Calcul Différentiel II

STEP, MINES ParisTech*

14 janvier 2020 (`#0b90acd`)

Table des matières

Théorème des fonctions implicites

Théorème des fonctions implicites

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

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

qui soit continûment différentiable et telle que la différentielle partielle $\partial_y f$ soit inversible en tout point de W. Si le point (x_0, y_0) de W vérifie $f(x_0, y_0) = 0$, alors il existe des voisinages ouverts U de x_0 et V de y_0 tels que $U \times V \subset W$ et une fonction implicite $\psi : U \to \mathbb{R}^m$, continûment différentiable, telle que pour tout $x \in U$ et tout $y \in V$,

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

De plus, la différentielle de ψ est donnée pour tout $x \in U$ par

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

Extensions

Il est possible d'affaiblir l'hypothèse concernant $\partial_y f$ en supposant uniquement celle-ci inversible en (x_0, y_0) au lieu d'inversible sur tout W. En effet, l'application

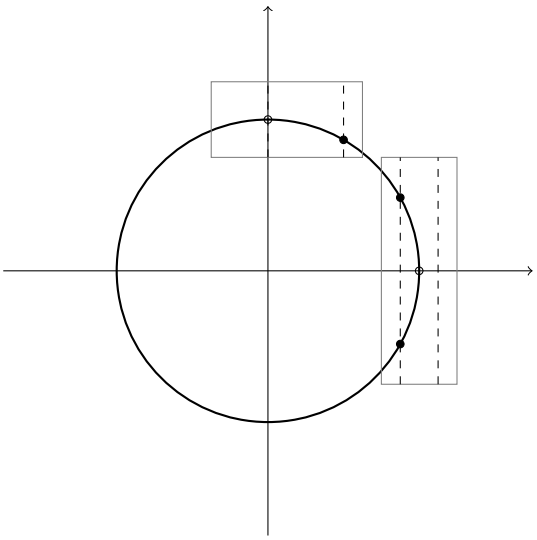

FIGURE 1 – Au voisinage de $(0,1)$, la fonction $x, y \mapsto x^2 + y^2 - 1$ satisfait les hypothèses du théorème des fonctions implicites. Au voisinage de $(1,0)$, cela n'est pas le cas ; on constate alors que même si x est arbitrairement proche de 1 et que l'on restreint la recherche des solutions y à un voisinage arbitrairement petit de 0, il peut exister 0 ou 2 solutions y.

qui a une application linéaire inversible $A : \mathbb{R}^m \to \mathbb{R}^m$ associe son inverse A^{-1} est définie sur un ouvert et continue [1]. Comme l'application linéaire $\partial_y f(x_0, y_0)$ est inversible et que l'application $\partial_y f$ est continue, il existe donc un voisinage ouvert de (x_0, y_0) contenu dans W où $\partial_y f$ est inversible. Nous retrouvons donc les hypothèses initiales du théorème, à ceci près qu'elle sont satisfaites dans un voisinage de (x_0, y_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 $f(x, \psi(x)) = 0$ étant satisfaite identiquement sur U et la fonction $x \in U \mapsto f(x, \psi(x))$ étant différentiable comme composée de fonctions différentiables, la règle de dérivation en chaîne fournit en tout point de U:

$$\partial_x f(x, \psi(x)) + \partial_y f(x, \psi(x)) \cdot d\psi(x) = 0.$$

1. **Continuité de l'inversion.** Une application linéaire de $\mathbb{R}^m \to \mathbb{R}^m$ est inversible si et seulement si le déterminant de la matrice $[A]$ qui la représente dans $\mathbb{R}^{m\times m}$ 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 applications linéaires inversibles de $\mathbb{R}^m \to \mathbb{R}^m$ 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

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

où $\mathrm{co}([A])$ 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.

On en déduit donc que

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

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 x suffisamment proche de x_0 construire une suite convergente d'approximations y_k, proches de y_0, dont la limite y sera solution de $f(x, y) = 0$.

L'idée de cette construction repose sur l'analyse suivante : si nous partons d'une valeur y_k proche de y_0 (a priori telle que $f(x, y_k) \neq 0$) et que nous recherchons une valeur y_{k+1} proche, qui soit une solution approchée de $f(x, y) = 0$ (meilleure que ne l'est y_k), comme au premier ordre

$$f(x, y_{k+1}) \approx f(x, y_k) + \partial_y f(x, y_k) \cdot (y_{k+1} - y_k),$$

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

$$y_{k+1} := y_k - (\partial_y f(x, y_k))^{-1} \cdot f(x, y_k)$$

vérifie $f(x, y_{k+1}) \approx 0$. On peut espérer que répéter ce processus en partant de y_0 détermine une suite convergente dont la limite soit une solution exacte y de $f(x, y) = 0$.

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 la *méthode de Newton modifiée*, qui n'utilise pas $\partial_y f(x, y_k)$ mais la valeur constante $\partial_y f(x_0, y_0)$, c'est-à-dire qui définit la suite

$$y_{k+1} := y_k - Q^{-1} \cdot f(x, y_k) \text{ où } Q = \partial_y f(x_0, y_0).$$

Cette définition par récurrence se réécrit sous la forme $y_{k+1} = \phi_x(y_k)$ où

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

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

$$d\phi_x(y) = I - Q^{-1} \cdot \partial_y f(x, y)$$

où I désigne la fonction identité. En écrivant que $\partial_y f(x, y)$ est la somme de $\partial_y f(x_0, y_0)$ et de $\partial_y f(x, y) - \partial_y f(x_0, y_0)$, on obtient

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

La fonction f étant supposée continûment différentiable, il existe un $r > 0$ tel que tout couple (x, y) tel que $\|x - x_0\| \leq r$ et $\|y - y_0\| \leq r$ appartienne à W et vérifie $\|\partial_y f(x, y) - Q\| \leq \kappa \|Q^{-1}\|^{-1}$ avec par exemple $\kappa = 1/2$, ce qui

entraîne $\|d\phi_x(y)\| \leq \kappa$. Par l'inégalité des accroissements finis, la restriction de ϕ à $\{y \in \mathbb{R}^m \mid \|y - y_0\| \leq r\}$ (que l'on continuera à noter ϕ_x) est κ-contractante:

$$\|\phi_x(y) - \phi_x(z)\| \leq \kappa \|y - z\|.$$

Par ailleurs,

$$\|\phi_x(y) - y_0\| \leq \|\phi_x(y) - \phi_x(y_0)\| + \|\phi_x(y_0) - \phi_{x_0}(y_0)\|.$$

On a d'une part

$$\|\phi_x(y) - \phi_x(y_0)\| \leq \kappa \|y - y_0\| \leq \kappa r$$

et d'autre part, par continuité de ϕ en (x_0, y_0), il existe un r' tel que $0 < r' < r$ et tel que si $\|x - x_0\| \leq r'$, alors $\|\phi_x(y_0) - \phi_{x_0}(y_0)\| \leq (1 - \kappa)r$. Pour de telles valeurs de x,

$$\|\phi_x(y) - y_0\| \leq \kappa r + (1 - \kappa)r = r.$$

L'image de la boule fermée $B = \{y \in \mathbb{R}^m \mid \|y - y_0\| \leq r\}$ par l'application ϕ_x est donc incluse dans B. Tant que $\|x - x_0\| \leq r'$, les hypothèses du théorème de point fixe de Banach sont donc satisfaites pour $\phi_x : B \to B$, ce qui montre l'existence et l'unicité de la fonction implicite ψ associée aux voisinages ouverts $U = B(x_0, r')$ et $V = B(y_0, r)$.

Montrons la continuité de la fonction implicite ψ. Soit x_1, x_2 deux points de V ; notons $y_1 = \psi(x_1)$ et $y_2 = \psi(x_2)$. Ces valeurs sont des solutions des équations de point fixe

$$y_1 = \phi_{x_1}(y_1) \text{ et } y_2 = \phi_{x_2}(y_2).$$

En formant la différence de y_2 et y_1, on obtient donc

$$\begin{aligned} \|y_2 - y_1\| &= \|\phi_{x_2}(y_2) - \phi_{x_1}(y_1)\| \\ &\leq \|\phi_{x_2}(y_2) - \phi_{x_2}(y_1)\| + \|\phi_{x_1}(y_1) - \phi_{x_2}(y_1)\|. \end{aligned}$$

La fonction ϕ_{x_2} étant κ-contractante, le premier terme du membre de droite de cette inégalité est majoré par $\kappa\|y_2 - y_1\|$, par conséquent

$$\|y_2 - y_1\| \leq \frac{1}{1 - \kappa} \|\phi_{x_1}(y_1) - \phi_{x_2}(y_1)\|.$$

L'application $x \mapsto \phi_x(y_1)$ étant continue en x_1, nous pouvons conclure que y_2 tend vers y_1 quand x_2 tend vers x_1 ; autrement dit : la fonction implicite ψ est continue en x_1.

Montrons finalement la différentiabilité de ψ en x_1. Pour cela, il suffit d'exploiter la différentiabilité de f en (x_1, y_1) où $y_1 = \psi(x_1)$. Elle fournit l'existence d'une fonction ε qui soit un $o(1)$ telle que

$$\begin{aligned} f(x, y) = f(x_1, y_1) &+ \partial_x f(x_1, y_1) \cdot (x - x_1) + \partial_y f(x_1, y_1) \cdot (y - y_1) \\ &+ \varepsilon((x - x_1, y - y_1))(\|x - x_1\| + \|y - y_1\|) \end{aligned}$$

On a par construction $f(x_1, y_1) = 0$; en prenant $y = \psi(x)$, on annule également $f(x, y) = 0$. En notant $P = \partial_x f(x_1, y_1)$ et $Q = \partial_y f(x_1, y_1)$, on obtient

$$\begin{aligned} \psi(x) = \psi(x_1) &- Q^{-1} \cdot P \cdot (x - x_1) \\ &- Q^{-1} \cdot P \cdot \varepsilon((x - x_1, \psi(x) - \psi(x_1))(\|x - x_1\| + \|\psi(x) - \psi(x_1)\|). \end{aligned}$$

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

$$\varepsilon_x(x - x_1) := \varepsilon((x - x_1, \psi(x) - \psi(x_1))$$

est un $o(1)$ du fait de la continuité de ψ en x_1. En choisissant x dans un voisinage suffisamment proche de x_1, on peut donc garantir que ce terme est arbitrairement petit, par exemple, tel que

$$\|Q^{-1} \cdot P\| \times \|\varepsilon_x(x - x_1)\| \leq \frac{1}{2},$$

ce qui permet d'obtenir

$$\|\psi(x) - \psi(x_1)\| \leq \|Q^{-1}P\| \times \|x - x_1\| + \frac{1}{2}\|x - x_1\| + \frac{1}{2}\|\psi(x) - \psi(x_1)\|$$

et donc

$$\|\psi(x) - \psi(x_1)\| \leq \alpha\|x - x_1\| \text{ avec } \alpha := 2\|Q^{-1}P\| + 1.$$

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

$$\|\psi(x) - \psi(x_1) - Q^{-1} \cdot P \cdot (x - x_1)\| \leq \|\varepsilon'_x(x - x_1)\| \times \|x - x_1\|.$$

où la fonction ε'_x est le $o(1)$ défini par

$$\varepsilon'_x(x - x_1) := (1 + \alpha) \times \|Q^{-1} \cdot P\| \times \|\varepsilon_x(x - x_1)\|,$$

ce qui prouve la différentiabilité de ψ en x_1 et conclut la démonstration. ■

Difféomorphisme

Une fonction $f : U \subset \mathbb{R}^n \to V \subset \mathbb{R}^n$, 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.

Inverse de la différentielle

Si $f : U \to V$ 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 d'inversion locale

Soit $f : U \subset \mathbb{R}^n \to \mathbb{R}^n$ continûment différentiable sur l'ouvert U et telle que $df(x)$ soit inversible en tout point x de U. Alors 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 $\phi : U \times \mathbb{R}^n \to \mathbb{R}^n$ 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 $\psi : B \to \mathbb{R}^m$ 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. ■

Analyse numérique

Les exemples utilisés dans cette section exploitent la librairie numérique Python NumPy ; assurons-nous tout de suite d'avoir importé tous ses symboles :

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

Introduction

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

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

approximation 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* décroit 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, liée à la représentation approchée des nombres réels utilisée par le programme informatique.

Arithmétique des ordinateurs

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 computer arithmetic" (Goldberg 1991)

Premier contact

Dans un interpréteur Python, la façon la plus simple d'afficher un nombre consiste à invoquer son nom ; par exemple

```
>>> pi
3.141592653589793
```

Cette information est non-ambiguë ; par là nous voulons dire que nous disposons d'assez d'information pour reconstituer le nombre initial:

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

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

```
>>> def all_digits(number):
...     print(f"{number:.100g}")
>>> all_digits(pi)
3.1415926535897931159979634685441851615905761718750
```

Demander 100 chiffres après la virgule est suffisant : seul 49 chiffres sont 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, nous voyons que

```
>>> import mpmath
>>> mpmath.mp.dps = 49; mpmath.mp.pretty = True
>>> +mpmath.pi
3.14159265358979323846264338327950288419716939937 5
```

et que les deux représentations ne sont identiques que jusqu'au 16ème chiffre.

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 sont stockés avec une réprésentation *binaire*. En d'autres termes ; au lieu d'utiliser une suite de chiffres décimaux $f_i \in \{0, 1, \ldots, 9\}$ pour représenter un nombre réel x comme

$$x = \pm(f_0.f_1 f_2 \ldots f_i \ldots) \times 10^e$$

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

$$x = \pm(f_0.f_1 f_2 \ldots f_i \ldots) \times 2^e.$$

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 $(1.f_1 f_2 \ldots f_i \ldots)$.

En calcul scientifique, les nombres réels sont le plus souvent approximés sous la forme de "doubles" [2]. Dans la bibliothèque standard Python, les doubles sont disponibles comme instances du type `float` – ou alternativement comme `float64` dans NumPy.

Un triplet de

— *bit de signe* : $s \in \{0, 1\}$,
— *exposant (biaisé)* : $e \in \{1, \ldots, 2046\}$ (11-bit),
— *fraction* : $f = (f_1, \ldots, f_{52}) \in \{0, 1\}^{52}$.

représente un double *normalisé*

$$x = (-1)^s \times 2^{e-1023} \times (1.f_1 f_2 \ldots f_{52}).$$

2. "Double" est un raccourci pour "format à virgule flottante de précision double", comme défini dans le standard IEEE 754, cf. (IEEE Task P754 1985). Un format de simple précision est aussi défini, qui utilise uniquement 32 bits ; NumPy le propose sous le nom `float32`. Après un abandon progressif des "singles" au profit des "doubles", plus précis et mieux supportés par le CPUs modernes, le format de simple précision revient en force avec le développement de l'usage des GPUs comme unités de calcul génériques.

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) et les nombres dits *dénormalisés*. Dans la suite, nous ne parlerons pas de ces cas particuliers.

Précision

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 le double le plus proche $[x]$. Une telle méthode (*arrondi-au-plus-proche*) totalement spécifiée [3] dans le standard IEE754 (IEEE Task P754 1985), ainsi que des modes alternatifs d'arrondi (arrondis "orientés").

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 par la représentation de x par $[x]$. 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
>>> all_digits(after_one)
1.0000000000000002220446049250313080847263336181640625
>>> eps = after_one - 1.0
>>> all_digits(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.

```
>>> all_digits(finfo(float).eps)
2.220446049250313080847263336181640625e-16
```

Alternativement, l'examen de la structure des doubles normalisés fournit directement la valeur de ε : la fraction du nombre après 1.0 est $(f_1, f_2, \ldots, f_{51}, f_{52}) = (0, 0, \ldots, 0, 1)$, donc $\varepsilon = 2^{-52}$, un résultat confirmé par le code suivant :

```
>>> all_digits(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, pour n'importe quelle méthode d'arrondi raisonnable, la structure des doubles normalisés fournit :

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

3. Il faut préciser comme l'opération se comporte quand le réel est équidistant de deux doubles, comment les "nombres spéciaux" (`inf`, `nan`, ...) sont traités, etc. Autant de "détails" dont nous ne nous préoccuperons pas dans la suite.

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.

Chiffres significatifs

L'erreur relative détermine combien de chiffres décimaux sont significatifs dans la meilleure approximation d'un nombre réel par un double. Considérons la représentation de $[x]$ en notation scientifique :

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

On dira qu'elle est *significative jusqu'au n-ième chiffre* si

$$|x - [x]| \leq \frac{10^{e-(n-1)}}{2}.$$

D'autre part, la borne d'erreur sur $[x]$ fournit

$$|x - [x]| \leq \frac{\varepsilon}{2}|x| \leq \frac{\varepsilon}{2} \times 10^{e+1}.$$

Ainsi, la précision souhaitée est obtenue tant que

$$n \geq -\log_{10} \varepsilon = 52 \log_{10} 2 \approx 15.7.$$

Par conséquent, les doubles fournissent une approximation des nombres réels avec environ 15 ou 16 chiffres significatifs.

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 représentée exactement comme des fonctions opérant sur des doubles. Le mieux que nous puissions espérer est d'avoir des approximations *correctement arrondies*. Une approximation $[f]$ d'une fonction f de n variables est correctement arrondie si pour tout n-uplet $(x_1, \dots, x_n)$, on a

$$[f](x_1, \dots, x_n) = [f([x_1], \dots, [x_n])].$$

Autrement dit, tout se passe comme si le calcul de $[f](x_1, \dots, x_n)$ était effectué de la façon suivante : approximation au plus proche des arguments par des doubles, calcul **exact** de f sur ces arguments, et approximation de la valeur produite au plus proche par un double. Ou encore:

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

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 correctement arrondis est un problème difficile (cf. Fousse et al. (2007)).

Différences finies

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,

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

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*) – controllé 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

$$\mathrm{FD}(f, x, h) = \left[\frac{[[f]([x] + [h]) - [f]([x])]}{[h]}\right].$$

ou de façon équivalent en Python par:

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

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 $h \mapsto \mathrm{FD}(\exp, 0, h)$ 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 :

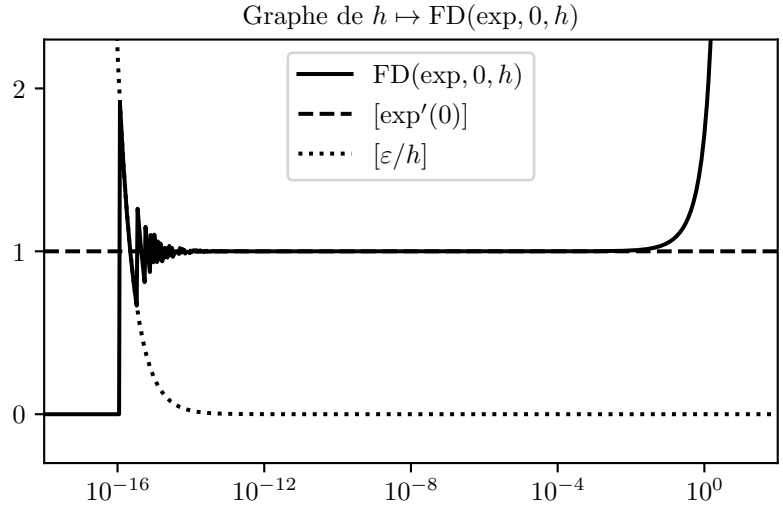

FIGURE 2 – Valeurs de la différence avant

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

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

— 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

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

— Si $\varepsilon \ll h \ll 1$, alors $[1 + h] = 1 + h \pm \varepsilon(1 + h)$ (le symbole $\pm$ est utilisé ici pour défini un intervalle de confiance [4]). Et donc

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

et

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

et par conséquent

$$\mathrm{FD}(\exp_1, 0, h) = \exp_1'(0) \pm \frac{\varepsilon}{h} \pm \varepsilon',\ \varepsilon' \ll \frac{\varepsilon}{h}.$$

Si l'on revient à $\mathrm{FD}(\exp, 0, h)$ et si l'on exploite des échelles log-log pour représenter l'erreur totale, on peut clairement distinguer la region ou 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.

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

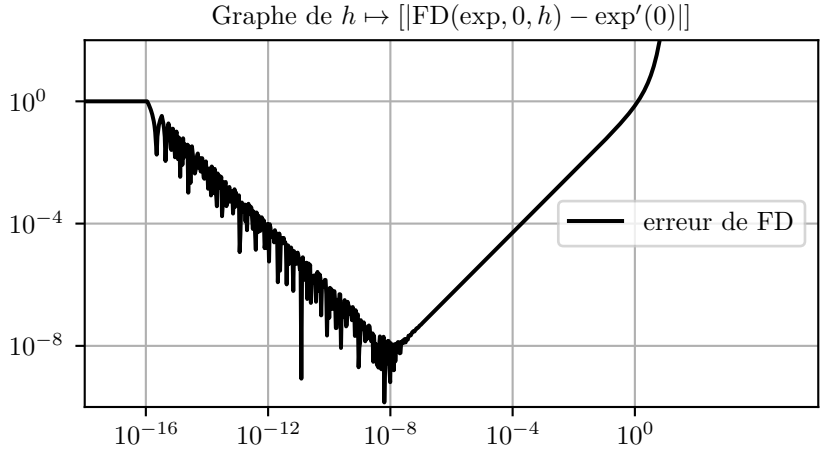

FIGURE 3 – Erreur de la différence avant

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 $f(x-h)$:

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

et

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

On en déduit

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

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

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

ou de façon équivalente en Python:

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

Le graphe d'erreur associé à la différence centrale 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.

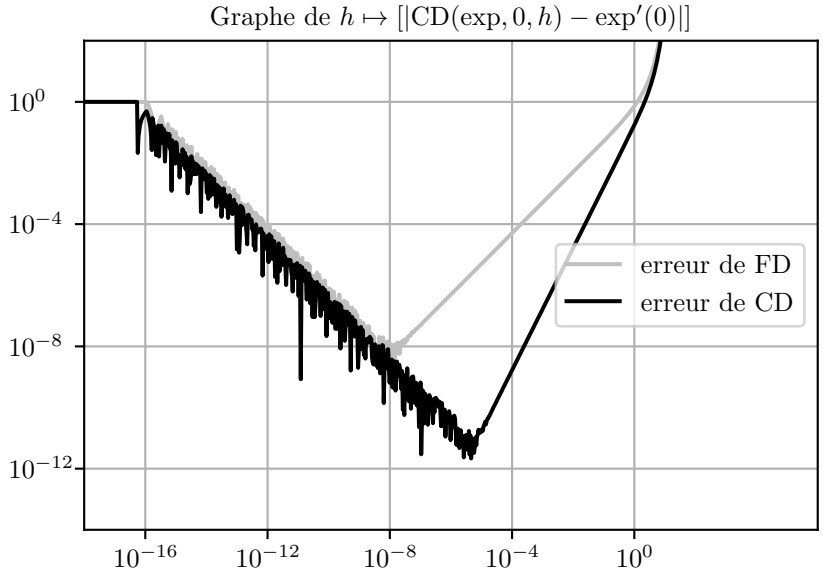

FIGURE 4 – Erreur de la différence centrée

Différentiation automatique

Introduction

La différentiation automatique désigne une famille de méthodes numériques permettant de calculer dérivées et différentielles de fonctions numériques. Elle se positionne comme une alternative aux algorithmes de différences finies. Ces méthodes ont l'avantage majeur d'éliminer quasi-totalement les erreurs d'arrondis – l'erreur est aussi faible que dans une dérivation symbolique "manuelle" des expressions utilisées dans le calcul de la fonction – et ce sans réglage délicat de paramètres.

Selon le langage informatique utilisé pour implémenter les fonctions numériques (C, Fortran, Python, langages "embarqués", etc.), différentes méthodes permettent de mettre en oeuvre la différentiation automatique. Le typage dynamique (ou *duck typing*) de Python permet de mettre en oeuvre simplement le *tracing* des fonctions numériques – l'enregistrement des opérations du calcul effectuées par une fonction lors de son exécution. A partir du graphe de calcul ainsi construit, les différentielles peuvent être calculées mécaniquement par la règle de différentiation en chaîne à partir des différentielles des opérations élémentaires.

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 additionnables [5].

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
```

5. 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".

```
3.0
```

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 noeud 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 noeud issu d'une opération devra mémoriser quelle opération a été appliquée, et quels étaient les arguments de l'opération (eux-mêmes des noeuds). 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 noeuds ; 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 noeuds. 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 foncton `sin` et l'opérateur `*` compatible avec la gestion de noeuds, 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 que 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 noeuds, puis appelle la fonction et renvoie le noeud 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 noeuds 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))
```

Calcul automatique des dérivées

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 noeud final produit par ce procédé extraire l'ensemble des noeuds amont, qui représentent les arguments utilisés dans le calcul de la valeur finale. Puis, pour préparer le calcul de la différentielle, nous ordonnerons les noeuds de telle sorte que les arguments d'une fonction apparaissent toujours avant la valeur qu'elle produit. L'implémentation suivante, relativement naïve [6], réalise cette opération:

```
def find_and_sort_nodes(end_node):
    todo = [end_node]
    nodes = []
```

6. 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.

```
    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 noeud en noeud, en se basant sur la règle de différentiation en chaîne ; ces variations intermédiaires sont stockées dans l'attribut `d_value` des noeuds 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
                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" [7] 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.

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'acquérir si besoin une perspective plus large sur le sujet.

Exercices

Cinématique d'un robot manipulateur

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

$$\left| \begin{array}{rcl} x & = & \ell_1 \cos\theta_1 + \ell_2 \cos(\theta_1 + \theta_2) \\ y & = & \ell_1 \sin\theta_1 + \ell_2 \sin(\theta_1 + \theta_2) \end{array} \right.$$

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) ? **(?)**

Question 2 Déterminer l'ensemble U des $q \in \mathbb{R}^2$ tel que la matrice $J_f(q)$ est inversible et comparer $f(U)$ avec V. **(?)**

7. "autograd" ou "autodiff" sont des termes plus ou moins génériques qu'utilisent de nombreuses librairies.

Question 3 Soit $\tau > 0$; on considère une trajectoire continue

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

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 τ est suffisamment petit, il existe une unique fonction continue $\gamma_q : [0, \tau] \to \mathbb{R}^2$ 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). **(?)**

Question 4 Montrer que si γ est différentiable, γ_q également. Déterminer la relation entre

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

(?)

Déformations

Soit U un ouvert convexe de $\mathbb{R}^n$ et $T : U \subset \mathbb{R}^n \to \mathbb{R}^n$ 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. **(?)**

Question 2 Montrer que l'image $V = T(U)$ est un ouvert et que T est difféomorphisme global de U sur V. **(?)**

Valeurs propres d'une matrice

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. **(?)**

Inversion de matrice

Question 1 Montrer que le produit matriciel

$$(A, B) \in \mathbb{R}^{m \times n} \times \mathbb{R}^{n \times p} \to A \times B \in \mathbb{R}^{m \times p}$$

est une application continûment différentiable. **(?)**

Question 2 Montrer que l'application qui à une matrice inversible $A \in \mathbb{R}^{n\times n}$ associe son inverse A^{-1} est définie sur un ouvert de $\mathbb{R}^{n\times n}$, continûment différentiable et que

$$d(A^{-1}) \cdot H = -A^{-1} \times H \times A^{-1}.$$

(?)

Méthode de Newton

L'analyse de la preuve du théorème des fonctions implicites nous a conduit à considérer la méthode de Newton modifiée, associée à la construction du point fixe de

$$\phi_x : y \mapsto y - (\partial_y f(x_0, y_0))^{-1} \cdot f(x, y).$$

Si la fonction f est deux fois continûment différentiable, il n'est plus nécessaire de modifier la méthode de Newton pour prouver le théorème : on peut exploiter directement la fonction

$$\phi_x : y \mapsto y - (\partial_y f(x, y))^{-1} \cdot f(x, y)$$

Montrer que ϕ_x est différentiable dans un voisinage de (x_0, y_0) et vérifier que $d\phi_x(y)$ est nul si $f(x, y) = 0$. **(?)**

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

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

on a également

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

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

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

soit une variation de 0.01% du pas, l'erreur disparaît purement et simplement ! 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 de générer l'erreur la plus élevée possible ? **(?)**

Solution des exercices

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

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

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 s'applique : la fonction f est un difféomorphisme sur un voisinage ouvert $V \subset U$ de q_0, d'inverse $g : W \to U$ défini sur l'ouvert $W = f(V)$. Si γ est une trajectoire continue

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

dans l'espace cartésien, dont l'image est incluse dans $f(U)$ et telle que $f(q_0) = (x(0), y(0))$, tant que τ est suffisamment petit pour que par continuité $\gamma(t)$ appartienne à W, 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 function $dH(x)$, 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 sont donc satisfaites en tout point x de U. La fonction f est donc un difféomorphisme local d'un voisinage ouvert V_x de x sur $W_x = f(V_x)$ qui est ouvert. Clairement,

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

et par conséquent $f(U)$ est ouvert. La fonction f est injective et surjective de U dans $f(U)$, donc inversible. En tout point y de $f(U)$, il existe $x \in U$ tel que $f(x) = y$, et un voisinage ouvert V_x de x tel que f soit un difféomorphisme local de V_x sur l'ouvert $W_x = f(V_x)$; la fonction f^{-1} est donc continûment différentiable dans un voisinage de y. C'est par conséquent un difféomorphisme global de U dans $f(U)$.

Valeurs propres d'une matrice

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, 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.

Inversion de matrice

Question 1 Pour tout $i \in \{1, \ldots, m\}$ et $k \in \{1, \ldots, p\}$, on a

$$[A \times B]_{ik} = \sum_{j=1}^{n} a_{ij} \times b_{jk}.$$

Par conséquent les dérivées partielles du coefficient (i, k) de $A \times B$ par rapport à $a_{\alpha\beta}$ existent et sont données par

$$\frac{\partial [A \times B]_{ik}}{\partial a_{\alpha\beta}} = \left| \begin{array}{ll} b_{\beta k} & \text{si } i = \alpha, \\ 0 & \text{sinon.} \end{array} \right.$$

et de façon similaire on a

$$\frac{\partial [A \times B]_{ik}}{\partial b_{\beta\gamma}} = \left| \begin{array}{ll} a_{i\beta} & \text{si } k = \gamma, \\ 0 & \text{sinon.} \end{array} \right.$$

Ces expressions sont des fonctions continues du couple (A, B). L'application produit de matrices est donc continûment différentiable.

Question 2 Soit $A_0 \in \mathbb{R}^{n\times n}$ une matrice inversible, d'inverse B_0. L'application

$$F : (A, B) \in \mathbb{R}^{n\times n} \times \mathbb{R}^{n\times n} \mapsto A \times B - I \in \mathbb{R}^{n\times n}$$

est continûment différentiable et s'annule en (A_0, B_0). Pour toute matrice A de $\mathbb{R}^{n\times n}$, l'application $B \to A \times B - I$ est affine, donc

$$\partial_B F(A,B) \cdot H = F(A, B+H) - F(A,B) = A \times H.$$

D'après le théorème des fonctions implicites, il existe des voisinages ouverts U de A_0 dans $\mathbb{R}^{n\times n}$ et V de $B_0 = A_0^{-1}$ dans $\mathbb{R}^{n\times n}$ et une fonction $\mathrm{inv} : U \to V$ continûment différentiable telle que

$$A \times B = I \text{ et } (A,B) \in U \times V \;\Leftrightarrow\; B = \mathrm{inv}(A).$$

La seule solution de $A \times B = I$ en B étant l'inverse de A si elle existe, l'inverse de A existe dans un voisinage de A_0 – la matrice inversible A_0 étant arbitraire, l'ensemble des matrices inversibles A est donc ouvert – et une fonction continûment différentiable de A.

De plus, la différentielle inv est donnée par

$$\begin{aligned} d\mathrm{inv}(A) &= -(\partial_B F(A, \mathrm{inv}(A)))^{-1} \cdot \partial_A F(A, \mathrm{inv}(A))) \\ &= -(H \mapsto A^{-1} \times H) \cdot (H \mapsto H \times A^{-1}) \\ &= (H \mapsto -A^{-1} \times H \times A^{-1}). \end{aligned}$$

ou sous une forme plus compacte

$$d(A^{-1}) = -A^{-1} \times dA \times A^{-1}.$$

Méthode de Newton

Si la fonction f est deux fois continûment différentiable, les termes $f(x,y)$ et $\partial_y f(x,y)$ sont des fonctions différentiables de y à x fixé. Soit

$$\phi_x(y) = y - (\partial_y f(x,y))^{-1} \cdot f(x,y)$$

Comme le produit de matrices $(A,B) \mapsto A \times B$ est différentiable, avec $d(A\times B) = dA \times B + A \times dB$ et que l'inversion de matrice $A \mapsto A^{-1}$, avec $d(A^{-1}) = -A^{-1} \times dA \times A^{-1}$, le terme $(\partial_y f(x,y))^{-1} \cdot f(x,y)$ est différentiable par rapport à y par la règle de dérivation en chaîne de différentielle partielle par rapport à y

$$-(\partial_y(\partial_y f(x,y))^{-1} \cdot dy) \cdot f(x,y) - (\partial_y f(x,y))^{-1} \cdot (\partial_y f(x,y) \cdot dy).$$

Le second terme de cette expression vaut dy et le premier vaut

$$(\partial_y f(x,y))^{-1} \cdot (\partial^2_{yy} f(x,y) \cdot dy) \cdot (\partial_y f(x,y))^{-1} \cdot f(x,y).$$

On obtient finalement

$$d\phi_x(y) = (\partial_y f(x,y))^{-1} \cdot (\partial^2_{yy} f(x,y) \cdot dy) \cdot (\partial_y f(x,y))^{-1} \cdot f(x,y).$$

En particulier, si $f(x,y) = 0$, on a bien $d\phi_x(y) = 0$.

Différences finies – erreur d'arrondi

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 `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 `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
```

Projet numérique : lignes de niveau

L'objectif de ce projet numérique est de développer un programme Python permettant de calculer les lignes de niveau d'une fonction f de deux variables réelles et à valeurs réelles (supposée continûment différentiable), c'est-à-dire les ensembles de la forme

$$\{(x, y) \in \mathbb{R}^2 \mid f(x, y) = c\} \text{ où } c \in \mathbb{R}.$$

La représentation graphique de ces courbes est un *tracé de contour* (cf. les exemples d'usage de la fonction `contour` de matplotlib).

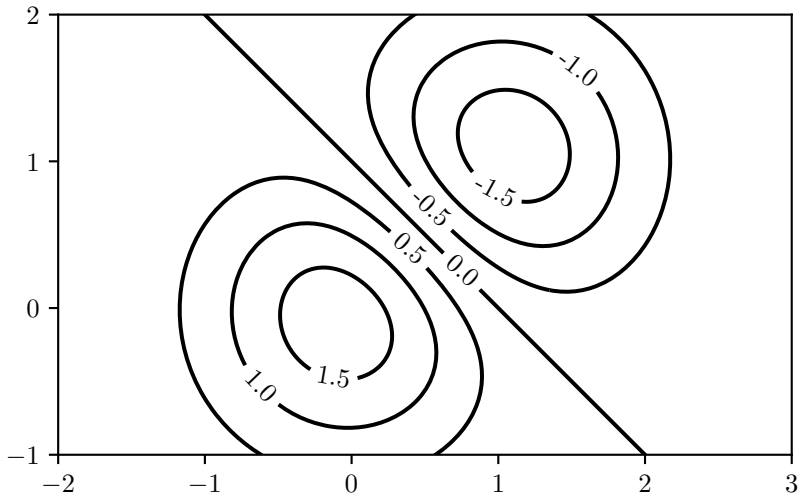

FIGURE 5 – Lignes de niveau de $(x, y) \mapsto 2(f(x, y) - g(x, y))$ où $f(x, y) = \exp(-x^2 - y^2)$ et $g(x, y) = \exp(-(x-1)^2 - (y-1)^2)$. Source: "Contour Demo" (matplotlib).

Contour simple

On suppose dans un premier temps que la fonction f est définie dans le carré unité $[0, 1]^2$ et on limite notre recherche aux lignes de niveau qui possèdent un point sur l'arête gauche du domaine de définition (de la forme $(0, y)$ pour un $0 \leq y \leq 1$.)

Amorce À quelle condition raisonnable portant sur $f(0, 0)$, $f(0, 1)$ et le réel c est-on certain qu'il existe un $t \in [0, 1]$ tel que $f(0, t) = c$? Développer une fonction, conforme au squelette suivant

```
def find_seed(g, c=0, eps=2**(-26)):
    ...
    return t
```

qui renvoie un flottant éloigné d'au plus `eps` d'un tel t ou `None` si la condition évoquée ci-dessus n'est pas satisfaite.

Propagation On souhaite implémenter une fonction dont la signature est :

```
def simple_contour(f, c=0.0, delta=0.01):
    ...
    return x, y
```

qui renvoie un fragment de ligne de niveau de valeur `c` de `f`, sous la forme de deux tableaux 1d d'abscisses et d'ordonnées de points de cette ligne. Les points devront être espacés d'approximativement `delta`. En cas d'impossibilité de générer un tel fragment deux tableaux vides devront être renvoyés.

Contour complexe

La signature de la fonction `contour` générale sera la suivante

```
def contour(f, c=0.0, xc=[0.0,1.0], yc=[0.0,1.0], delta=0.01):
    ...
    return xs, ys
```

Le domaine de f n'est plus nécessairement $[0,1] \times [0,1]$; les arguments `xc` et `yc` sont des listes (ou tableaux 1d) croissantes de nombres flottants qui découpent une portion rectangulaire de ce domaine en cellules carrées, telles que `xc[i] <= x <= xc[i+1]` et `yc[j] <= y <= yc[j+1]`. Les valeurs par défaut de `xc` et `yc` correspondent à une unique cellule qui est $[0,1]^2$; il correspond donc au contexte de `simple_contour`.

Dans chaque cellule, on exploitera le procédé utilisé dans `simple_contour`, mais en recherchant des amorces sur toute la frontière de la cellule et plus uniquement sur son arête gauche.

Les tableaux 1d `xs` et `ys` renvoyés par la fonction `contour` ne décrivent pas un fragment de contour, mais un ensemble de tels fragments ; cette multiplicité résulte de la présence de plusieurs cellules et/ou de l'existence de plusieurs fragments par cellule. Les valeurs `x = xs[i]` et `y = ys[i]` représentent un fragment de contour de `f` comme en produit `simple_contour` ; autrement dit, le tracé d'un contour peut être réalisé par le code

```
for x, y in zip(xs, ys):
    matplotlib.pyplot.plot(x, y)
```

Consignes

Le livrable de ce projet sera un notebook Jupyter. Ce support doit vous permettre de documenter l'ensemble de votre démarche – d'expliquer d'où viennent vos idées, de mener les calculs théoriques associés, d'écrire le code les mettant en oeuvre en pratique, de réaliser des expériences pour tester ce code, puis de mener leur analyse critique, analyse qui peut mener à de nouvelles idées, etc. En particulier, les échecs – quand ils sont instructifs – doivent être documentés !

Expérimenter suppose de pouvoir tester la génération de lignes de niveaux sur de “bonnes” fonctions de référence. Pour l'évaluation de `simple_contour`, les fonctions quadratiques sont de bonnes candidates ; pour `contour`, la fonction utilisée par la démo de la fonction `contour` de `matplotlib`, représentée dans la figure ci-dessus, est pertinente.

Ce projet devrait exploiter des algorithmes de point-fixe qui nécessitent du calcul matriciel et du calcul de gradients et/ou de matrices jacobiennes. Vous utiliserez de préférence NumPy pour le calcul matriciel et `HIPS/autograd` pour la différentiation automatique (cf. annexe).

Annexe – `HIPS/autograd`

Site Web: `https://github.com/HIPS/autograd`

```
>>> import autograd
>>> from autograd import numpy as np
```

La documentation de `HIPS/autograd` fournit une bonne illustration d'usage pour le cas des fonctions scalaires d'une variable:

```
>>> def f(x):
...     y = np.exp(-2.0 * x)
...     return (1.0 - y) / (1.0 + y)
>>> deriv_f = autograd.grad(np.tanh)
>>> deriv_f(1.0)
0.4199743416140261
```

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

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

Pour les fonctions à valeurs vectorielles, l'équivalent est :

```
>>> def f(x, y):
...     return np.array([np.exp(x), np.exp(y)])
>>> def J_f(x, y):
...     j = autograd.jacobian
...     return np.c_[j(f, 0)(x, y), j(f, 1)(x, y)]
>>> J_f(0.0, 0.0)
array([[1., 0.],
       [0., 1.]])
```

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)*.