

OUTILS NUMÉRIQUES



Vincent ROSSETTO

Université Grenoble Alpes

L3 Physique / L3 Physique & Musicologie 2020-2021

Contents

Contents	ii
List of Figures	v
Liste des algorithmes	v
1 Fonctions et dérivées numériques	1
1.1 Les fonctions	1
1.1.1 Les fonctions en informatique	1
1.1.2 Les fonctions vectorisées	1
1.2 Calcul de dérivées numériques	1
1.2.1 Série de Taylor	2
1.2.2 Trois différents schémas de calcul	3
1.2.3 Ordres supérieurs	3
1.3 Travaux dirigés	5
1.3.1 Écrire une fonction	5
1.3.2 Fonctions de dérivation	6
1.3.3 Tracer un graphique	6
2 Résolution d'équations à une inconnue	7
2.1 Pseudo-code, conditions et boucles	7
2.2 Méthode de dichotomie (ou bisection)	9
2.2.1 Principe mathématique	9
2.2.2 Limites de la méthode	9
2.3 Méthode de la sécante	10
2.3.1 Principe mathématique	10
2.3.2 Limites de la méthode	11
2.4 Méthode du point fixe	11
2.4.1 Principe mathématique	11
2.4.2 Limite de la méthode	12
2.5 Méthode de Newton	13
2.5.1 Principe mathématique	13
2.5.2 Limites de la méthode	13
2.6 Travaux dirigés	14
2.6.1 Méthode de dichotomie	14
2.6.2 Méthode de la sécante	14
2.6.3 Méthode du point fixe	15
2.6.4 Méthode de Newton	15

3	Méthodes de contrôle des erreurs et accélération	16
3.1	Méthode de convergence contrôlée	16
3.2	Accélération de Romberg	17
3.3	Travaux dirigés	18
3.3.1	Convergence contrôlée	18
3.3.2	Accélération de Romberg	18
4	Calcul d'intégrales	19
4.1	Méthodes de Newton-Cotes	19
4.1.1	Méthode des rectangles	20
4.1.2	Méthode des trapèzes	20
4.1.3	Méthode de Simpson	21
4.1.4	Précision et degrés supérieurs	22
4.2	Méthodes de quadrature de Gauss	22
4.2.1	★ Méthode de Gauss-Legendre	22
4.2.2	Méthodes de Gauss-Tchebychev	24
4.2.3	Mise en œuvre pratique	24
4.3	Travaux dirigés	25
5	Systèmes d'équations linéaires	26
5.1	Généralités sur les systèmes linéaires	26
5.1.1	Inversibilité des matrices	26
5.1.2	Conditionnement des matrices	27
5.2	Système diagonal	28
5.3	Systèmes triangulaires	28
5.3.1	Matrices triangulaires inférieures et supérieures	28
5.3.2	Décomposition de Crout (décomposition LU ou LR)	29
5.4	Méthode de Gauss	31
5.5	Méthodes itératives	33
5.5.1	Méthode de Jacobi	33
5.5.2	Méthode de Gauss-Seidel	35
5.6	Travaux dirigés	36
6	Résolution d'équations différentielles	38
6.1	Équations différentielles ordinaires	38
6.2	Algorithmes d'Euler	39
6.2.1	Algorithme explicite	40
6.2.2	Algorithme implicite	40
6.3	Algorithmes de Runge-Kutta	40
6.3.1	Algorithme d'ordre 2	40
6.3.2	Algorithme d'ordre 4	41
6.4	Travaux dirigés	41
7	Méthodes de descente	42
7.1	Principe mathématique	42
7.2	Méthode du gradient (plus forte pente)	42
7.3	Résolution de systèmes linéaires	43
7.3.1	Méthode du gradient	44
7.3.2	Méthode du gradient conjugué	45
7.3.3	Méthode du double gradient conjugué	45

7.4	Travaux dirigés	46
A	Référence python	47
A.1	Les modules	47
A.2	Types basiques	48
A.2.1	Les nombres	48
A.2.2	Les tableaux	48
A.2.3	Les chaînes de caractères	48
A.3	Les fonctions numériques (numpy)	49
A.3.1	Fonctions d'un nombre	49
A.3.2	Fonctions d'un vecteur	49
A.3.3	Fonctions à plusieurs résultats	49
A.3.4	Algèbre linéaire	50
A.3.5	Calcul vectoriel	51
A.4	Les fonctions graphiques (matplotlib)	53
A.4.1	Tracer une courbe (plot)	53
A.4.2	Légendes, titre etc.	54
A.4.3	Enregistrer une figure	54
A.4.4	Exemple complet	54
A.5	Autres fonctions	55
A.5.1	Mesure du temps	55
B	Référence MATLAB/octave	56
B.1	Types basiques	56
B.1.1	Les nombres	56
B.1.2	Les vecteurs	56
B.1.3	Les chaînes de caractères	57
B.2	Les fonctions	57
B.2.1	Fonctions d'un nombre	57
B.2.2	Les fonctions d'un vecteur	57
B.2.3	Fonctions à plusieurs résultats	58
B.3	Les fonctions graphiques	58
B.3.1	Tracer une courbe (plot)	58
B.3.2	Légendes, titre, etc.	59
B.3.3	Sauvegarder une figure (print)	60
B.3.4	Exemple complet	60
B.4	Algèbre linéaire	60
B.5	Autres fonctions	63
B.5.1	Mesure du temps (tic et toc)	63

List of Figures

1.1	Schéma de principe d'une fonction	2
1.2	Les différents schémas de calcul du taux d'accroissement.	3
2.1	Illustration de la méthode de dichotomie	10
2.2	Illustration de la méthode de la sécante	11
2.3	Convergences de la méthode du point fixe	12
2.4	Divergences de la méthode du point fixe	12
2.5	Illustration de la méthode de Newton	14
4.1	Illustration des méthodes de Newton-Cotes	19
4.2	Les méthodes de Newton-Cotes d'ordres 1, 2 et 3	21
7.1	Illustration des méthodes de descente	43

Liste des algorithmes

1	Recherche de solution	9
2	Recherche de solution, avec compteur	9
3	Contrôle de convergence naïf	17
4	Calcul d'intégrale par la méthode des rectangles	20
5	Calcul d'intégrale par la méthode des rectangles (version 2)	20
6	Résolution d'un système linéaire triangulaire inférieur	29
7	Algorithme de Crout (décomposition LU / LR)	30
8	Algorithme de Jacobi	34
9	Algorithme de Gauss-Seidel	36
10	Méthode du double gradient conjugué	46

Séance 1

Fonctions et dérivées numériques

Dans ce premier chapitre, nous allons étudier quelques aspects élémentaires du calcul numérique à travers un exemple simple, le calcul numérique d'une dérivée. Ce chapitre sera également l'occasion de discuter de l'organisation des différents fichiers d'un projet numériques, les différents éléments à garder en tête.

1.1 Les fonctions

1.1.1 Les fonctions en informatique

Une fonction dans un langage informatique est un ensemble d'opération qui calcule un certain nombre d'objets (qui peut être zéro) appelés *valeurs de retour* à partir d'autres objets (potentiellement aucun) appelés les *arguments*. Au cours de ces opérations, l'état de tous les composants de l'ordinateur peut être modifié, on appelle cela *les effets de bords*.

Pour écrire une fonction, les différents langages utilisent des syntaxes différentes, mais qui précisent quels sont les arguments attendus et les valeurs de retours. En revanche les effets de bords ne sont pas explicités, il faut lire le code pour les connaître. Dans les langages modernes, un certain cloisonnement est effectué entre les différents niveaux de fonctions de sorte que les effets de bords apparaissent plus explicitement.

1.1.2 Les fonctions vectorisées

Une fonction vectorisée est une fonction à laquelle on peut donner comme argument un vecteur (un tableau) de nombres et qui agit sur tous les éléments séparément. Cette propriété est très utile pour tracer des graphiques, puisqu'elle permet si \mathbf{x} est défini comme le tableau de toutes les abscisses d'obtenir le tableau des ordonnées avec la commande

```
y=f(x)
```

L'intérêt est donc de donner une meilleure lisibilité au programme en évitant d'avoir à écrire de nombreuses boucles. Les fonctions natives (celles qui sont déjà écrites pour le langage) de `octave` et de `python` sont vectorisées, mais les fonctions utilisateurs (celle que vous allez définir) ne le sont pas *a priori*. Voir les section A.2.2 et B.1.2 pour les caractéristiques spécifiques à chaque langage.

1.2 Calcul de dérivées numériques

La plupart des calculs que l'on a besoin d'effectuer dans un projet numérique font appel à la notion de dérivée. On peut par exemple penser à la recherche d'un maximum d'une fonction, le calcul d'une vitesse ou une accélération, l'extrapolation de données... D'une façon générale, mais dans la mesure du possible, le calcul de la dérivée d'une fonction doit être effectué de façon analytique.

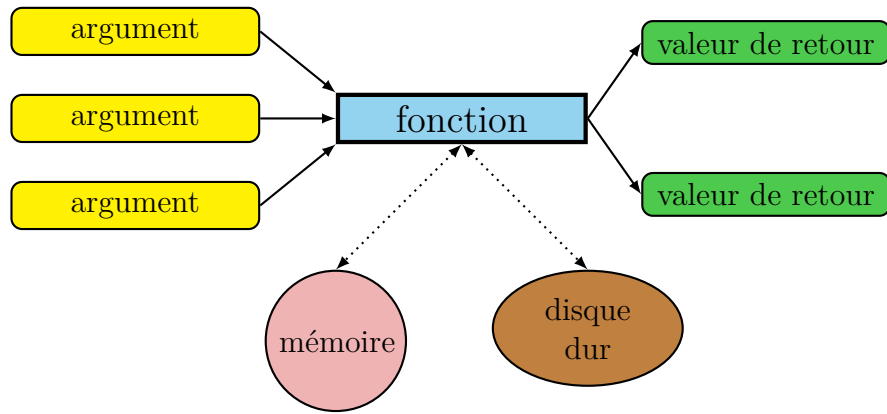


Figure 1.1: Schéma de principe d'une fonction. Les transmissions explicites sont indiqués par des flèches en trait continu, les modifications non explicitées sont représentées par des pointillés. Sur cette exemple, une fonction prend trois arguments et retourne deux objets. La fonction a aussi des effets de bords sur l'état de la mémoire qui subsistent après son exécution. Elle écrit aussi des données sur le disque dur.

Cette fonction pourrait par exemple prendre comme arguments (1) une adresse dans la mémoire, (2) une position sur le disque dur et (3) un nombre d'octets. Son action pourrait être d'échanger le nombre d'octets indiqué entre la mémoire et le disque à partir de l'adresse mémoire et la position du disque données. Les valeurs de retour pourraient être (1) le nombre d'octets qui étaient identiques sur le disque et dans la mémoire et (2) la durée de l'échange.

C'est-à-dire que si la fonction f est connue et écrite explicitement dans le code, alors f' devra également être écrite explicitement.

Le principe général d'une dérivée numérique repose sur le taux d'accroissement $\Delta_h f(x)/h$:

$$f'(x) = \lim_{h \rightarrow 0} \frac{\Delta_h f(x)}{h} \quad \text{avec } \Delta_h f(x) = f(x+h) - f(x). \quad (1.1)$$

Dans cette formule, h est un nombre qui tend vers 0. Numériquement si $h = 0$ on obtient une fraction indéterminée $0/0$, il faut donc choisir une valeur non nulle pour h , mais laquelle ?

Les valeurs de $f(x)$ et $f(x+h)$ sont potentiellement grandes, alors que h doit être à priori petit. Prenons une analogie matérielle et supposons que l'on cherche à mesurer le poids d'un paquet que l'on place dans un camion. On pèse le camion avec le paquet puis on enlève le paquet du camion et on pèse à nouveau. La différence est le poids du paquet. Si le paquet est une petite lettre très légère la balance ne permettra pas d'atteindre la précision nécessaire. Calculer $\Delta_h f(x)$ revient à mesurer une différence dans des conditions similaires (h est l'analogie du poids du paquet), dans lesquelles il est difficile d'obtenir une valeur précise. D'autre part, si on choisit h trop grand, la valeur de la dérivée donnée par le taux d'accroissement risque d'être également éloignée du résultat.

Il est important d'avoir conscience de la difficulté du choix de h qui est crucial pour le calcul numérique d'une limite, comme par exemple la dérivée.

1.2.1 Série de Taylor

Une des formules les plus utilisées en calcul numérique est la formule de la série de Taylor :

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \cdots + \frac{h^n}{n!}f^{(n)}(x) + R_n(f, x, h),$$

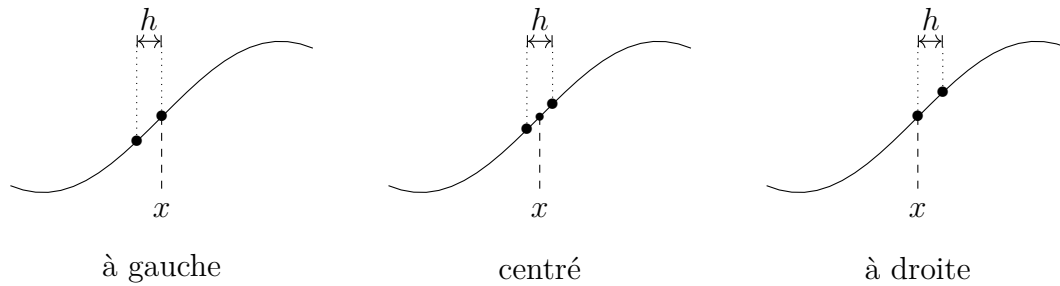


Figure 1.2: Les différents schémas de calcul du taux d'accroissement.

dans laquelle $n > 0$ est un nombre entier et $R_{n+1}(f, x, h)$ est une fonction appelé *reste d'ordre $n+1$* qui tend vers 0 strictement au moins aussi vite que h^{n+1} lorsque h tend vers 0. (on note souvent ce reste $\mathcal{O}(h^{n+1})$, que l'on lit « grand O de ... », même si l'expression de $R_n(f, x, h)$ est connue de façon précise, car il n'est pas utile de connaître cette expression). De cette expression on peut extraire $f'(x)$ en choisissant $n = 1$, ce qui donne

$$f'(x) = \frac{f(x+h) - f(x)}{h} - \frac{R_2(f, x, h)}{h} = \frac{f(x+h) - f(x)}{h} + \mathcal{O}(h).$$

Lorsque l'on prend la limite $h \rightarrow 0$, le terme $\mathcal{O}(h)$ tend vers 0, on retrouve donc bien la première expression (1.1). Cette formule nous enseigne également que le reste de la série de Taylor est

La figure suivante présente graphiquement trois schémas pour calculer la dérivée.

1.2.2 Trois différents schémas de calcul

Voici les trois schémas de dérivation les plus courants.

1. **Différence à droite** On obtient une dérivée à droite lorsque $h > 0$ dans la formule (1.1), ce qui correspond à prendre un point à droite de x car $x+h > x$.
2. **Différence à gauche** De même on obtient une dérivée à gauche lorsque $h < 0$.
3. **Différence centrée** La dérivée centrée correspond au cas où l'on prend deux points à la même distance de x , aux points d'abscisses $x-h/2$ et $x+h/2$.

Ces trois schémas sont représentés sur la figure 1.2.

Pour chacun des ces schémas, il est nécessaire de connaître la valeur de la fonction en deux points. Si la fonction est définie par une formule analytique cela demande seulement le temps de calcul de f . En revanche si la fonction n'est définie que par des valeurs dans un tableau, alors il sera nécessaire de se contenter des valeurs connues ou alors effectuer une interpolation entre les points de discrétisation. Si la valeur de f est connue en deux points $x+h$ et $x+h'$ (avec $h \neq h'$) alors la dérivée de f en x peut être estimée par

$$\frac{f(x+h) - f(x+h')}{h-h'} = f'(x) + \frac{h+h'}{2} f''(x) + \mathcal{O}((|h| + |h'|)^2).$$

1.2.3 Ordres supérieurs

Les méthodes précédentes permettent de déterminer aisément la dérivée première mais il est assez facile d'étendre les méthodes précédentes pour calculer les dérivées d'ordres supérieurs. On se limitera ici à présenter la technique pour les ordres 2 et 3 sur la méthode différence à droite. Toutefois, d'autres schémas aux ordres supérieurs se déterminent à l'aide de la même approche.

Différence à droite d'ordre deux

Pour ce cas de figure, on exprime le développement en série de Taylor en $x + h$ et $x + 2h$

$$f(x + h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \mathcal{O}(h^3), \quad (1.2)$$

$$f(x + 2h) = f(x) + 2hf'(x) + 2h^2f''(x) + \mathcal{O}(h^3). \quad (1.2')$$

Avec ces expressions, nous avons plusieurs possibilités : nous pouvons définir un schéma plus précis pour la dérivée première ou trouver un schéma pour la dérivée seconde. Pour estimer la dérivée seconde à partir de ces expressions, il nous faut annuler le terme d'ordre un (en h). Une possibilité consiste à multiplier par 2 la première ligne et à soustraire la deuxième.

$$2f(x + h) - f(x + 2h) = f(x) - h^2f''(x) + \mathcal{O}(h^3),$$

on en déduit le schéma numérique

$$f''(x) = \frac{f(x + 2h) - 2f(x + h) + f(x)}{h^2} + \mathcal{O}(h).$$

Remarque

À l'aide des développements en série (1.2) on peut également améliorer la précision sur la dérivée première. Pour cela, il faut annuler les termes d'ordre deux (en h^2) :

$$4f(x + h) - f(x + 2h) = 3f(x) + 2hf'(x) + \mathcal{O}(h^3),$$

on en déduit le schéma numérique

$$f'(x) = \frac{-f(x + 2h) + 4f(x + h) - 3f(x)}{2h} + \mathcal{O}(h^2). \quad (1.3)$$

On peut observer que ce schéma est plus précis que le schéma (1.1).

Différence à droite d'ordre trois

Utilisons la même méthode pour calculer la dérivée troisième. Les séries de Taylor en $x + h$, $x + 2h$ et $x + 3h$ sont

$$f(x + h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{6}f^{(3)}(x) + \mathcal{O}(h^4),$$

$$f(x + 2h) = f(x) + 2hf'(x) + 2h^2f''(x) + \frac{4h^3}{3}f^{(3)}(x) + \mathcal{O}(h^4), \quad (1.2.3')$$

$$f(x + 3h) = f(x) + 3hf'(x) + \frac{9h^2}{2}f''(x) + \frac{9h^3}{2}f^{(3)}(x) + \mathcal{O}(h^4). \quad (1.2.3'')$$

On s'aperçoit que l'on peut supprimer les termes d'ordre un et deux en multipliant la première équation par 3 et la seconde par -3 puis en faisant la somme :

$$3f(x + h) - 3f(x + 2h) + f(x + 3h) = f(x) + h^3f^{(3)}(x) + \mathcal{O}(h^4)$$

Ce qui nous conduit au schéma

$$f^{(3)}(x) = \frac{f(x + 3h) - 3f(x + 2h) + 3f(x + h) - f(x)}{h^3} + \mathcal{O}(h).$$

1.3 Travaux dirigés

0) **Choisir un langage de travail** Vous avez le choix (par binôme) du langage de programmation entre `python` et `octave` (MATLAB). Une liste des caractéristiques de chaque langage est présentée sur le tableau 1.1 de la présente page. Si vous maîtrisez `python`, vous n'aurez pas de difficulté à apprendre MATLAB si besoin. Si vous hésitez, choisissez `python`, qui est amené à se développer.

Table 1.1: Caractéristiques des langages de programmation.

python	MATLAB/octave
langage très répandu dans tous les domaines, y compris en recherche	langage surtout utilisé par des physiciens et des ingénieurs
langage moderne	langage « ancien » avec certaines incongruités (syntaxe, objets implicites, définitions des fonctions etc.)
langage polyvalent	principalement dédié au calcul
vous le connaissez déjà	interface MATLAB et <code>octave</code> faciles à prendre en main, avec console et outils graphiques très accessibles
libre et gratuit	licence MATLAB payante mais <code>octave</code> est libre et gratuit.
plus efficace numériquement (en utilisant <code>numba</code>)	de nombreux équipements utilisent encore MATLAB
syntaxe parfois compliquée pour effectuer des opérations simples	syntaxe des opérations mathématiques élémentaires assez intuitive
énormément d'outils disponibles (difficile de tout connaître), la gestion des bibliothèques peut être compliquée selon les systèmes	surchage des opérateurs difficile à appréhender

1.3.1 Écrire une fonction

1) **Écrire une fonction** nommée `signe` qui prend en argument un nombre réel et renvoie 1 si ce nombre est positif et -1 sinon.

2) **Écrire une fonction** que l'on nommera `periodique` qui permet de rendre périodique n'importe quelle fonction en la définissant sur un intervalle $[a, b]$. Cette fonction périodique prendra trois arguments :

- la fonction que l'on cherche à rendre périodique ;
- l'abscisse minimale de l'intervalle (a) ;
- l'abscisse maximale de l'intervalle (b),

et elle construira la fonction périodique.

On fera les tests sur la fonction `signe`. Le but est donc d'obtenir une fonction en créneaux. Pour effectuer les tests on supposera que l'on veut une fonction nommée `creneau` de période 2π définie sur l'intervalle $[-\pi, \pi]$.

3) **Écrire une fonction** `vectorise` qui construit une fonction vectorisée à partir d'une fonction qui n'accepte que des nombres réels. Utiliser `vectorise` pour construire la fonction `signevect` qui applique `signe` à tous les élément d'un tableau.

1.3.2 Fonctions de dérivation

4) **Écrire une fonction pour chacun des trois schémas de dérivation** de la page 3 permettant de construire la dérivée d'une fonction quelconque. Ces fonctions prendront deux arguments :

- la fonction à dériver f ;
- le pas de dérivation h ,

et le résultat sera une fonction. Ces trois fonctions auront comme nom : `derivee_droite`, `derivee_gauche` et `derivee_centre`.

1.3.3 Tracer un graphique

On voit apparaître dans le dernier exercice traité le besoin de tracer des courbes. Les langages `python` et `MATLAB/octave` fonctionnent de manière très différente, vous trouverez des explications dans les annexes (Annexe A.4 pour `python` et Annexe B.3 pour `MATLAB/octave`). L'exemple du tracé de la fonction sinus entre 0 et 10 y est présenté.

5) **Modifier le programme donné** pour tracer également votre fonction créneau périodique (`creneau`) définie dans la section précédente.

Nous allons maintenant vérifier le bon fonctionnement des fonctions en superposant la dérivée d'une fonction connue et les résultats numériques. Nous allons calculer la dérivée de $f : x \mapsto x \sin x$.

6) **Écrire une fonction** nommée `echelle_lin` qui prend qui prend comme arguments le point de départ, le point d'arrivée, le nombre de points désiré et qui retourne un vecteur avec une répartition linéaire des points (cette fonction pourra resservir dans les séances suivantes).

7) **Tracer l'erreur absolue** entre la dérivée exacte et les trois dérivées numériques sur l'intervalle $[0, 10]$ avec une discrétisation de l'intervalle en 100 points. Prendre un pas de dérivation $h = 0,001$. Quelle méthode vous semble la plus précise ? Pourquoi ne pas avoir choisi de tracer les erreurs relatives ?

8) **Modifier¹ la figure précédente** pour déterminer le comportement de l'erreur en $x = 8$ en faisant maintenant varier h . Dans un premier temps, on fera varier h linéairement entre 10^{-14} et 0,1 en plaçant 1000 points de discrétisation graphique. Que constate-t-on ?

Afin de pouvoir observer l'influence du pas de dérivation dans son ensemble, il est plus utile d'utiliser des échelles logarithmiques.

9) **Écrire une fonction** que l'on nommera `echelle_log` qui prend en arguments le point de départ, le point d'arrivée, le nombre de points désiré et qui retourne un vecteur avec une répartition logarithmique des points (cette fonction pourra resservir dans les séances suivantes). Exemple `echelle_log(10,1000,3)` doit retourner un tableau avec les valeurs 10, 100 et 1000.

10) **Modifier alors le graphique** de l'erreur pour l'afficher en échelle *log-log*.

On pourra faire varier la valeur de x afin de voir si le résultat en dépend. Que peut-on conclure de ces résultats ?

¹Modifier ne signifie pas que le travail précédent doit être perdu : copier-le et travailler uniquement sur la copie.

Séance 2

Résolution d'équations à une inconnue

L'objectif de cette deuxième session est de se familiariser avec quelques méthodes de résolution d'équation à une inconnue du type

$$f(x) = 0$$

ou équivalentes. Nous étudions quatre méthodes

- une méthode de dichotomie (ou bisection) ;
- une méthode de sécante ;
- une méthode de point fixe ;
- la méthode de Newton.

La méthode de la dichotomie fait évoluer un intervalle contenant la solution en rapprochant ses bornes de celle-ci. Quant à la méthode de la sécante fonctionne sur un principe similaire. La méthode du point fixe et celle de Newton cherchent une solution en faisant évoluer la valeur d'un réel jusqu'à ce qu'il soit assez proche de la solution.

Toutes ces méthodes fonctionnent sur le même principe général, en procédant par itérations successives jusqu'à ce qu'un critère de précision soit vérifié. Elles ont également toutes un nombre maximum d'itérations qui permet de s'assurer que la fonction se termine en un temps raisonnable et ne tourne pas indéfiniment.


Avant d'étudier les méthodes de résolution numérique, remarquons que chacune de ces méthodes sera inefficace si la solution exacte de l'équation est connue analytiquement, car les fonctions préprogrammées sont fortement optimisées et utilisent des algorithmes extrêmement spécifiques. Par exemple, utiliser une des méthodes présentées ici pour résoudre l'équation $x^2 - 3 = 0$ sera moins efficace que de calculer directement $x = \sqrt{3}$ à l'aide de la fonction `sqrt`.

Toutefois, il est intéressant de savoir que les algorithmes de calculs implémentés dans les langages `octave`, `python` et bien d'autres sont parfois basés sur des algorithmes de recherche de racine d'une équation. En revanche, ils contiennent de nombreuses vérifications et adaptations à tous les cas possibles, ce qui les rend également très robustes en plus d'être optimisés.

2.1 Pseudo-code, conditions et boucles

Avant de commencer ce chapitre, voici une petite introduction à un outil très utile : le *pseudo-code*. Il s'agit tout simplement d'une écriture des étapes d'un programme ou le plus souvent d'une partie technique d'un algorithme qui ne dépend pas du langage utilisé. Ceci permet de s'adresser à des programmeurs utilisant des langages différents tout en leur apportant les mêmes informations. Il reste alors aux programmeurs la tâche de traduire le pseudo-code dans leur langage de programmation, ce qui ne présente pas de difficulté, même pour des débutants. Dans la suite, nous abordons les différentes structures élémentaires.

Table 2.1: Les structures élémentaires en pseudo-code et leur traduction en python et octave.

Opération	Pseudo-code	code python	code octave
<i>Affectation</i> : crée un nouvel objet avec une valeur définie ou alors change la valeur d'un objet existant.	$x \leftarrow 1$	<code>x=1</code>	<code>x=1;</code>
<i>Condition</i> : effectue des opérations selon qu'une condition est vrai ou fausse. L'exemple ci-contre calcule i comme le signe de x .	<pre> si $x = 0$ alors $i \leftarrow 0$ sinon, si $x > 0$ alors $i \leftarrow 1$ sinon  $\text{cas } x < 0$ $i \leftarrow -1$ </pre>	<pre> if $x == 0$: $i=0$ elif $x > 0$: $i=1$ else : # $x < 0$ $i=-1$ </pre>	<pre> if $x == 0$ $i=0;$ elseif $x > 0$ $i=1;$ else % $x < 0$ $i=-1;$ endif </pre>
<i>Boucle inconditionnelle</i> : une variable prend successivement les valeurs prédéterminées et le contenu de la boucle est exécuté. L'exemple ci-contre calcule la somme des nombres de 1 à 10 en exécutant la boucle 10 fois.	<pre> $t \leftarrow 0$ pour tout $i \in \llbracket 1, 10 \rrbracket$ faire $t \leftarrow t + i$ </pre>	<pre> $t=0$ for i in <code>range</code>(1,11) : $t=t+i$ </pre>	<pre> $t=0;$ for $i=1:10$ $t=t+1;$ endfor </pre>
<i>Boucle conditionnelle</i> : la boucle est exécutée tant que la condition donnée est vraie. Dans l'exemple ci-contre, on somme dans la variable t les nombres entiers en partant de 1. À la fin de la boucle la valeur de t est supérieure à 1000.	<pre> $i \leftarrow 1$ $t \leftarrow 0$ tant que $t < 1000$ faire $t \leftarrow t + i$ $i \leftarrow i + 1$ </pre>	<pre> $i=1$ $t=0$ while $t < 1000$: $t=t+1$ $i=i+1$ </pre>	<pre> $i=1;$ $t=0;$ while $t < 1000$ $t=t+1;$ $i=i+1;$ endwhile </pre>
<i>Définir une fonction</i> : une fonction est définie par ses arguments (valeurs en entrée) et ses sorties (valeurs retournées). On représente par le symbole $\bullet \rightarrow$ l'affectation initiale des arguments de la fonction avant son exécution et par $\bullet \leftarrow$ la valeur de retour.	<pre> définir la fonction <i>moyenne</i> $\bullet \rightarrow$ nombre a $\bullet \rightarrow$ nombre b $\bullet \leftarrow (a + b)/2$ </pre>	<pre> def <code>moyenne</code>(a, b) : return $(a+b)*0.5$ </pre>	<pre> function <code>m=moyenne</code>(a,b) $m=(a+b)*0.5;$ endfunction </pre>

Toutes les méthodes décrites dans le cours fonctionnent sur le même principe général, en procédant par itérations successives jusqu'à ce qu'un critère de précision soit vérifié. Si la fonction est notée f , la précision recherchée ε et l'approximation α (qui peut être un intervalle ou seulement un nombre) alors le pseudo-code général est le suivant :

Algorithme 1 : Recherche de solution


définir la fonction *solution*

- \rightarrow la fonction f
- \rightarrow la précision ε
- \rightarrow l'approximation initiale α
- tant que** l'approximation α est moins bonne que ε **faire**
 - $\alpha \leftarrow$ nouvelle approximation obtenue à partir de f et α
- $\leftarrow \alpha$

Cependant, si la précision demandée est inaccessible (par exemple une précision relative de 10^{-30} en double précision) la boucle ne s'arrêtera jamais. Pour éviter ce problème, on ajoute un compteur de tour et on le limite à un nombre raisonnable. Le pseudo-code est ainsi légèrement modifié :

Algorithme 2 : Recherche de solution, avec compteur

définir la fonction *solution*

- \rightarrow la fonction f
- \rightarrow la précision ε
- \rightarrow l'approximation initiale α
- \rightarrow le nombre de boucles maximum M
- $n \leftarrow 0$  initialisation du compteur
- tant que** l'approximation α est moins bonne que ε **et** $n < M$ **faire**
 - $\alpha \leftarrow$ nouvelle approximation obtenue à partir de f et α
 - $n \leftarrow n + 1$
- $\leftarrow \alpha$

2.2 Méthode de dichotomie (ou bisection)

2.2.1 Principe mathématique

La méthode de dichotomie (dite aussi bisection) part de l'hypothèse que l'on connaît un intervalle $[a, b]$ dans lequel $f(x) = 0$ a une unique solution. Elle consiste à réduire de moitié l'intervalle à chaque itération en choisissant le demi-intervalle dans lequel se situe la solution. La précision demandée finit toujours par être atteinte car la largeur de l'intervalle après n itérations est toujours $(b - a)/2^n$. Voir l'illustration sur la figure 2.1.

Bien entendu, si l'équation $f(x) = 0$ n'a pas de solution ou bien plusieurs dans l'intervalle $[a, b]$ le résultat n'est pas garanti !

2.2.2 Limites de la méthode

Si l'on sait de façon sûre qu'il n'y a qu'une seule racine de l'équation $f(x) = 0$ dans l'intervalle initial, alors cet algorithme est efficace bien qu'assez lent. Il présente l'avantage de ne pas imposer

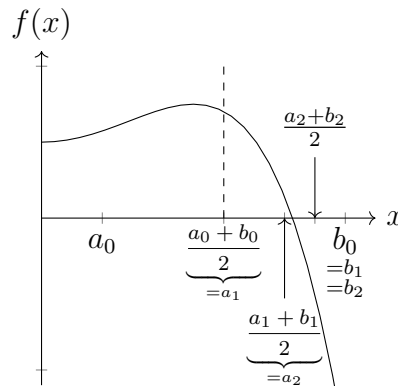


Figure 2.1: Illustration de la méthode de dichotomie. On fait l'hypothèse qu'il y a une unique solution dans $[a, b]$. On compare le signe de $f(a)$ (ou $f(b)$ qui est l'opposé) et celui de $f(\frac{a+b}{2})$ et on en déduit si la solution est dans l'intervalle $[a, \frac{a+b}{2}]$ ou dans l'intervalle $[\frac{a+b}{2}, b]$.

de conditions fortes sur f , ce qui permet de l'utiliser avec n'importe quelle fonction continue sans condition supplémentaire, au contraire de la méthode du point fixe, qui nécessite d'avoir un contrôle sur la dérivée.

En revanche, la méthode de dichotomie peut rencontrer différentes situations problématiques s'il y a plusieurs solutions dans l'intervalle initial : la méthode peut échouer ou bien trouver une des racines, mais ce comportement est difficilement prévisible car il dépend de l'intervalle initial d'une façon très peu contrôlée.

2.3 Méthode de la sécante

2.3.1 Principe mathématique

Cette méthode reprend dans le principe la méthode de Newton et la méthode de dichotomie mais en en supprimant certains inconvénients. Cette méthode démarre comme celle de la bisection avec un intervalle dans lequel il y a une racine, mais plutôt que de couper l'intervalle au milieu, on trace une droite (la *sécante*) puis on détermine où cette droite intersecte l'axe des abscisses. On itère cette opération jusqu'à obtenir un écart suffisamment faible entre les bornes de l'intervalle.

Contrairement à la méthode de dichotomie, la méthode de la sécante ne garantit pas que le point suivant soit situé dans l'intervalle de départ. Cela signifie que la solution trouvée peut être en dehors de cet intervalle comme nous le verrons sur des exemples. Cependant, si la fonction est monotone *partout*, la solution est bien contenue dans l'intervalle de départ. Les bonnes conditions pour cette méthode sont assez rares ce qui fait qu'elle est peu employée.

Notons a et b les bornes de l'intervalle et déterminons l'équation de la sécante. Elle passe par les points $(a, f(a))$ et $(b, f(b))$. Son équation est donc

$$\begin{vmatrix} x - a & b - a \\ y - f(a) & f(b) - f(a) \end{vmatrix} = 0 \quad \text{soit} \quad y = \frac{f(b) - f(a)}{b - a}(x - a) + f(a)$$

et elle coupe l'axe des abscisses en $x = a - \frac{f(a)(b-a)}{f(b)-f(a)} = \frac{af(b)-bf(a)}{f(b)-f(a)}$.

La méthode ainsi définie consiste à prendre pour valeurs initiales $x_0 = a$ et $x_1 = b$ puis calculer l'abscisse d'intersection de la sécante que l'on note x_2 . Si $x_1 - x_2$ est inférieur à la précision recherchée alors on arrête l'algorithme pour garder comme solution $\frac{x_1+x_2}{2}$, sinon on répète l'opération en calculant x_3 à partir de x_1 et x_2 et ainsi de suite.

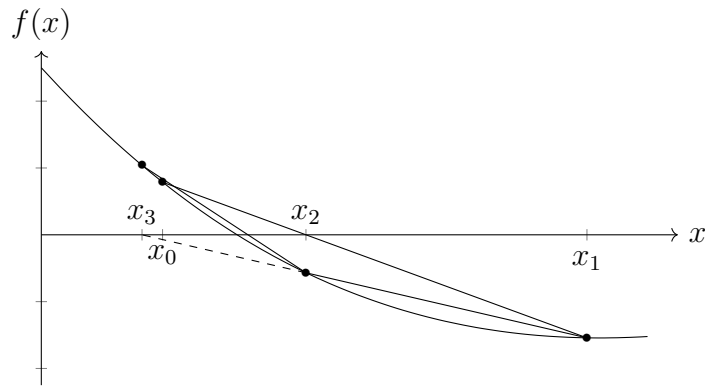


Figure 2.2: Illustration de la méthode de la sécante. Les images des points initiaux, $a = x_0$ et $b = x_1$, sont de signes opposés, ce qui assure de l'existence d'au moins une solution dans $[a, b]$. On voit sur cet exemple que l'on peut sortir de l'intervalle initial, mais que l'on peut y revenir plus près de la solution.

2.3.2 Limites de la méthode

La convergence est plus rapide que la méthode de dichotomie mais elle est plus lente que la méthode de Newton. Elle ne nécessite pas de connaître la dérivée, ni de faire des hypothèses sur son signe, comme la méthode de Newton.

La méthode de la sécante peut parfois donner un résultat correct alors que $f(a)$ et $f(b)$ sont de même signe, (dans le cas où f est monotone) ce qui amène à l'utiliser pour trouver des racines près d'une valeur singulière. Cela peut amener à l'employer à tort dans des situations où elle produira un résultat faux.

2.4 Méthode du point fixe

2.4.1 Principe mathématique

Le principe de la méthode du point fixe est de modifier la fonction f en une fonction g telle que si $g(x) = x$ alors $f(x) = 0$. La recherche du point fixe est effectuée en calculant une suite (x_n) telle que

$$x_{n+1} = g(x_n).$$

La méthode du point fixe est donc une méthode *itérative* c'est-à-dire qu'elle consiste en la répétition d'une opération, un nombre de fois inconnu au départ, chaque répétition calculant le terme suivant d'une suite. Si la suite converge, alors sa limite est un point fixe de g . L'existence de ce point fixe est assurée sous certaines conditions par le théorème de Picard. Une des conditions de convergence est que la dérivée de g au point fixe soit telle que

$$|g'(x)| < 1.$$

Les figures 2.3 et 2.4 ci-dessous montrent des exemples de situations où l'algorithme converge ou diverge et permettent de comprendre son principe sur des exemples représentant les cas les plus courants.

Pour l'utiliser, il faut donc trouver une fonction g qui permet à l'algorithme de converger. On voit que dès que f est croissante, on ne peut pas utiliser $g(x) = f(x) + x$ car cela conduit à une dérivée de g toujours supérieure à 1. Mais on peut utiliser n'importe quelle fonction g de la forme

$$g(x) = \gamma(x)f(x) + x \quad (2.1)$$

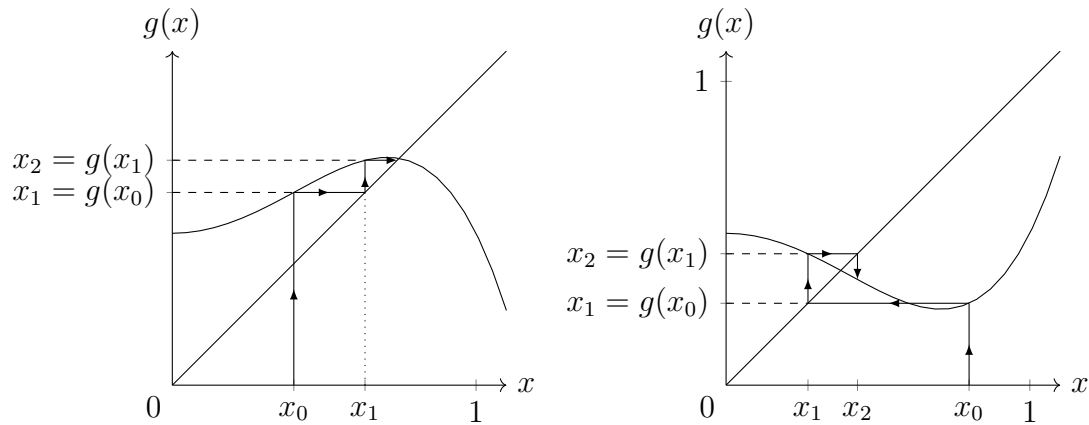


Figure 2.3: Illustrations de la convergence. La figure de gauche montre une situation où $0 < g'(x) < 1$ dans laquelle la suite (x_n) s'approche de façon monotone de la solution. La figure de droite montre une situation où $-1 < g'(x) < 0$ et où la suite converge en alternant les valeurs plus petites et les valeurs plus grandes que la solution.

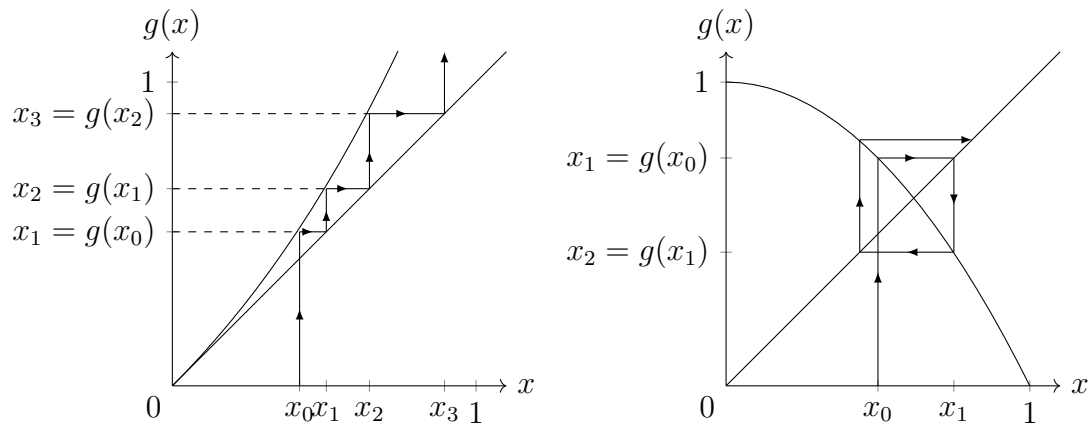


Figure 2.4: Illustrations de la divergence. La figure de gauche montre une situation où $g'(x) > 1$ dans laquelle la suite (x_n) croît et s'éloigne de la solution $x = 0$. La figure de droite montre une situation où $g'(x) < -1$ et où la suite diverge en alternant les valeurs plus petites et les valeurs plus grandes que la solution.

puisque si $g(x) = x$, cela signifie que $\gamma(x)f(x) = 0$. En choisissant une fonction γ qui ne s'annule pas, on a bien l'implication recherchée. Pour une fonction f croissante, si $0 \leq f'(x) < M$ sur un intervalle contenant la solution, alors on peut choisir $\gamma(x) = -\frac{1}{2M}$ et un point initial dans cet intervalle. Le choix de g (et donc de γ) peut se révéler difficile.

D'autres choix sont possibles pour g , qui seront ou non adaptés au problème. Par exemple

$$g(x) = x \exp[-af(x)].$$

Nous verrons avec la méthode de Newton un choix très judicieux de fonction g qui permet d'accélérer la convergence.

2.4.2 Limite de la méthode

Comme on l'a déjà mentionné, la méthode du point fixe nécessite parfois des ajustements spécifiques à l'équation que l'on cherche à résoudre. De plus il faut noter que si l'algorithme

trouve une solution, celle-ci n'est peut-être pas la seule. Le choix de la fonction g se révèle là encore tout à fait crucial, car une fonction g peut introduire des solutions artificielles (par exemple si $g(x) = \gamma(x)f(x) + x$, la solution trouvée pourra être un zéro de γ).

La solution trouvée dépend de la condition initiale x_0 , ce qui rend possible de trouver plusieurs solutions en essayant plusieurs valeurs de x_0 différentes, mais sans garantie.

2.5 Méthode de Newton

2.5.1 Principe mathématique

La méthode de Newton est une méthode très efficace de calcul de solution, surtout utilisée pour obtenir une grande précision. Il s'agit d'une des méthodes les plus utilisées, parfois avec un nombre très limité d'itérations, notamment pour affiner un résultat approché obtenu par d'autres moyens.

Son principe consiste à considérer le point d'intersection de la tangente à la courbe de f au point x_n avec l'axe des abscisses. Ce point alors défini comme x_{n+1} et on a

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}. \quad (2.2)$$

L'intérêt de la méthode de Newton est sa rapidité de convergence. Pour le voir, appliquons f à l'équation (2.2) en supposant que x_n est proche de la solution x_∞ , c'est-à-dire que $f(x_n) \ll 1$ et prenons le développement limité au deuxième ordre du membre de droite, en considérant $h = f(x_n)/f'(x_n)$ comme petit.

$$\begin{aligned} f(x_{n+1}) &= f(x_n - h) \\ &= f(x_n) - hf'(x_n) + \frac{h^2}{2}f''(x_n) + o(h^2) \\ &\approx f(x_n) - \frac{f(x_n)}{f'(x_n)}f'(x_n) + \frac{1}{2}\left(\frac{f(x_n)}{f'(x_n)}\right)^2 f''(x_n). \end{aligned}$$

Les deux premiers termes se compensent. Si $a = f'(x_\infty) \neq 0$ et $b = f''(x_\infty) \neq 0$ alors ce développement limité se réécrit $f(x_{n+1}) \approx \frac{b}{2a^2}f(x_n)^2$. Or au voisinage de la solution $f(x) \approx a(x - x_\infty)$, on a donc

$$|x_{n+1} - x_\infty| \propto |x_n - x_\infty|^2,$$

c'est-à-dire que si x_n est proche de la solution à 10^{-3} près, x_{n+1} sera proche à 10^{-6} près ! La convergence est donc rapide. C'est pour cette raison que la méthode de Newton est si utilisée. Ce type de convergence est dit *quadratique*.

Pour programmer la méthode de Newton, il peut être utile de remarquer qu'elle n'est rien d'autre que d'une version particulière de la méthode du point fixe, avec $\gamma(x) = -\frac{1}{f'(x)}$ dans les notations de l'équation (2.1). On peut donc repartir du programme de la méthode du point fixe et adapter le code.

2.5.2 Limites de la méthode

Cette méthode nécessite de connaître la dérivée f' de la fonction f . Cette dérivée peut se calculer numériquement si elle n'est pas connue, mais cela peut introduire une source d'erreur supplémentaire. La méthode de Newton ne permet pas de contrôler vers quelle racine l'algorithme converge, car si la dérivée $f'(x_n)$ est proche de zéro, le point x_{n+1} peut se trouver loin de x_n et s'approcher une autre racine. On observe également sur la figure 2.5 que si le signe de la dérivée change, alors le comportement de la suite des points obtenus n'est plus prévisible. L'hypothèse d'une dérivée de signe constant est plus forte celle de la racine unique dans l'intervalle de recherche que l'on fait pour la dichotomie. La mise en œuvre de cette méthode nécessite donc de prendre des précautions.

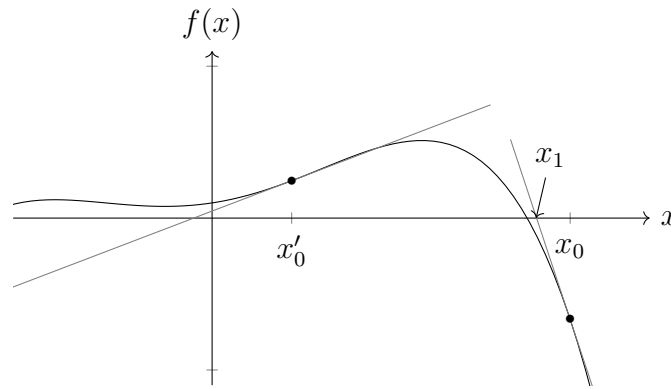


Figure 2.5: Illustration de la méthode de Newton. Le point de départ x_0 est déjà proche de la racine de f , et on observe que x_1 s'en rapproche encore. D'un autre côté, le point x'_0 éloigne la suite de la racine positive. Ceci illustre l'importance que la dérivée f' ne change pas de signe.

2.6 Travaux dirigés

2.6.1 Méthode de dichotomie

1) **Programmer une fonction** *dichotomie* prenant comme arguments une fonction $f : \mathbf{R} \rightarrow \mathbf{R}$, les bornes de l'intervalle initial $[a, b]$, et le critère d'arrêt ε et qui renvoie la racine trouvée et le nombre d'itérations effectuées. Quel est le nombre d'itérations nécessaires en fonction de a , b et ε ?

2) **Résoudre les équations suivantes** avec *dichotomie*

$$\begin{array}{ll} x^2 - x = 0 & \text{sur } [-1, 2] \\ e^x - \sin x = 0 & \text{sur } [-5, 0] \end{array}$$

en demandant une précision de 10^{-3} puis 10^{-6} .

3) **Tracer le nombre d'itérations et la valeur de la solution** obtenue pour ε variant entre 10^{-14} et 10^{-1} . On utilisera un diagramme *log-log*.

2.6.2 Méthode de la sécante

4) **Programmer une fonction** *secante* prenant comme arguments une fonction $f : \mathbf{R} \rightarrow \mathbf{R}$, les points initiaux a et b , le critère d'arrêt ε et le nombre maximum d'itérations et qui renvoie la racine trouvée et le nombre d'itérations effectuées.

5) **Résoudre les équations suivantes** avec *secante*

$$\begin{array}{ll} x^2 - x = 0 & \text{avec } a = 0,5 \text{ et } b = 10, \\ & a = -9 \text{ et } b = 10, \\ & a = 5 \text{ et } b = 10 \\ e^x - \sin x = 0 & \text{avec } a = -5 \text{ et } b = 0 \end{array}$$

en demandant une précision de 10^{-6} .

6) **Tracer le nombre d'itérations et la valeur de la solution** obtenue pour ε variant entre 10^{-14} et 10^{-1} . On utilisera un diagramme *log-log*.

2.6.3 Méthode du point fixe

7) **Donner 2 fonctions** g permettant de résoudre l'équation

$$x^3 + 3x + 2 = 0 \quad (2.3)$$

avec la méthode du point fixe.

8) **Programmer une fonction** `point_fixe` prenant comme arguments une fonction $g : \mathbf{R} \rightarrow \mathbf{R}$, une valeur initiale x_0 , le critère d'arrêt ε et le nombre maximum d'itérations N et dont le résultat est le point fixe x trouvé et le nombre d'itérations effectuées n .

Pour avoir plusieurs valeurs de retours, on renvoie un tableau. Voir les sections A.3.3 et B.2.3 pour voir la façon de faire cela en `python` ou `octave` respectivement.

9) **Résoudre l'équation** $x^2 - x = 0$ avec la fonction `point_fixe` avec $g(x) = f(x) + x$ en prenant $\varepsilon = 10^{-4}$ et $x_0 = 0,9$ puis $x_0 = 5$. Expliquer les résultats.

10) **Trouver une autre fonction** g pour résoudre $x^2 - x = 0$ avec les mêmes paramètres.

11) **Résoudre l'équation** (2.3) avec la fonction `point_fixe` en prenant $\varepsilon = 10^{-4}$ en choisissant g judicieusement. Comparer les résultats obtenus avec différentes fonctions g de la question 7.

2.6.4 Méthode de Newton

12) **Programmer une fonction** `newton` prenant comme arguments une fonction $f : \mathbf{R} \rightarrow \mathbf{R}$, une valeur initiale x_0 , le critère d'arrêt ε et le nombre maximum d'itérations N et qui renvoie la racine trouvée par la méthode de Newton et le nombre d'itérations effectuées n .

13) **Résoudre les équations suivantes** avec `newton`

$$\begin{array}{ll} x^2 - x = 0 & \text{avec } x_0 = -5 \text{ et } x_0 = 10 \\ \tan x - x = 0 & \text{sur } \left[\pi, \frac{3\pi}{2} \right] \text{ avec } x_0 = 3,85 \quad 3,86 \quad 3,9 \quad 4,2 \quad 4,3 \text{ et } 4,5 \end{array}$$

en demandant une précision de 10^{-7} . Donner vos conclusions sur la méthode de Newton au regard de ces résultats.

14) **Comparer les différentes méthodes.**

Séance 3

Méthodes de contrôle des erreurs et accélération

Dans les séances précédentes, nous avons abordé deux types de calcul qui ont des rapports différents avec la précision de calcul. Pour le calcul des dérivées, nous avons écrit des fonctions qui dépendent d'un petit paramètre h dont dépend la précision du résultat final, alors que pour la résolution d'équations, nous avons écrit des fonctions qui calculent pour atteindre une précision ε requise. De manière générale, il est préférable de demander la précision souhaitée directement plutôt que de chercher le paramètre h qui la fournit.

Dans cette séance, nous allons exiger une précision ε plutôt que de donner un petit élément h pour le calcul des dérivées. Plus précisément, nous allons étudier deux méthodes qui cherchent la valeur de h correspondant à la précision demandée ε . La première méthode s'inspire des méthodes de résolution d'équations du chapitre 2, l'autre est plus complexe, mais plus efficace.

Dans ces deux méthodes, nous avons une fonction qui associe à h un résultat $g(h)$. Par exemple g peut être la dérivée à droite d'une fonction f calculée avec la méthode de dérivée à droite du chapitre 1 en un point x fixé :

$$g(h) = \frac{f(x+h) - f(x)}{h}. \quad (3.1)$$

Bien entendu, nous avons déjà résolu pour ce cas simple la question de la précision, mais il sera intéressant de comparer notre solution à celle que fournit la méthode de convergence choisie. De plus les méthodes que nous allons étudier s'appliquent à n'importe quel algorithme dépendant d'un petit paramètre, comme le calcul d'intégrales (chapitre suivant), les recherches de d'extrémums ou les résolutions d'équations différentielles.

3.1 Méthode de convergence contrôlée

La première méthode que nous allons étudier est très simple. Elle consiste à calculer le résultat obtenu avec une valeur de $h = h_0$ assez grande, puis de la diviser par deux jusqu'à ce que l'écart avec le résultat précédent soit inférieur à ε . Ce principe ressemble beaucoup à la méthode de dichotomie vu en 2.2. Voici le pseudo-code de cette méthode :

Algorithme 3 : Contrôle de convergence naïf

définir la fonction *converge*

- \rightarrow une fonction $g : h \mapsto$ résultat d'un calcul avec petit élément h
- \rightarrow la valeur initiale de $h : h_0$
- \rightarrow une précision ε
- \rightarrow un nombre d'itérations maximum M

$x \leftarrow g(h_0)$

$h \leftarrow h_0/2$

$x' \leftarrow g(h)$

$n \leftarrow 0$  initialisation du compteur

tant que $|x - x'| > \varepsilon$ **et** $n < M$ **faire**

$x \leftarrow x'$

$h \leftarrow h/2$

$x' \leftarrow g(h)$

$n \leftarrow n + 1$

• $\leftarrow x, h$ **et** n

3.2 Accélération de Romberg

La deuxième méthode que nous allons aborder est celle de l'accélérateur de Romberg. Son principe consiste à considérer le résultat du calcul avec une valeur de h donnée comme le développement limité du résultat cherché :

$$g(h) = g(0) + hg'(0) + \frac{h^2}{2}g''(0) + \dots \quad (3.2)$$

Dans cette expression, le résultat recherché est $g(0)$. On note dans la suite $G = g(0)$ et $g'(0) = g_1$, $g''(0) = g_2$, $g^{(n)}(0) = g_n$. Le principe de l'accélérateur consiste à éliminer successivement les termes du développement. Prenons par exemple la première étape et calculons $g(h)$ et $g(h/2)$:

$$g(h) = G + hg_1 + \frac{h^2}{2}g_2 + \dots \quad (3.3)$$

$$g\left(\frac{h}{2}\right) = G + \frac{h}{2}g_1 + \frac{h^2}{8}g_2 + \dots \quad (3.4)$$

on peut alors calculer $2g(h/2) - g(h)$:

$$2g\left(\frac{h}{2}\right) - g(h) = G - \frac{h^2}{4}g_2 + \dots \quad (3.5)$$

Nous avons éliminé le terme d'ordre un ! Ainsi en utilisant $g(h)$ et $g(h/2)$, nous avons gagné un ordre de précision. On peut remarquer que si g est la fonction dérivée à droite (3.1), le schéma obtenu par cette première étape d'accélération est le même que celui obtenu en (1.3).

L'accélération de Romberg consiste à éliminer les termes suivants du développement limité. Pour cela, appliquons l'expression (3.5) en remplaçant h par $h/2$. Nous obtenons

$$2g\left(\frac{h}{4}\right) - g\left(\frac{h}{2}\right) = G - \frac{h^2}{16}g_2 + \dots \quad (3.6)$$

On peut alors réemployer la même technique pour éliminer le terme en h^2 . Notons $R_{n,k}$ le terme obtenu à partir de $g(h/2^n)$ dans lequel on a éliminé k termes du développement. Alors la formule

(3.3) est $R_{0,0}$, la formule (3.4) est $R_{1,0}$ et (3.5) est $R_{1,1}$ et enfin la formule (3.6) est $R_{2,1}$. On obtient le $R_{2,2}$ en combinant $R_{1,1}$ et $R_{2,1}$ de la façon suivante

$$R_{2,2} = \frac{4R_{2,1} - R_{1,1}}{3} = \frac{4G - 4\frac{h^2}{16}g_2 - G + \frac{h^2}{4}g_2}{3} = G + \mathcal{O}(h^3)$$

Le terme général s'obtient donc de la façon suivante

$$R_{n,0} = g\left(\frac{h}{2^n}\right)$$

$$R_{n,k} = \frac{1}{2^k - 1} (2^k R_{n,k-1} - R_{n-1,k-1}) \quad \text{pour } 1 \leq k \leq n$$

La valeur donnée par l'accélérateur de Romberg d'ordre n est $R_{n,n}$. C'est donc cette valeur que nous utiliserons comme élément de comparaison pour approcher la précision ε demandée.

Note —

L'accélérateur de Romberg décrit ici est le plus fréquemment utilisé, il existe néanmoins d'autres versions qui sont déterminées par des développements limités différents de celui de l'équation (3.2). Par exemple si les premier et deuxième termes ne sont pas d'ordre 1 et 2 :

$$g(h) = G + g_1 h^p + \dots$$

alors l'accélération est encore plus forte en utilisant

$$R_{1,1} = \frac{2^p R_{1,0} - R_{0,0}}{2^p - 1}.$$

3.3 Travaux dirigés

3.3.1 Convergence contrôlée

- 1) **Écrire une fonction** `converge` qui implémente l'algorithme de convergence contrôlée.
- 2) **Comparer les résultats** avec la fonction de dérivée optimisée du chapitre 1.

3.3.2 Accélération de Romberg

En pratique on calcule les termes du tableau suivant, ligne par ligne de gauche à droite. On note qu'il faut garder les valeurs à la rangée $n - 1$ pour calculer la rangée n .

$$\begin{array}{lll} R_{0,0} = g(h) & & \\ R_{1,0} = g\left(\frac{h}{2}\right) & R_{1,1} = 2R_{1,0} - R_{0,0} & \\ R_{2,0} = g\left(\frac{h}{4}\right) & R_{2,1} = 2R_{2,0} - R_{1,0} & R_{2,2} = \frac{1}{3}(4R_{2,1} - R_{1,1}) \\ \vdots & \vdots & \dots \\ R_{n,0} = g\left(\frac{h}{2^n}\right) & R_{n,1} = 2R_{n,0} - R_{n-1,0} & R_{n,2} = \frac{1}{3}(4R_{n,1} - R_{n-1,1}) \quad \dots \end{array}$$

- 3) **Écrire une fonction** `romberg` prenant les mêmes arguments que la fonction `converge`.
- 4) **Comparer les valeurs** de h finales obtenues par `converge` et `romberg`. Où se situent ces valeurs par rapport à la valeur optimale de h pour la dérivée à droite ?

Séance 4

Calcul d'intégrales

Une des grandes utilisations de l'ordinateur en physique réside dans le calcul d'intégrales, par exemple pour calculer des moyennes. Comme dans le calcul des dérivées — et tous les calculs différentiels en général — les méthodes les plus simples consistent à découper l'intervalle d'intégration en intervalles très petits et calculer une somme. Nous verrons cependant qu'il existe des méthodes beaucoup plus efficaces.

4.1 Méthodes de Newton-Cotes

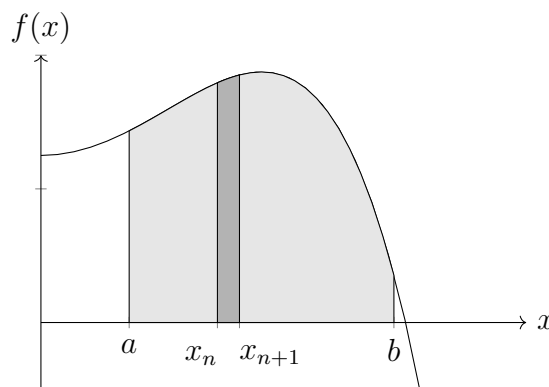


Figure 4.1: Illustration du calcul d'une intégrale avec une méthode de Newton-Cotes : l'intégrale du domaine foncé est approchée par le calcul d'une interpolation de f . L'intégrale sur $[a, b]$ est alors approchée par la somme des interpolations sur chaque petit domaine.

Certaines intégrales n'ont pas besoin d'être calculées numériquement, leur résultat est immédiat, par exemple $\int_0^1 x dx = \frac{1}{2}$. D'autres sont calculables analytiquement, mais demandent des techniques (voire des astuces) qui nécessitent du temps, par exemple $\int_0^\pi \ln(\sin x) dx = -\pi \ln 2$. Enfin il existe des intégrales que l'on ne peut pas calculer par aucune méthode autre que le calcul numérique, par exemple $\int_0^1 e^{-x^2} dx \approx 0,746824$.

La stratégie la plus utilisée pour calculer numériquement l'intégrale d'une fonction f consiste à découper l'intervalle d'intégration en intervalles plus petits de largeur h et à calculer l'intégrale de f sur tous ces petits intervalles approchant la fonction par un polynôme de petit degré. L'intégrale recherchée est alors la somme des valeurs approchées sur tous les intervalles. Ces techniques portent le nom de méthodes de Newton-Cotes.

4.1.1 Méthode des rectangles

Pour commencer, prenons un intervalle $[a, b]$ et découpons-le en intervalles de largeur $h = \frac{b-a}{1000}$. Pour approcher une fonction f par une constante (polynôme de degré 0) sur un intervalle $[a + nh, a + nh + h]$, nous pouvons prendre comme valeur $f(a + nh + \frac{h}{2})$ et ainsi la contribution de l'intervalle $[a + nh, a + (n + 1)h]$ à l'intégrale est $f(a + \frac{h}{2}) \times h$. Cela nous donne un premier algorithme d'intégration appelé *méthode des rectangles* :

Algorithme 4 : Calcul d'intégrale par la méthode des rectangles

définir la fonction *rectangles*

```
• → une fonction  $f$  à intégrer
• → un intervalle d'intégration  $[a, b]$ 
• → un pas d'intégration  $h$  (petit)
  total ← 0
   $x \leftarrow a + h/2$ 
  tant que  $x < b$  faire
    | total ← total +  $hf(x)$ 
    |  $x \leftarrow x + h$ 
  ← total
```

Note

Attention, dans cet algorithme, on ne vérifie pas que $b - a$ est bien un multiple entier de h . C'est pourquoi on préfère parfois la version légèrement modifiée suivante :

Algorithme 5 : Calcul d'intégrale par la méthode des rectangles (version 2)

définir la fonction *rectangles*

```
• → une fonction  $f$  à intégrer
• → un intervalle d'intégration  $[a, b]$ 
• → le nombre de pas d'intégration  $N$  (grand)
  total ← 0
   $h \leftarrow \frac{b - a}{N}$ 
  pour tout  $n \in \llbracket 0, N - 1 \rrbracket$  faire
    | total ← total +  $hf(a + nh + h/2)$ 
  ← total
```

La valeur approchée de l'intégrale est alors, en notant $x'_n = a + \frac{h}{2} + nh$,

$$\int_a^b f(x)dx \approx \frac{b-a}{N} \sum_{n=0}^{N-1} f(x'_n) \quad (4.1)$$

4.1.2 Méthode des trapèzes

Avec le même principe que la méthode des rectangles, mais avec une approximation linéaire (ordre 1), on obtient la méthode des trapèzes. L'approximation linéaire de f sur le petit intervalle

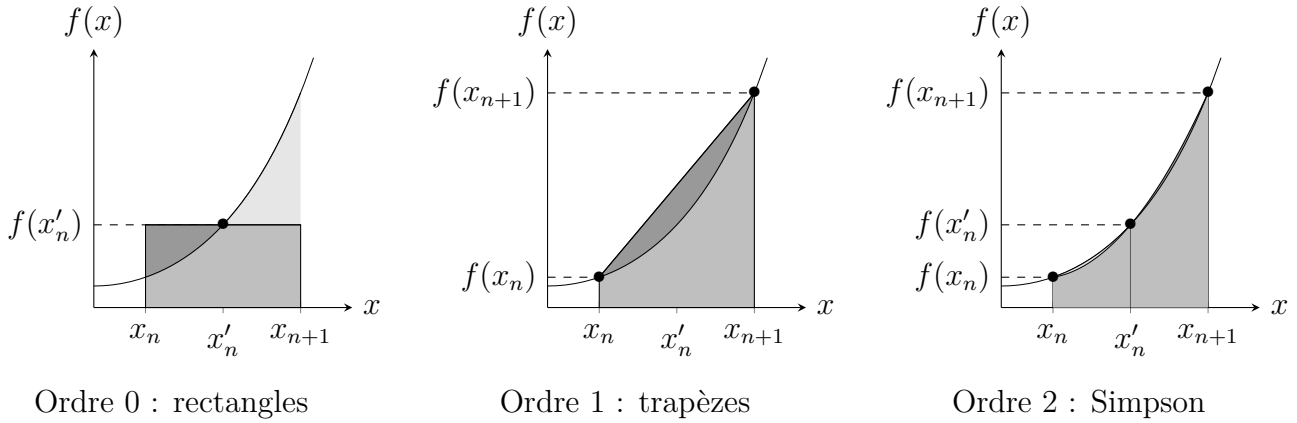


Figure 4.2: Les trois premiers ordre d'intégration par méthodes de Newton-Cotes. Les déficits d'aire sont représentés par les zones colorées en gris clair, les excès par les zones gris foncé. L'aire exacte s'obtient en ajoutant l'aire colorée en gris moyen à l'aire en gris clair.

$[a+nh, a+nh+h]$ consiste à prendre la droite qui passe par les deux points extrêmes de l'intervalle. Introduisons la notation

$$x_n = a + nh.$$

L'approximation linéaire de f sur l'intervalle $[a+nh, a+nh+h] = [x_n, x_{n+1}]$ est la droite d'équation

$$y = \frac{f(x_{n+1}) - f(x_n)}{x_{n+1} - x_n}(x - x_n) + f(x_n).$$

L'aire du trapèze est alors

$$\int_{x_n}^{x_{n+1}} y(x)dx = (x_{n+1} - x_n) \frac{f(x_n) + f(x_{n+1})}{2} = h \frac{f(x_n) + f(x_{n+1})}{2},$$

et l'approximation de l'intégrale est donc

$$\int_a^b f(x)dx \approx \frac{f(a) + f(b)}{2N} + \frac{b-a}{N} \sum_{n=1}^{N-1} f(x_n) \quad (4.2)$$

car en faisant la somme sur tous les intervalles, les contributions des points x_n comptent pour deux intervalles, sauf pour le point $x_0 = a$ et le point $x_N = b$. On peut remarquer la proximité très grande avec la formule des rectangles (4.1).

4.1.3 Méthode de Simpson

La méthode de Simpson est basée sur le même principe que les précédentes méthodes, en utilisant l'interpolation quadratique de f sur $[x_n, x_{n+1}]$ pour laquelle on utilise aussi le point intermédiaire $x'_n = a + \frac{h}{2} + nh$ qui est au milieu de $[x_n, x_{n+1}]$. Ce polynôme s'écrit

$$p(x) = f(x_n) \frac{x'_n - x}{h/2} \frac{x_{n+1} - x}{h} + f(x'_n) \frac{x - x_n}{h/2} \frac{x_{n+1} - x}{h/2} + f(x_{n+1}) \frac{x - x_n}{h} \frac{x - x'_n}{h/2}$$

Nous avons utilisé une technique appelée *interpolation de Lagrange* pour l'obtenir. On peut vérifier que p prend la même valeur que f aux points x_n , x'_n et x_{n+1} et que p est un polynôme de degré 2. C'est donc l'unique polynôme interpolateur de f passant par ces points. L'intégrale de p vaut

$$\int_{x_n}^{x_{n+1}} p(x)dx = \frac{h}{6} [f(x_n) + 4f(x'_n) + f(x_{n+1})]$$

ce qui conduit à la formule de Simpson

$$\int_a^b f(x)dx \approx \frac{f(a) + f(b)}{6N} + \frac{b-a}{3N} \sum_{n=1}^{N-1} (f(x_n) + 2f(x'_n)).$$

On remarque alors que si on note R la formule des rectangles (4.1), T la formule des trapèzes (4.2) et S la formule de Simpson ci-dessus, alors $S = \frac{1}{3}(2R + T)$.

4.1.4 Précision et degrés supérieurs

Étudions la précision des méthodes présentées sur un intervalle $[x_n, x_{n+1}]$. Le terme d'erreur d'une méthode de Newton-Cotes d'ordre n pour une fonction f quelconque est de la forme

$$\Delta = rh^{p+1} \sup_{[x_n, x_{n+1}]} |f^{(p)}|, \quad (4.3)$$

avec $r \in \mathbb{Q}^+$ et $p = 2 + 2\lfloor \frac{n}{2} \rfloor$. Voici les *majorants* de l'erreur pour les méthodes que nous avons déjà rencontrées :

$$\Delta_R = \frac{h^3}{24} \sup_{[x_n, x_{n+1}]} |f''|, \quad \Delta_T = \frac{h^3}{12} \sup_{[x_n, x_{n+1}]} |f''|, \quad \Delta_S = \frac{h^5}{2880} \sup_{[x_n, x_{n+1}]} |f^{(4)}|.$$

Pour une même fonction, la méthode d'ordre 0 est donc meilleure que la méthode d'ordre 1, mais la méthode d'ordre 2 les combine exactement comme il faut pour compenser ces erreurs.

Faut-il pour autant augmenter l'ordre de l'interpolation de Lagrange pour augmenter la précision ? Malheureusement ce n'est pas si simple. Lorsque l'on augmente l'ordre de l'interpolation se produit ce que l'on appelle le *phénomène de Runge* : la fonction polynôme interpolatrice oscille énormément lorsque le nombre de points augmente (ses dérivées deviennent de plus en plus grandes). Cela vient en partie du fait que les points d'interpolation sont régulièrement espacés. De plus les dérivées successives de f ($f^{(6)}$, $f^{(8)}$...) augmentent aussi très rapidement, et d'après la formule des erreurs (4.3), ce n'est pas une bonne chose. C'est pourquoi on adopte la stratégie mixte consistant à garder un degré petit et diviser l'intervalle d'intégration en sous-intervalles. Les méthodes de quadrature de Gauss utilisent des points avec une meilleure répartition, ce qui remédie aux défauts des méthodes de Newton-Cotes.

4.2 Méthodes de quadrature de Gauss

Les méthodes de quadrature de Gauss utilisent un principe radicalement différent des méthodes vues précédemment puisqu'elle effectue une somme pondérée des valeurs de f à des points et avec des poids judicieusement choisis. L'idée de départ consiste à calculer le résultat avec des fonctions particulières. La méthode de Gauss-Legendre utilise comme référence les polynômes de Legendre, alors que la méthode de Gauss-Tchebychev utilise les polynômes de Tchebychev ! (Il en existe d'autres : Gauss-Laguerre, Gauss-Hermite...)

4.2.1 ★ Méthode de Gauss-Legendre

La méthode de Gauss-Legendre¹ à l'ordre n est conçue pour donner *un résultat exact pour tous les polynômes de degré égal au plus à $2n - 1$* . Comment-cela est-il possible ?

¹Cette section décrit en détail le calcul des points et des coefficients, elle ne fait pas partie du programme à connaître.

Position du problème

Il suffit de remarquer qu'un polynôme de degré au plus $2n - 1$ s'écrit

$$P(X) = a_{2n-1}X^{2n-1} + \dots + a_1X + a_0,$$

c'est-à-dire qu'il est déterminé par $2n$ coefficients $a_0, a_1, \dots, a_{2n-1}$. L'intégrale de P sur $[a, b]$ pour un tel polynôme dépend donc de $2n$ coefficients car elle vaut

$$\int_a^b P(x)dx = a_{2n-1} \frac{b^{2n} - a^{2n}}{2n} + \dots + a_1 \frac{b^2 - a^2}{2} + a_0(b - a). \quad (4.4)$$

L'estimateur de Gauss-Legendre pour l'intégrale du fonction f est défini par

$$\int_a^b f(x)dx \approx J_{[a,b]}(f) = \sum_{i=1}^n w_i f(x_i). \quad (4.5)$$

Avec n points $x_i \in [a, b]$ ayant chacun un poids $w_i > 0$ nous avons à notre disposition $2n$ variables! Pour les polynômes de degré au plus $2n - 1$ il s'exprime donc

$$J_{[a,b]}(P) = \sum_{i=1}^n w_i (a_{2n-1}x_i^{2n-1} + \dots + a_1x_i + a_0). \quad (4.6)$$

Il «suffit» alors que l'égalité entre (4.4) et (4.6) soit valable quelques soient les coefficients $a_0, a_1, \dots, a_{2n-1}$ et le tour est joué. En identifiant entre ces deux équations on trouve

$$\sum_{i=1}^n w_i x_i^{k-1} = \frac{b^k - a^k}{k} \quad (1 \leq k \leq 2n) \quad (4.7)$$

soit un système de $2n$ équations à $2n$ inconnues.

Résolution du système

Nous allons simplement montrer que la solution du système s'obtient à partir des polynômes de Legendre. Pour cela nous allons définir un *produit scalaire* de fonctions:

$$\langle f, g \rangle = \int_{-1}^1 f(x)g(x)dx.$$

Rappelons alors quelques propriétés des polynômes de Legendre :

1. les polynômes de Legendre sont orthogonaux pour le produit scalaire décrit plus haut. Plus précisément

$$\int_{-1}^1 P_m(x)P_n(x)dx = \begin{cases} \frac{2}{2n+1} & \text{si } m = n, \\ 0 & \text{sinon;} \end{cases}$$

2. le polynôme de Legendre d'ordre n , P_n , est de degré n ;
3. P_n possède n racines distinctes sur $[-1, 1]$;
4. $P_{n+1}(x) = \frac{2n+1}{n+1}xP_n(x) - \frac{n}{n+1}P_{n-1}(x)$.

Calculons alors $\langle Q(x), P_n(x) \rangle$ avec Q un polynôme de degré k tel que $0 \leq k < n$. La première propriété nous indique de ce produit scalaire vaut zéro. On remarque alors que l'on a $\langle Q(x), P_n(x) \rangle = \langle 1, Q(x)P_n(x) \rangle$ et que le polynôme $Q(x)P_n(x)$ est de degré inférieur ou égal à $2n - 1$. On peut donc utiliser la formule (4.5)

$$\int_{-1}^1 Q(x)P_n(x)dx = \sum_{i=1}^n w_i Q(x_i)P_n(x_i) = 0.$$

Ceci étant valable pour tous les polynômes Q de degré strictement inférieur à n , la dernière égalité nous dit que $P(x_i) = 0$ pour tous les i de 1 à n . On en déduit que les x_i sont des racines de P_n et comme elles sont au nombre de n , les x_i sont exactement les racines de P_n . Le calcul des poids w_i consiste alors à choisir un polynôme interpolateur de Lagrange qui prend la valeur 1 en x_j et 0 pour les autres x_i ($i \neq j$) et à calculer l'intégrale. Ce calcul est assez long, fait appel à la quatrième propriété et demande quelques astuces de calcul, mais il conduit à

$$w_i = \frac{2}{n^2} \frac{1 - x_i^2}{P_{n-1}(x_i)^2}.$$

Chaque racine du polynôme P_{n+1} se situe soit entre deux racines du polynôme P_n soit entre la plus petite racine de P_n et -1 soit entre la plus grande racine de P_n et 1 . Il est alors possible de calculer numériquement autant de racines que l'on souhaite en utilisant une méthode de résolution d'équation.

4.2.2 Méthodes de Gauss-Tchebychev

Les méthodes de Gauss-Tchebychev fonctionne de la même manière avec les polynômes de Tchebychev T_n (tels que $T_n(\cos \theta) = \cos(n\theta)$) avec comme résultats à l'ordre n

$$x_i = \cos\left(\frac{2i-1}{2n}\pi\right), \quad w_i = \frac{\pi}{n}, \quad \int_{-1}^1 f(x) \frac{dx}{\sqrt{1-x^2}} \approx \sum_{i=1}^n w_i f(x_i).$$

$$x_i = \cos\left(\frac{i}{n+1}\pi\right), \quad w_i = \pi \frac{1-x_i^2}{n+1}, \quad \int_{-1}^1 f(x) \sqrt{1-x^2} dx \approx \sum_{i=1}^n w_i f(x_i).$$

Ces méthodes présentent l'avantage de posséder une formule simple pour tous les coefficients.

4.2.3 Mise en œuvre pratique

Plutôt que de calculer les coefficients à chaque nouvelle intégrale, on peut se ramener à l'intervalle $[-1, 1]$ ce qui simplifie l'opération et permet de calculer les x_i et les w_i une seule fois avant de les utiliser pour n'importe quelle intégrale ! Néanmoins le calcul des coefficients n'est pas aisé puisque le système n'est pas linéaire. Tant que la fonction n'est pas trop « agitée », la méthode donne d'excellent résultat dès l'ordre 6, ce qui est très petit et permet en pratique de conserver les coefficients dans une table. Bien sûr toute la difficulté est cachée dans la résolution du système (4.7).

Il existe plusieurs variantes de cette méthode permettant de calculer des intégrales impropres (sur $[0, +\infty[$ par exemple, ou lorsque la fonction f est singulière à une des bornes de l'intervalle). En revanche, contrairement aux méthodes de Newton-Cotes, elles s'étendent très difficilement aux intégrales multiples.

4.3 Travaux dirigés

Nous allons étudier l'intégrale suivante

$$I = \int_0^4 e^x dx = e^4 - 1 \approx 53,59815003314423.$$

Le résultat exact de cette intégrale est connu, ce qui permettra de le comparer avec les valeurs retournées par les différentes méthodes. Nous allons également nous concentrer sur le nombre de fois que la procédure calcule une valeur $f(x)$ afin de comparer, car cela a un impact important sur le temps de calcul.

- 1) **Écrire une fonction** `rectangles` et une fonction `trapezes` qui retourne également le nombre d'appels à la fonction f effectués.
- 2) **Calculer l'intégrale** I à l'aide de `rectangles` et `trapezes` en prenant un pas $h = 0,001$ (soit $N = 4000$). Comparer à la valeur exacte.
- 3) **Adapter les fonctions** `converge` et `romberg` du chapitre 3 de sorte qu'elles comptent et retournent le nombre d'appels à la fonction f effectués par `rectangles` et `trapezes`.
- 4) **Calculer l'intégrale** I à une précision de 10^{-10} avec la méthode de convergence `converge` et avec l'accélérateur de Romberg appliqués à la fonction `trapezes`.
- 5) **Écrire une fonction** `gauss` qui calcule une intégrale avec la méthode de Gauss-Legendre. Pour cela on utilisera la formule (4.5). Les valeurs des points z_i et des poids w_i pour l'intervalle $[-1, 1]$ sont données dans un tableau pour chaque ordre jusqu'à 10, vous pouvez télécharger le fichier `coeff_gauss.py` ou `coeff_gauss.m` sur moodle et lire les instructions d'utilisation qui y sont données. Notez bien que les poids x_i doivent être linéairement adaptés à l'intervalle $[a, b]$ avec les valeurs

$$x_i = \frac{a+b}{2} + z_i \frac{b-a}{2}.$$

La fonction `gauss` devra également retourner le nombre d'appels à la fonction f .

- 6) **Calculer l'intégrale** I avec la méthode de Gauss-Legendre et comparer sa précision et son efficacité avec les autres méthodes utilisées.

Séance 5

Systemes d'équations linéaires

Dans de nombreux problèmes physiques, la résolution pratique passe — souvent en dernière étape — par la résolution d'un système linéaire, c'est-à-dire de l'équation matricielle

$$Ax = b \quad (5.1)$$

dans laquelle A est une matrice carrée de taille $n \times n$ et b est un vecteur. Si elle existe, la solution recherchée est le vecteur x .

Les systèmes linéaires apparaissent dès que l'on discrétise un problème physique, par exemple en électrostatique, mécanique, acoustique *etc.* Ils traduisent souvent une équation différentielle en dimension spatiale 2 ou 3. Cela a pour conséquence que n peut prendre de très grandes valeurs (de l'ordre de 10^4 à 10^5).

Dans ce chapitre, nous allons explorer quelques méthodes qui permettent de résoudre exactement l'équation (5.1). D'autres méthodes permettent de trouver un résultat approché plus rapidement et seront abordées avec les problèmes de minimisation fonctionnelle.

Il existe une solution unique si et seulement si la matrice A est inversible. En pratique, dans les systèmes physiques, c'est toujours le cas¹. Cependant comment savoir si la matrice A est inversible avant d'essayer de résoudre le système (5.1) ? Le calcul du déterminant demande des calculs aussi coûteux que la résolution du système elle-même, ce n'est donc pas une option envisageable. C'est donc au cours de la résolution du système que nous pourrions détecter les cas problématiques.

Nous comparerons également les méthodes proposées ici en comptant combien d'opérations elles effectuent (addition, soustraction, multiplication ou division de deux nombres).

5.1 Généralités sur les systèmes linéaires

5.1.1 Inversibilité des matrices

Pour résoudre le système (5.1), il faut qu'il y ait une solution. La solution est unique si le déterminant de A est non nul. S'il est nul, il y a soit une infinité de solutions, soit aucune. Calculer un déterminant est une opération qui demande beaucoup d'opérations, les définitions mathématiques du déterminant donnent $N \sim n!$ ce qui est plus grand que tout polynôme en n . Nous verrons que l'on peut l'obtenir en environ $2n^3/3$ opérations, mais que cela revient à résoudre le système $Ax = b$. C'est pourquoi les méthodes présentées ici ne vérifient pas l'inversibilité de A .

Pour certaines matrices A , il n'est pas nécessaire de calculer le déterminant, car certaines conditions assurent qu'une matrice est inversible :

¹La raison profonde en est qu'en général on discrétise une équation différentielle faisant intervenir le laplacien, qui a des propriétés garantissant l'unicité.

- *matrice triangulaire* : les coefficients de la diagonale d'une matrice triangulaire sont les valeurs propres de cette matrice. Si aucun d'eux n'est nul, alors la matrice est inversible.
- *matrice symétrique réelle* : toute matrice symétrique (telle que ${}^tA = A$) à coefficients réels est inversible. Cette situation est très courante dans les problèmes physiques.
- *matrice à diagonale dominante stricte* : une matrice est dite à diagonale dominante stricte si pour chacune de ses lignes le coefficient de la diagonale est de module strictement plus grand que la somme des modules des autres coefficients :

$$\forall i, 1 \leq i \leq n, |a_{ii}| > \sum_{\substack{1 \leq j \leq n \\ j \neq i}} |a_{ij}|.$$

On dira aussi qu'une matrice est à diagonale dominante stricte si sa transposée l'est. Cette situation se produit également souvent en physique.

5.1.2 Conditionnement des matrices

Un système linéaire de la forme (5.1) est dit *bien conditionné* si une incertitude sur A ou sur b entraîne une incertitude du même ordre de grandeur sur la solution x obtenue. Le conditionnement du système est une mesure de la façon dont les erreurs d'arrondi risquent d'être amplifiées pendant la résolution.

Un mauvais conditionnement ne conduit pas nécessairement à un mauvais résultat, mais un bon conditionnement garantit un bon résultat.

Le meilleur indicateur de conditionnement est $\sigma = \frac{\lambda_{\max}}{\lambda_{\min}}$ où λ_{\max} et λ_{\min} sont respectivement la plus grande et la plus petite valeur propre (en module) de A , mais évaluer les valeurs propres revient presque à inverser la matrice. Si σ est proche de 1, le conditionnement est bon. Plus sigma est grand, plus il est mauvais.

Le nombre d'opérations nécessaire pour évaluer ces valeurs peut être supérieur à celui de la résolution en elle-même si on utilise une méthode itérative (Jacobi ou Gauss-Seidel, ou encore des méthodes de minimisation, voir chapitre 7). De plus les différentes techniques de recherche des valeurs propres sont elles-mêmes très sensibles au conditionnement.

D'une façon générale, on cherche rarement à calculer le conditionnement, on préfère vérifier a posteriori si le résultat obtenu est correct ou non. Par exemple si l'on appelle x^* la solution obtenue numériquement, on peut calculer le second membre $Ax^* = b^*$ et le comparer à b si l'écart est suffisamment faible cela signifie que le résultat est correct, sinon cela révèle un problème de conditionnement. Cette vérification a toutefois un coût en nombre d'opérations.

Une autre mesure du conditionnement est celle d'Hadamard :

$$H(A) = \frac{1}{|\det(A)|} \left[\prod_{j=1}^n \left(\sum_{i=1}^n a_{ij}^2 \right) \right]^{1/2} \quad (5.2)$$

On a nécessairement $H(A) \geq 1$. Plus la valeur de $H(A)$ est grande, plus le système est mal conditionné ; en pratique, la matrice est mal conditionnée si $H(A) > 100$ et bien conditionnée si $H(A) < 10$. Pour $H(A)$ compris entre 10 et 100, le conditionnement n'est pas clairement déterminé. La formule (5.2) est simple à calculer (mais coûteuse, à cause du déterminant), elle surestime cependant souvent les difficultés de calcul.

Lorsque l'on est en présence d'un système mal conditionné, il y a malheureusement peu de choses à faire. Une solution consiste à augmenter la précision du calcul mais en général elle est difficile à mettre en place. Lorsque l'on utilise la méthode de Gauss, on peut essayer d'effectuer des permutations afin d'utiliser les plus grandes valeurs possibles pour le pivot. On peut également modifier le système afin d'améliorer le conditionnement, comme multiplier certaines lignes.

5.2 Système diagonal

On dit qu'un système est diagonal si la matrice est diagonale. La résolution d'un tel système est évidente, puisque toutes les lignes sont indépendantes et la solution est donnée par

$$\text{pour tout } i \text{ tel que } 1 \leq i \leq n, \quad x_i = \frac{b_i}{a_{ii}}.$$

Pour obtenir la solution, il suffit donc de n opérations (n divisions), $N_{\text{diag}} = n$.

Complexité algorithmique

Lorsque l'on compare des algorithmes, seul le comportement asymptotique est important. Ici le comportement est «en n », c'est-à-dire que le nombre d'opération augmente aussi vite que n , on parle de *complexité linéaire*. Le coefficient devant le terme n , qui vaut 1, est appelé *préfacteur*. Ce préfacteur est utile pour comparer deux algorithmes de même complexité, c'est pourquoi nous utiliserons une notation de la complexité qui comprend le préfacteur. Nous noterons la classe de complexité avec le symbole \mathfrak{C} de sorte que la complexité C_{diagonal} de la résolution du système diagonal appartient à la classe

$$C_{\text{diagonal}} \in \mathfrak{C}(n, 1).$$

Nous calculerons la complexité algorithmique des différentes méthodes présentées dans la suite pour en faire un bilan.

5.3 Systèmes triangulaires

5.3.1 Matrices triangulaires inférieures et supérieures

Une matrice A est dite *triangulaire* si tous les indices «au-dessous» ou «au-dessus» de la diagonale sont nuls ; on dit alors qu'elle est, respectivement, triangulaire supérieure ou triangulaire inférieure. Les matrices 4×4 ci dessous sont triangulaires : A_1 est triangulaire inférieure, A_2 triangulaire supérieure.

$$A_1 = \begin{bmatrix} a_{11} & 0 & 0 & 0 \\ a_{21} & a_{22} & 0 & 0 \\ a_{31} & a_{32} & a_{33} & 0 \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \quad A_2 = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ 0 & a_{22} & a_{23} & a_{24} \\ 0 & 0 & a_{33} & a_{34} \\ 0 & 0 & 0 & a_{44} \end{bmatrix}$$

Pour résoudre un système dont la matrice est de la forme A_1 (triangulaire inférieure) il suffit de commencer par résoudre pour x_1 :

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

et de procéder ensuite à la résolution des lignes dans l'ordre :

$$x_2 = \frac{b_2 - a_{21}x_1}{a_{22}}.$$

Note —

Avant de décrire la procédure générale, c'est-à-dire l'algorithme, de résolution de système triangulaire supérieur, il faut prendre note de la convention d'écriture mathématique suivante : dans la suite de ce chapitre, les algorithmes sont décrits avec une convention supplémentaire

concernant les sommes :

Si $N > M$ la somme $\sum_{i=N}^M (\dots)$ est nulle.

L'algorithme de résolution d'un système triangulaire inférieur est le suivant

Algorithme 6 : Résolution d'un système linéaire triangulaire inférieur

définir la fonction *triangulaire_inf*

• \rightarrow une matrice A de taille $n \times n$

• \rightarrow un vecteur b de taille n

pour tout i allant de 1 à n **faire**

$$\left[x_i \leftarrow \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j \right) \right]$$

• $\rightarrow x = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}$

On observe que l'on divise par les éléments de la diagonale. Si l'un de ces éléments est nul, c'est que le déterminant (qui est ici simplement le produit $a_{11}a_{22} \cdots a_{nn}$) est nul et donc que la matrice n'est pas inversible. Prendre en compte cette possibilité est facile à tester avant de lancer le calcul, ou même au fur et à mesure du calcul.

Complexité algorithmique

Pour calculer x_i (avec $1 \leq i \leq n$), il faut donc effectuer on doit effectuer $i - 1$ multiplications et $(i - 1) - 1 + 1$ additions/soustractions (en comptant la soustraction de b_i) et une division, soit un total de $2i - 1$ opérations. En utilisant la formule de sommation

$$\sum_{k=1}^n (2k - 1) = n^2, \quad \left(\text{soit } \sum_{k=1}^n k = \frac{n(n+1)}{2} \right) \quad (5.3)$$

on en déduit que le nombre total d'opérations est n^2 , on a donc

$$C_{\text{triangle}} \in \mathfrak{C}(n^2, 1)$$

La résolution des systèmes triangulaires supérieurs s'effectue à l'aide d'un algorithme très proche de l'algorithme présenté et il a la même complexité algorithmique.

On observe une complexité algorithmique (n^2) plus forte que celle du système diagonal, (n), qui vient du fait que les inconnues sont dépendantes. Cependant, une matrice triangulaire reste un cas particulier. Nous allons maintenant voir comment un algorithme, la décomposition de Crout, qui permet de revenir à un système triangulaire à partir d'une matrice quelconque.

5.3.2 Décomposition de Crout (décomposition LU ou LR)

La méthode de décomposition de Crout permet de décomposer une matrice de forme générale A (non-diagonale, non triangulaire) en produit

$$A = LU$$

dans lequel L est une matrice triangulaire inférieure (*lower* en anglais) et U est une matrice triangulaire supérieure (*upper* en anglais). On l'appelle aussi parfois simplement «méthode de décomposition LU» ou encore «méthode de décomposition LR» (pour *left-right*, le L désignant toujours la matrice triangulaire inférieure).

Algorithme 7 : Algorithme de Crout (décomposition LU / LR)

définir la fonction *crout*

→ une matrice A de taille $n \times n$

pour tout j allant de 1 à $n - 1$ **faire**

pour tout i allant de 1 à j **faire**

$$u_{ij} \leftarrow a_{ij} - \sum_{k=1}^{i-1} \ell_{ik} u_{kj} \quad \blacksquare \text{ (a)}$$

pour tout i allant de $j + 1$ à n **faire**

$$\ell_{ij} \leftarrow \frac{1}{u_{jj}} \left(a_{ij} - \sum_{k=1}^{j-1} \ell_{ik} u_{kj} \right) \quad \blacksquare \text{ (b)}$$

pour tout i allant de 1 à n **faire**

$$\ell_{ii} \leftarrow 1$$

$$u_{in} \leftarrow a_{in} - \sum_{k=1}^{i-1} \ell_{ik} u_{kn} \quad \blacksquare \text{ (c)}$$

$$L = \begin{bmatrix} \ell_{11} & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & 0 \\ \ell_{n1} & \cdots & \cdots & \ell_{nn} \end{bmatrix} \text{ et } U = \begin{bmatrix} u_{11} & \cdots & \cdots & u_{1n} \\ 0 & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & u_{nn} \end{bmatrix}$$

Une fois la décomposition effectuée il suffit de résoudre deux systèmes dont les matrices sont triangulaires et pour lesquelles la procédure de résolution a été abordée précédemment. En effet, si $Ax = b$ alors $LUx = b$. Résolvons $Ly = b$, nous connaissons donc le vecteur y , puis résolvons $Ux = y$. On a bien

$$Ax = LUx = L(Ux) = Ly = b.$$

Complexité algorithmique

Le nombre d'opérations de la décomposition d'une matrice $n \times n$ est détaillé ici, avec des rappels notés (a), (b) et (c) vers les lignes correspondantes de l'algorithme 5.3.2.

$$\begin{aligned} N_{\text{Crout}} &= \sum_{j=1}^{n-1} \left[\sum_{i=1}^j \underbrace{(2i-1)}_{(a)} + \sum_{i=j+1}^n \underbrace{(2i)}_{(b)} \right] + \sum_{i=1}^n \underbrace{(2i)}_{(c)} \\ &= \sum_{j=1}^{n-1} \left[j^2 + (n(n+1) - j(j+1)) \right] + 2 \frac{n(n+1)}{2} \\ &= (n-1)n(n+1) - \frac{n(n-1)}{2} + n^2 + n = n^3 + \frac{n^2 + n}{2}. \end{aligned}$$

Dans ce calcul, nous avons utilisé la relation (5.3). On observe que la première boucle principale sur j appartient à la classe de complexité $\mathfrak{C}(n^3, 1)$. La deuxième boucle principale ajuste les

derniers coefficients, elle est de complexité $\mathfrak{C}(n^2)$ donc inférieure. On voit ici apparaître une complexité encore plus élevée que pour les matrices triangulaires, avec un terme dominant n^3 . Cette complexité correspond à une matrice arbitraire. Nous verrons dans la suite que la complexité $\mathfrak{C}(n^3)$ est caractéristique des résolutions de systèmes généraux et que le préfacteur (ici égal à 1) pourra être amené à prendre des valeurs différentes, mais l'exposant de la complexité ne descendra pas en dessous de 3^2 .

Application de la méthode de Crout à des systèmes matriciels.

La méthode de Crout présente l'avantage de fournir les matrices L et U sans prendre en compte le second membre b , ce qui permet de résoudre un autre système dont les coefficients sont les mêmes sans avoir à recalculer L et U . Ainsi chaque résolution du système consistant en la résolution de deux systèmes triangulaires, une fois L et U calculées, la complexité devient $\mathfrak{C}(n^2)$.

Cela permet d'étendre la résolution à des systèmes linéaires du type

$$AX = B$$

pour lesquels B est le second membre connu de taille $n \times p$ (p est le nombre de colonne de B). L'inconnue X est une matrice de taille $n \times p$ telle que, si on note $x_{[j]}$ la j^{e} colonne de X et $b_{[j]}$ la j^{e} colonne de B alors si X est solution de $AX = B$ cela signifie que pour tout j tel que $1 \leq j \leq p$ on a

$$Ax_{[j]} = b_{[j]}.$$

Cette remarque permet également le calcul de la matrice inverse de A en posant $m = n$ et $B = I$ la matrice identité de taille $n \times n$. La solution X du système

$$AX = I$$

est donc $X = A^{-1}$. La complexité pour obtenir l'inverse de A par cette méthode est

$$N_{\text{inverse LU}} = N_{\text{Crout}} + 2nN_{\text{triangle}} = n^3 + \frac{n(n+1)}{2} + 2n n^2 = 3n^3 + \frac{n(n+1)}{2}.$$

C'est-à-dire que la complexité de l'inversion de matrice est d'au plus $\mathfrak{C}(n^3, 3)$.

Remarques supplémentaires

La diagonale la matrice L est composée uniquement de 1, son déterminant vaut donc 1. Comme $\det A = \det L \det U$, on en déduit que le déterminant de A est égal à celui de U , qui est le produit des éléments de sa diagonale.

Le nombre total de coefficients calculés par l'algorithme de Crout est n^2 , il est donc possible de tous les stocker dans une seule matrice $n \times n$, en évitant de prendre de la place en mémoire pour conserver des zéros.

5.4 Méthode de Gauss

La méthode de Gauss est une méthode consistant à éliminer des coefficients par combinaisons linéaires des lignes. Chaque combinaison linéaire sur les lignes de la matrice s'accompagne de la même combinaison linéaire sur les coefficients du second membre. Après avoir effectué un «pivot» pour chaque ligne à partir de la deuxième, on obtient un système triangulaire que l'on peut alors

²Des méthodes très complexes permettent de faire baisser la complexité jusqu'à $\mathfrak{C}(n^{2,7})$ environ mais le préfacteur est tellement grand que ces méthodes ne sont utilisées que pour des systèmes de très grandes tailles.

facilement résoudre ou bien on peut effectuer une seconde série de pivots. Cette méthode est générale, elle est souvent utilisée pour résoudre manuellement les systèmes dont l'ordre est de l'ordre de 3 à 10. C'est en effet la méthode de résolution de système général que l'on apprend le plus souvent en cours d'algèbre linéaire.

Principe de la méthode

Les combinaisons linéaires que l'on peut effectuer simultanément sur la matrice et le second membre sans en changer la solution sont les suivantes

(**M**)^{*} : multiplier une ligne par un nombre non nul ;

(**P**)^{*} : permuter deux lignes ;

(**C**) : ajouter une ligne multipliée par un nombre à une autre ligne.

Les opérations marquées d'une étoile modifient le déterminant de la matrice³. Si on souhaite conserver le déterminant inchangé, il faut utiliser exclusivement l'opération (**C**) mais c'est possible comme le montre l'exemple suivant

Matrice	Opérations	Second membre	Commentaire
$\begin{bmatrix} 5 & 2 & -3 \\ 1 & 4 & -1 \\ -2 & -1 & 6 \end{bmatrix}$	L_1 L_2 L_3	$\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$	Matrice et second membre initiaux.
$\begin{bmatrix} 5 & 2 & -3 \\ 0 & \frac{18}{5} & -\frac{2}{5} \\ 0 & -\frac{1}{5} & \frac{24}{5} \end{bmatrix}$	L_1 $L_2 \leftarrow L_2 - \frac{1}{5}L_1$ $L_3 \leftarrow L_3 + \frac{2}{5}L_1$	$\begin{bmatrix} 1 \\ \frac{9}{5} \\ \frac{17}{5} \end{bmatrix}$	On ajoute la première ligne multipliée par $-a_{i1}/a_{11}$ à la ligne i , ce qui fait apparaître des zéros dans la première colonne de la matrice. On appelle le coefficient a_{11} le <i>pivot</i> de cette opération.
$\begin{bmatrix} 5 & 2 & -3 \\ 0 & \frac{18}{5} & -\frac{2}{5} \\ 0 & 0 & \frac{43}{9} \end{bmatrix}$	L_1 L_2 $L_3 \leftarrow L_3 + \frac{1}{18}L_2$	$\begin{bmatrix} 1 \\ \frac{9}{5} \\ \frac{7}{2} \end{bmatrix}$	On ajoute la deuxième ligne multipliée par $\frac{1}{18}$ à la troisième pour faire apparaître un zéro supplémentaire. La matrice est maintenant <i>triangulaire</i> .

Nous pouvons maintenant conclure la résolution avec la méthode de résolution des systèmes triangulaires. Bien entendu, cette méthode échoue si le déterminant est nul, ce qui fera tôt ou tard faire apparaître une ligne de zéros. Sans adaptation, elle échouera également si l'un des pivots est nul. C'est pourquoi une stratégie consiste à permuter des lignes en choisissant le meilleur pivot, qui sera celui de plus grande valeur absolue (ou de plus grand module pour les nombres complexes).

Complexité algorithmique

Considérons pour commencer l'algorithme simple, tel que présenté ci-dessus, sans recherche de meilleur pivot. On se place dans le cas général avec une matrice $n \times n$. Une opération (**C**) avec pour pivot le coefficient a_{ii} consiste à modifier les lignes numérotées $i + 1$ jusqu'à n en effectuant $n - i + 1$ multiplications et autant d'additions sur la matrice. Ce qui fait $2(n - i)(n - i + 1)$

³(**M**) multiplie le déterminant par le nombre utilisé et (**P**) en change le signe.

opérations. Pour atteindre la situation où la matrice est triangulaire, on utilise $n - 1$ pivots pour i allant de 2 à n soit un total de

$$N_{\text{Gauss}} = \sum_{i=2}^n 2(n-i)(n-i+1) \underset{(k \leftarrow n-i+1)}{=} 2 \sum_{k=1}^{n-1} k(k-1) = \frac{2}{3}(n^3 - n).$$

La classe de complexité est $\mathfrak{C}(n^3, 2/3)$, c'est-à-dire $\mathfrak{C}(n^3)$ comme la décomposition de Crout, mais avec un préfacteur $2/3$. Nous allons voir comment l'exploiter ce facteur. Nous avons de nouveau utilisé l'identité (5.3) ainsi que l'égalité

$$\sum_{k=1}^n k^2 = \frac{n(n+1)(2n+1)}{6} \quad \left(\text{soit } \sum_{k=1}^n (3k^2 - 3k + 1) = n^3 \right).$$

Ce faisant, nous avons également effectué des opérations sur le second membre au nombre de

$$M_{\text{Gauss}} = \sum_{i=2}^n 2(n-i+1) = 2 \sum_{k=1}^{n-1} k = n^2 - n,$$

ce qui n'affecte pas la complexité.

Faut-il s'arrêter lorsque la matrice est triangulaire et utiliser ensuite une méthode de résolution pour matrices triangulaires ? Si l'on effectue les mêmes opérations que précédemment, en commençant par la dernière ligne et en éliminant les coefficients au-dessus de la diagonale, on obtient un système diagonal en effectuant au total $2N_{\text{Gauss}} + 2M_{\text{Gauss}} = \frac{4}{3}n^3 + 2n^2 - \frac{4}{3}n$ opérations (complexité $\mathfrak{C}(n^3, 4/3)$) alors que si l'on résout le système triangulaire, on a seulement $N_{\text{Gauss}} + M_{\text{Gauss}} + N_{\text{triangle}} = \frac{2}{3}n^3 + 2n^2 - \frac{5}{3}n$ opérations, de complexité $\mathfrak{C}(n^3, 2/3)$. C'est donc la résolution du système triangulaire qu'il faut utiliser. On constate que la méthode de Gauss est également plus efficace que la décomposition LU de Crout.

Qu'en est-il pour calculer l'inverse d'une matrice ? Pour inverser une matrice, il suffit d'appliquer la méthode de Gauss en modifiant simultanément la matrice A et une autre matrice initialement égale à I , la matrice identité, qui compte donc comme n vecteurs au second membre. La complexité obtenue est alors

$$N_{\text{inv. Gauss}} = N_{\text{Gauss}} + nM_{\text{Gauss}} + nN_{\text{triangle}} = \frac{2}{3}(n^3 - n) + n(n^2 - n) + nn^2 = \frac{8n^3 - 3n^2 - 2n}{3}.$$

La méthode est encore de complexité $\mathfrak{C}(n^3, 8/3)$ ce qui est meilleur que la complexité $\mathfrak{C}(n^3, 3)$ obtenue avec la méthode LU. Il est donc plus avantageux d'inverser une matrice avec la méthode de Gauss.

5.5 Méthodes itératives

Contrairement aux méthodes présentées précédemment, les méthodes itératives effectuent un nombre d'opérations variable qui dépend des coefficients de la matrice A . Ces méthodes présentent l'avantage ne pas exiger beaucoup de mémoire et de permettre la résolution des systèmes matriciels, comme la méthode de Crout. En revanche, ces méthodes nécessitent de définir un critère de convergence et rien n'assure qu'elle vont effectivement converger.

5.5.1 Méthode de Jacobi

Le principe de la méthode de Jacobi est de définir un vecteur x et de résoudre ligne par ligne le système comme si les n équations étaient indépendantes, en ne considérant comme inconnue que

le coefficient x_i de la ligne i . Par exemple, la première étape de la méthode de Jacobi appliquée au système donné en exemple pour la méthode de Gauss serait la suivante, ici présentée avec le vecteur initial $x^0 = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$

$$\begin{cases} 5x_1 & +2 \times \underset{(\text{= } x_2^0)}{1} & -3 \times \underset{(\text{= } x_3^0)}{1} & = 1 \\ 1 \times \underset{(\text{= } x_1^0)}{1} & +4x_2 & -1 \times 1 & = 2 \\ -2 \times 1 & -1 \times 1 & +6 \times x_3 & = 3 \end{cases} \quad (5.4)$$

La solution de ce système simple est $x^1 = \begin{bmatrix} 2/5 \\ 1/2 \\ 1 \end{bmatrix}$ et on a $Ax^1 = b^1 = \begin{bmatrix} 0 \\ 7/5 \\ 47/10 \end{bmatrix}$. Il n'est pas surprenant que $b^1 \neq b$ car notre méthode ne constitue pas une résolution correcte du système. Toutefois, il est possible de comparer b^1 et b . Si ces deux vecteurs sont proches, alors le vecteur x^1 sera proche de la solution. La méthode consiste alors à répéter l'opération ci-dessus en partant du vecteur x^1 obtenu et de recommencer jusqu'à ce que, après k itérations, $b^k = Ax^k$ et b soient assez « proches ».

La conception de cet algorithme repose entièrement sur le sens que l'on donne à cette proximité. En utilisant une norme $\|\cdot\|$ sur les vecteurs (comme $\|x\|_1 = \sum_{i=1}^n |x_i|$ ou $\|x\|_\infty = \max_i |x_i|$) on peut mesurer l'écart entre b^k et b . Pour des questions de convergence, nous utiliserons un écart relatif, noté $\|b' - b\|/\|b\|$ dans l'algorithme ci-dessous.

Algorithme 8 : Algorithme de Jacobi

définir la fonction *jacobi*

- \rightarrow A , matrice de taille $n \times n$
- \rightarrow b , vecteur second membre de dimension n
- \rightarrow x , vecteur initial de dimension n
- \rightarrow ε , précision demandée (doit vérifier $\varepsilon \ll 1$)

$b' \leftarrow$ vecteur nul de dimension n

tant que $\|b' - b\|/\|b\| > \varepsilon$ **faire**

pour tout i de 1 à n **faire**

$$x'_i \leftarrow \frac{1}{a_{ii}} \left(b_i - \sum_{\substack{1 \leq j \leq n \\ j \neq i}} a_{ij} x_j \right)$$

 (*) on peut modifier la relaxation ici, voir texte

$b' \leftarrow Ax'$

$x \leftarrow x'$

• \leftarrow x

Pour éviter les situations où l'algorithme ne converge pas, il peut être judicieux d'ajouter compteur à la boucle **tant que**. Il serait utile de compléter la valeur de retour x par une indication de comment l'algorithme s'est arrêté, à savoir si c'est par succès de la convergence ou par dépassement du maximum de boucles effectuées.

Relaxation modifiée

Pour améliorer les performances de l'algorithme de Jacobi, on peut utiliser une opération complémentaire dite de contrôle de la relaxation (ce mot désigne le fait de se rapprocher de la solution).

Après avoir calculé x' , on effectue l'opération suivante (à l'endroit indiqué par $(*)$ dans le pseudo-code)

$$x' \leftarrow x + \omega(x' - x)$$

où $0 < \omega < 2$. Le vecteur $x' - x$ représente d'une certaine façon la vitesse de déplacement du vecteur x et ω un facteur qui modifie cette vitesse. Si $\omega > 1$ on parle de sur-relaxation, et si $0 < \omega < 1$ on parle de sous-relaxation (la valeur $\omega = 1$ correspond à la situation de relaxation non modifiée). Les valeurs $\omega > 2$ et $\omega < 0$ conduisent généralement à des instabilités et risquent de briser la convergence. La valeur utilisée le plus fréquemment est $\omega = 2/3$, qui permet d'améliorer significativement la vitesse de convergence.

Complexité algorithmique

On ne peut que calculer la complexité algorithmique d'une étape, car le nombre d'étape est inconnu. On notera ce nombre P_{Jacobi} . En tenant compte du fait que le produit Ax' demande $2n^2 - n$ opérations, on obtient

$$P_{\text{Jacobi}} = 2n - 1 + 2n^2 - n + N_{\text{norme}} + 1 = 2n^2 + n + N_{\text{norme}}$$

où N_{norme} est le coût de calcul de la norme d'un vecteur. La classe de complexité d'une itération est notée $\mathfrak{c}(n^2, 2)$. Comme on a toujours $C_{\text{norme}} \in \mathfrak{C}(n)$, on voit que le coût d'une étape est dominé par le calcul du produit Ax' . Les modifications de relaxation sont également de complexité $\mathfrak{C}(n)$, donc ne changent pas la complexité de l'algorithme de Jacobi, qui est $\mathfrak{c}(n^2)$. On en déduit que si la méthode de Jacobi fournit un résultat à la précision ε demandée en moins de $\frac{2}{3}n$ tours, alors elle est plus efficace que la méthode de Gauss. En pratique, malheureusement, la méthode de Jacobi converge assez lentement.

5.5.2 Méthode de Gauss-Seidel

La méthode de Gauss-Seidel consiste à modifier la méthode de Jacobi en résolvant le système (5.4) en utilisant les coefficients de x' qui viennent d'être calculés plutôt que les coefficients de x . Ainsi, pour résoudre l'équation correspondant à la ligne i on utilisera les coefficients de x' déjà calculés (de x'_1 à x'_{i-1}) et on utilisera les coefficients de x pour les valeurs de x_{i+1} à x_n , ce qui se traduit par l'algorithme suivant.

Algorithme 9 : Algorithme de Gauss-Seidel

définir la fonction *gauss_seidel*

- $\rightarrow A$, matrice de taille $n \times n$
- $\rightarrow b$, vecteur second membre de dimension n
- $\rightarrow x$, vecteur initial de dimension n
- $\rightarrow \varepsilon$, précision demandée (doit vérifier $\varepsilon < 1$)
- $b' \leftarrow$ vecteur nul de dimension n
- tant que** $\|b' - b\|/\|b\| > \varepsilon$ **faire**
 - pour tout** i de 1 à n **faire**
 - $$x'_i \leftarrow \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij}x'_j - \sum_{j=i+1}^n a_{ij}x_j \right)$$
 - (*) on peut modifier la relaxation ici*
 - $b' \leftarrow Ax'$
 - $x \leftarrow x'$
- $\leftarrow x$

Il est également possible d'ajouter un contrôle de la relaxation à l'endroit marqué (*). La complexité est la même que celle de la méthode de Jacobi dès que l'on utilise le même critère de convergence et la même norme :

$$C_{\text{Gauss-Seidel}} = C_{\text{Jacobi}}.$$

L'expérience montre que cette méthode converge plus rapidement vers une solution que la méthode de Jacobi, c'est celle-ci qui est donc à privilégier.

5.6 Travaux dirigés

1) **Implémenter les fonctions** *triangulaire_inf* et *triangulaire_sup*. Dans un souci d'efficacité, les sommes comme

$$\sum_{j=1}^{i-1} a_{ij}x_j$$

ne doivent pas être implémentées avec une boucle, mais elles doivent utiliser les fonctionnalités de calcul vectoriel du langage, comme le produit scalaire ou l'extraction de sous-matrice. L'exemple ci-dessus doit, par exemple, procéder à l'extraction des coefficients $a_{i1}, \dots, a_{i,i-1}$ de la matrice A et faire le produit scalaire avec le « sous-vecteur » de x contenant les coefficients x_1 à x_{i-1} . Vous trouverez dans les annexes des explications sur ces opérations d'algèbre linéaire (Annexe A.3.4 pour python et Annexe B.4 pour MATLAB/octave).

2) **Résoudre numériquement** l'équation $Ax = b$ avec

$$A = \begin{bmatrix} 1 & 3 & 3 & 1 \\ 0 & 1 & 2 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \text{et} \quad b = \begin{bmatrix} 1 \\ 2 \\ 4 \\ 8 \end{bmatrix}.$$

3) **Implémenter la décomposition de Crout** ou décomposition LU en suivant l'algorithme de

la page 30. Résoudre les équations $A_1x = b$ et $A_2x = b$ avec

$$A_1 = \begin{bmatrix} 10 & 4 & 2 & 3 \\ 2 & 7 & 1 & 3 \\ 6 & 4 & 14 & 2 \\ 2 & 6 & 3 & 15 \end{bmatrix}, \quad A_2 = \begin{bmatrix} 10 & 7 & 8 & 7 \\ 7 & 5 & 6 & 5 \\ 8 & 6 & 10 & 9 \\ 7 & 5 & 9 & 10 \end{bmatrix} \quad \text{et} \quad b = \begin{bmatrix} 1 \\ 7 \\ 15 \\ 2 \end{bmatrix}.$$

4) **Utiliser la décomposition LU** de la matrice A_2 ci-dessus pour calculer son inverse. Vérifier le résultat en utilisant le produit de matrice du langage utilisé.

5) **Implémenter les méthodes de Jacobi et de Gauss-Seidel.** Commencer par la méthode de Jacobi puis remarquer que Gauss-Seidel s'obtient *en changeant un seul caractère* !

6) **Recalculer les solutions des équations précédentes à l'aide de ces deux méthodes** en partant des différents vecteurs initiaux :

$$x_1 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \quad x_2 = \begin{bmatrix} 10000 \\ 10000 \\ 10000 \\ 10000 \end{bmatrix}.$$

7) **Évaluer les temps de convergence** en utilisant les fonctions de chronométrage décrites en annexe A.5.1 pour python et B.5.1 pour MATLAB/octave.

Séance 6

Résolution d'équations différentielles

L'objectif de cette séance est voir quelques méthodes numériques pour résoudre une équation différentielle ordinaire de la forme

$$\frac{dy}{dt} = f(y, t), \quad (6.1)$$

dans laquelle $f : \mathbf{R}^d \times \mathbf{R}^+ \rightarrow \mathbf{R}^d$ est une fonction continue par rapport au premier argument (y).

Les méthodes présentées sont d'efficacité croissante. Elles sont pour la plupart simples à implémenter et très générales. Elles ne pourront pas être utilisées dans des situations où le chaos joue un rôle important, mais elles sont efficaces dans la plupart des situations rencontrées en physique.

6.1 Équations différentielles ordinaires

Les équations différentielles du premier ordre sont bien sûr de la forme (6.1) pour une fonction y à valeurs dans \mathbf{R} ou \mathbf{C} , comme par exemple l'équation de relaxation

$$\frac{dy}{dt} = \frac{E - y}{\tau}.$$

Cependant, nous allons voir que les équations différentielles d'ordre $p > 1$ peuvent être reformulées comme des équations différentielles de la forme (6.1) dans lesquelles y est une fonction à valeur dans \mathbf{R}^p ou \mathbf{C}^p .

Équation différentielle linéaire homogène à coefficients constants

Prenons un exemple simple et supposons que l'on cherche à résoudre l'équation différentielle — du deuxième ordre — de l'oscillateur harmonique. La fonction $\theta(t)$ est notre solution et elle vérifie l'équation différentielle

$$\frac{d^2\theta}{dt^2} + \omega_0^2\theta = 0 \quad \text{avec } \theta(0) = \theta_0.$$

Pour utiliser la formule (6.1), on pose

$$y(t) = \begin{bmatrix} \theta(t) \\ \dot{\theta}(t) \end{bmatrix}.$$

(On note de façon abrégée $\dot{\theta}$ la dérivée $\frac{d\theta}{dt}$.) Calculons alors $\frac{dy}{dt}$

$$\frac{dy}{dt} = \begin{bmatrix} \dot{\theta}(t) \\ \ddot{\theta}(t) \end{bmatrix} = \begin{bmatrix} \dot{\theta}(t) \\ -\omega_0^2\theta(t) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -\omega_0^2 & 0 \end{bmatrix} y(t).$$

Ainsi la fonction f pour cet exemple est $f(y, t) = Hy$ où H est la matrice $\begin{bmatrix} 0 & 1 \\ -\omega_0^2 & 0 \end{bmatrix}$. Cette méthode se généralise à tous les ordres.

Coefficients dépendant du temps

Un exemple important d'équation différentielle est celui d'une équation différentielle linéaire à coefficients variables comme

$$(1 - t^2) \frac{d^2 t}{dy^2} - t \frac{dt}{dy} + y = 0$$

En réécrivant l'équation sous la forme

$$\frac{d^2 t}{dy^2} = -\frac{y}{1 - t^2} + \frac{t}{1 - t^2} \frac{dt}{dy},$$

on déduit que $f(y, t) = A(t)y$ avec

$$A(t) = \begin{bmatrix} 0 & 1 \\ -\frac{1}{1-t^2} & \frac{t}{1-t^2} \end{bmatrix}.$$

Équation différentielle non homogène (avec second membre)

Il se peut que le second membre f dépende du temps t . Par exemple, si on considère l'oscillateur forcé régit par l'équation

$$\frac{d^2 \theta}{dt^2} + \omega_0^2 \theta = a \cos \omega t \quad (\omega \neq \omega_0)$$

on a alors la fonction $f(y, t) = Hy + \begin{bmatrix} 0 \\ a \cos \omega t \end{bmatrix}$.

Équation différentielle non linéaire

La fonction peut également ne pas être linéaire, par exemple le pendule anharmonique amorti dont l'équation est

$$\frac{d^2 \theta}{dt^2} + \gamma \frac{d\theta}{dt} + \omega_0^2 \sin \theta(t) = 0.$$

Dans ce cas, f n'est pas linéaire, mais est définie par

$$f\left(\begin{bmatrix} y_0 \\ y_1 \end{bmatrix}, t\right) = \begin{bmatrix} y_1 \\ -\gamma y_1 - \omega_0^2 \sin(y_0) \end{bmatrix}.$$

6.2 Algorithmes d'Euler

Les algorithmes d'Euler se basent sur le développement limité au premier ordre de $y(t+h)$, lorsque h est un nombre réel petit devant 1. Ils permettent de calculer $y(t+h)$ à partir de $y(t)$ et f . En répétant ce processus à partir d'une condition initiale $y(t_0) = y_0$, on obtient les valeurs de y aux temps $t_k = t_0 + kh$, que l'on notera par la suite $y_k = y(t_k) = y(t_0 + kh)$, c'est-à-dire que la solution est déterminée par pas de temps discrets et réguliers. On pourra alors en tracer la courbe approchée par des segments.

6.2.1 Algorithme explicite

L'algorithme explicite d'Euler s'obtient en exprimant la dérivée $\frac{dy}{dt}$ par son expression comme dérivée à droite dans l'équation (6.1) (voir page 3 dans le chapitre 1 portant sur les dérivées) :

$$\frac{y_{k+1} - y_k}{h} = f(y_k, t_k)$$

ce qui donne en résolvant pour y_{k+1}

$$y_{k+1} = y_k + hf(y_k, t_k).$$

6.2.2 Algorithme implicite

L'algorithme d'Euler implicite se base sur l'utilisation du même développement limité mais en utilisant la dérivée à gauche, ainsi

$$\frac{y_k - y_{k-1}}{h} = f(y_k, t_k),$$

la résolution de l'équation obtenue (en décalant l'indice k de 1)

$$y_{k+1} = y_k + hf(y_{k+1}, t_{k+1}) \quad (6.2)$$

n'est plus immédiate comme pour la méthode explicite, car l'inconnue y_{k+1} apparaît des deux côtés du signe $=$. Il faut alors utiliser une méthode de résolution d'équation, comme par exemple celle du point fixe, décrite au chapitre 2 page 11 qui donnera de bons résultats car le petit élément h permet de s'assurer que la dérivée du second membre $g' : y \mapsto h \frac{df}{dy}(y, t_{k+1})$ est compatible avec une convergence rapide. Pour améliorer l'algorithme implicite d'Euler, il peut être intéressant d'utiliser la méthode de Newton (décrite page 13) pour résoudre (6.2) en limitant le nombre de boucle à deux ou trois.

6.3 Algorithmes de Runge-Kutta

6.3.1 Algorithme d'ordre 2

L'idée de l'algorithme de Runge-Kutta est d'utiliser le fait que la dérivée à droite de y en t_k , $\frac{y_{k+1} - y_k}{h}$ peut s'interpréter comme la dérivée centrée de y à $t_k + \frac{h}{2}$. Or on a vu au chapitre 1 que la dérivée centrée est bien meilleure que la dérivée à droite car elle est précise à l'ordre deux. L'idée est alors de considérer un «demi-pas» $h/2$ et de calculer une valeur de y'_k au temps $t'_k = t_k + \frac{h}{2}$. Pour cela, on utilise la méthode d'Euler vue précédemment :

$$y'_k = y_k + \frac{h}{2} f(y_k, t_k)$$

On calcule alors le pas de y_k à y_{k+1} en utilisant la valeur calculée au demi-pas

$$y_{k+1} = y_k + hf(y'_k, t'_k).$$

Cette méthode est un ordre supérieure à la méthode d'Euler, elle donne un résultat précis à l'ordre h^2 .

6.3.2 Algorithme d'ordre 4

La formule de Runge-Kutta la plus utilisée est celle d'ordre 4. Elle se base sur une pondération de plusieurs incréments. Nous ne la démontrerons pas. Voici les calculs effectués à chaque pas.

$$\begin{aligned}\delta_1 &= f(y_k, t_k), & y'_k &= y_k + \frac{h}{2}\delta_1, & (\text{même } y'_k \text{ que pour l'ordre 2}) \\ \delta_2 &= f(y'_k, t'_k), & y''_k &= y_k + \frac{h}{2}\delta_2, & \text{avec } t'_k = t_k + \frac{h}{2} \\ \delta_3 &= f(y''_k, t'_k), & y_{k+1}^* &= y_k + h\delta_3, \\ \delta_4 &= f(y_{k+1}^*, t_{k+1}), & y_{k+1} &= y_k + \frac{h}{6}(\delta_1 + 2\delta_2 + 2\delta_3 + \delta_4).\end{aligned}$$

6.4 Travaux dirigés

1) **Écrire un programme** `euler` qui calcule la solution de l'équation différentielle $\dot{y} = f(y, t)$ avec un intervalle de temps h en effectuant n pas. La fonction doit retourner deux tableaux, `y` et `t` qui contiennent les temps t_k et la valeur de y de sorte que $y_k = y(t_k)$.

2) **Calculer et tracer** la solution de l'équation différentielle

$$\dot{y} = \frac{E - y}{\tau} \quad (6.3)$$

avec pour condition initiale $y(0) = 0$, pour 100 pas de temps $h = 0,05$. On prendra $E = 10$ et $\tau = 1$. Calculer et tracer la solution exacte de cette équation sur le même graphique. Que constate-t-on ?

3) **Implémenter** l'algorithme de Runge-Kutta d'ordre 2 et celui d'ordre 4, `runge_kutta2` et `runge_kutta4`, qui prennent les mêmes arguments que `euler`.

4) **Utiliser ces algorithmes** pour calculer la solution de l'équation (6.3) et la tracer sur le même graphique que la solution obtenue avec `euler` avec un 10 pas de temps $h = 0,8$ (ce qui permettra d'observer les différences).

5) **Résoudre** l'équation différentielle

$$(1 - t^2)\ddot{y} - t\dot{y} + y = 0, \quad \begin{cases} y(0) = 1, \\ \dot{y}(0) = 1. \end{cases}$$

Comparer les résultats obtenus avec les trois méthodes et la solution exacte $t \mapsto t + \sqrt{1 - t^2}$.

6) **Utiliser la méthode d'accélération** de votre choix pour contrôler la convergence de la solution de l'équation $\dot{y} + y^2 = 0$ avec $y(0) = 1$. On prendra comme critère de convergence la valeur de y en $t = 5$.

Séance 7

Méthodes de descente

7.1 Principe mathématique

Illustration en deux dimensions

Les méthodes de descente sont des méthodes générales de minimisation que l'on utilise énormément en physique numérique pour calculer et localiser le minimum d'une fonction telle que le potentiel ou l'énergie. Ces méthodes sont *itératives* : elles sont basées sur des déplacements successifs. À chaque déplacement on détermine une direction puis la distance à parcourir dans cette direction en optimisant les choix pour converger vers le minimum de la fonction, comme illustré sur la figure 7.1.

Généralités

Considérons maintenant une fonction f définie sur \mathbf{R}^n dont on cherche un minimum, en partant d'un point $x_0 \in \mathbf{R}^n$. Lors d'un déplacement $u \in \mathbf{R}^n$, la fonction f varie de

$$f(x + u) - f(x) \approx u \cdot \nabla f(x).$$

Comme l'on cherche à minimiser f , on cherche à avoir $f(x + u) < f(x)$, c'est-à-dire $u \cdot \nabla f(x) < 0$. La méthode itérative consiste donc à construire une suite de positions x_0, x_1, \dots telle que

$$f(x_0) > f(x_1) > f(x_2) > \dots$$

et les méthodes consistent à calculer x_{k+1} à partir de x_k avec

$$x_{k+1} = x_k + \lambda_k u_k, \quad \text{avec } u_k \cdot \nabla f(x_k) < 0 \text{ et } \lambda_k > 0.$$

Les différentes méthodes que nous allons étudier proposent différentes façons de trouver un vecteur u_k et un scalaire λ_k qui satisfont ces demandes tout en essayant de faire en sorte que $f(x_{k+1})$ soit le plus petit possible.

7.2 Méthode du gradient (plus forte pente)

Analogie mécanique

La méthode de la plus forte pente est une méthode de descente assez générale dans laquelle on utilise le gradient comme direction de déplacement. Illustrons cette méthode dans le cas d'une fonction $f : \mathbf{R}^2 \rightarrow \mathbf{R}$. La valeur de f en un point (x, y) peut être vue comme une altitude et le gradient est l'opposé de la direction que prendrait un ballon ou un liquide si on le laissait libre de ses mouvements. Le ballon, par exemple, descendra jusqu'à s'arrêter en un point où il est stable. La méthode de plus forte pente est semblable, à la différence que le point qui se déplace n'a pas d'inertie (comme un ballon qui aurait une masse nulle).

Recherche d'un minimum de $f(x, y)$

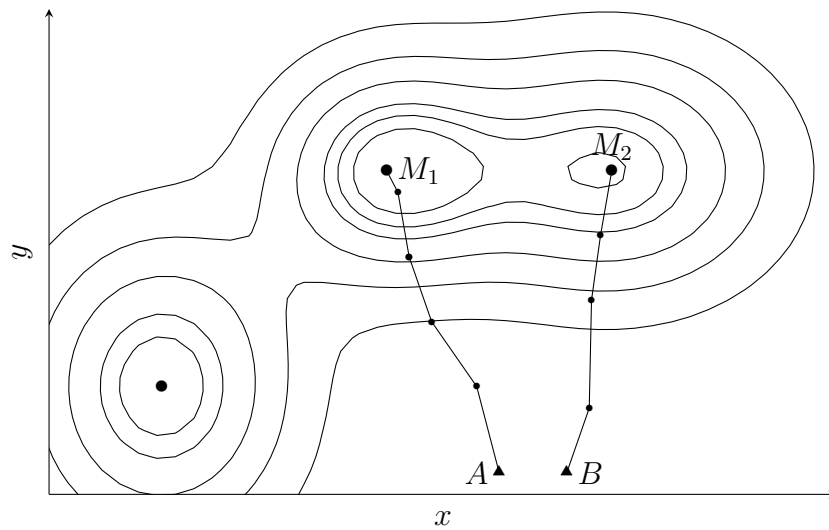


Figure 7.1: On a représenté les lignes de niveau d'une fonction f dont on cherche un minimum. Les minima locaux de f sont représentés par le symbole \bullet . Plusieurs chemins de convergence vers un minimum sont également représentés. Celui qui débute à la position A progresse vers un minimum M_1 alors que celui qui débute en B converge vers un autre minimum, M_2 . Le chemin dépend à la fois du point initial et de la méthode utilisée. En fonction de ces éléments, l'algorithme peut converger vers un minimum local ou un autre.

Construction de la méthode

Comme son nom l'indique, la méthode du gradient consiste à choisir le vecteur déplacement $u_k = -\nabla f(x_k)$. Il reste à déterminer λ_k , pour cela on estime la dérivée de $f(x_{k+1})$ par rapport à λ_k :

$$\frac{df(x_{k+1})}{d\lambda_k} = \frac{df(x_k + \lambda_k u_k)}{d\lambda_k} = u_k \cdot \nabla f(x_{k+1}) = -\nabla f(x_k) \cdot \nabla f(x_{k+1}). \quad (7.1)$$

Or si $\lambda_k = 0$ cette quantité est négative mais si $f(x)$ diverge vers $+\infty$ lorsque $|x| \rightarrow \infty$, alors elle changera de signe. Le minimum de f le long de ce déplacement sera atteint lorsque la dérivée (7.1) s'annule, il faut donc choisir λ_k de sorte que le gradient $\nabla f(x_{k+1})$ soit orthogonal à u_k . Il n'est pas toujours facile ni même possible de trouver λ_k de cette façon, mais nous allons voir maintenant que pour l'application la plus importante, la résolution d'un système linéaire, ce calcul est simple.

7.3 Résolution de systèmes linéaires

Nous avons vu à la séance 5 des méthodes pour résoudre un système linéaire, mais ces méthodes deviennent rapidement très lourdes lorsque la dimension du système augmente. L'idée d'utiliser un principe de recherche de minimum se base sur le fait que dans le cas du système linéaire, le gradient est très facile à calculer, donc toutes les méthodes basées sur des calculs de gradient, dont nous avons déjà vu la plus simple, seront efficaces. On considère donc de nouveau le système linéaire (5.1)

$$Ax = b.$$

On introduit la notion de *résidu* qui vaut

$$r = b - Ax$$

à partir duquel on construit la fonction

$$f(x) = r \cdot A^{-1}r = (Ax - b) \cdot (x - A^{-1}b). \quad (7.2)$$

Si la matrice A est *définie positive*, le minimum de f est unique et il se situe au point solution du système. Nous ne calculerons en pratique jamais cette fonction, car nous ne connaissons pas A^{-1} , nous avons seulement besoin de calculer son gradient, qui se calcule sans utiliser A^{-1} :

$$\nabla f(x) = 2(Ax - b) = -2r.$$

Pour tester la convergence, on utilisera un test $|r| > \varepsilon$ avec une norme quelconque sur \mathbf{R}^n et éventuellement un compteur.

7.3.1 Méthode du gradient

Reprenons l'explication de la méthode du gradient où nous l'avons laissée dans le cas général pour continuer dans le cas particulier de la minimisation d'une fonction f définie comme en (7.2). Nous allons maintenant pouvoir trouver la valeur de λ_k optimale grâce à l'expression du gradient. À l'étape k de l'itération, x_k est connu, donc $r_k = b - Ax_k$ aussi et nous avons

$$\boxed{u_k = -\nabla f(x_k) = 2r_k}. \quad (7.3)$$

Nous cherchons donc λ_k tel que $-u_k \cdot (-2r_{k+1}) = 0$ soit $u_k \cdot (b - A(x_k + \lambda_k u_k)) = 0$. Posons alors

$$w_k = Au_k,$$

on obtient

$$\lambda_k = \frac{u_k \cdot r_k}{u_k \cdot w_k}.$$

Nous avons donc complètement déterminé $x_{k+1} = x_k + \lambda_k u_k$.

Complexité algorithmique

Comme pour les méthodes itératives vu précédemment, nous allons seulement calculer la complexité algorithmique d'une itération. À chaque itération, nous calculons $w_k = Au_k$, ce qui représente $n^2 - n$ opérations. Un produit scalaire est de complexité algorithmique $N_{\text{prod. scal.}} = n$, la complexité de calcul pour λ_k vaut donc $2n$, et celle de x_{k+1} nécessite également $2n$ opérations. Enfin, le résidu r_{k+1} se calcule grâce à l'égalité

$$r_{k+1} = b - Ax_{k+1} = b - Ax_k - \lambda_k Au_k = r_k - \lambda_k w_k,$$

c'est-à-dire avec $2n$ opérations. En conclusion, nous retiendrons que

$$c_{\text{gradient}} \in \mathfrak{C}(n^2, 1).$$

Nous n'avons pas ajouté ici, comme nous l'avons fait pour les méthodes de Jacobi et de Gauss-Seidel, la complexité du calcul liée au test de convergence, mais comme ce test sera nécessairement de complexité $\mathfrak{C}(n)$, cela n'affecte pas la complexité de l'algorithme global.

7.3.2 Méthode du gradient conjugué

Dans la méthode du gradient conjugué, on cherche à optimiser la direction de déplacement en fonction du gradient au point de départ *et* du gradient au point d'arrivée, ce qui est possible grâce au fait que l'équation est linéaire. Cette méthode est spécifique aux systèmes linéaires.

Dans la méthode du gradient, le choix de λ_k équivaut à imposer $u_k \cdot u_{k+1} = 0$ (voir l'équation (7.1)). Cette condition est remplacée dans la méthode du gradient conjugué par la condition plus adaptée

$$\boxed{u_k \cdot Au_{k+1} = 0}. \quad (7.4)$$

La relation (7.3) n'est donc plus utilisée dans cette méthode. Il existe plusieurs solutions, dont la suivante présente l'avantage d'utiliser la même relation entre x_k et x_{k+1} que la méthode du gradient :

$$\begin{aligned} w_k &= Au_k \\ \lambda_k &= \frac{u_k \cdot r_k}{u_k \cdot w_k}, \\ x_{k+1} &= x_k + \lambda_k u_k, \\ r_{k+1} &= b - Ax_{k+1} = r_k - \lambda_k w_k, \\ \mu_k &= -\frac{r_{k+1} \cdot w_k}{u_k \cdot w_k}, \\ u_{k+1} &= r_{k+1} + \mu_k u_k. \end{aligned} \quad (7.5)$$

Il en existe beaucoup d'autres. On peut voir que la condition (7.4) est bien vérifiée. Les conditions initiales pour cet algorithme nécessitent de donner non seulement un point initial x_0 , mais également une direction initiale u_0 qui, comme on peut le voir sur l'opération (7.5), est nécessaire pour calculer w_0 et poursuivre le calcul. On peut choisir sans risque $u_0 = r_0 = b - Ax_0$.

Complexité algorithmique

Les opérations indiquées ci-dessus représentent tous les calculs de l'algorithme. Toutes les opérations sont de complexité $\mathfrak{C}(n)$ à l'exception de la première (7.5) qui est de complexité $\mathfrak{C}(n^2)$. On a donc


$$c_{\text{gradient conj.}} \in \mathfrak{c}(n^2, 1),$$

qui est exactement la même classe que celle de la méthode du gradient.

7.3.3 Méthode du double gradient conjugué

La méthode du double gradient conjugué utilise une astuce supplémentaire, celle de résoudre deux systèmes simultanément avec la méthode du gradient conjugué. Cela résulte en une efficacité accrue de la convergence. On résout simultanément les équations $Ax = b$ et ${}^tAx' = b$.

Algorithme 10 : Méthode du double gradient conjugué

```
définir la fonction doble_gradient_conjugué
• → matrice  $A$  de taille  $n \times n$ 
• → vecteur  $b$  de dimension  $n$ 
• → vecteur initial  $x$ 
• → précision  $\varepsilon$ 
   $r \leftarrow b - Ax$ 
   $r' \leftarrow r$ 
   $u \leftarrow r$ 
   $u' \leftarrow r'$ 
  tant que  $\|r\| > \varepsilon$  faire
     $w \leftarrow Au$       et  $w' \leftarrow {}^tAu'$     complexité  $\in \mathfrak{c}(n^2, 1)$ 

     $\lambda \leftarrow \frac{r \cdot r'}{w \cdot u'}$ 
     $r_1 \leftarrow r - \lambda w$    et  $r'_1 \leftarrow r' - \lambda w'$ 
     $x \leftarrow x + \lambda u$ 

     $\mu \leftarrow \frac{r_1 \cdot r'_1}{r \cdot r'}$ 
     $u \leftarrow r_1 + \mu u$    et  $u' \leftarrow r'_1 + \mu u'$ 
     $r \leftarrow r_1$        et  $r' \leftarrow r'_1$ 
  fin
• →  $x$ 
```

Toutes les opérations sont de complexité algorithmique $\mathfrak{C}(n)$ excepté le calcul de w et w' , la complexité d'une itération est donc

$$c_{\text{double gradient}} \in \mathfrak{c}(n^2, 2).$$

Conclusion

En conclusion de cette séance et de la séance 5, nous avons vu que la résolution de systèmes linéaires est un problème plus difficile qu'il ne pourrait paraître au premier abord. Dans la pratique, la résolution numérique de problèmes de physique, de mécanique ou de mathématiques font souvent appel à des résolutions de systèmes linéaires de très grande taille $n \gg 1$. (Par exemple, c'est une méthode utile pour résoudre des équations aux dérivées partielles). C'est pourquoi on se tourne souvent vers les méthodes de descente, à cause du coût très élevé des méthodes directes (Gauss, décomposition LU). Cependant, ces méthodes présentent des difficultés : elles sont sensibles au conditionnement, nécessitent que la matrice soit définie positive.

7.4 Travaux dirigés

1) **Comparer les méthodes itératives** et les méthodes de descente sur les exemples de la séance 5 page 36. Utiliser les méthodes de chronométrage et conclure.

Appendix A

Référence python

Préambule

La syntaxe de python permet de couper les lignes trop longues, mais il est recommandé, pour que le code reste lisible, d'adopter une indentation claire. Par exemple

```
def une_fonction_compliquee(x) :  
    return x+numpy.sin(x)+numpy.exp(-x)*numpy.cos(numpy.sqrt(x**2+1))-  
           x**numpy.pi
```

L'exemple suivant montre à l'inverse des pratiques à éviter (couper l'expression à un endroit qui complique sa lecture, ne pas aligner les débuts de la formule verticalement)

```
def une_fonction_compliquee(x) :  
    return  
    x+numpy.sin(x)+numpy.exp(-x)*numpy.cos(numpy.sqrt(x**2+1))-x**numpy.pi
```

A.1 Les modules

Une des particularités de **python** concerne les modules. Lorsque **python** démarre, il ne sait pratiquement rien faire. Selon les besoins, on doit lui fournir un ensemble de fonctions appelé *module*. Il existe des milliers de modules pour **python**, allant du très général au très spécifique. L'intérêt de ce système est de sélectionner uniquement les modules qui seront utilisés dans le programme, cela permet de garder de la mémoire dans l'ordinateur disponible pour notre travail.

Au début d'un programme **python** on trouve donc des directives qui donnent à **python** la liste des modules qu'il doit charger. Le module que nous utiliserons le plus s'appelle **numpy** et on peut le charger grâce à la ligne suivante

```
import numpy
```

Supposons que nous souhaitons utiliser la fonction «valeur absolue». On peut l'appeler à l'aide de la syntaxe **numpy.abs**.

On peut aussi préférer utiliser un autre nom plus court à la place du nom original :

```
import numpy as npy
```

La valeur absolue s'obtient alors à l'aide de la syntaxe `np.abs`. Cette méthode est très souvent utilisée, mais nécessite de bien garder en tête les synonymes donnés en début de programme.¹

On peut aussi préférer importer explicitement les fonctions dont on aura besoin

```
from numpy import abs, cos, exp
```

auquel cas on les obtient sans préfixe : `abs`. Les autres fonctions du module `numpy` sont en revanche inaccessibles.

A.2 Types basiques

A.2.1 Les nombres

Python connaît les nombres entiers, les nombres réels et les nombres complexes. Cependant, la représentation numérique de ces nombres a des conséquences importantes sur les calculs et la manière de les effectuer. À la fin du TD1, par exemple, il y a un exemple du rôle de la représentation des nombres réels par des «nombres décimaux à virgule flottante».

Dans ce cours, il n'y a pas à se soucier de savoir si `python` utilise un entier ou un réel dans les calculs, les conversions éventuelles se font automatiquement. Sachez toutefois que le module `numpy`, que nous utiliserons beaucoup dans tout le cours, permet de gérer différents types numériques pour les nombres entiers, réels et complexes, c'est-à-dire différentes représentations des nombres sous forme de 0 et 1, qui ont toutes des propriétés différentes.

Un nombre étrange apparaît parfois dans un résultat, il s'agit de `nan`, ce qui signifie *not a number*, autrement «résultat absurde» ou «impossible». C'est par exemple ce que l'on obtient en calculant `numpy.sqrt(-1)` ou bien en multipliant l'infini `inf` par 0. `nan` est un «nombre» particulier qui indique qu'une erreur a eu lieu dans un calcul. Tous les calculs contenant un élément `nan` ont pour résultat `nan`, de même les comparaisons avec un autre nombre sont toujours fausses, même la condition `nan==nan` est fausse ! Pour tester si un nombre vaut `nan`, il faut utiliser la fonction `numpy.isnan`.

A.2.2 Les tableaux

Le langage `python` permet de créer facilement des tableaux de nombres pouvant servir dans les fonctions vectorisées. Il faut pour cela utiliser la fonction `numpy.linspace`. Elle s'utilise en donnant les bornes du tableau et le nombre de points souhaités.

```
x=numpy.linspace(0,10,11)
```

retourne le tableau [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10].

A.2.3 Les chaînes de caractères

Pour ajouter bout-à-bout («concaténer») deux chaînes de caractères, on utilise le signe `+` :

```
s='Hello ' + 'world' + ' !'
```

est équivalent à `s='Hello world !'`.

`str` convertit un objet en chaîne de caractère, pour afficher un résultat à l'écran.

`print` permet d'afficher des objets à l'écran, quand cela est possible. On peut utiliser autant d'argument que l'on souhaite, qui seront affichés sur une même ligne.

À la fin de son exécution, `print` revient à la ligne.

¹On évitera aussi, bien sûr de donner des noms sans rapport ou trompeurs comme `import numpy as toto` ou `import numpy as images` !

A.3 Les fonctions numériques (`numpy`)

A.3.1 Fonctions d'un nombre

Ces fonctions prennent en argument un nombre (le plus souvent réel) et retournent un nombre. Elles ne sont pas intégrées par défaut dans `python`, mais sont accessibles dans le module `numpy`. Voici une liste des fonctions les plus simples de `numpy`.

<code>abs</code>	La fonction valeur absolue $x \mapsto x $.
<code>arccos</code>	La fonction arc-cosinus Arccos ou \cos^{-1} .
<code>arccosh</code>	la fonction argument-cosinus hyperbolique $x \mapsto \text{Argch}x = \ln(x + \sqrt{x^2 - 1})$.
<code>arcsin</code>	la fonction arc-sinus Arcsin ou \sin^{-1} .
<code>arcsinh</code>	la fonction argument-sinus hyperbolique. $x \mapsto \text{Argsh}x = \ln(x + \sqrt{1 + x^2})$.
<code>arctan</code>	la fonction arc-tangente, Arctan ou \tan^{-1} .
<code>arctanh</code>	la fonction argument-tangente hyperbolique $x \mapsto \text{Argth}x = \frac{1}{2} \ln \left \frac{1+x}{1-x} \right $.
<code>ceil</code>	La fonction arrondi à l'entier supérieur.
<code>cos</code>	La fonction cosinus.
<code>cosh</code>	La fonction cosinus hyperbolique ch .
<code>exp</code>	La fonction exponentielle.
<code>floor</code>	La fonction arrondi à l'entier inférieur.
<code>log</code>	La fonction logarithme népérien \ln .
<code>round</code>	La fonction arrondi à l'entier le plus proche.
<code>sin</code>	La fonction sinus.
<code>sinh</code>	La fonction sinus hyperbolique sh .
<code>sqrt</code>	La fonction racine carrée $x \mapsto \sqrt{x}$.
<code>tan</code>	La fonction tangente, $x \mapsto \tan x = \frac{\sin x}{\cos x}$.
<code>tanh</code>	la fonction tangente hyperbolique $x \mapsto \text{th} x = \frac{\text{sh} x}{\text{ch} x}$.

A.3.2 Fonctions d'un vecteur

Les fonctions suivantes prennent en argument un vecteur.

<code>max</code>	calcule le plus grand élément du vecteur.
<code>min</code>	calcule le plus petit élément du vecteur.
<code>prod</code>	calcule le produit des éléments du vecteur.
<code>sum</code>	calcule la somme des éléments du vecteur.

A.3.3 Fonctions à plusieurs résultats

Parfois on a besoin d'obtenir plusieurs résultats avec une seule fonction. Par exemple, le résultat d'un calcul et le nombre de boucles effectuées. Dans ce cas, on peut retourner un *couple*, un *triplet* et plus généralement un *n-uplet*, c'est-à-dire une structure du type

```
(3.14159, 1343, 'pi')
```

qui peut contenir autant d'objets que l'on souhaite (3, sur cet exemple, c'est donc un triplet), sans se soucier de leur type. Supposons que le triplet ci-dessus soit le résultat d'une fonction `calcule_pi`. Dans le code de cette fonction on a la commande `return` suivante :

```
def calcule_pi(precision) :
    #plein de lignes...
    return value, n_iter, name
```

On doit alors enregistrer toutes les valeurs retournées, comme ceci

```
x, n, s=calcule_pi(0.001)
```

La variable `x` vaut maintenant 3,14159, la variable `n` vaut 1343 et la chaîne `s` contient `'pi'`.

A.3.4 Algèbre linéaire

Les vecteurs (`array`)

Les vecteurs en `python` sont des tableaux qui ressemblent au type `list`. On les crée en convertissant un objet `list` avec la fonction `array`, qui crée un objet de type `numpy.ndarray`.

Les indices des vecteurs fonctionnent comme les indices de liste.

Les vecteurs possèdent de nombreuses méthodes, dont beaucoup ont une fonction associée qui donne le même résultat. Le choix entre les deux syntaxes est laissé au programmeur.

Opérations élémentaires avec les vecteurs

fonction	méthode ou descripteur	descriptif
<code>in</code>		test d'appartenance, <code>x in vect</code> est vrai si une coordonnée de <code>vect</code> vaut <code>x</code> ;
<code>not in</code>		le contraire de <code>in</code> ;
<code>max</code>	<code>v.max()</code>	le plus grand élément ;
<code>min</code>	<code>v.min()</code>	le plus petit élément ;
<code>sorted</code>		retourne une liste triée (type <code>list</code>) des éléments du vecteur ;
	<code>v.sort()</code>	trie les éléments du vecteur ;
<code>numpy.array</code>		convertit un objet en vecteur ;
<code>numpy.arange</code>		fonctionne comme <code>range</code> mais construit un vecteur avec les nombres obtenus ;
<code>numpy.dot</code>	<code>v.dot(vecteur)</code>	produit scalaire : <code>numpy.dot(u,v)</code> et <code>v.dot(u)</code> retournent le produit scalaire de <code>u</code> et <code>v</code> ;
<code>numpy.mean</code>	<code>v.mean()</code>	la moyenne des éléments ;
<code>numpy.prod</code>	<code>v.prod()</code>	le produit des éléments ;
<code>numpy.size</code>	<code>v.size</code>	nombre d'éléments ;
<code>numpy.sum</code>	<code>v.sum()</code>	la somme des éléments ;
<code>numpy.zeros(uptlet)</code>		crée un vecteur contenant uniquement des zéros dont les dimensions sont données par l'uplet, on écrira donc <code>numpy.zeros((3,4))</code> .

Les matrices (`array`)

Une matrice est un vecteur de vecteurs. Pour désigner un élément de matrice, on peut utiliser deux syntaxes équivalentes.

Les syntaxes `m[I][J]` et `m[I, J]` sont équivalentes.

Ceci est vrai que I et J soit de simples indices ou des séries d'indices de *slicing* telles que

`0:3` `:-1` ou encore `2:`

La première d'une colonne d'une matrice A est donc obtenue avec la syntaxe

`A[0, :]` ou `A[0][:]`

Certaines fonctions sur les vecteurs prennent sens seulement lorsqu'elles utilisées avec des matrices et sont donc mentionnées ici.

Opérations élémentaires avec les matrices

fonction	méthode ou descripteur	descriptif
<code>numpy.ndim</code>	<code>a.ndim</code>	le nombre de dimensions de l'objet de type <code>numpy.ndarray</code> (vaut 1 pour un vecteur, 2 pour une matrice) ;
<code>numpy.shape</code>	<code>m.shape</code>	un uplet contenant les dimensions de la matrice ;
<code>numpy.trace</code>	<code>m.trace()</code>	la trace de la matrice ;
<code>numpy.transpose</code>	<code>m.transpose()</code>	retourne la matrice transposée.

A.3.5 Calcul vectoriel

Les tableaux suivants présentent les opérations d'algèbre linéaire les plus courantes. **Attention !** certaines opérations ont des syntaxes inhabituelles et trompeuses.

Table A.2: Opérations entre un nombre a et un vecteur x dont le résultat est un vecteur y

code	Signification	formule
<code>y=x+a</code> <code>y=a+x</code>	Ajoute le nombre a à tous les éléments de x .	$y_i = x_i + a$
<code>y=x*a</code> <code>y=a*x</code>	Multiplie par a tous les éléments de y .	$y_i = ax_i$
<code>y=x-a</code> <code>y=a-x</code>	Retire a de tous les éléments de x . Soustrait de a tous les éléments de x .	$y_i = x_i - a$ $y_i = a - x_i$
<code>y=x/a</code> <code>y=a/x</code>	Divise par a tous les éléments de x (avec $a \neq 0$). Divise a par les tous éléments de x ($\forall i, x_i \neq 0$).	$y_i = x_i/a$ $y_i = a/x_i$
<code>y=x**a</code> <code>y=a**x</code>	Élève tous les éléments de x à la puissance a . Calcule a à la puissance des éléments de x .	$y_i = (x_i)^a$ $y_i = a^{x_i}$

Table A.3: Opérations entre deux vecteurs x et y dont le résultat est un vecteur z

code	Signification	formule
<code>z=x+y</code>	Somme des vecteurs $z = x + y$.	$z_i = x_i + y_i$
<code>z=x-y</code>	Différence des vecteurs $z = x - y$.	$z_i = x_i - y_i$
<code>z=x*y</code>	Produit <i>terme à terme</i> des éléments de vecteurs (pas de notation mathématique).	$z_i = x_i y_i$
<code>z=x/y</code>	Quotient terme à terme des éléments des vecteurs (pas de notation mathématique).	$z_i = x_i / y_i$
<code>z=x**y</code>	Exponentiation terme à terme des éléments des vecteurs (pas de notation mathématique).	$z_i = (x_i)^{y_i}$

Table A.4: Opérations entre un vecteur x et une matrice A de taille $n \times p$

code	Signification	formule
<code>y=A.dot(x)</code> <code>y=numpy.dot(a,x)</code>	Calcule le produit à droite (produit habituel) $y = Ax$. La taille de x doit être égale au nombre de colonnes de A . La taille de y est égale au nombre de lignes de A .	$y_i = \sum_{j=1}^p m_{ij} x_j$
<code>B=A*x</code> <code>B=x*A</code>	Calcule la matrice B dont chaque colonne est multipliée par l'élément correspondant de x . La taille de x doit être égale au nombre de colonnes de A .	$b_{ij} = a_{ij} x_j$
<code>y=x.dot(A)</code> <code>y=numpy.dot(x,A)</code>	Calcule le produit à gauche $y = {}^t x A$. La taille de x doit être égale au nombre de lignes de A .	$y_j = \sum_{i=1}^n x_i a_{ij}$

A.4 Les fonctions graphiques (`matplotlib`)

La bibliothèque qui permet de faire des graphiques avec `python` que nous utiliserons s'appelle `matplotlib.pyplot`. Pour simplifier on lui donne souvent un alias. Dans ce cours, j'utiliserai `mplot`, ce qui se déclare de la façon suivante

```
import matplotlib.pyplot as mplot
```

en début de fichier. À moins que les graphiques ne soient pas destinés à une visualisation immédiate, il vaut mieux activer le mode «interactif» avec la commande

```
mplot.ion()
```

A.4.1 Tracer une courbe (`plot`)

Cette fonction permet de tracer une ou plusieurs courbes dans une fenêtre graphique. Elle s'utilise avec deux tableaux *de même taille*, le premier contient les valeurs des abscisses et le second celui des ordonnées. Par exemple :

```
mplot.plot(x,y, label='y(x)')
```

Pour tracer plusieurs courbes, il suffit d'ajouter une ligne avec la même fonction. `python` trace automatiquement les courbes sur la même figure.

Pour modifier le tracé d'une courbe, on peut ajouter après les vecteurs d'abscisses et d'ordonnées une chaîne de caractères qui modifie le tracé de la paire précédente. Voici quelques exemples, la liste complète se trouve dans l'aide disponible avec la commande `help(mplot.plot)`.

Styles de points		Couleurs		Styles de ligne	
'x'	croix	'b'	bleu foncé	'-'	ligne continue
'o'	ronds	'c'	bleu clair	'--'	tirets
'*'	étoiles	'g'	vert clair	':'	pointillés
's'	carrés	'k'	noir	'.-'	points et tirets
'd'	losanges	'm'	magenta		
'^', '>', 'v', '<'	triangles	'r'	rouge		
'p'	pentagones	'w'	blanc		
'h'	hexagones				

Ainsi pour tracer une courbe en pointillés magenta avec des étoiles on écrit

```
mplot.plot(x,y, 'm*:')
```

Il est également possible de modifier la largeur du tracé, mais pour cela, la syntaxe est différente. Par exemple `mplot.plot(x,y,linewidth=2)` trace les lignes avec une largeur double.

Note

Les styles de ligne peuvent aussi être donnés sous la forme plus explicite : `color='red', marker='x', linestyle=':'`.

- `semilogx` fait la même chose que `plot` avec l'axe des abscisses en échelle logarithmique.
- `semilogy` Fait la même chose que `plot` avec l'axe des ordonnées en échelle logarithmique.
- `loglog` Fait la même chose que `plot` avec les deux axes en échelle logarithmique.

A.4.2 Légendes, titre etc.

- title** Cette commande permet de donner un titre à la figure de la figure courante.
- xlabel** Cette commande permet de nommer l'axe des abscisses de la figure courante.
- ylabel** Cette commande permet de nommer l'axe des ordonnées de la figure courante.
- legend** Cette commande permet de nommer les différentes courbes de la figure courante. Les noms des courbes doivent être entrés dans le même ordre que les données dans la commande **plot** (ou son équivalent).

A.4.3 Enregistrer une figure

La commande **savefig** permet d'enregistrer la dernière figure (la figure *courante*) dans un fichier image. Il existe de nombreux formats disponibles, le format est choisi automatiquement en fonction de l'extension donnée. Par exemple

```
matplotlib.savefig('nom_du_fichier.png')
```

enregistre la figure courante au format PNG.

```
savefig(fname, dpi=None, facecolor='w', edgecolor='w',
        orientation='portrait', papertype=None, format=None,
        transparent=False, bbox_inches=None, pad_inches=0.1,
        frameon=None, metadata=None)
```

A.4.4 Exemple complet

Voici un exemple complet de tracé d'une fonction simple.

```
# Importation des fonctions dont nous aurons besoin
from numpy import linspace, sin
import matplotlib.pyplot as plt

# Matplotlib en mode interactif
plt.ion()

# Définition des paramètres pour le tracé
xmin=0
xmax=10
n_points=100

# Le vecteur des abscisses
x=linspace(xmin, xmax, n_points)

# Le vecteur des ordonnées
y=sin(x)

# Définition de la taille de la figure
plt.figure(figsize=(13,8))
```

```
# Définition de la taille des caractères
matplotlib.rcParams.update({'font.size': 22})

# Placement de la légende dans le coin en bas à gauche
matplotlib.rcParams.update({'legend.loc': 'lower left'})

# On trace la courbe
matplotlib.pyplot.plot(x, y, label='sinus')

# On ajoute les légendes des axes
matplotlib.pyplot.legend()
matplotlib.pyplot.xlabel(r'$x$')          # Le 'r' devant la chaîne indique qu'on
matplotlib.pyplot.ylabel(r'$y=\sin(x)$') # utilise LaTeX pour écrire les légendes.
```

A.5 Autres fonctions

A.5.1 Mesure du temps

Pour chronométrer le temps mis par une opération, le module `time` est très simple à utiliser. L'instruction `time.perf_counter()` permet d'obtenir l'horaire d'une horloge interne à `python` exprimé en secondes sous forme d'un nombre réel. On pourra donc l'utiliser pour enregistrer l'horaire avant et après une opération et calculer la différence.

Appendix B

Référence MATLAB/octave

Ce document contient des tableaux résumant les éléments du langage `octave`/MATLAB.

Préambule

Il est utile de se souvenir que si une ligne est trop longue, on peut ajouter trois points en fin de ligne puis continuer sur la ligne suivante. Exemple

```
x=1+sqrt(2)+pi+sqrt(pi)+exp(0.5)+pi^2/6...  
-exp(-1)-sqrt(5)+0.57721566490153;
```

B.1 Types basiques

B.1.1 Les nombres

`octave` connaît les nombres entiers, les nombres réels et les nombres complexes. Cependant, la représentation numérique de ces nombres a des conséquences importantes sur les calculs et la manière de les effectuer. À la fin du TD1, par exemple, il y a un exemple du rôle de la représentation des nombres réels par des «nombres décimaux à virgule flottante».

Dans ce cours, il n'y a pas à se soucier de savoir si `octave` utilise un entier ou un réel dans les calculs, les conversions éventuelles se font automatiquement.

Un nombre étrange apparaît parfois dans un résultat, il s'agit de `NaN`, ce qui signifie *Not a number*, autrement «résultat absurde» ou «impossible». C'est par exemple ce que l'on obtient en calculant $0/0$ ou bien en multipliant l'infini `Inf` par 0. `NaN` donc est un «nombre» particulier qui indique qu'une erreur a eu lieu dans un calcul. Tous les calculs contenant un élément `NaN` ont pour résultat `NaN`, de même les comparaisons avec un autre nombre sont toujours fausses, même la condition `NaN==NaN` est fausse ! Pour tester si un nombre vaut `NaN`, il faut utiliser la fonction `isnan`.

B.1.2 Les vecteurs

Les vecteurs sont des objets naturels en `octave`. Pour créer un vecteur contenant les nombres entiers de 1 à 10, le constructeur `:` est très efficace :

```
x=1:10;
```

`x` contient alors le vecteur `[1 2 3 4 5 6 7 8 9 10]`. On peut également ajouter un incrément et utiliser n'importe quelles bornes. Le vecteur créé contient tous les points jusqu'à la borne supérieure incluse. Par exemple `0:0.5:3` donne le vecteur `[0 0.5 1 1.5 2 2.5 3]` et `0:0.6:2` le vecteur `[0 0.6 1.2 1.8]`.

B.1.3 Les chaînes de caractères

Une chaîne de caractères est dans **octave** un vecteur ligne de caractères. Ainsi pour ajouter bout à bout («concaténer») deux chaînes de caractères, il suffit d'utiliser la même syntaxe qu'avec des vecteurs de nombres :

```
s=['Hello ' 'world' ' !'];
```

est équivalent à `s='Hello world !';`.

Conversion d'un nombre

La fonction **num2str** convertit un nombre en chaîne de caractère, pour afficher un résultat à l'écran.

Affichage

Les fonctions **disp** et **display** permettent d'afficher une chaîne de caractères. **display** ajoute une ligne vide après la ligne affichée.

B.2 Les fonctions

B.2.1 Fonctions d'un nombre

Ces fonction prennent en argument un nombre et retournent un nombre.

abs	la fonction valeur absolue $ x $
acos	la fonction arc-cosinus Arccos ou \cos^{-1}
acosh	la fonction argument-cosinus hyperbolique $\text{Argch}x = \ln(x + \sqrt{x^2 - 1})$
asin	la fonction arc-sinus Arcsin ou \sin^{-1}
asinh	la fonction argument-sinus hyperbolique $\text{Argsh}x = \ln(x + \sqrt{1 + x^2})$
atan	la fonction arc-tangente, Arctan ou \tan^{-1}
atanh	la fonction argument-tangente hyperbolique $\text{Argth}x = \frac{1}{2} \ln \left \frac{1+x}{1-x} \right $
ceil	la fonction arrondi à l'entier supérieur
cos	la fonction cosinus
cosh	la fonction cosinus hyperbolique ch
exp	la fonction exponentielle
factorial	la factorielle (uniquement pour les entiers)
floor	la fonction arrondi à l'entier inférieur
log	la fonction logarithme népérien \ln
round	la fonction arrondi à l'entier le plus proche
sin	la fonction sinus
sinh	la fonction sinus hyperbolique sh
sqrt	la fonction racine carrée
tan	la fonction tangente, $\tan x = \frac{\sin x}{\cos x}$
tanh	la fonction tangente hyperbolique $\text{th}x = \frac{\text{sh}x}{\text{ch}x}$

B.2.2 Les fonctions d'un vecteur

Les fonctions suivantes prennent en argument un vecteur.

max calcule le plus grand élément du vecteur.
Avec la syntaxe `[m,i]=max(v)`; on obtient le maximum et son indice dans `v`.

min calcule le plus petit élément du vecteur.
Avec la syntaxe `[m,i]=min(v)`; on obtient le minimum et son indice dans `v`.

prod calcule le produit des éléments du vecteur.

sum calcule la somme des éléments du vecteur.

B.2.3 Fonctions à plusieurs résultats

Parfois on a besoin d'obtenir plusieurs résultats avec une seule fonction. Par exemple, le résultat d'un calcul et le nombre de boucles effectuées. Dans ce cas, on peut retourner un vecteur-ligne

```
[3.14159, 1343, 'pi']
```

qui peut contenir autant d'objets que l'on souhaite, sans se soucier de leur type. Supposons que le vecteur ci-dessus soit le résultat d'une fonction `calcule_pi`. La définition de cette fonction se fait avec la ligne suivante

```
function [value, n_iter, name]=calcule_pi(precision)
```

On peut alors enregistrer les valeurs retournées, comme ceci

```
[x, n, s]=calcule_pi(0.001);
```

La variable `x` vaut maintenant 3,14159, la variable `n` vaut 1343 et la chaîne `s` contient `'pi'`.

En `octave` on peut ne récupérer que les premiers éléments du vecteur. Ainsi on peut simplement écrire

```
x=calcule_pi(0.00001);
```

et seule la valeur du premier élément sera récupérée. **Les valeurs des autres éléments sont définitivement perdues.**

B.3 Les fonctions graphiques

B.3.1 Tracer une courbe (`plot`)

La commande `plot` permet de tracer une ou plusieurs courbes dans une fenêtre graphique. Elle s'utilise avec deux vecteurs *de même taille*, le premier contient les valeurs des abscisses et le second celui des ordonnées.

```
plot(x,y)
```

Pour tracer plusieurs courbes, il suffit d'ajouter une paire d'arguments d'abscisses et d'ordonnées. Les vecteurs d'une paire doivent avoir la même taille, mais deux paires peuvent être formées avec des vecteurs de tailles différentes.

Styles de points		Couleurs		Styles de ligne	
'x'	croix	'b'	bleu foncé	'--'	ligne continue
'o'	ronds	'c'	bleu clair	'--'	tirets
'*'	étoiles	'g'	vert clair	':'	pointillés
's'	carrés	'k'	noir	'.-'	points et tirets
'd'	losanges	'm'	magenta		
'^', '>', 'v', '<'	triangles	'r'	rouge		
'p'	pentagones	'w'	blanc		
'h'	hexagones				

```
plot(x,y,x,y2,x3,y3)
```

Dans l'exemple ci-dessus, on trace trois courbes. Les vecteurs **x**, **y** et **y2** ont la même taille mais **x3** et **y3** peuvent avoir une taille plus grande ou plus petite. On a utilisé le vecteur **x** deux fois pour les abscisses.

Pour modifier le tracé d'une courbe, on peut ajouter après les vecteurs d'abscisses et d'ordonnées une chaîne de caractères qui modifie le tracé de la paire précédente. Voici quelques exemples, la liste complète se trouve dans l'aide disponible avec la commande **help plot**.

Ainsi pour tracer une courbe en pointillés magenta avec des étoiles on écrit

```
plot(x,y,':*m')
```

Il est également possible de modifier la largeur du tracé, mais pour cela, la syntaxe est différente. Par exemple

```
plot(x,y,'linewidth',2)
```

trace les lignes avec une largeur double.

- semilogx** Fait la même chose que **plot** mais avec l'axe des abscisses en échelle logarithmique.
- semilogy** Fait la même chose que **plot** mais avec l'axe des ordonnées en échelle logarithmique.
- loglog** Fait la même chose que **plot** mais avec les deux axes en échelle logarithmique.

B.3.2 Légendes, titre, etc.

- title** Cette commande permet de donner un titre à la figure de la figure courante.
- xlabel** Cette commande permet de nommer l'axe des abscisses de la figure courante.
- ylabel** Cette commande permet de nommer l'axe des ordonnées de la figure courante.
- legend** Cette commande permet de nommer les différentes courbes de la figure courante. Les noms des courbes doivent être entrés dans le même ordre que les données dans la commande **plot** (ou son équivalent).
- set(gca,...)** Le mot-clé **gca** désigne les axes de la figure courante. Avec la fonction **set**, on peut modifier les paramètres des axes, par exemple on peut élargir le tracé avec **set(gca,'linewidth',2)**.

B.3.3 Sauvegarder une figure (`print`)

La commande `print` permet d'enregistrer la dernière figure (la figure *courante*) dans un fichier image. Il existe de nombreux formats disponibles. Pour créer un fichier au format PNG, la syntaxe est

```
print -dpng 'nom_du_fichier.png'
```

B.3.4 Exemple complet

Voici un exemple complet de tracé d'une fonction simple.

```
% Définition des paramètres pour le tracé
tmin=0;
tmax=10;
n_points=100;

% Calcul du pas entre deux points
pas=(tmax-tmin)/(n_points-1);

% Initialisation des vecteurs à la bonne taille
t=tmin:pas:tmax;    % On utilise le constructeur :
y=sin(t);           % La fonction sin est vectorisée.

% On trace la courbe
% 'linewidth',2 permet de grossir la dernière courbe tracée.
plot(t,y,'linewidth',2);

% On agrandit la taille des caractères avec 'fontwidth' et
% la largeur des traits avec linewidth
set(gca,'fontsize',16,'linewidth',2);

% On ajoute les légendes des axes
xlabel('x');
ylabel('y=sin(x)');
```

B.4 Algèbre linéaire

L'algèbre linéaire est le cœur de MATLAB/octave les opérateurs algébriques en octave sont très condensés, mais recèlent des subtilités.

Attention ! Les opérations d'algèbre linéaire En MATLAB/octave, il y a une distinction entre les vecteurs horizontaux et vecticaux. Les opérations entre eux peuvent donner des résultats inattendus si on ne maîtrise pas bien les opérateurs.

Table B.2: Opérations entre un nombre a et un vecteur x dont le résultat est un vecteur y

code	signification	formule
$y=x+a$ $y=a+x$	Ajoute le nombre a à tous les éléments de x .	$y_i = x_i + a$
$y=x*a$ $y=a*x$	Multiplie par a tous les éléments de y .	$y_i = ax_i$
$y=x-a$ $y=a-x$	Retire a de tous les éléments de x . Soustrait de a tous les éléments de x .	$y_i = x_i - a$ $y_i = a - x_i$
$y=x/a$ $y=a./x$	Divise par a tous les éléments de x (avec $a \neq 0$). Divise a par les tous éléments de x ($\forall i, x_i \neq 0$).	$y_i = x_i/a$ $y_i = a/x_i$
$y=x.^a$ $y=a.^x$	Élève tous les éléments de x à la puissance a . Calcule a à la puissance des éléments de x .	$y_i = (x_i)^a$ $y_i = a^{x_i}$

Table B.3: Opérations entre deux vecteurs x et y ayant la même orientation dont le résultat est un vecteur z

code	signification	formule
$z=x+y$	Somme des vecteurs $z = x + y$.	$z_i = x_i + y_i$
$z=x-y$	Différence des vecteurs $z = x - y$.	$z_i = x_i - y_i$
$z=x.*y$	Produit terme à terme des éléments de vecteurs (pas de notation mathématique).	$z_i = x_i y_i$
$z=x./y$	Quotient terme à terme des éléments des vecteurs (pas de notation mathématique).	$z_i = x_i / y_i$
$z=x.^y$	Exponentiation terme à terme des éléments des vecteurs (pas de notation mathématique).	$z_i = (x_i)^{y_i}$

Table B.4: Opérations entre un vecteur horizontal h et un vecteur vertical v .

code	signification	formule
$A=v+h$ $A=h+v$	La matrice des sommes.	$a_{ij} = v_i + h_j$
$A=v-h$ $B=h-v$	La matrice des différences.	$a_{ij} = -b_{ij} = v_i - h_j$
$A=v.*h$ $A=h.*v$	La matrice des produits.	$a_{ij} = v_i h_j$
$A=v./h$ $B=h./v$	La matrice des rapports.	$a_{ij} = 1/b_{ij} = v_i / h_j$
$A=v.^h$ $B=h.^v$	La matrice des puissances.	$a_{ij} = (v_i)^{h_j}$ $b_{ij} = (h_j)^{v_i}$
$a=h*v$	Calcule le produit scalaire de h et v . Les deux vecteurs doivent être de même taille.	$a = h \cdot v = \sum_{i=1}^n h_i v_i$
$A=v*h$	Calcule la matrice de Gram $A = v^t h$.	$a_{ij} = v_i h_j$

Table B.5: Opérations entre un vecteur vertical v et une matrice A

code	signification	formule
$y=A*v$	Calcule le produit à droite (produit habituel) $y = Av$. La taille de v doit être égale au nombre de colonnes de A . La taille de y est égale au nombre de lignes de A .	$y_i = \sum_{j=1}^p a_{ij}v_j$
$y=v \backslash A$	Division à droite (habituelle). La matrice doit être inversible et avoir la même dimension que v .	$y = A^{-1}v$

Table B.6: Opérations entre un vecteur horizontal h et une matrice A

code	signification	formule
$y=h*A$	Calcule le produit à gauche $y = hA$. La taille de h doit être égale au nombre de lignes de A . La taille de y est égale au nombre de colonnes de A .	$y_j = \sum_{i=1}^n h_i m_{ij}$
$y=h/M$	Division à gauche. La matrice doit être inversible et avoir la même dimension que h .	$y = hA^{-1}$

B.5 Autres fonctions

B.5.1 Mesure du temps (`tic` et `toc`)

Les fonctions `tic` et `toc` permettent de mesurer le temps écoulé. `tic` déclenche un chronomètre et `toc` note le temps écoulé. On peut utiliser plusieurs `toc` après un `tic`. Si on souhaite enregistrer le temps dans une variable `temps`, il suffit d'écrire `temps=``toc`.

Mémento python

Mémento python	1
1 Premiers pas	2
1.1 Écrire une instruction	2
1.2 Les objets	2
1.3 ★ Les descripteurs	3
1.4 Les méthodes	3
1.5 Les modules, la directive import	3
1.6 Les erreurs	4
1.7 Les commentaires	5
1.8 À l'aide !	5
2 Les types de python	5
2.1 Les nombres entiers (int)	5
2.2 Les booléens (bool)	6
2.3 Les nombres réels (float)	7
2.4 ★ Les nombres complexes (complex)	7
2.5 Les chaînes de caractères (str)	7
2.6 Les listes (list)	8
2.7 Les dictionnaires (dict)	10
2.8 Les multiplats (tuple)	11
2.9 ★ Les ensembles (set)	12
2.10 ★ Généralités sur les classes	13
3 Les structures	13
3.1 Les structures conditionnelles (if , elif et else)	14
3.2 Les boucles (for et while)	14
3.3 Les fonctions (def , return et lambda)	16
3.4 Arguments nommés et arguments optionnels	17
3.5 Fonctions retournant plusieurs résultats	17
3.6 ★ Les espaces de noms	17
3.7 ★ La gestion des erreurs (try , except , else et finally)	19
4 Les fonctions numériques (numpy)	20
4.1 Les tableaux multidimensionnels (numpy.ndarray)	20
4.2 Fonctions mathématiques de numpy	20
5 Les fonctions graphiques (matplotlib.pyplot)	21
5.1 Tracer une courbe (matplotlib.pyplot.plot)	21
5.2 Légendes, titre etc.	22
5.3 Enregistrer une figure	22
5.4 Exemple complet	23

Le symbole ★ indique des concepts avancés

1 Premiers pas

Un langage informatique est un langage destiné à être compris par un ordinateur d'une part et par des humains d'autre part, d'où parfois certaines difficultés. Heureusement **python** fait partie des langages modernes, qui présentent les concepts plutôt du côté humain. Dans le langage **python**, il existe différents *types*, c'est-à-dire différentes organisations de données qui incarnent des concepts proches de ce que nous comprenons en tant qu'humain (côté ordinateur, ce sont des suites de chiffres plus ou moins complexes, mais on ne s'en occupera pas). Par exemple, les concepts de *nombre entier* ou de *nombre réel* existent en **python**. Nous en verrons d'autres.

Il est possible de demander à l'ordinateur de faire une place dans sa mémoire pour y enregistrer un exemplaire de données organisées au format d'un type. Par exemple, on peut lui demander de conserver en mémoire un nombre entier ou un nombre réel. Il gardera alors ces données en mémorisant leur type et donnant un nom à la zone de mémoire concernée. On appelle *objet* cette zone de mémoire, et il est utile de la considérer de la même façon que concevons son contenu dans notre cerveau, c'est-à-dire comme un nombre, une fonction (comme en mathématiques), un son, une image...

1.1 Écrire une instruction

En **python**, chaque ligne décrit une instruction. **python** exécute les lignes les unes après les autres, dans l'ordre, en respectant les structures éventuelles (voir section 3). Le premier caractère d'une ligne ne peut pas être un espace ou une tabulation. Une instruction peut s'étendre sur plusieurs lignes en ajoutant le caractère `\` à la toute fin de chaque ligne (pas d'espace après) sauf la dernière.

1.2 Les objets

Pour créer un objet, on le définit avec le signe `=` (que l'on représente en pseudo-code de manière plus fidèle à son action : \leftarrow) comme ceci ¹

```
objet = expression
```

Les noms en italique ne sont pas des éléments de code, mais le nom que l'on donne à l'élément. Un nom d'objet doit commencer par une *lettre* ou par le caractère `_` et peut contenir des lettres, des chiffres et le caractère `_`. On peut utiliser des caractères accentués. L'expression est une écriture codée des données représentées. Chaque type a une écriture codée distincte. Par exemple on peut créer un objet nommé **var** représentant le nombre entier 137 avec

```
var = 137
```

Il suffit alors d'utiliser le nom **var** pour que l'ordinateur utilise les données contenues en mémoire pour effectuer des opérations. Ces données sont la *valeur* ou le *contenu* de l'objet. Il sera possible de changer le contenu dans la case mémoire en lui attribuant de nouvelles données. L'ancien contenu sera effacé et perdu. Comme le contenu peut changer on dit que **var** est une *variable*.

Pour afficher le contenu de la variable **var**, on utilise la commande **print** de la façon suivante :

```
print(var)
```

et **python** affiche alors le contenu ²

```
137
```

Les différentes catégories d'objets sont appelées *types* ou *classes*. Pour connaître le type d'un objet, on utilise la commande **type**.

¹Les commandes sont écrites sur fond jaune pâle.

²Les résultats d'une commande sont affichés sur fond gris.

```
type(var)
```

```
<class 'int'>
```

affiche de quelle catégorie fait partie le contenu de `var` (ici `int`, qui signifie nombre entier, voir la section 2.1).

L'instruction `del` permet de libérer de la place en mémoire en effaçant le contenu d'une variable. Après l'exécution de

```
del(var)
```

le contenu de `var` est perdu et la variable n'existe plus.

1.3 ★ Les descripteurs

Un descripteur est une valeur fixe attachée à un objet :

```
var.descripteur
```

Cette valeur ne peut pas être modifiée autrement qu'en modifiant le contenu de la variable.

1.4 Les méthodes

Pour certains type d'objet, il existe des *méthodes*, c'est-à-dire des fonctions qui sont attachées à la valeur de l'objet et que l'on appelle avec la syntaxe

```
var.méthode()
```

ou bien

```
var.méthode(arguments)
```

L'action d'une méthode dépend du contenu de `var`. Une méthode peut retourner un ou plusieurs objets ou aucun. Si le contenu de la variable `var` est modifiable, une méthode peut le modifier.

1.5 Les modules, la directive `import`

Une des particularités de `python` concerne les modules. Lorsque `python` démarre, il ne sait pratiquement rien faire. Selon les besoins, on doit lui fournir un ensemble de fonctions appelé *module*. Il existe des milliers de modules pour `python`, allant du très général au très spécifique. L'intérêt de ce système est de sélectionner uniquement les modules qui seront utilisés dans le programme, cela permet de garder de la mémoire dans l'ordinateur disponible pour notre travail.

Au début d'un programme `python` on trouve donc des directives qui donnent à `python` la liste des modules qu'il doit charger. Le module que nous utiliserons le plus s'appelle `numpy` et on peut le charger grâce à la ligne suivante

```
import numpy
```

Supposons que nous souhaitons utiliser la fonction «cosinus». On peut l'appeler à l'aide de la syntaxe `numpy.cos`.

On peut aussi préférer utiliser un autre nom plus court à la place du nom original :

```
import numpy as npy
```

Le cosinus s'obtient alors à l'aide de la syntaxe `npy.cos`. Cette méthode est très souvent utilisée, mais nécessite de bien garder en tête les synonymes donnés en début de programme.³

On peut aussi préférer importer explicitement les fonctions dont on aura besoin

³On évitera aussi, bien sûr de donner des noms sans rapport ou trompeurs comme `import numpy as toto` ou `import numpy as images` !

```
from numpy import abs, cos, exp
```

auquel cas on les obtient sans préfixe : `cos`. Les autres fonctions du module `numpy` sont en revanche inaccessibles.

1.6 Les erreurs

Lorsque `python` atteint la fin d'un programme, si aucune erreur n'a été rencontrée, il n'affiche aucun message de lui-même, ce qui signifie que l'exécution a eu lieu avec succès.

Or, lorsqu'une instruction cause une erreur, contrairement à ce que l'on pourrait penser, `python` atteint aussi la fin du programme et c'est uniquement une fois la fin atteinte qu'un message d'erreur s'affiche. Avant cela, et après que l'erreur soit apparue, l'exécution continue ligne à ligne, *mais aucune instruction n'est exécutée* : `python` sort de toutes les structures (conditionnelles, boucles, fonctions, voir section 3) jusqu'à atteindre la fin du programme. C'est un peu comme un chemin d'évacuation en cas d'incendie. Tout au long du chemin, `python` garde en mémoire les points de sortie des fonctions, les numéros de ligne et les noms de fichiers (on appelle cela le mécanisme de *traceback*).

Une fois arrivé à la fin du programme, `python` affiche un message décrivant l'erreur et donnant les éléments contextes mémorisés.

La lecture des messages d'erreur se fait de bas en haut.

Dans l'exemple de message d'erreur ci-dessous, l'erreur `ZeroDivisionError` (division par zéro) a été provoquée par la commande `print(1/0)` à la ligne 15 du fichier `romberg.py` dans la définition de la fonction `rombug`. Cette fonction a été appelée depuis la définition de la fonction `f` à la ligne 21 du fichier `bidule.py` par la commande `y=rombug(x,2)`. La fonction `f` a elle-même été appelée depuis `"<stdin>"` c'est-à-dire la ligne de commande interactive, par la commande `y=f(1)`. Il est très important de savoir lire les messages d'erreur pour les corriger efficacement.

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    y=f(1)
  File "bidule.py", line 21, in f
    y=rombug(x,2)
  File "romberg.py", line 15, in rombug
    print(1/0)
ZeroDivisionError: division by zero
```

Les erreurs courantes

<code>AssertionError</code>	l'assertion demandée n'est pas vérifiée, voir page 7 ;
<code>ImportError</code>	le module ou la fonction recherchée n'a pas été trouvée ;
<code>IndentationError</code>	la délimitation des structures par indentation comporte une erreur ;
<code>IndexError</code>	l'indice demandé n'existe pas pour l'objet ;
<code>KeyError</code>	la clé recherchée n'existe pas ;
<code>MemoryError</code>	la mémoire est pleine ;
<code>NameError</code>	la variable demandée n'a pas été trouvée ;
<code>SyntaxError</code>	erreur de syntaxe ;
<code>TypeError</code>	une opération impossible a été demandée ;
<code>ValueError</code>	la valeur donnée à une fonction est inexploitable ;
<code>ZeroDivisionError</code>	une division par zéro a été demandée.

Note

Si on écrit `2+2=4`, cela fait apparaître le message d'erreur suivant

```
SyntaxError: can't assign to operator
```

c'est-à-dire que nous avons demandé à ce que le nombre 4 soit enregistré dans une case mémoire appelée `2+2`. Le message d'erreur nous dit que l'on ne peut pas assigner une valeur à «`2+2`» car cette expression contient une opération (+) et n'est pas un nom de variable valide. De même si on essaie avec `4=2+2`, le message d'erreur est

```
SyntaxError: can't assign to literal
```

qui nous dit que le nombre 4 est « littéral » c'est-à-dire que sa valeur est la même chose que la façon de l'écrire. On ne peut pas donner de valeur au nombre 4 (même si elle vaut 4 aussi).

★ *Simuler une erreur*

Pour effectuer des tests ou gérer le fonctionnement d'un programme, il est possible de simuler une erreur à l'aide de l'instruction `raise` (le message est facultatif)

```
raise erreur('message')
```

par exemple :

```
raise ZeroDivisionError("Diviser par zéro, c'est pas beau")
```

1.7 Les commentaires

L'ajout de commentaires dans un code est très important puisqu'il permet de garder des informations sur la conception du programme, d'expliquer pourquoi certains choix ont été faits, de donner les sources de la méthode utilisée. Un commentaire commence par le caractère `#`. Tout ce qui se trouve à la droite de ce caractère sur la même ligne constitue le commentaire. Une seule exception : lorsque le caractère `#` se trouve dans une chaîne de caractères, alors il ne débute pas un commentaire. Voici un exemple de commentaire :

```
z=1.570796326794896 # pi/2
```

1.8 À l'aide !

Pour obtenir des informations sur un objet (une fonction, une variable, etc.) un type, un module ou une erreur nous avons à notre disposition la commande `help`

```
help(nom)
```

2 Les types de python

2.1 Les nombres entiers (`int`)

Pour créer un nombre entier, il suffit de l'écrire

```
m = 42
```

Il est également possible d'exprimer le nombre avec un calcul :

```
n = 4*5-12/4
```

le contenu de la variable `n` est alors

```
print(n)
```

17

Les opérations suivantes sont possibles avec les nombres entiers:

+	addition
-	soustraction
*	multiplication
**	puissance (<code>pow(x,y)</code> est synonyme de <code>x**y</code>)
//	division entière
%	reste de la division entière
/	division, le résultat peut être un nombre réel
== ⁴	test d'égalité
!=	test de différence
>	strictement supérieur
>=	supérieur (au sens large)
<, <=	strictement inférieur, inférieur
int	convertit n'importe quel objet en entier

Pour la division entière, on peut aussi utiliser `divmod` dont le résultat est un doublet (voir 2.8) `divmod(x,y)` vaut `(x//y, x%y)`.

Note —

Le caractère `^` décrit aussi une opération entre nombres entiers, mais qui n'est pas une opération algébrique !

2.2 Les booléens (`bool`)

Les nombres booléens⁵ sont des valeurs logiques. Il n'y a que deux valeurs possibles : *vrai* (`True`) et *faux* (`False`). Les opérateurs de comparaison des entiers nous fournissent des exemples

```
print(2**4==4**2)
```

True

car $2^4 = 4^2$ donc l'égalité est vérifiée,

```
print(2**3==3**2)
```

False

car $2^3 \neq 3^2$ donc l'égalité n'est pas vérifiée.

Opérations des booléens

and	et logique
not	négation (<code>not False</code> vaut <code>True</code>)
or	ou logique
bool	convertit n'importe quel objet en booléen

⁴Attention, la comparaison s'écrit avec deux signes égal `==`.

⁵On prononce «boulé-ain».

★ La commande `assert`

Il est possible à l'aide d'un test de causer une erreur si le test est faux. Par exemple dans le code suivant

```
1 assert x != 0
2 y=1/x
```

si `x` vaut zéro, python cause une erreur `AssertionError` et ne cherche pas à calculer `1/x` à la ligne 2. Il est possible d'ajouter un message :

```
assert x !=0, "x est égal à zéro"
```

2.3 Les nombres réels (`float`)

Tout comme les nombres entiers, les nombres réels se définissent par leur écriture, cependant on peut utiliser la notation scientifique en ajoutant la lettre `e` suivant de l'exposant. Le séparateur décimal est le point. Pour définir la variable `x` comme valant $1,38 \times 10^{-23}$ on écrit donc

```
x=1.38e-23
```

Toutes les opérations pour les entiers s'appliquent aussi aux nombres réels.

Pour la division entière `x//y` est la partie entière du résultat de `x/y`.

Des nombres étranges apparaissent parfois dans un résultat, il s'agit de `inf` et `nan`. Le nombre `inf` est l'infini ∞ , certaines opérations sont autorisées avec lui, comme par exemple `inf*2` qui donne `inf` bien sûr ! Quant à `nan`, cela signifie *not a number*, autrement «résultat absurde» ou «impossible». C'est par exemple ce que l'on obtient en calculant $\sqrt{-1}$ ou en multipliant `inf` par 0. `nan` est un «nombre» particulier qui indique qu'une erreur a eu lieu dans un calcul. Tous les calculs contenant un élément `nan` ont pour résultat `nan`, de même les comparaisons avec un autre nombre sont toujours fausses, même la condition `nan==nan` est fausse ! Pour tester si un nombre vaut `nan`, il faut utiliser la fonction `numpy.isnan` (voir la section 4).

La fonction `float` permet de convertir un nombre en nombre réel.

2.4 ★ Les nombres complexes (`complex`)

Il existe aussi des nombres complexes (`complex`), dont l'écriture utilise la lettre `j` pour l'unité imaginaire `i`. La partie réelle et la partie imaginaire doivent toutes les deux être exprimées sous forme d'un nombre réel valide. Ainsi, le nombre $-\frac{1}{2} + i$ s'écrit `-0.5+1j`. Comme il doit toujours y avoir un chiffre avant le `j`, cela n'empêche pas de nommer une variable `j`.

Les opérations et les méthodes des nombres complexes sont les mêmes que pour les entiers et les réels.

Descripteurs des nombres complexes

```
z.imag la partie imaginaire de z ;  
z.real la partie réelle de z.
```

2.5 Les chaînes de caractères (`str`)

Pour ajouter bout-à-bout («concaténer») deux chaînes de caractères, on utilise le signe `+`:

```
s='Hello ' + 'world' + ' !'
```

est équivalent à `s='Hello world !'`.

Opérations élémentaires avec des chaînes de caractères

- `+` concaténation des chaînes de caractères ;
- `len` la longueur d'une chaîne (son nombre de caractères)
- `str` convertit un objet en chaîne de caractère, pour afficher un résultat à l'écran.

Méthodes de chaînes de caractères

- | | |
|---------------------------------------|---|
| <code>s.find(s2)</code> | cherche la chaîne <code>s2</code> dans <code>s</code> et retourne la position en nombre de caractères depuis le début de <code>s</code> . Si <code>s2</code> ne se trouve pas dans <code>s</code> , la position retournée est <code>-1</code> ; |
| <code>s.format(variables)</code> | permet d'afficher des valeurs de variables dans une chaîne de caractère. Par exemple :
<code>'Valeur={}'.format(17)</code> vaut <code>'Valeur=17'</code> ; |
| <code>s.index(s2)</code> | comme <code>find</code> mais cause une erreur si <code>s2</code> ne se trouve pas dans <code>s</code> ; |
| <code>s.join(liste de chaînes)</code> | construit une chaîne de caractères en mettant toutes les chaînes de la liste avec <code>s</code> entre elles. Par exemple :
<code>'!@#'.join(['A', 'B', 'C'])</code> vaut <code>'A!@#B!@#C'</code> ; |
| <code>s.lower()</code> | construit une nouvelle chaîne dans laquelle toutes les lettres de <code>s</code> sont en minuscule. |
| <code>s.split()</code> | découpe la chaîne <code>s</code> aux endroits avec des espaces et renvoie la liste des éléments ; |
| <code>s.upper()</code> | construit une nouvelle chaîne dans laquelle toutes les lettres de <code>s</code> sont en majuscule. |

2.6 Les listes (`list`)

Nous abordons maintenant les objets ordonnés, qui sont des objets qui contiennent d'autres objets de façon organisée. Il existe trois sortes d'objets ordonnés, les listes, les dictionnaires et les multiplets.

Les listes contiennent des objets numérotés. Les objets sont numérotés de 0 à $n - 1$.

Les éléments d'une liste peuvent être de n'importe quel type, on peut utiliser des types différents dans une même liste.

Pour créer une liste, on utilise les syntaxes suivantes

```
liste1=[12,-7,32,8,0,5]
liste2=list(1, 'a', 0.23, True, [1,2,3])
```

Le nombre d'éléments de la liste s'obtient avec la fonction `len` :

```
print(len(liste1))
```

6

Si on veut utiliser la valeur d'un seul élément, on utilise les crochets en indiquant l'indice désiré (Attention le premier élément porte l'indice 0) :

```
print(liste1[2])
```

```
32
```

Contrairement aux nombres entiers, réels, complexes, booléens et aux chaînes de caractères, **une liste est un objet modifiable**. On peut par exemple effacer un élément avec `del` ⁶

```
del liste1[2]
print(liste1)
```

```
[12, -7, 8, 0, 5]
```

ou bien ajouter un élément :

```
liste1.append(13)
print(liste1)
```

```
[12, -7, 8, 0, 5, 13]
```

La liste à laquelle on a ajouté l'objet 13 est *la même* que celle de départ, mais avec des modifications. Un peu comme un livre sur lequel on peut écrire, dont on peut arracher une page, *etc.*

«Assembler» deux listes — ce que l'on appelle la *concaténation* — se fait avec le signe `+`

```
print(liste1+liste2)
```

```
[12, -7, 8, 0, 5, 13, 1, 'a', 0.23, True, [1,2,3] ]
```

Il est également possible de trier une liste si ses éléments sont comparables deux à deux, avec la méthode `sort` :

```
liste1.sort()
print(liste1)
```

```
[-7, 0, 5, 8, 12, 13]
```

(on a changé l'ordre des pages du livre, mais c'est toujours le même livre).

Le « slicing », la gestion des indices

Plus généralement on peut extraire des sous-listes avec de différentes façons, on appelle toutes ces façons du «tranchage» (*slicing* en anglais).

```
liste1[2:5]
```

retourne une *nouvelle liste* qui contient les éléments de `liste1` du numéro 2 (c'est-à-dire le troisième) **inclu** au numéro 5 (le sixième) **exclu** :

```
[5, 8, 12]
```

Les indices autorisés vont de $-n$ à $n - 1$. L'indice 0 désigne le premier élément, l'indice -1 désigne le dernier et $-n$ le premier. Il faut que le premier indice désigne un élément classé avant celui que désigne le deuxième indice. Ainsi `liste1[-4:5]` est la même chose que `liste1[2:5]` et `liste1[-1:3]` est une liste vide.

```
liste1[3:]
```

retourne une *nouvelle liste* qui contient les éléments de `liste1` du numéro 3 au dernier :

```
[8, 12, 13]
```

⁶ Attention, dans ce cas, les indices des éléments changent et `liste1[2]` vaut maintenant 8.

```
liste1[:2]
```

retourne une *nouvelle liste* constituée des éléments de `liste1` du premier au numéro 2 **exclu** :

```
[-7, 0]
```

La syntaxe `liste1[:]` retourne donc une *nouvelle liste* qui est une copie de `liste1`.

Une autre fonctionnalité de tranchage très utile permet de prendre des séries d'indices espacés régulièrement avec un pas au choix avec la syntaxe `liste1[début (inclus):fin (exclu):pas]`.

```
liste1[1::2]
```

permet d'obtenir la sous-liste des éléments d'indice impair.

```
[0, 8, 13]
```

Il est possible d'utiliser des pas négatifs, ainsi `liste1=liste1[::-1]` est équivalent à `liste1.reverse()`.

Opérations élémentaires avec les listes

- +** concaténation (assemblage) de listes ;
- all** teste si tous les éléments sont **True** (ou bien différents de zéro) ;
- any** teste si au moins un élément est **True** (ou bien différent de zéro) ;
- in** test d'appartenance : `x in liste` est vrai si `x` est un élément de la liste et faux sinon ;
- not in** le contraire de **in** ;
- len** le nombre d'éléments de la liste ;
- sorted** une copie triée de la liste ;
- list** convertit un objet en liste.

Méthodes de listes

- 1. **append**(*objet*) ajoute un objet à la fin de la liste ;
- 1. **copy**() retourne une copie de la liste (comme `l[:]`)
- 1. **count**(*objet*) compte le nombre d'éléments de la liste égaux à l'objet ;
- 1. **index**(*objet*) si l'objet est présent dans la liste, donne l'indice auquel il se trouve, sinon cause une erreur ;
- 1. **remove**(*objet*) si l'objet est présent dans la liste, retire le premier élément égal à cet objet, sinon cause une erreur ;
- 1. **reverse**() inverse l'ordre de la liste ;
- 1. **sort**() trie la liste.

2.7 Les dictionnaires (**dict**)

Dans un dictionnaire, on associe une *clé* à une *valeur*. Une clé joue le rôle de l'indice dans une liste, elle sert à localiser un objet. Chaque clé d'un dictionnaire est *unique*. La valeur associée à une clé peut être n'importe quel objet.

Un dictionnaire est un ensemble modifiable de correspondances entre une clé et une valeur.

Une clé doit nécessairement être un objet **non modifiable**. Cela peut donc être un nombre (entier, réel, complexe) ou une chaîne de caractère, mais pas une liste ou un dictionnaire.

Pour créer un dictionnaire, on utilise la syntaxe :

```
dict1={'a':1, 2:'x'}
```

cet exemple crée un dictionnaire avec deux clés : 'a' et 2 et deux valeurs : 1 associée à 'a' et 'x' associée à 2.

Les associations d'un dictionnaire ne peuvent pas être triées, c'est pourquoi il n'y a pas de méthode de tri. Si l'on fait une boucle sur un dictionnaire, l'ordre dans lequel les éléments seront utilisés n'est pas prévisible.

Opérations élémentaires avec les dictionnaires

in test d'appartenance : **x in dict1** est vrai si **x** est une clé du dictionnaire **dict1** et faux sinon ;
not in le contraire de **in** ;
len le nombre d'associations dans le dictionnaire ;
dict convertit un objet en dictionnaire.

Méthodes de dictionnaires

d.copy() retourne une copie du dictionnaire ;
d.items() retourne la liste des doublets (clé, valeur) ;
d.keys() retourne la liste des clés ;
d.update(d2) pour chaque paire clé-valeur du dictionnaire **d2**, si la clé est présente dans le dictionnaire **d**, la valeur associée est modifiée pour celle de **d2**, sinon la paire clé-valeur est ajoutée à **d**. Par exemple :
 {'a':1, 2:'x'}.update({'a':0, 'b':2}) vaut
 {'a':0, 2:'x', 'b':2}
d.values() retourne la liste des valeurs ;

2.8 Les multiplets (tuple)

Un doublet est une paire d'objets, un triplet est formé de trois objets, etc. Le néologisme *multiplet* désigne la généralisation à n'importe quel nombre d'élément de ce type de structure dans laquelle on a plusieurs objets qui n'en forment qu'un seul. À la différence d'une liste et d'un dictionnaire, un multiplet est non-modifiable si tous les objets qu'ils contient le sont. Dans ce cas, ils sont des clés de dictionnaires valides.

Pour récupérer tous les éléments d'un multiplet, il suffit d'affecter un autre multiplet en indiquant des noms de variables pour chaque élément. Voici un exemple :

```
t1=(1,2,3,5,7)
(a,b,c,d,e)=t1
```

Après exécution de ces commandes, nous avons créé cinq variables **a**, **b**, **c**, **d** et **e** et **a** vaut 1, **b** vaut 2, **c** vaut 3, **d** vaut 5 et **e** vaut 7. On peut même échanger les valeurs de deux variables :

```
a,b=b,a
```

échange les objets contenus dans **a** et **b**.

Pour récupérer un seul élément d'un multiplet, on peut utiliser des indices :

Les indices d'un multiplet fonctionnent comme les indices de liste.

Opérations élémentaires avec des multiplats

in test d'appartenance : **x in u1** est vrai si **x** appartient au multiplat **u1** et faux sinon ;
not in le contraire de **in** ;
len le nombre d'éléments du multiplat ;
tuple convertit un objet en multiplat.

Méthodes de multiplats

t.count(objet) compte le nombre d'éléments du multiplat égaux à l'objet ;
t.index(objet) si l'objet est présent dans **t**, donne l'indice auquel il se trouve, sinon cause une erreur.

2.9 ★ Les ensembles (set)

Les ensembles sont des dictionnaires dans lesquels il n'y a que des *clés* (pas de valeur associée). Les fonctions et les méthodes qui font intervenir les valeurs d'un dictionnaire (**values**, **update**...) n'existent donc pas pour les ensembles. Pour créer un ensemble, on utilise la même syntaxe que pour un dictionnaire sans indiquer de valeurs :

```
s1={2,4,8,16}
```

Attention, la commande **s={}** crée un dictionnaire vide. Pour créer un ensemble vide, il faut utiliser **set()**.

Opérations élémentaires avec des ensembles

in test d'appartenance : **x in s1** est vrai si **x** appartient à l'ensemble **s1** et faux sinon ;
not in le contraire de **in** ;
len le nombre d'éléments de l'ensemble ;
set convertit un objet en ensemble.

Méthodes d'ensembles

s.add(objet) ajoute un objet à l'ensemble **s** ;
s.clear() vide l'ensemble **s**, équivalent à **s=set()** ;
s.copy() crée une copie de l'ensemble ;
s.discard(objet) retire l'objet de l'ensemble, si l'objet n'est pas dans l'ensemble, rien ne se passe ;
s1.intersection(s2) retourne un ensemble formé des éléments présents dans **s1** et **s2** à la fois ;
s1.intersection_update(s2) équivaut à **s1=s1.intersection(s2)** ;
s1.isdisjoint(s2) retourne **True** si **s1** et **s2** n'ont aucun élément en commun ;
s1.issubset(s2) retourne **True** si tous les éléments de **s1** sont aussi des éléments de **s2** ;
s1.issuperset(s2) même chose que **s2.issubset(s1)** ;
s.remove(objet) retire l'objet de l'ensemble, cause une erreur si l'objet n'est pas dans l'ensemble ;
s1.symmetric_difference(s2) crée un ensemble formé des éléments se trouvant soit dans **s1**, soit dans **s2** mais pas dans les deux ;


```
s1.symmetric_difference_update(s2)
                                équivaut à s1=s1.symmetric_difference(s2) ;
s1.union(s2)                    retourne un ensemble formé des éléments présents dans s1 ou dans s2 ;
s1.update(s2)                   équivaut à s1=s1.union(s2) ;
```

2.10 ★ Généralités sur les classes

Les classes sont le nom que donne **python** aux types. Il en existe de nombreuses. Une classe contient la définition de l'ensemble de propriétés, opérations, méthodes *etc.* Tout objet défini en **python** possède des caractéristiques qui font de lui un membre d'une classe. Un objet correspond toujours à une zone de mémoire qui peut avoir un nom et dont *toutes les caractéristiques sont définies*. On dit alors que c'est une *instance*.

Un objet est une instance d'une classe.

Pour obtenir des informations sur une classe, on peut exécuter `help(classe)`, par exemple `help(list)`, ce qui fournit des informations complètes sur toutes les caractéristiques de cette classe.

3 Les structures

Les types de **python** font partie de la structure interne du langage. Lorsque l'on écrit du pseudo-code, on ne se soucie pas de parler de types. En revanche, dans le pseudo-code, on organise les différentes étapes et décision selon un nombre de structures relativement restreint : les conditions, les boucles et les fonctions. En **python**, la structure apparaît à l'aide d'un décalage des lignes qui se trouvent à l'intérieur que l'on appelle *indentation*. Il est obligatoire d'ajouter des espaces en début de ligne pour indiquer à quel niveau de la structure on se situe.

À l'intérieur d'une structure, l'indentation doit être la même au début de chaque ligne.

Toutes les structures commencent par **if**, **while**, **for** ou **def** ⁷. À la fin de l'instruction de début de structure on doit ajouter deux points `:`, ce qui indique que l'on ouvre la structure. Ensuite toutes les lignes appartenant à la structure doivent être décalées par rapport au début de la ligne d'ouverture d'au moins un espace. Il est recommandé d'en ajouter au moins deux, voire quatre. Par exemple (les espaces sont représentés par `□`) :

```
def bidule(A):                # ouverture d'une définition de fonction
    □□for i in range(10) :    # ouverture d'une boucle
        □□□instructions      # instructions exécutées pour chaque i
        □□□if i==0 :         # ouverture d'une structure conditionnelle
            □□□□instructions  # instructions exécutées si i=0
        □□□else :            # délimitation interne de la structure conditionnelle
            □□□□instructions  # instructions exécutées si i≠0
        □□□instructions      # instructions exécutées après la structure conditionnelle
                                # pour chaque i
    □□instructions           # instructions de la définition de la fonction
                                # exécutées après la boucle
instructions                  # ces instructions ne sont pas dans la définition de la fonction
```

⁷Ou encore **try**, voir section 3.7 ★.

3.1 Les structures conditionnelles (`if`, `elif` et `else`)

Une condition fonctionne sur un principe très simple : le mot-clé `if` est suivi d'expression logique dont le résultat est un booléen, par exemple une comparaison (égalité ou inégalité). On entre alors dans une *structure conditionnelle*.

- **Si la condition est vérifiée**, python exécute les lignes décalées immédiatement suivantes jusqu'à rencontrer une ligne sans indentation par rapport à `if` ou la fin du fichier. Si cette ligne commence par l'un des mots-clés `else` ou `elif`, python saute toutes les lignes décalées qui se trouvent après, sinon l'exécution reprend.
- **Si la condition n'est pas vérifiée**, python saute toutes les lignes décalées suivantes. S'il y a une ligne non décalée ensuite, le travail de python dépend du premier mot-clé de la ligne.
 - s'il s'agit de `else` (suivi de `:`) python exécute les lignes décalées suivantes jusqu'à rencontrer une ligne sans décalage.
 - s'il s'agit de `elif`, python teste la condition qui suit ce mot (`elif` est la contraction de `else if`) et ensuite tout se passe comme après une condition `if` selon le résultat du test de la condition.
 - dans tous les autres cas, alors la structure conditionnelle est terminée.

Il peut y avoir autant de `elif` que l'on veut. En revanche il ne peut y avoir qu'un seul `else` et il ne faut aucun `elif` après.

Exemple : les commandes suivantes affichent le signe du nombre `x` (préalablement défini).

```
if x==0 :
    print(0)  # si x vaut 0 on affiche 0
    # à partir d'ici, x ne vaut jamais 0
elif x>0 :  # 2e condition testée
    print(1) # si x est positif on affiche 1
    # à partir d'ici, x ne vaut jamais 0 et n'est jamais positif
else :      # annonce ce que l'on fait si toutes les conditions sont fausses.
    print(-1)
```

3.2 Les boucles (`for` et `while`)

Les boucles sont des structures répétitives qui permettent d'exécuter plusieurs fois la même portion de programme. Si l'on connaît à l'avance le nombre de fois qu'il faut exécuter cette portion de code, on utilise une boucle `for`, dans le cas contraire on utilise `while` avec une condition.

Note —

Les instructions `break` et `continue` permettent de briser les structures en forçant la sortie ou la poursuite de la boucle. Leur usage est *fortement déconseillé* car il obscurcit la lecture du code.

Boucle `for` et itérateurs

Supposons pour commencer que l'on dispose d'une liste de nombres `x` et que l'on souhaite créer la liste `y` qui contient les nombres de `x` multipliés par 2,

```
y=x.copy() # réserve la place en mémoire pour une liste de même taille que x
for i in range(len(x)) :
    y[i]=x[i]*2
```

La première ligne sert à créer une liste `y` de la même taille que `x`. On pourrait aussi utiliser (entre autres) `y=[0]*len(x)`.

L'instruction `range` permet de parcourir des nombres entiers :

- `range(10)` crée la liste des entiers de 0 à 9 (c'est-à-dire jusqu'à 10 non inclu, comme pour les indices de tranchage de listes).
- `range(1,6)` crée la liste des entiers de 1 à 5.
- `range(0,10,2)` crée la liste des entiers pairs de 0 à 8 (elle compte de 2 en 2).

L'instruction `range` ne crée pas une liste, mais un *itérateur* qui est une sorte de compteur. Grâce à ce mécanisme, si l'on veut faire une boucle de 10 milliards de tour, il n'est pas nécessaire de stocker une liste de 10 milliards d'éléments en mémoire.

Pour renverser l'ordre des éléments d'un itérateur, il existe l'instruction `reversed`, par exemple

```
for i in reversed(range(100)) :
```

parcoure les entiers de 99 à 0.

Une liste n'est pas un itérateur, mais on peut l'utiliser comme si c'en était un (on dit qu'une liste est *itérable*). Voici comment afficher successivement tous les éléments d'une liste :

```
for x in [0.1, 0.4, 4.692, 2e-7] :
    print(x)
```

Ici l'indice n'est pas explicitement utilisé, seule la valeur de l'élément est créée dans la boucle.

Pour utiliser l'indice *et* la valeur dans une boucle, il existe l'instruction `enumerate` :

```
for i, x in enumerate(liste1) :
    print("L'objet {} est l'élément d'indice {} dans la liste.".format(x,i))
```

L'instruction `print` fait appel à `x` et à `i`. Dans la boucle, `x` est un synonyme de `liste1[i]`. Cela fonctionne de la même façon avec des multiplets et des dictionnaires.

Boucle `while`

Une boucle `while` est utile lorsque l'on ne connaît pas à l'avance le nombre de répétitions à effectuer. Elle est initiée par une ligne contenant `while` suivi d'une condition. Lorsque `python` rencontre une instruction `while` la première fois, c'est comme une instruction `if` sauf qu'il n'y a ni `else` ni `elif`.

- Si la condition n'est pas vérifiée, `python` passe directement à la fin de la structure en sautant toutes les lignes décalées.
- Si la condition est vérifiée, `python` exécute toutes les lignes jusqu'à la fin de la structure mais, à la différence de `if`, il revient à la ligne `while` ensuite.

Exemple

Calculons le nombre d'or φ à une précision de $\varepsilon = 10^{-8}$ comme limite de la suite (u) définie par

$$\begin{cases} u_0 = 1, \\ u_{n+1} = 1 + \frac{1}{u_n}, \end{cases} \quad \varphi = \lim_{n \rightarrow \infty} u_n.$$

Il est *a priori* très difficile de dire combien de termes il faut calculer, car ce nombre dépend de u_0 et ε de façon complexe. La méthode donnée ici consiste à calculer les premiers termes de la suite u jusqu'à ce que la précision demandée soit atteinte.

```
u,v=1,2    # valeurs de  $u_n$  et  $u_{n-1}$ 
epsilon=1e-8 # on définit la précision  $\varepsilon = 10^{-8}$ 
while abs(u-v) > epsilon :
    v=u      # (n augmente de 1) maintenant u et v valent  $u_{n-1}$ 
    u=1+1/u  # on calcule la nouvelle valeur de  $u_n$ 
print("Le nombre d'or vaut {} à la précision {}".format(u,epsilon))
```

3.3 Les fonctions (def, return et lambda)

Définir une fonction est une possibilité essentielle dans un projet en **python** car c'est grâce à des fonctions que l'on peut construire des opérations plus complexes. La définition d'une fonction se fait grâce au mot-clé **def**. Si la fonction renvoie un résultat celui-ci est défini par ce qui suit le mot-clé **return**. Une fonction peut ne retourner aucun résultat, dans ce cas **return** est inutile.

```
def nom_de_la_fonction (arguments) :
    """description de la fonction"""
    instructions
    return résultat # facultatif
```

Dès que le mot-clé **return** est rencontré, le résultat est renvoyé et l'exécution de la fonction est terminée. Tout ce qui suit sera ignoré. La description de la fonction est un commentaire particulier puisqu'il est enregistré et c'est ce qui s'affiche lorsque l'on utilise la commande **help** avec le nom de la fonction créée.

Fonctions simples : lambda

Il existe une syntaxe courte pour définir une fonction dont le résultat est une simple expression sans test ni boucle, qui utilise le mot clé **lambda**, en hommage à la théorie du λ -calcul des années 1970. La syntaxe est la suivante

```
lambda arguments : expression
```

dans laquelle les arguments doivent être séparés par des virgules. L'expression retournée par la fonction peut être de n'importe quel type. L'intérêt de cette écriture est de ne pas avoir à donner un nom à une fonction qui n'est utilisée qu'une seule fois, alors qu'avec **def** il est nécessaire de nommer la fonction. Ainsi, plutôt que d'avoir à définir la fonction **carre** dans l'exemple suivant

```
def carre(x) :
    return x*x
print(integrale(carre, 0, 1, 1000))
```

on obtiendra le même résultat avec la commande

```
print(integrale(lambda x : x*x, 0, 1, 1000))
```

Il est toutefois possible de nommer la fonction définie par **lambda**.

```
carre= lambda x : x*x
```

cette définition donne *exactement le même résultat* que la définition

```
def carre(x) :
    return x*x
```

3.4 Arguments nommés et arguments optionnels

Le noms des arguments donnés dans une définition avec `def` ou `lambda` peuvent être utilisés pour spécifier les arguments dans un ordre différent. Ainsi une fonction définie avec `def f(x,y)` peut être exécutée avec l'instruction `f(y=0,x=1)` qui est équivalente à l'instruction `f(1,0)`.

Ce mécanisme permet de définir des arguments optionnels, c'est-à-dire des arguments dont la valeur n'a pas besoin d'être spécifiée à chaque appel de la fonction. Dans ce cas, les arguments en question ont une valeur par défaut qui est utilisée si aucune valeur n'est donnée. Voici un tableau d'exemples avec la fonction

```
def f(x=1, y=2) :
    return x*x+y
```

expression	résultat	explication
<code>f()</code>	3	on utilise les valeurs par défaut <code>x=1</code> et <code>y=2</code> : $1 \times 1 + 2 = 3$;
<code>f(y=0)</code>	1	la valeur par défaut <code>x=1</code> est utilisée et <code>y</code> vaut 0 : $1 \times 1 + 0 = 1$;
<code>f(2,1)</code>	5	les arguments sont affectés dans l'ordre, soit <code>x=2</code> et <code>y=1</code> : $2 \times 2 + 1 = 5$;
<code>f(y=3,x=5)</code>	28	l'ordre des arguments nommés n'a pas d'importance : $5 \times 5 + 3 = 28$;
<code>f(0)</code>	2	s'il manque des arguments, ils sont pris dans l'ordre : $0 \times 0 + 2 = 2$.

3.5 Fonctions retournant plusieurs résultats

Parfois on a besoin d'obtenir plusieurs résultats avec une seule fonction. Par exemple, le résultat d'un calcul et le nombre de boucles effectuées. Dans ce cas, on peut retourner un multiplet, comme par exemple `(3.14159, 1343, 'pi')`. Il peut donc y avoir autant de résultats qu'on le souhaite (trois sur cet exemple), sans se soucier de leur type. Supposons que le triplet ci-dessus soit le résultat d'une fonction `calcule_pi`. Dans le code de cette fonction on a la commande `return` suivante : `return value, n_iter, name`. On doit alors enregistrer toutes les valeurs retournées :

```
x, n, s=calcule_pi(0.001)
```

La variable `x` vaut maintenant 3,14159, la variable `n` vaut 1343 et la chaîne `s` contient `'pi'`.

3.6 ★ Les espaces de noms

Un espace de noms est une *zone de la mémoire* dans laquelle `python` cherche les variables qui sont appelées dans le code. Par exemple un module importé par `import module`, le module est un espace de noms dans lequel on ne cherche une variable que si cela est explicitement demandé par l'appel `module.fonction`. En revanche avec la directive `from module import fonction`, la fonction est ajoutée à l'espace de noms global de `python` et est accessible sans préciser où chercher. On peut également renommer un unique objet avec `from module import objet as nouveau_nom`. L'espace de noms global est celui dans lequel commence toute session `python`, il est vide au lancement de `python`⁸.

Lors de l'appel d'une fonction, un espace de noms, appelée mémoire locale, est créé, et il ne contient *aucune variable*. Avant d'exécuter les instructions de la fonction, `python` crée une variable dans la mémoire locale pour chaque argument passé à la fonction, en lui donnant le nom qui apparaît dans la définition. Lorsque l'on appelle une variable au cours de l'exécution de la fonction, `python` commence par la chercher dans la mémoire locale. S'il ne trouve pas de variable portant le nom demandé, `python` cherche alors dans la mémoire *globale*.

Exemple : supposons que nous avons défini une fonction `f` comme ceci

⁸Il existe un troisième niveau de mémoire, la mémoire implémentée, qui contient les définitions des types d'objet, des fonctions de base et des mots-clés comme `def`. Ce niveau ne dépend que de la version de `python` utilisée, il est impossible de le modifier.

```
f-1 def f(x, y) :  
f-2     """Fonction inutile"""  
f-3     x=x+2  
f-4     return x*a+y
```

Nous allons maintenant décrire ce qui se passe si l'on lance les commandes suivantes

```
1 x=6  
2 a=2  
3 print(f(3,7))  
4 print(x)  
5 print(y)  
6 a=4  
7 print(f(y=x, x=2))
```

Les actions effectuées par `python` sont décrites ligne par ligne, les numéros de lignes sont repris dans le descriptif.

1. Définition de la variable `x` dans la mémoire globale.
2. Définition de la variable `a` dans la mémoire globale.
3. Appel de la fonction `f` :
 - f-1. `(x, y)=(3, 7)` crée les variables `x` et `y` dans la mémoire *locale* avec pour valeurs 3 et 7 respectivement,
 - f-3. on ajoute 2 à la variable `x` *locale* qui vaut donc maintenant $3 + 2 = 5$,
 - f-4. comme `a` n'est pas définie dans la mémoire locale, `python` utilise la valeur de la mémoire globale, soit 2. Le résultat $5 \times 2 + 7 = 17$ est renvoyé par `return` au niveau global.
`python` quitte la fonction car le mot-clé `return` a été rencontré :
la mémoire locale est effacée entièrement.

`python` reprend l'exécution au niveau global :

- l'expression `f(3,7)` est remplacée par 17,
 - `python` exécute `print(17)` et affiche 17.
4. Affichage de la valeur de `x` dans la mémoire globale : ce nombre n'a pas changé, il vaut encore 6.
 5. La variable `y` n'existe pas dans la mémoire globale, la commande renvoie donc une erreur :

```
NameError: name 'y' is not defined
```

6. Changement de la valeur de la variable `a` dans la mémoire globale,
7. Appel de la fonction `f` :
 - f-1. `(x, y)=(2, x)` crée les variables `x` et `y` dans la mémoire *locale* avec 2 pour valeur de `x` et la variable `y` prend la valeur de `x` de la mémoire *globale* (soit 6) car `x` n'existe pas encore dans la mémoire locale au moment de cette affectation,
 - f-3. on ajoute 2 à `x` dans la mémoire *locale* qui vaut donc maintenant $2 + 2 = 4$,
 - f-4. comme `a` n'est pas définie dans la mémoire locale, `python` utilise la valeur de la mémoire globale, soit 4. Le résultat $4 \times 4 + 6 = 22$ est renvoyé par `return` au niveau global.
`python` quitte la fonction car le mot-clé `return` a été rencontré :
la mémoire locale est effacée entièrement.

python reprend l'exécution au niveau global :

- l'expression `f(y=x, x=2)` est remplacée par 22,
- python exécute `print(22)` et affiche 22.

Une représentation graphique des différents appels à la mémoire se trouve page 24.

3.7 ★ La gestion des erreurs (`try`, `except`, `else` et `finally`)

Le structure de contrôle d'erreur permet de sécuriser une portion de code pour réagir aux erreurs et arrêter l'évacuation (à une sorte de point de rassemblement intermédiaire). Elle se présente de la façon suivante

```
try:
    instructions      # code pouvant causer des erreurs
except erreur :      # balise d'interception d'une erreur
    instructions      # code exécuté si cette erreur se produit
else:
    instructions      # à mettre après toutes les clauses except
finally:
    instructions      # à mettre après else
    instructions      # code exécuté quoiqu'il arrive
```

Il peut y avoir plusieurs clauses `except` pour différentes erreurs, mais au plus une clause `else` et au plus une clause `finally`. Étudions l'exemple suivant :

```
1 try:
2     x= -1
3     assert x>=0, "il faut que x soit positif"
4     print(1/x)
5 except ZeroDivisionError:
6     print("Hmmm, on dirait que quelqu'un veut diviser par zéro")
7 except (AssertionError, NameError) as erreur:
8     print("Message d'erreur : '{}'.format(erreur)")
9 else:
10    print("Tout va bien")
11 finally:
12    print("Au revoir !")
```

Son exécution affichera

```
Message d'erreur : 'il faut que x soit positif'
Au revoir !
```

car l'assertion de la ligne 3 n'est pas vérifiée et python passe alors à la ligne 7, exécute la ligne 8 puis va à ligne 11 et exécute la ligne 12.

Si l'on remplace la ligne 2 par

```
2 x=0
```

on obtient alors

```
Hmmm, on dirait que quelqu'un veut diviser par zéro
Au revoir !
```

Car cette fois l'assertion est vérifiée, mais on calcule $1/0$. Continuons avec la ligne 2 suivante


```
2 y=0
```

on obtient alors un message d'erreur indiquant que `x` n'est pas défini

```
Message d'erreur : 'name 'x' is not defined'
Au revoir !
```

Enfin avec la ligne 2 suivante

```
2 x=1
```

on obtient

```
1.0
Tout va bien
Au revoir !
```

4 Les fonctions numériques (`numpy`)

Le module `numpy` permet de manipuler des tableaux de nombres réels ou complexes, multidimensionnels, de type `numpy.ndarray`, qui peuvent représenter des vecteurs, des matrices ou des tenseurs, et de faire du calcul numérique.

4.1 Les tableaux multidimensionnels (`numpy.ndarray`)

Constructeurs de tableaux multidimensionnels

<code>numpy.array(liste)</code>	construit un tableau à partir d'une liste ;
<code>numpy.eye(n)</code>	matrice identité de taille $n \times n$;
<code>numpy.linspace(minimum, maximum, nombre)</code>	vecteur de nombres espacés régulièrement, <code>numpy.linspace(0,10,11)</code> donne le tableau <code>array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])</code> ;
<code>numpy.zeros(dimensions)</code>	tableau rempli de 0, les dimensions sont données sous forme d'un entier ou d'un multiplet d'entiers. Il existe aussi <code>numpy.ones</code> .

Descripteurs de tableaux multidimensionnels

<code>tableau.ndim</code>	nombre de dimensions du tableau ;
<code>tableau.shape</code>	multiplet contenant les dimensions du tableau ;
<code>tableau.T</code>	tableau transposé (pour les matrices).

Fonctions pour les tableaux multidimensionnels

<code>numpy.max</code>	calcule le plus grand élément du tableau ⁹
<code>numpy.min</code>	calcule le plus petit élément du tableau ⁹ ;
<code>numpy.prod</code>	calcule le produit des éléments du tableau ;
<code>numpy.sum</code>	calcule la somme des éléments du tableau ⁹ .

4.2 Fonctions mathématiques de `numpy`

Ces fonctions prennent en entrée un nombre (réel ou complexe) ou un tableau `numpy.ndarray` et retournent un objet du même type et de mêmes dimensions que l'entrée. Voici une liste des fonctions les plus simples du module `numpy`.

⁹Attention, `python` possède une fonction du même nom qui ne donne pas le même résultat pour les listes et les tableaux multidimensionnels.

<code>numpy.abs</code>	La fonction valeur absolue ⁹ $x \mapsto x $;
<code>numpy.arctan</code>	la fonction arc-tangente, Arctan ou \tan^{-1} ;
<code>numpy.ceil</code>	la fonction arrondi à l'entier supérieur ;
<code>numpy.cos</code>	la fonction cosinus ;
<code>numpy.exp</code>	la fonction exponentielle ;
<code>numpy.floor</code>	la fonction arrondi à l'entier inférieur ;
<code>numpy.log</code>	la fonction logarithme népérien \ln ;
<code>numpy.round</code>	la fonction arrondi à l'entier le plus proche ;
<code>numpy.sin</code>	la fonction sinus ;
<code>numpy.sqrt</code>	la fonction racine carrée $x \mapsto \sqrt{x}$;
<code>numpy.tan</code>	la fonction tangente, $x \mapsto \tan x = \frac{\sin x}{\cos x}$.

Opérations d'algèbre linéaire

<code>numpy.dot(t1,t2)</code>	produit scalaire ;
<code>numpy.matmul(m1,m2)</code>	produit de matrices.

5 Les fonctions graphiques (`matplotlib.pyplot`)

La bibliothèque qui permet de faire des graphiques avec `python` que nous utiliserons s'appelle `matplotlib.pyplot`. Pour simplifier on lui donne souvent un alias. Dans ce document, on utilise `mplot`, ce qui se déclare de la façon suivante

```
import matplotlib.pyplot as mplot
```

en début de fichier. À moins que les graphiques ne soient pas destinés à une visualisation immédiate, il vaut mieux activer le mode «interactif» avec la commande

```
mplot.ion()
```

5.1 Tracer une courbe (`matplotlib.pyplot.plot`)

Cette fonction permet de tracer une ou plusieurs courbes dans une fenêtre graphique. Elle s'utilise avec deux tableaux *de même taille*, le premier contient les valeurs des abscisses et le second celui des ordonnées. Par exemple :

```
mplot.plot(x,y, label='y(x)')
```

Pour tracer plusieurs courbes, il suffit d'ajouter une ligne avec la même fonction. `python` trace automatiquement les courbes sur la même figure.

Pour modifier le tracé d'une courbe, on peut ajouter après les vecteurs d'abscisses et d'ordonnées une chaîne de caractères qui modifie le tracé de la paire précédente. Voici quelques exemples, la liste complète se trouve dans l'aide disponible avec la commande `help(mplot.plot)`.

couleur :	'b'	bleu foncé
	'c'	bleu clair
	'g'	vert clair
	'k'	noir
	'm'	magenta
	'r'	rouge
style des points :	'+'	dessine des signes plus
	'x'	dessine des croix
	'o'	dessine des ronds
	'*'	dessine des étoiles

	's'	dessine des carrés
	'd', 'D'	dessinent des losanges
	'^', '>', 'v', '<'	dessinent des triangles avec différentes orientations
style de ligne :	'-'	trace une ligne continue
	'--'	trace des tirets
	':'	trace des pointillés
	'-.'	alterne points et tirets

Ainsi pour tracer une courbe en pointillés magenta avec des étoiles on écrit

```
matplotlib.pyplot(x,y,'m*:')
```

Il est également possible de modifier la largeur du tracé, mais pour cela, la syntaxe est différente. Par exemple `matplotlib.pyplot(x,y,linewidth=2)` trace les lignes avec une largeur double.

`matplotlib.semilogx` fait la même chose que `plot` avec l'axe des abscisses en échelle logarithmique.
`matplotlib.semilogy` Fait la même chose que `plot` avec l'axe des ordonnées en échelle logarithmique.
`matplotlib.loglog` Fait la même chose que `plot` avec les deux axes en échelle logarithmique.

5.2 Légendes, titre etc.

`matplotlib.title` Cette commande permet de donner un titre à la figure de la figure courante.
`matplotlib.xlabel` Cette commande permet de nommer l'axe des abscisses de la figure courante.
`matplotlib.ylabel` Cette commande permet de nommer l'axe des ordonnées de la figure courante.
`matplotlib.legend` Cette commande permet de nommer les différentes courbes de la figure courante. Les noms des courbes doivent être entrés dans le même ordre que les données dans la commande `plot` (ou son équivalent).

5.3 Enregistrer une figure

La commande `matplotlib.savefig` permet d'enregistrer la dernière figure (la figure *courante*) dans un fichier image. Il existe de nombreux formats disponibles, le format est choisi automatiquement en fonction de l'extension donnée. Par exemple

```
matplotlib.savefig('nom_du_fichier.png')
```

enregistre la figure courante au format PNG. Voici toutes les options disponibles

```
matplotlib.savefig(fname, dpi=None, facecolor='w', edgecolor='w',  
                   orientation='portrait', papertype=None, format=None,  
                   transparent=False, bbox_inches=None, pad_inches=0.1,  
                   frameon=None, metadata=None)
```

5.4 Exemple complet

```
# Importation des fonctions dont nous aurons besoin
from numpy import linspace, sin
import matplotlib.pyplot as mplot

# Matplotlib en mode interactif
mplot.ion()

# Définition des paramètres pour le tracé
xmin=0
xmax=10
n_points=100

# Le vecteur des abscisses
x=linspace(xmin, xmax, n_points)

# Le vecteur des ordonnées
y=sin(x)

# Définition de la taille de la figure
mplot.figure(figsize=(13,8))

# Définition de la taille des caractères
mplot.rcParams.update({'font.size': 22})

# Placement de la légende dans le coin en bas à gauche
mplot.rcParams.update({'legend.loc': 'lower left'})

# On trace la courbe
mplot.plot(x, y, label='sinus')

# On ajoute les légendes des axes
mplot.legend()
mplot.xlabel(r'$x$')          # Le 'r' devant la chaîne indique qu'on
mplot.ylabel(r'$y=\sin(x)$') # utilise LaTeX pour écrire les légendes.
```

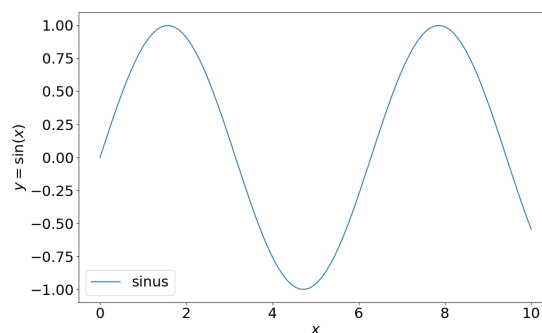


Figure 1: Figure affichée par le script de l'exemple 5.4.

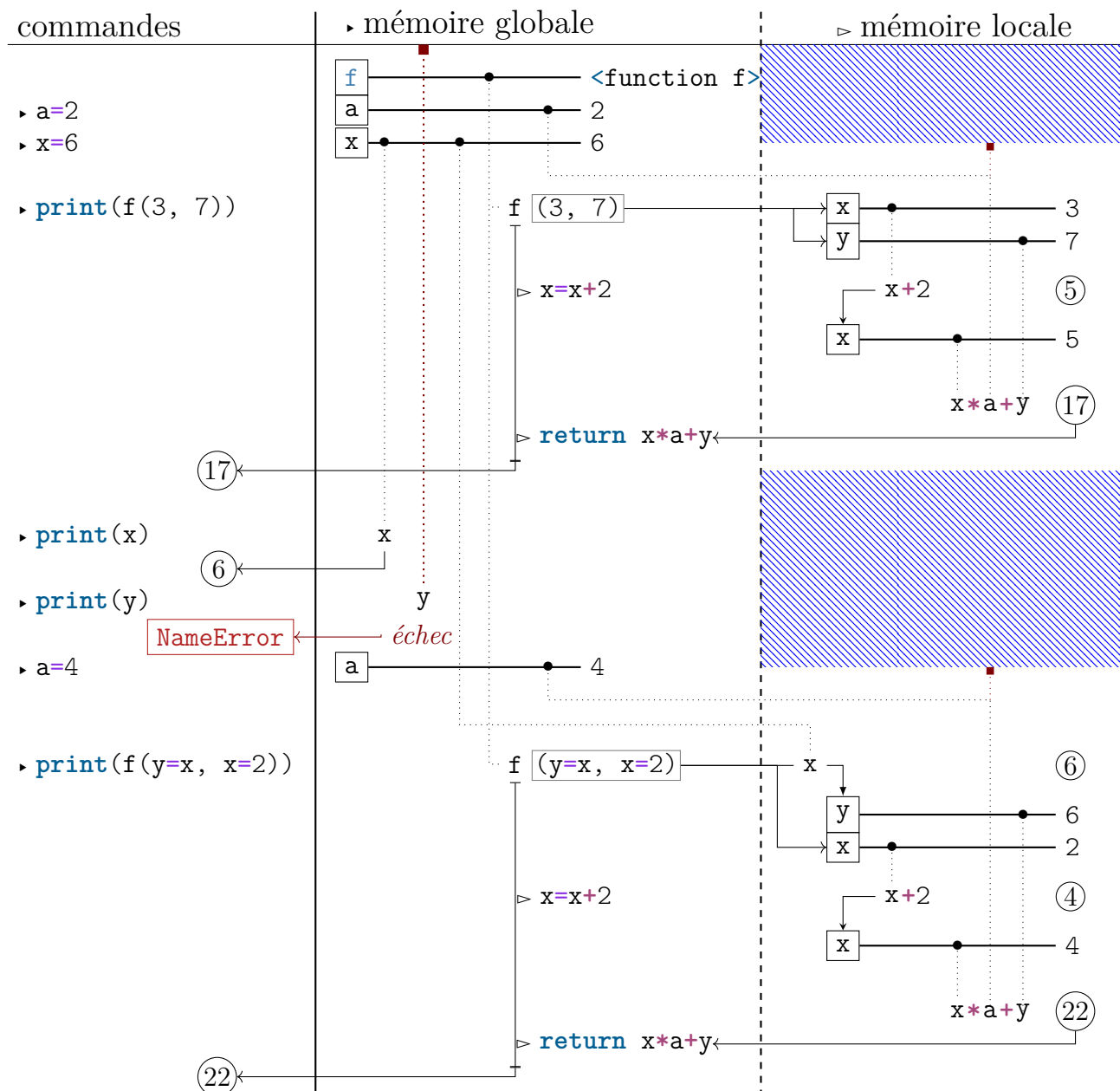


Figure 2: Schéma de fonctionnement de la mémoire montrant la mémoire globale et le fonctionnement de la mémoire locale lors d'appels à une fonction. Ce schéma reprend l'exemple traité en page 17. Les commandes de la colonne de gauche sont celles qui sont entrées directement dans l'interface. Les valeurs entourées, comme ④, sont dans la mémoire vive mais n'ont pas de nom, elles sont en attente d'être nommées (c'est-à-dire sauvegardées) ou utilisées (par exemple ici affichées avec `print`). Les valeurs sauvegardées dans chaque zone de mémoire sont encadrées par un rectangle et sont rattachées à leur contenu par une ligne horizontale épaisse, comme `x` — 3. Les affectations d'une valeur en mémoire sont indiquées par des flèches comme `→ x` et les appels à la mémoire sont indiqués par des lignes pointillées `.....↓` où le point, placé sur une ligne épaisse reliant un nom de variable à son contenu, indique la réussite de la recherche en mémoire et le résultat trouvé. Un échec de recherche en mémoire est représenté par un carré rouge `.....■`. Le diagramme se lit de haut en bas. Les zones hachurées correspondent aux séquences pendant lesquelles la mémoire locale est vide.