


Calcul Différentiel II

STEP, MINES ParisTech*

5 mars 2021 (#72befad)

Table des matières

Objectifs d'apprentissage	2
Théorème des fonctions implicites	3
Théorème des fonctions implicites	3
Extensions	6
Difféomorphisme	9
Inverse de la différentielle	9
Théorème d'inversion locale	9
Calcul avec les nombres flottants	10
Nombres flottants binaires	11
Un modèle simplifié : \mathbb{D}	12
Représentation d'un nombre réel comme un double	13
Précision et erreur relative	14
Chiffres significatifs de la représentation décimale	15
Arrondi des fonctions	15
Différentiation automatique	16
Introduction	16
Autograd	18
Différences finies	19
Différence avant	19
Erreur d'arrondi	20
Schémas d'ordre supérieur	22

*Ce document est un des produits du projet  boisgera/CDIS, initié par la collaboration de (S)ébastien Boisgérault (CAOR), (T)homas Romary et (E)milie Chautru (GEOSCIENCES), (P)auline Bernard (CAS), avec la contribution de Gabriel Stoltz (Ecole des Ponts ParisTech, CERMICS). Il est mis à disposition selon les termes de la licence Creative Commons “attribution – pas d'utilisation commerciale – partage dans les mêmes conditions” 4.0 internationale.

Annexe – Différentiation automatique	23
Tracer le graphe de calcul	23
Différentielle des fonctions élémentaires	29
Différentielle des fonctions composées	30
Exploitation	31
Pour aller plus loin	33
Exercices complémentaires	34
Cinématique d’un robot manipulateur	34
Déformations	35
Valeurs propres d’une matrice	35
Différences finies – erreur d’arrondi	35
Solution des exercices	36
Exercices essentiels	36
Cinématique d’un robot manipulateur	40
Déformations	41
Valeurs propres d’une matrice	42
Différences finies – erreur d’arrondi	43
Références	44

Objectifs d’apprentissage

Cette section s’efforce d’expliciter et de hiérarchiser les acquis d’apprentissages associés au chapitre. Ces objectifs sont organisés en paliers :

(o) Prérequis (●) Fondamental (●●) Standard (●●●) Avancé (●●●●) Expert

Sauf mention particulière, les objectifs “Expert”, les démonstrations du document¹ et les contenus en annexe ne sont pas exigibles (“hors-programme”).

Théorème des fonctions implicites

- connaître la portée du théorème des fonctions implicites (TFI) qui
 - o explicite une relation fonctionnelle implicite entre des variables liées,
 - ● localement au voisinage d’un jeu de valeurs admissibles,
 - ● sous des hypothèses de régularité et d’inversibilité.
- ●● savoir reconnaître un problème relevant directement du TFI,
- ●●● savoir transformer un problème pour que le TFI devienne applicable.
- ● savoir calculer/exploiter la différentielle de la fonction implicite,
- ●● connaître la méthode de Newton (/modifiée) sous-tendant le résultat,
- ● savoir caractériser et exploiter les C^1 -difféomorphismes,
- ●● savoir exploiter le théorème d’inversion locale qui résulte du TFI.

1. l’étude des démonstrations du cours peut toutefois contribuer à votre apprentissage, au même titre que la résolution d’exercices.

Calcul avec les nombres flottants

- o savoir que les calculs numériques avec les flottants sont inexacts,
- • savoir représenter un nombre réel en base 10 et en base 2,
- pour les nombres flottants à double précision :
 - • connaître la définition et la valeur de ε ,
 - •• savoir caractériser un double (modèle simplifié des doubles \mathbb{D}),
 - • connaître les propriétés importantes de l'opération d'arrondi.
- savoir évaluer les erreurs numériques introduites par :
 - •• la représentation d'un réel par un double (arrondi),
 - ••• une opération mathématique correctement arrondie.

Différentiation automatique

- • connaître la fonction d'un outil de différentiation automatique,
- • savoir comparer cette approche au calcul de différences finies,
- •• savoir exploiter la bibliothèque autograd dans un projet numérique,
- •••• comprendre les principes de fonctionnement d'un tel outil.

Différences finies

- o connaître le principe fondant les méthodes de différences finies,
- • savoir les mettre en œuvre dans un projet numérique,
- • savoir expliquer les deux types d'erreurs qu'elles engendrent,
- •• savoir quantifier les erreurs de troncature,
- ••• savoir quantifier les erreurs d'arrondi.

Théorème des fonctions implicites

Théorème – Théorème des fonctions implicites

Soit f une fonction définie sur un ouvert W de $\mathbb{R}^m \times \mathbb{R}^n$ et à valeurs dans \mathbb{R}^m

$$f : (x, \lambda) \in W \subset \mathbb{R}^m \times \mathbb{R}^n \rightarrow \mathbb{R}^m$$

qui soit continûment différentiable et telle que la différentielle partielle² $\partial_x f$ soit inversible en tout point de W . Si le point (x_0, λ_0) de W vérifie $f(x_0, \lambda_0) = 0$, alors il existe des voisinages ouverts U de x_0 et V de λ_0 tels que $U \times V \subset W$ et une unique fonction implicite $\psi : V \rightarrow \mathbb{R}^m$, continûment différentiable, telle que pour tout $\lambda \in V$ et tout $x \in U$,

$$f(x, \lambda) = 0 \Leftrightarrow x = \psi(\lambda).$$

2. Les différentielles partielles de f par rapport aux variables x et λ sont définies en fixant la valeur des variables complémentaire :

$$\partial_x f(x, \lambda) := d(x' \mapsto f(x', \lambda))(x) \text{ et } \partial_\lambda f(x, \lambda) := d(\lambda' \mapsto f(x, \lambda'))(\lambda).$$

La notion généralise celle de dérivée partielle au cas où les variables sont vectorielles.

De plus, la différentielle de ψ est donnée pour tout $\lambda \in V$ par

$$d\psi(\lambda) = -(\partial_x f(x, \lambda))^{-1} \cdot \partial_\lambda f(x, \lambda) \text{ où } x = \psi(\lambda).$$

Exemple – Description locale d’un ensemble comme un graphe Les points (x_1, x_2) du cercle unité de \mathbb{R}^2 centré sur l’origine sont exactement les solutions de l’équation

$$x_1^2 + x_2^2 = 1.$$

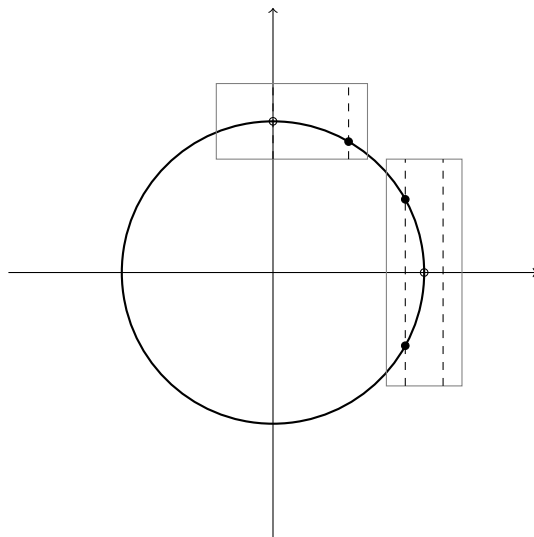


FIGURE 1 – Au voisinage de $(0, 1)$ le cercle unité est localement le graphe d’une fonction de l’abscisse du point, mais cela n’est pas le cas au voisinage du point $(1, 0)$.

Au voisinage de $(0, 1)$, l’expression $x_1^2 + x_2^2 - 1$ satisfait les hypothèses du théorème des fonctions implicites avec $x := x_2$ et $\lambda := x_1$. En particulier, la dérivée partielle $\partial_{x_2}(x_1^2 + x_2^2 - 1) = 2x_2$ ne s’annule pas localement, la différentielle partielle associée $h \in \mathbb{R} \mapsto 2x_2 h \in \mathbb{R}$ est donc inversible. Pour les points du cercle unité dans un voisinage suffisamment petit de ce point, l’ordonnée x_2 est donc fonction de l’abscisse x_1 .

Au voisinage de $(1, 0)$, ces hypothèses ne sont pas satisfaites ; on constate d’ailleurs que même si x_1 est arbitrairement proche de 1 et que l’on restreint la recherche des solutions x_2 à un voisinage arbitrairement petit de 0, il peut toujours exister 0 ou 2 solutions x_2 .

Exercice – Le cercle unité (•) Déterminer les points (x_{10}, x_{20}) du cercle

$$C := \{(x_1, x_2) \in \mathbb{R}^2 \mid x_1^2 + x_2^2 = 1\}$$

au voisinage desquels le cercle est le graphe d'une fonction continûment différentiable $x_2 = \psi(x_1)$. On déterminera alors explicitement des intervalles ouverts V contenant x_{10} et U contenant x_{20} ainsi que la fonction $\psi : V \rightarrow \mathbb{R}$ tels que $x_1^2 + x_2^2 = 1 \Leftrightarrow x_2 = \psi(x_1)$ et l'on calculera $\psi'(x_1)$. (Solution p. 36.)

Exercice – Abscisse curviligne (••) Soit $f :]a, b[\rightarrow \mathbb{R}^2$ une fonction continûment différentiable, dont la dérivée f' ne s'annule pas. Soit $c \in]a, b[$; montrer qu'il existe une unique fonction x définie dans un voisinage ouvert de $0 \in \mathbb{R}$ et à valeurs dans \mathbb{R} telle que $x(0) = c$ et

$$\int_c^{x(s)} \|f'(t)\| dt = s.$$

(Solution p. 36.)

Exercice – Courbes de niveau (••) Soit $f : (x_1, x_2) \in \mathbb{R}^2 \rightarrow \mathbb{R}$ une fonction continûment différentiable et $c \in \mathbb{R}$ tels que l'ensemble

$$C = \{(x_1, x_2) \in \mathbb{R}^2 \mid f(x_1, x_2) = c\}$$

est non vide et que ∇f ne s'annule pas sur C . Soit $(x_{10}, x_{20}) \in C$,

$$v = \frac{\nabla f(x_{10}, x_{20})}{\|\nabla f(x_{10}, x_{20})\|}$$

et $u \in \mathbb{R}^2$ le vecteur tel que (u, v) soit une base orthonormée. Soit (w, z) les coordonnées d'un point P dans cette base; montrer que pour tout point de C , il existe un voisinage de ce point et une fonction ψ pour lesquels $P \in C \Leftrightarrow z = \psi(w)$. (Solution p. 37.)

Exercice – Détermination de l'angle (•••) Soit $u = (u_1, u_2)$ un vecteur de \mathbb{R}^2 distinct de l'origine. Une *détermination de l'angle* de u est un nombre réel θ tel que

$$\frac{u}{\|u\|} = \begin{bmatrix} \cos \theta \\ \sin \theta \end{bmatrix}.$$

Montrer que pour tout vecteur $u_0 \neq 0$ dont l'angle est déterminé par θ_0 , il existe dans un voisinage ouvert U de u_0 une unique fonction $\Theta : U \rightarrow \mathbb{R}$ continûment différentiable telle que $\Theta(u)$ soit une détermination de l'angle de u . Montrer ensuite que

$$\nabla \Theta(u_0) = \frac{1}{\|u_0\|} \begin{bmatrix} -\sin \theta_0 \\ \cos \theta_0 \end{bmatrix}.$$

(Indication : réécrire l'équation initiale reliant θ et u sous la forme d'une équation scalaire qui lui est localement équivalente.) (Solution p. 37.)

Extensions

Il est possible d'affaiblir l'hypothèse concernant $\partial_x f$ en supposant uniquement celle-ci inversible en (x_0, λ_0) au lieu d'inversible sur tout W . En effet, l'application qui à une application linéaire inversible de \mathbb{R}^m dans lui-même associe son inverse est définie sur un ouvert et continue³. Comme l'application linéaire $\partial_x f(x_0, \lambda_0)$ est inversible et que l'application $\partial_x f$ est continue, il existe donc un voisinage ouvert de (x_0, λ_0) contenu dans W où $\partial_x f$ est inversible. Nous retrouvons donc les hypothèses initiales du théorème, à ceci près qu'elle sont satisfaites dans un voisinage de (x_0, λ_0) qui peut être plus petit que l'ouvert initial W .

Démonstration La partie la plus technique de la démonstration concerne l'existence et la différentiabilité de la fonction implicite ψ . Mais si l'on admet temporairement ces résultats, établir l'expression de $d\psi$ est relativement simple. En effet, l'égalité $f(\psi(\lambda), \lambda) = 0$ étant satisfaite identiquement sur U et la fonction $\lambda \in V \mapsto f(\psi(\lambda), \lambda)$ étant différentiable comme composée de fonctions différentiables, la règle de dérivation en chaîne fournit en tout point de U :

$$\partial_x f(\psi(\lambda), \lambda) \cdot d\psi(\lambda) + \partial_\lambda f(\psi(\lambda), \lambda) = 0.$$

On en déduit donc que

$$d\psi(\lambda) = -(\partial_x f(\psi(\lambda), \lambda))^{-1} \cdot \partial_\lambda f(\psi(\lambda), \lambda).$$

L'inversion d'opérateurs linéaires ainsi que leur composition étant des opérations continues, si f est continûment différentiable et que $d\psi$ existe, elle est nécessairement continue.

Pour établir l'existence de la fonction implicite ψ , nous allons pour une valeur λ suffisamment proche de λ_0 construire une suite convergente d'approximations x_k , proches de x_0 , dont la limite x sera solution de $f(x, \lambda) = 0$.

L'idée de cette construction repose sur l'analyse suivante : si nous partons d'une valeur x_k proche de x_0 (a priori telle que $f(x_k, \lambda) \neq 0$) et que nous recherchons une valeur x_{k+1} proche, qui soit une solution approchée de $f(\lambda, x) = 0$ (meilleure que ne l'est x_k), comme au premier ordre

$$f(x_{k+1}, \lambda) \approx f(x_k, \lambda) + \partial_x f(x_k, \lambda) \cdot (x_{k+1} - x_k),$$

3. **Continuité de l'inversion.** Une matrice $A \in \mathbb{R}^{m \times m}$ est inversible si et seulement si son déterminant est non-nul. Or, la fonction $A \mapsto \det A$ est continue car le déterminant ne fait intervenir que des produits et des sommes des coefficients de A . Par conséquent, les matrices inversibles de $\mathbb{R}^{m \times m}$ sont l'image réciproque de l'ouvert $\mathbb{R} \setminus \{0\}$ par une application continue : cet ensemble est donc ouvert. Quand A est inversible, on a

$$A^{-1} = \frac{\text{co}([A])^t}{\det[A]}$$

où $\text{co}(A)$ désigne la comatrice de A . Chaque coefficient de cette comatrice ne faisant également intervenir que des sommes et des produits des coefficients de A , l'application $A \mapsto A^{-1}$ est continue sur son domaine de définition.

nous en déduisons que la valeur x_{k+1} définie par

$$x_{k+1} := x_k - (\partial_x f(x_k, \lambda))^{-1} \cdot f(x_k, \lambda)$$

vérifie $f(x_{k+1}, \lambda) \approx 0$. On peut espérer que répéter ce processus en partant de x_0 détermine une suite convergente dont la limite soit une solution exacte x de $f(x, \lambda) = 0$.

Le procédé décrit ci-dessus constitue la *méthode de Newton* de recherche de zéros. Nous allons prouver que cette heuristique est ici justifiée, à une modification mineure près : nous allons lui substituer une *méthode de Newton modifiée*, qui n'utilise pas $\partial_x f(x_k, \lambda)$ mais la valeur constante $\partial_x f(x_0, \lambda_0)$, c'est-à-dire qui définit la suite

$$x_{k+1} := x_k - Q^{-1} \cdot f(x_k, \lambda) \quad \text{où } Q = \partial_x f(x_0, \lambda_0).$$

Cette définition par récurrence se réécrit sous la forme $x_{k+1} = \phi_\lambda(x_k)$ où

$$\phi_\lambda(x) = x - Q^{-1} \cdot f(x, \lambda).$$

La fonction ϕ_λ est différentiable sur l'ensemble $\{x \in \mathbb{R}^m \mid (x, \lambda) \in W\}$ et sa différentielle est donnée par

$$d\phi_\lambda(x) = I - Q^{-1} \cdot \partial_x f(x, \lambda)$$

où I désigne la fonction identité. En écrivant que $\partial_x f(x, \lambda)$ est la somme de $\partial_x f(x_0, \lambda_0)$ et de $\partial_x f(x, \lambda) - \partial_x f(x_0, \lambda_0)$, on obtient

$$\begin{aligned} \|d\phi_\lambda(x)\| &\leq \|I - Q^{-1} \cdot Q\| + \|Q^{-1} \cdot (\partial_x f(x, \lambda) - Q)\| \\ &\leq \|Q^{-1}\| \times \|\partial_x f(x, \lambda) - Q\|. \end{aligned}$$

La fonction f étant supposée continûment différentiable, il existe un $r > 0$ tel que tout couple (x, λ) tel que $\|x - x_0\| \leq r$ et $\|\lambda - \lambda_0\| \leq r$ appartienne à W et vérifie $\|\partial_x f(x, \lambda) - Q\| \leq \kappa \|Q^{-1}\|^{-1}$ avec par exemple $\kappa = 1/2$, ce qui entraîne $\|d\phi_\lambda(x)\| \leq \kappa$. Par l'inégalité des accroissements finis, la restriction de ϕ_λ à $\{x \in \mathbb{R}^m \mid \|x - x_0\| \leq r\}$ (que l'on continuera à noter ϕ_λ) est κ -contractante:

$$\|\phi_\lambda(x) - \phi_\lambda(w)\| \leq \kappa \|x - w\|.$$

Par ailleurs,

$$\|\phi_\lambda(x) - x_0\| \leq \|\phi_\lambda(x) - \phi_\lambda(x_0)\| + \|\phi_\lambda(x_0) - \phi_{\lambda_0}(x_0)\|.$$

On a d'une part

$$\|\phi_\lambda(x) - \phi_\lambda(x_0)\| \leq \kappa \|x - x_0\| \leq \kappa r$$

et d'autre part, par continuité de $\lambda \mapsto \phi_\lambda(x_0)$ en λ_0 , il existe un r' tel que $0 < r' < r$ et tel que si $\|\lambda - \lambda_0\| \leq r'$, alors $\|\phi_\lambda(x_0) - \phi_{\lambda_0}(x_0)\| \leq (1 - \kappa)r$. Pour de telles valeurs de λ ,

$$\|\phi_\lambda(x) - x_0\| \leq \kappa r + (1 - \kappa)r = r.$$

L'image de la boule fermée $B = \{x \in \mathbb{R}^m \mid \|x - x_0\| \leq r\}$ par l'application ϕ_λ est donc incluse dans B . Tant que $\|\lambda - \lambda_0\| \leq r'$, les hypothèses du théorème de point fixe de Banach sont donc satisfaites pour $\phi_\lambda : B \rightarrow B$, ce qui montre l'existence et l'unicité de la fonction implicite ψ associée aux voisinages ouverts $V = B(\lambda_0, r')$ et $U = B(x_0, r)$.

Montrons la continuité de la fonction implicite ψ . Soit λ_1, λ_2 deux points de V ; notons $x_1 = \psi(\lambda_1)$ et $x_2 = \psi(\lambda_2)$. Ces valeurs sont des solutions des équations de point fixe

$$x_1 = \phi_{\lambda_1}(x_1) \text{ et } x_2 = \phi_{\lambda_2}(x_2).$$

En formant la différence de x_2 et x_1 , on obtient donc

$$\begin{aligned} \|x_2 - x_1\| &= \|\phi_{\lambda_2}(x_2) - \phi_{\lambda_1}(x_1)\| \\ &\leq \|\phi_{\lambda_2}(x_2) - \phi_{\lambda_2}(x_1)\| + \|\phi_{\lambda_1}(x_1) - \phi_{\lambda_2}(x_1)\|. \end{aligned}$$

La fonction ϕ_{λ_2} étant κ -contractante, le premier terme du membre de droite de cette inégalité est majoré par $\kappa\|x_2 - x_1\|$, par conséquent

$$\|x_2 - x_1\| \leq \frac{1}{1 - \kappa} \|\phi_{\lambda_1}(x_1) - \phi_{\lambda_2}(x_1)\|.$$

L'application $\lambda \mapsto \phi_\lambda(x_1)$ étant continue en λ_1 , nous pouvons conclure que x_2 tend vers x_1 quand λ_2 tend vers λ_1 ; autrement dit : la fonction implicite ψ est continue en λ_1 .

Montrons finalement la différentiabilité de ψ en λ_1 . Pour cela, il suffit d'exploiter la différentiabilité de f en (x_1, λ_1) où $x_1 = \psi(\lambda_1)$. Elle fournit l'existence d'une fonction ε qui tende vers 0 en 0 telle que

$$\begin{aligned} f(x, \lambda) &= f(x_1, \lambda_1) + \partial_x f(x_1, \lambda_1) \cdot (x - x_1) + \partial_\lambda f(x_1, \lambda_1) \cdot (\lambda - \lambda_1) \\ &\quad + \varepsilon((x - x_1, \lambda - \lambda_1))(\|x - x_1\| + \|\lambda - \lambda_1\|) \end{aligned}$$

On a par construction $f(x_1, \lambda_1) = 0$; en prenant $x = \psi(\lambda)$, on annule également $f(x, \lambda) = 0$. En notant $Q = \partial_x f(x_1, \lambda_1)$ et $P = \partial_\lambda f(x_1, \lambda_1)$, on obtient

$$\begin{aligned} \psi(\lambda) &= \psi(\lambda_1) - Q^{-1} \cdot P \cdot (\lambda - \lambda_1) \\ &\quad - Q^{-1} \cdot \varepsilon((\psi(\lambda) - \psi(\lambda_1), \lambda - \lambda_1))(\|\psi(\lambda) - \psi(\lambda_1)\| + \|\lambda - \lambda_1\|). \end{aligned}$$

Nous allons exploiter une première fois cette égalité. Notons tout d'abord que

$$\varepsilon_\lambda(\lambda - \lambda_1) := \varepsilon((\psi(\lambda) - \psi(\lambda_1), \lambda - \lambda_1))$$

tend vers 0 en 0 en raison de la continuité de ψ en λ_1 . En choisissant λ dans un voisinage suffisamment proche de λ_1 , on peut donc garantir que ce terme est arbitrairement petit, par exemple, tel que

$$\|Q^{-1}\| \times \|\varepsilon_\lambda(\lambda - \lambda_1)\| \leq \frac{1}{2},$$

ce qui permet d'obtenir

$$\|\psi(\lambda) - \psi(\lambda_1)\| \leq \|Q^{-1}P\| \times \|\lambda - \lambda_1\| + \frac{1}{2}\|\lambda - \lambda_1\| + \frac{1}{2}\|\psi(\lambda) - \psi(\lambda_1)\|$$

et donc

$$\|\psi(\lambda) - \psi(\lambda_1)\| \leq \alpha \|\lambda - \lambda_1\| \text{ avec } \alpha := 2\|Q^{-1}P\| + 1.$$

En exploitant une nouvelle fois la même égalité, on peut désormais conclure que

$$\|\psi(\lambda) - \psi(\lambda_1) - Q^{-1} \cdot P \cdot (\lambda - \lambda_1)\| \leq \|\varepsilon'_\lambda(\lambda - \lambda_1)\| \times \|\lambda - \lambda_1\|.$$

où la fonction ε'_λ est la fonction tendant vers 0 et définie par

$$\varepsilon'_\lambda(\lambda - \lambda_1) := (1 + \alpha) \times \|Q^{-1}\| \times \|\varepsilon_\lambda(\lambda - \lambda_1)\|,$$

ce qui prouve la différentiabilité de ψ en λ_1 et conclut la démonstration. ■

Définition – Difféomorphisme

Une fonction $f : U \subset \mathbb{R}^n \rightarrow V \subset \mathbb{R}^n$, où les ensembles U et V sont ouverts est un *C^1 -difféomorphisme* (de U sur V) si f est bijective et que f ainsi que son inverse f^{-1} sont continûment différentiables.

Théorème – Inverse de la différentielle

Si $f : U \rightarrow V$ est un C^1 -difféomorphisme, sa différentielle df est inversible en tout point x de U et

$$(df(x))^{-1} = df^{-1}(y) \text{ où } y = f(x).$$

Démonstration Les fonctions f et f^{-1} sont différentiables et vérifient

$$f \circ f^{-1} = I \text{ et } f^{-1} \circ f = I.$$

La règle de différentiation en chaîne, appliquée en $y = f(x)$ et x respectivement, fournit donc

$$df(x) \cdot df^{-1}(y) = I \text{ et } df^{-1}(y) \cdot df(x) = I.$$

La fonction $df(x)$ est donc inversible et son inverse est $df^{-1}(y)$. ■

Théorème – Théorème d'inversion locale

Soit $f : U \subset \mathbb{R}^n \rightarrow \mathbb{R}^n$ continûment différentiable sur l'ouvert U et telle que $df(x)$ soit inversible en tout point x de U . Alors f est un *C^1 -difféomorphisme local* : pour tout $x_0 \in U$, il existe un voisinage ouvert $V \subset U$ de x_0 tel que $W = f(V)$ soit ouvert et que la restriction de la fonction f à V soit un C^1 -difféomorphisme de V sur W .

Démonstration Considérons la fonction $\phi : U \times \mathbb{R}^n \rightarrow \mathbb{R}^n$ définie par

$$\phi(x, y) = f(x) - y.$$

Par construction $\phi(x, y) = 0$ si et seulement si $f(x) = y$. De plus, ϕ est continûment différentiable et $\partial_x \phi(x, y) = df(x)$. On peut donc appliquer le théorème des fonctions implicites au voisinage du point $(x_0, f(x_0))$ et en déduire l'existence de voisinages ouverts A et B de x_0 et $f(x_0)$ tels que $A \times B \subset U \times \mathbb{R}^n$, et d'une fonction continûment différentiable $\psi : B \rightarrow \mathbb{R}^m$ telle que pour tout $(x, y) \in A \times B$,

$$f(x) = y \Leftrightarrow x = \psi(y).$$

Par continuité de f , $V := A \cap f^{-1}(B)$ est un sous-ensemble ouvert de A . La fonction $x \in V \mapsto f(x) \in W := B$ est bijective par construction et son inverse est la fonction $y \in W \mapsto \psi(y) \in V$; nous avons donc affaire à un C^1 -difféomorphisme de V sur W . ■

Exercice – Coordonnées polaires (•) Montrer que l'application

$$f : (r, \theta) \in]0, +\infty[\times \mathbb{R} \rightarrow (r \cos \theta, r \sin \theta) \in \mathbb{R}^2 \setminus \{(0, 0)\}$$

est un difféomorphisme local. Est-ce un difféomorphisme global? (Solution p. 37.)

Calcul avec les nombres flottants

Cette section introduit la représentation des nombres réels sur ordinateur comme des “doubles” – le type le plus utilisé des nombres à virgule flottante – et leur propriétés élémentaires. Pour avoir plus d'informations sur le sujet, vous pouvez vous reporter au document classique “What every computer scientist should know about floating-point arithmetic” (Goldberg 1991).

Les exemples de code utilisés dans la suite utiliseront Python 3 et NumPy :

```
>>> from numpy import *
```

NumPy fournit un nombre nommé `pi` dont nous pouvons afficher les décimales :

```
>>> pi
3.141592653589793
```

Cette représentation de `pi` fournie par l'interpréteur Python est non-ambiguë ; par là nous voulons dire que la chaîne de caractère “3.141592653589793” fournit assez d'information pour reconstituer le nombre initial `pi` si nécessaire :

```
>>> number = eval("3.141592653589793")
>>> number == pi
True
```

Mais cette représentation n'en est pas moins un subtil mensonge, car elle n'est pas une représentation décimale exacte du nombre `pi` qui est stocké en mémoire. Pour avoir sa représentation exacte, nous pouvons demander l'affichage d'un grand nombre de décimales :

```
>>> def print_exact_number(number):
...     print(f"{number:.100g}")
>>> print_exact_number(pi)
3.141592653589793115997963468544185161590576171875
```

Demander 100 chiffres après la virgule s'avère suffisant : seuls 49 chiffres sont effectivement affichés car les suivants sont tous nuls.

Remarquez que nous avons obtenu une représentation exacte du nombre flottant `pi` avec 49 chiffres. Cela ne signifie pas que tous ces chiffres – ou même la plupart d'entre eux – sont significatifs dans la représentation du nombre réel π . En effet, si nous utilisons la bibliothèque Python `mpmath` (Johansson and others 2013) pour l'arithmétique flottante multi-précision, pour avoir les vraies décimales de π , nous voyons que

```
>>> import mpmath
>>> mpmath.mp.dps = 49; mpmath.mp.pretty = True
>>> +mpmath.pi
3.141592653589793238462643383279502884197169399375
```

Les deux représentations ne sont identiques que jusqu'au 16ème chiffre (3.141592653589793...).

Nombres flottants binaires

Si la représentation des nombres flottants peut apparaître complexe à ce stade, c'est que nous avons insisté pour utiliser une représentation *décimale* quand ces nombres exploitent en réalité une représentation *binnaire*. En d'autres termes ; au lieu d'utiliser une suite de chiffres décimaux $f_i \in \{0, 1, \dots, 9\}$ pour représenter un nombre réel x comme

$$x = \pm(f_0.f_1f_2\dots f_i\dots) \times 10^e$$

nous devrions utiliser des *chiffres binaires* – ou *bits* – $f_i \in \{0, 1\}$ pour écrire

$$x = \pm(f_0.f_1f_2\dots f_i\dots) \times 2^e.$$

Ces représentations sont *normalisées* si le chiffre avant la virgule est non nul. Par exemple, avec cette convention, le nombre rationnel 999/1000 serait représenté en base 10 comme 9.99×10^{-1} et non comme 0.999×10^0 . En base 2, le seul chiffre non-nul est 1, donc la *mantisse* d'une représentation normalisée est toujours de la forme $(1.f_1f_2\dots f_i\dots)$.

En calcul scientifique, les nombres réels sont le plus souvent décrits de façon approchés par des “doubles”. Dans la bibliothèque standard Python, les doubles sont les instances du type `float` ; pour NumPy des instances du type `float64`⁴.

“Double” est un raccourci pour “nombre à virgule flottante de précision double”, comme défini dans le standard IEEE 754, cf. (IEEE Task P754 1985) ; chaque double occupe en mémoire un espace de 64 bits, d’où le nom `float64`. Un format de simple précision, qui utilise uniquement 32 bits est aussi défini ; NumPy le propose sous le nom `float32`. Après un abandon progressif des “singles” au profit des “doubles”, plus précis et mieux supportés par les CPUs modernes, le format de simple précision revient désormais en force avec le développement de l’usage des GPUs comme unités de calcul génériques⁵ ;

Un double *normalisé* x prend la forme

$$x = (-1)^s \times (1.f_1f_2 \dots f_{52}) \times 2^{e-1023}.$$

Les doubles qui ne sont pas normalisés sont *not-a-number* (`nan`), plus ou moins l’infini (`inf`) et zero (`0.0`) (en fait ± 0.0 ; car il existe deux zéros distincts, qui diffèrent par leur signe) ou les nombres dits *dénormés* qui existent dans un voisinage de 0. Dans la suite, nous ne parlerons pas de ces cas particuliers.

TABLE 1: Anatomie d’un double normalisé x (`float64`)

Composante	$x = (-1)^s \times (1.f_1f_2 \dots f_{52}) \times 2^{e-1023}$	Nombre de bits
Bit de signe	$s \in \{0, 1\}$	1/64
Exposant (biaisé)	$e \in \{1, \dots, 2046\}$	11/64
Mantisse	$f = (f_1, \dots, f_{52}) \in \{0, 1\}^{52}$	52/64

Un modèle simplifié : \mathbb{D}

Si l’on décide d’oublier les nombres particuliers que nous venons de décrire pour nous concentrer sur les nombres normalisés et que de plus nous permettons à l’exposant e de décrire \mathbb{Z} tout entier, nous obtenons un modèle simplifié des doubles, sous la forme d’un sous-ensemble \mathbb{D} de \mathbb{R} :

$$\mathbb{D} = \{0\} \cup \{(-1)^s \times (1.f_1f_2 \dots f_{52}) \times 2^e \mid s \in \{0, 1\}, e \in \mathbb{Z}, f \in \{0, 1\}^{52}\}.$$

Pour simplifier la situation, ce modèle ignore délibérément la possibilité d’un “dépassement” (*overflow*) – le choix d’un exposant trop grand pour un vrai double – ou d’un “soutassement” (*underflow*) – c’est-à-dire un exposant trop petit. Il

4. Fort heureusement, les conversions entre `float` et `float64`, quand elles sont nécessaires, sont transparentes pour l’utilisateur de Python et NumPy.

5. On trouve même un format de demi-précision (“half”), accessible en NumPy sous le nom de `float16`.

retient néanmoins l’essentiel des caractéristiques des doubles comme le caractère discret de l’ensemble et la mesure de leur espacement. Il constitue donc un bon modèle d’étude pour la suite.

Représentation d’un nombre réel comme un double

Presque aucun nombre réel ne peut être représenté exactement comme un double. Pour faire face à cette difficulté, il est raisonnable d’associer à un nombre réel x “sa meilleure approximation” parmi les doubles, notée \mathbf{x} ou $[x]$:

$$\mathbf{x} = [x].$$

On choisit en général en tant que *méthode d’arrondi* (*round-off*)

$$[\cdot] : \mathbb{R} \rightarrow \mathbb{D}$$

la méthode *arrondi-au-plus-proche*⁶ :

$$[x] := \arg \min_{\mathbf{x} \in \mathbb{D}} |\mathbf{x} - x|.$$

Mais il existe des modes alternatifs d’arrondi (arrondis “orientés”, vers $+\infty$ ou $-\infty$) qui peuvent être utiles. Ces opérations ont en commun les propriétés suivantes :

- si x est un double, $[x] = x$,
- sinon, $[x]$ est :
 - soit le double immédiatement inférieur à x ,
 - soit le double immédiatement supérieur à x .

Exercice – Représentation des entiers (••) Le type `int32` de Python permet de représenter tous les entiers entre -2147483648 (-2^{31}) et 2147483647 ($2^{31} - 1$). Est-ce que tous ces entiers sont des doubles ? (Solution p. 38.)

Exercice – Nombres réels et doubles (•) Parmi les réels 1.0 , $2/3$, 0.5 et 0.1 , lesquels sont des doubles ? (Indication : en base 2, $2/3 = 0.1010101010\dots$ et $0.1 = 0.0001100110011\dots$) (Solution p. 38.)

6. Il faudrait préciser comment l’opération se comporte quand le réel est équidistant de deux doubles, un niveau de détail dont nous ne nous préoccupons pas dans la suite. Comme il existe deux façons de faire qui sont standard (cf. la norme IEEE 754, [IEEE Task P754 (1985)]), il faudrait pour être exact parler des méthodes d’arrondi au plus proche.

Précision et erreur relative

Pour avoir la moindre confiance dans le résultats des calculs que nous effectuons avec des doubles, nous devons être en mesure d'évaluer l'erreur faite en représentant x par $[x]$, nommée *erreur d'arrondi* :

$$e = [x] - x.$$

L'*epsilon machine* ε est une grandeur clé à cet égard : il est défini comme l'écart entre 1.0 – qui peut être représenté exactement comme un double – et le double qui lui est immédiatement supérieur.

```
>>> after_one = nextafter(1.0, +inf)
>>> after_one
1.0000000000000002
>>> print_exact_number(after_one)
1.0000000000000002220446049250313080847263336181640625
>>> eps = after_one - 1.0
>>> print_exact_number(eps)
2.220446049250313080847263336181640625e-16
```

Ce nombre est également disponible comme un attribut de la classe `finfo` de NumPy qui rassemble les constantes limites de l'arithmétique pour les types flottants.

```
>>> print_exact_number(finfo(float64).eps)
2.220446049250313080847263336181640625e-16
```

Alternativement, l'examen de la structure des doubles normalisés fournit directement la valeur de ε : la mantisse du nombre après 1.0 est $(f_1, f_2, \dots, f_{51}, f_{52}) = (0, 0, \dots, 0, 1)$, et son exposant 0 donc

$$\varepsilon = 2^{-52},$$

un résultat confirmé par l'expérience :

```
>>> print_exact_number(2**(-52))
2.220446049250313080847263336181640625e-16
```

L'epsilon machine importe autant parce qu'il fournit une borne simple sur l'erreur relative de la représentation d'un nombre réel comme un double. En effet, si $2^e \leq x < 2^{e+1}$, comme dans cette région la distance entre deux doubles consécutifs est $2^{-52} \times 2^e$, l'erreur d'arrondi vérifie $|[x] - x| \leq \varepsilon \times 2^e$ et comme $2^e \leq |x|$, nous obtenons l'inégalité :

$$\frac{|[x] - x|}{|x|} \leq \varepsilon.$$

(si la méthode "arrondi-au-plus-proche" est utilisée, il est même possible de garantir la borne plus contraignante $\varepsilon/2$ au lieu de ε). L'epsilon machine contrôle donc l'*erreur relative* introduite par l'opération d'arrondi.

Exercice – π contre \mathbf{pi} (•) Par quel nombre peut-on borner l'écart entre π et $\mathbf{pi} = [\pi]$? (Solution p. 38.)

Exercice – Constantes de la Physique (•) Quelle précision peut-on attendre des doubles $\mathbf{N} = [N]$ et $\mathbf{h} = [h]$ représentant respectivement le nombre d'Avogadro N et la constante de Planck h ?

$$N = 6.02214076 \times 10^{23} \text{ mol}^{-1}, \quad h = 6.62607015 \times 10^{-34} \text{ J} \times \text{s}.$$

(Solution p. 38.)

Chiffres significatifs de la représentation décimale

L'erreur relative détermine la précision de la représentation décimale utilisée pour représenter un nombre réel par un double. Considérons la représentation de x en notation scientifique (on suppose x positif pour simplifier l'analyse) :

$$[x] = (f_0.f_1 \dots f_{p-1} \dots) \times 10^e.$$

Dans cette notation par convention, $[x]$ est compris entre $1.000 \dots \times 10^e$ et $9.999 \dots \times 10^e \leq 10^{e+1}$. La borne dont nous disposons sur l'erreur relative nous garantit donc

$$|x - [x]| \leq \varepsilon \times 10^{e+1} \approx 2.2 \times 10^{e-15},$$

voire une borne deux fois plus petite si $[\cdot]$ est l'arrondi au plus proche. On peut donc considérer que les chiffres f_0 à f_{14} ou f_{15} sont significatifs dans la représentation de x , et les suivants a priori sans intérêt.

Arrondi des fonctions

La plupart des nombres réels ne pouvant être représentés par des doubles, la plupart des fonctions à valeur réelle et à variables réelles ne peuvent pas non plus être représentées exactement comme des fonctions opérant sur des doubles. Le mieux que nous puissions espérer est d'avoir des approximations *correctement arrondies*. Une approximation notée \mathbf{f} (ou $[f]$) d'une fonction f de n variables est correctement arrondie si pour tout n -uplet $(\mathbf{x}_1, \dots, \mathbf{x}_n) \in \mathbb{D}^n$ de doubles dans le domaine de définition de f , on a

$$\mathbf{f}(\mathbf{x}_1, \dots, \mathbf{x}_n) = [f(\mathbf{x}_1, \dots, \mathbf{x}_n)].$$

Autrement dit, tout se passe comme si le calcul de $\mathbf{f}(\mathbf{x}_1, \dots, \mathbf{x}_n)$ était effectué de la façon suivante : calcul **exact** de f sur ces arguments, puis arrondi du résultat pour transformer le réel ainsi obtenu en un double. Si l'on veut exploiter une telle fonction avec des arguments réels (qui ne sont pas nécessairement des doubles),

il suffit d'arrondir ses arguments avant d'entreprendre les calculs ; on a alors la fonction correctement arrondie

$$\mathbf{f} = [\cdot] \circ f \circ ([\cdot], \dots, [\cdot]).$$

Le standard IEEE 754 (IEEE Task P754 1985) impose que certaines fonctions aient des implémentations correctement arrondies ; nommément, l'addition, la soustraction, la multiplication, la division, le reste d'une division entière et la racine carrée. D'autres fonctions élémentaires – comme sinus, cosinus, exponentielle, logarithme – ne sont en général pas correctement arrondies ; la conception d'algorithmes de calcul qui aient une performance décente et qui soient correctement arrondis est un problème difficile (cf. Fousse et al. (2007)).

Exercice – WTF Python ? (●) Expliquer pourquoi en Python on a comme on s'y attend :

```
>>> 0.3
0.3
```

mais aussi le résultat moins intuitif :

```
>>> 0.1 + 0.2
0.30000000000000004
```

(On se contentera de donner un scénario plausible et qualitatif de ce qui se passe quand on approxime $0.1 + 0.2$ par $0.1 + 0.2$.) (Solution p. 38.)

Exercice – WTF Python, vraiment ? (●●●) Pour faire taire les sceptiques, détailler les calculs de $0.1 + 0.2$ en base 2 pour étayer votre explication. (Solution p. 39.)

Exercice – Non-associativité de l'addition (●●) On note $\cdot + \cdot : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{D}$ la version correctement arrondie de l'addition entre réels quand $[\cdot]$ désigne l'arrondi au double le plus proche. Calculer

$$(((1.0 + \varepsilon/4) + \varepsilon/4) + \varepsilon/4) + \varepsilon/4 \text{ et } 1 + (\varepsilon/4 + (\varepsilon/4 + (\varepsilon/4 + \varepsilon/4))).$$

Pouvez-vous déterminer expérimentalement si Python interprète $\mathbf{x} + \mathbf{y} + \mathbf{z}$ comme $(\mathbf{x} + \mathbf{y}) + \mathbf{z}$ ou comme $\mathbf{x} + (\mathbf{y} + \mathbf{z})$? (Solution p. 39.)

Différentiation automatique

Introduction

La différentiation automatique désigne une famille de méthodes numériques permettant de calculer dérivées, gradients et matrices jacobiniennes de fonctions

numériques. Ces méthodes ont l'avantage majeur d'éliminer une grande partie des erreurs d'arrondis que l'on trouve typiquement associées aux méthodes classiques qui seront étudiées dans la section suivante. L'erreur est en fait aussi faible que dans une dérivation symbolique “manuelle” des fonctions à dériver et ce sans réglage délicat de paramètres à effectuer.

Concrètement, pour dériver la fonction

$$f : x \in \mathbb{R} \mapsto \frac{1 - e^{-2x}}{1 + e^{-2x}} \in \mathbb{R},$$

avec une librairie de différentiation automatique en Python comme autograd, il faut tout d'abord implémenter la fonction f , par exemple sous la forme

```
def f(x):
    y = exp(-2.0 * x)
    u = 1.0 - y
    v = 1.0 + y
    w = u / v
    return w
```

L'algorithme de différentiation automatique construit alors à partir de f une fonction dérivée qui est fonctionnellement équivalente à la fonction que vous auriez probablement implémentée manuellement :

```
def g(x):
    y = exp(-2.0 * x)
    u = 1.0 - y
    v = 1.0 + y
    w = u / v
    dx = 1.0
    dy = -2.0 * exp(-2.0 * x) * dx
    du = 0.0 - dy
    dv = 0.0 + dy
    dw = du / v + u * (- dv) / (v * v)
    return dw
```

Les erreurs numériques induites par ce procédé ne sont donc pas totalement nulles (f n'est pas f et g n'est pas $g := f'$).

Selon le langage informatique utilisé pour implémenter les fonctions numériques (C, Fortran, Python, langages “embarqués”, etc.), différentes méthodes permettent de mettre en œuvre la différentiation automatique. Le typage dynamique (ou *duck typing*) de Python permet de mettre en œuvre simplement le *tracing* des fonctions numériques – l'enregistrement des opérations de calcul effectuées par une fonction lors de son exécution. À partir de ce graphe de calcul, les différentielles peuvent être calculées mécaniquement par la règle de différentiation en chaîne à partir des différentielles des opérations élémentaires. L'annexe “Différentiation automatique” vous explique comment développer une

micro-bibliothèque de différentiation automatique en Python ... à des fins pédagogiques uniquement ! Si vous avez besoin d'utiliser la différentiation automatique dans un projet, consultez la section suivante (p. 18).

Autograd

La bibliothèque Python autograd (<https://github.com/HIPS/autograd>) est un bon choix par défaut si vous avez besoin de différentiation automatique et que vous souhaitez continuer à utiliser NumPy comme vous en avez l'habitude.

```
>>> import autograd
>>> from autograd.numpy import *
```

Le second appel est un peu surprenant – pourquoi pas simplement `from numpy import *` – mais il va nous permettre d'utiliser la version des fonctions NumPy fournie par autograd. Cela est rendu nécessaire par l'utilisation du tracing pour déterminer les graphes de calcul, une méthode qui suppose une forme de coopération des fonctions numériques impliquées. Comme les fonction de NumPy en sont incapables, autograd les modifie puis les met ensuite à disposition dans son propre module NumPy ; c'est la version que vous devrez utiliser et non le module original, sans quoi des erreurs cryptiques sont à craindre.

La documentation d'autograd fournit une bonne illustration d'utilisation pour calculer la dérivée d'une fonction scalaire d'une variable :

```
>>> def f(x):
...     y = exp(-2.0 * x)
...     return (1.0 - y) / (1.0 + y)
>>> deriv_f = autograd.grad(f)
>>> deriv_f(1.0)
0.419974341614026
```

Pour les fonctions scalaires de plusieurs variables, le fragment de code suivant fournit un exemple de calcul du gradient :

```
>>> def f(x, y):
...     return sin(x) + 2.0 * sin(y)
>>> def grad_f(x, y):
...     g = autograd.grad
...     return array([g(f, 0)(x, y), g(f, 1)(x, y)])
>>> grad_f(0.0, 0.0)
array([1., 2.]
```

Pour les fonctions vectorielles, le calcul de la matrice jacobienne peut prendre la forme suivante :

```
>>> def f(x, y):
...     return array([1.0 * x + 2.0 * y, 3.0 * x + 4.0 * y])
>>> def J_f(x, y):
```

```

...     j = autograd.jacobian
...     return array([j(f, 0)(x, y), j(f, 1)(x, y)]).T
>>> J_f(0.0, 0.0)
array([[1., 2.],
       [3., 4.]])

```

Différences finies

Compte tenu de la définition de la dérivée d'une fonction, la méthode de différentiation numérique la plus naturelle pour évaluer une dérivée repose sur le schéma des *différences finies* de Newton, qui exploite l'approximation

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

valable lorsque la valeur du *pas* h est suffisamment faible.

L'implémentation de ce schéma en Python est simple :

```

def FD(f, x, h):
    return (f(x + h) - f(x)) / h

```

Néanmoins, la relation entre la valeur du pas h et la précision de cette évaluation – c'est-à-dire l'écart entre la valeur de la dérivée et son estimation – est plus complexe. Considérons les échantillons de données suivants :

```

>>> FD(exp, 0, 1e-4)
1.000050001667141
>>> FD(exp, 0, 1e-8)
0.999999993922529
>>> FD(exp, 0, 1e-12)
1.000088900582341

```

La valeur théorique de $\exp'(0)$ étant 1.0, la valeur la plus précise de la dérivée numérique est obtenue pour $h = 10^{-8}$ et uniquement 8 nombres après la virgule du résultat sont significatifs.

Pour la valeur plus grande $h = 10^{-4}$, la précision est limitée par la qualité du développement de Taylor de \exp au premier ordre ; cette erreur dite *de troncature* décroît linéairement avec la taille du pas. Pour la valeur plus petite de $h = 10^{-12}$, la précision est essentiellement limitée par les erreurs d'*arrondi* dans les calculs, liées à la représentation approchée des nombres réels utilisée par le programme informatique.

Différence avant

Soit f une fonction à valeurs réelles définie sur un intervalle ouvert. Dans de nombreux cas concrets, on peut faire l'hypothèse que la fonction f est indéfiniment

dérivable sur son domaine de définition ; le développement de Taylor avec reste intégral fournit alors localement à tout ordre n ,

$$f(x+h) = f(x) + f'(x)h + \frac{f''(x)}{2}h^2 + \dots + \frac{f^{(n)}(x)}{n!}h^n + O(h^{n+1})$$

où le terme $O(h^k)$ (notation “grand o” de Landau), désigne une expression de la forme $O(h^k) := M(h)h^k$ où M est une fonction définie et bornée dans un voisinage de 0.

Un calcul direct montre que

$$f'(x) = \frac{f(x+h) - f(x)}{h} + O(h)$$

Le comportement asymptotique de ce schéma de *différence avant* (*forward difference*) – contrôlé par le terme $O(h^1)$ – est dit d’ordre 1. Une implémentation de ce schéma est définie pour les réels x et h par

$$\text{FD}(f, x, h) = \left\lceil \frac{[f([x] + [h])] - [f([x])]}{[h]} \right\rceil = (\mathbf{f}(\mathbf{x} + \mathbf{h}) - \mathbf{f}(\mathbf{x})) / \mathbf{h}$$

où $\mathbf{x} = [x]$, $\mathbf{h} = [h]$, $\mathbf{f} = [\cdot] \circ f \circ [\cdot]$, $a - b = [[a] - [b]]$ et $a / b = [[a] / [b]]$.

Erreur d’arrondi

Nous considérons à nouveau la fonction $f(x) = \exp(x)$ utilisée dans l’introduction et nous calculons la dérivée numérique basée sur la différence avant à $x = 0$ pour différentes valeurs de h .

Le graphe de $h \mapsto \text{FD}(\exp, 0, h)$ (p. 21) montre que pour les valeurs de h proches ou inférieures à l’epsilon machine, la différence entre la dérivée numérique et la valeur exacte de la dérivée n’est pas expliquée par l’analyse classique liée au développement de Taylor.

Toutefois, si nous prenons en compte la représentation des réels comme des doubles, nous pouvons expliquer et quantifier le phénomène. Pour étudier exclusivement l’erreur d’arrondi, nous aimerions nous débarrasser de l’erreur de troncature ; à cette fin, dans les calculs qui suivent, au lieu de \exp , nous utilisons \exp_1 , le développement de Taylor de \exp à l’ordre 1 à $x = 0$, c’est-à-dire $\exp_1(x) := 1 + x$.

Supposons que l’arrondi soit calculé au plus proche ; sélectionnons un double $h > 0$ et comparons-le à l’epsilon machine :

- Si $h \ll \varepsilon$, alors $1 + h$ est proche de 1, en fait plus proche de 1 que du double immédiatement supérieur à 1 qui est $1 + \varepsilon$. Par conséquent, on a $[\exp_1](h) = 1$; une *annulation catastrophique* survient :

$$\text{FD}(\exp_1, 0, h) = \left\lceil \frac{[\exp_1](h) - 1}{h} \right\rceil = 0.$$

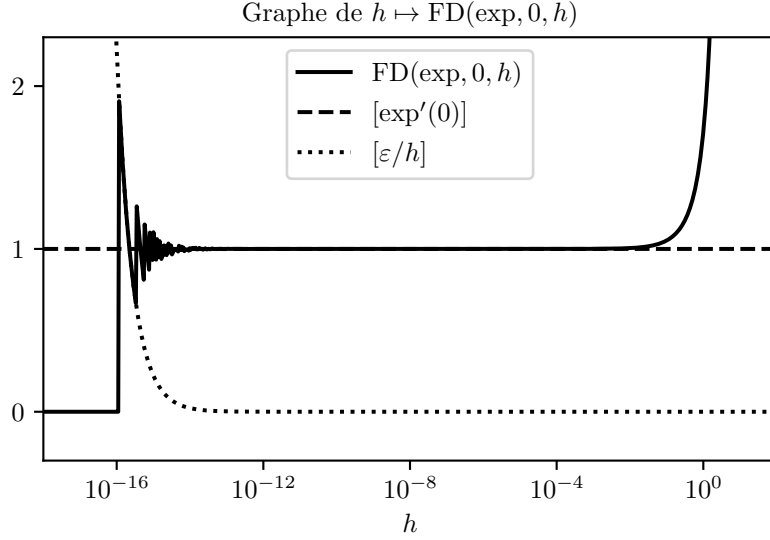


FIGURE 2 – Valeurs de la différence avant

- Si $h \approx \varepsilon$, alors $1+h$ est plus proche $1+\varepsilon$ que de 1, et donc $[\exp_1](h) = 1+\varepsilon$, ce qui entraîne

$$\text{FD}(\exp_1, 0, h) = \left\lceil \frac{[\exp_1](h) - 1}{h} \right\rceil = \left\lceil \frac{\varepsilon}{h} \right\rceil.$$

- Si $\varepsilon \ll h \ll 1$, alors $[1+h] = 1+h \pm \varepsilon(1+h)$ (le symbole \pm est utilisé ici pour définir un intervalle de confiance⁷). Et donc

$$[\exp_1](h) - 1 = h \pm \varepsilon \pm \varepsilon(2h + \varepsilon + \varepsilon h)$$

et

$$\left\lceil \frac{[\exp_1](h) - 1}{h} \right\rceil = 1 \pm \frac{\varepsilon}{h} + \frac{\varepsilon}{h}(3h + 2\varepsilon + 3h\varepsilon + \varepsilon^2 + \varepsilon^2 h)$$

et par conséquent

$$\text{FD}(\exp_1, 0, h) = \exp'_1(0) \pm \frac{\varepsilon}{h} \pm \varepsilon', \quad \varepsilon' \ll \frac{\varepsilon}{h}.$$

Si l'on revient à $\text{FD}(\exp, 0, h)$ et si l'on exploite des échelles log-log pour représenter l'erreur totale, on peut clairement distinguer la région où l'erreur est dominée par l'erreur d'arrondi – l'enveloppe de cette section du graphe est $h \mapsto \log(\varepsilon/h)$ – et où elle est dominée par l'erreur de troncature – une pente 1 étant caractéristique des schémas d'ordre 1.

7. L'équation $a = b \pm c$ est à interpréter comme l'inégalité $|a - b| \leq |c|$.

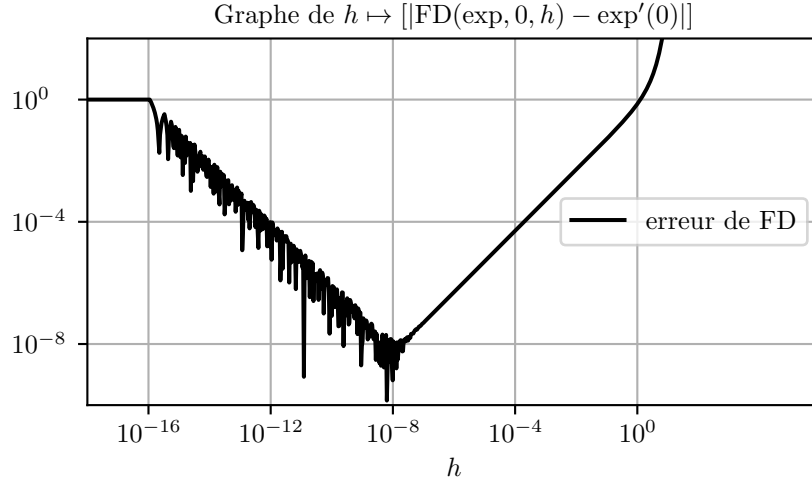


FIGURE 3 – Erreur de la différence avant

Schémas d'ordre supérieur

Le comportement asymptotique de la différence avant peut être amélioré, par exemple si au lieu de la différence avant nous utilisons un schéma de différence centrée. Considérons les développements de Taylor à l'ordre 2 de $f(x+h)$ et $f(x-h)$:

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

et

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

On en déduit

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

et donc, le schéma de *différence centrée* est d'ordre 2. Son implémentation sur ordinateur est donnée par

$$\text{CD}(f, x, h) = \left[\frac{[[f]([x] + [h]) - [f]([x] - [h])]}{2 \times [h]} \right].$$

ou de façon équivalente en Python :

```
def CD(f, x, h):
    return 0.5 * (f(x + h) - f(x - h)) / h
```

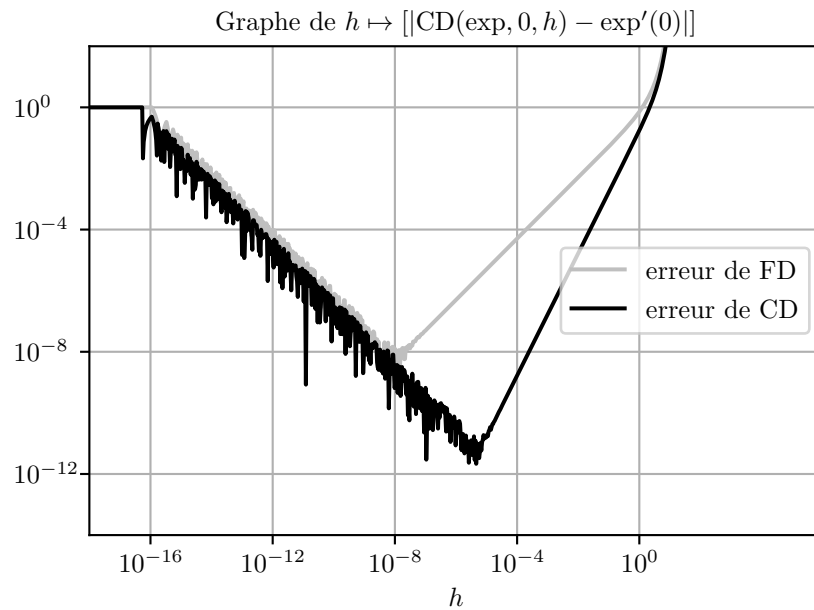


FIGURE 4 – Erreur de la différence centrée

Le graphe d'erreur associé à la différence centrale (p. 23) confirme qu'une erreur de troncature d'ordre 2 améliore la précision. Toutefois, il montre aussi que l'utilisation d'un schéma d'ordre plus élevé augmente également la région où l'erreur est dominée par l'erreur d'arrondi et rend la sélection d'un pas correct h encore plus difficile.

Annexe – Différentiation automatique

Tracer le graphe de calcul

Python étant typé dynamiquement, il n'attribue pas de type aux arguments des fonctions lors de leur définition. Ainsi, la fonction d'addition,

```
def add(x, y):
    return x + y
```

permet bien sûr d'additionner des nombres flottants

```
>>> add(1.0, 2.0)
3.0
```

mais elle marchera aussi parfaitement avec des entiers ou des tableaux NumPy ou même des types non-numériques comme des chaînes de caractères

```
>>> add(1, 2)
3
>>> add(array([1.0, 1.0]), array([2.0, 2.0]))
array([3., 3.])
>>> add("un", "deux")
'undeux'
```

Le tout est qu'à l'exécution, les objets `x` et `y` supportent l'opération d'addition – dans le cas contraire, une exception sera générée. Notre fonction `add` est donc définie implicitement pour les objets additionnables⁸.

Cela est également confirmé par l'examen du *bytecode* de la fonction `add`, qui ne fait aucune référence au type des arguments `x` et `y`.

```
>>> from dis import dis
>>> dis(add)
2           0 LOAD_FAST           0 (x)
           2 LOAD_FAST           1 (y)
           4 BINARY_ADD
           6 RETURN_VALUE
```

Dans le cas de l'addition, l'opération `x + y` est déléguée à la méthode `__add__` de l'objet `x`. Pour intercepter cet appel, il est donc nécessaire de modifier le type de nombre flottant que nous allons utiliser et de surcharger la définition de cette méthode :

```
class Float(float):
    def __add__(self, other):
        print(f"trace: {self} + {other}")
        return super().__add__(other)
```

Notre classe dérivant du type standard `float`, les opérations que nous n'avons pas redéfinies explicitement seront gérées comme d'habitude. Nous avons donc juste modifié l'addition des instances de `Float`, et encore de façon très limitée puisque nous avons délégué le calcul du résultat à la classe parente `float`.

Une fois cet effort fait, nous pouvons bien tracer les additions effectuées

```
>>> x = Float(2.0) + 1.0
trace: 2.0 + 1.0
>>> x
3.0
```

8. les objets “additionnables” sont des “canards” dans le contexte du terme *duck typing* de Python ; on ne demande pas qu'ils soient (ou dérivent) d'une classe particulière comme `numpy.number` par exemple – mais juste qu'ils se comportent de façon adéquate à l'exécution : “if it walks like a duck and it quacks like a duck, then it must be a duck”.

à condition bien sûr de travailler avec des instances de `Float` et non de `float` ! Pour commencer à généraliser cet usage, nous allons faire en sorte de générer des instances de `Float` dans la mesure du possible. Pour commencer, nous pouvons faire en sorte que les opérations sur nos flottants renvoient notre propre type de flottant :

```
class Float(float):
    def __add__(self, other):
        print(f"trace: {self} + {other}")
        return Float(super().__add__(other))
```

Mais cela n'est pas suffisant : les fonctions de la library `math` de Python vont renvoyer des flottants classiques, il nous faut donc à nouveau les adapter ; avant tout importons le module `math`

```
import math
```

puis définissons notre propre fonction `cos`:

```
def cos(x):
    print(f"trace: cos({x})")
    return Float(math.cos(x))
```

Vérifions le résultat:

```
>>> cos(pi) + 1.0
trace: cos(3.141592653589793)
trace: -1.0 + 1.0
0.0
```

Malheureusement, nous ne savons pas encore tracer correctement l'expression pourtant très similaire `1.0 + cos(pi)`:

```
>>> 1.0 + cos(pi)
trace: cos(3.141592653589793)
0.0
```

En effet, c'est la méthode `__add__` de `1.0`, une instance de `float` qui est appelée ; cet appel n'est donc pas tracé. Pour réussir à gérer correctement ce type d'appel, il va falloir ... le faire échouer ! La méthode appelée pour effectuer la somme jusqu'à présent confie l'opération à la méthode `__add__` de `1.0` parce que cet objet sait prendre en charge l'opération, car il s'agit d'ajouter lui-même avec une autre instance (qui dérive) de `float`. Si nous faisons en sorte que le membre de gauche soit incapable de prendre en charge cette opération, elle sera confiée au membre de droite et à la méthode `__radd__` ; pour cela il nous suffit de remplacer `Float`, un type numérique, par `Node`, une classe qui contient (encapsule) une valeur numérique :

```
class Node:
    def __init__(self, value):
        self.value = value
```

Nous n'allons pas nous attarder sur cette version 0 de `Node`. Si elle est ainsi nommée, c'est parce qu'elle va représenter un nœud dans un graphe de calculs. Au lieu d'afficher les opérations réalisées sur la sortie standard, nous allons enregistrer les opérations que subit chaque variable et comment elles s'organisent ; chaque nœud issu d'une opération devra mémoriser quelle opération a été appliquée, et quels étaient les arguments de l'opération (eux-mêmes des nœuds). Pour supporter cette démarche, `Node` devient :

```
class Node:
    def __init__(self, value, function=None, *args):
        self.value = value
        self.function = function
        self.args = args
```

Il nous faut alors rendre les opérations usuelles compatibles avec la création de nœuds ; en examinant les arguments de la fonction, on doit décider si elle est dans un mode "normal" (recevant des valeurs numériques, produisant des valeurs numériques) ou en train de tracer les calculs. Par exemple :

```
def cos(x):
    if isinstance(x, Node):
        cos_x_value = math.cos(x.value)
        cos_x = Node(cos_x_value, cos, x)
        return cos_x
    else:
        return math.cos(x)
```

ou

```
def add(x, y):
    if isinstance(x, Node) or isinstance(y, Node):
        if not isinstance(x, Node):
            x = Node(x)
        if not isinstance(y, Node):
            y = Node(y)
        add_x_y_value = x.value + y.value
        return Node(add_x_y_value, add, x, y)
    else:
        return x + y
```

La fonction `add` ne sera sans doute pas utilisée directement, mais appelée au moyen de l'opérateur `+` ; elle doit donc nous permettre de définir les méthodes `__add__` et `__radd__` :

```
Node.__add__ = add
Node.__radd__ = add
```

On remarque de nombreuses similarités entre les deux codes ; plutôt que de continuer cette démarche pour toutes les fonctions dont nous allons avoir besoin, au prix d'un effort d'abstraction, il serait possible de définir une fonction opérant

automatiquement cette transformation. Il s'agit d'une fonction d'ordre supérieur car elle prend comme argument une fonction (la fonction numérique originale) et renvoie une nouvelle fonction, compatible avec la gestion des nœuds. On pourra ignorer son implémentation en première lecture.

```
def autodiff(function):
    def autodiff_function(*args):
        if any([isinstance(arg, Node) for arg in args]):
            node_args = []
            values = []
            for arg in args:
                if isinstance(arg, Node):
                    node_args.append(arg)
                    values.append(arg.value)
                else:
                    node_args.append(Node(arg))
                    values.append(arg)
            output_value = function(*values)
            output_node = Node(
                output_value, autodiff_function, *node_args
            )
            return output_node
        else:
            return function(*args)
    autodiff_function.__qualname__ = function.__qualname__
    return autodiff_function
```

Malgré sa complexité apparente, l'utilisation de cette fonction est simple ; ainsi pour rendre la fonction `sin` et l'opérateur `*` compatible avec la gestion de nœuds, il suffit de faire :

```
sin = autodiff(math.sin)
```

et

```
def multiply(x, y):
    return x * y
multiply = autodiff(multiply)
Node.__mul__ = Node.__rmul__ = multiply
```

ce qui est sensiblement plus rapide et lisible que la démarche entreprise pour `cos` et `+` ; mais encore une fois, le résultat est le même.

Il est désormais possible d'implémenter le traceur. Celui-ci encapsule les arguments de la fonction à tracer dans des nœuds, puis appelle la fonction et renvoie le nœud associé à la valeur retournée par la fonction :

```
>>> def trace(f, args):
...     args = [Node(arg) for arg in args]
...     end_node = f(*args)
```

```
...     return end_node
```

Pour vérifier que tout se passe bien comme prévu, faisons en sorte d'afficher une représentation lisible et sympathique des contenus des nœuds sous forme de chaîne de caractères :

```
def node_str(node):
    if node.function is None:
        return str(node.value)
    else:
        function_name = node.function.__qualname__
        args_str = ", ".join(str(arg) for arg in node.args)
        return f"{function_name}({args_str})"
```

Puis, faisons en sorte qu'elle soit utilisée quand on invoque la fonction `print`, plutôt que l'affichage standard :

```
Node.__str__ = node_str
```

Nous complétons cette description par une seconde représentation, plus explicite mais également plus verbeuse :

```
def node_repr(node):
    reprs = [repr(node.value)]
    if node.function is not None:
        reprs.append(node.function.__qualname__)
    if node.args:
        reprs.extend([repr(arg) for arg in node.args])
    args_repr = ", ".join(reprs)
    return f"Node({args_repr})"
Node.__repr__ = node_repr
```

Nous sommes prêts à faire notre vérification :

```
>>> def f(x):
...     return 1.0 + cos(x)
>>> end = trace(f, [pi])
>>> end
Node(0.0, add, Node(-1.0, cos, Node(3.141592653589793)), Node(1.0))
>>> print(end)
add(cos(3.141592653589793), 1.0)
```

Le résultat se lit de la façon suivante : le calcul de `f(pi)` produit la valeur 0.0, issue de l'addition de -1.0, calculé comme `cos(3.141592653589793)`, et de la constante 1.0. Cela semble donc correct !

Un autre exemple – à deux arguments – pour la route :

```
>>> def f(x, y):
...     return x * (x + y)
>>> t = trace(f, [1.0, 2.0])
```

```
>>> t
Node(3.0, multiply, Node(1.0), Node(3.0, add, Node(1.0), Node(2.0)))
>>> print(t)
multiply(1.0, add(1.0, 2.0))
```

Différentielle des fonctions élémentaires

Pour exploiter le graphe de calcul que nous savons désormais déterminer, il nous faut déclarer les différentielles des opérations et fonctions primitives dans un “registre” de différentielles, indexées par la fonction à différencier.

```
differential = {}
```

Pour l’addition et la multiplication, nous exploitons les identités $d(x+y) = dx+dy$

```
def d_add(x, y):
    return add
differential[add] = d_add
```

et $d(x \times y) = x \times dy + dx \times y$

```
def d_multiply(x, y):
    def d_multiply_xy(dx, dy):
        return x * dy + dx * y
    return d_multiply_xy
differential[multiply] = d_multiply
```

Pour une fonction telle que \cos , nous exploitons l’identité $d(\cos(x)) = -\sin(x)dx$

```
def d_cos(x):
    def d_cos_x(dx):
        return - sin(x) * dx
    return d_cos_x
differential[cos] = d_cos
```

Mais il ne s’agit que d’un cas particulier de l’identité $d(f(x)) = f'(x)dx$. Nous pouvons nous doter d’une fonction qui calculera la différentielle df à partir de la dérivée f' :

```
def d_from_deriv(g):
    def d_f(x):
        def d_f_x(dx):
            return g(x) * dx
        return d_f_x
    return d_f
```

La déclaration de différentielles s’en trouve simplifiée ; ainsi on déduit de l’identité $(\sin x)' = \cos x$ la déclaration

```
differential[sin] = d_from_deriv(cos)
```

Différentielle des fonctions composées

Pour exploiter le tracing d'une fonction, il nous faut à partir du nœud final produit par ce procédé extraire l'ensemble des nœuds amont, qui représentent les arguments utilisés dans le calcul de la valeur finale. Puis, pour préparer le calcul de la différentielle, nous ordonnerons les nœuds de telle sorte que les arguments d'une fonction apparaissent toujours avant la valeur qu'elle produit. L'implémentation suivante, relativement naïve⁹, réalise cette opération:

```
def find_and_sort_nodes(end_node):
    todo = [end_node]
    nodes = []
    while todo:
        node = todo.pop()
        nodes.append(node)
        for parent in node.args:
            if parent not in nodes + todo:
                todo.append(parent)
    done = []
    while nodes:
        for node in nodes[:]:
            if all([parent in done for parent in node.args]):
                done.append(node)
                nodes.remove(node)
    return done
```

Le calcul de la différentielle en tant que tel ne consiste plus qu'à propager la variation des arguments de nœud en nœud, en se basant sur la règle de différentiation en chaîne; ces variations intermédiaires sont stockées dans l'attribut `d_value` des nœuds du graphe.

```
def d(f):
    def df(*args): # args=(x1, x2, ...)
        start_nodes = [Node(arg) for arg in args]
        end_node = f(*start_nodes)
        if not isinstance(end_node, Node): # constant value
            end_node = Node(end_node)
        nodes = find_and_sort_nodes(end_node).copy()
        def df_x(*d_args): # d_args = (d_x1, d_x2, ...)
            for node in nodes:
                if node in start_nodes:
                    i = start_nodes.index(node)
                    node.d_value = d_args[i]
                elif node.function is None: # constant node
                    node.d_value = 0.0
```

9. Comme toujours, si vous ou l'un des membres de votre unité était surpris par un informaticien en possession de ce code, l'UE 11 niera avoir connaissance de vos activités.

```

        else:
            _d_f = differential[node.function]
            _args = node.args
            _args_values = [_node.value for _node in _args]
            _d_args = [_node.d_value for _node in _args]
            node.d_value = _d_f(*_args_values)(*_d_args)
        return end_node.d_value
    return df_x
return df

```

Exploitation

Pour exploiter simplement notre calcul de différentielle, nous pouvons dans le cas d'une fonction d'une variable réelle en déduire la dérivée; rappelons qu'on a alors $f'(x) = df(x) \cdot 1$.

```

def deriv(f):
    df = d(f)
    def deriv_f(x):
        return df(x)(1.0)
    return deriv_f

```

Vérifions que le comportement de ces opérateurs de différentiation est conforme à nos attentes dans le cas de fonction d'une variable; d'abord dans le cas d'une fonction constante

```

>>> def f(x):
...     return pi
>>> g = deriv(f)
>>> g(0.0)
0.0
>>> g(1.0)
0.0

```

puis dans le cas d'une fonction affine

```

>>> def f(x):
...     return 2 * x + 1.0
>>> g = deriv(f)
>>> g(0.0)
2.0
>>> g(1.0)
2.0
>>> g(2.0)
2.0

```

et enfin dans le cas d'une fonction quadratique

```
>>> def f(x):
...     return x * x + 2 * x + 1
>>> g = deriv(f)
>>> g(0.0)
2.0
>>> g(1.0)
4.0
>>> g(2.0)
6.0
```

Pour finir dans ce cadre, testons deux fonctions utilisant les fonctions trigonométriques `sin` et `cos` :

```
>>> def f(x):
...     return cos(x) * cos(x) + sin(x) * sin(x)
>>> g = deriv(f)
>>> g(0.0)
0.0
>>> g(pi/4)
0.0
>>> g(pi/2)
0.0
```

```
>>> def f(x):
...     return sin(x) * cos(x)
>>> g = deriv(f)
>>> def h(x):
...     return cos(x) * cos(x) - sin(x) * sin(x)
>>> g(0.0) == h(0.0)
True
>>> g(pi/4) == h(pi/4)
True
>>> g(pi/2) == h(pi/2)
True
```

Dans le cas général – puisque nos fonctions sont toujours à valeurs réelles – nous pouvons déduire le gradient de la différentielle :

```
def grad(f):
    df = d(f)
    def grad_f(*args):
        n = len(args)
        grad_f_x = n * [0.0]
        df_x = df(*args)
        for i in range(0, n):
            e_i = n * [0.0]; e_i[i] = 1.0
            grad_f_x[i] = df_x(*e_i)
        return grad_f_x
```



```
    return grad_f
```

Les fonctions constantes, affines et quadratiques permettent là aussi de réaliser des tests élémentaires :

```
>>> def f(x, y):
...     return 1.0
>>> grad(f)(0.0, 0.0)
[0.0, 0.0]
>>> grad(f)(1.0, 2.0)
[0.0, 0.0]

>>> def f(x, y):
...     return x + 2 * y + 1
>>> grad(f)(0.0, 0.0)
[1.0, 2.0]
>>> grad(f)(1.0, 2.0)
[1.0, 2.0]

>>> def f(x, y):
...     return x * x + y * y
>>> grad(f)(0.0, 0.0)
[0.0, 0.0]
>>> grad(f)(1.0, 2.0)
[2.0, 4.0]
```

Pour aller plus loin

Le code de cette section, légèrement étendu (essentiellement avec plus d’opérateurs et de fonctions supportées) est disponible dans le fichier `autodiff.py` du dépôt git `boisgera/CDIS`. Mais il présente des limitations très importantes (pas de support de NumPy, pas de différentiations d’ordre supérieur, pas de support pour la différentiation rétrograde, etc.) et son intérêt est essentiellement pédagogique.

La librairie `autodidact` est également à vocation pédagogique, mais pallie largement à ces défauts. Le document compagnon est le chapitre 6 du cours CSC 421/2516 “Neural Networks and Deep Learning” de l’université de Toronto.

La librairie `autodidact` est une version volontairement simplifiée de la librairie `autograd` du groupe “Harvard Intelligent Probabilistic Systems”¹⁰ qui est sans doute l’implémentation “sérieuse” de différentiation automatique en Python qu’il conviendrait d’utiliser à l’issue de ce cours introductif. JAX, une implémentation optimisée de `autograd`, reposant sur XLA (pour *accelerated linear algebra*) est une autre option.

10. “autograd” ou “autodiff” sont des termes plus ou moins génériques qu’utilisent de nombreuses librairies.

La différentiation automatique fait également partie intégrante de nombreuses plate-formes de calcul, qu'il s'agisse de machine learning (pytorch, tensorflow, etc.) ou de programmation probabiliste (PyMC3, Stan, etc.). Pour finir, l'étude "Automatic Differentiation in Machine Learning: a Survey" (Baydin et al. 2015) vous permettra si besoin d'acquiescer si besoin une perspective plus large sur le sujet.

Exercices complémentaires

Cinématique d'un robot manipulateur

On revient à l'étude du robot plan dont les coordonnées cartésiennes (x, y) se déduisent des coordonnées articulaires $q = (\theta_1, \theta_2)$ par la relation

$$\begin{cases} x &= \ell_1 \cos \theta_1 + \ell_2 \cos(\theta_1 + \theta_2) \\ y &= \ell_1 \sin \theta_1 + \ell_2 \sin(\theta_1 + \theta_2) \end{cases}$$

On note f la fonction de \mathbb{R}^2 dans \mathbb{R}^2 telle que $f(\theta_1, \theta_2) = (x, y)$.

Question 1 (•) Supposons que $\ell_1 \neq \ell_2$. Déterminer l'ensemble des valeurs (x, y) du plan qui ne correspondent à aucun couple q de coordonnées articulaires et l'ensemble de celles qui correspondent à des coordonnées articulaires. Quand (x, y) appartient à l'intérieur V de ce second ensemble, ces coordonnées articulaires sont-elles uniques (modulo 2π) ? (Solution p. 40.)

Question 2 (•) Déterminer l'ensemble U des $q \in \mathbb{R}^2$ tel que la matrice $J_f(q)$ soit inversible et comparer $f(U)$ avec V . (Solution p. 40.)

Question 3 (•••) On considère une trajectoire continue

$$\gamma : t \in [0, 1] \mapsto (x(t), y(t)) \in \mathbb{R}^2$$

dans l'espace cartésien, dont l'image est incluse dans $f(U)$. Soit $q_0 = (\theta_{10}, \theta_{20})$ tel que $f(q_0) = (x(0), y(0))$. Montrer que si l'image de γ reste dans un voisinage suffisamment petit de $(x(0), y(0))$, il existe une unique fonction continue $\gamma_q : [0, 1] \rightarrow \mathbb{R}^2$ telle que $\gamma = f \circ \gamma_q$ et $\gamma_q(0) = q_0$ (γ_q est la trajectoire correspondant à γ dans l'espace articulaire). (Solution p. 41.)

Question 4 (••) Montrer que si γ est différentiable, γ_q également. Montrer que l'on peut déduire

$$\dot{q} := (\dot{\theta}_1, \dot{\theta}_2) := \gamma'_q(t) \text{ de } (\dot{x}, \dot{y}) := \gamma'(t).$$

(Solution p. 41.)

Déformations

Soit U un ouvert convexe de \mathbb{R}^n et $T : U \subset \mathbb{R}^n \rightarrow \mathbb{R}^n$ une fonction continûment différentiable. On suppose que T est de la forme $T = I + H$ où la fonction H vérifie

$$\sup_{x \in U} \|dH(x)\| := \kappa < 1.$$

(H est une *perturbation de l'identité*).

Question 1 (••) Montrer que la fonction T est injective. (Solution p. 41.)

Question 2 (•••) Montrer que l'image $V = T(U)$ est un ouvert et que T est un difféomorphisme global de U sur V . (Solution p. 42.)

Valeurs propres d'une matrice

Question 1 (••) Déterminer une condition “raisonnable” qui garantisse qu’une valeur propre $\lambda \in \mathbb{R}$ d’une matrice $A \in \mathbb{R}^{n \times n}$ varie continûment avec les coefficients de A . (Solution p. 42.)

Différences finies – erreur d’arrondi

Non seulement les erreurs d’arrondis sont susceptibles de générer une erreur importante dans les schémas de différences finies, mais cette erreur est susceptible de varier très rapidement avec la valeur du pas, d’une façon qui peut sembler aléatoire. Ainsi, si

```
>>> FD(exp, 0.0, h=1e-12)
1.000088900582341
```

on a également

```
>>> FD(exp, 0.0, h=9.999778782798785e-13)
1.0001110247585212
```

soit une erreur 25% plus élevée, pour une variation de 0.002% du pas seulement. Inversement, avec

```
>>> FD(exp, 0.0, h=9.99866855977416e-13)
1.0
```

soit une variation de 0.01% du pas, l’erreur disparaît purement et simplement !

Question 1 (•••) Pouvez-vous contrôler la chance et expliquer comment déterminer au voisinage de $h = 10^{-12}$ les valeurs du pas susceptibles d'annuler l'erreur et inversement de générer l'erreur la plus élevée possible ? (Solution p. 43.)

Solution des exercices

Exercices essentiels

Le cercle unité La fonction $(x_1, x_2) \in \mathbb{R}^2 \mapsto x_1^2 + x_2^2 - 1$ est continûment différentiable et la dérivée partielle $\partial_{x_2} f(x_1, x_2) = 2x_2$ est non nulle sur le cercle unité sauf quand $(x_1, x_2) = (1, 0)$ ou $(x_1, x_2) = (-1, 0)$. On peut donc appliquer le théorème des fonctions implicites dans un voisinage de tout point $(x_{10}, x_{20}) \in C$ à l'exception de ces deux points. On peut déterminer directement que $x_1^2 + x_2^2 = 1$ est équivalent à $x_2 = \pm \sqrt{1 - x_1^2}$; si $x_{20} > 0$, $\psi(x_1) := \sqrt{1 - x_1^2}$ est donc l'unique solution de $x_1^2 + x_2^2 = 1$ telle que $(x_1, x_2) \in]-1, 1[\times]0, +\infty[$. Quand $x_{20} < 0$, $\psi(x_1) := -\sqrt{1 - x_1^2}$ est l'unique solution de $x_1^2 + x_2^2 = 1$ telle que $(x_1, x_2) \in]-1, 1[\times]-\infty, 0[$. Dans les deux cas, $\psi'(x_1) = -(\partial_{x_2} f(x_1, x_2))^{-1} \cdot \partial_{x_1} f(x_1, x_2)$ où $x_2 = \psi(x_1)$, donc dans le premier cas on a

$$\psi'(x_1) = -(2\psi(x_1))^{-1}(2x_1) = -\frac{x_1}{\sqrt{1 - x_1^2}}$$

et dans le second

$$\psi'(x_1) = -(2\psi(x_1))^{-1}(2x_1) = \frac{x_1}{\sqrt{1 - x_1^2}}.$$

Abscisse curviligne On considère l'équation

$$F(x, s) := \int_c^x \|f'(t)\| dt - s = 0$$

où F est définie sur l'ouvert $]a, b[\times \mathbb{R} \subset \mathbb{R} \times \mathbb{R}$. La dérivées partielles de F existent et vérifient

$$\partial_x F(x, s) = \|f'(x)\| \neq 0 \text{ et } \partial_s F(s, x) = -1.$$

Les dérivées partielles étant continues, F est continûment différentiable. De plus, la différentielle partielle de F par rapport à x est inversible. On a également $F(c, 0) = 0$. Par conséquent, dans un voisinage $U \times V$ ouvert de $(c, 0) \in \mathbb{R}^2$, l'équation $F(x, s) = 0$ détermine de façon unique x comme une fonction différentiable de s .

Courbes de niveau Avec les coordonnées (w, z) , l'appartenance d'un point P à la courbe de niveau C est caractérisée par

$$f(x_1, x_2) - c = f(wu_1 + zv_1, wu_2 + zv_2) - c = 0.$$

L'expression est continûment différentiable par rapport au couple (w, z) et

$$\begin{aligned} \partial_z (f(wu_1 + zv_1, wu_2 + zv_2) - c) &= df(x_1, x_2) \cdot (v_1, v_2) \\ &= \left\langle \nabla f(x_1, x_2), \frac{\nabla f(x_{10}, x_{20})}{\|\nabla f(x_{10}, x_{20})\|} \right\rangle \end{aligned}$$

Cette dérivée partielle est égale à $\|\nabla f(x_{10}, x_{20})\| > 0$ en (x_{10}, x_{20}) . Le gradient étant continu, cette dérivée partielle est inversible dans un voisinage de (x_{10}, x_{20}) et le théorème des fonctions implicites (p. 3) est donc applicable : localement, l'appartenance d'un point P à C peut être caractérisé par une relation fonctionnelle de la forme $z = \psi(w)$.

Détermination de l'angle On remarque qu'on ne peut pas utiliser directement le théorème des fonctions implicite sur l'équation définissant les déterminations de l'angle, car l'équation est à valeurs dans \mathbb{R}^2 mais nous souhaitons la résoudre par rapport à une variable scalaire. Mais cette équation est redondante car ses deux membres sont de norme 1 ; si u_0 et θ_0 en sont solutions, localement u et θ en seront solutions si et seulement si u et $(\cos \theta, \sin \theta)$ sont colinéaires, c'est-à-dire si et seulement si

$$f(u_1, u_2, \theta) := u_1 \sin \theta - u_2 \cos \theta = 0.$$

Nous pouvons alors appliquer le théorème des fonctions implicites (p. 3) à cette équation. On a

$$\partial_{u_1} f(u, \theta) = \sin \theta, \quad \partial_{u_2} f(u, \theta) = -\cos \theta,$$

et

$$\partial_\theta f(u, \theta) = u_1 \cos \theta + u_2 \sin \theta = \langle u, (\cos \theta, \sin \theta) \rangle.$$

Les expressions sont continues et au point d'intérêt, $\partial_\theta f(u_0, \theta) = \|u_0\| > 0$, ce qui est donc encore localement vrai par continuité. La différentielle de la fonction implicite $\theta = \Theta(u)$ est donnée par

$$d\Theta(u) = -(\partial_\theta f(u, \Theta(u)))^{-1} \cdot (\partial_u f(u, \Theta(u))),$$

donc

$$\nabla \Theta(u_0) = -\frac{\nabla_u f(u_0, \theta_0)}{\partial_\theta f(u_0, \theta_0)} = \frac{1}{\|u_0\|} \begin{bmatrix} -\sin \theta_0 \\ \cos \theta_0 \end{bmatrix}.$$

Coordonnées polaires La fonction f est continûment différentiable et sa matrice jacobienne est donnée par

$$J_f(r, \theta) = \begin{bmatrix} \cos \theta & -r \sin \theta \\ \sin \theta & r \cos \theta \end{bmatrix}.$$

Comme $\det J_f(r, \theta) = r > 0$, f est un C^1 -difféomorphisme local. Par contre, comme $f(r, \theta) = f(r, \theta + 2\pi)$, f n'est pas injective, elle n'est donc pas un C^1 -difféomorphisme global.

Représentation des entiers Oui, car l'écriture d'un tel entier n en binaire va demander au plus 32 bits non nuls b_i pour être décrit :

$$n = (-1)^s \times b_0 b_1 \dots b_{31}.$$

Si $b_0 = b_1 = \dots = b_{p-1} = 0$ et $b_p = 1$, il sera donc de la forme

$$n = (-1)^s \times (1.b_p \dots b_{31} 000 \dots) \times 2^e,$$

Nombres réels et doubles Le nombre 1.0 s'écrit en base 2 sous la forme

$$1.0 = (-1)^0 \times (1.000 \dots) \times 2^0,$$

et 0.5 sous la forme

$$0.5 = (-1)^0 \times (1.000 \dots) \times 2^{-1}.$$

Ils sont donc des doubles. Par contre, les développements en base 2 de $2/3$ et 0.1 sont périodiques, donc nécessiterait une mantisse de taille infinie pour être décrits exactement ; ils ne sont pas des doubles.

π contre pi Le nombre π est dans l'intervalle $[2^1, 2^2[$, donc sa représentation sous forme de double **pi** en base 2 est de la forme

$$\text{pi} = (1.??? \dots ???) \times 2^1.$$

(avec 52 points d'interrogations représentant 0 ou 1). Dans cet intervalle, l'espace entre deux doubles consécutifs est de $2^{-52} \times 2^1 = 2\epsilon$. L'erreur est donc bornée par $\approx 4.4 \times 10^{-16}$ (deux fois moins si on arrondit au plus proche).

Constantes de la Physique En utilisant la formule $|[x] - x| \leq \epsilon|x|$, on obtient

$$|[N] - N| \leq 1.4 \times 10^8 \text{ et } |[h] - h| \leq 1.5 \times 10^{-49}.$$

WTF Python ? Ni 0.1, ni 0.2 ni 0.3 ne sont représentés exactement par des doubles. Il est tout à fait possible que dans le calcul de

$$0.1 + 0.2 = [[0.1] + [0.2]]$$

les arrondis $[0.1]$ et $[0.2]$ soient légèrement supérieurs aux nombres réels exacts et que l'addition de ces nombres arrondis produise un nombre plus proche d'un double strictement supérieur à $[0.3]$ que de $[0.3]$ lui-même.

WTF Python, vraiment ? On remarque qu'en base 2, on a

$$\begin{aligned} 0.1 &= 0.00011001100\dots = (1.100110011001\dots 1001|1001\dots) \times 2^{-4} \\ 0.2 &= 0.00110011001\dots = (1.100110011001\dots 1001|1001\dots) \times 2^{-3} \\ 0.3 &= 0.01001100110\dots = (1.001100110011\dots 0011|0011\dots) \times 2^{-2} \end{aligned}$$

où la barre verticale indique la limite entre le 52-ème et le 53-ème bit après le point. Si $[\cdot]$ est l'arrondi au double le plus proche, on a donc

$$\begin{aligned} [0.1] &= (1.100110011001\dots 1010|0000\dots) \times 2^{-4} \\ [0.2] &= (1.100110011001\dots 1010|0000\dots) \times 2^{-3} \\ [0.3] &= (1.001100110011\dots 0011|0000\dots) \times 2^{-2} \end{aligned}$$

et par conséquent

$$\begin{aligned} [0.1] + [0.2] &= (001.100110011001\dots 10011010|) \times 2^{-4} \\ &\quad + (011.001100110011\dots 10110100|) \times 2^{-4} \\ &= (100.110011001100\dots 01001110|) \times 2^{-4} \\ &= (1.001100110011\dots 010011|10) \times 2^{-2} \end{aligned}$$

Le nombre $[0.1] + [0.2]$ est exactement à mi-distance des doubles

$$(1.001100110011\dots 010011|) \times 2^{-2} \text{ et } (1.001100110011\dots 010100|) \times 2^{-2}.$$

Dans ce cas, Python 3 semble appliquer la règle *round-to-nearest, ties-to-nearest-even* de l'IEEE754 (IEEE Task P754 (1985)) qui préfère le double se terminant par un 0 (entier pair) et qui est ici le plus grand des deux doubles possibles. On a donc

$$\begin{aligned} [0.3] &= (1.001100110011\dots 0011|) \times 2^{-2} \\ [[0.1] + [0.2]] &= (1.001100110011\dots 0100|) \times 2^{-2} \end{aligned}$$

Les deux nombres sont différents, leur écart est $2^{-52} \times 2^{-2} \approx 5.56 \times 10^{-17}$, qui justifie l'écart observé dans la représentation décimale simplifiée affichée dans l'interpréteur ($\approx 4 \times 10^{-17}$).

Non-associativité de l'addition On a $[1.0] = 1.0$ et $[\varepsilon/4] = \varepsilon/4$ car 1.0 et $\varepsilon/4$ appartiennent à \mathbb{D} . Par conséquent,

$$1.0 + \varepsilon/4 = [1.0 + \varepsilon/4] = 1.0,$$

car $1.0 + \varepsilon/4$ est encadré par les doubles consécutifs 1.0 et $1.0 + \varepsilon$ et que 1.0 est plus proche. On a donc

$$(((1.0 + \varepsilon/4) + \varepsilon/4) + \varepsilon/4) + \varepsilon/4 = 1.0.$$

D'autre part, $\varepsilon/4 + \varepsilon/4 = [\varepsilon/2] = \varepsilon/2$, donc $\varepsilon/4 + \varepsilon/2 = [3\varepsilon/4] = 3\varepsilon/4$ et donc $\varepsilon/4 + 3\varepsilon/4 = [\varepsilon] = \varepsilon$. Par conséquent,

$$1 + (\varepsilon/4 + (\varepsilon/4 + (\varepsilon/4 + \varepsilon/4))) = [1 + \varepsilon] = 1 + \varepsilon.$$

Les deux valeurs étant différentes, l'opérateur $+$ n'est donc pas associatif. Il est donc nécessaire de préciser ce que l'on entend par $x + y + z$. Livrons-nous à une expérience en Python :

```
>>> eps = 2**(-52)
>>> e4 = eps / 4
>>> s1 = 1.0 + e4 + e4 + e4 + e4
>>> s2 = (((1.0 + e4) + e4) + e4) + e4
>>> s3 = 1.0 + (e4 + (e4 + (e4 + e4)))
>>> print_exact_number(s1)
1
>>> print_exact_number(s2)
1
>>> print_exact_number(s3)
1.00000000000000002220446049250313080847263336181640625
```

Il semble donc que par défaut l'opérateur $+$ en Python associe à gauche, c'est-à-dire interprète $x + y + z$ comme $(x + y) + z$.

Cinématique d'un robot manipulateur

Question 1 Remarquons tout d'abord que $r = \sqrt{x^2 + y^2}$ ne dépend que de la valeur de θ_2 , pas de celle θ_1 puisque $(x, y) = f(\theta_1, \theta_2)$ est obtenu par rotation d'un angle θ_1 de $f(0, \theta_2)$. On a donc

$$r = \sqrt{(\ell_1 + \ell_2 \cos \theta_2)^2 + (\ell_2 \sin \theta_2)^2} = \sqrt{\ell_1^2 + \ell_2^2 + 2\ell_1 \ell_2 \cos \theta_2}.$$

L'ensemble des valeurs de cette fonction de θ_2 est l'intervalle $[|\ell_1 - \ell_2|, \ell_1 + \ell_2]$. Une fois θ_2 sélectionné (modulo 2π) pour atteindre la valeur r , il est évident par rotation que l'on peut trouver un angle θ_1 tel que $f(\theta_1, \theta_2) = (x, y)$. Pour des raisons de symétrie, si θ est un angle de (x, y) et (θ_1, θ_2) convient, alors $(2\theta - \theta_1, -\theta_2)$ également. Ces deux paires sont différentes si le bras manipulateur n'est ni totalement déplié, ni totalement plié.

Pour résumer : les points (x, y) tels que $r < |\ell_1 - \ell_2|$ ou $\ell_1 + \ell_2 < r$ ne peuvent pas être atteints. Les points vérifiant les égalités correspondantes sont associés à exactement un jeu de coordonnées articulaires (modulo 2π). Et dans le cas restant, dans

$$V = \{(x, y) \in \mathbb{R}^2 \mid |\ell_1 - \ell_2| < \sqrt{x^2 + y^2} < \ell_1 + \ell_2\},$$

plusieurs coordonnées articulaires (modulo 2π) peuvent correspondre.

Question 2 On rappelle qu'avec les notations $s_1 = \sin \theta_1$, $s_{12} = \sin(\theta_1 + \theta_2)$, $c_1 = \cos \theta_1$ et $c_{12} = \cos(\theta_1 + \theta_2)$, on a

$$J_f(\theta_1, \theta_2) = \begin{bmatrix} -\ell_1 s_1 - \ell_2 s_{12} & -\ell_2 s_{12} \\ \ell_1 c_1 + \ell_2 c_{12} & \ell_2 c_{12} \end{bmatrix}.$$

Comme soustraire à la première colonne la seconde ne change pas le rang de cette matrice, la matrice jacobienne est inversible si et seulement si

$$\begin{vmatrix} -\ell_1 s_1 & -\ell_2 s_{12} \\ \ell_1 c_1 & \ell_2 c_{12} \end{vmatrix} = -\ell_1 \ell_2 (s_1 c_{12} - c_1 s_{12}) = 0,$$

soit $\ell_1 \ell_2 \sin(\theta_1 + \theta_2 - \theta_1) = \ell_1 \ell_2 \sin \theta_2 = 0$. La matrice jacobienne de f est donc inversible à moins que θ_2 soit égal à 0 modulo π (bras totalement déplié ou totalement plié), soit

$$U = \mathbb{R}^2 \setminus (\mathbb{R} \times \pi\mathbb{Z}).$$

En coordonnées cartésiennes, cela correspond aux points (x, y) à distance minimale $|\ell_1 - \ell_2|$ ou maximale $\ell_1 + \ell_2$ de l'origine. Les points à une distance intermédiaire

$$|\ell_1 - \ell_2| < \sqrt{x^2 + y^2} < \ell_1 + \ell_2$$

soit V , correspondent à une matrice jacobienne $J_f(q)$ inversible quel que soit l'antécédent q de (x, y) par f .

Question 3 Si $q_0 = (\theta_{10}, \theta_{20}) \in U$, comme la matrice jacobienne de f est inversible sur U , le théorème d'inversion locale (p. 9) s'applique : la fonction f est un difféomorphisme sur un voisinage ouvert $V \subset U$ de q_0 , d'inverse $g : W \rightarrow U$ défini sur l'ouvert $W = f(V)$. Si γ est une trajectoire continue

$$\gamma : t \in [0, 1] \mapsto (x(t), y(t)) \in \mathbb{R}^2$$

dans l'espace cartésien dont l'image est incluse dans $W = f(V)$ et telle que $f(q_0) = (x(0), y(0))$, alors $f(\gamma_q(t)) = \gamma(t)$ si et seulement si $\gamma_q(t) = g(\gamma(t))$.

Question 4 Si γ est différentiable, comme $\gamma_q(t) = g(\gamma(t))$ et que g est continûment différentiable, γ_q l'est également. L'application de la règle de différentiation en chaîne à $\gamma = f \circ \gamma_q$ fournit

$$\gamma'(t) = df(\gamma_q(t)) \cdot \gamma'_q(t),$$

soit

$$\begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix} = J_f(\theta_1, \theta_2) \times \begin{bmatrix} \dot{\theta}_1 \\ \dot{\theta}_2 \end{bmatrix} \quad \text{ou encore} \quad \begin{bmatrix} \dot{\theta}_1 \\ \dot{\theta}_2 \end{bmatrix} = J_f(\theta_1, \theta_2)^{-1} \times \begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix}$$

Déformations

Question 1 Par le théorème des accroissements finis, si x et y appartiennent à U , comme par convexité $[x, y] \subset U$, on a

$$\|H(x) - H(y)\| \leq \kappa \|x - y\|.$$

Par conséquent,

$$\begin{aligned}\|T(x) - T(y)\| &= \|x + H(x) - (y + H(y))\| \\ &\geq \|x - y\| - \|H(x) - H(y)\| \\ &\geq (1 - \kappa)\|x - y\|\end{aligned}$$

et donc si $T(x) = T(y)$, $x = y$: T est bien injective.

Question 2 La différentielle $dT(x)$ de T en x est une application de \mathbb{R}^n dans \mathbb{R}^n de la forme

$$dT(x) = I + dH(x).$$

Comme \mathbb{R}^n est ouvert et que la fonction $h \mapsto dH(x) \cdot h$ a pour différentielle en tout point y de \mathbb{R}^n la fonction $dH(x)$, la fonction linéaire $h \mapsto dT(x) \cdot h$ est une perturbation de l'identité ; elle est donc injective, et inversible car elle est linéaire de \mathbb{R}^n dans \mathbb{R}^n . Les hypothèses du théorème d'inversion locale (p. 9) sont donc satisfaites en tout point x de U . La fonction T est donc un difféomorphisme local d'un voisinage ouvert V_x de x sur $W_x = T(V_x)$ qui est ouvert. Clairement,

$$T(U) = f\left(\bigcup_{x \in U} V_x\right) = \bigcup_{x \in U} f(V_x)$$

et par conséquent $T(U)$ est ouvert. La fonction T est injective et surjective de T dans $T(U)$, donc inversible. En tout point y de $T(U)$, il existe $x \in U$ tel que $T(x) = y$, et un voisinage ouvert V_x de x tel que T soit un difféomorphisme local de V_x sur l'ouvert $W_x = T(V_x)$; la fonction T^{-1} est donc continûment différentiable dans un voisinage de y . C'est par conséquent un difféomorphisme global de U dans $T(U)$.

Valeurs propres d'une matrice

Question 1 Le nombre $\lambda \in \mathbb{R}$ est une valeur propre de $A \in \mathbb{R}^{n \times n}$ si et seulement si

$$p(A, \lambda) = \det(A - \lambda I) = 0.$$

La fonction p est un polynôme dont les variables sont λ et les coefficients a_{ij} de la matrice A ; cette fonction est donc continûment différentiable. Si λ_0 est une racine simple de $\lambda \mapsto p(A_0, \lambda)$, on a $\partial_\lambda p(A_0, \lambda_0) \neq 0$ et donc par continuité, dans un voisinage de λ_0 et de A_0 , $\partial_\lambda p(A, \lambda) \neq 0$.

Par le théorème des fonctions implicites (p. 3), il existe donc localement une unique valeur propre réelle λ associée à A , et elle est continûment différentiable – et a fortiori continue – par rapport aux coefficients de A .

Différences finies – erreur d’arrondi

Question 1 Tout d’abord, pour $x = 1$ et un pas de l’ordre de $h = 10^{-12}$, l’erreur faite en approximant $\exp(x)$ par $1 + x$ sera de l’ordre de

$$\exp''(0) \times (10^{-12})^2 / 2 = 5 \times 10^{-25},$$

très petit par rapport au ε machine

$$\varepsilon = 2^{-52} \approx 2.220446049250313 \times 10^{-16}.$$

Dans la suite, on fera donc les calculs en faisant comme si l’on avait $\exp(x) = 1 + x$. Le numérateur de $\text{FD}(\exp, 1.0, h)$ évalue donc le nombre

$$[[[1.0] + [h]] - [1.0]] = [[1.0 + [h]] - 1.0]$$

Pour un h entre 0 et 1, le mieux qui puisse arriver est d’avoir un multiple de ε , car si c’est le cas, on a $[h] = h$, puis $[1.0 + h] = 1.0 + h$,

$$[[[1.0] + [h]] - [1.0]] = [h]$$

et finalement

$$\left[\frac{[[[1.0] + [h]] - [1.0]]}{[h]} \right] = [1.0] = 1.0,$$

soit la valeur exacte de $\exp'(0)$. Pour obtenir une valeur de h de ce type proche de $h = 10^{-12}$, il suffit de calculer

```
>>> eps = 2**-52
>>> floor(1e-12 / eps) * eps
9.99866855977416e-13
```

A l’inverse, pour maximiser l’erreur faite dans l’estimation de $1 + h$ par $[1.0 + [h]]$, il faut prendre un h de la forme $h = (k + 0.5) \times \varepsilon$ avec k entier ; on aura alors $[h] = h$, puis

$$|[1.0 + [h]] - (1.0 + h)| = \frac{\varepsilon}{2}$$

et donc

$$[[1.0 + [h]] - 1.0] \approx h \pm \frac{\varepsilon}{2}$$

soit au final

$$\left[\frac{[[[1.0] + [h]] - [1.0]]}{[h]} \right] \approx 1.0 \pm \frac{\varepsilon}{2h},$$

soit ici une erreur faite par $\text{FD}(\exp, 0.0, h)$ de l’ordre de

```
>>> 0.5 * eps / 1e-12
0.00011102230246251565
```

Pour trouver un tel h proche de 10^{-12} , il suffit de calculer

```
>>> (floor(1e-12 / eps) + 0.5) * eps
9.999778782798785e-13
```

Références

- Baydin, Atilim Gunes, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. 2015. “Automatic Differentiation in Machine Learning: A Survey.”
- Fousse, Laurent, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. 2007. “MPFR: A Multiple-Precision Binary Floating-Point Library with Correct Rounding.” *ACM Trans. Math. Softw.* 33 (2). <https://doi.org/10.1145/1236463.1236468>.
- Goldberg, David. 1991. “What Every Computer Scientist Should Know About Floating-Point Arithmetic.” *ACM Computing Surveys* 23 (1): 5–48. <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.22.6768>.
- IEEE Task P754. 1985. *ANSI/IEEE 754-1985, Standard for Binary Floating-Point Arithmetic*. New York, NY, USA: IEEE.
- Johansson, Fredrik, and others. 2013. *Mpmath: A Python Library for Arbitrary-Precision Floating-Point Arithmetic (Version 0.18)*.