Notes de cours

Analyse Numérique

Fabio Nobile

2023-2024

Dernière mise à jour : 13 février 2024

EPFL

Table des matières

Chapitre 1

Équations non linéaires

Dans ce chapitre on s'intéresse à calculer numériquement les zéros d'une fonction continue $f(x) : [a, b] \to \mathbb{R}$, c'est-à-dire les valeurs $\alpha \in [a, b]$ telles que

$$f(\alpha) = 0. \tag{1.1}$$

Pour cela, on étudiera plusieurs méthodes numériques qui permettent de calculer une solution approchée de (1.1)

1.1 Exemple : circuit électrique

On considère le circuit électrique montré en Figure 1.1 (gauche) qui

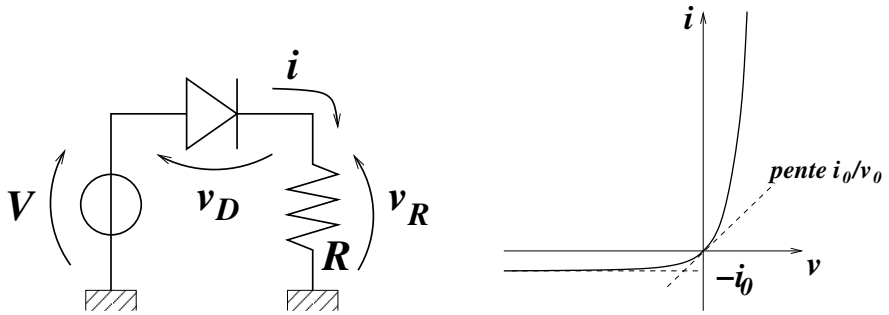


FIGURE 1.1 – Gauche : Circuit électrique contenant un générateur de tension, une résistance R et une diode normale. Droite : courbe caractéristique d'une diode normale

contient :

- un générateur de tension, qui génère la tension constante V
- une résistance électrique R qui donne une relation linéaire $v_R = Ri$ entre courant et tension

— une diode normale dont la courbe caractéristique entre courant et tension est

$$i = i_0 \left(e^{v_D/v_0} - 1 \right).$$

La relation est montrée en Figure 1.1 (droite).

On souhaite calculer la tension sur la diode. On peut écrire les relations suivantes :

$$\begin{cases} v_R + v_D = V \\ v_R = Ri \\ i = i_0 \left(e^{v_D/v_0} - 1 \right) \end{cases} \qquad \Longrightarrow \qquad Ri_0 \left(e^{v_D/v_0} - 1 \right) + v_D = V.$$

La tension v_D sur la diode est donc le zéro de l'*équation non linéaire*

$$f(x) = 0, \qquad \text{où} \quad f(x) = Ri_0 \left(e^{x/v_0} - 1 \right) + x - V. \tag{1.2}$$

1.2 Méthode de dichotomie (ou bissection)

Cette méthode est basée sur l'observation suivante :

Remarque 1.1. *Soit* $f : [a, b] \to \mathbb{R}$ *une fonction* continue *telle que* $f(a)f(b) < 0$. *Alors, nécessairement,* f *a au moins un zéro dans* $[a, b]$.

Prenons maintenant le point milieu de l'intervalle $[a, b]$: $x_m = \frac{a+b}{2}$. On a les trois possibilités suivantes :

— soit $f(x_m)f(a) < 0 \implies$ alors il y a un zéro dans l'intervalle $[a, x_m]$;
— soit $f(x_m)f(b) < 0 \implies$ alors il y a un zéro dans l'intervalle $[x_m, b]$;
— si jamais $f(x_m) = 0 \quad \implies$ on a trouvé un zéro.

On peut donc itérer la procédure à partir de l'un des deux sous-intervalles qui contient un zéro.

Algorithme 1.1 : Méthode de bissection (sans critère d'arrêt)

On pose $a^{(0)} = a$, $b^{(0)} = b$ et $x^{(0)} = \frac{a^{(0)}+b^{(0)}}{2}$;
pour $k = 0, 1, \ldots$ **faire**
  si $f(x^{(k)})f(a^{(k)}) < 0$ **alors** `// nouvel intervalle` $[a^{(k)}, x^{(k)}]$
    $a^{(k+1)} = a^{(k)}$, $b^{(k+1)} = x^{(k)}$;
  sinon `// nouvel intervalle` $[x^{(k)}, b^{(k)}]$
    $a^{(k+1)} = x^{(k)}$, $b^{(k+1)} = b^{(k)}$;
  fin
  $x^{(k+1)} = \frac{a^{(k+1)}+b^{(k+1)}}{2}$;
fin

L'algorithme présenté ci-dessus est encore incomplet car il faut ajouter un critère d'arrêt des itérations, que nous allons discuter maintenant.

Contrôle de l'erreur

À l'itération k, le zéro est contenu dans l'intervalle $I^{(k)} = [a^{(k)}, b^{(k)}]$ dont la longueur est $|I^{(k)}| = \frac{(b-a)}{2^k}$. Le zéro est estimé par le point milieu $x^{(k)} = \frac{a^{(k)}+b^{(k)}}{2}$. Si α dénote le vrai zéro, l'erreur commise à l'étape k de l'algorithme est bornée par

$$|\alpha - x^{(k)}| \leq \frac{1}{2}|I^{(k)}| = \left(\frac{1}{2}\right)^{k+1}(b-a). \tag{1.3}$$

Si on veut, maintenant, trouver une approximation du zéro avec une tolérance fixée *tol*, on peut donc arrêter les itérations dans l'algorithme 1.1 lorsque $|I^{(k)}| < 2\,tol$. Il faudra effectuer un nombre d'itérations k_{min} tel que

$$\left(\frac{1}{2}\right)^{k_{min}+1}(b-a) \leq tol \qquad \Longrightarrow \qquad k_{min} > \log_2\left(\frac{b-a}{tol}\right) - 1.$$

On remarque que la borne (1.3) est *garantie*, c'est-à-dire que si on effectue k_{min} itérations on est garanti d'avoir une erreur plus petite que *tol*. Voici donc l'algorithme complet avec critère d'arrêt :

Algorithme 1.2 : Méthode de bissection

Données : $f(x)$, $[a,b]$, $x^{(0)}$, tol
Résultat : α, niter
$a^{(0)} = a$, $b^{(0)} = b$, $x^{(0)} = \frac{a^{(0)}+b^{(0)}}{2}$;
$k_{min} = \lceil \log_2\left(\frac{b-a}{tol}\right) - 1 \rceil$;
pour $k = 0, 1, \ldots, k_{min} - 1$ **faire**
  si $f(x^{(k)})f(a^{(k)}) < 0$ **alors** `// nouvel intervalle` $[a^{(k)}, x^{(k)}]$
    $a^{(k+1)} = a^{(k)}$, $b^{(k+1)} = x^{(k)}$;
  sinon `// nouvel intervalle` $[x^{(k)}, b^{(k)}]$
    $a^{(k+1)} = x^{(k)}$, $b^{(k+1)} = b^{(k)}$;
  fin
  $x^{(k+1)} = \frac{a^{(k+1)}+b^{(k+1)}}{2}$;
fin
$\alpha = x^{(k_{min})}$, niter=k_{min}.

Avantages et désavantages

+ Si on trouve un intervalle où la fonction change de signe, l'algorithme converge sûrement vers un zéro.
+ De plus, on a un contrôle précis sur l'erreur.
- Si la fonction ne change pas de signe autour d'un zéro, on ne peut pas utiliser cet algorithme.
- La convergence de l'algorithme est assez lente ; l'erreur est divisée seulement par deux à chaque itération.

Exemple 1.1 (Circuit éléctrique)**.** *On revient à l'exemple du circuit électrique. On peut visualiser en* **Python** *la fonction $f(x)$. On se donne les valeurs suivantes : $i_0 = 1$, $v_0 = 0.1$, $R = 1$; $V = 1$.*

```
import numpy as np
import matplotlib.pyplot as plt
i0=1; v0=0.1; R=1; V=1;
f=lambda x: R*i0*(np.exp(x/v0)-1)+x-V;
x=np.linspace(-0.2,0.2,40)
plt.plot(x,f(x),color='b',LineWidth=2)
plt.plot(x,0*x,'--k',LineWidth=2)
plt.grid(True)
```

La figure 1.2 montre la fonction $f(x)$. Il y a clairement un zéro dans l'intervalle $[-0.2, 0.2]$.

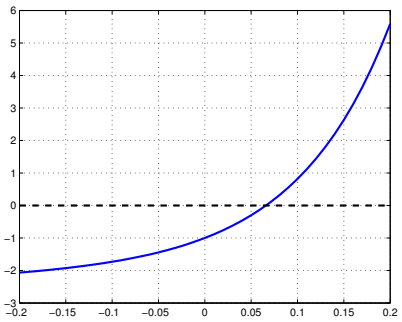


FIGURE 1.2 – Graphique de la fonction (1.2)

On applique la méthode de bissection à l'aide de la fonction `bisection.m` *disponible sur la page web Moolde (voir* `help bisection` *pour un détail sur les paramètres d'entrée et sortie de la fonction).*

```
import numpy as np
from bisection import bisection
i0=1; v0=0.1; R=1; V=1;
f = lambda x:  R*i0*(np.exp(x/v0)-1)+x-V;
zero,res,niter,inc,err=bisection(f,-0.2,0.2,1e-8,10000)
print('zero '+str(zero))
print('res '+str(res))
print('iterations '+str(niter))

# OUTPUT
# zero 0.06596105694770812
# res 7.085173225895858e-08
# iterations 25
```

1.3 Méthode de point fixe

1.3.1 Introduction sur l'exemple du circuit électrique

Considérons encore l'exemple du circuit électrique. Pour trouver une approximation du zéro, on aurait pu procéder de la façon suivante :

Méthode 1

- Supposons qu'on ait une idée initiale de la tension sur la diode, que l'on appellera $v_D^{(0)}$. Par exemple, si la diode est "ouverte" on s'attend à ce que la tension sur la diode soit presque zéro et on peut prendre $v_D^{(0)} = 0$.
- Étant donnée la tension $v_D^{(0)}$ sur la diode, on peut calculer la tension sur la résistance $v_R^{(0)} = V - v_D^{(0)}$ et donc le courant $i^{(0)} = v_R^{(0)}/R = (V - v_D^{(0)})/R$.
- Puisque le courant dans la résistance et dans la diode est le même, on peut calculer une nouvelle estimation de la tension sur la diode en inversant la courbe caractéristique de la diode :

$$v_D^{(1)} = v_0 \log\left(\frac{i^{(0)}}{i_0} + 1\right).$$

On espère que la nouvelle estimation $v_D^{(1)}$ soit meilleure que la précédente.

À ce moment là, on peut itérer la procédure et calculer une deuxième estimation $v_D^{(2)}$ et ainsi de suite. On espère que la suite $v_D^{(0)}, v_D^{(1)}, v_D^{(2)}, \ldots$ converge vers la "vraie" tension de la diode.

La procédure qu'on vient de décrire peut être formalisée de la façon suivante : étant donnée une estimation $v_D^{(k)}$ à l'itération k on calcule la nouvelle estimation $v_D^{(k+1)}$ par

$$v_D^{(k+1)} = v_0 \log\left(\frac{V - v_D^{(k)}}{Ri_0} + 1\right). \tag{1.4}$$

On essaye cette procédure en `Python` :

```
import numpy as np
i0=1; v0=0.1; R=1; V=1;
vD=0
for i in range(10):
    vD = v0*np.log((V-vD)/(R*i0)+1)
    #formats to 15 decimal places
    formatted_string = "{:.15f}".format(vD)
    vD = float(formatted_string)
    print('vD = '+str(vD))
```

```
#OUTPUT
# vD = 0.069314718055995
# vD = 0.065787500825971
# vD = 0.065970026647302
# vD = 0.065960589502512
# vD = 0.065961077453676
# vD = 0.065961052224034
# vD = 0.065961053528539
# vD = 0.065961053461089
# vD = 0.065961053464577
# vD = 0.065961053464397
```

On voit bien que la suite converge vers la même valeur qu'on a déjà trouvée par la méthode de bissection. De plus, après 10 itérations, déjà 12 chiffres sont "stabilisés" et on s'attend à ce que l'erreur soit plus petite que 10^{-12}. Cette méthode a l'air de converger plus rapidement que la méthode de bissection.

En fait, on aurait aussi pu imaginer une autre procédure :

Méthode 2

- Étant donnée une idée initiale de la tension $v_D^{(0)}$ sur la diode, on peut calculer le courant $i^{(0)}$ qui traverse la diode : $i^{(0)} = i_0 \left(e^{v_D^{(0)}/v_0} - 1 \right)$.
- On calcule ensuite la tension sur la résistance $v_R^{(0)} = Ri^{(0)}$.
- Finalement, on peut calculer une nouvelle estimation de la tension sur la diode par $v_D^{(1)} = V - v_R^{(0)}$,

et on continue ainsi de suite. Cette procédure peut être formalisée de la façon suivante : étant donnée une estimation $v_D^{(k)}$ à l'itération k on calcule la nouvelle estimation $v_D^{(k+1)}$ par

$$v_D^{(k+1)} = V - Ri_0 \left(e^{v_D^{(k)}/v_0} - 1 \right). \tag{1.5}$$

On essaye aussi cette procédure en `Python` :

```
import numpy as np
i0=1; v0=0.1; R=1; V=1;
vD=0
for i in range(10):
    vD = V-R*i0*(np.exp(vD/v0)-1)
    #formats to 15 decimal places
    formatted_string = "{:.15f}".format(vD)
    vD = float(formatted_string)
    print('vD = '+str(vD))

# OUTPUT
# vD = 1.0
# vD = -22024.465794806718
```

```
# vD = 2.0
# vD = -485165193.4097903
# vD = 2.0
# vD = -485165193.4097903
# vD = 2.0
# vD = -485165193.4097903
# vD = 2.0
# vD = -485165193.4097903
```

Quoique cette procédure semble aussi raisonnable que la précédente, elle n'a pas du tout l'air de converger. On ne peut donc pas l'utiliser pour calculer la tension sur la diode.

On essaye maintenant de formaliser ce qu'on a fait. On avait à résoudre l'équation

$$f(x) = Ri_0\left(e^{x/v_0} - 1\right) + x - V = 0.$$

Dans la première procédure on a réécrit cette équation sous la forme équivalente

$$\begin{aligned} &\text{courant dans la diode :} && i_0\left(e^{x/v_0} - 1\right) = \frac{V - x}{R} \\ &\text{tension sur la diode :} && x = v_0 \log\left(\frac{V - x}{Ri_0} + 1\right), \end{aligned}$$

puis on a construit les itérations

$$x^{(k+1)} = v_0 \log\left(\frac{V - x^{(k)}}{Ri_0} + 1\right).$$

Dans la deuxième procédure, par contre, on a réécrit l'équation tout simplement sous la forme

$$x = V - Ri_0\left(e^{x/v_0} - 1\right)$$

et on a construit les itérations $x^{(k+1)} = V - Ri_0\left(e^{x^{(k)}/v_0} - 1\right)$.

Dans les deux cas, on a donc réécrit l'équation non linéaire $f(x) = 0$ sous la forme équivalente

$$x = \phi(x), \tag{1.6}$$

puis on a construit la méthode itérative

$$x^{(k+1)} = \phi(x^{(k)}), \quad k = 0, 1, \ldots \tag{1.7}$$

Une équation de la forme (1.6) est appelée *équation de point fixe* car la valeur α qui satisfait l'équation : $\alpha = \phi(\alpha)$ est telle que, si on part de α et on applique la fonction ϕ, on retrouve la valeur α elle même. La valeur α est appelée *point fixe* de la fonction ϕ.

La procédure itérative (1.7) est appelée *méthode de point fixe* ou *méthode des itérations de point fixe*.

1.3.2 Méthode de point fixe

Étant donnée une équation non linéaire $f(x) = 0$, la méthode de point fixe consiste d'abord à réécrire sous forme équivalente l'équation $f(x) = 0$ comme une équation de point fixe $x = \phi(x)$ de sorte que si α est le zéro de f

$$f(\alpha) = 0 \qquad \Longrightarrow \qquad \alpha = \phi(\alpha).$$

Algorithme 1.3 : Méthode de point fixe (sans critère d'arrêt)

On choisit un point de départ $x^{(0)}$ suffisamment proche du point fixe α ;
pour $k = 0, 1, \ldots$ **faire**
  $x^{(k+1)} = \phi(x^{(k)})$
fin

Interprétation graphique

On donne d'abord une interprétation graphique de la méthode. La solution de l'équation de point fixe $x = \phi(x)$ peut être vue comme la solution du système

$$\begin{cases} y = \phi(x) \\ y = x. \end{cases}$$

Graphiquement, les points fixes de ϕ sont donnés par l'intersection entre la fonction $y = \phi(x)$ et la droite $y = x$.

De même, dans les itérations de point fixe, on part d'une valeur $x^{(k)}$ et on calcule d'abord $y^{(k+1)} = \phi(x^{(k)})$ et ensuite $x^{(k+1)} = y^{(k+1)}$. Donc, la valeur $x^{(k+1)}$ en abscisse a la même distance de l'origine que la valeur $y^{(k+1)}$ en ordonnée. La Figure 1.3 donne une interprétation graphique de la méthode de point fixe ; on a considéré l'équation

$$f(x) = x + \log(x + 1) - 2 = 0$$

et les 4 équations de point fixe équivalentes

$$\begin{aligned} x &= \phi_1(x) = x - \frac{1}{2}(\log(x + 1) + x - 2) \\ x &= \phi_2(x) = 2 - \log(x + 1) \\ x &= \phi_3(x) = e^{(2-x)} - 1 \\ x &= \phi_4(x) = \frac{1}{2}x(\log(x + 1) + x). \end{aligned} \tag{1.8}$$

Dans les graphiques on a aussi visualisé la droite de pente -1 qui passe par le point fixe. On remarque d'après la figure que les méthodes de point fixe pour les fonctions ϕ_1 et ϕ_2 convergent tandis que les méthodes pour les fonctions ϕ_3 et ϕ_4 ne convergent pas.

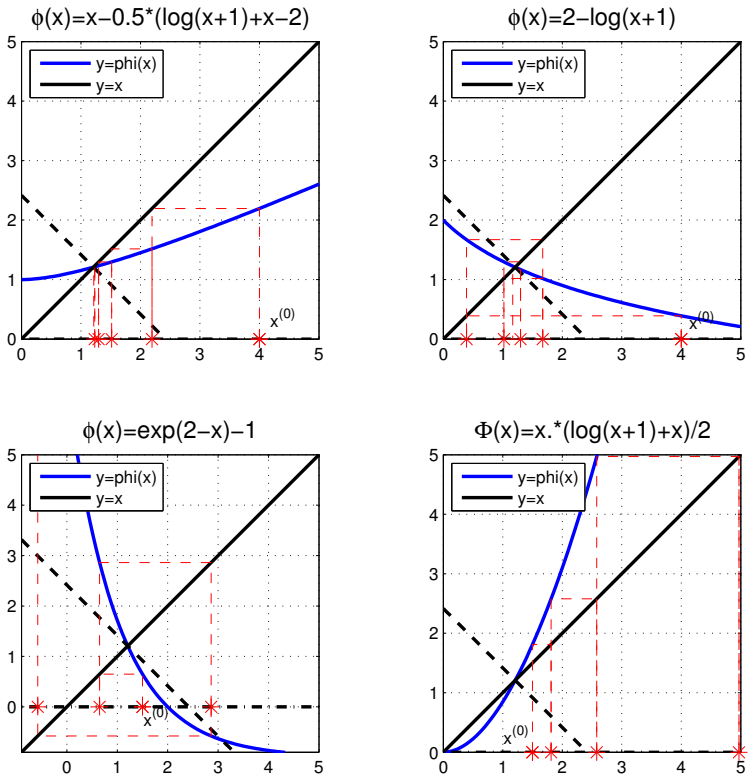


FIGURE 1.3 – 5 premières itérations de la méthode de point fixe appliquée aux équations de point fixe (1.8). La ligne en trait pointillé est la droite de pente -1 qui passe par le point fixe.

Analyse de convergence

D'après une inspection plus approfondie des graphiques en figure 1.3, on note que la méthode de point fixe converge lorsque la pente de la fonction ϕ au point fixe, $\phi'(\alpha)$, est comprise entre $(-1, 1)$ tandis qu'elle diverge lorsque $|\phi'(\alpha)| > 1$. On arrive à cette conclusion au moins pour tous les points de départ $x^{(0)}$ dans l'intervalle $[0, 5]$. Pour des points de départ plus loin du point fixe, cette conclusion pourrait être fausse.

On va maintenant établir ce résultat de façon plus rigoureuse. On s'intéresse à étudier le comportement de l'erreur $e^{(k)} = |x^{(k)} - \alpha|$ à la k-ème itération. On rappelle le développement de Taylor du 1^{er} ordre de ϕ autour du point fixe α

$$\phi(x) = \underbrace{\phi(\alpha)}_{=\alpha} + \phi'(\alpha)(x - \alpha) + \underbrace{R(x)}_{o(|x-\alpha|)}$$

où

$$R(x) = o(|x - \alpha|) \quad \Longleftrightarrow \quad \lim_{x \to \alpha} \frac{R(x)}{|x - \alpha|} = 0,$$

c'est-à-dire la fonction $R(x)$ tend vers zéro plus rapidement que la fonction $|x-\alpha|$, lorsque $x \to \alpha$. On peut donc estimer l'erreur à la première itération de la façon suivante

$$x^{(1)} - \alpha = \phi(x^{(0)}) - \phi(\alpha) = \phi'(\alpha)(x^{(0)} - \alpha) + \underbrace{o(|x^{(0)} - \alpha|)}_{\text{terme petit}}$$

et donc

$$e^{(1)} = |\phi'(\alpha)|e^{(0)} + \underbrace{o(e^{(0)})}_{\text{terme petit}} .$$

Le terme $o(e^{(0)})$ est en fait négligeable par rapport au terme $|\phi'(\alpha)|e^{(0)}$ si $x^{(0)}$ est suffisamment proche de α. On peut maintenant itérer le raisonnement

$$\begin{aligned} e^{(k)} &= |\phi'(\alpha)|e^{(k-1)} + \text{terme petit} \\ &= |\phi'(\alpha)|^2 e^{(k-2)} + \text{terme petit} \\ &\dots \\ &= |\phi'(\alpha)|^k e^{(0)} + \text{terme petit.} \end{aligned}$$

Donc, *l'erreur $e^{(k)}$ tend vers zéro si $|\phi'(\alpha)| < 1$* comme on l'avait déjà deviné graphiquement. Le résultat que l'on vient de montrer est récapitulé dans le théorème suivant :

Theorème 1.1. *Soit $\phi : [a, b] \to \mathbb{R}$ une fonction de classe C^1 (continûment différentiable) et $\alpha \in [a, b]$ un point fixe de ϕ.*
Si

$$|\phi'(\alpha)| < 1$$

***alors**, il existe $\delta > 0$ tel que*

— *pour tout* $x^{(0)} \in [\alpha - \delta, \alpha + \delta]$ *les itérations de point fixe* $x^{(k+1)} = \phi(x^{(k)})$ *convergent vers* α*, c'est-à-dire :* $\lim_{k\to\infty} x^{(k)} = \alpha$
— *de plus,*

$$\lim_{k\to\infty} \frac{|x^{(k+1)} - \alpha|}{|x^{(k)} - \alpha|} = |\phi'(\alpha)|.$$

Contrôle de l'erreur

L'algorithme 1.3 n'est pas complet car dans la pratique il faut donner un critère d'arrêt des itérations (on veut éviter des boucles infinies !).

Idéalement, on aimerait terminer les itérations lorsque l'erreur $e^{(k)} = |x^{(k)} - \alpha|$ est plus petite qu'une tolérance donnée. Malheureusement, on ne peut pas utiliser ce critère car on ne connaît pas la solution exacte. On procède donc différemment.

Imaginons d'avoir calculé l'itération $x^{(k)}$. Si $x^{(k)}$ était le point fixe de ϕ, alors on aurait $x^{(k)} = \phi(x^{(k)})$. Fort probablement ceci n'est pas vrai et $x^{(k)} - \phi(x^{(k)}) \neq 0$. On définit le *résidu*

$$r^{(k)} = |x^{(k)} - \phi(x^{(k)})|.$$

On s'attend à ce que le résidu soit petit si $x^{(k)}$ est proche du point fixe α et il peut donc être utilisé comme *indicateur d'erreur*. Ceci nous porte à introduire le

$$\textbf{critère d'arrêt} : \quad |x^{(k)} - \phi(x^{(k)})| \leq tol.$$

Voici une version complète de la méthode de point fixe

Algorithme 1.4 : Méthode de point fixe avec critère d'arrêt

Données : ϕ, $x^{(0)}$, tol
Résultat : α, res, niter
$r^{(0)} = tol + 1$;
k=0;
tant que $r^{(k)} > tol$ **faire**
  $k = k + 1$;
  $x^{(k+1)} = \phi(x^{(k)})$;
  $r^{(k)} = |x^{(k+1)} - x^{(k)}|$;
fin
$\alpha = x^{(k)}$, res=$r^{(k)}$, niter=k ;

L'algorithme 1.4 s'arrête donc à l'itération k telle que $r^{(k)} \leq tol$. Il reste quand même à vérifier si le fait que $r^{(k)} \leq tol$ implique que la *vraie erreur* $|x^{(k)} - \alpha|$ est aussi inférieure à la tolérance tol ou au moins, du même ordre

de grandeur. Ceci n'est pas garanti, en général, mais on a le résultat suivant :

$$\begin{aligned} x^{(k)} - \alpha &= x^{(k)} - \phi(x^{(k)}) + \phi(x^{(k)}) - \alpha \\ &= x^{(k)} - \phi(x^{(k)}) + \phi(x^{(k)}) - \phi(\alpha) \\ &= x^{(k)} - \phi(x^{(k)}) + \phi'(\xi^{(k)})(x^{(k)} - \alpha) \end{aligned}$$

où $\xi^{(k)}$ est un point convenable de l'intervalle $[\alpha, x^{(k)}]$. Donc

$$|x^{(k)} - \alpha| = \frac{1}{|1 - \phi'(\xi^{(k)})|} r^{(k)}.$$

Puisque le résidu est inférieur à *tol*, tout ce qu'on sait dire sur l'erreur est

$$e^{(k)} \leq \frac{1}{|1 - \phi'(\xi^{(k)})|} tol.$$

On voit que si $|\phi'(\xi^{(k)})| \ll 1$ on aura $e^{(k)} \approx tol$ tandis que si $|\phi'(\xi^{(k)})| \approx 1$ la vraie erreur peut être beaucoup plus grande que le résidu et notre contrôle de l'erreur n'est pas fiable. Notez que $\phi'(\xi^{(k)}) \xrightarrow{k\to\infty} \phi'(\alpha)$ si la méthode converge, donc ce qui est important, encore une fois, est la valeur de la pente de ϕ au point fixe.

1.3.3 Méthodes d'ordre supérieur

Soit α le point fixe de ϕ. On donne la définition suivante :

Définition 1.1. *Une méthode de point fixe $x^{(k+1)} = \phi(x^{(k)})$ est dite* d'ordre p *si, lorsque la suite $\{x^{(k)}\}$ converge à α, il existe $C > 0$ tel que*

$$\lim_{k\to\infty} \frac{|x^{(k+1)} - \alpha|}{|x^{(k)} - \alpha|^p} = C.$$

Si la méthode est d'ordre 1 alors on doit avoir $C < 1$.

Les méthodes qu'on a vu pour les fonctions ϕ_1 et ϕ_2 en (1.8) sont des méthodes d'ordre 1. On peut le vérifier en `Python` par exemple pour la fonction ϕ_1. Puisqu'on ne connaît pas la solution exacte, on utilise le résultat

$$|x^{(k)} - \alpha| = \frac{1}{|1 - \phi'(\xi^{(k)})|} r^{(k)}$$

et donc

$$\lim_{k\to\infty} \frac{r^{(k+1)}}{(r^{(k)})^p} = \lim_{k\to\infty} \frac{|1 - \phi'(\xi^{(k+1)})|}{|1 - \phi'(\xi^{(k)})|^p} \frac{|x^{(k+1)} - \alpha|}{|x^{(k)} - \alpha|^p} = \frac{1}{|1 - \phi'(\alpha)|^{p-1}} C \neq 0.$$

On vérifie donc si la limite $\lim_{k\to\infty} \frac{r^{(k+1)}}{(r^{(k)})^p}$ tend vers une constante (non nulle) pour $p = 1$. On a utilisé la fonction `fixedpoint` disponible sur Moodle :

```
import numpy as np
from fixedPoint import fixed_point
phi=lambda x: x-0.5*(np.log(x+1)+x-2); x0=4; tol=1e-3; nmax=1000;
x, res, niter = fixed_point( phi, x0, tol, nmax )
print('x ')
print(x)
print('res')
print(res)
print('Number of iterations')
print(niter)
# Prints ratio of res:
print('Ratio of RES')
print(res[1:]/(res[:-1]**2))

## OUTPUT
# x
# [4.         2.19528104 1.516803   1.29690678 1.23267172 1.21473639
#  1.2098015  1.2084494 ]
# res
# [1.80471896e+00 6.78478046e-01 2.19896221e-01 6.42350574e-02
#  1.79353340e-02 4.93488414e-03 1.35209708e-03 3.70023111e-04]
# Number of iterations
# 7
# Ratio of RES
# [  0.20831313   0.47769002   1.3284236    4.34675574  15.34115371
#   55.52057452 202.40120959]
```

On voit bien que si on prend $p = 1$ le rapport $r^{(k+1)}/r^{(k)}$ tend vers une constante tandis que si on prend $p = 2$ le rapport diverge.

D'après la définition 1.1 pour une méthode d'ordre p on a

$$|x^{(k+1)} - \alpha| \approx C|x^{(k)} - \alpha|^p$$

lorsque $x^{(k)}$ est suffisamment proche de α. Illustrons l'intérêt d'avoir une méthode d'ordre supérieur à 1 sur un exemple : imaginons d'avoir une erreur initiale $|x^{(0)} - \alpha| = 10^{-1}$ et d'avoir deux méthodes de point fixe ; la première d'ordre 1 avec constance $C = 0.5$ et la deuxième d'ordre 2 avec constante $C = 1$. On s'attend à avoir les erreurs indiquées dans le tableaux

	méthode d'ordre 1 ($C = 0.5$)	méthode d'ordre 2 ($C = 1$)
$e^{(0)} = \|x^{(0)} - \alpha\|$	0.1	0.1
$e^{(1)} = \|x^{(1)} - \alpha\|$	0.05	10^{-2}
$e^{(2)} = \|x^{(2)} - \alpha\|$	0.025	10^{-4}
$e^{(3)} = \|x^{(3)} - \alpha\|$	0.0125	10^{-8}

Il est clair d'après le tableaux que l'erreur de la méthode d'ordre 2 décroît énormément plus vite que celle de la méthode d'ordre 1 ! On est

donc intéressé à comprendre sous quelles conditions une méthode est d'ordre supérieur à 1. Ceci nous permettra de développer de nouvelles méthodes d'ordre élevé.

On revient à l'estimation d'erreur des méthodes de point fixe :

$$x^{(k+1)} - \alpha = \phi(x^{(k)}) - \phi(\alpha) = \phi'(\alpha)(x^{(k)} - \alpha) + \underbrace{o(|x^{(k)} - \alpha|)}_{\text{terme petit}}.$$

Que se passe-t-il si $\phi'(\alpha) = 0$? Dans ce cas on peut écrire un développement de Taylor à l'ordre 2

$$x^{(k+1)} - \alpha = \phi(x^{(k)}) - \phi(\alpha) = \underbrace{\phi'(\alpha)(x^{(k)} - \alpha)}_{=0} + \frac{1}{2}\phi''(\alpha)(x^{(k)} - \alpha)^2 + \underbrace{o(|x^{(k)} - \alpha|^2)}_{\text{terme petit}}.$$

Si on néglige les termes "petits" on peut dire que

$$x^{(k+1)} - \alpha \approx \frac{1}{2}\phi''(\alpha)(x^{(k)} - \alpha)^2$$

lorsque $x^{(k)}$ est près de α. La méthode sera donc d'ordre 2.

Plus généralement, si $\phi'(\alpha) = \phi''(\alpha) = \ldots = \phi^{p-1}(\alpha) = 0$ et $\phi^p(\alpha) \neq 0$, la méthode sera d'ordre p. On recueille ces considérations dans le théorème suivant :

Theorème 1.2. *Soit $\phi : [a, b] \to \mathbb{R}$ une fonction de classe C^p (p dérivées continues) et $\alpha \in [a, b]$ un point fixe de ϕ.*
Si

$$\phi'(\alpha) = \phi''(\alpha) = \ldots = \phi^{p-1}(\alpha) = 0 \quad et \quad \phi^p(\alpha) \neq 0, \;\; p > 1$$

alors *il existe $\delta > 0$ tel que, $\forall x_0 \in [\alpha - \delta, \alpha + \delta]$*

- *les itérations de point fixe $x^{(k+1)} = \phi(x^{(k)})$ convergent à α*
- *la méthode de point fixe est d'ordre p*
- *de plus,*

$$\lim_{k\to\infty} \frac{|x^{(k+1)} - \alpha|}{|x^{(k)} - \alpha|^p} = \frac{1}{p!}|\phi^p(\alpha)|.$$

On étudiera dans la prochaine Section une méthode d'ordre 2, notamment la méthode de Newton.

Contrôle de l'erreur

On reprend l'estimation qui permet de lier l'erreur au résidu :

$$|x^{(k)} - \alpha| = \frac{1}{|1 - \phi'(\xi^{(k)})|} r^{(k)}.$$

Pour une méthode d'ordre $p > 1$ on a $\phi'(\alpha) = 0$ et dans ce cas

$$|x^{(k)} - \alpha| \approx r^{(k)}$$

lorsque $x^{(k)}$ est près de α. Il en suit que pour une méthode au moins quadratique l'estimation de l'erreur par le résidu est très fiable.

1.4 Méthode de Newton (ou Newton Raphson)

La méthode de Newton est une des méthodes les plus utilisées pour résoudre des équations non linéaires. Elle est basée sur un développement de Taylor à l'ordre 1. Soit $f(x) = 0$ l'équation à résoudre et α le zéro cherché. On part d'une donnée initiale $x^{(0)}$ et on écrit le développement à l'ordre 1 :

$$f(\alpha) = f(x^{(0)}) + f'(x^{(0)})(\alpha - x^{(0)}) + \text{“terme petit”}.$$

Si on néglige le “terme petit” et on tient compte du fait que $f(\alpha) = 0$, on a

$$f(x^{(0)}) + f'(x^{(0)})(\alpha - x^{(0)}) \approx 0$$

d'où on peut obtenir une approximation du zéro

$$\alpha \approx x^{(0)} - \frac{f(x^{(0)})}{f'(x^{(0)})}.$$

On appelle cette approximation $x^{(1)}$. On a obtenu ainsi la formule

$$x^{(1)} = x^{(0)} - \frac{f(x^{(0)})}{f'(x^{(0)})}$$

que l'on peut itérer et calculer $x^{(2)}$ à partir de $x^{(1)}$ et ainsi de suite.

Algorithme 1.5 : Méthode de Newton (sans critère d'arrêt)

Étant donnés f, f' et $x^{(0)}$;
pour $k = 0, 1, \ldots$ **faire**
  $x^{(k+1)} = x^{(k)} - \dfrac{f(x^{(k)})}{f'(x^{(k)})}$
fin

La figure 1.4 donne une interprétation graphique de la méthode. À partir de la donnée initiale $x^{(0)}$, on approche la courbe $f(x)$ par la droite tangente en $x^{(0)}$ et on calcule le zéro de la droite tangente.

Analyse de convergence

Il suffit de reconnaître que l'algorithme 1.5 peut être vu comme un algorithme de point fixe

$$x^{(k+1)} = \phi(x^{(k)}) \quad \text{où} \quad \phi(x) = x - \frac{f(x)}{f'(x)}.$$

Pour comprendre l'ordre de la méthode il suffit de calculer $\phi'(\alpha)$, $\phi''(\alpha)$, ... On a (après quelques calculs)

$$\phi'(x) = \frac{f(x)f''(x)}{(f'(x))^2}$$
$$\phi''(x) = \frac{f'(x)^2 f''(x) + f(x)f'(x)f'''(x) - 2f(x)f''(x)^2}{(f'(x))^3}$$

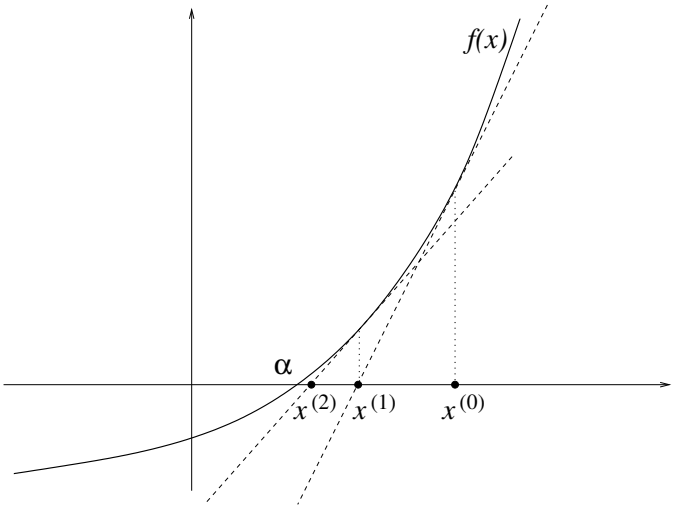


FIGURE 1.4 – Illustration graphique de la méthode de Newton

et donc

$$\phi'(\alpha) = 0 \qquad \text{si } f'(\alpha) \neq 0$$
$$\phi''(\alpha) = \frac{f''(\alpha)}{f'(\alpha)} \neq 0 \qquad \text{si } f''(\alpha) \neq 0.$$

La méthode sera, en générale, d'ordre 2.

Theorème 1.3. *Soit f une fonction de classe C^2 et α un zéro de f. Si $f'(\alpha) \neq 0$ et $f''(\alpha) \neq 0$, la méthode de Newton converge à l'ordre 2 pour tout $x^{(0)}$ suffisamment proche de α. De plus*

$$\lim_{k\to\infty} \frac{|x^{(k+1)} - \alpha|}{|x^{(k)} - \alpha|^2} = \frac{1}{2}\frac{|f''(\alpha)|}{|f'(\alpha)|}.$$

Il est important de remarquer que la convergence établie par le théorème 1.3 est seulement *locale*, c'est-à-dire la convergence est garantie seulement si la donnée initiale $x^{(0)}$ est suffisamment proche du zéro α. La figure 1.5 montre deux cas de non convergence de la méthode de Newton si la donnée initiale est trop loin du zéro.

Remarque 1.2. *Si $f'(\alpha) = 0$ on peut montrer que la méthode de Newton est seulement d'ordre 1.*

Contrôle de l'erreur

Comme on l'a vu dans la section 1.3.3, pour une méthode d'ordre 2, le contrôle de l'erreur par l'incrément $|x^{(k+1)} - x^{(k)}|$ (résidu de l'équation de point fixe correspondante) est très fiable pourvu que $x^{(k)}$ soit suffisamment

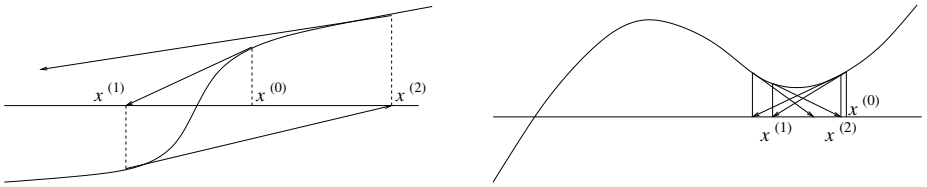


FIGURE 1.5 – Deux cas de non convergence de la méthode de Newton si la donnée initiale est trop loin du zéro

proche de α. On peut donc utiliser le critère d'arrêt suivant pour la méthode de Newton

$$|x^{(k+1)} - x^{(k)}| \leq tol.$$

L'algorithme complet est donné ci après :

Algorithme 1.6 : Méthode de Newton avec critère d'arrêt

Données : f, f', $x^{(0)}$, tol, nmax
Résultat : α, res, niter
$r^{(0)} = tol + 1$;
k=0;
tant que $r^{(k)} > tol$ *AND* $k < nmax$ **faire**
  $x^{(k+1)} = x^{(k)} - \frac{f(x^{(k)})}{f'(x^{(k)})}$;
  $r^{(k+1)} = |x^{(k+1)} - x^{(k)}|$;
  $k = k + 1$;
fin
$\alpha = x^{(k)}$, res=$r^{(k)}$, niter=k ;

Exemple 1.2 (Circuit éléctrique)**.** *On revient à l'exemple du circuit électrique et on résout l'équation* (1.2) *par la méthode de Newton à partir du point initial* $x^{(0)} = 0.1$. *Pour cela on utilise la fonction* `newton` *disponible sur Moodle*

```
import numpy as np
from newton import newton

i0=1; v0=0.1; R=1; V=1
f =lambda x: R*i0*(np.exp(x/v0)-1)+x-V
df= lambda x: R*i0*np.exp(x/v0)/v0+1
x0=0
zero, res,niter,inc=newton(f,df,x0,1e-8,100000)
print('Results')
print('zero = '+str(zero))
print('residual = '+str(res))
print('number of iterations = '+str(niter))

## OUTPUT
```

```
#Results
#zero = 0.06596105346440571
#residual = 3.5527136788005O1e-15
#number of iterations = 5
```

On voit bien que la méthode de Newton converge bien plus vite que la méthode de point fixe (1.4). *La fonction* `newton` *retourne aussi la liste des incréments* $|x^{(k+1)} - x^{(k)}|$ *dans la variable* `inc`. *Ceci nous permet de contrôler l'ordre de convergence*

```
import numpy as np
from newton import newton

i0=1; v0=0.1; R=1; V=1
f =lambda x: R*i0*(np.exp(x/v0)-1)+x-V
df= lambda x: R*i0*np.exp(x/v0)/v0+1
x0=0
zero, res,niter,inc=newton(f,df,x0,1e-8,100000)
print('convergence')
print('--------------------------------')
for i in range(1,5):
    print('order '+str(i))
    print(inc[1:]/inc[:-1]**i)

##OUTPUT
#order 1
#[2.44095936e-01 1.22648213e-01 1.31765451e-02 1.70533784e-04]
#order 2
#[2.68505529 5.5270496  4.8414167  4.75532045]
#order 3
#[2.95356082e+01 2.49072339e+02 1.77886658e+03 1.32601717e+05]
#order 4
#[3.24891691e+02 1.12242579e+04 6.53603376e+05 3.69758790e+09]
```

On voit en fait que le rapport $|x^{(k+1)} - x^{(k)}|/|x^{(k)} - x^{(k-1)}|^2$ *pour* $k = 1, 2, 3, 4$ *se stabilise sur une valeur presque constante (environ 5) tandis que le rapport* $|x^{(k+1)} - x^{(k)}|/|x^{(k)} - x^{(k-1)}|$ *tend vers zéro, ce qui confirme que la méthode converge à l'ordre 2.*

1.5 Systèmes d'équations non linéaires

On souhaite maintenant généraliser les méthodes qu'on a vu au cas de la résolution d'un système d'équations non linéaires

$$\begin{cases} f_1(x_1, \ldots, x_n) = 0 \\ f_2(x_1, \ldots, x_n) = 0 \\ \ldots \\ f_n(x_1, \ldots, x_n) = 0. \end{cases}$$

On introduit la notation compacte

$$\mathbf{f}(\mathbf{x}) = \mathbf{0}$$

où

$$\mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}, \qquad \mathbf{f}(\mathbf{x}) = \begin{bmatrix} f_1(x_1, \ldots, x_n) \\ \vdots \\ f_n(x_1, \ldots, x_n) \end{bmatrix}.$$

On appelle $\boldsymbol{\alpha} = [\alpha_1, \ldots, \alpha_n]^T$ le zéro de $\mathbf{f}$, c'est-à-dire $\boldsymbol{\alpha} \in \mathbb{R}^n$ est le point de $\mathbb{R}^n$ qui satisfait $\mathbf{f}(\boldsymbol{\alpha}) = \mathbf{0}$.

Exemple 1.3. *Considérons le système de deux équations non linéaires à deux inconnues (x_1, x_2) :*

$$\begin{cases} f_1(x_1, x_2) = x_1^2 + x_1 x_2 - 10 = 0 \\ f_2(x_1, x_2) = x_2 + 3x_1 x_2^2 - 57 = 0. \end{cases} \tag{1.9}$$

La première équation définit implicitement la courbe

$$f_1(x_1, x_2) = x_1^2 + x_1 x_2 - 10 = 0 \qquad \Longrightarrow \qquad x_2 = g_1(x_1) = \frac{10 - x_1^2}{x_1}$$

tandis que la deuxième équation définit la courbe

$$f_2(x_1, x_2) = x_2 + 3x_1 x_2^2 - 57 = 0 \qquad \Longrightarrow \qquad x_1 = g_2(x_2) = \frac{57 - x_2}{3x_2^2}.$$

On peut visualiser ces deux courbes en `Python` *. Leur intersection donne le zéro cherché.*

```
import numpy as np
import matplotlib.pyplot as plt
g1= lambda x1: (10-x1**2)/x1;
g2= lambda x2: (57-x2)/(3*x2**2);
#plot de x2=g1(x1)
x1=np.linspace(1,3,50)
plt.plot(x1,g1(x1),'b',LineWidth=2)
#plot de x1=g2(x2)
x2=np.linspace(1,4.5,50)
plt.plot(g2(x2),x2,'r',LineWidth=2)
plt.legend([r'$f_1(x_1,x_2)$',r'$f_2(x_1,x_2)$'])
plt.grid(True)
plt.xlim([1,4.5])
```

La figure 1.6 montre les deux courbes $x_2 = g_1(x_2)$ et $x_1 = g_2(x_2)$.

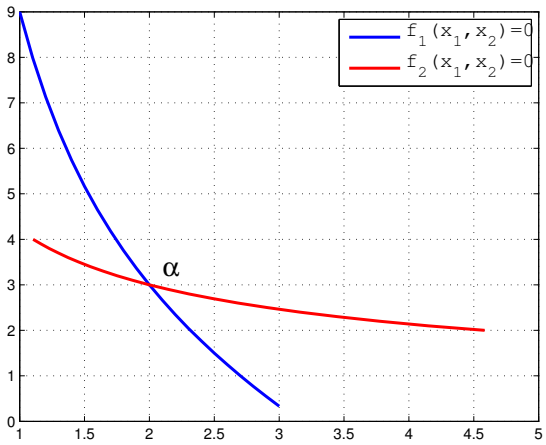


FIGURE 1.6 – Intersection des deux fonctions $f_1(x_1, x_2) = 0$ (bleu) et $f_2(x_1, x_2) = 0$ (rouge) de (1.9)

1.5.1 Méthode de Newton pour des systèmes d'équations

Pour dériver la méthode de Newton pour des systèmes d'équations, on procède comme dans le cas d'une équations scalaire par développement de Taylor à l'ordre 1. Étant donné un point initial $\mathbf{x}^{(0)}$ on a

$$\mathbf{0} = \mathbf{f}(\boldsymbol{\alpha}) = \mathbf{f}(\mathbf{x}^{(0)}) + J_{\mathbf{f}}(\mathbf{x}^{(0)})(\boldsymbol{\alpha} - \mathbf{x}^{(0)}) + \text{“terme petit”} \tag{1.10}$$

où la matrice *Jacobienne* $J_{\mathbf{f}} \in \mathbb{R}^{n\times n}$ est définie par

$$J_{\mathbf{f}} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \vdots \\ \vdots & & & \vdots \\ \frac{\partial f_n}{\partial x_1} & \cdots & & \frac{\partial f_n}{\partial x_n} \end{bmatrix}.$$

Si on néglige le “terme petit” dans (1.10), on peut résoudre l'équation vectorielle et trouver une approximation du zéro $\boldsymbol{\alpha}$ que l'on appellera $\mathbf{x}^{(1)}$ satisfaisant

$$J_{\mathbf{f}}(\mathbf{x}^{(0)})(\mathbf{x}^{(1)} - \mathbf{x}^{(0)}) = -\mathbf{f}(\mathbf{x}^{(0)}).$$

On prend $\mathbf{x}^{(1)}$ comme nouveau point de départ et on procède ainsi de suite. Remarquez que pour calculer $\mathbf{x}^{(1)}$ il faut *résoudre un système linéaire*. On verra dans les Chapitres 4 et 5 comment ceci peut être fait de façon efficace. On se limite à dire ici que la résolution d'un système linéaire en `Python` peut se faire tout simplement par la commande \, c'est-à-dire pour résoudre le système $A\mathbf{x} = \mathbf{b}$, on peut utiliser la commande `x=A\b`.

On peut arrêter les itérations lorsque $\|\mathbf{x}^{(k+1)}-\mathbf{x}^{(k)}\| \leq tol$ où on a indiqué par $\|v\|$ la norme Euclidienne du vecteur $v \in \mathbb{R}^n$, $\|v\| = \sqrt{v_1^2 + \ldots + v_n^2}$.

Algorithme 1.7 : Méthode de Newton pour des systèmes d'équations

Données : $\mathbf{f}$, $J_{\mathbf{f}}$, $\mathbf{x}^{(0)}$, tol, nmax
Résultat : $\boldsymbol{\alpha}$, res, niter
$r^{(0)} = tol + 1$;
k=0;
tant que $r^{(k)} > tol$ *AND* $k < nmax$ **faire**
  $J_{\mathbf{f}}(\mathbf{x}^{(k)})\delta\mathbf{x} = -\mathbf{f}(\mathbf{x}^{(k)})$;
  $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \delta\mathbf{x}$;
  $r^{(k+1)} = \|\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}\|$;
  $k = k + 1$;
fin
$\boldsymbol{\alpha} = \mathbf{x}^{(k)}$, res=$r^{(k)}$, niter=$k$;

L'analyse de convergence se fait de la même manière que pour le cas scalaire. On se limite à énoncer le résultat principal

Theorème 1.4. *Soit* $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^n$ *une fonction de classe* C^2 *et* $\boldsymbol{\alpha}$ *un zéro de* $\mathbf{f}$.
Si $\det(J_{\mathbf{f}}(\alpha)) \neq 0$ *(la matrice Jacobienne est inversible),*
alors *pour tout* $\mathbf{x}^{(0)}$ *suffisamment proche de* $\boldsymbol{\alpha}$ *la méthode de Newton converge à l'ordre 2.*

Exemple 1.4. *On reprend le système* (1.9) *et on considère le point de départ* $\mathbf{x}^{(0)} = (1, 0)$. *Calculons la première itération de Newton.*

$$\mathbf{f}(\mathbf{x}^{(0)}) = \begin{bmatrix} f_1(1,0) \\ f_2(1,0) \end{bmatrix} = \begin{bmatrix} -9 \\ -57 \end{bmatrix}$$

$$J_{\mathbf{f}}(\mathbf{x}) = \begin{bmatrix} 2x_1 + x_2 & x_1 \\ 3x_2^2 & 1 + 6x_1x_2 \end{bmatrix} \qquad \Longrightarrow \qquad J_{\mathbf{f}}(\mathbf{x}^{(0)}) = \begin{bmatrix} 2 & 1 \\ 0 & 1 \end{bmatrix}.$$

On doit donc résoudre le système linéaire

$$\begin{bmatrix} 2 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \delta x_1 \\ \delta x_2 \end{bmatrix} = \begin{bmatrix} 9 \\ 57 \end{bmatrix}$$

dont la solution est $\delta\mathbf{x} = (-24, 57)$. *Finalement on a* $\mathbf{x}^{(1)} = \mathbf{x}^{(0)} + \delta\mathbf{x} = (-23, 57)$.

Exemple 1.5. *On résout maintenant le système* (1.9) *en* **Python** *par la méthode de Newton, à l'aide de la fonction* `newtonsys` *disponible sur Moodle. On choisit comme point de départ* $\mathbf{x}^{(0)} = [3, 4]^T$.

```
import numpy as np
from newtonsys import *

f=lambda x: np.array([x[0]**2+x[0]*x[1]-10,
                      x[1]+3*x[0]*x[1]**2-57])
df= lambda x: np.array([[2*x[0]+x[1], x[0]],
                        [3*x[1]**2,1+6*x[0]*x[1]]])
x0=[3,4]
x,inc,niter=newtonsys(f,df,x0,1e-8,1000);
#OUTPUT
print('x= '+str(x[-1]))
print('number of iterations '+str(niter))
print('increments=' +str(inc))
#     x= [2. 3.]
# number of iterations= 5
# increments =[1.11487660e+00 3.27957413e-01
#       3.58120064e-02 1.96157782e-04
#  1.10946117e-09]
```

On voit que la méthode converge vers la solution exacte $\boldsymbol{\alpha} = (2, 3)$ *en 5 itérations. On peut aussi contrôler l'ordre de convergence. Comme dans le cas scalaire, la fonction* `newtonsys` *retourne aussi la liste des incréments* $\|\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}\|$ *dans la variable* `inc`*. On peut donc évaluer le rapport de deux incréments consécutifs*

$$\frac{\|\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}\|}{\|\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}\|^p}$$

pour des p *différents.*

```
import numpy as np
from newtonsys import *

f=lambda x: np.array([x[0]**2+x[0]*x[1]-10,
                      x[1]+3*x[0]*x[1]**2-57])
df= lambda x: np.array([[2*x[0]+x[1], x[0]],
                        [3*x[1]**2,1+6*x[0]*x[1]]])
x0=[3,4]
x,inc,niter=newtonsys(f,df,x0,1e-8,1000);
#OUTPUT
print('x= '+str(x[-1]))
print('number of iterations '+str(niter))
print('increments=' +str(inc))
print('ratio with p=1')
print(inc[1:]/inc[:-1])
```

```
# ratio with p=1
# [2.94164765e-01 1.09197124e-01 5.47743064e-03 5.65596310e-06]
print('ratio with p=2')
print(inc[1:]/inc[:-1]**2.0)
# ratio with p=2
# [0.2638541  0.33296129 0.15294956 0.02883374]
print('ratio with p=3')
print(inc[1:]/inc[:-1]**3.0)
# ratio with p=3
# [  0.23666664   1.01525771   4.27090173 146.99260448]
```

On voit que le rapport qui semble être le plus constant est celui qui correspond à $p = 2$. On en conclut que la méthode converge à l'ordre 2.

Chapitre 2

Approximation de données (Curve fitting)

Supposons qu'on ait à disposition des mesures d'une quantité y, par exemple la température de l'eau du Lac Leman, à n différentes profondeurs. Soient x_i, $i = 1, \ldots, n$ les profondeurs auxquelles on a acquis les mesures et y_i, $i = 1, \ldots, n$ les températures mesurées correspondantes.

On essaye de trouver une fonction continue $p(x)$ qui permet de décrire au mieux les données (x_i, y_i), c'est-à-dire $p(x_i) \approx y_i$.

Une autre façon de voir ce problème est la suivante : on fait l'hypothèse qu'il existe une fonction f qui décrit le lien entre y et x, c'est-à-dire $y = f(x)$, définie pour tout x et non seulement pour les x_i où on a acquis la mesure. On dit que $y = f(x)$ est un *modèle conceptuel*. Dans l'exemple précédent, on imagine qu'il existe une fonction $y = f(x)$ qui donne la température de l'eau à chaque profondeur du lac. Toutefois, on ne connaît de cette fonction que les valeurs (x_i, y_i) que l'on a mesuré. On s'intéresse ici à reconstruire la fonction $f(x)$ à partir des mesures disponibles.

Ceci nous permettra d'estimer la température de l'eau aussi dans toute autre profondeur que les valeurs x_i.

Exemple 2.1 (Viscosité dynamique de l'eau)**.** *La viscosité de l'eau dépend de la température. Le Tableau 2.1 est pris de Wikipedia. La Figure 2.1 (gauche) visualise les valeurs du tableau. Supposons qu'on soit intéressé à la valeur de viscosité de l'eau à 25°C. Cette température n'est pas dans le tableau. Pour trouver une bonne approximation de la viscosité à cette température, on peut procéder de la façon suivante : on cherche d'abord une fonction continue $p(T)$ qui* interpole *les données, c'est-à-dire*

$$\textit{on cherche } p(T) \textit{ telle que} \qquad p(T_i) = \nu_i$$

*où $T_1 = 10, T_2 = 20, \ldots, T_{10} = 100$ sont les valeurs de température du tableau et $\nu_1 = 1.308, \nu_2 = 1.002, \ldots, \nu_{10} = 0.2822$ les valeurs de viscosité correspondantes. On parle dans ce cas d'*interpolation de données.

Température [°C]	Viscosité [mPa·s]
10	1.308
20	1.002
30	0.7978
40	0.6531
50	0.5471
60	0.4658
70	0.4044
80	0.3550
90	0.3150
100	0.2822

TABLE 2.1 – Viscosité de l'eau en fonction de la température (source Wikipedia)

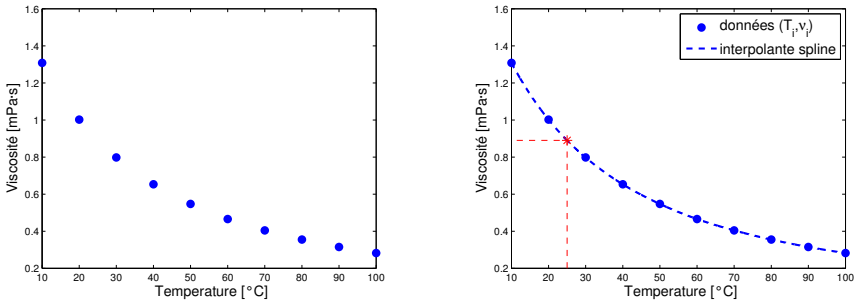


FIGURE 2.1 – Gauche : données de viscosité de l'eau en fonction de la température. Droite : fonction interpolant les données, obtenue par interpolation spline. En rouge la viscosité a 25°C

Une fois une telle fonction trouvée, on peut évaluer la viscosité à la température de 25 °C par $\nu = p(25)$. *La Figure 2.1 (droite) montre une fonction interpolant possible obtenue par* interpolation spline *qui sera présentée dans la Section 2.3.*

Exemple 2.2 (Dog bone tensile test)**.** *On souhaite mesurer les caractéristiques mécaniques d'un matériau. Pour cela, on fait un test de traction sur un échantillon de la forme montrée en Figure 2.2 (dog bone). L'échantillon est sujet à une contrainte* σ *(force par unité de surface). La déformation* ε *correspondante est mesurée par un dispositif appelé* jauge de déformation. *On applique plusieurs contraintes* σ_i, $i = 1, \ldots, n$ *équiréparties entre* 0 *et* σ_{max} *et on mesure les déformations correspondantes* ε_i. *Par cela on souhaite caractériser la loi contrainte–déformation du matériau. Toutefois, les mesures* ε_i *sont affectées par une erreur de mesure non négligeable. La Figure 2.3 (gauche) montre des données possibles obtenues dans l'expérience.*

On imagine qu'il existe une "vraie" fonction f *qui lie la déformation à la*

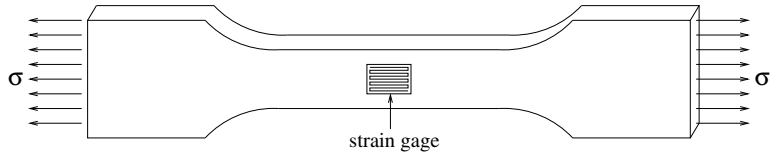


FIGURE 2.2 – Échantillon d'un matériau pour une épreuve de traction

contrainte : $\varepsilon = f(\sigma)$ et que les mesures ε_i sont données par $\varepsilon_i = f(\sigma_i) + \eta_i$ où η_i sont des erreurs de mesure.

On souhaite trouver une approximation $p(\sigma)$ de la "vraie" fonction $f(\sigma)$ à partir des données $(\sigma_i, \varepsilon_i)$. Néanmoins, cette fois-ci, ce n'est pas une bonne idée de chercher une fonction interpolant *car on va interpoler aussi l'erreur de mesure. Il vaut mieux chercher une fonction $p(\sigma)$ qui approche "au mieux" les données et en même temps "filtre" l'erreur de mesure. On peut chercher une fonction p dans une classe de fonctions $\mathcal{W}$ (par exemple des polynômes de degré fixé) qui minimise la somme des distances des données au carré*

$$p = \underset{q \in \mathcal{W}}{\operatorname{argmin}} \sum_{i=1}^{n} |\varepsilon_i - q(\sigma_i)|^2$$

*On parle dans ce cas d'*approximation aux moindres carrés*. La Figure 2.3 montre au centre une reconstruction de la loi contrainte-déformation par interpolation spline des données, et à droite une reconstruction obtenue par approximation polynomiale de degré 3 aux moindres carrés. Notre intuition physique nous porte à préférer la deuxième méthode.*

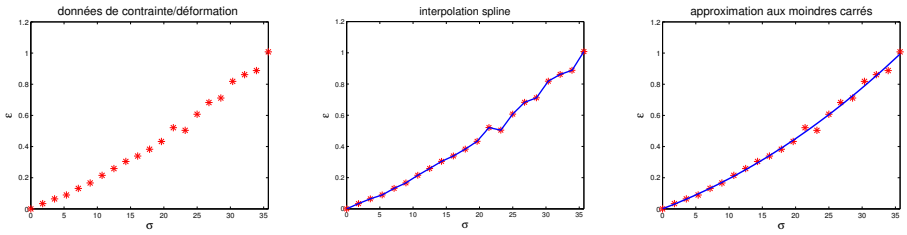


FIGURE 2.3 – Gauche : mesures de déformation obtenues par une jauge de déformation. Centre : interpolation des données par spline. Droite : approximation polynomiale de degré 3 aux moindres carrés.

2.1 Interpolation polynomiale des données

On considère d'abord le cas où les mesures ne sont pas affectées par erreur de mesure. On suppose d'avoir à disposition $n+1$ données (x_i, y_i), $i = 1, \ldots, n+1$ et qu'il existe une fonction f (inconnue) qui lie les variables

y et x. Notre modèle est donc

$$y_i = f(x_i), \qquad i = 1, \ldots, n+1.$$

La fonction f étant inconnue, on essaye de la découvrir à partir des mesures. Une première idée est celle de *chercher un polynôme de degré n qui interpole les données.*

Définition 2.1. ***Polynôme interpolant*** *les donnée (x_i, y_i), $i = 1, \ldots, n+1$: est un polynôme de degré n, qu'on appelle $p_n(x)$, tel que*

$$y_i = p_n(x_i), \qquad i = 1, \ldots, n+1.$$

Un résultat d'algèbre nous garantit l'existence d'un unique polynôme. En fait, on peut trouver une seule droite (polynôme de degré 1) qui passe par deux points (x_1, y_1) et (x_2, y_2). De même, on peut trouver une seule parabole (polynôme de degré 2) qui passe par trois points (x_1, y_1), (x_2, y_2) et (x_3, y_3). Ce résultat se généralise : on peut trouver un seul polynôme de degré n qui passe par $n+1$ points $(x_1, y_1), \ldots, (x_{n+1}, y_{n+1})$.

Construction par la matrice de Vandermonde

Un polynôme quelconque de degré n a la forme suivante

$$p_n(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n.$$

Une première idée pour trouver le polynôme interpolant les données (x_i, y_i), $i = 1, \ldots, n+1$ est de déterminer les $n+1$ coefficients inconnus $a_0, \ldots, a_n$ en imposant les $n+1$ conditions d'interpolation $y_i = p_n(x_i)$:

$$\begin{cases} p_n(x_1) = a_0 + a_1 x_1 + a_2 x_1^2 + \ldots + a_n x_1^n = y_1 \\ p_n(x_2) = a_0 + a_1 x_2 + a_2 x_2^2 + \ldots + a_n x_2^n = y_2 \\ \ldots \\ p_n(x_{n+1}) = a_0 + a_1 x_{n+1} + a_2 x_{n+1}^2 + \ldots + a_n x_{n+1}^n = y_{n+1}. \end{cases} \tag{2.1}$$

Ceci est un système linéaire de $n+1$ équations dans les $n+1$ inconnues $a_0, \ldots, a_n$. Il peut être écrit sous forme matricielle

$$\underbrace{\begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^n \\ 1 & x_2 & x_2^2 & \cdots & x_2^n \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_{n+1} & x_{n+1}^2 & \cdots & x_{n+1}^n \end{bmatrix}}_{V} \underbrace{\begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix}}_{\mathbf{a}} = \underbrace{\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_{n+1} \end{bmatrix}}_{\mathbf{y}}. \tag{2.2}$$

La matrice V est appelée *matrice de Vandermonde*. Grâce au fait qu'il existe un seul polynôme interpolant de degré n, le système linéaire (2.2) a une seule

solution et la matrice V est toujours inversible (si les x_i sont distincts) et $\det(V) \neq 0$.

Toutefois, la matrice V est un exemple typique de matrice *mal conditionnée* (on verra ça plus en détail dans le chapitre 4) et la résolution numérique de (2.2) est difficile car de toutes petites erreurs sur le terme de droite (par exemple les erreurs d'arrondis dues à la représentation à virgule flottante) sont amplifiées dans la solution et peuvent donner des solutions de très mauvaise qualité. Il faut donc faire attention à utiliser cette méthode pour des n grands (de l'ordre de quelques dizaines).

Exemple 2.3 (Viscosité dynamique de l'eau)**.** *On reprend l'exemple sur la viscosité de l'eau et on souhaite trouver le polynôme de degré 9 qui interpole les 10 mesures en Tableau 2.1. La méthode de Vandermonde est implémentée en* ***Python*** *dans la commande* `p_coef=numpy.polyfit(x,y,n)` *où* `x` *est le vecteur des nœuds (température dans notre cas),* `y` *le vecteur des mesures (viscosité de l'eau) et* `p_coef` *le vecteur des coefficients du polynôme interpolant de degré* `n` *en ordre décroissant* `p_coef`$=(a_n, a_{n-1}, \ldots, a_0)$.

```
import numpy as np
T=[10, 20, 30, 40, 50, 60, 70, 80, 90, 100];
nu=[1.308, 1.002, 0.7978, 0.6531, 0.5471, 0.4658,
    0.4044, 0.3550, 0.3150, 0.2822];
p_coef=np.polyfit(T,nu,9)
print("p_coef:", p_coef)

# OUTPUT
# p_coef: [-3.76432981e-16  1.86061508e-13 -3.93806217e-11  4.66770833e-09
#  -3.40309606e-07  1.58054062e-05 -4.71143269e-04  9.11859623e-03
#  -1.30803159e-01  2.04700000e+00]
```

Pour visualiser le polynôme trouvé, on peut utiliser la commande `np.polyval`

```
import matplotlib.pyplot as plt
T_fine=np.arange(10,100.1,0.1)  # fine grid of nodes for visualization
p=np.polyval(p_coef,T_fine)     # evaluation of the polynomial in T_fine
plt.plot(T_fine,p,'b')
plt.plot(T,nu,'r*')
```

La Figure 2.4 montre le graphe du polynôme interpolant. On voit que dans ce cas le polynôme capture bien le comportement des données.

Construction par la base de Lagrange

On présente ici une autre méthode pour construire le polynôme interpolant. Elle est beaucoup plus pratique que la méthode de Vandermonde si on doit calculer le polynôme interpolant "à la main". De plus, elle est aussi plus stable que la méthode de Vandermonde pour des n élevés.

On commence par définir les *polynômes de base de Lagrange*

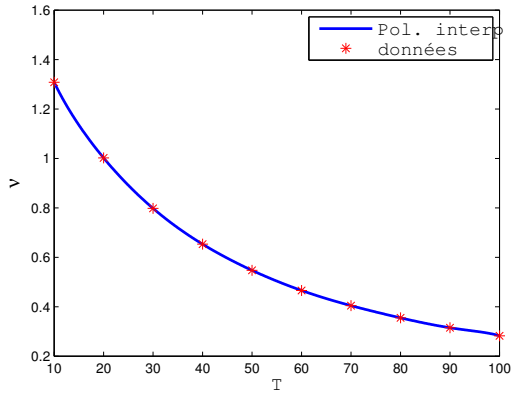


FIGURE 2.4 – Données de viscosité de l'eau en fonction de la température (Tableau 2.1) et polynôme interpolant de degré 9.

Définition 2.2. ***Polynômes de base de Lagrange*** *associés aux nœuds* $x_1, x_2, \dots x_{n+1}$ *: sont les* $n+1$ *polynômes*

$$\phi_i(x) = \frac{(x-x_1)(x-x_2)\dots(x-x_{i-1})(x-x_{i+1})\dots(x-x_{n+1})}{(x_i-x_1)(x_i-x_2)\dots(x_i-x_{i-1})(x_i-x_{i+1})\dots(x_i-x_{n+1})} = \prod_{\substack{j=1\\ j\neq i}}^{n+1} \frac{(x-x_j)}{(x_i-x_j)}$$

pour $i = 1, \dots, n+1$.

Ces polynômes ont les propriétés suivantes :

— chaque ϕ_i est un polynôme de degré n,

— $\phi_i(x_k) = 0$ si $k \neq i$ et $\phi_i(x_i) = 1$.

Autrement dit, le polynôme $\phi_i(x)$ s'annule en tous les nœuds x_k sauf au nœud x_i où il vaut 1. La Figure 2.5 montre les polynômes ϕ_1, ϕ_3 et ϕ_5 associés aux nœuds $x_1 = 0, x_2 = 0.2, x_3 = 0.4, \dots, x_6 = 1$.

Grâce aux polynômes de base de Lagrange, on peut construire le polynôme p_n interpolant les données (x_i, y_i), $i = 1, \dots, n+1$ de la façon suivante :

Proposition 2.1. *le polynôme* p_n *interpolant les données* (x_i, y_i), $i = 1, \dots, n+1$ *est donné par*

$$p_n(x) = \sum_{i=1}^{n+1} y_i \phi_i(x). \tag{2.3}$$

En fait, p_n est clairement un polynôme de degré n car il est une combinaison linéaire des polynômes ϕ_i qui sont tous des polynômes de degré n. De plus, on a

$$p_n(x_k) = y_1 \underbrace{\phi_1(x_k)}_{=0} + y_2 \underbrace{\phi_2(x_k)}_{=0} + \dots + y_k \underbrace{\phi_k(x_k)}_{=1} + \dots + y_{n+1} \underbrace{\phi_{n+1}(x_k)}_{=0} = y_k$$

et donc p_n est bien le polynôme interpolant.

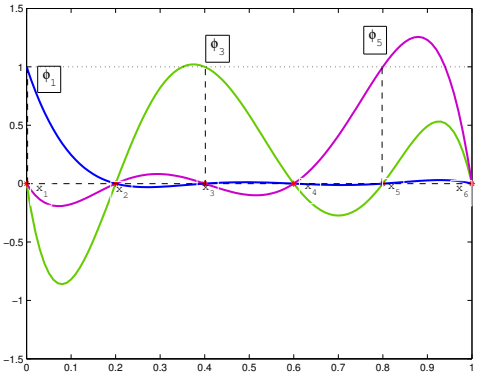


FIGURE 2.5 – Polynômes de base de Lagrange ϕ_1, ϕ_3, ϕ_5 associés aux nœuds $x_1 = 0, x_2 = 0.2, x_3 = 0.4, \ldots, x_6 = 1$.

Exemple 2.4. *On veut construire le polynôme interpolant les données* $(x_1 = 0, y_1 = 1)$ *et* $(x_2 = 1, y_2 = 2)$. *On commence par construire la base de Lagrange*

$$\phi_1(x) = \frac{(x - x_2)}{(x_1 - x_2)} = \frac{x - 1}{-1} = 1 - x,$$
$$\phi_2(x) = \frac{(x - x_1)}{(x_2 - x_1)} = \frac{x - 0}{1} = x.$$

Finalement, le polynôme interpolant est

$$p_1(x) = y_1\phi_1(x) + y_2\phi_2(x) = 1(1 - x) + 2x = x + 1$$

comme on peut le vérifier facilement.

Analyse d'erreur

On se pose maintenant la question si le polynôme interpolant p_n est une bonne approximation de la "vraie" fonction f d'où les mesures y_i proviennent.

Cette question est mal posée car il y a une infinité des fonctions f qui prennent les mêmes valeurs y_i aux nœuds x_i et qui peuvent être arbitrairement loin de p_n dans tous les autres points x.

On peut quand même se poser une autre question : si on avait de plus en plus de mesures, est-ce qu'on aurait une approximation p_n de plus en plus précise ? Et encore, si le nombre de mesures n tend vers l'infini, est-ce que le polynôme interpolant p_n tend vers la "vraie" fonction f ?

Il s'agit clairement d'une question très théorique mais non moins importante. Si jamais p_n ne tend pas vers f lorsque $n \to \infty$, alors notre méthode

de reconstruction n'est pas bonne. Si, par contre, $p_n \to f$ lorsque $n \to \infty$ il est aussi important de savoir à quelle vitesse p_n converge vers f. Le plus vite p_n converge vers f le moins de mesures on aura besoin pour pouvoir reconstruire f avec précision.

L'exemple suivant montre que l'interpolation polynomiale p_n ne converge pas toujours vers la vraie fonction f.

Exemple 2.5. *On considère deux fonctions*

$$f_1(x) = \sin(5x), \qquad f_2(x) = \frac{1}{1+(5x)^2}$$

définies sur l'intervalle $[-1, 1]$. *On prend un nombre croissant de mesures*

$$y_i = f_1(x_i), \qquad z_i = f_2(x_i), \quad i = 1, \ldots, n+1$$
$$\{x_i\}_{i=1}^{n+1} \text{ nœuds équirépartis dans l'intervalle } [-1, 1]$$

et on calcule le polynôme $p_n^{(1)}$ *interpolant les données* (x_i, y_i) *et* $p_n^{(2)}$ *interpolant les données* (x_i, z_i).

La Figure 2.6 montre à gauche le polynôme $p_n^{(1)}$ *et à droite le polynôme* $p_n^{(2)}$, *pour* $n = 4, 8, 16$. *On voit que dans le cas de la fonction* $f_1(x) = \sin(5x)$

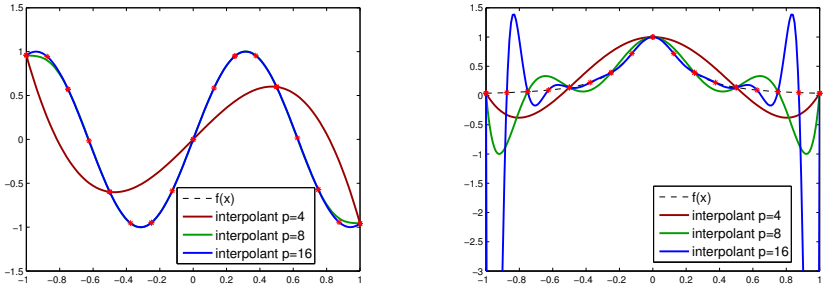


FIGURE 2.6 – Polynôme interpolant $p_n^{(1)}$ (gauche) et $p_n^{(2)}$ (droite) pour $n = 4, 8, 16$.

le polynôme interpolant de degré 16 est presque superposé à la fonction exacte et l'interpolation polynomiale a bien l'air de converger vers la fonction exacte lorsqu'on augmente le nombre de mesures. Par contre, dans le cas de la fonction $f_2(x) = 1/(1+(5x)^2)$, *le polynôme interpolant de degré 16 donne en fait une bonne approximation de la vraie fonction au centre de l'intervalle mais une approximation très mauvaise aux extrémités, pire que le polynôme de degré 8. Si on augmente encore le degré polynomial, on s'aperçoit que le polynôme interpolant* diverge *de la fonction* f_2 *pour* $|x|$ *plus grand de environs* 0.6. *Ce phénomène est connu comme* phénomène de Runge.

On a à disposition une estimation de l'erreur commise par l'interpolation polynomiale de degré n :

Theorème 2.2. *Soit $f : [a,b] \to \mathbb{R}$ une fonction de classe C^{n+1} et $a = x_1 < x_2 < \ldots < x_{n+1} = b$ des nœuds équirépartis dans $[a,b]$. Le polynôme p_n de degré n interpolant les données $(x_i, f(x_i))$, $i = 1, \ldots, n+1$ satisfait l'estimation d'erreur*

$$\max_{x\in[a,b]} |f(x) - p_n(x)| \leq \frac{1}{4(n+1)} \left(\frac{b-a}{n}\right)^{n+1} \max_{x\in[a,b]} |f^{(n+1)}(x)|. \qquad (2.4)$$

Ce résultat est compliqué à démontrer. On renvoie le lecteur intéressé au livre [Quarteroni, Sacco, Saleri, "Méthodes Numériques", Springer].

Il est toutefois intéressant car il montre que si $\max_{x\in[a,b]} |f^{(n+1)}(x)|$ croit plus lentement que $\frac{4(n+1)n^{n+1}}{(b-a)^{n+1}}$ lorsque n tend vers l'infini, alors la convergence est garantie.

Ceci est le cas de la fonction $f_1(x) = \sin(5x)$ de l'exemple 2.5 où les dérivées croissent de façon exponentielle

$$\max_{x\in[-1,1]} |f_1^{(n+1)}(x)| \leq 5^{n+1}$$

et

$$\max_{x\in[a,b]} |f(x) - p_n(x)| \leq \frac{2^{n+1}}{4(n+1)n^{n+1}} \max_{x\in[-1,1]} |f_1^{n+1}(x)| \leq \frac{10^{n+1}}{4(n+1)n^{n+1}} \xrightarrow{n\to\infty} 0.$$

Par contre, on peut montrer que pour la fonction f_2 de l'exemple 2.5, les dérivées croissent comme $\max_{x\in[-1,1]} |f_2^{(n)}(x)| \sim n!5^n$ et la convergence n'est pas garantie.

Il est important de mentionner que le problème de la non convergence de l'interpolation polynomiale, même pour des fonctions très régulières, est lié au fait qu'on a utilisé des nœuds équirépartis dans l'intervalle $[a,b]$. Il existe d'autres choix de nœuds qui sont plus denses vers les extrémités de l'intervalle et qui permettent d'obtenir des interpolations toujours convergentes si la fonction f est au moins dérivable. Un tel choix de nœuds est donné par les nœuds de Clenshaw-Curtis.

Définition 2.3. *On appelle **nœuds de Clenshaw-Curtis** la suite de nœuds sur l'intervalle $[a,b]$ donnée par*

$$x_i = \frac{a+b}{2} - \frac{b-a}{2} \cos\left(\frac{\pi(i-1)}{n}\right), \qquad i = 1, \ldots, n+1.$$

Les nœuds de Clenshaw-Curtis sont donnés par la projection sur l'axe horizontale de nœuds équidistribués sur le demi-cercle de centre $(a+b)/2$ et rayon $(b-a)/2$, voir Figure 2.7. D'autres choix de nœuds sont aussi possibles. On mentionne par exemple les nœuds de Chebyshev ou les nœuds de Gauss-Legendre. Néanmoins, dans le contexte de l'interpolation de données, on n'a très souvent pas vraiment la possibilité de choisir la position des nœuds x_i, qui sont normalement distribués uniformément dans l'intervalle.

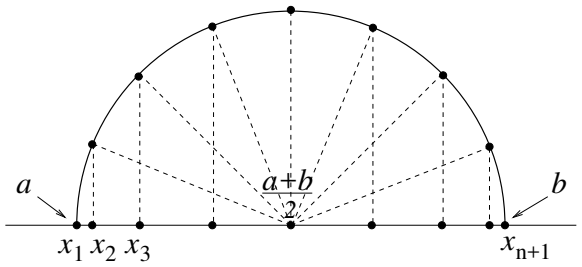


FIGURE 2.7 – Nœuds de Clenshaw-Curtis

Stabilité de l'interpolation polynomiale

On considère maintenant l'effet des erreurs de mesure sur les données. Supposons qu'on veuille trouver le polynôme $p_n(x)$ qui interpole les données (x_i, y_i), $i = 1, \ldots, n+1$, où $a = x_1 < x_2 < \ldots < x_{n+1} = b$ et $y_i = f(x_i)$ proviennent de l'évaluation d'une fonction continue qu'on souhaite identifier.

Toutefois, notre instrument de mesure introduit une erreur de mesure, donc les données qu'on a à disposition sont $\tilde{y}_i = f(x_i) + \epsilon_i$ où ϵ_i représente l'erreur de mesure que l'on suppose petite : $|\epsilon_i| \leq \epsilon, \ \ \forall i = 1, \ldots, n+1$.

Si $\tilde{p}_n$ dénote le polynôme interpolant les données perturbées, on se pose la question d'estimer la distance entre le "vrai" polynôme interpolant p_n et le polynôme $\tilde{p}_n$ qu'on calcul. Utilisons le développement sur la base de Lagrange :

$$p_n(x) = \sum_{i=1}^{n+1} y_i \phi_i(x), \qquad \tilde{p}_n(x) = \sum_{i=1}^{n+1} (y_i + \epsilon_i)\phi_i(x),$$

d'où, en prenant la différence on a pour tout $x \in [a, b]$

$$|\tilde{p}_n(x) - p_n(x)| = \left| \sum_{i=1}^{n+1} \phi_i(x)\epsilon_i \right| \leq \sum_{i=1}^{n+1} |\phi_i(x)||\epsilon_i| \leq \left(\sum_{i=1}^{n+1} |\phi_i(x)| \right) \epsilon.$$

Définition 2.4. *On appelle* ***constante de Lebesgue*** *associée aux noeuds* $(x_1, \ldots, x_{n+1}) \in [a, b]$ *la quantité*

$$L_n = \sup_{x \in [a,b]} \sum_{i=1}^{n+1} |\phi_i(x)|.$$

On a donc l'estimation suivante :

$$\max_{x \in [a,b]} |\tilde{p}(x) - p(x)| \leq L_n \, \epsilon, \qquad \epsilon = \max_{i=1,\ldots,n+1} |\epsilon_i|. \tag{2.5}$$

Le résultat (2.5) est un résultat de *stabilité* : si la constante de Lebesgue L_n est petite, des petites erreurs de mesure $|\epsilon_i| \leq \epsilon$ induisent des petites perturbations sur le polynôme interpolant. On dit dans ce cas que l'interpolation polynomiale sur les nœuds $x_1, \ldots, x_{n+1}$ est *stable* ou *bien conditionnée*.

Si, au contraire, la constante de Lebesgue est très grande, alors l'interpolation polynomiale sur les nœuds $x_1, \ldots, x_{n+1}$ est *mal conditionnée*, c'est-à-dire, elle a des mauvaises propriétés de stabilité.

Pour l'interpolation polynomiale sur des nœuds équirépartis ou de Clenshaw-Curtis, on peut montrer les résultats suivants :

$$\textit{nœuds équirépartis} \qquad L_n \sim \frac{2^{n+1}}{en\log n}, \quad n \to \infty, \tag{2.6}$$

$$\textit{nœuds de Clenshaw-Curtis} \qquad L_n \sim \frac{2}{\pi}\log n, \quad n \to \infty. \tag{2.7}$$

On voit donc que l'interpolation polynomiale sur des nœuds de Clenshaw-Curtis est relativement bien conditionnée (stable) car la constante de Lebesgue croît seulement de façon logarithmique en n. Par contre, l'interpolation polynomiale sur des nœuds équirépartis est très mal conditionnée car la constante de Lebesgue croît de façon exponentielle en n et donc des petites perturbations sur les données sont fortement amplifiées.

Exemple 2.6. *Considérons la fonction $f(x) = \sin(5x)$ de l'exemple 2.5 pour laquelle on a vu que l'interpolation polynomiale donne des bons résultats.*

Soient $\{x_i\}_{i=1}^{n+1}$ des nœuds distincts sur l'intervalle $[-1, 1]$ et $y_i = f(x_i)$ les évaluations exactes de la fonction. Considérons aussi des évaluations perturbées

$$\tilde{y}_i = f(x_i) + \eta_i$$

où $\eta_i \in [-10^{-2}, 10^{-2}]$ sont des erreurs aléatoires d'amplitude $|\eta_i| \leq 10^{-2}$, et calculons en `Python` *le polynôme p_n interpolant les données (x_i, y_i) ainsi que le polynôme $\tilde{p}_n$ interpolant les données perturbées $(x_i, \tilde{y}_i)$. La figure 2.8 montre à gauche les deux polynômes interpolants sur des nœuds équirépartis et à droite les deux polynômes interpolants sur des nœuds de Clenshaw-Curtis.*

On voit bien que dans le premier cas, la distance maximale entre les deux polynômes est de l'ordre de 1, donc 100 fois plus grande que les perturbations η_i. Par contre, dans le deuxième cas, la distance entre les deux polynômes est très petite et, en faite, on n'arrive pas à distinguer les deux polynômes sur la figure.

2.2 Interpolation linéaire par morceaux

On a vu dans la section précédente que l'interpolation polynomiale de degré n n'est pas toujours convergente et peut être très mal conditionnée. On considère ici une autre forme d'interpolation de données : l'*interpolation linéaire par morceaux*.

Soient $a = x_1 < x_2 < \ldots < x_{n+1} = b$ des nœuds équirépartis dans l'intervalle $[a, b]$ et $y_i = f(x_i)$ les mesures correspondantes qui proviennent

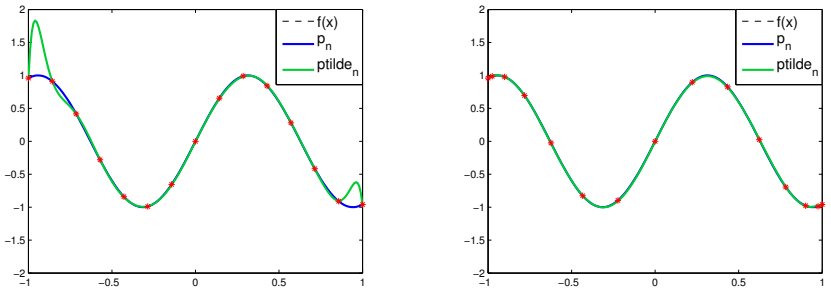


FIGURE 2.8 – Exemple 2.6 : polynômes p_n interpolant les données (x_i, y_i) et $\tilde{p}_n$ interpolant les données $(x_i, \tilde{y}_i)$ perturbées par des erreurs aléatoires $|\eta_i| \leq 10^{-2}$. Gauche : nœuds équirépartis ; droite : nœuds de Clenshaw-Curtis

de l'évaluation d'une fonction f inconnue. On note I_i l'intervalle $[x_i, x_{i+1}]$ de longueur $h = (b-a)/n$.

Définition 2.5. ***Polynôme linéaire par morceaux*** *interpolant les données (x_i, y_i) : est une fonction $p_{1,h}$ telle que*

- *$p_{1,h}$ est un polynôme de degré 1 dans chaque intervalle I_i, $i = 1, \ldots, n$,*
- *$p_{1,h}(x_i) = y_i$, $i = 1, \ldots, n+1$.*

Donc dans chaque intervalle I_i, $p_{1,h}$ est un polynôme de degré 1 qui interpole les donnée (x_i, y_i) et (x_{i+1}, y_{i+1}). Son expression peut être trouvée en utilisant la méthode de la base de Lagrange

$$p_{1,h}(x) = y_i \frac{x - x_{i+1}}{x_i - x_{i+1}} + y_{i+1} \frac{x - x_i}{x_{i+1} - x_i}, \quad \forall x \in I_i.$$

En `Python` on peut le calculer en utilisant la commande `p1h=numpy.interp(x,y,x_fine` où `x` est le vecteur des nœuds, `y` le vecteur des mesures, `x_fine` un point ou un vecteur des points où on souhaite évaluer le polynôme linéaire par morceaux et `p1h` le vecteur des évaluations $p_{1,h}$(`x_fine`).

Exemple 2.7. *On reprend l'exemple de la fonction $f_2(x) = 1/(1+(5x)^2)$ déjà considérée dans l'exemple 2.5, pour lequel l'interpolation polynomiale de degré n n'est pas convergente. On calcule en* ***Python*** *l'interpolation linéaire par morceaux sur 9 nœuds équirépartis dans l'intervalle $[-1, 1]$:*

```
import numpy as np
import matplotlib.pyplot as plt
f = lambda x : 1./(1+(5*x)**2)
n=8
x=np.linspace(-1,1,n+1)       # interpolation nodes
y=f(x);                       # measures
xfine=np.linspace(-1,1,201)   # fine grid
```

```
p1h=np.interp(xfine,x,y)      # evaluation of p1h in the fine grid
plt.plot(xfine,p1h,'b')
plt.plot(xfine,f(xfine),'k--')
plt.legend(["p1h, n=8","f(x)","data"])
```

La Figure 2.9 montre le résultat obtenu. On voit que $p_{1,h}$ ne présente pas

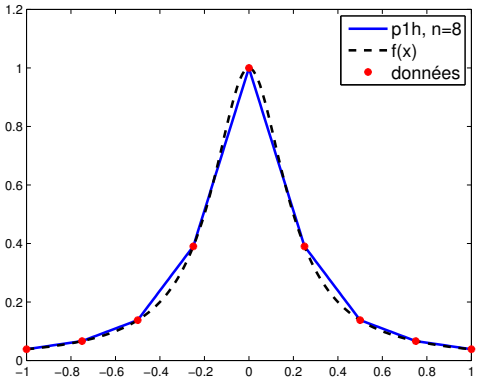


FIGURE 2.9 – Interpolation linéaire par morceaux de la fonction $f(x) = 1/(1 + (5x)^2)$ sur 9 nœuds équirépartis dans l'intervalle $[-1, 1]$.

d'oscillations et donne globalement une bonne approximation de la fonction f_2.

Analyse d'erreur

L'analyse d'erreur pour l'interpolation linéaire par morceaux est une conséquence du résultat du Théorème 2.2. En fait, dans chaque intervalle I_i, on est en train de calculer une interpolation linéaire des données (x_i, y_i) et (x_{i+1}, y_{i+1}). On peut donc appliquer le résultat du Théorème 2.2 avec $n = 1$ et $b - a = x_{i+1} - x_i = h$:

$$\max_{x\in I_i} |f(x) - p_{1,h}(x)| \leq \frac{h^2}{8} \max_{x\in I_i} |f^{(2)}(x)|$$

et donc

$$\max_{x\in[a,b]} |f(x) - p_{1,h}(x)| = \max_{i=1,\ldots,n} \max_{x\in I_i} |f(x) - p_{1,h}(x)| \leq \frac{h^2}{8} \max_{x\in[a,b]} |f^{(2)}(x)|.$$

Le Théorème suivant récapitule le résultat :

Theorème 2.3. *Soit $f : [a, b] \to \mathbb{R}$ une fonction de classe C^2 et $a = x_1 < x_2 < \ldots < x_{n+1} = b$ des nœuds équirépartis dans $[a, b]$. Soit $h = (b-a)/n$ la longueur de chaque intervalle $I_i = [x_i, x_{i+1}]$, $i = 1, \ldots, n+1$. Le*

polynôme linéaire par morceaux interpolant les données $(x_i, f(x_i))$ *satisfait l'estimation d'erreur*

$$\max_{x\in[a,b]} |f(x) - p_{1,h}(x)| \leq Ch^2 \max_{x\in[a,b]} |f^{(2)}(x)| \tag{2.8}$$

avec $C = 1/8$.

On dit dans ce cas que la convergence est *quadratique* ou *d'ordre 2*. Plus généralement

Définition 2.6. *Soit* p_h *une approximation par morceaux de* $f : [a, b] \to \mathbb{R}$ *sur des sous-intervalles de longueur* h. *On dit que la convergence est d'ordre* q *s'il existe une constance* $C > 0$ *indépendante de* h *telle que*

$$\max_{x\in[a,b]} |f(x) - p_h(x)| \leq Ch^q.$$

Il y a deux choses a retenir dans l'estimation (2.8). La première est que l'estimation est vraie à condition que la dérivée seconde de la fonction soit continue.

La deuxième est que l'erreur est proportionnelle à la longueur h de chaque sous-intervalle au carré (convergence d'ordre 2). Ceci implique que si on double le nombre de points (et donc divise par 2 le longueur de chaque sous-intervalle), l'erreur sera divisée grosso modo par 4.

On peut vérifier ça en `Python` , toujours sur la fonction $f_2(x) = 1/(1 + (5x)^2)$ de l'exemple 2.5. On prend $n = 16, 32, 64, 128, 256$ sous-intervalles et on estime l'erreur de l'interpolation linéaire par morceaux pour chaque n.

```
import numpy as np
f = lambda x : 1./(1+(5*x)**2)
xfine=np.linspace(-1,1,201)         # fine grid
err=np.array([])
h=np.array([])
for n in 2**np.arange(4,9):
    h=np.append(h,2./n)
    x=np.linspace(-1,1,n+1)    # interpolation nodes
    y=f(x)                     # measures
    p1h=np.interp(xfine,x,y)   # evaluation of p1h in the fine grid
    err=np.append(err, max(abs(f(xfine)-p1h)))
print("h: ", h)
print("err: ",err)
```

On voit que l'erreur est effectivement divisée par un facteur 4 (approximativement) lorsqu'on double le nombre de sous-intervalles.

Pour mieux apprécier ce comportement, on peut tracer le graphe de l'erreur d'interpolation en fonction de h. On note que l'erreur est de la forme

$$err_h = \max_{x\in[a,b]} |f(x) - p_{1,h}(x)| \sim Ch^2$$

où $C = \max_{x\in[a,b]} |f^{(2)}(x)|/8$. Si on prend le logarithme en base 10 (ou n'importe quelle autre base) des deux cotés, on a

$$\log_{10}(err_h) \sim \log_{10}(Ch^2) = \log_{10}(C) + 2\log_{10}(h).$$

Donc $\log_{10}(err_h)$ *est une fonction linéaire de* $\log_{10}(h)$ avec pente 2 qui correspond bien à l'ordre de l'approximation. On peut donc tracer sur un graphe la quantité $log_{10}(err_h)$ en fonction de la quantité $\log_{10}(h)$ et vérifier que la pente est effectivement égale à 2. Ceci peut être fait facilement en `Python` par la commande graphique `loglog(x,y)` de `matplotlib.pyplot` qui visualise sur un graphe $\log_{10}(y)$ en fonction de $\log_{10}(x)$.

```
import matplotlib.pyplot as plt
plt.loglog(h,err,'b')
plt.loglog(h,h,'k--')
plt.loglog(h,h**2,'k-.')
plt.grid(True)
plt.legend(["err_h","slope 1","slope 2"])
```

La Figure 2.10 montre le résultat obtenu. Sur le même graphe on a tracé

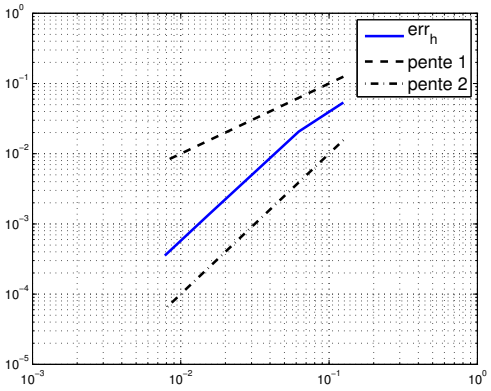


FIGURE 2.10 – Graphe en échelle logarithmique de l'erreur de l'interpolation linéaire par morceaux de la fonction $f(x) = 1/(1+(5x)^2)$ sur n sous-intervalles en fonction de la longueur de chaque sous-intervalle $h = 2/n$.

aussi les courbes $y = h$ et $y = h^2$. Notez qu'en échelle logarithmique, ces deux courbes correspondent à deux droites de pente 1 et 2 respectivement.

Stabilité de l'interpolation linéaire par morceaux

On considère à nouveau l'effet des erreurs de mesure sur les données. Les données qu'on a à disposition sont $\tilde{y}_i = f(x_i) + \epsilon_i$ où ϵ_i représente l'erreur de mesure que l'on suppose petite : $|\epsilon_i| \leq \epsilon, \quad \forall i = 1, \ldots, n+1$. Soient $p_{1,h}$ le polynôme linéaire par morceaux interpolant les données non-perturbées

(x_i, y_i), $i = 1, ..., n+1$ et $\tilde{p}_{1,h}$ celui interpolant les données perturbées $(x_i, \tilde{y}_i)$, $i = 1, ..., n+1$. Sur chaque sous-intervalle $I_i = [x_i, x_{i+1}]$ on a donc

$$p_{1,h}(x) = y_i \frac{x - x_{i+1}}{x_i - x_{i+1}} + y_{i+1} \frac{x - x_i}{x_{i+1} - x_i}$$

et

$$\tilde{p}_{1,h}(x) = (y_i + \epsilon_i) \frac{x - x_{i+1}}{x_i - x_{i+1}} + (y_{i+1} + \epsilon_{i+1}) \frac{x - x_i}{x_{i+1} - x_i},$$

d'où, en prenant la différence, on a pour tout $x \in I_i$

$$|\tilde{p}_{1,h}(x) - p_{1,h}(x)| \leq |\epsilon_i| \left| \frac{x - x_{i+1}}{x_i - x_{i+1}} \right| + |\epsilon_{i+1}| \left| \frac{x - x_i}{x_{i+1} - x_i} \right| \leq \epsilon.$$

On a utilisé le fait que $|\epsilon_i| \leq \epsilon$ pour tout $i = 1, ..., n+1$ ainsi que la relation

$$\left| \frac{x - x_{i+1}}{x_i - x_{i+1}} \right| + \left| \frac{x - x_i}{x_{i+1} - x_i} \right| = \frac{x_{i+1} - x}{x_{i+1} - x_i} + \frac{x - x_i}{x_{i+1} - x_i} = 1 \quad \forall x \in I_i.$$

En prenant finalement le maximum sur tous les sous-intervalles, on obtient

$$\max_{x \in [a,b]} |\tilde{p}_{1,h}(x) - p_{1,h}(x)| = \max_{i=1,...,n} \max_{x \in I_i} |\tilde{p}_{1,h}(x) - p_{1,h}(x)| \leq \epsilon. \tag{2.9}$$

On peut donc conclure que l'interpolation linéaire par morceaux est stable : des petites erreurs de mesure induisent des petites perturbations sur l'interpolant linéaire par morceaux.

Exemple 2.8. *Comme dans l'exemple 2.6, on considère la fonction* $f(x) = \sin(5x)$ *de l'exemple 2.5. Soient* $\{x_i\}_{i=1}^{n+1}$ *des nœuds équirépartis sur l'intervalle* $[-1, 1]$, $y_i = f(x_i)$ *les évaluations exactes de la fonction et*

$$\tilde{y}_i = f(x_i) + \eta_i$$

des évaluations perturbées où $\eta_i \in [-10^{-2}, 10^{-2}]$ *sont des erreurs aléatoires d'amplitude* $|\eta_i| \leq 10^{-2}$. *On calcule en* `Python` *le polynôme* $p_{1,h}$ *linéaire par morceaux interpolant les données* (x_i, y_i) *ainsi que le polynôme* $\tilde{p}_{1,h}$ *interpolant les données perturbées* $(x_i, \tilde{y}_i)$ *et on trace les résultats sur la figure 2.11. Comme pour l'interpolation polynomiale sur noeuds de Clenshaw-Curtis, on voit que la distance entre les deux polynômes* $p_{1,h}$ *et* $\tilde{p}_{1,h}$ *est très petite, les deux polynômes étant même indistinguables. Plus précisément, on peut vérifier que la distance maximale* $\max_{x \in [a,b]} |\tilde{p}_{1,h} - p_{1,h}|$ *entre les deux polnyômes est bien inférieure à* 10^{-2}, *l'amplitude maximale des perturbations, en accord avec l'estimation (2.9).*

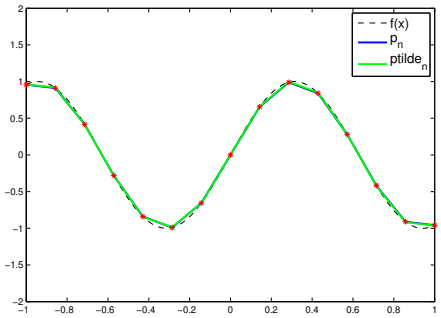


FIGURE 2.11 – Exemple 2.8 : polynôme $p_{1,h}$ interpolant les données (x_i, y_i) et $\tilde{p}_{1,h}$ interpolant les données $(x_i, \tilde{y}_i)$ perturbées par des erreurs aléatoires $|\eta_i| \leq 10^{-2}$.

2.3 Interpolation spline

L'interpolation linéaire par morceaux est toujours convergente lorsque $n \to \infty$ (ou de façon équivalente $h \to 0$). De plus, le polynôme interpolant linéaire par morceaux ne présente pas d'oscillations, contrairement au polynôme interpolant de degré n. Néanmoins, il a deux limitations importantes :

- la convergence est seulement d'ordre 2,
- le polynôme linéaire par morceaux est globalement seulement continu. Il ne peut pas être utilisé pour approcher les dérivées de la fonction f.

Il y a en fait plusieurs situations, notamment dans les applications graphiques comme CAO (conception assistée par ordinateur – en anglais CAD, Computer Aided Design) où on aimerait avoir des interpolations de données plus "lisses" par exemple avec dérivée seconde continue.

On considère ici une troisième méthode d'interpolation de donnée appelée *interpolation spline*. Voici la définition d'une spline interpolant :

Définition 2.7. *Soient (x_i, y_i), $i = 1, \ldots, n+1$ des données. On suppose que les x_i sont ordonnés $a = x_1 < x_2 < \ldots < x_{n+1} = b$ et on appelle I_i le sous-intervalle $[x_i, x_{i+1}]$. Une* ***spline cubique interpolant*** *est une fonction $s_{3,h}$ telle que*

- *$s_{3,h} \in C^2([a, b])$,*
- *$s_{3,h}$ est un polynôme de degré 3 dans chaque intervalle I_i,*
- *$s_{3,h}$ interpole les données : $s_{3,h}(x_i) = y_i$, $i = 1, \ldots, n+1$.*

Le fait que $s_{3,h}$ soit de classe C^2 implique que la fonction $s_{3,h}$ ainsi que la dérivée première et seconde sont des fonctions continues. Clairement, dans chaque intervalle I_i, $s_{3,h}$ est un polynôme donc toutes ses dérivées

sont continues. La condition $s_{3,h} \in C^2([a,b])$ revient donc à demander que la fonction et les premières deux dérivées soient *continues aux nœuds* x_i, $i = 2, \ldots, n$:

$$s_{3,h}(x_i^-) = s_{3,h}(x_i^+), \quad i = 2, \ldots, n \tag{2.10}$$
$$s'_{3,h}(x_i^-) = s'_{3,h}(x_i^+), \quad i = 2, \ldots, n \tag{2.11}$$
$$s''_{3,h}(x_i^-) = s''_{3,h}(x_i^+), \quad i = 2, \ldots, n. \tag{2.12}$$

Or, sur chaque sous-intervalle I_i, la spline est un polynôme de degré 3 et peut être écrite sous forme générale comme

$$s_{3,h}\big|_{I_i} = a_i + b_i x + c_i x^2 + d_i x^3, \qquad i = 1, \ldots, n,$$

dont les coefficients (a_i, b_i, c_i, d_i) sont à déterminer. On voit donc qu'il y a $4n$ coefficients inconnus (4 sur chaque sous-intervalle). De l'autre coté, on a $3(n-1)$ conditions de continuité aux nœuds internes et $n+1$ conditions d'interpolation $s_{3,h}(x_i) = y_i$.

inconnues	4n
conditions de continuité	3(n-1)
conditions d'interpolation	(n+1)
degrés de liberté	4n - 3(n-1) - (n+1) = 2

Il en suit qu'il reste encore 2 degrés de liberté, c'est-à-dire qu'il faut encore imposer 2 conditions pour pouvoir déterminer une spline de manière unique. Il y a plusieurs choix sur les 2 conditions supplémentaires. Voici les 3 alternatives les plus utilisées :

- *spline naturelle* : on impose $s''_{3,h}(x_1) = 0$, $s''_{3,h}(x_{n+1}) = 0$,
- *pente aux extrémités fixée* : $s'_{3,h}(x_1) = \alpha_1$, $s'_{3,h}(x_{n+1}) = \alpha_2$,
- *condition not-a-knot* (défaut en `Python`) : continuité de la dérivée troisième en x_2 et x_n

$$s'''_{3,h}(x_2^-) = s'''_{3,h}(x_2^+), \qquad s'''_{3,h}(x_n^-) = s'''_{3,h}(x_n^+).$$

Exemple 2.9. *On considère les données suivantes :* $(x_1, y_1) = (-1, -1)$, $(x_2, y_2) = (0, 1)$, $(x_3, y_3) = (1, 1)$. *On souhaite trouver la spline naturelle interpolant les données.*

L'expression générale de la spline est

$$s_{3,h}(x) = a_1 + b_1 x + c_1 x^2 + d_1 x^3 \qquad \text{pour } x \in [-1, 0]$$
$$s_{3,h}(x) = a_2 + b_2 x + c_2 x^2 + d_2 x^3 \qquad \text{pour } x \in [0, 1].$$

Conditions de continuité :

$$s_{3,h}(x_2^-) = s_{3,h}(x_2^+), \quad \Longrightarrow \quad a_1 = a_2$$
$$s'_{3,h}(x_2^-) = s'_{3,h}(x_2^+), \quad \Longrightarrow \quad b_1 = b_2$$
$$s''_{3,h}(x_2^-) = s''_{3,h}(x_2^+), \quad \Longrightarrow \quad c_1 = c_2.$$

Condition de spline naturelle

$$\begin{aligned} s''_{3,h}(x_1) = 0, \quad &\Longrightarrow \quad 2c_1 - 6d_1 = 0 \\ s''_{3,h}(x_3) = 0, \quad &\Longrightarrow \quad 2c_2 + 6d_2 = 0. \end{aligned}$$

On peut donc simplifier l'expression générale de la façon suivante :

$$\begin{aligned} s_{3,h}(x) &= a + bx + cx^2 + \frac{1}{3}cx^3 \qquad \text{pour } x \in [-1, 0] \\ s_{3,h}(x) &= a + bx + cx^2 - \frac{1}{3}cx^3 \qquad \text{pour } x \in [0, 1]. \end{aligned}$$

Finalement, on impose les 3 conditions d'interpolation

$$\begin{aligned} s_{3,h}(-1) = -1, \quad &\Longrightarrow \quad a - b + \frac{2}{3}c = -1 \\ s_{3,h}(0) = 1, \quad &\Longrightarrow \quad a = 1 \\ s_{3,h}(1) = 1, \quad &\Longrightarrow \quad a + b + \frac{2}{3}c = 1 \end{aligned}$$

et on trouve $a = 1$, $b = 1$ *et* $c = -\frac{3}{2}$. *Finalement la spline cubique interpolant est*

$$s_{3,h}\big|_{[-1,0]} = 1 + x - \frac{3}{2}x^2 - \frac{1}{2}x^3, \qquad s_{3,h}\big|_{[0,1]} = 1 + x - \frac{3}{2}x^2 + \frac{1}{2}x^3.$$

2.3.1 Analyse d'erreur

Pour l'interpolation spline cubique on a le résultat suivant :

Theorème 2.4. *Soit* $f : [a, b] \to \mathbb{R}$ *une fonction de classe* C^4 *et* $a = x_1 < x_2 < \ldots < x_{n+1} = b$ *des nœuds équirépartis dans* $[a, b]$. *Soit* $h = (b - a)/n$ *la longueur de chaque intervalle* $I_i = [x_i, x_{i+1}]$, $i = 1, \ldots, n + 1$. *La spline cubique interpolant les données* $(x_i, f(x_i))$ *satisfait l'estimation d'erreur*

$$\max_{x\in[a,b]} |f(x) - s_{3,h}(x)| \leq C_0 h^4 \max_{x\in[a,b]} |f^{(4)}(x)|, \tag{2.13}$$

$$\max_{x\in[a,b]} |f'(x) - s'_{3,h}(x)| \leq C_1 h^3 \max_{x\in[a,b]} |f^{(4)}(x)|, \tag{2.14}$$

$$\max_{x\in[a,b]} |f''(x) - s''_{3,h}(x)| \leq C_2 h^2 \max_{x\in[a,b]} |f^{(4)}(x)|, \tag{2.15}$$

où les constantes C_0, C_1 *et* C_2 *ne dépendent pas de* h.

Donc la méthode d'interpolation par spline cubique est d'ordre 4 dans l'approximation de la fonction f. Elle permet aussi d'approcher la dérivée première et deuxième de la fonction f. Dans ce cas, la méthode est d'ordre 3 dans l'approximation de la dérivée première et d'ordre 2 dans l'approximation de la dérivée deuxième.

Exemple 2.10. *On applique cette méthode pour approcher toujours la fonction* $f_2(x) = 1/(1+(5x)^2)$ *de l'exercice 2.5 sur un nombre croissant de sous-intervalles* $n = 16, 32, 64, 128, 256$.

```
import numpy as np
from scipy.interpolate import CubicSpline
f = lambda x : 1./(1+(5*x)**2)
err=np.array([])
h=np.array([])
for n in 2**np.arange(4,9):
    h=np.append(h,2./n)
    x=np.linspace(-1,1,n+1)  # interpolation nodes
    y=f(x)                   # measures
    s3h=CubicSpline(x,y)     # defines cubic spline s3h
    err=np.append(err, max(abs(f(xfine)-s3h(xfine))))

print("h: ", h)
print("err: ",err)

# OUTPUT
# h:  [0.125     0.0625    0.03125   0.015625  0.0078125]
# err:  [3.71093415e-03 6.37335632e-04 3.60663244e-05 2.05343200e-06
#  1.24035520e-07]
```

On note que chaque fois que le nombre des sous-intervalles est doublé (et donc h *est divisé par 2), l'erreur diminue grosso modo d'un facteur 16. De plus, on peut visualiser l'erreur en fonction de* h *sur un graphe en échelle logarithmique.*

```
import matplotlib.pyplot as plt
plt.loglog(h,err,'b')
plt.loglog(h,h**4,'k--')
plt.grid(True)
plt.legend(["err_h","slope 4"])
```

La Figure 2.12 montre le résultat obtenu. On a aussi tracé sur le même graphe une droite de pente 4. On voit bien que l'erreur décroît proportionnellement à h^4.

2.4 Approximation au sens des moindres carrés

On s'intéresse dans cette section au cas où les mesures à disposition sont polluées par des *erreurs de mesure*. On suppose qu'on ait à disposition $n+1$ données (x_i, y_i), $i = 1, \ldots, n+1$ et que les valeurs y_i proviennent de l'évaluation d'une fonction $f(x)$ inconnue, affectée pas une erreur aléatoire ε_i de mesure. Notre modèle est donc

$$y_i = f(x_i) + \varepsilon_i$$

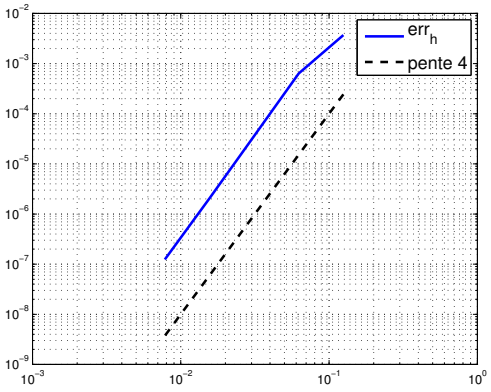


FIGURE 2.12 – Graphe en échelle logarithmique de l'erreur de l'interpolation spline de la fonction $f(x) = 1/(1+(5x)^2)$ sur n sous-intervalles en fonction de la longueur de chaque sous-intervalle $h = 2/n$.

et on essaye de reconstruire la fonction inconnue $f(x)$ à partir des données. On va faire l'hypothèse que les erreurs ε_i sont des variables aléatoires, indépendantes et identiquement distribuées avec espérance $\mathbb{E}[\varepsilon_i] = 0$ et variance $\mathbb{V}\text{ar}[\varepsilon_i] = \sigma^2$. Par exemple, les variables ε_i pourraient avoir distribution normale, $\varepsilon_i \sim N(0, \sigma^2)$.

On considère ici la méthode d'approximation polynomiale au sens des moindres carrés. On dénote par $\mathbb{P}_m$ l'espace des polynômes de degré $\leq m$.

Définition 2.8. *Le* ***polynôme de degré*** m ***approchant au sens des moindres carrés*** *les données* (x_i, y_i), $i = 1, \ldots, n+1$ *est le polynôme, qu'on appelle* p_m^{MC}, *satisfaisant*

$$p_m^{MC} = \underset{q \in \mathbb{P}_m}{\operatorname{argmin}} \sum_{i=1}^{n+1} (y_i - q(x_i))^2.$$

Autrement dit, p_m^{MC} *est tel que*

$$\sum_{i=1}^{n+1} (y_i - p_m^{MC}(x_i))^2 \leq \sum_{i=1}^{n+1} (y_i - q(x_i))^2, \qquad \forall q \in \mathbb{P}_m.$$

La Figure 2.13 montre un ensemble de 21 mesures et le polynôme de degré 1 (*droite de régression*) approchant les données au sens des moindres carrés.

Calcul du polynôme de moindres carrés (cas $m = 1$)

On se pose maintenant la question de calculer explicitement le polynôme des moindres carrés. On considère d'abord le cas de la droite de régression

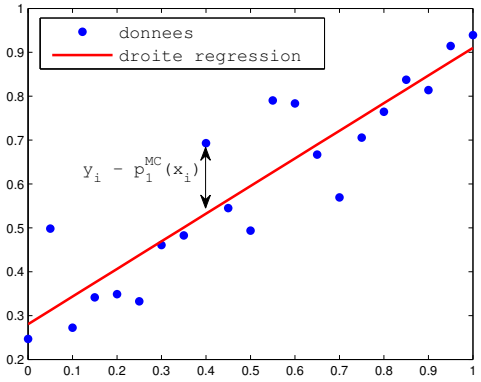


FIGURE 2.13 – Droite de régression (en rouge) des 21 mesures (en bleu).

(polynôme de degré 1). Une droite quelconque peut s'écrire sous la forme

$$q(x) = a_0 + a_1 x,$$

paramétrée par les deux constantes a_0 et a_1. On essaye de trouver

$$\min_{q\in\mathbb{P}_1} \sum_{i=1}^{n+1} (y_i - q(x_i))^2 = \min_{(a_0,a_1)\in\mathbb{R}^2} \sum_{i=1}^{n+1} (y_i - (a_0 + a_1 x_i))^2$$

c'est-à-dire on cherche le minimum de la fonction

$$\phi(a_0, a_1) = \sum_{i=1}^{n+1} (y_i - (a_0 + a_1 x_i))^2.$$

Le minimum doit satisfaire les conditions

$$\frac{\partial \phi}{\partial a_0} = 0, \qquad \frac{\partial \phi}{\partial a_1} = 0$$

ce qui correspond à

$$\begin{cases} \dfrac{\partial \phi}{\partial a_0} = -2 \displaystyle\sum_{i=1}^{n+1} (y_i - a_0 - a_1 x_i) = 0 \\ \dfrac{\partial \phi}{\partial a_1} = -2 \displaystyle\sum_{i=1}^{n+1} x_i (y_i - a_0 - a_1 x_i) = 0 \end{cases}$$

$$\Longrightarrow \begin{cases} (n+1) a_0 + \left(\sum_{i=1}^{n+1} x_i \right) a_1 = \sum_{i=1}^{n+1} y_i \\ \left(\sum_{i=1}^{n+1} x_i \right) a_0 + \left(\sum_{i=1}^{n+1} x_i^2 \right) a_1 = \sum_{i=1}^{n+1} x_i y_i. \end{cases}$$

Ce dernier est un système linéaire de deux équations dans les deux inconnues (a_0, a_1) :

$$\underbrace{\begin{bmatrix} \sum_{i=1}^{n+1} 1 & \sum_{i=1}^{n+1} x_i \\ \sum_{i=1}^{n+1} x_i & \sum_{i=1}^{n+1} x_i^2 \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} a_0 \\ a_1 \end{bmatrix}}_{\mathbf{a}} = \underbrace{\begin{bmatrix} \sum_{i=1}^{n+1} y_i \\ \sum_{i=1}^{n+1} x_i y_i \end{bmatrix}}_{\mathbf{b}}. \tag{2.16}$$

La résolution de ce système nous donne donc les coefficients de la droite de régression.

On remarque la chose suivante : si on introduit la matrice de Vandermonde $V \in \mathbb{R}^{(n+1)\times 2}$ correspondante aux monômes 1 et x évalués aux points x_i, $i = 1, \ldots, n+1$

$$V = \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots \\ & \\ 1 & x_{n+1} \end{bmatrix}$$

et le vecteur $\mathbf{y} = (y_1, y_2, \ldots, y_{n+1})^T$, alors on a

$$A = V^T V, \qquad \mathbf{b} = V^T \mathbf{y}$$

et le système (2.16) peut être écrit sous forme compacte comme

$$V^T V \mathbf{a} = V^T \mathbf{y}$$

Calcul du polynôme de moindres carrés (cas général)

Dans le cas général d'un polynôme de degré m, on introduit la matrice de Vandermonde $V \in \mathbb{R}^{(n+1)\times(m+1)}$ correspondante aux monômes $1, x, \ldots, x^m$ évalués aux points x_i, $i = 1, \ldots, n+1$:

$$V = \begin{bmatrix} 1 & x_1 & x_1^2 & \ldots & x_1^m \\ 1 & x_2 & x_2^2 & \ldots & x_2^m \\ \vdots & \vdots & & & \vdots \\ & & & & \\ 1 & x_{n+1} & x_{n+1}^2 & \ldots & x_{n+1}^m \end{bmatrix}.$$

Étant donné un polynôme $q(x) = a_0 + a_1 x + \ldots + a_m x^m$ de degré m, on a

$$\begin{bmatrix} q(x_1) \\ q(x_2) \\ \vdots \\ q(x_{n+1}) \end{bmatrix} = \begin{bmatrix} 1 & x_1 & x_1^2 & \ldots & x_1^m \\ 1 & x_2 & x_2^2 & \ldots & x_2^m \\ \vdots & \vdots & & & \vdots \\ 1 & x_{n+1} & x_{n+1}^2 & \ldots & x_{n+1}^m \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_m \end{bmatrix} = V\mathbf{a}$$

et

$$\sum_{i=1}^{n+1} (y_i - q(x_i))^2 = \|\mathbf{y} - V\mathbf{a}\|^2. \tag{2.17}$$

Le problème de trouver le polynôme de degré m approchant les données (x_i, y_i) au sens des moindres carrés revient donc à chercher le vecteur des coefficients $\mathbf{a} = (a_0, a_1, \ldots, a_m)^T$ qui minimise la quantité (2.17)

$$\min_{q\in\mathbb{P}_m} \sum_{i=1}^{n+1} (y_i - q(x_i))^2 \qquad \Leftrightarrow \qquad \min_{\mathbf{a}\in\mathbb{R}^{m+1}} \|\mathbf{y} - V\mathbf{a}\|^2.$$

On a le résultat suivant qui généralise celui qu'on a obtenu pour la droite de régression :

Proposition 2.5. *$p_m^{MC}(x) = a_0 + a_1 x + \ldots + a_m x^m$ est le polynôme approchant au sens des moindres carrés les données (x_i, y_i), $i = 1, \ldots, n+1$ si et seulement si le vecteur des coefficients $\mathbf{a} = (a_0, a_1, \ldots, a_m)^T$ est solution du système linéaire*

$$V^T V\mathbf{a} = V^T\mathbf{y} \tag{2.18}$$

appelé système des équations normales.

Remarque 2.1. *Si on prend $m = n$, la matrice de Vandermone correspondante est une matrice carrée inversible. Donc*

$$V^T V\mathbf{a} = V^T\mathbf{y} \qquad \Leftrightarrow \qquad V\mathbf{a} = \mathbf{y}$$

et le vecteur des coefficients correspond à celui du polynôme interpolant *les données (x_i, y_i), $i = 1, \ldots n+1$.*

En `Python` , le polynôme de degré m au sens des moindres carrés peut être calculé par la commande `pcoef=polyfit(x,y,m)`. Si on prend $m = n$, on obtient bien le polynôme interpolant.

Analyse d'erreur

L'analyse de l'erreur d'approximation $E_m = \max_{x\in[x_1,x_{n+1}]} |f(x) - p_m^{MC}(x)|$ est en général assez compliquée. On se limite à quelques considérations.

Imaginons que la "vraie" fonction $f(x)$ soit elle-même un polynôme de degré $m \ll n$: $f(x) = q_m(x) \in \mathbb{P}_m$.

S'il n'y avait pas d'erreur de mesure, c'est-à-dire $y_i = q_m(x_i)$, $i = 1, \ldots, n+1$, alors l'approximation polynomiale de degré m au sens des moindres carrés donnerait exactement la fonction q_m : $p_m^{MC}(x) = q_m(x)$. En fait, le polynôme $p_m^{MC} = q_m(x)$ est le seul polynôme de degré m pour lequel la somme des écarts au carré est zéro

$$\sum_{i=1}^{n+1} (y_i - p_m^{MC}(x_i))^2 = \sum_{i=1}^{n+1} (q_m(x_i) - p_m^{MC}(x_i))^2 = 0.$$

Si on considère maintenant les erreurs de mesure ε_i, alors ceci n'est plus vrai et on peut montrer que $E_m = \max_{x\in[x_1,x_{n+1}]} |q_m(x) - p_m^{MC}(x)|$ est

contrainte	déformation	contrainte	déformation
1.7850	0.0292	19.6350	0.4454
3.5700	0.0609	21.4200	0.5043
5.3550	0.0950	23.2050	0.5122
7.1400	0.1327	24.9900	0.6111
8.9250	0.1449	26.7750	0.7277
10.7100	0.2062	28.5600	0.7392
12.4950	0.2692	30.3450	0.8010
14.2800	0.2823	32.1300	0.8329
16.0650	0.3613	33.9150	0.9302
17.8500	0.4014	35.7000	1.0116

TABLE 2.2 – Données de contrainte et déformation du test de traction de l'exemple 2.2.

proportionnel à $\sigma\sqrt{\frac{m+1}{n+1}}$ où $\sigma = \sqrt{\mathbb{V}\mathrm{ar}[\varepsilon_i]}$ est l'écart-type des variables ε_i, si m est suffisamment petit par rapport à n.

Donc, même si chaque mesure a été polluée par une erreur ε_i de l'ordre de σ, l'erreur d'approximation est plus petit, de l'ordre de $\sigma\sqrt{\frac{m+1}{n+1}}$ et devient de plus en plus petit lorsqu'on augment le nombre de mesures à disposition. La méthode des moindres carrés permet donc de "filtrer" l'erreur de mesure.

La variance des variables ε_i peut être estimée par la formule suivante

$$\hat{\sigma}^2 = \frac{1}{n-m}\sum_{i=1}^{n+1}(y_i - p_m^{MC}(x_i))^2.$$

Si on considère le cas général où la fonction f n'est pas forcément un polynôme, on s'attend à ce qu'en augmentant le degré polynomial, l'approximation au sens des moindres carrés devient de plus en plus précise. Toutefois :

- On aura toujours une erreur due à l'erreur de mesure de l'ordre de $\sigma\sqrt{\frac{m+1}{n+1}}$ que l'on n'arrive pas à éliminer.
- Si m devient trop élevé, proche de n, le polynôme de moindres carrés se rapproche du polynôme interpolant et on a vu que le polynôme interpolant devient instable pour des degrés élevés. Donc l'approximation au sens des moindres carrés peut en fait se détériorer lorsque $m \to n$.

Exemple 2.11. *On reprend l'exemple du test de traction présenté au début du chapitre (exemple 2.2). Le tableau 2.2 montre les mesures obtenues. On calcule en* `Python` *les polynômes d'approximation au sens des moindres carrés de degré $m = 1, 5, 15$. Les données sont sauvegardées dans les variables* `x` *et* `y`*. On montre le code pour le cas $m = 15$.*

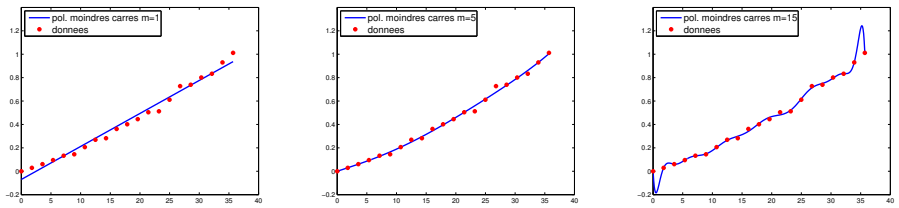


FIGURE 2.14 – Polynômes d'approximation des données du tableau 2.2 au sens des moindres carrés de degré $m = 1, 5, 15$.

```
import numpy as np
import matplotlib.pyplot as plt
xfine=np.arange(0,35.8,0.1)
pcoef=np.polyfit(x,y,15)
p=np.polyval(pcoef,xfine)
plt.plot(xfine,p,'b-')
plt.plot(x,y,'r.',linewidth=2,markersize=15)
plt.legend(["pol. least squares m=15","data"])
```

La figure 2.14 montre les trois courbes obtenues. On voit que le polynôme de degré $m = 5$ capture mieux le comportement des données par rapport au polynôme de degré $m = 1$. Par contre, le polynôme de degré $m = 15$ présente des oscillations très visibles dues au fait qu'on est proche de la condition d'interpolation.

Chapitre 3

Dérivation et Intégration numérique

Dans ce chapitre, on s'intéresse à développer des méthodes numériques pour calculer de façon approchée la dérivée d'une fonction $f(x)$ régulière au point $x = \bar{x}$

$$f'(\bar{x}),$$

ou bien l'intégrale de f sur l'intervalle $[a, b]$

$$\int_a^b f(x)dx.$$

Ces méthodes s'avèrent très utiles lorsque la fonction f a une expression compliquée et le calcul exact de la dérivée ou de l'intégrale définie est très compliqué voire impossible.

3.1 Approximation des dérivées par différences finies

Soit $f : [a, b] \to \mathbb{R}$ une fonction régulière. On souhaite approcher numériquement sa dérivée f' en un point $\bar{x} \in [a, b]$, en utilisant seulement des évaluations ponctuelles $f(x_i)$, $x_i \in [a, b], i = 1, \ldots, n$. La première idée à laquelle on pourrait penser est de remplacer la dérivée exacte $f'(\bar{x})$ par un *taux d'accroissement* :

$$f'(\bar{x}) \approx \frac{f(\bar{x} + h) - f(\bar{x})}{h}$$

pour un $h > 0$ tel que $\bar{x} + h \in [a, b]$. On s'attend à ce que l'approximation soit de plus en plus précise lorsque h devient petit.

On introduit la formule appelée **formule aux différences finies progressives**

$$\delta_h^+ f(\bar{x}) = \frac{f(\bar{x} + h) - f(\bar{x})}{h}. \tag{3.1}$$

On peut analyser la précision de cette formule en utilisant un développement de Taylor de f en $\bar{x}$:

$$f(\bar{x}+h) = f(\bar{x}) + f'(\bar{x})h + \frac{f''(\xi)}{2}h^2, \quad \text{pour un certain } \xi \in (\bar{x}, \bar{x}+h).$$

On a alors

$$\delta_h^+ f(\bar{x}) = \frac{1}{h}\left[f(\bar{x}) + f'(\bar{x})h + \frac{f''(\xi)}{2}h^2 - f(\bar{x})\right] = f'(\bar{x}) + \frac{f''(\xi)}{2}h$$

et donc

$$|f'(\bar{x}) - \delta_h^+ f(\bar{x})| \leq \frac{1}{2}\max_{x\in[a,b]}|f''(x)|h.$$

On a donc montré le résultat suivant :

Lemme 3.1. *Soit $f : [a,b] \to \mathbb{R}$ une fonction de classe C^2 et δ_h^+ la formule aux différences finies progressives. Alors*

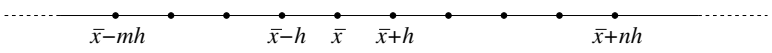
$$|f'(\bar{x}) - \delta_h^+ f(\bar{x})| \leq Ch, \quad \forall \bar{x} \in [a, b-h]$$

où la constante C est donnée par $C = \frac{1}{2}\max_{x\in[a,b]}|f''(x)|$.

On dit dans ce cas que la formule est d'ordre 1 car l'erreur est proportionnelle à h. Donc, si on divise le pas h par 2, on s'attend à ce que l'erreur d'approximation soit aussi divisée par deux.

La formule δ_h^+ utilise seulement les points $\bar{x}$ et $\bar{x}+h$ et les évaluations $f(\bar{x})$ et $f(\bar{x}+h)$ pour obtenir une approximation de $f'(\bar{x})$.

On peut imaginer construire des formules aux différences finies qui utilisent plus de points et d'évaluations de la fonction f. En général, imaginons utiliser le point $\bar{x}$ ainsi que n points à sa droite : $\bar{x}+h$, $\bar{x}+2h$, ..., $\bar{x}+nh$ et m points à sa gauche : $\bar{x}-h$, $\bar{x}-2h$, ..., $\bar{x}-mh$



On peut donner les définitions générales suivantes :

Définition 3.1. *Une formule aux différences finies qui utilise les $n+m+1$ points $\bar{x}+ih$, $i = -m, \ldots, n$, pour approcher la dérivée première d'une fonction régulière f en $\bar{x}$ est une expression du type*

$$D_h f(\bar{x}) = \frac{1}{h}\sum_{i=-m}^{n}\alpha_i f(\bar{x}+ih)$$

où les coefficients α_i ne dépendent pas de h.

Définition 3.2. *Une formule aux différences finies D_h est* ***consistante*** *si*

$$\lim_{h\to 0} D_h f(\bar{x}) = f'(\bar{x})$$

pour f suffisamment régulière.

Définition 3.3. *Une formule aux différences finies D_h est* ***d'ordre*** *p s'il existe une constante C dépendant de f mais autrement indépendante de h telle que*

$$|f'(\bar{x}) - D_h f(\bar{x})| \leq Ch^p$$

pourvu que la fonction f soit suffisamment régulière.

La formule aux différences finies progressives (3.1) utilise seulement les deux points $\bar{x}$ et $\bar{x}+h$ et est d'ordre 1.

On aurait aussi pu définir les formules

différences finies rétrogrades $$\delta_h^- f(\bar{x}) = \frac{f(\bar{x}) - f(\bar{x}-h)}{h} \tag{3.2}$$

différences finies centrées $$\delta_h^c f(\bar{x}) = \frac{f(\bar{x}+h) - f(\bar{x}-h)}{2h}. \tag{3.3}$$

On montre facilement que la formule aux différences finies rétrogrades est aussi d'ordre 1. Par contre, pour la formule aux différences finies centrées on a

$$\begin{aligned}\delta_h^c f(\bar{x}) &= \frac{f(\bar{x}+h) - f(\bar{x}-h)}{2h} \\ &= \frac{1}{2h}\left[f(\bar{x}) + f'(\bar{x})h + \frac{f''(\bar{x})}{2}h^2 + \frac{f'''(\xi_1)}{6}h^3 - f(\bar{x}) + f'(\bar{x})h - \frac{f''(\bar{x})}{2}h^2 + \frac{f'''(\xi_2)}{6}h^3\right] \\ &= f'(\bar{x}) + \frac{h^2}{12}\left[f'''(\xi_1) + f'''(\xi_2)\right] \end{aligned} \tag{3.4}$$

où $\xi_1 \in (\bar{x}, \bar{x}+h)$ et $\xi_2 \in (\bar{x}-h, \bar{x})$. Donc la formule est d'ordre 2.

Lemme 3.2. *Soit $f : [a,b] \to \mathbb{R}$ une fonction de classe C^3 et δ_h^c la formule aux différences finies centrées. Alors*

$$|f'(\bar{x}) - \delta_h^c f(\bar{x})| \leq Ch^2, \quad \forall \bar{x} \in [a+h, b-h]$$

où la constante C est donnée par $C = \frac{1}{6}\max_{x\in[a,b]} |f'''(x)|$.

Construction par un polynôme interpolant

La formule aux différences finies progressives peut être interprétée de la façon suivante : pour approcher la dérivée $f'(\bar{x})$ on calcule d'abord la droite interpolant les points $(\bar{x}, f(\bar{x}))$ et $(\bar{x}+h, f(\bar{x}+h))$

$$p_1(x) = f(\bar{x})\frac{\bar{x}+h-x}{h} + f(\bar{x}+h)\frac{x-\bar{x}}{h}$$

et on calcule ensuite la pente de la droite

$$\delta_h^+ f(\bar{x}) = p_1'(\bar{x}) = -\frac{1}{h} f(\bar{x}) + \frac{1}{h} f(\bar{x}+h).$$

Cette procédure peut être généralisée : étant donnés les $n+m+1$ points $\bar{x}+ih$, $i = -m \ldots, n$ on calcule d'abord le polynôme p_{n+m} de degré $n+m$ interpolant les valeurs $(\bar{x}+ih, f(\bar{x}+ih))$, $i = -m, \ldots, n$ et on construit ensuite la formule aux différences finies

$$D_h f(\bar{x}) = p_{n+m}'(\bar{x}).$$

Exemple 3.1. *On applique la procédure en prenant les trois points $\bar{x}-h$, $\bar{x}$ et $\bar{x}+h$. Le polynôme p_2 de degré 2 interpolant les valeurs $(\bar{x}+ih, f(\bar{x}+ih))$, $i = -1, 0, 1$ est*

$$p_2(x) = f(\bar{x}-h)\frac{(x-\bar{x})(x-\bar{x}-h)}{2h^2} + f(\bar{x})\frac{(x-\bar{x}+h)(x-\bar{x}-h)}{-h^2} + f(\bar{x}+h)\frac{(x-\bar{x}+h)(x-\bar{x})}{2h^2}. \quad (3.5)$$

Sa dérivée est

$$p_2'(x) = f(\bar{x}-h)\frac{2x-2\bar{x}-h}{2h^2} + f(\bar{x})\frac{2x-2\bar{x}}{-h^2} + f(\bar{x}+h)\frac{2x-2\bar{x}+h}{2h^2}$$

et la formule aux différences finies correspondante est

$$D_h f(\bar{x}) = p_2'(\bar{x}) = -\frac{f(\bar{x}-h)}{2h} + \frac{f(\bar{x}+h)}{2h}$$

qui coïncide avec la formule aux différences finies centrées.

Construction par la méthode des coefficients indéterminés

Une autre façon de construire des formules aux différences finies est d'écrire une expression générale

$$D_h f(\bar{x}) = \frac{1}{h} \sum_{i=-m}^{n} \alpha_i f(\bar{x}+ih)$$

et chercher ensuite les coefficients α_i de sorte que la formule soit la plus précise possible. Voyons ça sur un exemple.

Exemple 3.2. *On considère encore les trois points $\bar{x}-h$, $\bar{x}$, $\bar{x}+h$ et la formule générale*

$$D_h f(\bar{x}) = \frac{1}{h}\left[\alpha_{-1} f(\bar{x}-h) + \alpha_0 f(\bar{x}) + \alpha_1 f(\bar{x}+h)\right].$$

En utilisant un développement de Taylor on a

$$\begin{aligned} D_h f(\bar{x}) = & \frac{\alpha_{-1}}{h}\left[f(\bar{x}) - f'(\bar{x})h + \frac{f''(\bar{x})}{2}h^2 + O(h^3)\right] + \\ & \frac{\alpha_0}{h}f(\bar{x}) + \\ & \frac{\alpha_1}{h}\left[f(\bar{x}) + f'(\bar{x})h + \frac{f''(\bar{x})}{2}h^2 + O(h^3)\right] \end{aligned}$$

et donc

$$D_h f(\bar{x}) = \frac{1}{h}\underbrace{(\alpha_{-1} + \alpha_0 + \alpha_1)}_{=0} f(\bar{x}) + \underbrace{(\alpha_1 - \alpha_{-1})}_{=1} f'(\bar{x}) + \frac{h}{2}\underbrace{(\alpha_{-1} + \alpha_1)}_{=0} f''(\bar{x}) + O(h^2).$$

Pour que la formule donne une approximation de la dérivée première d'ordre 2, il faut imposer les relations

$$\begin{cases} \alpha_{-1} + \alpha_0 + \alpha_1 = 0 \\ \alpha_1 - \alpha_{-1} = 1 \\ \alpha_1 + \alpha_{-1} = 0. \end{cases}$$

La solution est $\alpha_{-1} = -\frac{1}{2}$, $\alpha_0 = 0$ *et* $\alpha_1 = \frac{1}{2}$, *qui correspond encore une fois à la formule aux différences finies centrées.*

3.1.1 Approximation des dérivées d'ordre supérieur

Les deux méthodes présentées dans les sections précédentes permettent aussi de construire des formules aux différences finies pour approcher des dérivées d'ordre supérieure. Par exemple, si on reprend l'exemple 3.1, on peut utiliser le polynôme interpolant p_2 pour obtenir une formule aux différences finies qui approche la dérivée seconde de f en $\bar{x}$:

$$D_h^2 f(\bar{x}) = p_2''(\bar{x}) = \frac{f(\bar{x} - h) - 2f(\bar{x}) + f(\bar{x} + h)}{h^2}$$

On vérifie facilement, en utilisant le développement de Taylor, que cette formule est d'ordre 2.

3.1.2 Effets des erreurs d'arrondis

Calculons avec `Python` la dérivée de la fonction $f(x) = \log(x)$ en $\bar{x} = 1$ par la formule aux différences finies progressives. On utilise des h de plus en plus petits : $h = 10^{-1}, 10^{-2}, \ldots, 10^{-14}$.

```
import numpy as np
f = lambda x : np.log(x)
for i in range(1,16):
    h=10**(-i)
```

```
    dhf = (f(1+h)-f(1))/h
    print("h=%1.0e"%h,"   dhf=",dhf)

# OUTPUT
# h=1e-01    dhf= 0.9531017980432493
# h=1e-02    dhf= 0.9950330853168092
# h=1e-03    dhf= 0.9995003330834232
# h=1e-04    dhf= 0.9999500033329731
# h=1e-05    dhf= 0.999995000398841
# h=1e-06    dhf= 0.9999994999180668
# h=1e-07    dhf= 0.9999999505838705
# h=1e-08    dhf= 0.9999999889225291
# h=1e-09    dhf= 1.000000082240371
# h=1e-10    dhf= 1.000000082690371
# h=1e-11    dhf= 1.000000082735371
# h=1e-12    dhf= 1.000088900581841
# h=1e-13    dhf= 0.9992007221625909
# h=1e-14    dhf= 0.9992007221626359
# h=1e-15    dhf= 1.1102230246251559
```

La dérivée exacte est $f'(1) = 1$. On voit que la formule nous donne une bonne approximation de la dérivée pour $h = 10^{-8}$. Toutefois, si on diminue ultérieurement h, l'approximation se dégrade. Ceci est dû aux erreurs d'arrondis qui interviennent dans le calcul de la différence $f(1+h) - f(1)$. On peut quantifier plus précisément cet effet.

Dans l'évaluation de la fonction $f(x)$, l'ordinateur va toujours introduire de petites erreurs d'arrondis sur le seizième chiffre décimal. On peut modéliser ça en disant que l'ordinateur va évaluer la quantité

$$\hat{f}(x) = f(x)(1+\eta)$$

où η est l'erreur d'arrondis de l'ordre de 10^{-16}. Notez que l'erreur d'arrondis est toujours une erreur *relative* si on utilise une représentation des nombres en virgule flottante. La formule que l'ordinateur vas évaluer est donc

$$\begin{aligned}
\hat{\delta}_h^+ f(\bar{x}) &= \frac{\hat{f}(\bar{x}+h) - \hat{f}(\bar{x})}{h} = \frac{f(\bar{x}+h)(1+\eta_1) - f(\bar{x})(1+\eta_2)}{h} \\
&= \frac{f(\bar{x}+h) - f(\bar{x})}{h} + \frac{\eta_1}{h} f(\bar{x}+h) - \frac{\eta_2}{h} f(\bar{x}) \\
&= f'(\bar{x}) + \frac{f''(\xi)}{2} h + \frac{\eta_1}{h} f(\bar{x}+h) - \frac{\eta_2}{h} f(\bar{x})
\end{aligned}$$

où $|\eta_1|, |\eta_2| \leq 10^{-16}$ et $\xi \in (\bar{x}, \bar{x}+h)$. Finalement, on a

$$|f'(\bar{x}) - \hat{\delta}_h^+ f(\bar{x})| \leq \max_{x\in[a,b]} |f''(x)| \frac{h}{2} + 2 \max_{x\in[a,b]} |f(x)| \frac{10^{-16}}{h} \tag{3.6}$$

et on voit qu'il y a un terme d'erreur qui décroît proportionnellement à h (erreur de troncature des différences finies) ainsi qu'un terme qui croit proportionnellement à $1/h$ dû aux erreurs d'arrondis. Si on prend une valeur de h trop petite, ce dernier terme d'erreur va dominer et l'approximation sera mauvaise.

On peut se poser la question de quelle est la valeur optimale de h à choisir. L'équation (3.6) nous fournit l'estimation d'erreur suivante

$$\varepsilon(h) \approx C_1 h + \frac{C_2 10^{-16}}{h}$$

où $C_1 = \frac{1}{2}\max_{x\in[a,b]} |f''(x)|$ et $C_2 = 2\max_{x\in[a,b]} |f(x)|$. La fonction $\varepsilon(h)$ a un seul minimum qu'on peut calculer par

$$\frac{d\varepsilon}{dh} = 0 \qquad \Longrightarrow \qquad C_1 - \frac{C_2 10^{-16}}{h^2} = 0,$$

ce qui donne la valeur optimale $h_{opt} = \sqrt{C_2 10^{-16}/C_1}$. En particulier, h_{opt} est de l'ordre de $\sqrt{10^{-16}} = 10^{-8}$ comme on l'a observé numériquement.

Plus généralement, si on considère une formule aux différences finies d'ordre p, on aura une erreur

$$\varepsilon(h) \approx C_1 h^p + \frac{C_2 10^{-16}}{h}$$

et la valeur optimale h_{opt} est de l'ordre de $\sqrt[p+1]{10^{-16}}$.

3.2 Intégration numérique

Soit $f : [a, b] \to \mathbb{R}$ une fonction régulière. On souhaite approcher numériquement l'intégrale

$$I = \int_a^b f(x)dx$$

en utilisant seulement des évaluations ponctuelles de la fonction f. Choisissons des points distincts $a \le x_1 < x_2 < \ldots < x_n \le b$ dans l'intervalle $[a, b]$, pas forcement équirépartis.

Définition 3.4. *On appelle* ***formule de quadrature*** $Q(f)$ *pour approcher l'intégrale* I *une formule du type*

$$Q(f) = \sum_{i=1}^{n} \alpha_i f(x_i)$$

où α_i *sont des nombres réels. Les points* x_i *sont appelés* ***nœuds de quadrature*** *et les coefficients* α_i ***poids*** *(ou coefficients de pondération).*

Formule composite du point milieu

Une idée très simple pour approcher l'intégrale $I = \int_a^b f(x)dx$ est illustrée en Figure 3.1. On divise l'intervalle $[a, b]$ en n sous-intervalles $I_i = [x_{i-1}, x_i]$

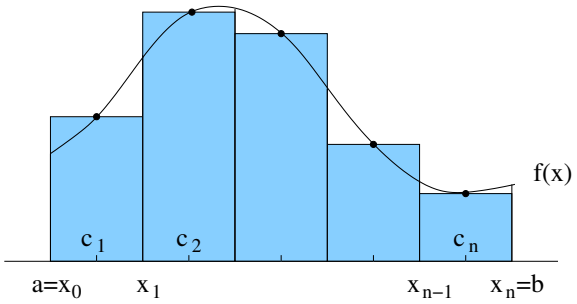


FIGURE 3.1 – Illustration graphique de la formule composite du point milieu

de longueur $h = (b - a)/n$, où $x_i = a + ih$, $i = 0, \ldots, n$. Dans chaque intervalle I_i, l'intégrale $\int_{x_{i-1}}^{x_i} f(x)dx$ est approché par l'aire du rectangle de base h et hauteur $f(\frac{x_{i-1}+x_i}{2})$.

Si on appelle $c_i = \frac{x_{i-1}+x_i}{2}$ le point milieu de l'intervalle I_i, la procédure précédente correspond à définir la formule de quadrature

$$Q_h^{pm}(f) = \sum_{i=1}^{n} hf(c_i) \tag{3.7}$$

dont les nœuds de quadrature sont les n points milieux c_i, $i = 1, \ldots, n$ et les poids α_i sont tous égaux à h. Cette formule est appelée *formule composite du point milieu*. Le terme *composite* indique que la formule du point milieu est appliquée sur chaque sous-intervalle I_i. Lorsqu'on utilise un seul intervalle ($i = 1$), la formule est appelée *simple*.

On peut facilement implémenter en `Python` la formule composite du point milieu. La fonction suivante donne une implémentation possible :

```
def midpoint(a,b,n,f):
# Composite midpoint formula
#  - a,b: boundaries of the integration interval
#  - n: number of sub-intervals
#  - f: function to integrate (anonymous function f= lambda x: ...)
    h=(b-a)/n
    xi = np.linspace(a+h/2,b-h/2,n)  # quadrature nodes
    alphai=np.full(n,h)              # weights
    Qh_mp=np.dot(alphai,f(xi))       # quadrature formula
    return Qh_mp
```

Formule composite du trapèze

On procède comme pour la formule composite du point milieu et on divise l'intervalle $[a, b]$ en n sous-intervalles $I_i = [x_{i-1}, x_i]$ de longueur $h = (b-a)/n$. Cette fois ci, au lieu d'approcher l'intégrale $\int_{x_{i-1}}^{x_i} f(x)dx$ dans chaque sous-intervalle par un rectangle, on calcule l'aire du *trapèze* défini par les quatre points $(x_{i-1}, 0)$, $(x_i, 0)$, $(x_i, f(x_i))$, $(x_{i-1}, f(x_{i-1}))$. Cette aire est $\frac{h}{2}(f(x_{i-1}) + f(x_i))$ et on peut définir la *formule composite du trapèze*

$$\begin{aligned} Q_h^{trap}(f) &= \sum_{i=1}^{n} \frac{h}{2}\left(f(x_{i-1}) + f(x_i)\right) \\ &= \frac{h}{2}f(x_0) + hf(x_1) + \ldots + hf(x_{n-1}) + \frac{h}{2}f(x_n). \end{aligned} \tag{3.8}$$

La figure 3.2 donne une interprétation graphique de la méthode. La formule

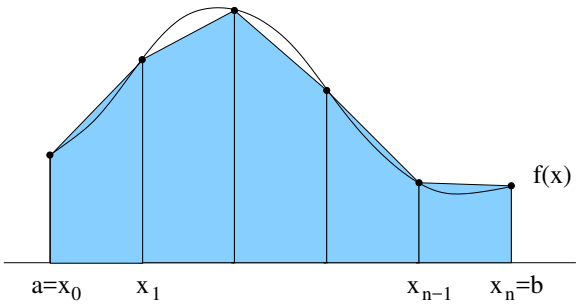


FIGURE 3.2 – Illustration graphique de la formule composite du trapèze

composite du trapèze utilise les $n+1$ nœuds x_i, $i = 0, \ldots, n$ et les poids $\alpha_0 = \alpha_n = h/2$ et $\alpha_i = h$, $i = 1, \ldots, n-1$.

À première vue, cette formule pourrait sembler plus précise que la formule du point milieu, mais on verra que ce n'est en fait pas forcement le cas.

Voici une implémentation possible en `Python`

```
def trap(a,b,n,f):
# Composite trapezoidal formula
#  - a,b: boundaries of the integration interval
#  - n: number of sub-intervals
#  - f: function to integrate (anonymous function f= lambda x: ...)
    h=(b-a)/n
    xi = np.linspace(a,b,n+1)                    # quadrature nodes
    alphai=np.hstack((h/2,np.full(n-1,h),h/2)) # weights
    Qh_trap=np.dot(alphai,f(xi))                 # quadrature formula
    return Qh_trap
```

Formule composite de Simpson

Une troisième idée pour construire une formule de quadrature est d'utiliser sur chaque sous-intervalle les points x_{i-1}, x_i ainsi que le point milieu $c_i = \frac{x_{i-1}+x_i}{2}$.

Sur chaque sous-intervalle, on peut calculer le polynôme $p_2^{(i)}$ de degré 2 interpolant les données $(x_{i-1}, f(x_{i-1}))$, $(x_i, f(x_i))$, $(c_i, f(c_i))$ puis remplacer l'intégrale $\int_{x_{i-1}}^{x_i} f(x)dx$ par l'approximation $\int_{x_{i-1}}^{x_i} p_2^{(i)}(x)dx$. On laisse les calculs comme exercice. La formule qu'on obtient est connue sous le nom de *formule composite de Simpson*

$$\begin{aligned} Q_h^{simp}(f) &= \sum_{i=1}^{n} \frac{h}{6}\left(f(x_{i-1}) + 4f(c_i) + f(x_i)\right) \\ &= \frac{h}{6}f(x_0) + \frac{h}{3}\sum_{i=1}^{n-1} f(x_i) + \frac{2h}{3}\sum_{i=1}^{n} f(c_i) + \frac{h}{6}f(x_n) \end{aligned} \tag{3.9}$$

et utilise les nœuds x_i, $i = 0, \ldots, n$ et c_i, $i = 1, \ldots, n$ ($2n+1$ en total).

Une implémentation possible en `Python` est donnée ci-dessous :

```
def simpson(a,b,n,f):
# Composite Simpson formula
#  - a,b: boundaries of the integration interval
#  - n: number of sub-intervals
#  - f: function to integrate (anonymous function f= lambda x: ...)
    h=(b-a)/n
    xi=np.linspace(a,b,n+1);        # sub-inteval boundaries
    alphai=(h/3)*np.hstack((0.5,np.ones(n-1),0.5)) # weights associated to x_i
    ci=np.linspace(a+h/2,b-h/2,n); # sub-interval mid-points
    betai=(2*h/3)*np.ones(n);   # weights associated to c_i
    Qh_simp = np.dot(alphai,f(xi))+np.dot(betai,f(ci))
    return Qh_simp
```

3.2.1 Analyse d'erreur

Pour analyser la précision d'une formule de quadrature, on commence par introduire un concept important.

Définition 3.5. *Une formule de quadrature $Q(f) = \sum_{i=1}^{n} \alpha_i f(x_i)$ a* ***degré d'exactitude*** *r si elle intègre exactement tout polynôme de degré $\leq r$, c'est-à-dire si*

$$Q(p) = \int_a^b p(x)dx, \qquad \forall p(x) \text{ polynôme de degré} \leq r$$

mais non pas les polynômes de degré $r+1$.

On remarque que pour vérifier si une formule a degré d'exactitude r, il suffit de vérifier qu'elle intègre exactement tout monôme x^s, $s \leq r$, c'est-à-dire

$$Q(x^s) = \int_a^b x^s dx, \qquad 0 \leq s \leq r, \quad s \text{ entier.} \tag{3.10}$$

En fait, si une formule de quadrature satisfait (3.10), grâce à sa linéarité on a alors pour tout polynôme $p(x) = \sum_{k=0}^{r} a_k x^k$

$$Q(p) = \sum_{i=1}^{n} \alpha_i p(x_i) = \sum_{i=1}^{n} \alpha_i \sum_{k=0}^{r} a_k x_i^k$$
$$= \sum_{k=0}^{r} a_k \underbrace{\sum_{i=1}^{n} \alpha_i x_i^k}_{=\int_a^b x^k dx} = \int_a^b \sum_{k=0}^{r} a_k x^k dx = \int_a^b p(x)dx$$

et donc la formule de quadrature a degré d'exactitude r.

Exemple 3.3. *On vérifie que la formule de Simpson a degré d'exactitude $r = 3$. Prenons un sous-intervalle $I_i = [x_{i-1}, x_i]$. On va vérifier que*

$$\frac{h}{6}\left[p(x_{i-1}) + 4p(c_i) + p(x_i)\right] = \int_{x_{i-1}}^{x_i} p(x)dx, \qquad \textit{pour } p(x) = 1, x, x^2, x^3.$$

Pour simplifier les calculs, considérons l'intervalle $I_i = [-1, 1]$ de longueur $h = 2$, avec $x_{i-1} = -1$, $x_i = 1$ et $c_i = 0$ (mais le résultat est valable sur n'importe quel intervalle). On a

$$p(x) = 1, \qquad \frac{h}{6}(1 + 4 + 1) = 2 = \int_{-1}^{1} 1dx$$
$$p(x) = x, \qquad \frac{h}{6}(-1 + 4 * 0 + 1) = 0 = \int_{-1}^{1} xdx$$
$$p(x) = x^2, \qquad \frac{h}{6}((-1)^2 + 4 * (0)^2 + 1^2) = \frac{2}{3} = \int_{-1}^{1} x^2 dx$$
$$p(x) = x^3, \qquad \frac{h}{6}((-1)^3 + 4 * (0)^3 + 1^3) = 0 = \int_{-1}^{1} x^3 dx.$$

On vérifie facilement aussi que

$$p(x) = x^4, \qquad \frac{h}{6}((-1)^4 + 4 * (0)^4 + 1^4) = \frac{2}{3} \neq \int_{-1}^{1} x^4 dx = \frac{2}{5}.$$

Ainsi, la formule de Simpson intègre exactement tout polynôme de degré 3 mais non pas les polynômes de degré 4.

Pour les formules du point milieu, du trapèze et de Simpson, on peut construire le tableau suivant :

	point milieu	trapèze	Simpson
degré d'exactitude	1	1	3

On revient maintenant à la question de quantifier la précision d'une formule de quadrature.

Définition 3.6. *Une formule de quadrature composite $Q_h(f)$ sur n sous-intervalles de longueur $h = (b-a)/n$ pour approcher l'intégrale $I = \int_a^b f(x)dx$ est* ***d'ordre*** **p** *s'il existe une constante C dépendant de f mais indépendante de h telle que*

$$\left| \int_a^b f(x)dx - Q_h(f) \right| \leq Ch^p$$

pourvu que la fonction f soit suffisamment régulière.

On a le résultat général suivant

Theorème 3.3. *Soit $Q_h(f)$ une formule de quadrature composite sur n sous-intervalles de longueur $h = (b-a)/n$ pour approcher l'intégrale $I = \int_a^b f(x)dx$. Si $Q_h(f)$ a degré d'exactitude r, alors elle est d'ordre $r+1$ pourvu que la fonction soit de classe C^{r+1} ($(r+1)$ fois continûment dérivable sur l'intervalle $[a,b]$). En particulier, on a*

$$\left| \int_a^b f(x)dx - Q_h(f) \right| \leq C \max_{x\in[a,b]} |f^{(r+1)}(x)|h^{r+1} \tag{3.11}$$

où la constante $C = \frac{(b-a)}{2^r(r+1)!}$ ne dépend ni de f ni de h.

Démonstration. Puisque la formule est composite, on peut l'écrire comme $Q_h(f) = \sum_{i=1}^n Q^{(i)}(f)$, où $Q^{(i)}$ est la formule de quadrature appliquée a l'i-ème sous-intervalle.

Considérons un sous-intervalle $I_i = [x_{i-1}, x_i]$ et le développement de Taylor de la fonction f autour du point milieu $c_i = \frac{x_{i-1}+x_i}{2}$ jusqu'à l'ordre $r+1$

$$f(x) = \underbrace{f(c_i) + f'(c_i)(x-c_i) + \ldots + \frac{f^{(r)}(c_i)}{r!}(x-c_i)^r}_{T_f^r(x)} + \underbrace{\frac{f^{(r+1)}(\xi_i)}{(r+1)!}(x-c_i)^{r+1}}_{R_f^r(x)}$$

où $\xi_i \in (c_i, x)$. On a indiqué par T_f^r le polynôme de Taylor de degré r et par R_f^r le reste de Lagrange.

On a que $Q^{(i)}(T_f^r) = \int_{x_{i-1}}^{x_i} T_f^r(x)dx$ grâce au fait que la formule intègre

exactement tout polynôme de degré $\leq r$. Donc

$$\left|\int_{x_{i-1}}^{x_i} f(x)dx - Q^{(i)}(f)\right| = \left|\int_{x_{i-1}}^{x_i} R_f^r(x)dx - Q^{(i)}(R_f^r)\right| \leq \left|\int_{x_{i-1}}^{x_i} R_f^r(x)dx\right| + \left|Q^{(i)}(R_f^r)\right|$$
$$\leq \frac{(h/2)^{r+1}}{(r+1)!} \max_{x\in[x_{i-1},x_i]} |f^{(r+1)}(x)| \left(\left|\int_{x_{i-1}}^{x_i} 1dx\right| + |Q^{(i)}(1)|\right)$$
$$\leq \frac{h^{r+1}}{2^{r+1}(r+1)!} \max_{x\in[x_{i-1},x_i]} |f^{(r+1)}(x)|2h.$$

Finalement

$$\left|\int_a^b f(x)dx - Q_h(f)\right| = \left|\sum_{i=1}^n \int_{x_{i-1}}^{x_i} f(x)dx - Q^{(i)}(f)\right| \leq \sum_{i=1}^n \left|\int_{x_{i-1}}^{x_i} f(x)dx - Q^{(i)}(f)\right|$$
$$\leq \frac{h^{r+1}}{2^r(r+1)!} \max_{x\in[a,b]} |f^{(r+1)}(x)| \sum_{i=1}^n h = \frac{(b-a)}{2^r(r+1)!} \max_{x\in[a,b]} |f^{(r+1)}(x)|h^{r+1}$$

ce qui montre le résultat. □

Grâce au théorème 3.3, on peut établir le tableau suivant pour les formules composites qu'on a vues jusqu'à présent :

	point milieu	trapèze	Simpson
ordre	2	2	4

Exemple 3.4. *On veut calculer l'intégrale*

$$I = \int_0^{\frac{\pi}{2}} \frac{\sin(x)\cos^3(x)}{4-\cos^2(2x)}dx. \tag{3.12}$$

Dans ce cas, après de longs calculs, on arrive à trouver la solution exacte $I = \frac{\log(3)}{16}$.

On calcule l'intégrale par la formule composite du trapèze, en prenant un nombre croissant de sous-intervalles $n = 2, 4, 8, \ldots, 1024$.

```
f = lambda x: (np.sin(x)*np.cos(x)**3)/(4-np.cos(2*x)**2)
a=0; b=np.pi/2
Iex=np.log(3)/16
print("Iex=",Iex)
Qhtrap=np.array([]); errQhtrap=np.array([]);
N=np.array([2**i for i in range(1,11)])
h=(b-a)/N
for n in N:
    Q=trap(a,b,n,f)
    print("n=%4d"%n,"  Qh=",Q)
    Qhtrap=np.append(Qhtrap,Q)
```

```
# OUTPUT
# Iex= 0.06866326804175686
# n=   2   Qh= 0.04908738521234053
# n=   4   Qh= 0.06421229026829854
# n=   8   Qh= 0.06758370289650516
# n=  16   Qh= 0.06839501624127321
# n=  32   Qh= 0.06859630316226781
# n=  64   Qh= 0.06864653288981529
# n= 128   Qh= 0.0686590846320792
# n= 256   Qh= 0.06866222221296711
# n= 512   Qh= 0.06866300658603605
# n=1024   Qh= 0.06866320267791895
```

On voit que l'approximation devient de plus en plus précise en augmentant le nombre de sous-intervalles. En utilisant $n = 1024$ *on obtient un résultat avec 6 chiffres significatifs corrects.*

Puisqu'on connaît dans ce cas la valeur exacte de l'intégrale, on peut aussi calculer l'erreur commise par la formule du trapèze

```
errQhtrap=abs(Iex-Qhtrap);
for i in range(0,len(errQhtrap)):
    print("n=%4d"%N[i],"  err=%2.16f"%errQhtrap[i])

# OUTPUT
# n=   2   err=0.0195758828294163
# n=   4   err=0.0044509777734583
# n=   8   err=0.0010795651452517
# n=  16   err=0.0002682518004836
# n=  32   err=0.0000669648794891
# n=  64   err=0.0000167351519416
# n= 128   err=0.0000041834096777
# n= 256   err=0.0000010458287898
# n= 512   err=0.0000002614557208
# n=1024   err=0.0000000653638379
```

On voit que l'erreur diminue lorsqu'on augmente le nombre de sous-intervalles. De plus, si on double n *(et donc on divise par deux la longueur de chaque sous-intervalle), l'erreur est approximativement divisée par* 4*, ce qui confirme que la formule composite du trapèze est d'ordre* 2*.*

Ceci est encore mieux perçu si on visualise l'erreur en fonction de h *en échelle logarithmique*

```
import matplotlib.pyplot as plt
plt.loglog(h,errQhtrap,'b-',linewidth=2)
plt.loglog(h,h**2,'k--',linewidth=2)
plt.loglog(h,h**4,'k-.',linewidth=2)
plt.grid(True)
plt.legend(['err. trapeze','pente 2','pente 4'])
```

On peut répéter les mêmes calculs pour les formules composites du point milieu et de Simpson. Les résultats sont montrés dans la Figure 3.3. On

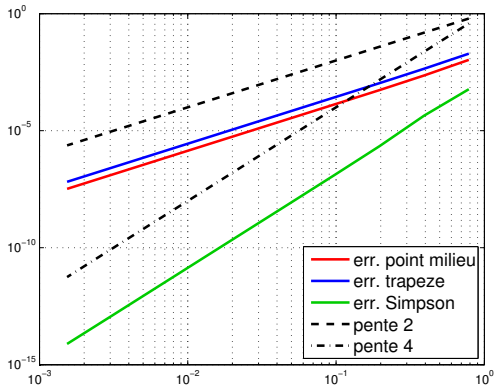


FIGURE 3.3 – Erreur des formules composites du point milieu (rouge), trapèze (bleu), Simpson (vert) en fonction de h pour approcher l'intégrale (3.12).

remarque que la convergence des formules du point milieu et du trapèze est d'ordre 2 tandis que la convergence de la formule de Simpson est d'ordre 4, comme prévu par la théorie. De plus, on note que l'erreur de la formule du trapèze est plus grand (d'un facteur 2 environ) que celui de la formule du point milieu.

3.2.2 Extrapolation de Richardson

La méthode d'extrapolation de Richardson est une méthode générale qui permet, à partir d'une formule d'approximation d'une certaine précision, d'en obtenir d'autres plus précises. On va l'appliquer aux formules composites de quadrature, bien qu'elle puisse aussi être appliquée dans beaucoup d'autres situations comme par exemple la dérivation numérique ou l'interpolation de fonctions.

Pour approcher l'intégrale $I = \int_a^b f(x)dx$, considérons une formule composite de quadrature $Q_h(f)$ qui utilise une partition de l'intervalle $[a, b]$ en n sous-intervalles $I_i = [x_{i-1}, x_i]$, $i = 1, \ldots, n$, de longueur $h = (b - a)/n$.

En supposant que la formule soit d'ordre p, on peut écrire le développement suivant :

$$Q_h(f) = I + Ch^p + O(h^q), \qquad q > p \tag{3.13}$$

où C est une constante convenable qui dépend en général d'une certaine dérivée de f.

Exemple 3.5. *Considérons la formule composite du point milieu* (3.7)

$$Q_h^{pm}(f) = \sum_{i=1}^{n} hf(c_i), \qquad c_i = \frac{x_i + x_{i-1}}{2}.$$

Sur chaque intervalle $[x_{i-1}, x_i]$ *on peut écrire un développement de Taylor de* f *autour de* c_i *jusqu'à l'ordre 4. On a*

$$\begin{aligned}\int_{x_{i-1}}^{x_i} f(x)dx &= \int_{x_{i-1}}^{x_i} f(c_i)dx + \int_{x_{i-1}}^{x_i} f'(c_i)(x-c_i)dx + \int_{x_{i-1}}^{x_i} \frac{f''(c_i)}{2}(x-c_i)^2dx \\ &\quad + \int_{x_{i-1}}^{x_i} \frac{f'''(c_i)}{6}(x-c_i)^3dx + \int_{x_{i-1}}^{x_i} O((x-c_i)^4)dx \\ &= hf(c_i) + \frac{f''(c_i)}{2}\frac{h^3}{12} + O(h^5)\end{aligned}$$

où $O(h^5)$ *désigne un reste de l'ordre de* h^5 *et on a utilisé le fait que* $\int_{x_{i-1}}^{x_i}(x-c_i)^r dx$ *est zéro pour tout* r *impair. Pour la formule composite du point milieu, on a donc*

$$\begin{aligned}Q_h^{pm}(f) - I &= \sum_{i=1}^{n}\left(hf(c_i) - \int_{x_{i-1}}^{x_i} f(x)dx\right) \\ &= \sum_{i=1}^{n}\left(-\frac{f''(c_i)}{2}\frac{h^3}{12} + O(h^5)\right) = -\frac{1}{24}\left(\sum_{i=1}^{n} hf''(c_i)\right)h^2 + O(h^4)\end{aligned}$$

où on a tenu compte du fait que $\sum_{i=1}^{n} O(h^5) = nO(h^5) = \frac{(b-a)}{h}O(h^5) = O(h^4)$. *En observant maintenant que*

$$\sum_{i=1}^{n} hf''(c_i) = Q_h^{pm}(f'') = \int_a^b f''(x)dx + O(h^2),$$

on arrive finalement à

$$Q_h^{pm}(f) = I + Ch^2 + O(h^4), \qquad \text{avec } C = -\frac{1}{24}\int_a^b f''(x)dx$$

qui correspond à l'expression générale (3.13) *avec* $p = 2$ *et* $q = 4$.

Considérons maintenant la même formule de quadrature mais avec la moitié de sous-intervalles, c'est-à-dire avec un pas d'espace de $2h$. On aura

$$Q_{2h}(f) = I + C(2h)^p + O(h^q).$$

En multipliant cette dernière relation par 2^{-p} et soustrayant (3.13), on obtient

$$2^{-p}Q_{2h}(f) - Q_h(f) = (2^{-p} - 1)I + O(h^q)$$

et donc

$$\frac{2^p Q_h(f) - Q_{2h}(f)}{2^p - 1} = I + O(h^q).$$

On peut alors définir la nouvelle formule de quadrature sur n sous-intervalles

$$\tilde{Q}_h(f) = \frac{2^p Q_h(f) - Q_{2h}(f)}{2^p - 1}$$

et l'argument précédent montre que cette formule est d'ordre $q > p$.

Il est assez remarquable qu'à partir des deux formules $Q_h(f)$ et $Q_{2h}(f)$ d'ordre p, on ait pu obtenir la formule $\tilde{Q}_h(f)$ qui est d'ordre $q > p$ et donc plus précise.

La technique qu'on vient de décrire est connue sous le nom de *technique d'extrapolation de Richardson*. Elle fonctionne en général très bien, pour autant qu'on connaisse l'ordre p de la méthode de départ. On remarque également que pour que la formule $\tilde{Q}_h(f)$ soit d'ordre $q > p$, on demande plus de régularité sur la fonction f. Par exemple, la formule composite du point milieu $Q_h^{pm}(f)$ est d'ordre $p = 2$ si la fonction est de classe C^2 tandis que son extrapolation de Richardson sera d'ordre $q = 4$ si la fonction est de classe C^4 (voir Exemple 3.5).

3.2.3 Estimation a posteriori de l'erreur

Une question typique qu'on se pose lorsqu'on calcule une intégrale par une formule de quadrature est de savoir combien de sous-intervalles il faut utiliser pour avoir une erreur plus petite qu'une tolérance donnée.

Il est difficile de répondre à cette question car en général, on ne connaît pas la valeur exacte de l'intégrale et on ne peut donc pas évaluer l'erreur. Toutefois, on peut procéder de la façon suivante. On calcule la formule de quadrature sur n sous-intervalles, $Q_h(f)$, et sur $n/2$ sous-intervalles, $Q_{2h}(f)$. On peut alors calculer l'extrapolation de Richardson $\tilde{Q}_h(f)$, qui donne normalement une bien meilleure approximation de l'intégrale que la formule $Q_h(f)$ car elle est d'ordre plus élevé. On peut donc estimer l'erreur commise par la formule $Q_h(f)$ par l'*indicateur d'erreur*

$$\eta_h = |\tilde{Q}_h(f) - Q_h(f)|.$$

On a enfait

$$|I - Q_h(f)| \leq |I - \tilde{Q}_h(f)| + |\tilde{Q}_h(f) - Q_h(f)| = \underbrace{\eta_h}_{O(h^p)} + O(h^q) \approx \eta_h.$$

Si l'erreur estimée ne satisfait par la tolérance donnée, on double alors le nombre de sous-intervalles et on continue itérativement jusqu'à ce que l'erreur estimée soit plus petite que la tolérance fixée. Cette procédure est résumée dans l'algorithme suivant

Algorithme 3.1 : Formule de quadrature adaptative basée sur l'extrapolation de Richardson

Données : $f(x)$, $[a, b]$, n_0, tol
Résultat : I, err, n
$h = (b-a)/n_0$ // n_0: nombre initial de sous-interv.;
$I_0 = Q_h(f)$ // estimation initiale de I;
$k = 0$;
$err = tol + 1$;
tant que $err > tol$ **faire**
  $k = k + 1$;
  $h = h/2$;
  $I_k = Q_h(f)$ // nouvelle approximation de I;
  $\tilde{I}_k = \frac{2^p I_k - I_{k-1}}{2^p - 1}$ // extrapolation de Richardson;
  $err = |\tilde{I}_k - I_k|$;
fin
$I = \tilde{I}_k$, $n = (b-a)/h$.

Chapitre 4

Systèmes linéaires – méthodes directes

Dans ce chapitre, on s'intéresse à la résolution numérique d'un système linéaire de n équations dans les n inconnues $x_1, \ldots, x_n$:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = b_2 \\ \ldots \\ a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n = b_n. \end{cases} \tag{4.1}$$

Si on définit les vecteurs $\mathbf{x}, \mathbf{b} \in \mathbb{R}^n$ et la matrice $A \in \mathbb{R}^{n\times n}$

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \qquad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}, \qquad A = \begin{bmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \vdots & & & \vdots \\ a_{n1} & a_{n2} & \ldots & a_{nn} \end{bmatrix},$$

alors le système (4.1) s'écrit de façon compacte comme

$$A\mathbf{x} = \mathbf{b}.$$

4.1 Systèmes triangulaires

Il y a des systèmes qui sont particulièrement simples à résoudre. Considérons pas exemple un **système linéaire triangulaire inférieur**

$$\begin{cases} a_{11}x_1 & & & & = b_1 \\ a_{21}x_1 & + a_{22}x_2 & & & = b_2 \\ a_{31}x_1 & + a_{32}x_2 & + a_{33}x_3 & & = b_3 \\ \ldots & & & & \\ a_{n1}x_1 & + a_{n2}x_2 & + \ldots & + a_{nn}x_n & = b_n \end{cases} \tag{4.2}$$

avec $a_{ii} \neq 0$, $\forall i = 1, \ldots, n$. Pour résoudre ce système, on peut commencer par la première équation

$$x_1 = \frac{b_1}{a_{11}},$$

puis on passe à la deuxième équation

$$x_2 = \frac{1}{a_{22}} (b_2 - a_{21} x_1)$$

et ainsi de suite jusqu'à la dernière équation.

On peut donc construire l'algorithme suivant dit de *substitution directe* (en anglais forward substitution)

Algorithme 4.1 : Algorithme de substitution directe

pour $i = 1, \ldots, n$ **faire**
  $x_i = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j \right)$;
fin

De même, si on a un **système triangulaire supérieur**

$$\begin{cases} a_{11}x_1 + & a_{12}x_2 + & \ldots + & a_{1,n-1}x_{n-1} + & a_{1n}x_n = & b_1 \\ & a_{22}x_2 + & \ldots + & a_{2,n-1}x_{n-1} + & a_{2n}x_n = & b_2 \\ & & & & \ldots & \\ & & & a_{n-1,n-1}x_{n-1} + & a_{n-1,n}x_n = & b_{n-1} \\ & & & & a_{nn}x_n = & b_n \end{cases} \tag{4.3}$$

avec $a_{ii} \neq 0$, $\forall i = 1, \ldots, n$, on peut commencer par résoudre la dernière équation $x_n = b_n / a_{nn}$ et remonter jusqu'à la première équation. Ceci donne l'algorithme de *substitution rétrograde*

Algorithme 4.2 : Algorithme de substitution rétrograde

pour $i = n, \ldots, 1$ **faire**
  $x_i = \frac{1}{a_{ii}} \left(b_i - \sum_{j=i+1}^{n} a_{ij} x_j \right)$;
fin

4.2 Algorithme d'élimination de Gauss et factorisation LU

Étant donné un système quelconque d'équations linéaires, l'algorithme d'élimination de Gauss permet de le transformer dans un système triangulaire supérieur à l'aide des opérations élémentaires de combinaison linéaire de lignes de la matrice. Comme on l'a vu dans la section précédente, une fois obtenu un système triangulaire supérieur, celui ci peut être résolu facilement par l'algorithme de substitution rétrograde. On montre l'algorithme d'élimination de Gauss sur un exemple.

Exemple 4.1. *On considère le système linéaire* $A\mathbf{x} = \mathbf{b}$ *où*

$$A = \begin{bmatrix} 2 & 1 & 0 \\ -4 & 3 & -1 \\ 4 & -3 & 4 \end{bmatrix}, \qquad \mathbf{b} = \begin{bmatrix} 4 \\ 2 \\ -2 \end{bmatrix}.$$

On note $r_1^{(1)}$, $r_2^{(1)}$ *et* $r_3^{(1)}$ *les trois équations du système :*

$$\begin{aligned} r_1^{(1)}: \quad & 2x_1 + x_2 = 4 \\ r_2^{(1)}: \quad & -4x_1 + 3x_2 - x_3 = 2 \\ r_3^{(1)}: \quad & 4x_1 - 3x_2 + 4x_3 = -2 \end{aligned}$$

On fait les opérations suivantes sur le système.
Étape 1 *:*

$$\begin{aligned} r_1^{(2)} &\leftarrow r_1^{(1)} && \Longrightarrow 2x_1 + x_2 = 4 \\ r_2^{(2)} &\leftarrow r_2^{(1)} - \underbrace{\left(\frac{\mathbf{-4}}{\mathbf{2}}\right)}_{l_{21}} r_1^{(1)} && \Longrightarrow 5x_2 - x_3 = 10 \\ r_3^{(2)} &\leftarrow r_3^{(1)} - \underbrace{\left(\frac{\mathbf{4}}{\mathbf{2}}\right)}_{l_{31}} r_1^{(1)} && \Longrightarrow -5x_2 + 4x_3 = -10. \end{aligned}$$

Étape 2 *:*

$$\begin{aligned} r_1^{(3)} &\leftarrow r_1^{(2)} && \Longrightarrow 2x_1 + x_2 = 4 \\ r_2^{(3)} &\leftarrow r_2^{(2)} && \Longrightarrow 5x_2 - x_3 = 10 \\ r_3^{(3)} &\leftarrow r_3^{(2)} - \underbrace{\left(\frac{\mathbf{-5}}{\mathbf{5}}\right)}_{l_{32}} r_2^{(2)} && \Longrightarrow 3x_3 = 0. \end{aligned}$$

On a ainsi obtenu un système triangulaire supérieur. On note que la matrice du système a été transformée de la façon suivante

$$A = A^{(1)} = \begin{bmatrix} 2 & 1 & 0 \\ -4 & 3 & -1 \\ 4 & -3 & 4 \end{bmatrix} \Longrightarrow A^{(2)} = \begin{bmatrix} 2 & 1 & 0 \\ 0 & 5 & -1 \\ 0 & -5 & 4 \end{bmatrix} \Longrightarrow A^{(3)} = \begin{bmatrix} 2 & 1 & 0 \\ 0 & 5 & -1 \\ 0 & 0 & 3 \end{bmatrix} = U.$$

Donc, à la première étape, la première colonne a été annulée au-dessous du premier élément tandis qu'à la deuxième étape, la deuxième colonne a été annulée au dessous du deuxième élément.

De plus, à chaque étape de l'algorithme, on peut sauvegarder dans une matrice les multiplicateurs l_{ij} *qu'on a utilisés dans les combinaisons linéaires*

des lignes pour annuler les colonnes de la matrice A. On met les éléments diagonaux égaux à 1 :

$$L^{(1)} = \begin{bmatrix} 1 & & \\ -\frac{4}{2} & & \\ \frac{4}{2} & & \end{bmatrix}, \implies L^{(2)} = \begin{bmatrix} 1 & & \\ -2 & 1 & \\ 2 & -\frac{5}{5} & \end{bmatrix}, \implies L^{(3)} = \begin{bmatrix} 1 & & \\ -2 & 1 & \\ 2 & -1 & 1 \end{bmatrix} = L$$

et on remarque que $LU = A$.

L'algorithme présenté dans l'exemple précédent est connu comme *algorithme d'élimination de Gauss*. Il porte, par une séquence précise d'opérations, sur une matrice triangulaire supérieure qu'on appelle U (en anglais "Upper"). De plus, si on sauvegarde tous les multiplicateurs utilisés dans les combinaisons linéaires des lignes, dans une matrice triangulaire inférieure L (en anglais "Lower") dont les éléments de la diagonale principale sont tous égaux à 1, on a

$$A = LU.$$

La matrice A est donc factorisée dans le produit de deux matrices triangulaires. Cette factorisation est appelée **factorisation** LU.

L'algorithme d'élimination de Gauss peut être écrite de façon systématique pour un système quelconque

Algorithme 4.3 : Algorithme d'élimination de Gauss (et factorisation LU)

Données : $A = \{a_{ij}\} \in \mathbb{R}^{n\times n}$, $\mathbf{b} = \{b_i\} \in \mathbb{R}^n$
Résultat : $U, L \in \mathbb{R}^{n\times n}$, $\mathbf{b}^{(n)} \in \mathbb{R}^n$
$A^{(1)} = A$;
pour $k = 1, \ldots, n-1$ **faire** `// étapes de l'algorithme`
  $l_{kk} = 1$;
  pour $i = k+1, \ldots, n$ **faire** `// boucle sur les lignes`
    $l_{ik} = \frac{a_{ik}^{(k)}}{a_{kk}^{(k)}}$;
    pour $j = k+1, \ldots, n$ **faire** `// boucle sur les colonnes`
      $a_{ij}^{(k+1)} = a_{ij}^{(k)} - l_{ik} a_{kj}^{(k)}$;
    fin
    $b_i^{(k+1)} = b_i^{(k)} - l_{ik} b_k^{(k)}$;
  fin
fin
$U = A^{(n)}$, $L = \{l_{ij}\}$;

À la fin de l'algorithme d'élimination de Gauss, il faut encore résoudre le système triangulaire supérieur $U\mathbf{x} = \mathbf{b}^{(n)}$, ce qui peut être fait simplement par l'algorithme de la substitution rétrograde.

On observe que, même si on ne calcule pas au cours de l'algorithme d'élimination de Gauss le terme de droite modifié $\mathbf{b}^{(n)}$, il peut toujours être reconstruit à partir de la matrice L.

En fait, imaginons qu'on ait calculé la factorisation LU de la matrice A et qu'on veuille résoudre le système linéaire

$$A\mathbf{x} = \mathbf{b} \qquad \Leftrightarrow \qquad LU\mathbf{x} = \mathbf{b}.$$

Si on introduit le vecteur auxiliaire $\mathbf{y} = U\mathbf{x}$, le système linéaire peut être résolut par les opérations suivantes :

$$\begin{cases} L\mathbf{y} = \mathbf{b} & \text{(système triangulaire inférieur } \rightsquigarrow \text{ substitution directe)} \\ U\mathbf{x} = \mathbf{y} & \text{(système triangulaire supérieur } \rightsquigarrow \text{ substitution rétrograde).} \end{cases}$$

Notez que le vecteur $\mathbf{y}$ correspond bien au vecteur $\mathbf{b}^{(n)}$ qu'on obtient à la fin de l'algorithme d'élimination de Gauss.

4.3 Élimination de Gauss avec pivoting

L'algorithme de Gauss n'arrive pas toujours au but. En fait, si on regarde l'Algorithme 4.3, il pourrait arriver qu'à l'étape k le coefficient $a_{kk}^{(k)}$ soit nul. Dans ce cas, on ne peut pas calculer le multiplicateur l_{ik} et l'Algorithme s'arrête. Les éléments $a_{kk}^{(k)}$ sont appelles **pivots**.

Toutefois, il est toujours possible d'effectuer des changements de lignes dans la matrice de sorte à pouvoir continuer l'algorithme. Cette technique est connue comme *méthode de pivoting*. On montre l'idée sur un exemple.

Exemple 4.2. *Appliquons l'algorithme d'élimination de Gauss à la matrice*

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 5 \\ 7 & 8 & 9 \end{bmatrix}.$$

Après la première étape de l'algorithme, on obtient les matrices suivantes

$$A^{(2)} = \begin{bmatrix} 1 & 2 & 3 \\ 0 & \boxed{0} & -1 \\ 0 & -6 & -12 \end{bmatrix}, \qquad L^{(1)} = \begin{bmatrix} 1 & & \\ 2 & & \\ 7 & & \end{bmatrix}.$$

Puisque le pivot $a_{22}^{(2)}$ est nul, l'algorithme d'élimination de Gauss s'arrête.

Toutefois, on pourrait essayer d'échanger la deuxième et la troisième ligne aussi bien dans la matrice $A^{(2)}$ que dans la matrice $L^{(2)}$. Ceci donne

$$\tilde{A}^{(2)} = \begin{bmatrix} 1 & 2 & 3 \\ 0 & -6 & -12 \\ 0 & 0 & -1 \end{bmatrix}, \qquad \tilde{L}^{(1)} = \begin{bmatrix} 1 & & \\ 7 & & \\ 2 & & \end{bmatrix}.$$

En ce moment, le nouveau pivot $\tilde{a}_{22}^{(2)}$ n'est plus nul et l'algorithme peut continuer. En fait, il n'y a plus rien à faire puisque la matrice $\tilde{A}^{(2)}$ et déjà sous forme triangulaire supérieure. On a donc la forme finale de la factorisation

$$U = A^{(3)} = \tilde{A}^{(2)} = \begin{bmatrix} 1 & 2 & 3 \\ 0 & -6 & -12 \\ 0 & 0 & -1 \end{bmatrix}, \qquad L = L^{(3)} = \begin{bmatrix} 1 & & \\ 7 & 1 & \\ 2 & 0 & 1 \end{bmatrix}.$$

Si on calcule maintenant le produit LU :

$$\underbrace{\begin{bmatrix} 1 & 0 & 0 \\ 7 & 1 & 0 \\ 2 & 0 & 1 \end{bmatrix}}_{L} \underbrace{\begin{bmatrix} 1 & 2 & 3 \\ 0 & -6 & -12 \\ 0 & 0 & -1 \end{bmatrix}}_{U} = \begin{bmatrix} 1 & 2 & 3 \\ 7 & 8 & 9 \\ 2 & 4 & 5 \end{bmatrix}$$

on retrouve la matrice de départ, avec la deuxième et troisième ligne interchangées. On retrouve donc la matrice de départ avec la même permutation de ligne qu'on a effectué pendant l'algorithme.

On introduit la ***matrice de permutation***

$$P = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}.$$

Cette matrice est obtenue en partant de la matrice identité et en effectuant une permutation entre la deuxième et troisième ligne. Il est facile de voir que

$$LU = PA.$$

L'exemple précédent montre qu'en faisant une permutation de lignes chaque fois qu'un pivot est nul, on arrive en général à terminer l'algorithme d'élimination de Gauss et obtenir deux matrices triangulaires L et U. Leur produit est égal à PA, où la matrice de permutation P mémorise toute permutation qu'on a effectuée pendant l'algorithme.

En général, si un pivot $a_{kk}^{(k)}$ est zéro, on peut décider de permuter la ligne k avec n'importe quelle ligne $i > k$, pourvu que $a_{ik}^{k} \neq 0$. Pour rendre l'algorithme le plus stable possible, on peut choisir la ligne r pour laquelle $|a_{rk}^{(k)}| \geq |a_{ik}^{(k)}|$, pour tout $i = k, \ldots, n$, et échanger *toujours* la ligne k avec la ligne r, même si le pivot $a_{kk}^{(k)}$ n'est pas zéro. Une formulation rigoureuse de l'algorithme d'élimination de Gauss avec pivoting est donnée ci-dessous.

Algorithme 4.4 : Algorithme d'élimination de Gauss avec pivoting

Données : $A = \{a_{ij}\} \in \mathbb{R}^{n\times n}$, $\mathbf{b} = \{b_i\} \in \mathbb{R}^n$
Résultat : $U, L, P \in \mathbb{R}^{n\times n}$, $\mathbf{b}^{(n)} \in \mathbb{R}^n$
$A^{(1)} = A$, $P =$ matrice identité;
pour $k = 1, \ldots, n-1$ **faire**
  trouver r tel que $|a_{rk}^{(k)}| = \max_{i=k,\ldots,n} |a_{ik}^{(k)}|$;
  échanger la ligne k avec la ligne r dans les matrices $A^{(k)}$, L et P, ainsi que dans le vecteur $\mathbf{b}^{(k)}$;
  pour $i = k+1, \ldots, n$ **faire**
    $l_{ik} = \frac{a_{ik}^{(k)}}{a_{kk}^{(k)}}$;
    pour $j = k+1, \ldots, n$ **faire**
      $a_{ij}^{(k+1)} = a_{ij}^{(k)} - l_{ik} a_{kj}^{(k)}$;
    fin
    $b_i^{(k+1)} = b_i^{(k)} - l_{ik} b_k^{(k)}$;
  fin
  $l_{kk} = 1$;
fin
$U = A^{(n)}$, $L = \{l_{ij}\}$;

On conclut cette section par un résultat théorique important

Theorème 4.1. *Toute matrice $A \in \mathbb{R}^{n\times n}$ non singulière admet une factorisation*

$$PA = LU$$

où L est une matrice triangulaire inférieure, U une matrice triangulaire supérieure et P une matrice de permutation.

Ce théorème nous garantit en particulier que l'algorithme 4.4 se termine toujours. L'algorithme d'élimination de Gauss avec pivot est donc une bonne méthode pour résoudre un système linéaire.

Supposons maintenant qu'on ait calculé la factorisation (L, U, P) d'une matrice A et qu'on veuille résoudre le système linéaire $A\mathbf{x} = \mathbf{b}$. On remarque que

$$A\mathbf{x} = \mathbf{b} \quad \Leftrightarrow \quad PA\mathbf{x} = P\mathbf{b} \quad \Leftrightarrow \quad LU\mathbf{x} = P\mathbf{b}.$$

On peut donc résoudre le système linéaire par les opérations suivantes :

$$\begin{cases} L\mathbf{y} = P\mathbf{b} & \text{(système triangulaire inférieur } \rightsquigarrow \text{ substitution directe)} \\ U\mathbf{x} = \mathbf{y} & \text{(système triangulaire supérieur } \rightsquigarrow \text{ substitution rétrograde).} \end{cases}$$

L'algorithme d'élimination de Gauss avec pivoting est à la base de la commande `Python` `numpy.linalg.solve` ou `scipy.linalg.solve`, c'est-à-dire lorsqu'on résout un système linéaire en `Python` par la commande

`x=scipy.linalg.solve(A,b)`, c'est bien cet algorithme qui est utilisé (ou bien des versions plus performantes).

On peut aussi calculer explicitement la factorisation LU avec pivoting par la commande `P,L,U = scipy.linalg.lu(A)`.

Pour la matrice de l'exemple 4.2 on trouve

```
import numpy as np
import scipy.linalg as sc

A = np.array([[1, 2, 3], [2, 4, 5], [7, 8, 9]]);
P,L,U = sc.lu(A)

print("L: ",L)
print("U: ",U)
print("P: ",P)

# OUTPUT
# L:  [[1.         0.         0.        ]
#  [0.28571429 1.         0.        ]
#  [0.14285714 0.5        1.        ]]
# U:  [[7.         8.         9.        ]
#  [0.         1.71428571 2.42857143]
#  [0.         0.         0.5       ]]
# P:  [[0. 0. 1.]
#  [0. 1. 0.]
#  [1. 0. 0.]]
```

`Python` a donc utilisé une permutation différente que celle qu'on a utilisée dans l'exemple 4.2. On voit cependant que le produit LU donne

```
print("L*U: ",np.dot(L,U))

# OUTPUT
# L*U:  [[7. 8. 9.]
#  [2. 4. 5.]
#  [1. 2. 3.]]
```

qui est bien la matrice A avec une permutation entre la première et troisième ligne.

4.4 Occupation de mémoire et fill-in

Dans beaucoup d'applications de l'ingénierie, on a à résoudre des systèmes linéaires qui peuvent être de taille importante. Toutefois, la matrice du système a très souvent une structure particulière. Ceci a des conséquences aussi bien sur la façon de stocker la matrice dans la mémoire de l'ordina-

teur que sur le coût de calcul pour la résolution du système linéaire. Voyons quelques exemples.

Définition 4.1 (**Matrice pleine**). *On dit qu'une matrice $A \in \mathbb{R}^{n\times n}$ est* pleine *si le nombre d'éléments non nuls est de l'ordre de n^2 (en gros tous ou presque tous les éléments de la matrice sont non nuls).*

On dit que l'occupation de mémoire d'une matrice pleine est $O(n^2)$. Si on utilise une représentation de nombres en virgule flottante en double précision, chaque nombre utilise 64 bit de mémoire, ce qui correspond à 8 bytes (chaque byte est constitué de 8 bits).

Un ordinateur portable a de nos jours une mémoire RAM de quelque Giga-byte. Pour fixer les idées, considérons un ordinateur avec $4Gb = 4 \cdot 10^9$ *bytes* de RAM. Dans un tel ordinateur, on pourra stocker au plus $5 * 10^8$ nombres en virgule flottante double précision et la taille maximale d'une matrice pleine qu'on peut stocker est $n = \sqrt{5 * 10^8} \approx 20000$.

On aura donc du mal à résoudre un système linéaire avec plus de 20000 équations et 20000 inconnues.

Définition 4.2 (**Matrice creuse**). *On dit que $A \in \mathbb{R}^{n\times n}$ est une* matrice creuse *si le nombre d'éléments non nuls est de l'ordre de n.*

Puisqu'on sait à priori que beaucoup d'éléments sont nuls, on peut stocker en mémoire seulement les éléments non nuls de la matrice. L'occupation de mémoire d'une matrice creuse est $O(n)$. Dans ce cas, on peut aborder sur un ordinateur portable de $4Gb$ de mémoire, des systèmes linéaires de taille beaucoup plus grande que dans le cas d'une matrice pleine, même avec quelques millions d'inconnues.

En `Python` , on peut stocker des matrices creuses à l'aide de la commande `scipy.sparse.coo_matrix` (d'autres formats que `coo` sont aussi disponibles). Par exemple, pour stocker la matrice

$$A = \begin{pmatrix} 1 & 0 & 0 & 5 \\ -1 & 0 & 1 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 1 & 0 & -1 \end{pmatrix}$$

on peut utiliser les commandes

```
import scipy.sparse as sp
A=np.array([[1,0,0,5],[-1,0,1,0],[0,0,3,0],[0,1,0,-1]])
A=sp.coo_matrix(A)

print(A)

# OUTPUT
#   (0, 0)        1
```

```
#   (0, 3)        5
#   (1, 0)        -1
#   (1, 2)        1
#   (2, 2)        3
#   (3, 1)        1
#   (3, 3)        -1
```

On voit bien que `Python` a mémorisé seulement les éléments non nuls de la matrice ainsi que leur position.

Définition 4.3 (**Matrice bande**). *On dit que* $A \in \mathbb{R}^{n\times n}$ *est une* matrice bande de largeur K *si*

$$A_{ij} = 0, \quad \textit{lorsque } |j-i| > K.$$

Chaque ligne de la matrice contient donc au plus $2K+1$ *éléments non nuls.*

Une matrice bande est un cas particulier de matrice creuse où les éléments non nuls sont groupés autour de la diagonale de la matrice. Si on stocke en mémoire seulement les éléments non nuls, l'occupation de mémoire sera $O(Kn)$. (On peut compter exactement le nombre d'éléments non nuls mais ceci n'est pas intéressant.)

La commande `Python` `matplotlib.pyplot.spy(A)` permet de visualiser les éléments non nuls d'une matrice. La Figure 4.1 montre un exemple de matrice pleine, creuse et bande.

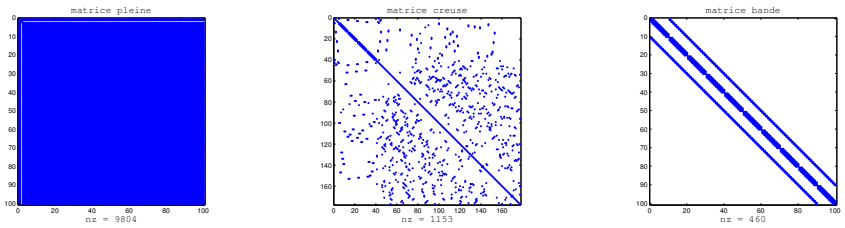


FIGURE 4.1 – Exemple de matrice pleine (gauche), creuse (centre), bande (droite). Visualisation obtenue par la commande `spy` de `Python` .

On se demande maintenant quelle est l'occupation de mémoire requise pour stocker la factorisation LU d'une matrice pleine, creuse ou bande.

La Figure 4.2 montre les éléments non nuls des matrices L et U qui correspondent aux matrices montrées dans la Figure 4.1. Dans les graphiques en Fig. 4.2, on a superposé la visualisation des matrices L et U. La matrice L correspond à la partie triangulaire inférieure et U à la partie triangulaire supérieure des matrices visualisées.

D'après la Figure 4.2, on voit bien que la factorisation d'une matrice pleine est encore une matrice pleine. Par contre, on voit que la factorisation LU d'une matrice creuse a beaucoup plus d'éléments non nuls que la matrice de départ. Ce phénomène est connu comme *remplissage* (en anglais *fill in*).

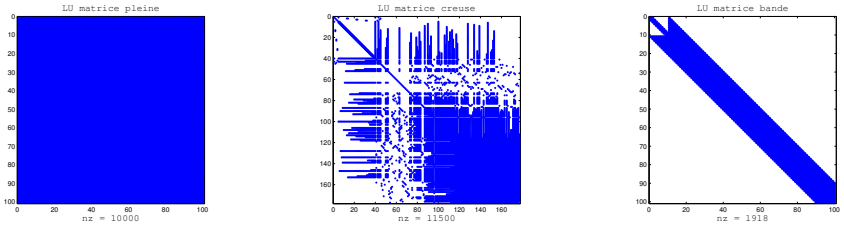


FIGURE 4.2 – Éléments non nuls de la factorisation LU des matrices montrées en Figures 4.1. L correspond à la partie triangulaire inférieure et U à la partie triangulaire supérieure des matrices visualisées. Gauche : matrice pleine ; centre : matrice creuse ; droite : matrice bande.

En particulier, la factorisation de la matrice bande a préservé la structure bande mais a rempli complètement l'intérieur de la bande. Par contre, la factorisation de la matrice creuse, montrée en Figure 4.1-centre, n'a pas du tout préservé la structure de la matrice et devient presque pleine vers le coin "sud-est". Si d'un coté l'occupation de mémoire d'une matrice creuse est $O(n)$ et on peut stocker des matrices de taille importante ($n \approx 10^7$) sur un ordinateur portable, l'occupation de mémoire de sa factorisation LU est $O(n^2)$ en général ! Il faut donc faire très attention lorsqu'on utilise la méthode de factorisation pour une matrice creuse car on risque de ne pas avoir assez de mémoire à disposition si n est trop grand.

4.5 Coût de calcul de la factorisation LU

Un ordinateur portable travaille de nos jours à une fréquence de quelques Giga-herz. Il peut donc faire environs 10^9 opérations par seconde. On se pose la question de combien de temps prennent certaines opérations matricielles sur un ordinateur portable. Considérons deux exemples :

— **Produit scalaire entre deux vecteurs** : soient $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ deux vecteurs de n éléments réels. Le produit scalaire

$$\mathbf{x} \cdot \mathbf{y} = \mathbf{y}^T \mathbf{x} = \sum_{i=1}^{n} x_i y_i$$

requière n multiplications et $n-1$ additions. Le nombre d'opérations nécessaires est donc de l'ordre de n (plus précisément $2n-1$) et on dit que le coût de calcul est $O(n)$.
Si on prend $n = 1000$, le coût est $O(1000)$ et, puisque l'ordinateur peut faire 10^9 opérations par seconde, le temps nécessaire pour calculer le produit scalaire sera d'environ 10^{-6} *sec*, c'est-à-dire très petit.

— **Produit entre une matrice pleine et un vecteur** : soient $\mathbf{x} \in \mathbb{R}^n$

et $A \in \mathbb{R}^{n\times n}$. On veut calculer

$$\mathbf{y} = A\mathbf{x}, \qquad \Longrightarrow \qquad y_i = \sum_{j=1}^{n} A_{ij}x_j, \;\; i = 1, \ldots, n.$$

Pour calculer chaque élément y_i du vecteur $\mathbf{y}$ il faut faire n multiplications et $(n-1)$ additions. Puisqu'on doit calculer les n éléments du vecteur $\mathbf{y}$, le coût de calcul de cette opération est $O(n^2)$.
Si $n = 1000$, sur notre ordinateur portable cette opération prend $10^{-3}\,sec$, ce qui est encore un temps très petit mais plus important que le produit scalaire entre deux vecteurs.

On se demande quel est le coût pour calculer la factorisation LU d'une matrice pleine.

Prenons l'algorithme 4.3 (ou bien la version avec pivoting 4.4). L'instruction à exécuter maintes fois est

$$a_{ij}^{(k+1)} = a_{ij}^{(k)} - l_{ik}a_{kj}^{(k)}$$

qui contient une multiplication et une soustraction, donc 2 opérations élémentaires. Cette instruction est à l'intérieur de trois boucles emboîtées (sur k, i et j respectivement) où les indices varient de 1 à n. Le nombre total d'opérations élémentaires est donc de l'ordre de n^3. On dit que le coût de calcul de la factorisation LU d'une matrice pleine est $O(n^3)$.

Si on prend $n = 1000$, sur notre ordinateur portable, la factorisation LU d'une matrice pleine prend environ $1sec$.

Par contre, si on prend $n = 10^4$, qui est à la limite de la mémoire à disposition, le coût de calcul de la factorisation LU est de l'ordre de 10^{12} opérations et le temps de calcul devient $\sim 10^3\,sec$, c'est-à-dire plus d'un quart d'heure.

Pour des matrices de grosse traille, le coût de calcul de la factorisation LU peut devenir important.

Par contre, si on a une matrice bande de largeur de bande K, dans l'algorithme d'élimination de Gauss 4.3 on peut limiter les deux boucles internes à K itérations et le coût de calcul devient $O(nK^2)$.

On résume les résultats principaux sur le coût de calcul et l'occupation de mémoire pour la factorisation LU dans le tableau suivant

	coût calcul	occupation mémoire
matrice pleine	$O(n^3)$	$O(n^2)$
matrice bande (largeur K)	$O(nK^2)$	$O(nK)$

4.6 Effet des erreurs d'arrondis

La méthode d'élimination de Gauss (avec pivoting si nécessaire) nous permet d'arriver à la solution exacte du système linéaire en un nombre fini d'opérations.

Toutefois, il ne faut jamais oublier que l'ordinateur ne travaille pas en arithmétique exacte mais qu'il utilise la représentation en virgule flottante des nombres réels qui introduit de petites erreurs d'arrondis de l'ordre de 10^{-16} (si on travail en double précision).

On se demande donc quel est l'effet de ces erreurs d'arrondis sur la solution obtenue par l'ordinateur. On peut formaliser la question de la façon suivante.

On souhaite résoudre le système linéaire

$$A\mathbf{x} = \mathbf{b}$$

qu'on appellera par la suite le "système exact". La représentation dans l'ordinateur du vecteur $\mathbf{b}$ et de la matrice A n'est pas exacte à cause des erreurs d'arrondis. On peut dire donc que dans l'ordinateur, on résout un système légèrement modifié

$$\hat{A}\hat{\mathbf{x}} = \hat{\mathbf{b}}$$

qu'on appelle par la suite le "système perturbé". Le terme de droite $\hat{\mathbf{b}}$ est une perturbation due aux erreurs d'arrondis du vrai terme $\mathbf{b}$. On peut dire que

$$\hat{b}_i = b_i(1+\epsilon_i), \qquad \text{où } \epsilon_i \text{ est de l'ordre de } 10^{-16}.$$

De même, on aura

$$\hat{A}_{ij} = A_{ij}(1+\eta_{ij}), \qquad \text{avec } \eta_{ij} \text{ de l'ordre de } 10^{-16}.$$

On vise à estimer la distance entre la solution exacte $\mathbf{x}$ et la solution obtenue par l'ordinateur $\hat{\mathbf{x}}$. L'exemple suivant montre que même de toutes petites perturbations sur le terme de droite peuvent amener à de mauvaises solutions.

Exemple 4.3. *Considérons le système linéaire "exact"*

$$A\mathbf{x} = \mathbf{b} \qquad \Longleftrightarrow \qquad \begin{bmatrix} 1 & 10^{-16} \\ 1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

dont la solution exacte est $(x_1, x_2) = (1, 0)$*, et le système perturbé*

$$\hat{A}\hat{\mathbf{x}} = \hat{\mathbf{b}} \qquad \Longleftrightarrow \qquad \begin{bmatrix} 1 & 10^{-16} \\ 1 & 0 \end{bmatrix} \begin{bmatrix} \hat{x}_1 \\ \hat{x}_2 \end{bmatrix} = \begin{bmatrix} 1+10^{-16} \\ 1 \end{bmatrix}.$$

On a donc perturbé seulement la première composante du terme de droite. La solution du système perturbé est $(\hat{x}_1, \hat{x}_2) = (1, 1)$ *et la deuxième composante de la solution est complètement fausse ! La petite perturbation de* 10^{-16} *sur le terme de droite a été énormément amplifiée au niveau de la solution !*

Pour analyser l'erreur $\mathbf{x} - \hat{\mathbf{x}}$, il nous faut d'abord introduire quelques notions. Soit $\mathbf{x} \in \mathbb{R}^n$ un vecteur et $A \in \mathbb{R}^{n\times n}$ une matrice. On rappelle que la norme euclidienne d'un vecteur est $\|\mathbf{x}\| = \sqrt{\sum_{i=1}^n x_i^2}$.

En particulier, on veut estimer l'*erreur relative*

$$\frac{\|\hat{\mathbf{x}} - \mathbf{x}\|}{\|\mathbf{x}\|}.$$

La notion de norme peut se généraliser à des matrices.

Définition 4.4 (**Norme d'une matrice**). *Soit $A \in \mathbb{R}^{m\times n}$ une matrice réelles (non nécessairement carrée). On définit la* norme de A *par*

$$\|A\| = \sup_{\substack{\mathbf{x}\in\mathbb{R}^n \\ \mathbf{x}\neq\mathbf{0}}} \frac{\|A\mathbf{x}\|}{\|\mathbf{x}\|}.$$

La définition précédente implique en particulier que

$$\|A\mathbf{x}\| \leq \|A\|\|\mathbf{x}\|, \qquad \forall \mathbf{x} \in \mathbb{R}^n.$$

Soit $B \in \mathbb{R}^{n\times n}$ une matrice carrée, réelle et symétrique. On indique par $\lambda_i(B)$, $i = 1, \ldots, n$, les valeurs propres de B et par $\lambda_{max}(B) = \max_{i=1,\ldots n} \lambda_i(B)$ et $\lambda_{min}(B) = \min_{i=1,\ldots n} \lambda_i(B)$.

On a la caractérisation suivante de la norme d'une matrice.

Lemme 4.2. *Pour une matrice $A \in \mathbb{R}^{m\times n}$ quelconque*

$$\|A\| = \sqrt{\lambda_{max}(A^T A)}.$$

De plus, si A est carrée et inversible, alors

$$\|A^{-1}\| = \sqrt{\lambda_{max}(A^{-T}A^{-1})} = \frac{1}{\sqrt{\lambda_{min}(A^T A)}}.$$

Définition 4.5 (**Nombre de conditionnement**). *Soit $A \in \mathbb{R}^{n\times n}$ une matrice carrée et inversible. On appelle* nombre de conditionnement de A, *noté $\mathcal{K}(A)$, la quantité*

$$\mathcal{K}(A) = \|A^{-1}\|\|A\| = \frac{\sqrt{\lambda_{max}(A^T A)}}{\sqrt{\lambda_{min}(A^T A)}}.$$

On note que si A est une matrice symétrique, c'est-à-dire $A^T = A$, alors

$$\lambda_i(A^T A) = \lambda_i(A^2) = \lambda_i(A)^2, \qquad i = 1, \ldots, n.$$

De plus, si A est symétrique et définie positive, alors toutes les valeurs propres sont réelles et positives. Donc

Lemme 4.3. *Si $A \in \mathbb{R}^{n\times n}$ est une matrice symétrique et définie positive, alors*

$$\mathcal{K}(A) = \frac{\lambda_{max}(A)}{\lambda_{min}(A)}.$$

On a maintenant tous les outils pour analyser l'erreur relative $\frac{\|\hat{\mathbf{x}}-\mathbf{x}\|}{\|\mathbf{x}\|}$. On se limite à analyser le cas où la perturbation est seulement sur le terme de droite (donc $\eta_{ij} = 0$).

On a

$$\begin{array}{l} A\mathbf{x} = \mathbf{b} \\ \hat{A}\hat{\mathbf{x}} = \hat{\mathbf{b}} \end{array} \qquad \Longrightarrow \qquad A(\hat{\mathbf{x}} - \mathbf{x}) = \hat{\mathbf{b}} - \mathbf{b}$$

et donc

$$\hat{\mathbf{x}} - \mathbf{x} = A^{-1}(\hat{\mathbf{b}} - \mathbf{b}), \qquad \Longrightarrow \|\hat{\mathbf{x}} - \mathbf{x}\| \leq \|A^{-1}\|\|\hat{\mathbf{b}} - \mathbf{b}\|,$$

où

$$\|\hat{\mathbf{b}} - \mathbf{b}\| = \sqrt{\sum_{i=1}^{n} b_i^2\epsilon_i^2} \leq \max_{i=1,\ldots,n} |\epsilon_i|\|\mathbf{b}\|.$$

De l'autre coté,

$$\|\mathbf{b}\| \leq \|A\|\|\mathbf{x}\|, \qquad \Longrightarrow \qquad \frac{1}{\|\mathbf{x}\|} \leq \frac{\|A\|}{\|\mathbf{b}\|}.$$

Si on multiplie les deux inégalités on obtient

$$\frac{\|\hat{\mathbf{x}} - \mathbf{x}\|}{\|\mathbf{x}\|} \leq \|A\|\|A^{-1}\|\frac{\|\hat{\mathbf{b}} - \mathbf{b}\|}{\|\mathbf{b}\|} \leq \mathcal{K}(A) \max_{i=1,\ldots,n} |\epsilon_i|.$$

On a donc montré le résultat suivant :

Lemme 4.4. *Soit $\mathbf{x} \in \mathbb{R}^n$ la solution du système linéaire $A\mathbf{x} = \mathbf{b}$ et $\hat{\mathbf{x}}$ la solution du système $A\mathbf{x} = \hat{\mathbf{b}}$ où $\hat{b}_i = b_i(1 + \epsilon_i)$, $i = 1, \ldots, n$. On note $\epsilon_{max} = \max_{i=1,\ldots,n} |\epsilon_i|$. Alors*

$$\frac{\|\hat{\mathbf{x}} - \mathbf{x}\|}{\|\mathbf{x}\|} \leq \mathcal{K}(A)\,\epsilon_{max}. \tag{4.4}$$

L'inégalité (4.4) montre que le nombre de conditionnement de la matrice A joue le rôle de *facteur d'amplification* des erreurs d'arrondis introduites dans la représentation en virgule flottante du terme de droite.

Même si ces erreurs sont très petites, en général, certaines matrices ont un nombre de conditionnement très élevé qui peut conduire à des solutions totalement imprécises.

Exemple 4.4. *Reprenons l'exemple 4.3 et calculons le nombre de conditionnement de la matrice A. On a*

$$A^TA = \begin{pmatrix} 1 & 1 \\ 10^{-16} & 0 \end{pmatrix} \begin{pmatrix} 1 & 10^{-16} \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} 2 & 10^{-16} \\ 10^{-16} & 10^{-32} \end{pmatrix}$$

et

$$\det(A^TA - \lambda I) = \lambda^2 - (2 + 10^{-32})\lambda + 10^{-32}.$$

Les deux valeurs propres de A^TA sont

$$\lambda_{1,2}(A^TA) = \frac{1}{2}(2 + 10^{-32} \pm \sqrt{4 + 10^{-64}})$$

et donc

$$\lambda_{max}(A^TA) \approx 2, \qquad \lambda_{min}(A^TA) \approx \frac{1}{2}10^{-32}.$$

On en conclut que le nombre de conditionnement de A est

$$\mathcal{K}(A) = \sqrt{\frac{\lambda_{max}(A^TA)}{\lambda_{min}(A^TA)}} \approx 2 * 10^{16}$$

ce qui explique les résultats trouvés dans l'exemple 4.3.

Finalement, on mentionne aussi le résultat dans le cas de perturbations aussi bien sur le terme de droite que sur la matrice :

Lemme 4.5. *Soit $\mathbf{x} \in \mathbb{R}^n$ la solution du système linéaire $A\mathbf{x} = \mathbf{b}$ et $\hat{\mathbf{x}}$ la solution du système $\hat{A}\mathbf{x} = \hat{\mathbf{b}}$ où $\hat{b}_i = b_i(1 + \epsilon_i)$ et $\hat{A} = A + N$, $N_{ij} = \eta_{ij}A_{ij}$, $i, j = 1, \ldots, n$. On note $\epsilon_{max} = \max_{i=1,\ldots,n} |\epsilon_i|$ et $\eta_{max} = \max_{i,j=1,\ldots,n} |\eta_{ij}|$. Si $\eta_{max} < 1/(\sqrt{n}\mathcal{K}(A))$ alors*

$$\frac{\|\hat{\mathbf{x}} - \mathbf{x}\|}{\|\mathbf{x}\|} \leq \frac{\mathcal{K}(A)}{1 - \sqrt{n}\eta_{max}\mathcal{K}(A)} \left(\sqrt{n}\eta_{max} + \epsilon_{max}\right). \tag{4.5}$$

Chapitre 5

Systèmes linéaires – méthodes itératives

Dans le chapitre précédent, on a vu une méthode qui peut être utilisée pour la résolution d'un système linéaire $A\mathbf{x} = \mathbf{b}$, notamment la méthode d'élimination de Gauss avec pivoting. Toutefois, on a aussi vu que cette méthode peut demander un coût de calcul ou une occupation de mémoire excessif dans le cas d'une matrice pleine de grosse taille.

On étudie dans ce chapitre une famille de méthodes, dites *méthodes itératives*, alternatives aux méthodes directes du chapitre précédent qui peuvent réduire ce problème dans certains cas. L'idée est de construire une suite de vecteurs $\mathbf{x}^{(k)}$ qui, à la limite, converge vers la solution $\mathbf{x}$ du système $A\mathbf{x} = \mathbf{b}$:

$$\lim_{k\to\infty} \mathbf{x}^{(k)} = \mathbf{x}.$$

5.1 Méthode de Richardson

Une procédure générale pour construire des méthodes itératives consiste à réécrire le système $A\mathbf{x} = \mathbf{b}$ sous la forme équivalente

$$P\mathbf{x} = (P - A)\mathbf{x} + \mathbf{b}$$

où P est une matrice *inversible* quelconque. Ensuite, étant donné un vecteur initial $\mathbf{x}^{(0)}$ arbitraire, on construit les itérations

$$P\mathbf{x}^{(k+1)} = (P - A)\mathbf{x}^{(k)} + \mathbf{b}, \qquad k = 0, 1, \ldots$$

Clairement, si la suite $\{\mathbf{x}^{(k)}\}_{k\geq 0}$ converge vers un vecteur $\mathbf{y}$, $\lim_{k\to\infty} \mathbf{x}^{(k)} = \mathbf{y}$, alors le vecteur limite $\mathbf{y}$ doit satisfaire l'équation

$$P\mathbf{y} = (P - A)\mathbf{y} + \mathbf{b} \qquad \Longleftrightarrow \qquad A\mathbf{y} = \mathbf{b}$$

et donc la valeur limite est bien la solution du système linéaire en question.

Le système précédent peut être aussi écrit sous la forme équivalente

$$P(\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}) = \mathbf{r}^{(k)}, \qquad \mathbf{r}^{(k)} = \mathbf{b} - A\mathbf{x}^{(k)}. \tag{5.1}$$

Les méthodes de la forme (5.1) sont dites *méthodes de Richardson*. La matrice P est appelée *préconditionneur* (ou *matrice de préconditionnement*) et le vecteur $\mathbf{r}^{(k)}$ est appelé *résidu* à l'étape k. Si $\mathbf{r}^{(k)} = 0$, alors $\mathbf{x}^{(k)}$ est la solution exacte du système. Si, par contre, $\mathbf{r}^{(k)} \neq 0$, il peut être pris comme une mesure de combien le vecteur $\mathbf{x}^{(k)}$ est loin de la solution exacte.

L'algorithme suivant donne une implémentation possible des méthodes de Richardson.

Algorithme 5.1 : Méthode de Richardson (sans critère d'arrêt)

Étant donnés $\mathbf{x}^{(0)}$ et $P \in \mathbb{R}^{n\times n}$ inversible;
pour $k = 0, 1, \ldots$ **faire**
  calculer $\mathbf{r}^{(k)} = \mathbf{b} - A\mathbf{x}^{(k)}$;
  résoudre le système linéaire $P\mathbf{z}^{(k)} = \mathbf{r}^{(k)}$;
  calculer $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \mathbf{z}^{(k)}$;
fin

L'algorithme précédent est encore incomplet car il faudra ajouter un critère d'arrêt pour les itérations. On discutera de ça dans la section 5.4.

L'étape 2 de l'algorithme entraîne la résolution d'un système linéaire. Pour que cette approche soit attractive, le système $P\mathbf{z}^{(k)}$ doit être beaucoup plus simple à résoudre que le système de départ $A\mathbf{x} = \mathbf{b}$. Ceci est le cas si, par exemple, la matrice P est diagonale ou triangulaire.

On remarque que cette approche peut être vue comme une *méthode de point fixe* (comme celles qu'on a vues au Chapitre 1) : on réecrit l'équation $\mathbf{f}(\mathbf{x}) = A\mathbf{x} - \mathbf{b} = \mathbf{0}$ sous une forme équivalente

$$\mathbf{x} = \boldsymbol{\phi}(\mathbf{x}) = P^{-1}[(P - A)\mathbf{x} + \mathbf{b}]$$

et on applique les itérations de point fixe $\mathbf{x}^{(k+1)} = \boldsymbol{\phi}(\mathbf{x}^{(k)})$.

5.1.1 Coût de calcul

Analysons le coût de chaque étape de l'algorithme 5.1 dans le cas d'une matrice A pleine. Les opérations les plus coûteuses sont :

- le calcul de $A\mathbf{x}^{(k)}$ à l'étape 1, qui coûte $O(n^2)$ si la matrice A est pleine.
- La résolution du système linéaire $P\mathbf{z}^{(k)} = \mathbf{r}^{(k)}$ à l'étape 2. Le coût de cette étape dépend fortement du choix de la matrice de préconditionnement P. Par exemple, si on prend une matrice diagonale, le coût de calcul sera $O(n)$ alors que si on prend une matrice triangulaire, le coût sera $O(n^2)$.

En général, si on choisit bien la matrice P, l'opération la plus coûteuse à chaque itération de la méthode est le produit matrice-vecteur $A\mathbf{x}^{(k)}$. Cela est une caractéristique typique des méthodes itératives.

On rappelle que le coût de calcul de la méthode d'élimination de Gauss est $O(n^3)$ pour une matrice pleine. On voit donc bien qu'une méthode itérative sera plus avantageuse par rapport à la méthode directe d'élimination de Gauss si on arrive à une solution approchée très précise en beaucoup moins que n itérations. Le coût total sera donc plus petit que $O(n^3)$! Si, de plus, la matrice est creuse, alors le coût de calcul de l'étape 1 est $O(n)$ au lieu de $O(n^2)$ et une méthode itérative est fort probablement plus avantageuse qu'une méthode directe.

La vitesse de convergence des méthodes itératives sera analysée dans la Section 5.3.

5.2 Méthodes de Jacobi et Gauss-Seidel

Les méthodes de Jacobi et Gauss-Seidel sont des cas particuliers de la méthode de Richardson.

Dans la **méthode de Jacobi**, on prend comme matrice de préconditionnement la matrice P formée de la seule diagonale principale de A (en `Python P=diag(diag(A))`) :

$$P = \begin{bmatrix} a_{11} & & & \\ & a_{22} & & \\ & & \ddots & \\ & & & a_{nn} \end{bmatrix}.$$

L'itération $k+1$ de la méthode de Jacobi est donc

$$\begin{bmatrix} a_{11} & & & \\ & a_{22} & & \\ & & \ddots & \\ & & & a_{nn} \end{bmatrix} \begin{bmatrix} x_1^{(k+1)} \\ x_2^{(k+1)} \\ \vdots \\ x_n^{(k+1)} \end{bmatrix} = - \begin{bmatrix} 0 & a_{12} & \cdots & a_{1n} \\ a_{21} & 0 & & \vdots \\ \vdots & & \ddots & \\ & \cdots & a_{n,n-1} & 0 \end{bmatrix} \begin{bmatrix} x_1^{(k)} \\ x_2^{(k)} \\ \vdots \\ x_n^{(k)} \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}.$$

La i-ème composante du vecteur $\mathbf{x}^{(k+1)}$ peut être calculée par la formule

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{\substack{j=1 \\ j \neq i}}^{n} a_{ij} x_j^{(k)} \right), \qquad i = 1, \ldots, n \tag{5.2}$$

pourvu que $a_{ii} \neq 0$ pour tout $i = 1, \ldots, n$.

Dans la **méthode de Gauss-Seidel**, par contre, on prend comme matrice de préconditionnement la matrice P formée de la partie triangulaire

inférieure de A, y compris sa diagonale (en `Python P=tril(A)`) :

$$P = \begin{bmatrix} a_{11} & & & \\ a_{21} & a_{22} & & \\ \vdots & & \ddots & \\ a_{n1} & \cdots & \cdots & a_{nn} \end{bmatrix}.$$

L'itération $k+1$ de la méthode de Gauss-Seidel est donc

$$\begin{bmatrix} a_{11} & & & \\ a_{21} & a_{22} & & \\ \vdots & & \ddots & \\ a_{n1} & \cdots & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1^{(k+1)} \\ x_2^{(k+1)} \\ \vdots \\ x_n^{(k+1)} \end{bmatrix} = - \begin{bmatrix} 0 & a_{12} & \cdots & a_{1n} \\ & 0 & \ddots & \vdots \\ & & \ddots & a_{n-1,n} \\ & & & 0 \end{bmatrix} \begin{bmatrix} x_1^{(k)} \\ x_2^{(k)} \\ \vdots \\ x_n^{(k)} \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}.$$

Ce système linéaire peut être résolu par l'algorithme de substitution directe, la i-ème composante du vecteur $\mathbf{x}^{(k+1)}$ étant donnée par la formule

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right), \qquad i = 1, \ldots, n \quad (5.3)$$

pourvu que $a_{ii} \neq 0$ pour tout $i = 1, \ldots, n$.

La méthode de Gauss-Seidel est donc très similaire à la méthode de Jacobi, sauf que dans le calcul de la nouvelle composante $x_i^{(k+1)}$ on utilise les composantes précédentes $x_j^{(k+1)}$, $j < i$, déjà calculées.

5.3 Analyse de convergence

On se pose maintenant la question si une méthode itérative converge vers la solution exacte du système linéaire $A\mathbf{x} = \mathbf{b}$. Prenons la famille des méthodes de Richardson

$$P\mathbf{x}^{(k+1)} = (P - A)\mathbf{x}^{(k)} + \mathbf{b} \quad (5.4)$$

avec matrice de préconditionnement P inversible. Si on arrête la méthode après k itérations et on prend $\mathbf{x}^{(k)}$ comme approximation de la solution exacte, l'erreur commise par la méthode est $\mathbf{e}^{(k)} = \mathbf{x} - \mathbf{x}^{(k)}$. On souhaite savoir sous quelles conditions l'erreur $\mathbf{e}^{(k)}$ tend vers zéro lorsque $k \to \infty$ et, le cas échéant, quantifier la vitesse de convergence par rapport à k.

Or, la solution exacte satisfait le système

$$P\mathbf{x} = (P - A)\mathbf{x} + \mathbf{b} \quad (5.5)$$

qui est équivalent au système de départ $A\mathbf{x} = \mathbf{b}$. Si on soustrait (5.4) à (5.5), on a

$$P\mathbf{e}^{(k+1)} = (P - A)\mathbf{e}^{(k)}$$

et donc l'erreur $\mathbf{e}^{(k)}$ à la k-ème itération satisfait l'équation

$$\mathbf{e}^{(k+1)} = P^{-1}(P-A)\mathbf{e}^{(k)} = \underbrace{(I - P^{-1}A)}_{B}\mathbf{e}^{(k)}.$$

Définition 5.1. *On appelle* matrice d'itération de la méthode itérative (5.4) *la matrice* $B = (I - P^{-1}A)$.

En rappelant la définition 4.4 de *norme de matrice*, on a que

$$\begin{aligned}\|\mathbf{e}^{(k+1)}\| &\leq \|B\|\|\mathbf{e}^{(k)}\| \\ &\leq \|B\|^2\|\mathbf{e}^{(k-1)}\| \\ &\cdots \\ &\leq \|B\|^{k+1}\|\mathbf{e}^{(0)}\|.\end{aligned}$$

On voit donc que la norme de l'erreur tend vers zéro si $\|B\| < 1$. On a montré le résultat suivant :

Theorème 5.1. *La méthode de Richardson* (5.4) *avec matrice d'itération* $B = (I - P^{-1}A)$ *converge pour toute donnée initiale* $\mathbf{x}^{(0)}$ *si* $\|B\| < 1$. *De plus*

$$\|\mathbf{x} - \mathbf{x}^{(k)}\| \leq \|B\|^k\|\mathbf{x} - \mathbf{x}_0\|. \tag{5.6}$$

Notez la similarité entre la condition de convergence $\|B\| < 1$ pour la méthode itérative de Richardson et la condition $|\phi'(\alpha)| < 1$ pour la convergence d'une méthode de point fixe (voir Chapitre 1). La différence par rapport aux méthodes de point fixe pour une équation non-linéaire est que si la méthode (5.4) converge, elle converge pour n'importe quelle donnée initiale $\mathbf{x}^{(0)}$ (ceci n'est pas vrai en générale pour des équations non linéaires).

De plus, d'après (5.6) on voit que plus $\|B\|$ est petit, plus vite l'erreur tend vers zéro. La quantité $\|B\|$ donne donc une indication sur la vitesse de convergence.

On mentionne aussi une condition plus précise qui est en fait *nécessaire et suffisante* pour la convergence d'une méthode itérative.

Définition 5.2 (Rayon spectral). *Étant donnée une matrice carrée* $B \in \mathbb{R}^{n\times n}$ *et ses valeurs propres* $\lambda_i(B)$, $i = 1, \ldots, n$, *on appelle* rayon spectral *de la matrice* B, *noté* $\rho(B)$, *la quantité*

$$\rho(B) = \max_{i=1,\ldots,n} |\lambda_i(B)|. \tag{5.7}$$

On a le résultat suivant

Theorème 5.2. *Condition nécessaire et suffisante pour que la méthode itérative de Richardson* (5.4) *avec matrice d'itération* $B = (I - P^{-1}A)$ *converge est que*

$$\rho(B) < 1.$$

5.4 Contrôle de l'erreur et critère d'arrêt

L'algorithme 5.1 n'est pas complet car il faut encore donner un critère d'arrêt des itérations.

Idéalement, on aimerait terminer les itérations lorsque la norme de l'erreur $\|\mathbf{e}^{(k)}\| = \|\mathbf{x} - \mathbf{x}^{(k)}\|$ est plus petite qu'une tolérance donnée. Malheureusement, on n'a pas accès à l'erreur puisque on ne connaît pas la solution exacte. Toutefois, on peut utiliser le résidu

$$\mathbf{r}^{(k)} = \mathbf{b} - A\mathbf{x}^{(k)}$$

pour contrôler la convergence. Si $\mathbf{x}^{(k)}$ est proche de la solution exacte, on s'attend à ce que le résidu $\mathbf{r}^{(k)}$ soit petit. On peut donc imposer que le résidu soit suffisamment petit, par exemple par rapport au terme de droite

$$\textbf{critère d'arrêt :} \quad \frac{\|\mathbf{r}^{(k)}\|}{\|\mathbf{b}\|} \leq tol.$$

Voici la version complète de l'algorithme :

Algorithme 5.2 : Méthode de Richardson (avec critère d'arrêt)

Données : A, $\mathbf{b}$, $\mathbf{x}^{(0)}$, P, tol
Résultat : $\mathbf{x}$, res, niter
$\mathbf{r}^{(0)} = \mathbf{b} - A\mathbf{x}^{(0)}$;
$k = 0$;
tant que $\|\mathbf{r}^{(k)}\| > tol\|\mathbf{b}\|$ **faire**
  $P\mathbf{z}^{(k)} = \mathbf{r}^{(k)}$;
  $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \mathbf{z}^{(k)}$;
  $\mathbf{r}^{(k+1)} = \mathbf{b} - A\mathbf{x}^{(k+1)}$;
  $k = k + 1$;
fin
$\mathbf{x} = \mathbf{x}^{(k)}$, res= $\|\mathbf{r}^{(k)}\|$, niter= k;

L'algorithme termine lorsque la norme du résidu (normalisée par rapport à la norme du terme de droite) est plus petite que la tolérance fixée. Qu'est-ce qu'on sait dire sur la vraie erreur $\|\mathbf{e}^{(k)}\| = \|\mathbf{x} - \mathbf{x}^{(k)}\|$?

On remarque que la solution exacte satisfait le système

$$A\mathbf{x} = \mathbf{b}$$

alors que la solution approchée $\mathbf{x}^{(k)}$ satisfait le système

$$A\mathbf{x}^{(k)} = \mathbf{b} - \mathbf{r}^{(k)},$$

la dernière étant une simple identité. On peut donc voir la solution approchée $\mathbf{x}^{(k)}$ comme la solution exacte d'un système linéaire de matrice A et terme

de droite perturbé $\hat{\mathbf{b}} = \mathbf{b} - \mathbf{r}^{(k)}$. Pour cela, on peut utiliser les résultats de la section 4.6 et déduire l'estimation suivante

$$\frac{\|\mathbf{x} - \mathbf{x}^{(k)}\|}{\|\mathbf{x}\|} \leq \mathcal{K}(A) \frac{\|\mathbf{b} - \hat{\mathbf{b}}\|}{\|\mathbf{b}\|} = \mathcal{K}(A) \frac{\|\mathbf{r}^{(k)}\|}{\|\mathbf{b}\|} \leq \mathcal{K}(A)\, tol$$

où $\mathcal{K}(A) = \sqrt{\lambda_{max}(A^T A)/\lambda_{min}(A^T A)}$ est le nombre de conditionnement de la matrice A. On voit que si le conditionnement de la matrice est proche de 1, le contrôle sur le résidu est fiable alors que si le conditionnement est très grand, le résidu ne donne pas un bon contrôle sur la vraie erreur $\|\mathbf{x} - \mathbf{x}^{(k)}\|$.

5.5 Méthodes pour des matrices symétriques et définies positives

Une classe importante de matrices qui apparaît souvent dans les applications de la physique et de l'ingénierie est donnée par les matrices symétriques et définies positives (brièvement *s.d.p.*). Ces matrices apparaissent surtout lorsqu'on cherche des configurations d'équilibre d'un système physique qui minimisent une énergie. On rappelle que

Définition 5.3. *Une matrice $A \in \mathbb{R}^{n \times n}$ est **définie positive** si*

$$\mathbf{v}^T A \mathbf{v} > 0 \qquad \forall \mathbf{v} \in \mathbb{R}^n,\ \mathbf{v} \neq \mathbf{0}.$$

En particulier, si une matrice est s.d.p., alors toutes ses valeurs propres sont réelles et positives.

Soit maintenant $A\mathbf{x} = \mathbf{b}$ un système linéaire dont la matrice A est s.p.d. On peut lui associer une *fonction énergie*, à valeurs dans $\mathbb{R}$

$$\phi : \mathbb{R}^n \to \mathbb{R}, \qquad \phi(\mathbf{v}) = \frac{1}{2}\mathbf{v}^T A \mathbf{v} - \mathbf{v}^T \mathbf{b}, \qquad \mathbf{v} \in \mathbb{R}^n. \tag{5.8}$$

On a l'importante caractérisation suivante de la solution

Proposition 5.3. *La solution du système $A\mathbf{x} = \mathbf{b}$, avec matrice A s.d.p., est l'unique minimum de la fonction ϕ, c'est-à-dire*

$$\mathbf{x} = \underset{\mathbf{v} \in \mathbb{R}^n}{\operatorname{argmin}}\, \phi(\mathbf{v}).$$

On détaille ci-dessous le cas en dimension $n = 2$.

Exemple 5.1. *Considérons le système linéaire*

$$\underbrace{\begin{bmatrix} a_{11} & a_{12} \\ a_{12} & a_{22} \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} x \\ y \end{bmatrix}}_{\mathbf{x}} = \underbrace{\begin{bmatrix} b_1 \\ b_2 \end{bmatrix}}_{\mathbf{b}}.$$

La fonction énergie est

$$\begin{aligned}\phi(x,y) &= \frac{1}{2}\begin{bmatrix} x & y \end{bmatrix}\begin{bmatrix} a_{11} & a_{12} \\ a_{12} & a_{22} \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix} - \begin{bmatrix} x & y \end{bmatrix}\begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \\ &= \frac{1}{2}a_{11}x^2 + a_{12}xy + \frac{1}{2}a_{22}y^2 - b_1x - b_2y.\end{aligned}$$

Son gradient est

$$\nabla\phi(x,y) = \begin{bmatrix} \frac{\partial\phi}{\partial x} \\ \frac{\partial\phi}{\partial y} \end{bmatrix} = \begin{bmatrix} a_{11}x + a_{12}y - b_1 \\ a_{12}x + a_{22}y - b_2 \end{bmatrix} = A\mathbf{x} - \mathbf{b}$$

tandis que la matrice Hessienne est

$$H_\phi(x,y) = \begin{bmatrix} \frac{\partial^2\phi}{\partial x^2} & \frac{\partial^2\phi}{\partial x\partial y} \\ \frac{\partial^2\phi}{\partial x\partial y} & \frac{\partial^2\phi}{\partial y^2} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{12} & a_{22} \end{bmatrix} = A.$$

Les points stationnaires de ϕ satisfont donc la condition

$$\textit{points stationnaires :} \quad \nabla\phi(x,y) = A\mathbf{x} - \mathbf{b} = \mathbf{0}.$$

Puisque le système $A\mathbf{x} = \mathbf{b}$ a une seule solution, on conclut qu'il y a un seul point stationnaire qui coïncide avec la solution du système linéaire. De plus, puisque la matrice Hessienne $H_\phi = A$ est s.d.p. par hypothèse (et donc ses valeurs propres sont positives), ce point stationnaire est l'unique minimum de la fonction ϕ.

Les calculs montrés dans l'exemple précédent en dimension $n = 2$ sont valables en toute dimension. On a, en particulier,

$$\text{Gradient :} \qquad \nabla\phi(\mathbf{x}) = A\mathbf{x} - \mathbf{b}, \qquad \forall \mathbf{x} \in \mathbb{R}^n \tag{5.9}$$

$$\text{Hessienne :} \qquad H_\phi(\mathbf{x}) = A, \qquad \forall \mathbf{x} \in \mathbb{R}^n. \tag{5.10}$$

Grâce à cette interprétation de la solution du système linéaire comme le minimum d'une fonction énergie, on peut construire d'autres méthodes itératives qui essayent d'approcher le minimum de ϕ au lieu d'approcher la solution de $A\mathbf{x} = \mathbf{b}$.

5.5.1 Méthode du gradient (ou méthode de la plus grande pente)

L'idée de la méthode du gradient est relativement simple. Supposons d'avoir une estimation $\mathbf{x}^{(k)}$ de la solution de $A\mathbf{x} = \mathbf{b}$, qui est aussi une estimation du point minimum de ϕ. On cherche à construire une meilleure approximation $\mathbf{x}^{(k+1)}$ pour laquelle $\phi(\mathbf{x}^{(k+1)}) < \phi(\mathbf{x}^{(k)})$. Pour cela, on suit la direction de la plus grande pente de la fonction ϕ en espérant, de cette

façon, se rapprocher au plus vite du minimum de ϕ. Or, on a vu que le gradient de ϕ en $\mathbf{x}^{(k)}$ est

$$\nabla\phi(\mathbf{x}^{(k)}) = A\mathbf{x}^{(k)} - \mathbf{b} = -\mathbf{r}^{(k)}.$$

Donc, le résidu $\mathbf{r}^{(k)}$ donne bien la direction de décroissance maximale. On pose alors

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k \mathbf{r}^{(k)}. \tag{5.11}$$

Le paramètre α_k indique de combien on se déplace dans la direction $\mathbf{r}^{(k)}$ à partir de $\mathbf{x}^{(k)}$. Notez que si on prend $\alpha_k = 1$ on retrouve la méthode de Richardson sans préconditionneur (c'est-à-dire avec $P = I$ la matrice identité). La méthode du gradient est donc une méthode de Richardson.

Dans (5.11) il faut encore choisir le paramètre α_k. L'idée est de choisir α_k de sort que $\phi(\mathbf{x}^{(k+1)})$ soit le plus petit possible (notre but est de minimiser ϕ). On a

$$\begin{aligned}
\phi(\mathbf{x}^{(k+1)}) &= \phi(\mathbf{x}^{(k)} + \alpha_k \mathbf{r}^{(k)}) \\
&= \frac{1}{2}(\mathbf{x}^{(k)} + \alpha_k \mathbf{r}^{(k)})^T A(\mathbf{x}^{(k)} + \alpha_k \mathbf{r}^{(k)}) - (\mathbf{x}^{(k)} + \alpha_k \mathbf{r}^{(k)})^T \mathbf{b} \\
&= \frac{1}{2}(\mathbf{r}^{(k)})^T A\mathbf{r}^{(k)} \alpha_k^2 + \left(\frac{1}{2}(\mathbf{r}^{(k)})^T A\mathbf{x}^{(k)} + \frac{1}{2}(\mathbf{x}^{(k)})^T A\mathbf{r}^{(k)} - (\mathbf{r}^{(k)})^T \mathbf{b}\right) \alpha_k \\
&\quad + \frac{1}{2}(\mathbf{x}^{(k)})^T A\mathbf{x}^{(k)} - (\mathbf{x}^{(k)})^T \mathbf{b} \\
&= a\alpha_k^2 + b\alpha_k + c
\end{aligned}$$

où

$$a = \frac{1}{2}(\mathbf{r}^{(k)})^T A\mathbf{r}^{(k)}, \qquad c = \frac{1}{2}(\mathbf{x}^{(k)})^T A\mathbf{x}^{(k)} - (\mathbf{x}^{(k)})^T \mathbf{b}$$

et

$$\begin{aligned}
b &= \frac{1}{2}(\mathbf{r}^{(k)})^T A\mathbf{x}^{(k)} + \frac{1}{2}(\mathbf{x}^{(k)})^T A\mathbf{r}^{(k)} - (\mathbf{r}^{(k)})^T \mathbf{b} \\
&= (\mathbf{r}^{(k)})^T A\mathbf{x}^{(k)} - (\mathbf{r}^{(k)})^T \mathbf{b} \\
&= -(\mathbf{r}^{(k)})^T \mathbf{r}^{(k)}.
\end{aligned}$$

Dans la deuxième égalité, on a utilisé le fait que pour tout $\mathbf{v}, \mathbf{w} \in \mathbb{R}^n$, $\mathbf{v}^T A\mathbf{w} = \mathbf{w}^T A\mathbf{v}$ puisque la matrice A est symétrique.

On voit que $\phi(\mathbf{x}^{(k+1)})$ est un polynôme de degré deux en la variable α_k. Pour le minimiser on pose alors

$$\frac{d\phi}{d\alpha_k}(\mathbf{x}^{(k+1)}) = 2a\alpha_k + b = 0$$

ce qui donne la valeur optimale

$$\alpha_k = -\frac{b}{2a} = \frac{(\mathbf{r}^{(k)})^T \mathbf{r}^{(k)}}{(\mathbf{r}^{(k)})^T A\mathbf{r}^{(k)}}.$$

Une fois la nouvelle approximation $\mathbf{x}^{(k+1)}$ calculée, on peut mettre à jour le résidu par la formule

$$\mathbf{r}^{(k+1)} = \mathbf{b} - A\mathbf{x}^{(k+1)} = \mathbf{b} - A(\mathbf{x}^{(k)} + \alpha_k \mathbf{r}^{(k)}) = \mathbf{r}^{(k)} - \alpha_k A\mathbf{r}^{(k)}.$$

Voici une implémentation possible de l'algorithme du gradient où on a utilisé le critère d'arrêt sur le résidu normalisé :

Algorithme 5.3 : Méthode du gradient

Données : A, $\mathbf{b}$, $\mathbf{x}^{(0)}$, tol
Résultat : $\mathbf{x}$, res, niter
$\mathbf{r}^{(0)} = \mathbf{b} - A\mathbf{x}^{(0)}$;
$k = 0$;
tant que $\|\mathbf{r}^{(k)}\| > tol\|\mathbf{b}\|$ **faire**
  $\mathbf{w}^{(k)} = A\mathbf{r}^{(k)}$;
  $\alpha_k = \frac{(\mathbf{r}^{(k)})^T \mathbf{r}^{(k)}}{(\mathbf{r}^{(k)})^T \mathbf{w}^{(k)}}$;
  $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k \mathbf{r}^{(k)}$;
  $\mathbf{r}^{(k+1)} = \mathbf{r}^{(k)} - \alpha_k \mathbf{w}^{(k)}$;
  $k = k + 1$;
fin
$\mathbf{x} = \mathbf{x}^{(k)}$, res= $\|\mathbf{r}^{(k)}\|$, niter= k;

Pour la méthode du gradient on a le résultat suivant

Theorème 5.4. *La méthode du gradient pour résoudre le système linéaire $A\mathbf{x} = \mathbf{b}$, avec A s.d.p., converge toujours vers la solution exacte pour n'importe quelle donnée initiale $\mathbf{x}^{(0)}$. De plus, on a l'estimation d'erreur suivante*

$$\|\mathbf{x} - \mathbf{x}^{(k)}\| \leq C \left(\frac{\mathcal{K}(A) - 1}{\mathcal{K}(A) + 1} \right)^k \|\mathbf{x} - \mathbf{x}^{(0)}\| \tag{5.12}$$

où C est une constante positive.

La bonne propriété de la méthode du gradient est qu'elle converge toujours. Toutefois, la convergence n'est pas toujours rapide comme on le voit en (5.12) si $\mathcal{K}(A) \gg 1$.

5.5.2 Généralisations

On mentionne brièvement ici quelques généralisations de la méthode du gradient.

Méthode du gradient préconditionné

D'après le Théorème 5.4, la vitesse de convergence de la méthode du gradient est liée au facteur $(\mathcal{K}(A) - 1)/(\mathcal{K}(A) + 1)$. Si ce facteur est très

proche de 1, ce qui peut arriver si $\mathcal{K}(A) \gg 1$, la convergence sera très lente. Pour accélérer la convergence, on peut utiliser l'idée suivante.

Soit $P \in \mathbb{R}^{n\times n}$ une matrice inversible, dite *matrice de préconditionnement*. Au lieu d'appliquer la méthode du gradient au système linéaire $A\mathbf{x} = \mathbf{b}$, on applique la méthode au système linéaire

$$P^{-1}A\mathbf{x} = P^{-1}\mathbf{b}. \tag{5.13}$$

La convergence de la méthode sera cette fois-ci liée à $\mathcal{K}(P^{-1}A)$:

$$\|\mathbf{x} - \mathbf{x}^{(k)}\| \leq C \left(\frac{\mathcal{K}(P^{-1}A) - 1}{\mathcal{K}(P^{-1}A) + 1}\right)^k \|\mathbf{x} - \mathbf{x}^{(0)}\|. \tag{5.14}$$

Si on choisit bien la matrice de préconditionnement de sorte que

$$\mathcal{K}(P^{-1}A) \ll \mathcal{K}(A),$$

la méthode du gradient, appliquée au système préconditionné (5.13), convergera beaucoup plus vite que la même méthode appliquée au système de départ $A\mathbf{x} = \mathbf{b}$.

Il faut toutefois faire attention dans la mise en œvre de la méthode du gradient préconditionné :

1. La matrice $P^{-1}A$ pourrait ne pas être s.d.p. On peut montrer que *si les matrices A et P sont les deux s.d.p.*, alors on peut appliquer la méthode du gradient au système (5.13) et elle va converger avec une vitesse de convergence donnée par (5.14).

   À l'itération k de la méthode, la mise à jour de la solution $\mathbf{x}^{(k+1)}$ est

   $$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k \mathbf{z}^{(k)}$$

   où $\mathbf{z}^{(k)}$ est le *résidu préconditionné* (résidu du système préconditionné (5.13))

   $$\mathbf{z}^{(k)} = P^{-1}\mathbf{b} - P^{-1}A\mathbf{x}^{(k)} = P^{-1}\mathbf{r}^{(k)}$$

   et α_k est donné par $\alpha_k = \frac{(\mathbf{z}^{(k)})^T \mathbf{r}^{(k)}}{(\mathbf{z}^{(k)})^T A\mathbf{z}^{(k)}}$.

   Puisque, en générale, on ne veut pas calculer explicitement la matrice inverse P^{-1}, la mise à jour de la solution entraîne les étapes

   $$P\mathbf{z}^{(k)} = \mathbf{r}^{(k)} \tag{5.15}$$
   $$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k \mathbf{z}^{(k)} \tag{5.16}$$

   et donc la résolution d'un système linéaire avec matrice P.

   L'algorithme complet est le suivant :

Algorithme 5.4 : Méthode du gradient préconditionné

Étant donnés $\mathbf{x}^{(0)}$ et $P \in \mathbb{R}^{n\times n}$ s.d.p. et $\mathbf{r}^{(0)} = \mathbf{b} - A\mathbf{x}^{(0)}$;
pour $k = 0, 1, \ldots$ **faire**
  $P\mathbf{z}^{(k)} = \mathbf{r}^{(k)}$;
  $\mathbf{w}^{(k)} = A\mathbf{z}^{(k)}$;
  $\alpha_k = \frac{(\mathbf{z}^{(k)})^T\mathbf{r}^{(k)}}{(\mathbf{z}^{(k)})^T\mathbf{w}^{(k)}}$;
  $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k\mathbf{z}^{(k)}$;
  $\mathbf{r}^{(k+1)} = \mathbf{r}^{(k)} - \alpha_k\mathbf{w}^{(k)}$;
fin

2. Le choix de la matrice P est délicat. D'un coté, elle doit être telle que $\mathcal{K}(P^{-1}A) \ll \mathcal{K}(A)$; de l'autre coté, le système linéaire (5.15) doit être facile à résoudre.

   Le choix *idéal* de préconditionneur, par rapport au premier critère, est $P = A$. En fait, dans ce cas on a $\mathcal{K}(P^{-1}A) = \mathcal{K}(I) = 1$ et la méthode converge en une seule itération. Toutefois, par ce choix, le système linéaire (5.15) à résoudre dans la boucle est aussi compliqué que le système de départ et donc il n'y a aucun avantage a utiliser $P = A$.

   On doit donc trouver un bon compromis : la matrice P doit être suffisamment proche de A, tout en restant relativement simple pour la résolution du système linéaire (5.15).

   Une possibilité qu'on a déjà vue est de prendre la matrice P formée de la seule diagonale de A.

   Dans le cas d'une matrice creuse, une autre possibilité qui est très utilisée dans la pratique est de faire une factorisation LU inexacte de la matrice, c'est-à-dire, $P = \hat{L}\hat{U}$ où $\hat{L}$ et $\hat{U}$ sont des approximations des facteurs L et U de A. Cette technique est appelée ILU (*incomplete LU*). En particulier, on peut faire une factorisation LU de A et sauvegarder seulement les éléments les plus "grands" des matrices L et U (ILUt : incomplete LU with threshold) ou encore, sauvegarder seulement les éléments qui correspondent en position aux éléments non nuls de la matrice A (ILU0).

   La technique ILU permet d'éviter le problème du fill-in et donne quand même une matrice $P = \hat{L}\hat{U}$ qui est assez proche de A et un système $P\mathbf{z}^{(k)} = \mathbf{r}^{(k)}$ facile à résoudre car la matrice P est déjà factorisée.

Méthode du gradient conjugué

Dans la méthode du gradient, on met à jour la solution par l'équation

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k\mathbf{r}^{(k)}$$

où $\mathbf{r}^{(k)} = -\nabla\phi(\mathbf{x}^{(k)})$ est la direction de décroissance maximale de la fonction ϕ. On peut se poser la question s'il existe des directions de descente autres que le gradient, qui permettraient d'atteindre le minimum de ϕ en moins d'itérations.

Une stratégie qui marche particulièrement bien consiste à choisir à chaque étape de la méthode une direction $\mathbf{p}^{(k)}$ qui a la propriété suivante :

$$(\mathbf{p}^{(k)})^T A\mathbf{p}^{(j)} = 0, \qquad j = 0, \ldots, k-1. \tag{5.17}$$

On appelle les vecteurs $\mathbf{p}^{(k)}$ qui satisfont (5.17) *A-conjugués* ou *A-orthogonaux*. Dans ce cas, la mise à jour de la solution est

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k \mathbf{p}^{(k)}$$

où

$$\alpha_k = \frac{(\mathbf{p}^{(k)})^T \mathbf{r}^{(k)}}{(\mathbf{p}^{(k)})^T A\mathbf{p}^{(k)}}$$

et on peut calculer les directions $\mathbf{p}^{(k)}$ A-conjuguées par la formule de récurrence

$$\mathbf{p}^{(k+1)} = \mathbf{r}^{(k+1)} - \beta_k \mathbf{p}^{(k)}, \qquad \beta_k = \frac{(A\mathbf{p}^{(k)})^T \mathbf{r}^{(k)}}{(\mathbf{p}^{(k)})^T A\mathbf{p}^{(k)}}.$$

Pour la méthode du gradient conjugué (appelée CG de l'anglais *conjugate gradient*) on a le résultat suivant

Theorème 5.5. *En arithmétique exacte, la méthode du gradient conjugué pour résoudre le système linéaire $A\mathbf{x} = \mathbf{b}$, avec A s.d.p., converge en au plus n itérations à la solution exacte pour n'importe quelle donnée initiale $\mathbf{x}^{(0)}$.*

De plus, l'erreur $\|\mathbf{x} - \mathbf{x}^{(k)}\|$ à la k-ème itération satisfait l'estimation suivante

$$\|\mathbf{x} - \mathbf{x}^{(k)}\| \leq C \left(\frac{\sqrt{\mathcal{K}(A)} - 1}{\sqrt{\mathcal{K}(A)} + 1} \right)^k \|\mathbf{x} - \mathbf{x}^{(0)}\| \tag{5.18}$$

où C est une constante positive.

On voit donc que la méthode du gradient conjugué converge plus rapidement que la méthode du gradient car le facteur de réduction de l'erreur fait intervenir $\sqrt{\mathcal{K}(A)}$ au lieu que $\mathcal{K}(A)$.

Bien évidemment, on peut encore combiner la méthode du gradient conjugué avec la technique du préconditionnement, ce qui donne la méthode PCG (*preconditioned conjugate gradient*). Avec un bon choix de préconditionneur, la méthode PCG est très performante.

Chapitre 6

Équations différentielles ordinaires

Dans ce chapitre, on s'intéresse à la résolution numérique d'une équation différentielle ordinaire : *trouver une fonction continûment dérivable* $u : \mathbb{R}_+ \to \mathbb{R}$ *telle que*

$$\begin{cases} \dfrac{du(t)}{dt} = f(t, u(t)), \quad t > 0, \\ u(0) = u_0. \end{cases} \tag{6.1}$$

Ce problème est appelé *problème de Cauchy.*

Dans les applications, on aboutit souvent à des *systèmes d'équations différentielles ordinaires.* Soient

$$\mathbf{u}(t) = \begin{bmatrix} u_1(t) \\ \vdots \\ u_m(t) \end{bmatrix} \qquad \text{et} \qquad \mathbf{f}(t, \mathbf{u}) = \begin{bmatrix} f_1(t, u_1, \ldots, u_m) \\ \vdots \\ f_m(t, u_1, \ldots, u_m) \end{bmatrix}.$$

On cherche une fonction vectorielle $\mathbf{u} : \mathbb{R}_+ \to \mathbb{R}^m$ *telle que*

$$\begin{cases} \dfrac{d\mathbf{u}(t)}{dt} = \mathbf{f}(t, \mathbf{u}(t)), \quad t > 0 \\ \mathbf{u}(0) = \mathbf{u}_0. \end{cases} \tag{6.2}$$

Exemple 6.1. *On considère une population de lapins. Soit* $u(t)$ *le nombre d'individus à l'instant* t *et* $u_0 = u(0)$ *le nombre d'individus à l'instant initial* $t = 0$*. Il est raisonnable de supposer que le* taux de croissance *de la population à l'instant* t*, c'est-à-dire le nombre d'individus nés moins le nombre d'individus décédés dans l'unité de temps, soit proportionnel au nombre total* $u(t)$ *d'individus de la population :*

$$\textit{taux de croissance à l'instant } t : \qquad \tau(t) = Cu(t).$$

On peut modéliser la dynamique de la population par l'équation différentielle ordinaire

$$\begin{cases} \dfrac{du(t)}{dt} = Cu(t), \quad t > 0, \\ u(0) = u_0. \end{cases} \tag{6.3}$$

La solution de cette équation est $u(t) = u_0 e^{Ct}$. *Ce modèle n'est pas tout à fait raisonnable car il prévoit que la population augmente indéfiniment en temps. Puisque la nourriture à disposition des lapins dans l'environnement n'est pas infinie, un modèle plus réaliste prévoit que le nombre d'individus ne puisse pas dépasser une valeur maximale* u_{max}. *Un modèle différentiel possible est le suivant*

$$\begin{cases} \dfrac{du(t)}{dt} = Cu(t)\left(1 - \dfrac{u(t)}{u_{max}}\right), \quad t > 0, \\ u(0) = u_0. \end{cases} \tag{6.4}$$

Notez que dans ce modèle, le taux de croissance $\tau(t) = Cu(t)\left(1 - \frac{u(t)}{u_{max}}\right)$ *devient zéro lorsque la population atteint la valeur maximale* u_{max} *et la population ne croît plus.*

Exemple 6.2. *On reprend l'exemple précédent mais cette fois-ci on considère deux populations : lapins et renards. Soit* $u_1(t)$ *le nombre de lapins et* $u_2(t)$ *le nombre de renards à l'instant* t. *Pour chacune des deux populations on peut écrire une équation du type* (6.3) *ou* (6.4). *Considérons, pour simplifier, le modère* (6.3). *Toutefois, la dynamique des deux populations est* couplée. *En fait, les renards mangent les lapins et donc la nourriture à disposition des renards dépend du nombre de lapins* $u_1(t)$. *De plus, le taux de mortalité de lapins dépend du nombre de renards en circulation. On peut donc écrire le modèle suivant*

$$\begin{cases} \dfrac{du_1(t)}{dt} = \alpha u_1(t) - \beta u_1(t)u_2(t), & t > 0, \\ \dfrac{du_2(t)}{dt} = -\gamma u_2(t) + \delta u_1(t)u_2(t), & t > 0, \\ u_1(0) = u_{1,0}, \qquad u_2(0) = u_{2,0} \end{cases}$$

où $\alpha u_1(t)$ *est le taux de naissance moins le taux de mortalité naturelle de la population de lapins ;* $-\beta u_1(t)u_2(t)$ *est le taux de mortalité des lapins dû à la présence des renards ;* $-\gamma u_2(t)$ *est le taux de mortalité naturelle des renards ;* $\delta u_1(t)u_2(t)$ *est le taux de naissance des renards qui est proportionnel au nombre de lapins en circulation. Ce modèle est connu comme* modèle de Lotka-Volterra. *Il s'agit d'un système de deux équations différentielles ordinaires couplées et peut être écrit sous forme vectorielle* (6.2) *où*

$$\mathbf{u}(t) = \begin{bmatrix} u_1(t) \\ u_2(t) \end{bmatrix}, \qquad \mathbf{f}(t, \mathbf{u}(t)) = \begin{bmatrix} \alpha u_1(t) - \beta u_1(t)u_2(t) \\ -\gamma u_2(t) + \delta u_1(t)u_2(t) \end{bmatrix}.$$

6.1 Rappel sur les résultats d'existence et unicité

L'existence d'une solution du problème (6.1) n'est pas garantie pour tout $t > 0$. De plus, ce problème n'a pas nécessairement une solution unique.

Exemple 6.3. *Considérons par exemple l'équation*

$$\frac{du(t)}{dt} = \sqrt{u(t)}, \quad t > 0, \qquad u(0) = 0.$$

Il est facile de vérifier que $u_1(t) = 0,\ t > 0$ *et* $u_2(t) = \frac{1}{4}t^2,\ t > 0$ *sont deux solutions de l'équation. Ce problème de Cauchy n'a donc pas une solution unique.*

Exemple 6.4. *Considérons le problème de Cauchy*

$$\frac{du(t)}{dt} = u^2(t), \quad t > 0, \qquad u(0) = 1.$$

La solution est $u(t) = \frac{1}{1-t}$ *qui existe seulement pour* $t < 1$.

Dans notre étude des schémas pour la résolution numérique d'une équation différentielle ordinaire, on veut exclure les situations présentées dans les exemples 6.3 et 6.4. On rappelle un résultat d'existence et unicité d'une solution de (6.1) pour tout $t > 0$:

Theorème 6.1. *Soit* $f : \mathbb{R} \times \mathbb{R}_+ \to \mathbb{R}$ *une fonction continue et* globalement Lipschitzienne *par rapport au deuxième argument, c'est-à-dire qu'il existe une constante* $L > 0$ *telle que*

$$|f(t, x) - f(t, y)| \leq L|x - y|, \qquad \forall x, y \in \mathbb{R},\ \forall t \in [0, \infty).$$

Alors, le problème de Cauchy (6.1) *a une solution unique* $u(t)$ *définie pour tout* $t \in [0, \infty)$ *et continûment dérivable.*

Ce théorème se généralise au cas de systèmes d'équations différentielles ordinaires. Dans la suite du chapitre on va toujours supposer que les hypothèses du théorème sont vérifiées.

6.2 Schémas à un pas

On souhaite résoudre le problème de Cauchy (6.1) dans l'intervalle $[0, T]$. Pour cela, on divise l'intervalle en N sous-intervalles $[t_n, t_{n+1}]$ de la même longueur $\Delta t = \frac{T}{N}$, où $t_n = n\Delta t,\ n = 0, ..., N$. Considérons l'équation différentielle écrite à l'instant t_n :

$$\frac{du(t_n)}{dt} = f(t_n, u(t_n)). \tag{6.5}$$

Schéma d'Euler progressif (ou explicite)

Une idée très simple pour construire un schéma de résolution numérique consiste à remplacer la dérivée $\frac{du(t_n)}{dt}$ par une formule aux différences finies progressives

$$\frac{du(t_n)}{dt} \approx \delta^+_{\Delta t} u(t_n) = \frac{u(t_{n+1}) - u(t_n)}{\Delta t}.$$

Si on dénote par $u^n \approx u(t_n)$ une approximation de $u(t_n)$, on peut introduire le **Schéma d'Euler progressif**

$$\begin{cases} \dfrac{u^{n+1} - u^n}{\Delta t} = f(t_n, u^n), \quad n = 0, 1, \ldots, N-1 \\ u^0 = u_0 \ \text{(connu)}. \end{cases} \tag{6.6}$$

On note que, puisqu'on a remplacé dans (6.6) la dérivée exacte $\frac{du}{dt}$ par une différence finie progressive $\delta^+_{\Delta t} u$, la solution u^n de (6.6) *ne sera pas* égale à la solution exacte $u(t_n)$ du problème de Cauchy. On s'intéresse dans la prochaine section à étudier le comportement de l'erreur $e_n = |u(t_n) - u^n|$.

Si on prend $n = 0$ dans (6.6), on peut calculer u^1 en fonction de u^0 qui est connu

$$u^1 = u^0 + \Delta t\, f(t_0, u^0).$$

Une fois u^1 calculé, en prennent $n = 1$ dans (6.6) on trouve la valeur u^2 et ainsi de suite. Ce schéma permet donc de construire une suite $(u^0, u^1, u^2, \ldots, u^N)$ qu'on espère être une bonne approximation de la solution exacte aux mêmes instants $(u_0 = u(t_0), u(t_1), u(t_2), \ldots, u(t_N) = u(T))$.

Le terme *explicite* fait allusion au fait qu'on peut calculer la solution u^{n+1} *explicitement* à partir de la solution u^n.

La fonction suivante donne une possible implémentation `Python` du schéma d'Euler progressif.

```
function [tn,un,dt]=euler(f,I,u0,N)
% resout le probleme de Cauchy
%   u'=f(t,u), t in (t0,T],  u(t0)=u0
% par la methode d'Euler avec pas de temps dt=(T-t0)/N
% Input:
%   f: fonction f(t,u) definie par inline f=@(t,u) ...
%   I=[t0,T]: interval d'integration
%   u0: donnee initiale
%   N: nombre de sous-intervalles
% Output:
%   tn: vecteur des temps tn
%   un: vecteur des solutions calculees
%   dt: pas de temps
dt=(I(2)-I(1))/N;   % pas de temps
tn=linspace(I(1),I(2),N+1);
un=zeros(1,N+1); un(1)=u0;
for i=1:N
```

```
    un(i+1) = un(i) + dt*f(tn(i),un(i));
end
```

Exemple 6.5. *On applique la méthode d'Euler progressive pour résoudre le problème de Cauchy*

$$\begin{cases} \frac{du(t)}{dt} = -(\frac{1}{2}u + 3te^{-t}), \quad t \in (0, 20], \\ u(0) = 1 \end{cases} \tag{6.7}$$

dont on peut calculer la solution exacte

$$u_{ex} = -11e^{-\frac{1}{2}t} + 12(1 + \frac{1}{2}t)e^{-t}.$$

On calcule en **Python** *la solution obtenue par le schéma d'Euler avec* $N = 20$ $(\Delta t = 1)$ *et on la compare avec la solution exacte*

```
>> a=.5; b=3;
>> f=@(t,u) -(a*u+b*t.*exp(-t));
>> u0=1; Tf=20; I=[0,Tf];
>> uex=@(t) (u0-b/(1-a)^2)*exp(-a*t)+b*(1+(1-a)*t).*exp(-t)/(1-a)^2;
>> t=0:.01:Tf;
>> plot(t,uex(t),'r-','LineWidth',2), hold on
>> grid on
>> N=20;
>> [tn,un,dt]=euler(f,I,u0,N);
>> plot(tn,un,'g*-','LineWidth',2), hold off
```

La figure 6.1 montre le résultat obtenu. Sur la même figure, on a aussi tracé les résultats obtenus avec $N = 40$ $(\Delta t = 0.5)$ *et* $N = 80$ $(\Delta t = 0.25)$. *On*

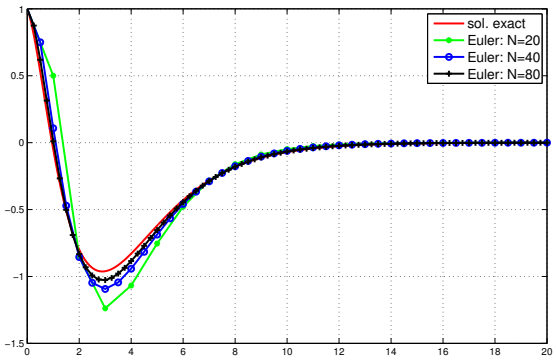


FIGURE 6.1 – Solution exacte de l'équation (6.7) et solution obtenue par la méthode d'Euler progressive avec $N = 20, 40, 80$

voit bien que la méthode d'Euler progressive donne une solution approchée qui devient de plus en plus précise lorsque le nombre de sous-intervalles N *augmente, c'est-à-dire lorsque le pas de temps* Δt *est réduit.*

Schéma d'Euler rétrograde (ou implicite)

Dans la construction d'un schéma numérique pour l'approximation de (6.5) on aurait aussi pu bien remplacer la dérivée $\frac{du(t_n)}{dt}$ par une différence finie rétrograde

$$\frac{du(t_n)}{dt} \approx \delta^-_{\Delta t} u(t_n) = \frac{u(t_n) - u(t_{n-1})}{\Delta t}.$$

Dans ce cas, on obtient le **Schéma d'Euler rétrograde**

$$\begin{cases} \dfrac{u^{n+1} - u^n}{\Delta t} = f(t_{n+1}, u^{n+1}), \quad n = 0, 1, \ldots, N-1 \\ u^0 = u_0 \ \text{(connu)}. \end{cases} \tag{6.8}$$

Étant donné la solution approchée u^n, pour calculer la nouvelle approximation u^{n+1} il faut résoudre l'équation

$$u^{n+1} - \Delta t\, f(t_{n+1}, u^{n+1}) = u^n.$$

Notez que celle-ci est une équation non-linéaire dont l'inconnue est u^{n+1}. Si on note $x = u^{n+1}$, ceci revient à chercher le zéro de la fonction

$$g_n(x) = x - \Delta t\, f(t_{n+1}, x) - u^n = 0.$$

À chaque pas de temps on doit donc résoudre une équation non linéaire (implicite) pour trouver la nouvelle solution u^{n+1} à partir de la solution précédente u^n, ce qui explique le nom *implicite* du schéma. On peut utiliser ici une des méthodes vues dans le Chapitre 1. Par exemple, on pourrait utiliser la méthode de point fixe suivante :

$$x^{(k+1)} = \Delta t\, f(t_{n+1}, x^{(k)}) + u^n$$

et puisqu'on s'attend à ce que la solution u^{n+1} ne soit pas trop différente de u^n, on peut prendre comme donnée initiale de la méthode de point fixe $x^{(0)} = u^n$.

Une autre possibilité est d'utiliser la méthode de Newton

$$x^{(k+1)} = x^{(k)} - \frac{x^{(k)} - \Delta t\, f(t_{n+1}, x^{(k)}) - u^n}{1 - \Delta t \frac{\partial f}{\partial x}(t_{n+1}, x^{(k)})}, \quad k = 0, 1, \ldots, \qquad x^{(0)} = u^n.$$

À première vue, on ne voit pas l'intérêt d'utiliser le schéma d'Euler rétrograde car il implique la résolution d'une équation non linéaire à chaque pas. Toutefois, on verra dans la Section 6.4 que cette méthode a de meilleures propriétés de stabilité absolue que la méthode d'Euler explicite.

Schéma de Crank-Nicolson

Une autre façon de dériver un schéma de discrétisation pour (6.1) est la suivante. On intègre (6.1) en temps entre t_n et t_{n+1} :

$$u(t_{n+1}) - u(t_n) = \int_{t_n}^{t_{n+1}} \frac{du(s)}{ds} ds = \int_{t_n}^{t_{n+1}} f(s, u(s)) ds,$$

puis on applique la formule du trapèze pour approcher la dernière intégrale

$$\int_{t_n}^{t_{n+1}} f(s, u(s)) ds \approx \frac{\Delta t}{2} \left[f(t_n, u(t_n)) + f(t_{n+1}, u(t_{n+1})) \right].$$

Ceci donne le **schéma de Crank-Nicolson**

$$\begin{cases} \dfrac{u^{n+1} - u^n}{\Delta t} = \dfrac{1}{2} f(t_n, u^n) + \dfrac{1}{2} f(t_{n+1}, u^{n+1}), \quad n = 0, 1, \ldots, N-1 \\ u^0 = u_0 \ \text{(connu)}. \end{cases} \tag{6.9}$$

Il s'agit encore d'un schéma *implicite*. On verra que ce schéma est en général plus précis que le schéma d'Euler (progressif ou rétrograde).

Schéma de Heun

Le désavantage du schéma de Crank-Nicolson est, tout comme le schéma d'Euler rétrograde, la nécessité de résoudre une équation non-linéaire à chaque pas de temps.

Une idée pour "rendre explicite" ce schéma est de calculer une première approximation de u^{n+1} par la méthode d'Euler progressif

$$\tilde{u}^{n+1} = u^n + \Delta t \, f(t_n, u^n)$$

et d'utiliser ensuite cette approximation dans le terme de droite de (6.9)

$$\frac{u^{n+1} - u^n}{\Delta t} = \frac{1}{2} f(t_n, u^n) + \frac{1}{2} f(t_{n+1}, \tilde{u}^{n+1}).$$

Si on met les deux équations ensemble, on obtient le **schéma de Heun**

$$\begin{cases} \dfrac{u^{n+1} - u^n}{\Delta t} = \dfrac{1}{2} f(t_n, u^n) + \dfrac{1}{2} f(t_{n+1}, u^n + \Delta t \, f(t_n, u^n)), \quad n = 0, 1, \ldots, N-1 \\ u^0 = u_0 \ \text{(connu)}. \end{cases} \tag{6.10}$$

Notez que ce schéma est *explicite* et on peut calculer la nouvelle solution u^{n+1} de façon explicite à partir de u^n par

$$u^{n+1} = u^n + \frac{\Delta t}{2} f(t_n, u^n) + \frac{\Delta t}{2} f(t_{n+1}, u^n + \Delta t \, f(t_n, u^n)).$$

Tous les schémas qu'on a vu jusqu'à présent permettent de calculer u^{n+1} à partir de la solution u^n au pas précédent. On parle dans ce cas de *schémas à un pas*.

Définition 6.1. *Un* ***schéma à un pas*** *pour approcher un problème de Cauchy* (6.1) *est un schéma de la forme*

$$\frac{u^{n+1}-u^n}{\Delta t} = \phi_f(u^n, u^{n+1}, t_n, \Delta t). \tag{6.11}$$

Si la fonction ϕ_f ne dépend pas de u^{n+1}, le schéma est explicite, autrement il est implicite.

Le tableau suivant montre la fonction ϕ_f pour les quatre schémas vus jusqu'à présent.

Schéma	$\phi_f(u^n, u^{n+1}, t_n, \Delta t) =$
Euler progressif	$f(t_n, u^n)$
Euler rétrograde	$f(t_n + \Delta t, u^{n+1})$
Crank Nicolson	$\frac{1}{2}f(t_n, u^n) + \frac{1}{2}f(t_n + \Delta t, u^{n+1})$
Heun	$\frac{1}{2}f(t_n, u^n) + \frac{1}{2}f(t_n + \Delta t, u^n + \Delta t\, f(t_n, u^n))$

On pourrait aussi imaginer construire la solution u^{n+1} en utilisant non seulement u^n mais également les solutions aux pas précédents u^{n-1}, u^{n-2}, etc. On parlerait alors de *schémas multipas.*

6.3 Analyse d'erreur

Étant donné un schéma à un pas (6.11), on s'intéresse maintenant à étudier le comportement de l'erreur entre la solution approchée et la solution exacte. On peut regarder ça à l'instant final $T = N\Delta t$:

$$\varepsilon_N = |u(T) - u^N|, \qquad \text{où } N = T/\Delta t.$$

En particulier, on souhaite comprendre comment l'erreur dépend du pas de temps Δt. On donne la définition suivante

Définition 6.2. *Un schéma numérique pour approcher le problème de Cauchy* (6.1) *est dit* ***convergent avec ordre*** *p s'il existe une constante $C > 0$ telle que*

$$|u(T) - u^N| \leq C\Delta t^p$$

pourvu que la solution exacte soit suffisamment régulière.

Pour étudier l'ordre de convergence d'un schéma numérique, il faut encore introduire quelques concepts supplémentaires.

Définition 6.3 (**erreur de troncature locale**)**.** *Soit $u(t_n)$ la solution exacte du problème de Cauchy* (6.1) *aux instants $t_n = n\Delta t$, $n = 0, 1, \ldots, N$. Pour un schéma à un pas* (6.11)*, on appelle* erreur de troncature locale *à l'instant t_{n+1} la quantité*

$$\tau_n = \frac{u(t_{n+1}) - u(t_n)}{\Delta t} - \phi_f(u(t_n), u(t_{n+1}), t_n, \Delta t). \tag{6.12}$$

Définition 6.4. *On appelle* ***erreur de troncature (globale)*** *la quantité*

$$\tau = \max_{n=0,1,\ldots,N-1} |\tau_n|.$$

Quelques commentaires sont obligatoires. Clairement, la solution exacte $u(t_n)$ ne coïncide en général pas avec la solution numérique u^n. Elle ne satisfait donc pas l'équation (6.11). La définition (6.12) nous dit que *l'erreur de troncature locale est le résidu du schéma numérique lorsqu'on remplace la solution numérique u^n par la solution exacte $u(t_n)$*. Il est donc un indicateur de combien la solution exacte ne satisfait pas le schéma numérique.

On donne aussi une autre interprétation de l'erreur de troncature locale, dans le cas d'un schéma explicite. Pour chaque t_n, $n = 0, 1, \ldots, N-1$, on introduit la quantité suivante :

$$\tilde{u}^{n+1} = u(t_n) + \Delta t \phi_f(u(t_n), t_n, \Delta t).$$

On note que $\tilde{u}^{n+1}$ est la solution numérique du schéma à un pas explicite à l'instant t_{n+1} si on prend à l'instant t_n la solution exacte $u(t_n)$. De plus,

$$\tau_n = \frac{u(t_{n+1}) - u(t_n)}{\Delta t} - \phi_f(u(t_n), t_n, \Delta t) = \frac{u(t_{n+1}) - \tilde{u}^{n+1}}{\Delta t}.$$

Donc, l'erreur de troncature locale τ_n mesure la distance entre la solution exacte $u(t_{n+1})$ et la solution numérique $\tilde{u}^{n+1}$ qu'on obtient en partant de la solution exacte $u(t_n)$ à l'instant t_n. Autrement dit, *l'erreur de troncature locale mesure l'erreur introduite par le schéma numérique dans un seul pas de temps, si on part de la solution exacte.*

Exemple 6.6 (Erreur de troncature du schéma d'Euler progressif)**.** *Pour le schéma d'Euler progressif, on a*

$$\tau_n = \frac{u(t_{n+1}) - u(t_n)}{\Delta t} - f(t_n, u(t_n)).$$

On rappelle maintenant l'erreur due à l'approximation aux différences finies progressives (voir Chapitre 3)

$$\left| u'(t_n) - \frac{u(t_{n+1}) - u(t_n)}{\Delta t} \right| \leq \frac{\Delta t}{2} \max_{t \in [t_n, t_{n+1}]} |u''(t)|.$$

Puisque $u(t)$ est la solution exacte *du problème de Cauchy* (6.1)*, elle satisfait $u'(t_n) = f(t_n, u(t_n))$ et donc*

$$|\tau_n| = \left| \frac{u(t_{n+1}) - u(t_n)}{\Delta t} - u'(t_n) \right| \leq \frac{\Delta t}{2} \max_{t \in [t_n, t_{n+1}]} |u''(t)|.$$

Finalement

$$\tau \leq C \Delta t \qquad \text{où } C = \frac{1}{2} \max_{t \in [0,T]} |u''(t)|.$$

Exemple 6.7 (Erreur de troncature du schéma de Crank Nicolson)**.** *Pour le schéma de Crank Nicolson, on a*

$$\tau_n = \frac{u(t_{n+1}) - u(t_n)}{\Delta t} - \frac{1}{2} f(t_n, u(t_n)) - \frac{1}{2} f(t_{n+1}, u(t_{n+1})).$$

On rappelle l'erreur de la formule de quadrature du trapèze sur un seul intervalle (voir Chapitre 3)

$$\left| \int_{t_n}^{t_{n+1}} g(s)ds - \frac{\Delta t}{2} \left(g(t_n) + g(t_{n+1}) \right) \right| \leq C \max_{t \in [t_n, t_{n+1}]} |g''(t)| \Delta t^3.$$

Si on prend $g(t) = u'(t) = f(t, u(t))$*, on a donc*

$$|\tau_n| = \frac{1}{\Delta t} \left| \int_{t_n}^{t_{n+1}} u'(s)ds - \frac{\Delta t}{2} \left[f(t_n, u(t_n)) + f(t_{n+1}, u(t_{n+1})) \right] \right| \leq C \max_{t \in [t_n, t_{n+1}]} |u'''(t)| \Delta t^2$$

et

$$\tau \leq C \max_{t \in [0,T]} |u'''(t)| \Delta t^2.$$

On a, en général, le résultat suivant :

Résultat *Si l'erreur de troncature est d'ordre p, c'est-à-dire* $\tau \leq C \Delta t^p$*, alors le schéma numérique converge avec ordre p.*

Preuve pour Euler progressif. On montre ce résultat pour le schéma d'Euler progressif. On introduit comme avant la quantité :

$$\tilde{u}^{n+1} = u(t_n) + \Delta t f(t_n, u(t_n)), \qquad n = 0, \ldots, N-1$$

de sorte que

$$\tau_n = \frac{u(t_{n+1}) - \tilde{u}^{n+1}}{\Delta t}.$$

On regarde maintenant l'erreur totale $u(t_{n+1}) - u^{n+1}$. On peut décomposer cette erreur en deux contributions

$$\begin{aligned}
|u(t_{n+1}) - u^{n+1}| &\leq \underbrace{|u(t_{n+1}) - \tilde{u}^{n+1}|}_{= \Delta t \tau_n} + |\tilde{u}^{n+1} - u^{n+1}| \\
&\leq \Delta t |\tau_n| + |u(t_n) - u^n + \Delta t (f(t_n, u(t_n)) - f(t_n, u_n))| \\
&\leq \Delta t |\tau_n| + |u(t_n) - u^n| + \Delta t \underbrace{|f(t_n, u(t_n)) - f(t_n, u_n)|}_{\leq L |u(t_n) - u^n| \text{ car } f \text{ Lipschitz.}} \\
&\leq \Delta t \tau + (1 + L \Delta t) |u(t_n) - u^n|.
\end{aligned}$$

Soit $\varepsilon_n = |u(t_n) - u^n|$ l'erreur à l'instant t_n. La relation précédente permet de lier l'erreur ε_{n+1} avec l'erreur ε_n au pas de temps précédent :

$$\begin{aligned}
\varepsilon_{n+1} &\leq \Delta t\tau + (1 + L\Delta t)\varepsilon_n \\
&\leq \Delta t\tau + (1 + L\Delta t)\Delta t\tau + (1 + L\Delta t)^2\varepsilon_{n-1} \\
&\quad \cdots \\
&\leq \sum_{i=0}^{n}(1 + L\Delta t)^i\Delta t\tau + (1 + L\Delta t)^{n+1}\varepsilon_0 \\
&= \frac{(1 + L\Delta t)^{n+1} - 1}{L}\tau + (1 + L\Delta t)^{n+1}\varepsilon_0.
\end{aligned}$$

On fait maintenant l'approximation $(1 + x) \leq e^x$ pour conclure

$$\varepsilon_N \leq \frac{e^{L\Delta tN} - 1}{L}\tau + e^{L\Delta tN}\varepsilon_0.$$

On remarque que $\varepsilon_0 = u(0) - u^0 = 0$ et $N\Delta t = T$. Donc

$$|u(T) - u^N| \leq \frac{e^{LT} - 1}{L}\tau$$

ce qui montre que l'erreur au temps final est du même ordre que l'erreur de troncature. □

L'ordre des schémas à un pas qu'on a vu dans la section précédente est résumé dans le tableau suivant

Schéma	ordre
Euler progressif	1
Euler rétrograde	1
Crank Nicolson	2
Heun	2

6.4 Stabilité absolue

On considère un système d'équations différentielles *autonome*

$$\begin{cases} \dfrac{d\mathbf{u}(t)}{dt} = \mathbf{f}(\mathbf{u}(t)), \quad t > 0 \\ \mathbf{u}(0) = \mathbf{u}_0 \end{cases} \tag{6.13}$$

où la fonction $\mathbf{f}$ ne dépend pas explicitement du temps. Le système de l'exemple 6.2 est un exemple de système autonome.

Les valeurs $\bar{\mathbf{u}}$ (si elles existent) pour lesquelles $\mathbf{f}(\bar{\mathbf{u}}) = \mathbf{0}$ sont appelées *points d'équilibre* car si on prend $\mathbf{u}_0 = \bar{\mathbf{u}}$, la solution de (6.13) est $\mathbf{u}(t) = \bar{\mathbf{u}}$, $\forall t > 0$, et le système ne s'éloigne pas de l'équilibre.

Un point d'équilibre est dit un *attracteur global* si, pour tout $\mathbf{u}_0$, la solution de (6.13) satisfait $\lim_{t\to\infty} \mathbf{u}(t) = \bar{\mathbf{u}}$. Autrement dit, toute solution de (6.13) tend asymptotiquement à la valeur d'équilibre. On se pose maintenant la question suivante.

Question : *étant donné un système autonome* (6.13) *avec un attracteur global* $\bar{\mathbf{u}}$ *de tel sorte que*

$$\forall\, \mathbf{u}_0, \qquad \lim_{t\to\infty} \mathbf{u}(t) = \bar{\mathbf{u}},$$

est-il vrai que la solution $\mathbf{u}^n$ *calculée par une méthode numérique satisfait également*

$$\forall\, \mathbf{u}_0, \qquad \lim_{n\to\infty} \mathbf{u}^n = \bar{\mathbf{u}}\ ?$$

Autrement dit, si la solution exacte tend vers la valeur d'équilibre, est-ce que la solution numérique tend aussi à la valeur d'équilibre ? On souhaiterait bien avoir cette propriété mais on verra que ceci n'est pas toujours le cas.

La réponse à cette question est en général très difficile à donner, mais il y a quand même un cas où on sait donner une réponse précise. C'est le cas d'un système *linéaire* de taille m :

$$\frac{d\mathbf{u}(t)}{dt} = A\mathbf{u}(t) + \mathbf{b}, \;\; t > 0, \qquad \mathbf{u}(0) = \mathbf{u}_0, \tag{6.14}$$

où $\mathbf{u}(t) = (u_1(t), \ldots, u_m(t))^T$, $A \in \mathbb{R}^{m\times m}$ et $\mathbf{b} \in \mathbb{R}^m$.

On a le résultat suivant concernant les points d'équilibre de (6.14) :

Theorème 6.2. *Soient* $\lambda_i(A)$ *les valeurs propres de la matrice* $A \in \mathbb{R}^{m\times m}$. *Si* $\Re(\lambda_i(A)) < 0$ *pour tout* $i = 1, \ldots, m$ *alors il existe un seul point d'équilibre* $\bar{\mathbf{u}} = -A^{-1}\mathbf{b}$ *qui est un attracteur global, c'est-à-dire*

$$\forall\, \mathbf{u}_0, \qquad \lim_{t\to\infty} \mathbf{u}(t) = \bar{\mathbf{u}} = -A^{-1}\mathbf{b}.$$

On considère maintenant un schéma numérique.

Définition 6.5. *Sous les hypothèses du Théorème 6.2*

- *on dit qu'un schéma numérique est* ***absolument stable*** *pour un* Δt *fixé si*

$$\forall\, \mathbf{u}_0 \qquad \lim_{n\to\infty} \mathbf{u}^n = \bar{\mathbf{u}} = -A^{-1}\mathbf{b}.$$

- *On dit que le schéma est* ***inconditionnellement absolument stable*** *(ou A-stable) s'il est stable pour tout* $\Delta t > 0$.
- *Si, au contraire, il est stable seulement pour certain* Δt, *on dit qu'il est* ***conditionnellement absolument stable***.

Pour vérifier si une méthode numérique est absolument stable, il faut donc étudier son comportement dans l'approximation du système (6.14). Dans cette analyse, le terme $\mathbf{b}$ n'a pas d'influence. En fait, on peut toujours se ramener au cas homogène ($\mathbf{b} = 0$) en étudiant la solution $\mathbf{v}(t) = \mathbf{u}(t) - \bar{\mathbf{u}}$ qui satisfait le système homogène

$$\frac{d\mathbf{v}(t)}{dt} = A\mathbf{v}(t), \quad t > 0, \qquad \mathbf{v}(0) = \mathbf{u}_0 - \bar{\mathbf{u}}.$$

Étudions d'abord le cas scalaire, puis le cas vectoriel.

6.4.1 Problème modèle scalaire

$$\frac{du(t)}{dt} = \lambda u(t), \quad t > 0, \qquad u(0) = u_0 \tag{6.15}$$

et $\lambda < 0$, dont la solution exacte $u(t) = u_0 e^{\lambda t} \to 0$ pour $t \to \infty$.

Lemme 6.3. *Le* Schéma d'Euler progressif *appliqué au problème modèle* (6.15) *est absolument stable si*

$$\Delta t < \frac{2}{|\lambda|}. \tag{6.16}$$

Démonstration.

$$u^n = u^{n-1} + \Delta t \lambda u^{n-1} = (1 + \Delta t \lambda) u^{n-1} = \ldots = (1 + \Delta t \lambda)^n u_0.$$

Donc $u^n \xrightarrow{n\to\infty} 0$ si et seulement si $|1 + \Delta t \lambda| < 1$, ce qui donne la condition (6.16). □

Lemme 6.4. *Le* Schéma d'Euler rétrograde *appliqué au problème modèle* (6.15) *est inconditionnellement absolument stable.*

Démonstration.

$$u^n = u^{n-1} + \Delta t \lambda u^n \qquad \Longrightarrow \qquad u^n = \frac{u^{n-1}}{1 - \Delta t \lambda} = \ldots = \frac{1}{(1 - \Delta t \lambda)^n} u_0.$$

Puisque $\lambda < 0$, on a $|\frac{1}{1-\Delta t\lambda}| < 1$ et $u^n \xrightarrow{n\to\infty} 0$ pour tout $\Delta t > 0$. □

6.4.2 Problème modèle vectoriel

$$\frac{d\mathbf{u}(t)}{dt} = A\mathbf{u}(t), \quad t > 0, \qquad \mathbf{u}(0) = \mathbf{u}_0 \tag{6.17}$$

où $\Re(\lambda_i(A)) < 0$, pour tout $i = 1, \ldots, m$.

Pour le schéma d'Euler progressif on a le résultat suivant qui généralise celui énoncé dans les Lemmes 6.3 et 6.4 :

Lemme 6.5. *Le* Schéma d'Euler progressif *appliqué au problème modèle* (6.17) *est absolument stable si*

$$\forall \lambda_i(A), \qquad |1+\Delta t\lambda_i(A)|<1 \qquad \Longleftrightarrow \qquad \Delta t < \min_{i=1,\dots,m} \frac{2|\Re\lambda_i|}{|\lambda_i|^2}. \quad (6.18)$$

Par contre, le schéma d'Euler rétrograde appliqué au problème modèle (6.17) *est inconditionnellement absolument stable.*

Démonstration. On ne considère dans cette démonstration que le cas où la matrice A est diagonalisable : $A = VDV^{-1}$, avec D matrice diagonale des valeurs propres de A et V matrice contenant les vecteurs propres de A. Le schéma d'Euler progressif s'écrit

$$\frac{\mathbf{u}^{n+1}-\mathbf{u}^n}{\Delta t} = A\mathbf{u}^n = VDV^{-1}\mathbf{u}^n.$$

Si on multiplie l'équation précédente par V^{-1} et on note $\mathbf{w}^n = V^{-1}\mathbf{u}^n$, on a

$$\frac{\mathbf{w}^{n+1}-\mathbf{w}^n}{\Delta t} = V^{-1}VDV^{-1}\mathbf{u}^n = D\mathbf{w}^n.$$

Il s'agit d'un système d'équations diagonal dont la i-ème équation est

$$\frac{w_i^{n+1}-w_i^n}{\Delta t} = \lambda_i w_i^n, \qquad \Longrightarrow \qquad w_i^{n+1} = (1+\lambda_i\Delta t)w_i^n, \;\; i=1,\dots,m$$

et chaque composante du vecteur $\mathbf{w}^n$, et donc aussi du vecteur $\mathbf{u}^n = V\mathbf{w}^n$, tend vers zéro pour $n\to\infty$ si et seulement si

$$|1+\lambda_i\Delta t|<1, \qquad \forall i=1,\dots,m,$$

ce qui équivaut à la condition

$$(1+\Delta t\Re(\lambda_i))^2 + \Delta t^2\Im(\lambda_i)^2 < 1$$

$$\Longleftrightarrow \quad 2\Re(\lambda_i) + \Delta t|\lambda_i|^2 < 0,$$

ce qui donne la condition (6.18) puisque $\Re(\lambda_i)<0$ par hypothèse et $\Delta t>0$.

Si on applique, par contre, le schéma d'Euler rétrograde

$$\frac{\mathbf{u}^{n+1}-\mathbf{u}^n}{\Delta t} = A\mathbf{u}^{n+1} = VDV^{-1}\mathbf{u}^{n+1},$$

et qu'on effectue le changement de variable $\mathbf{w}^n = V^{-1}\mathbf{u}^n$, on obtient

$$\frac{\mathbf{w}^{n+1}-\mathbf{w}^n}{\Delta t} = D\mathbf{w}^{n+1},$$

ce qui donne par composantes

$$\frac{w_i^{n+1} - w_i^n}{\Delta t} = \lambda_i w_i^{n+1}, \qquad \Longrightarrow \qquad w_i^{n+1} = \frac{1}{1 - \lambda_i \Delta t} w_i^n, \;\; i = 1, \ldots, m.$$

Puisque par hypothèse $\Re(\lambda_i) < 0$ pour tout $i = 1, \ldots, m$, on a

$$|w_i^{n+1}| = \frac{|w_i^n|}{|1 - \lambda_i \Delta t|} = \frac{|w_i^n|}{\sqrt{(1 - \Re(\lambda_i)\Delta t)^2 + \Im(\lambda_i)^2 \Delta t^2}} < |w_i^n|$$

et $\mathbf{w}^n = V^{-1}\mathbf{u}^n \to 0$ pour $n \to \infty$ pour tout $\Delta t > 0$ ce qui montre que le schéma est A-stable. □

6.5 Contrôle de l'erreur : un algorithme adaptatif

Dans cette section, on se pose la question de comment choisir en pratique le pas de temps Δt d'un schéma à un pas pour discrétiser une équation différentielle ordinaire.

Supposons qu'on soit intéressé à la solution du problème de Cauchy à l'instant final $t = T$. D'un coté, le pas de temps Δt doit être choisi de façon à avoir une erreur finale $\varepsilon_N = |u(T) - u^N|$ plus petite qu'une tolérance donnée. De l'autre coté, le pas de temps Δt doit garantir la stabilité du schéma numérique.

Prenons par exemple le schéma d'Euler progressif. On rappelle (voir la Section 6.3) que l'erreur $\varepsilon_{n+1} = |u(t_{n+1}) - u^{n+1}|$ à l'instant t_{n+1} satisfait l'inégalité

$$\varepsilon_{n+1} \leq \Delta t |\tau_n| + (1 + L\Delta t)\varepsilon_n$$

où L est la constante de Lipschitz de la fonction $f(t, u)$ et τ_n est l'erreur de troncature locale

$$\tau_n = \frac{u(t_{n+1}) - u(t_n)}{\Delta t} - f(t_n, u(t_n)).$$

L'erreur à l'instant t_{n+1} est donc la somme d'une *erreur locale* $\Delta t|\tau_n|$ plus l'accumulation des erreurs aux instants précédents.

On va faire maintenant l'hypothèse (assez forte) que l'accumulation des erreurs est négligeable, c'est-à-dire, la constante L de Lipschitz est soit très petite soit négative. On a alors pour l'erreur à l'instant final

$$\varepsilon_N \leq \Delta t |\tau_{N-1}| + (1 + L\Delta t)\varepsilon_{N-1} \approx \Delta t |\tau_{N-1}| + \varepsilon_{N-1} \approx \sum_{n=0}^{N-1} \Delta t |\tau_n|$$

où on a supposé que l'erreur initiale était nulle, c'est-à-dire $\varepsilon_0 = 0$.

Si on souhaite avoir une erreur finale inférieure à une tolérance fixée, $\varepsilon_N \leq tol$, il suffit d'imposer que

$$|\tau_n| \leq \frac{tol}{T} \tag{6.19}$$

de sorte que

$$\varepsilon_N \approx \sum_{n=0}^{N-1} \Delta t \frac{tol}{T} = \frac{tol}{T} N \Delta t = tol.$$

Soit Δt_n le pas de temps utilisé entre t_n et t_{n+1}. Puisque

$$|\tau_n| = \frac{1}{2} \Delta t_n \max_{t \in [t_n, t_{n+1}]} |u''(t)|$$

(voir exemple 6.6), on peut choisir

$$\Delta t_n \leq \frac{2tol}{T \max_{t \in [t_n, t_{n+1}]} |u''(t)|} \tag{6.20}$$

pour satisfaire (6.19). Cette stratégie nous porte naturellement à utiliser un pas de temps différent à chaque instant. On appelle une telle stratégie *adaptative* car le pas de temps est adapté à la solution de façon à satisfaire une certaine tolérance.

Le problème de cette stratégie est qu'on ne connaît pas la solution exacte ni sa dérivée deuxième et donc on ne peut pas calculer dans la pratique le pas de temps optimal (6.20).

On procède donc de manière différente : on essaye d'estimer l'erreur de troncature locale. Soit $\hat{\tau}_n$ une telle estimation obtenue en utilisant un pas de temps Δt_n. Si $\hat{\tau}_n \leq tol/T$ on accepte le pas de temps Δt_n ainsi que la solution u^{n+1} calculée et on passe à l'instant suivant. Si par contre $\hat{\tau}_n > tol/T$, on réduit alors Δt_n (pas exemple on le divise par deux) et on recalcule la solution u^{n+1} avec le nouveau pas de temps.

Il pourrait aussi arriver que $\hat{\tau}_n \ll tol/T$, c'est-à-dire l'erreur estimée est beaucoup plus petite que la tolérance fixée. Dans ce cas, il est judicieux d'incrémenter le pas de temps pour l'instant suivant (par exemple le doubler).

Il reste à voir comment on peut estimer l'erreur de troncature locale $\hat{\tau}_n$. Pour cela, on peut utiliser un schéma d'ordre plus élevé, comme par exemple le schéma d'Heun. Soit

$$\hat{u}^{n+1} = u^n + \frac{\Delta t_n}{2} f(t_n, u^n) + \frac{\Delta t_n}{2} f(t_n + \Delta t_n, u^n + \Delta t_n f(t_n, u^n))$$

la solution à l'instant $t_{n+1} = t_n + \Delta t_n$ calculée par le schéma de Heun. On peut alors estimer l'erreur de troncature locale par la formule

$$\hat{\tau}_n = \frac{|\hat{u}^{n+1} - u^{n+1}|}{\Delta t}$$

On résume la stratégie adaptative dans l'algorithme suivant :

Algorithme 6.1 : Algorithme avec pas de temps adaptatif : l'erreur de troncature locale du schéma d'Euler progressif est estimée en utilisant le schéma d'Heun.

Données : $f(t,u)$, u_0, $[t_0,T]$, Δt_{init}, tol
Résultat : N, $[t_0,t_1,\ldots,t_N=T]$, $[u_0,u_1,\ldots,u_N]$
$n=0$;
$\Delta t_0=\Delta t_{init}$ `// pas de temps initial`;
tant que $t_n<T$ **faire**
  $u^{n+1}=u^n+\Delta t_n f(t_n,u^n)$ `// pas d'Euler progr.`;
  $\hat{u}^{n+1}=u^n+\frac{\Delta t_n}{2}f(t_n,u^n)+\frac{\Delta t_n}{2}f(t_n+\Delta t_n,u^{n+1})$ `// pas d'Heun`;
  $\hat{\tau}_n=\frac{|\hat{u}^{n+1}-u^{n+1}|}{\Delta t_n}$ `// estimation de l'erreur`;
  si $\hat{\tau}_n>tol/T$ **alors** `// il faut réduire le pas de temps`
    $\Delta t_n=\Delta t_n/2$;
  sinon `// on accepte la solution` u_{n+1}
    $t_{n+1}=t_n+\Delta t_n$;
    si $\hat{\tau}_n<tol/2T$ **alors** `// on augmente le pas de temps`
      $\Delta t_{n+1}=\min\{2\Delta t_n,T-t_{n+1}\}$;
    sinon
      $\Delta t_{n+1}=\min\{\Delta t_n,T-t_{n+1}\}$;
    fin
    $n=n+1$;
  fin
fin

0	0
	1

Euler explicite

1	1
	1

Euler implicite

0	0	
1	$\frac{1}{2}$	$\frac{1}{2}$
	$\frac{1}{2}$	$\frac{1}{2}$

Crank Nicolson

0	0	
1	1	0
	$\frac{1}{2}$	$\frac{1}{2}$

Heun

TABLE 6.1 – Tableaux de Butcher correspondantes aux méthodes d'Euler explicite, Euler implicite, Crank-Nicolson et Heun.

6.6 Généralisations

6.6.1 Méthodes de Runge-Kutta

Toutes les méthodes à un pas qu'on a vu jusqu'à présent appartiennent à une famille plus générale dites *méthodes de Runge Kutta*, qui ont la forme suivante

$$\begin{aligned} K_i &= f\left(t_n + c_i\Delta t, u^n + \Delta t \sum_{j=1}^{s} a_{ij}K_j\right), \qquad i = 1, \ldots, s \\ u^{n+1} &= u^n + \Delta t \sum_{i=1}^{s} b_i K_i, \quad n \geq 0. \end{aligned} \tag{6.21}$$

La méthode précédente construit la nouvelle estimation u^{n+1} à partir de l'estimation au pas précédent u^n et des s évaluations K_i, $i = 1, \ldots, s$ de la fonction $f(\cdot, \cdot)$. Elle est donc appelée *méthode de Runge Kutta à s étapes*.

Une méthode de Runge Kutta est complètement idéntifiée pas les coefficients $\{a_{ij}\}$, $\{b_i\}$ et $\{c_i\}$ qui sont usuellement groupés dans un tableau (dit tableau de Butcher)

$$\begin{array}{c|ccc} c_1 & a_{11} & \cdots & a_{1s} \\ \vdots & \vdots & & \vdots \\ c_s & a_{s1} & \cdots & a_{ss} \\ \hline & b_1 & \cdots & b_s \end{array}$$

La méthode est explicite si $a_{ij} = 0$ pour tout $j \geq i$, car chaque K_i peut être calculé de façon explicite à partir des K_j, $j < i$, calculés précédemment. Dans ce cas, il faut s'attendre à ce que le schéma soit seulement conditionnellement stable.

Pour les schémas d'Euler explicite, Euler implicite, Crank-Nicolson et Heun on a les tableaux 6.1 (vérifiez-le comme exercice).

`Python` contient plusieures méthodes de Runge-Kutta déjà implémentées. Par exemple, la commande `ode45` utilise une méthode de Runge-Kutta explicite à 5 étapes. Cette méthode a deux choix possibles des coéfficients $\{b_i\}$ qui produisent, respectivement, une méthode d'ordre 5 et une méthode d'ordre 4. La différence entre les solutions obtenues par les deux méthodes

peut être utilisée pour estimer l'erreur de troncature locale et adapter par consequent le pas de temps Δt.

6.6.2 Méthodes multi-pas

Toutes les méthodes qu'on a vues jusqu'à maintenant sont des méthodes à un pas, c'est-à-dire que la solution u^{n+1} est calculée seulement à partir de la solution u^n (et d'un certain nombre d'évaluations de la focntion $f(t,u)$).

On peut aussi construire des *méthodes multi-pas* où la solution u^{n+1} est construite à partir d'un certain nombre de solutions précédentes $u^n, u^{n-1}, \ldots, u^{n+1-p}$.

La forme générale d'une méthode à p-pas est

$$a_0 u^{n+1} - \sum_{j=1}^{p} a_j u^{n+1-j} = \Delta t \sum_{k=-1}^{p} b_k f(t_{n-k}, u^{n+1-k}), \qquad n \geq p-1. \quad (6.22)$$

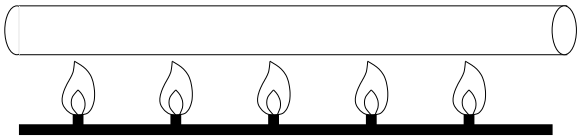
Une méthode à p-pas nécessite les p premières évaluations $u_0, u_1, \ldots, u_{p-1}$ pour pouvoir démarrer. Celles-ci peuvent être calculées, par exemple, par une méthode de Runge-Kutta.

Chapitre 7

Problèmes aux limites en dimension un

7.1 Exemple : Équation de la chaleur

Considérons une barre métallique de longueur L qui occupe l'intervalle $[0, L]$. On note $T(x,t)$ la température de la barre au point $x \in [0,L]$ et à l'instant t et $J(x,t)$ le flux de chaleur. Soit encore $f(x,t)$ une source de chaleur (par exemple générée par une flamme, voir figure ci-dessous), ρ la densité de masse volumique et c_p la chaleur spécifique massique.



La variation de température en temps dans un morceau $[x, x+dx]$ de la barre est alors donnée par

$$\rho c_p \frac{\partial T}{\partial t}(x,t)dx = J(x,t) - J(x+dx,t) + f(x,t)dx.$$

En divisant par dx et en prenant la limite pour $dx \to 0$, l'équation précédente devient

$$\rho c_p \frac{\partial T}{\partial t}(x,t) = -\frac{\partial J}{\partial x}(x,t) + f(x,t).$$

La loi de Fourier dit que le flux de chaleur est proportionnel au gradient de température

$$J(x,t) = -k\frac{\partial T}{\partial x}(x,t) \tag{7.1}$$

où k est la conductivité thermique. Finalement, l'équation qui décrit l'évolution de la température dans la barre (*équation de la chaleur 1D*) est

$$\rho c_p \frac{\partial T}{\partial t}(x,t) - k\frac{\partial^2 T}{\partial x^2}(x,t) = f(x,t), \qquad x \in (0,L), \;\; t > 0. \tag{7.2}$$

Il s'agit d'une équation aux dérivées partielles, l'inconnue étant la distribution de température $T(x,t)$ dans la barre à chaque instant. Pour trouver une solution, il faudra encore donner la distribution de la température au temps initial et préciser ce qui se passe aux deux extrémités de la barre. Par exemple, si la barre est en contact avec des réservoirs de chaleur de température constante, on peut ajouter les *conditions aux limites*

$$T(0,t) = T_g, \qquad T(L,t) = T_d, \qquad t > 0.$$

Ces conditions sont appelées *conditions de Dirichlet*.

Une autre possibilité est que la barre soit thermiquement isolée. Dans ce cas, il n'y aura pas de flux de chaleur aux extrémités de la barre et de bonnes conditions aux limites sont

$$J(0,t) = -k\frac{\partial T}{\partial x}(0,t) = 0, \qquad J(L,t) = -k\frac{\partial T}{\partial x}(L,t) = 0, \qquad t > 0.$$

Ces conditions sont appelées *conditions de Neumann* et font intervenir la valeur de la dérivée de la solution aux extrémités, au lieu de la valuer de la solution elle-même.

Bien évidemment, on peut mélanger ces types de conditions et considérer une condition de Dirichlet à l'extrémité de gauche et une condition de Neumann à l'extrémité de droite (ou l'inverse).

Finalement, si on ne s'intéresse qu'à la distribution de température à l'équilibre (en supposant que la source de chaleur ne varie pas en temps), on peut résoudre le *problème stationnaire*

$$-k\frac{d^2 T}{dx^2}(x) = f(x), \qquad x \in (0,L), \tag{7.3}$$

soit avec des conditions aux limites de Dirichlet :

$$\begin{cases} -k\frac{d^2T}{dx^2}(x) = f(x), & x \in (0,L), \\ T(0) = T_g, \;\; T(L) = T_d, \end{cases} \tag{7.4}$$

soit avec des conditions de Neumann :

$$\begin{cases} -k\frac{d^2T}{dx^2}(x) = f(x), & x \in (0,L), \\ k\frac{dT}{dx}(0) = J_g, \;\; k\frac{dT}{dx}(L) = J_d. \end{cases} \tag{7.5}$$

L'équation (7.3) est une équation différentielle du second'ordre. Les problèmes (7.4) et (7.5), avec des conditions aux bords, sont appelés *problèmes aux limites*.

La même équation (7.3) pourrait aussi décrire le mouvement d'un mobile le long d'une droite, où $T(x)$ représente dans ce cas la position du mobile à l'instant x, k sa masse et f la force agissant sur le mobile.

Toutefois, il y a une très grande différence entre l'exemple de la chaleur et celui du mobile. Dans le premier exemple, la physique du problème nous suggère d'ajouter une condition à l'extrémité de gauche et une condition à l'extrémité de droite. On obtient ainsi un problème aux limites.

Par contre, dans l'exemple du mobile, on ajoute plutôt deux conditions à l'instant initial $x = 0$ qui représentent la position et la vitesse. On obtiendrait ainsi un *problème aux valeurs initiales* comme ceux qu'on a analysés dans le chapitre précédent.

7.2 Approximation par différences finies du problème de la chaleur stationnaire

Considérons le problème de la chaleur stationnaire monodimensionnel, avec conditions au bord de Dirichlet. Ici et dans la suite, on dénote la fonction inconnue par $u(x)$ et pour simplifier l'exposé, on pose $k = 1$. On va aussi noter les conditions de Dirichlet par α et β.

$$\begin{cases} -\dfrac{d^2u}{dx^2}(x) = f(x), & x \in (0, L), \\ u(0) = \alpha, \;\; u(L) = \beta. \end{cases} \tag{7.6}$$

Pour calculer une solution approchée, on procède de la façon suivante : on divise l'intervalle $[0, L]$ en $n + 1$ sous-intervalles $I_j = [x_{j-1}, x_j]$ de longueur $h = \frac{L}{n+1}$, où $x_j = jh$, et on cherche une solution approchée $u_j \approx u(x_j)$ aux noeuds x_j.

On remarque qu'aux noeuds $x_0 = 0$ et $x_{n+1} = L$, la solution est connue grâce aux conditions aux limites de Dirichlet. On pose alors $u_0 = \alpha$ et $u_{n+1} = \beta$, les inconnues du problème étant seulement les valeurs u_j pour $j = 1, \ldots, n$ aux noeuds internes de l'intervalle.

Dans chaque noeud interne x_j, $j = 1, \ldots, n$, la solution exacte satisfait l'équation

$$-\frac{d^2u}{dx^2}(x_j) = f(x_j).$$

L'idée est maintenant de remplacer la dérivée seconde exacte par une différence finie

$$\frac{d^2u}{dx^2}(x_j) \approx \frac{u(x_{j-1}) - 2u(x_j) + u(x_{j+1})}{h^2}.$$

On obtient ainsi le schéma suivant

$$\begin{cases} \dfrac{-u_{j-1} + 2u_j - u_{j+1}}{h^2} = f(x_j), & j = 1, \ldots, n \\ u_0 = \alpha, \;\; u_{n+1} = \beta. \end{cases} \tag{7.7}$$

En tenant compte des conditions aux limites, la première et la dernière équation peuvent être réécrites comme suit

$$\begin{aligned} \frac{-u_0 + 2u_1 - u_2}{h^2} = f(x_1) \quad &\Longleftrightarrow \quad \frac{2u_1 - u_2}{h^2} = f(x_1) + \frac{\alpha}{h^2} \\ \frac{-u_{n-1} + 2u_n - u_{n+1}}{h^2} = f(x_n) \quad &\Longleftrightarrow \quad \frac{-u_{n-1} + 2u_n}{h^2} = f(x_n) + \frac{\beta}{h^2} \end{aligned}$$

et donc le système (7.7) devient

$$\begin{cases} \dfrac{2u_1 - u_2}{h^2} = f(x_1) + \dfrac{\alpha}{h^2} & \text{pour } j = 1 \\ \dfrac{-u_{j-1} + 2u_j - u_{j+1}}{h^2} = f(x_j), & \text{pour } j = 2, \ldots, n-1 \\ \dfrac{-u_{n-1} + 2u_n}{h^2} = f(x_n) + \dfrac{\beta}{h^2} & \text{pour } j = n. \end{cases} \tag{7.8}$$

Si on introduit le vecteur des inconnues $\mathbf{u} = [u_1, \ldots, u_n]^T$, le système linéaire (7.8) peut être écrit sous forme matricielle

$$A\mathbf{u} = \tilde{\mathbf{f}}$$

où la matrice $A \in \mathbb{R}^{n \times n}$ et le terme de droite $\tilde{\mathbf{f}} \in \mathbb{R}^n$ sont

$$A = \frac{1}{h^2} \begin{bmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & -1 & \ddots & \ddots & \\ & & \ddots & \ddots & -1 \\ & & & -1 & 2 \end{bmatrix}, \qquad \tilde{\mathbf{f}} = \begin{bmatrix} f(x_1) + \frac{\alpha}{h^2} \\ f(x_2) \\ \vdots \\ f(x_{n-1}) \\ f(x_n) + \frac{\beta}{h^2} \end{bmatrix}. \tag{7.9}$$

Pour résoudre de façon approchée le problème (7.6) par un schéma aux différences finies, il faut donc résoudre un système linéaire.

On remarque que la matrice est symétrique et tridiagonale. De plus, on peut montrer qu'elle est aussi définie positive. On peut donc utiliser la méthode de factorisation LU (ou bien de Cholesky) de façon très efficace.

D'un autre coté, on peut montrer que le nombre de conditionnement de la matrice A est

$$\mathcal{K}(A) = O(h^{-2})$$

et la matrice A devient de plus en plus mal conditionnée lorsque h tend vers zéro (c'est-à-dire, lorsqu'on augment le nombre de points).

7.2.1 Stabilité et analyse de l'erreur

Pour le système (7.8), on peut montrer le résultat de stabilité suivant :

Theorème 7.1. *Pour tout* $\mathbf{f} = [f(x_1), \ldots, f(x_n)]^T \in \mathbb{R}^n$ *et tout* $\alpha, \beta \in \mathbb{R}$, *le système* (7.8) *admet une solution unique* $\mathbf{u} = [u_1, \ldots, u_n]^T \in \mathbb{R}^n$ *qui satisfait*

$$\max_{j=1,\ldots,n} |u_j| \leq \frac{1}{8} \max_{j=1,\ldots,n} |f(x_j)| + \max\{|\alpha|, |\beta|\} \tag{7.10}$$

Le résultat (7.10) nous dit que la taille de la solution $\mathbf{u}$ ne peut pas être trop grande et qu'elle est controlée par la taille du vecteur $\mathbf{f}$ ainsi que la taille des valeurs α et β aux bords.

On s'intéresse maintenant à étudier le comportement de l'erreur $e_j = u(x_j) - u_j$, $j = 1, \ldots, n$, commise par l'approximation aux différences finies (7.8) par rapport à h (ou de façon équivalente, par rapport au nombre de points de discrétisation). On introduit ici des notions très similaires à celles déjà utilisées dans l'étude des schémas numériques pour les équations différentielles ordinaires (voir Chapitre 6).

Définition 7.1. *Un schéma numérique pour approcher le problème aux limites* (7.6) *est dit* ***convergent avec ordre p*** *s'il existe une constante* $C > 0$ *telle que*

$$\max_{j=0,\ldots,n+1} |u(x_j) - u_j| \leq Ch^p$$

pourvu que la solution exacte soit suffisamment régulière.

On introduit aussi la notion de

Définition 7.2 (**Erreur de troncature locale**). *Soit* $u(x_j)$, $j = 0, \ldots, n+1$ *la solution exacte du problème* (7.6) *aux noeuds* $x_j = jh$. *Pour le schéma aux différences finies* (7.8), *on définit l'*erreur de troncature locale *au noeud* x_j *par*

$$\tau_j = \frac{-u(x_{j-1}) + 2u(x_j) - u(x_{j+1})}{h^2} - f(x_j), \quad j = 1, \ldots, n. \tag{7.11}$$

C'est-à-dire, l'erreur de troncature locale est le résidu du schéma numérique lorsqu'on remplace la solution approchée u_j *par la solution exacte* $u(x_j)$.

Puisque la solution exacte satisfait le problème $-\frac{d^2u}{dx^2}(x_j) = f(x_j)$, l'erreur de troncature locale

$$\tau_j = -\frac{u(x_{j-1}) - 2u(x_j) + u(x_{j+1})}{h^2} + \frac{d^2u}{dx^2}(x_j)$$

n'est rien d'autre que l'approximation de la dérivée seconde de u par la différence finie. On a le résultat (dont la démonstration est laissée comme exercice)

Résultat 7.1. *Pour l'erreur de troncature locale* (7.11) *on a*

$$\max_{j=1,\ldots,n} |\tau_j| \leq \frac{h^2}{12} \max_{x\in[0,L]} |u''''(x)| \tag{7.12}$$

pourvu que la solution exacte soit quatre fois continûment dérivable.

D'après la définition d'erreur de troncature locale, la solution exacte satisfait le problème discret

$$\frac{-u(x_{j-1}) + 2u(x_j) - u(x_{j+1})}{h^2} = f(x_j) + \tau_j, \qquad j = 1, \ldots, n \tag{7.13}$$

alors que la solution approchée u_j satisfait le problème

$$\frac{-u_{j-1} + 2u_j - u_{j+1}}{h^2} = f(x_j), \qquad j = 1, \ldots, n. \tag{7.14}$$

Si on soustrait (7.14) à (7.13), on obtient le système d'équations pour l'erreur

$$\begin{cases} \frac{-e_{j-1}+2e_j-e_{j+1}}{h^2} = \tau_j, \qquad j = 1, \ldots, n, \\ e_0 = e_{n+1} = 0. \end{cases} \tag{7.15}$$

Ce système a la même forme que le système (7.7), ou de façon équivalente le système (7.8), avec l'erreur de troncature locale comme terme de force et des conditions aux bords homogènes ($\alpha = \beta = 0$). On peut donc appliquer le résutlat de stabilité du theorème 7.1 pour conclure que

$$\max_{j=1,\ldots,n} |e_j| \leq \frac{1}{8} \max_{i=1,\ldots,n} |\tau_i|,$$

qui donne une estimation de l'erreur sur la solution en fonction de l'erreur de troncature locale maximale.

On arrive donc au résultat final résumé dans le théorème suivant :

Theorème 7.2. *Le schéma aux différences finies* (7.8) *pour résoudre le problème* (7.6) *converge avec order 2 et*

$$\max_{j=1,\ldots,n} |u(x_j) - u_j| \leq Ch^2 \tag{7.16}$$

où $C = \frac{1}{96} \max_{x\in[0,L]} |u''''(x)|$*, pourvu que la solution exacte soit quatre fois continûment dérivable.*

Grâce au résultat de stabilité du théorème 7.1, on note que l'erreur $\max_{j=1,\ldots,n} |u(x_j) - u_j|$ est du même ordre, en h, que l'erreur de troncature locale. Cette conclusion est également vraie pour beaucoup d'autres problèmes aux limites et discretisations aux différences finies. Il est donc souvent suffisant d'étudier l'erreur de troncature locale pour conclure sur l'ordre de la méthode.

7.2.2 Conditions aux limites de Neumann

Considérons le problème aux limites avec conditions mixtes Dirichlet-Neumann (Dirichlet à l'extrémité gauche et Neumann à l'extrémité droite) :

$$\begin{cases} -\frac{d^2u}{dx^2}(x) = f(x), \qquad x \in (0, L), \\ u(0) = \alpha, \quad u'(L) = \beta. \end{cases} \tag{7.17}$$

Comme dans la section précédente, on divise l'intervalle $[0, L]$ en $n+1$ sous-intervalles $I_j = [x_{j-1}, x_j]$ de longueur $h = \frac{L}{n+1}$, où $x_j = jh$, $j = 0, \ldots, n+1$, et on note $u_j \approx u(x_j)$ la solution approchée au noeud x_j.

Cette fois-ci, la solution au noeud x_{n+1} (extrémité de droite) n'est pas connue, donc le vecteur des inconnues sera $\mathbf{u} = [u_1, \ldots, u_{n+1}]^T \in \mathbb{R}^{n+1}$.

Dans chaque noeud interne x_j, $j = 1, \ldots, n$, on écrit l'équation approchée

$$\frac{-u_{j-1} + 2u_j - u_{j+1}}{h^2} = f(x_j), \qquad j = 1, \ldots, n.$$

Il reste à déterminer quelle équation écrire à l'extrémité de droite (c'est-à-dire au noeud x_{n+1}) et comment discrétiser la condition de Neumann $u'(L) = \beta$. Pour cela, on a deux possibilités qu'on illustre ci-dessous.

Première méthode : différence finie décentrée d'ordre 1

La première idée est de discrétiser la condition de Neumann $u'(x_{n+1}) = \beta$ par une différence finie. Puisque le vecteur des inconnues contient les valeurs $u_1, \ldots, u_{n+1}$, il vaut mieux utiliser une différence finie rétrograde qui donne l'équation

$$\frac{u_{n+1} - u_n}{h} = \beta. \tag{7.18}$$

On remarque que cette formule est seulement d'ordre 1. Avec ce choix, on a le système d'équations de taille $n+1$

$$\begin{cases} \dfrac{2u_1 - u_2}{h^2} = f(x_1) + \dfrac{\alpha}{h^2}, & \text{pour } j = 1 \\ \dfrac{-u_{j-1} + 2u_j - u_{j+1}}{h^2} = f(x_j), & \text{pour } j = 2, \ldots, n, \\ \dfrac{-u_n + u_{n+1}}{h} = \beta, & \text{pour } j = n+1. \end{cases}$$

qui peut être écrit sous forme matricielle

$$A\mathbf{u} = \tilde{\mathbf{f}}$$

où

$$A = \frac{1}{h^2}\begin{bmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & -1 & \ddots & \ddots & \\ & & \ddots & \ddots & -1 \\ & & & -1 & 1 \end{bmatrix}, \qquad \tilde{\mathbf{f}} = \begin{bmatrix} f(x_1) + \frac{\alpha}{h^2} \\ f(x_2) \\ \vdots \\ f(x_n) \\ \frac{\beta}{h} \end{bmatrix}. \tag{7.19}$$

Le désavantage de cette approche est que l'approximation qu'on a fait dans (7.18) est seulement d'ordre 1, c'est-à-dire, l'erreur de troncature local au noeud x_{n+1} est

$$|\tau_{n+1}| = \left|\frac{u(x_{n+1}) - u(x_n)}{h} - \beta\right| = \left|u'(L) + \frac{h}{2}u''(\xi) - \beta\right| \leq \frac{h}{2}\max_{x\in[x_n, x_{n+1}]}|u''(x)|.$$

et l'approximation du problème (7.17) sera globalement seulement d'ordre 1.

Deuxième méthode : différence finie centrée d'ordre 2 – noeud fantôme

Pour récupérer une approximation d'ordre 2, on aimerait pouvoir utiliser une différence finie centrée en x_{n+1}. Pour cela, on introduit le noeud $x_{n+2} = (n+2)h$ qui est hors de l'intervalle $[0, L]$ et la valeur correspondante u_{n+2}, de façon à pouvoir écrire une différence finie centrée

$$\frac{u_{n+2} - u_n}{2h} = \beta, \quad \text{pour } j = n+1. \tag{7.20}$$

On a toutefois ajouté une nouvelle inconnue u_{n+2} et il faut donc également ajouter une équation. Au noeud x_{n+1}, la discrétisation aux différences finies de l'équation $-u'' = f$ s'écrit :

$$\frac{-u_n + 2u_{n+1} - u_{n+2}}{h^2} = f(x_{n+1}). \tag{7.21}$$

De l'équation (7.20), on obtient $u_{n+2} = u_n + 2h\beta$ que l'on remplace dans (7.21) pour obtenir

$$\frac{-2u_n + 2u_{n+1}}{h^2} = f(x_{n+1}) + \frac{2\beta}{h}. \tag{7.22}$$

On obtient finalement le système d'équations de taille $n+1$

$$\begin{cases} \dfrac{2u_1 - u_2}{h^2} = f(x_1) + \dfrac{\alpha}{h^2}, & \text{pour} j = 1 \\ \dfrac{-u_{j-1} + 2u_j - u_{j+1}}{h^2} = f(x_j), & \text{pour } j = 2, \ldots, n, \\ \dfrac{-u_n + u_{n+1}}{h^2} = \dfrac{1}{2} f(x_{n+1}) + \dfrac{\beta}{h}, & \text{pour } j = n+1 \end{cases} \tag{7.23}$$

qui peut être écrit sous forme matricielle comme $A\mathbf{u} = \tilde{\mathbf{f}}$ avec

$$A = \frac{1}{h^2} \begin{bmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & -1 & \ddots & \ddots & \\ & & \ddots & \ddots & -1 \\ & & & -1 & 1 \end{bmatrix}, \qquad \tilde{\mathbf{f}} = \begin{bmatrix} f(x_1) + \frac{\alpha}{h^2} \\ f(x_2) \\ \vdots \\ f(x_n) \\ \frac{1}{2} f(x_{n+1}) + \frac{\beta}{h} \end{bmatrix}. \tag{7.24}$$

On remarque que le noeud x_{n+2} et l'inconnue correspondante u_{n+2} ont été introduit seulement pour pouvoir écrire la différence finie centrée (7.20), mais ils n'apparaissent pas dans le système final (7.23). Le noeud x_{n+2} est appelé *noeud fantôme* (*ghost node* en anglais).

Notez aussi que la seule différence entre les systèmes (7.24) et (7.19) se trouve dans la dernière composante du vecteur $\mathbf{f}$, la matrice A étant la même dans les deux cas. Néanmoins, cette petite différence est suffisante pour pouvoir récupérer une méthode d'ordre 2.