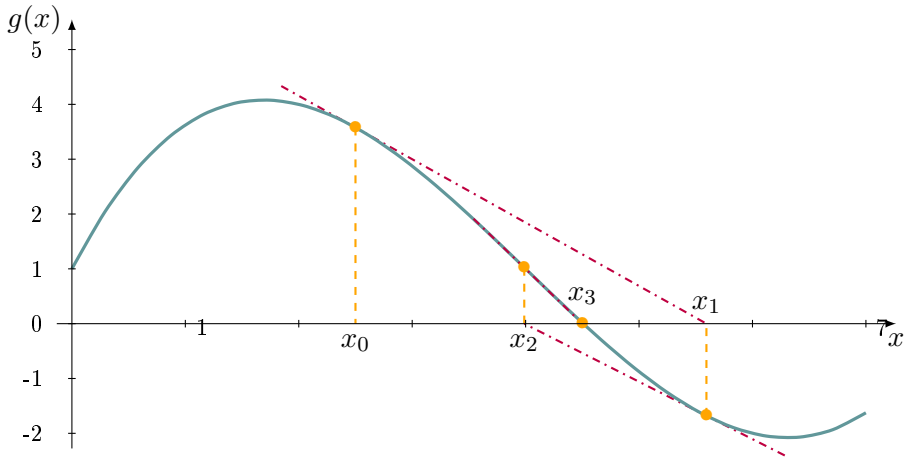


FIGURE 17 – Recherche du zéro par la méthode de Newton

L'expression de la tangente en x_0 s'écrit : $y = f'(x)(x - x_i) + f(x_i)$. Cette droite coupe l'axe des abscisses en $x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$, ce qui donne une nouvelle estimation de la solution.

```python
def newton(f,df,x0,epsilon):
    """
    Paramètres :
    - f : fonction
    - df : dérivée de f
    - x0 : solution initiale
    - epsilon : précision recherchée
    """
    xi = x0
    xi_plus_1 = x0-f(x0)/df(x0)
    while abs(xi_plus_1-xi) > epsilon:
        xi = xi_plus_1
        xi_plus_1 = xi-f(xi)/df(xi)  # Calcul de la nouvelle estimation de la solution
    return xi_plus_1
```

Cette méthode converge plus rapidement que la méthode de la dichotomie, mais elle nécessite l'expression de la dérivée de f, qui n'est pas toujours disponible. Dans ce cas, on pourra utiliser la méthode de la sécante, qui est très similaire à la méthode de Newton, mais qui calcule la pente de la tangente par un calcul numérique.

4.4 Réflexion sur la convergence de ces 2 algorithmes

Ces 2 algorithmes sont toujours opérants sur des fonctions continues et monotones. Cependant, il existe des contre-exemples. Par exemple, la dichotomie fonctionne sur les fonctions non monotones, à condition qu'elle ne passe qu'une seule fois par zéro. La méthode de Newton nécessite des critères plus exigeants :

- la dérivée en un point ne peut être nulle (sinon, il y aura une division par zéro lors du calcul de la nouvelle solution approchée). Ce cas de figure est néanmoins plutôt rare du fait du calcul numérique de la dérivée, qui donne rarement 0 ;
- si la valeur de départ est trop éloignée du vrai zéro, la méthode de Newton peut entrer en boucle infinie sans produire d'approximation améliorée ;
- si la pente de la courbe est très forte, du fait de la présence d'un critère d'arrêt sur deux valeurs successives de la solution, il y aura arrêt sur une solution éloignée de la vraie solution.

La méthode de la dichotomie semble donc plus robuste, mais plus lente à converger. On peut alors choisir une solution hybride en commençant la recherche par une méthode de la dichotomie et en affinant le résultat par la méthode de Newton.

A titre indicatif, la FIGURE 18 ci-dessous montre l'évolution de l'erreur en fonction du nombre d'itérations (en échelle log). On constate l'efficacité supérieure de la méthode de Newton. Les courbes ne peuvent descendre sous 10^{-16} du fait de la précision du codage des flottants dans les calculs.

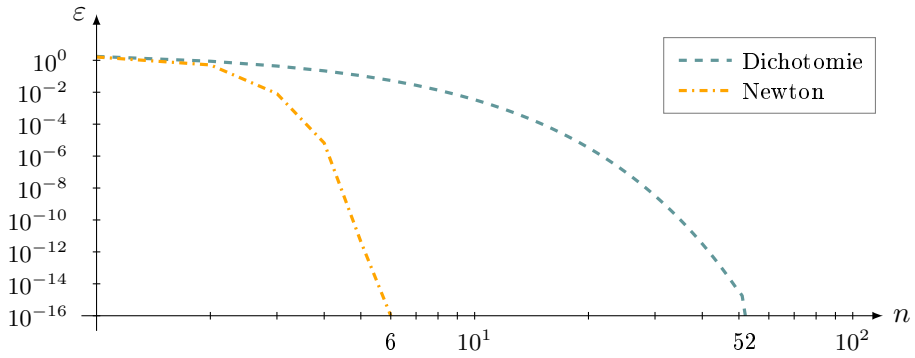


FIGURE 18 – Comparaison de l'évolution de l'erreur en fonction du nombre d'itérations

4.5 Bibliothèques de calcul numérique

Les 2 codes suivants permettent de mettre en œuvre les méthodes de la dichotomie et de Newton en utilisant la bibliothèque `scipy`.

```
"""
Méthode de la dichotomie
"""
from scipy.optimize import bisect
a,b = 1,6   # Intervalle de recherche
resultat = bisect(f,a,b)
```

```
"""
Méthode de Newton
"""
from scipy.optimize import newton
x0 = 3      # Point de départ
resultat = newton(f,x0,fprime=None)
```

5 Résolution de systèmes linéaires

5.1 Introduction

Beaucoup de problèmes de simulation numérique conduisent *in fine* à la résolution d'un système linéaire. Un exemple typique est celui d'une équation aux dérivées partielles (décrivant un système physique, par exemple), dont la transposition numérique peut se faire en remplaçant les dérivées par des différences finies (voir §3.4), ce qui conduit alors généralement à un système d'équations couplées.

Si l'équation de départ est linéaire, le système obtenu l'est également et, si le nombre d'inconnues n'est pas trop grand (au maximum quelques centaines de variables), ce système linéaire peut être résolu numériquement de manière efficace par la méthode du pivot de Gauss.

5.2 Méthode du pivot de Gauss avec pivot partiel

Considérons un système linéaire $Ax = b$, où A est une matrice carrée d'ordre n et b un vecteur colonne de taille n. Si A est une matrice inversible (ce que nous supposerons dans toute la suite), il s'agit d'un système de Cramer qui admet donc une unique solution x donnée par $x = A^{-1}b$. La résolution numérique d'un tel système ne se fait quasiment jamais en calculant d'abord l'inverse de A (problème qui n'est pas plus simple que la résolution du système $Ax = b$).

La méthode de Gauss repose sur deux remarques simples :

1. si la matrice A est triangulaire supérieure, alors la résolution du système $Ax = b$ est très facile ;
2. On ne change pas la solution d'un système linéaire en effectuant les mêmes opérations élémentaires sur les lignes de A et de b. Par opération élémentaire, nous entendons ici la permutation de deux lignes d'indices i et j (notées L_i et L_j), ou le remplacement de la ligne L_i par la ligne $L_i + \alpha L_j$ (opération terme à terme que l'on notera $L_i \longleftarrow L_i + \alpha L_j$, α étant un réel quelconque).

La méthode de Gauss consiste donc à transformer le système initial $Ax = b$ en un système triangulaire, en effectuant des opérations élémentaires adéquates sur les lignes de A et b , puis à résoudre simplement le système obtenu.

Transformation du système (élimination de Gauss) Supposons qu'à l'étape j la matrice A (transformée par les étapes antérieures) ait la structure suivante :

$$A = \begin{pmatrix} a_{11} & \cdots & \cdots & \cdots & \cdots & a_{1n} \\ 0 & a_{22} & \cdots & \cdots & \cdots & a_{2n} \\ \vdots & \ddots & \ddots & & & \vdots \\ \vdots & & 0 & a_{jj} & \cdots & a_{jn} \\ \vdots & & \vdots & \vdots & & \vdots \\ 0 & \cdots & 0 & a_{nj} & \cdots & a_{nn} \end{pmatrix}$$

Une façon simple d'obtenir une matrice de même structure pour l'étape $j+1$ consiste à effectuer les opérations élémentaires :

$$L_i \longleftarrow L_i - \frac{a_{ij}}{a_{jj}} L_j, \quad j+1 \ll i \ll n$$

de façon à faire apparaître naturellement des zéros sur la colonne j pour les indices $i \in \{j+1, \ldots, n\}$.

Cette méthode n'est pas toujours satisfaisante, car le pivot (le coefficient a_{jj} qui apparaît au dénominateur de l'équation précédente) peut être très petit (ce qui rend alors les calculs très sensibles aux erreurs d'arrondis), voire même nul (auquel cas la division ne peut bien sûr pas être effectuée). Dans la méthode de Gauss avec pivot partiel, on cherche donc dans un premier temps l'indice $k \in \{j, \ldots, n\}$ pour lequel $|a_{kj}|$ est maximal, puis (si $k \neq j$) les lignes j et k sont permutées avant d'effectuer les opérations élémentaires énoncées plus haut. La stabilité numérique de l'algorithme est ainsi assurée. L'inversibilité de la matrice A garantit l'inégalité $\max_{k\in\{j,\ldots,n\}}|a_{kj}| > 0$.

Remarque *Méthode du pivot total*

Signalons au passage l'existence de la méthode du pivot total, qui repose sur la sélection du plus grand coefficient (en valeur absolue) parmi les a_{kl}, $j \leq k \leq n$, $j \leq l \leq n$ et nécessite en outre une permutation de colonnes ; cette méthode est un peu plus robuste que celle du pivot partiel, mais le gain est généralement considéré comme faible par rapport à la complexité supplémentaire qu'elle introduit dans l'algorithme.

La transformation de la matrice A (et du vecteur b, soumis aux mêmes opérations élémentaires) peut s'implémenter sous la forme suivante :

```python
import numpy as np
def transforme_gauss(A0,b0):
    """
    Paramètres :
    - A0 : matrice carrée (array numpy)
    - b0 : vecteur (array numpy)
    """
    A,b = A0.copy(),b0.copy()
    n = np.size(A,0)
    # Boucle principale
    for j in range(n-1):
        # Recherche du pivot partiel (position k)
        # avec correction pour repasser dans le référentiel de A
        k = j + argmax (abs(A[j:n,j]))
        # échange des lignes j et k de b et de A
        b[[k,j]] = b[[j,k]]
        A[[k,j],:] = A[[j,k],:]
        # Transformation élémentaire des lignes j+1 à n
        for i in range (j+1,n):
            alpha = A[i,j]/A[j,j]
            b[i] -= alpha*b[j]
            A[i ,:] -= alpha*A[j,:]
    return A,b
```

Résolution du système transformé Une fois le système linéaire initial transformé en un système linéaire triangulaire (A étant alors une matrice triangulaire supérieure), la résolution devient immédiate puisque x_n est obtenu directement par :

$$x_n = \frac{b_n}{a_{nn}}$$

et les valeurs de x_{n-1}, x_{n-2}, …, x_1 successivement au moyen de la récurrence (descendante) :

$$x_1 = \frac{1}{a_{ii}}\left(b_i - \sum_{k=i+1}^{n} a_{ik}x_k\right)$$

Cette récurrence s'implémente simplement sous la forme suivante :

```
def solution_triangulaire(A,b):
    """
    Paramètres :
    - A : matrice carrée (array numpy)
    - b : vecteur (array numpy)
    """
    n = np.size(A,0)
    x = np.zeros(n)
    for i in range(n-1,-1,-1):
        s = b[i]
        for k in range(i+1,n):
            s -= A[i,k]*x[k]
        x[i] = s/A[i,i]
    return x
```

Résolution du système linéaire initial La combinaison des deux étapes ci-dessus permet donc de résoudre un système linéaire par la méthode de Gauss avec pivot partiel :

```
def gauss_pivot_partiel(A,b):
    """
    Paramètres :
    - A : matrice carrée (array numpy)
    - b : vecteur (array numpy)
    """
    AA,bb = transforme_gauss(A,b)
    x = solution_triangulaire(AA ,bb)
return x
```

5.3 Bibliothèques de calcul numérique

Pour résoudre un système linéaire avec Python, on peut utiliser la fonction `linalg.solve` de `numpy`.

Il est ainsi possible de vérifier le fonctionnement de la fonction `gauss_pivot_partiel` en comparant sur l'exemple simple le résultat obtenu à celui donné par la commande `linalg.solve(A,b)` de `numpy` (avec `A` un array correspondant à une matrice et `b` un array vecteur), qui renvoie également la solution du système $Ax = b$.

```
>>> A = np.array([[1,2,3],[1,2,1],[3,1,1]],float)
>>> b = np.array([4,5,6],float)
>>> np.linalg.solve(A,b)
array([1.5,2.,-0.5])
>>> gauss_pivot_partiel(A,b)
array([1.5,2.,-0.5])
```

6 Traitement de fichiers de mesures

6.1 Introduction

Très souvent, lors d'acquisition de mesures, on peut récupérer les données sous la forme d'un fichier `csv` (données séparées par les virgules). De plus, comme nous avons pu le voir dans le cours sur les capteurs ou au §2.3, les signaux recueillis sont très souvent bruités, et empêchent un traitement numérique efficace.

L'objectif de ce chapitre est donc double : reprendre les méthodes permettant de lire les données dans un fichier, et présenter quelques méthodes de filtrage numérique.

6.2 Lecture de fichiers et mise en forme des données

L'objectif n'est pas ici de faire un cours sur la lecture/écriture de fichiers en Python. Néanmoins, nous allons proposer un code simple et transposable qui permet de gérer le traitement d'un fichier de mesures.

Considérons le fichier de mesures `capsuleuse.csv` ci-dessous :

```
====================================
Acquisition capsuleuse
====================================
t;omega_e;omega_s;C_e;C_s
0,0;3,083881578947369;52,68297697368422;0,0;0,0
0,01;2,569901315789474;46,51521381578948;0,0;0,0
0,02;2,8268914473684212;46,258223684210535;0,0;0,0
0,03;2,3129111842105265;47,28618421052632;0,0;0,0
0,04;2,055921052631579;46,51521381578948;0,0;0,0
```

Le code ci-dessous permet alors d'extraire de ce fichier les données sous forme de trois listes de même dimensions : une pour le `temps`, et une pour `psipoint` (ω_e) et `thetapoint` (ω_s).

```
mon_fichier=open("capsuleuse.csv","r")  # Ouverture du fichier
for i in range(4):                      # On élimine les 4 lignes d'en-tête
    poubelle=mon_fichier.readline()
temps=[]                                # initialisation des listes
psiPoint=[]
thetaPoint=[]
# Pour chaque ligne du fichier
for ligne in mon_fichier:
    # On élimine le retour chariot et les séparateurs de colonne (point virgule)
    # On remplace aussi les virgules par des points (séparateur décimal en Python)
    data = ligne.replace(",",".").rstrip("\n\r").split(";")
    # On récupère la valeur des colonnes temps, omega_e (thetaPoint) et omega_s (psiPoint)
    temps.append(float(data[0]))
    psiPoint.append(float(data[1]))
    thetaPoint.append(float(data[2]))
mon_fichier.close()                     # Fermeture du fichier
```

6.3 Filtrage

Les capteurs renvoient un signal qui peut être bruité, du fait de leur mode de fonctionnement, de la quantification, ou d'éléments perturbateurs pendant la transmission du signal. Que ce soit dans un dans le cadre d'un asservissement numérique (en temps réel) ou dans l'analyse de mesures pré-enregistrées (post-traitement), le bruit dans le signal fourni par le capteur est toujours néfaste, car il peut dégrader les performances d'un système ou amener à des erreurs parfois spectaculaires de résolution numérique.

Très souvent, le signal brut issu du capteur est d'abord filtré de manière analogique, avec une fréquence de coupure liée à la fréquence d'échantillonnage du convertisseur analogique-numérique (on parle alors de filtre anti-repliement) puis par un filtre numérique sur le signal échantillonné dans le microcontrôleur.

6.3.1 Filtre à moyenne glissante

Le filtre à moyenne glissante consiste à calculer pour chaque y_i la moyenne des n termes centrés sur y_i. On pourra donc utiliser la méthode suivante :

$$\overline{y_i} = \frac{1}{n} \sum_{k=-\frac{n-1}{2}}^{\frac{n-1}{2}} y_{i+k}$$

En python, on peut coder cette fonction de la manière suivante (si on doit utiliser ce filtre pour un traitement de données en temps réel, la moyenne sera calculée sur les n valeurs précédant la valeur acquise en dernier) :

```
def moyenne_glissante(x,y,n):
    """
    Paramètres :
    - x : abscisses
    - y : ordonnees
    - n : nombre de points
    """
    if n%2 == 0:        # On souhaite un n impair
        n -= 1
    a = int((n-1)/2)
    y_filtres = []
    for i in range(a,len(y)-a):
        y_filtres.append(sum(y[i-a:i+a+1])/n) # Calcul de la moyenne
    return x[a:-a],y_filtres
```

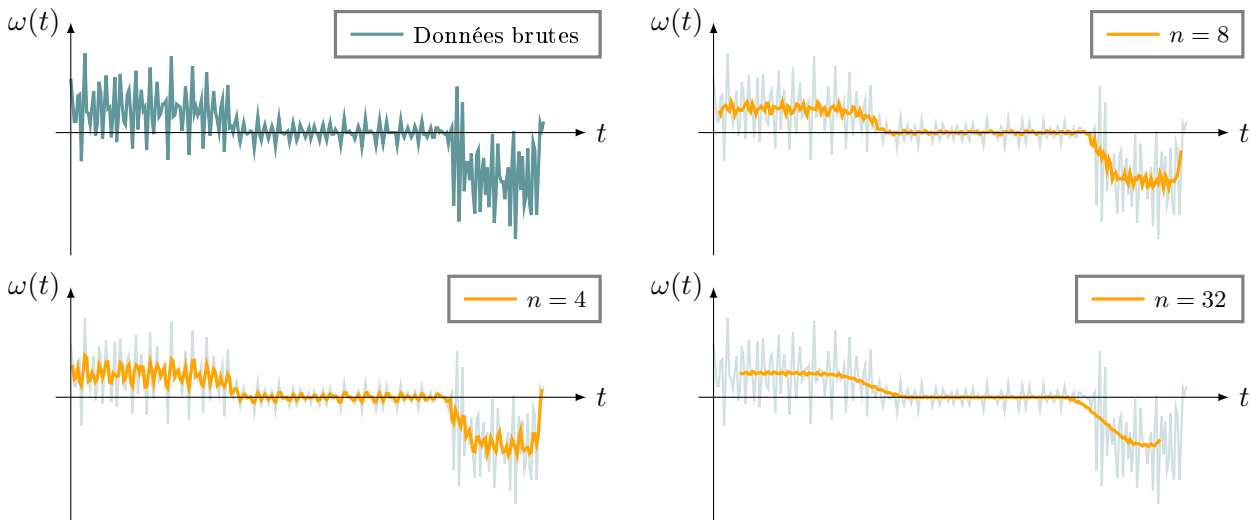


FIGURE 19 – Influence du nombre de points utilisés pour la moyenne glissante

6.3.2 Filtre passe-bas du premier ordre

Ce filtre a une fonction de transfert $F(p) = \frac{Y_f(p)}{Y(p)}$ avec Y le signal d'entrée et Y_f le signal filtré. Pour obtenir l'équation de récurrence, il suffit de discrétiser l'équation différentielle correspondant à ce filtre en partant de l'analyse de la relation entrée-sortie écrite sous la forme $Y_f(p) = F(p)Y(p)$.

Pour un filtre du premier ordre, la fonction de transfert est $F(p) = \frac{1}{1+\tau p}$, d'où :

$$Y_f(p)(1+\tau p) = Y(p) \quad \xrightarrow{\mathcal{L}^{-1}} \quad \tau \dot{y}_f(t) + y_f(t) = y(t)$$

L'équation différentielle est discrétisée en évaluant les différents termes pour $t_i = iT_e$ et la dérivée est approchée par $\dot{y}_f(t_i) = \dot{y}_{f_i} = \frac{y_{f_i} - y_{f_{i-1}}}{T_e}$ (voir §2.2). On obtient alors :

$$\tau \frac{y_{f_i} - y_{f_{i-1}}}{T_e} + y_{f_i} = y_i \quad \Rightarrow \quad \boxed{y_{f_i} = \frac{T_e}{\tau + T_e} y_i + \frac{\tau}{\tau + T_e} y_{f_{i-1}}}$$

Pour respecter le théorème de Shannon, il faut prendre $f_e \geq \frac{2}{\tau}$ soit $T_e \leq \frac{\tau}{2}$, on obtient alors :

$$y_{f_i} = 0,66 y_i + 0,33 y_{f_{i-1}}$$

L'implémentation de ce type de filtre est alors très simple, et peut s'écrire comme suit en Python :

```
1  def passe_bas_1er_ordre(x,y,tau):
2      """
3      Paramètres :
4      - x : abscisses
5      - y : ordonnees
6      - tau : constante de temps (idéalement, Te/2)
7      """
8      y_filtres = [y[0]]
9      for i in range(1,len(y)):
10         Te = x[i]-x[i-1]
11         y_filtres.append(Te/(tau+Te)*y[i]+tau/(tau+Te)*y_filtres[i-1])
12     return x[1:],y_filtres[1:]
```

La courbe suivante montre l'influence du choix de la constante de temps du filtre sur la réponse filtrée obtenue : plus la constante de temps est grande plus la courbe filtrée s'écarte de la courbe d'origine non filtrée essentiellement en régime transitoire.

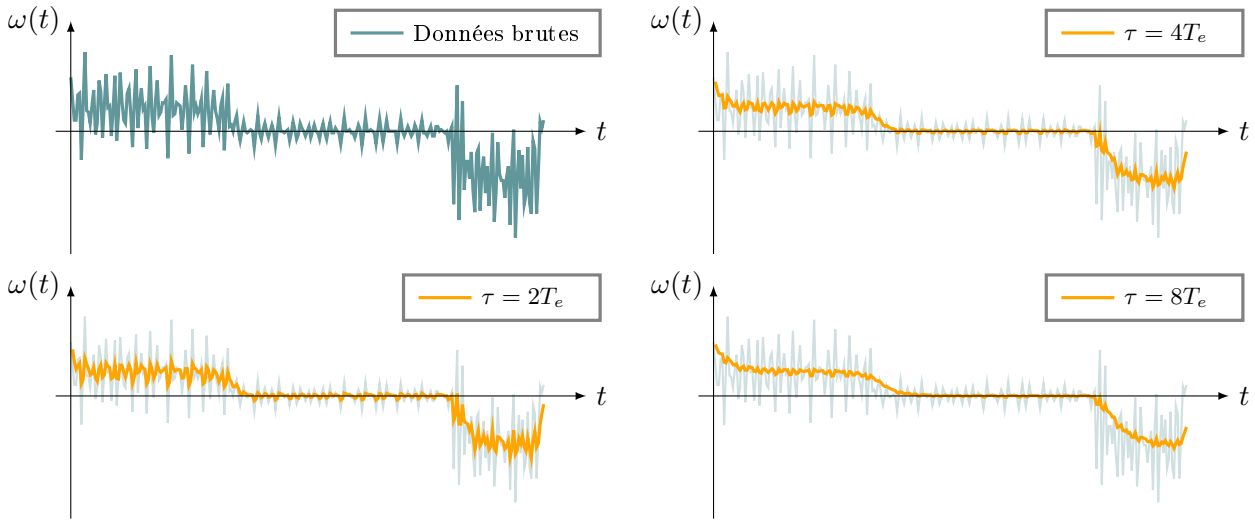


FIGURE 20 – Influence de la constante de temps d'un filtre passe-bas numérique

6.3.3 Filtre passe-bas du deuxième ordre

Le filtre passe-bas du deuxième ordre est plus sélectif que le filtre passe-bas du premier ordre. La fonction de transfert du deuxième ordre est $F(p) = \dfrac{1}{1+\frac{2\xi}{\omega_0}p+\frac{1}{{\omega_0}^2}p^2}$, d'où :

$$Y_f(p)\left(1+\frac{2\xi}{\omega_0}p+\frac{1}{{\omega_0}^2}p^2\right)=Y(p) \quad \xrightarrow{\mathcal{L}^{-1}} \quad \frac{1}{{\omega_0}^2}\ddot{y}_f(t)+\frac{2\xi}{\omega_0}\dot{y}_f(t)+y_f(t)=y(t)$$

On approche au temps iT_e, la dérivée première de la même façon que pour le premier ordre ($\dot{y}_f(t_i) = \dot{y}_{f_i} = \frac{y_{f_i}-y_{f_{i-1}}}{T_e}$) et la dérivée seconde par $\ddot{y}_f(t_i) = \ddot{y}_{f_i} = \frac{\dot{y}_{f_i}-\dot{y}_{f_{i-1}}}{T_e} = \frac{y_{f_i}-2y_{f_{i-1}}+y_{f_{i-2}}}{{T_e}^2}$(voir §2.2). L'équation discrétisée est alors :

$$\frac{1}{{\omega_0}^2}\frac{y_{f_i}-2y_{f_{i-1}}+y_{f_{i-2}}}{{T_e}^2}+\frac{2\xi}{\omega_0}\frac{y_{f_i}-y_{f_{i-1}}}{T_e}+y_{f_i}=y_i$$

Ce qui donne l'équation de récurrence suivante :

$$\boxed{y_{f_i}=a_0y_i+b_0y_{f_{i-1}}+b_1y_{f_{i-2}}}$$

avec :

$$a_0=\frac{{\omega_0}^2{T_e}^2}{1+2\xi\omega_0T_e+{\omega_0}^2{T_e}^2} \qquad b_0=2\frac{1+\xi\omega_0T_e}{1+2\xi\omega_0T_e+{\omega_0}^2{T_e}^2} \qquad b_1=-\frac{1}{1+2\xi\omega_0T_e+{\omega_0}^2{T_e}^2}$$

Il faut alors choisir la période T_e telle que $\frac{1}{T_e} \geq 2\omega_0$ (théorème de Shannon) : on choisit généralement un facteur d'amortissement $\xi = 1$ pour assurer un affaiblissement optimal de $-6\,\text{dB}$ pour la pulsation ω_0.

En prenant $\omega_0 = \frac{1}{2T_e}$ (valeur limite), on obtient :

$$y_{f_i}=0,444y_i+0,667y_{f_{i-1}}-0,111y_{f_{i-2}}$$

L'implémentation de ce filtre en Python est alors relativement aisée :

```
def passe_bas_2eme_ordre(x,y,xi,omega0):
    """
    Paramètres :
    - x : abscisses
    - y : ordonnees
    - xi : coefficient d'amortissement (idéalement, 1)
    - omega0 : pulsation propre (idéalement, 1/(2Te))
    """
    y_filtres = [y[0],y[1]]
    for i in range(2,len(y)):
        Te = (x[i]-x[i-2])/2
        a0 = omega0**2*Te**2/(1+(omega0*Te)**2+2*xi*omega0*Te)
        b0 = 2*(1+xi*omega0*Te)/(1+(omega0*Te)**2+2*xi*omega0*Te)
        b1 = -1/(1+(omega0*Te)**2+2*xi*omega0*Te)
        y_filtres.append(a0*y[i]+b0*y_filtres[i-1]+b1*y_filtres[i-2])
    return x[2:],y_filtres[2:]
```

La courbe suivante montre l'influence du choix de la fréquence de coupure du filtre : on constante qu'on observe peu de différence quant à la valeur de ω_0, même si les résultats semblent faussés quand ω_0 est trop petit.

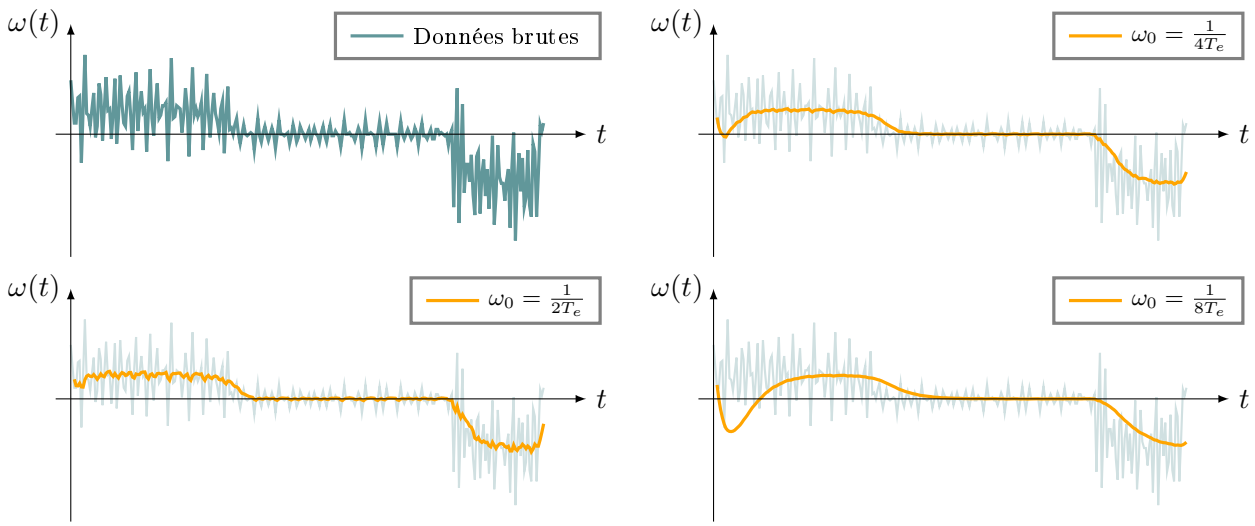


FIGURE 21 – Influence de la pulsation de coupure d'un filtre passe-bas numérique

Références

[1] A. CAIGNOT, V. CRESPEL et D. VIOLEAU : *Sciences Industrielles de l'Ingénieur - Tout en un - MP/MP*, PSI/PSI*, PT/PT**. Vuibert, 2022.

[2] A. CAIGNOT, M. DÉRUMAUX, J. LABASQUE et L. MOISAN : *Informatique pour Tous - CPGE scientifiques - 1ere et 2eme année*. Vuibert, 2015.

[3] E. BILLETTE : Cours d'informatique de tronc commun, 2021. PTSI - Lycée Jean Zay - Thiers.

[4] A. CAIGNOT et M. DÉRUMEAUX : Cours d'ingénierie numérique, 2014. UPSTI.

[5] D. DEFAUCHY : Cours d'ingénierie numérique, 2022. UPSTI.