Calcul Différentiel II

Table des matières

Théorème des Fonctions Implicites

Objectifs

- comprendre la portée du résultat: permettre la résolution *locale* d'équations non-linéaires paramétrique, autour d'une solution connue de référence.
- savoir mettre en oeuvre la version "inversion locale" du théorème des fonction implicites pour manipuler des changements de variables.
- connaître et savoir mettre en oeuvre dans les deux cas le ressort de la preuve: un théorème de point fixe qui exploite la différentielle.
- applications ? Géométriques d'abord ? Au changements de variables de la physique (ex: thermo). Etudier un scope raisonnable. Cf Salamon sur scope géom diff ?

TODO

- Différentielle partielle nécessaire en amont, pas que dérivée partielle.
- Donner un jeu d'hypothèse "non minimal" pour le théorème des fonctions implicite dans le but de simplifier le résultat. Par exemple, supposer au minimum l'existence de la différentielle dans un voisiange du point de référence ? Ou carrément son existence et sa continuité ? Et ajouter en remarque que l'on peut nuancer / décomposer le résultats en affinant les hypothèses, qui ne sont pas minimales ? En particulier, cela suffit pour énoncer le théorème d'inversion locale, donc go, simplifions.
- Nota: preuve IFT nécessite point fixe *avec paramètre*. Pas nécessairement si l'on se place directement dans les hypothèses de différentiabilité continue ?

TODO

Exploiter "THE IMPLICIT AND THE INVERSE FUNCTION THEOREMS: EASY PROOFS" (Oswaldo Rio Branco de Oliveira)

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 tous $x \in U$ et $y \in V$,

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

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 qui a une application linéaire $A : \mathbb{R}^m \to \mathbb{R}^m$ 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.

TODO: ref au résultat de Tao sur math overflow où l'on ne dispose que de la différentiabilité (pas du caractère continûment différentiable).

TODO: évoquer cas Lipschitz ?

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:

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

On en déduit donc que

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

1. 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 inversible sur son domaine de définition.

TODO: ici aussi, nécessaire d'invoquer la continuité de l'inversion pour conclure quand au caractère C^1 de la fonction implicite. Factoriser ce résultat avec la remarque précédente ?

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 (meilleure) solution approchée de $f(x, y) = 0$, 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

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

... **TODO:** reformuler sous forme de point fixe.

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

...

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 de classe C^1, on peut trouver 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 le théorème 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

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

De plus, par continuité de ϕ en (x_0, y_0), on peut choisir 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.

TODO. conclure existence et unicité (expliciter choix voisinages U et V).

Pour montrer la différentiabilité de la fonction implicite ψ, il est nécessaire au préalable de montrer sa continuité. 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 $y \mapsto \phi_{x_1}(y)$ étant continue en y_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

TODO

■

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

■

Analyse d'Erreur / Numérique

Objectifs

— Savoir quelles sont les options quand il s'agit de calculer des dérivées, gradient, différentielles: "manuelles", symboliques, différences finies, diff auto. et avoir au final une idée de la portée de chacune (applicabilité, avantages, pbs)

— Connaitre le principe des méthodes de type différence finie et mes deux sources d'erreurs potentielles associées (très général, pas limité au calcul diff): "erreur de troncature" et "erreur d'arrondi". Savoir calculer des estimations numériques dans les deux cas. (attention, il y a plein de choses ici: il faut en passer par le modèle de représentation des nombres flottants, etc.)

Différentiation Automatique

Objectifs

— avantage et portée de la méthode (plus détaillée: précision, dérivées à un ordre arbitraire, "workflow", usages en optimisation, machine learning, etc.)
— connaître les (une version des) principes des différents "morceaux" de la méthode dans le cas de Python: "tracer", "computation graph", etc. Solution: en construire un "à la main", au moins les étapes importantes. Note: permet aussi d'apprécier les limitations de la méthode.
— sur péda, essayer forward pass (plus près du cours), mais expliquer backward pass pour pouvoir se "plugger" dans l'existant.
— exploiter un système existant, type `autograd` en python (sans doute le plus facile en terme de courbe d'apprentissage)

Tracer le Graphe de Calcul

TODO: montrer que les fonction Python n'implémentent pas "ce que l'on croit", c'est-à-dire par exemple $[\cos([x])]$ pour la function `lambda x: cos(x)`, parce que le TYPE de l'argument n'est pas spécifié et que cela à des conséquences importantes: le disassembleur montre bien que le bytecode est agnostique et se contente de résoudre des variables et d'appliquer une séquence d'instructions, sans de référence au type, sans savoir ce qu'il manipule.

```
>>> from dis import dis

>>> from math import *
```

TODO: expliquer notation lambda-fonction pour les expressions (ou fonctions anonymes).

```
>>> f = lambda x: cos(x)

>>> dis(lambda x: cos(x))
  1           0 LOAD_GLOBAL              0 (cos)
              2 LOAD_FAST                0 (x)
```

```
              4 CALL_FUNCTION            1
              6 RETURN_VALUE

>>> dis(lambda x: x + 1)
  1           0 LOAD_FAST                0 (x)
              2 LOAD_CONST               1 (1)
              4 BINARY_ADD
              6 RETURN_VALUE
```

Expliquer en prenant des exemples frappants comment on peut en profiter pour "intercepter" cette séquence d'appels (big brother ...) pour savoir ce qui se passe dans la fonction, influencer le résultat, etc.

```
>>> math_cos = cos
>>> def cos(x):
...     print(f"trace: cos({x})")
...     return math_cos(x)

>>> y = cos(pi)
trace: cos(3.141592653589793)
>>> y
-1.0
```

Le cas des opérateurs est plus complexe: le calcul de `x + 1` par exemple est délégué 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:

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

Mais 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
```

... à condition que nous travaillions 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 fonction de la library `math` de Python vont renvoyer des flottants classiques, il nous faut donc à nouveau les adapter:

```
>>> 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
```

Mais nous ne savons pas encore tracer correctement l'expression `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 tracer ce type d'appel, il va falloir ... le faire échouer ! La méthode appellée pour effectuer la somme jusqu'à présent confie l'opération à la méthode `__add__` de `1.0` parce ce cette 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; 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 entreprendre d'enregistrer les opérations que subit chaque variable et comment elle s'organise; 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=None):
...         self.value = value
...         self.function = function
...         self.args = args if args is not None else []
```

Il nous faut alors rendre les opérations usuelles compatibles 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 numérique) 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 sous forme d'opérateur `+`; elle doit donc nous permettre de définir les méthodes `__add__` et `__radd__`:

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

On remarque de nombreuse 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 sont implémentation en première lecture.

```
>>> def wrap(function):
...     def wrapped_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 = wrapped_function(*values)
...             output_node = Node(output_value, wrapped_function, node_args)
...             return output_node
...         else:
...             return function(*args)
...     wrapped_function.__qualname__ = function.__qualname__
...     return wrapped_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 = wrap(sin)
```

et

```
>>> def multiply(x, y):
...     return x * y
>>> multiply = wrap(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:

```
>>> def node_repr(node):
...     if node.function is not None:
...         function_name = node.function.__qualname__
...         return f"Node({node.value}, {function_name}, {node.args})"
...     else:
...         return f"Node({node.value})"
```

Puis, faisons en sorte qu'elle soit utilisée par défaut par le noeuds plutôt que la représentation standard des objets:

```
>>> Node.__str__ = Node.__repr__ = node_repr
```

Nous somme prêts à faire notre vérification:

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

```
>>> trace(lambda x, y: x * (x + y), [1.0, 2.0])
Node(3.0, multiply, [Node(1.0), Node(3.0, add, [Node(1.0), Node(2.0)])])
```

Calcul Automatique des Dérivées

Registre des functions "élémentaires" dont on connaît la différentielle

```
>>> differential = {}

>>> def d_cos(x):
...     return lambda dx: - sin(x) * dx
>>> differential[cos] = d_cos

>>> def d_multiply(x, y):
...     return lambda dx, dy: x * dy + dx * y
>>> differential[multiply] = d_multiply

>>> def d_from_derivative(f_prime):
...     def d_f(x):
...        return lambda dx: f_prime(x) * dx
...     return d_f
>>> differential[sin] = d_from_derivative(cos)

>>> differential[add] = lambda x, y: add
```

Tri topologique

```
>>> def 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

>>> def d(f):
...     def df(*args): # args=(x1, x2, ...)
...         start_nodes = [Node(arg) for arg in args]
...         end_node = f(*start_nodes)
...         sorted_nodes = sort_nodes(end_node).copy()
...         def df_x(*d_args): # d_args = (d_x1, d_x2, ...)
...             for node in sorted_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

>>> def f(x):
...     return x * x + 2 * x + 1
>>> x = 1.0
>>> df_x = d(f)(2.0)
>>> df_x(1.0)
6.0
```

Derivative of f (manual computation)

```
>>> def f(x):
...    return cos(x) * cos(x) + sin(x) * sin(x)
>>> df = d(f)
>>> def f_prime(x):
...    return df(x)(1.0)
>>> f_prime(pi/4)
0.0
```

Exercices

Cinématique des Robots Manipulateurs

Position de référence en cartésien, robot plan articulaire (ou extension 3d), étudier sous quelle conditions on peut "résoudre" un déplacement de l'effecteur.

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

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

On appellera une telle fonction T une *perturbation de l'identité.*

1. Montrer que la foncton T est injective.
2. Montrer que l'image $V = T(U)$ est un ouvert et que T est difféomorphisme (global) de U sur V.

Réponses

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 injective.
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 $f(U)$.

Racines d'un Polynôme

Si racine simple, variation continue (et plus) par rapport aux coefficients.

Lier ça à la sensibilité des valeurs propres (et vecteurs propres ?) par rapport aux coefficients de la matrice associée ? Avantage: plus de travail de mise en forme pour se ramener au pb de fct implicite (à ajouter aux objectifs).

Différentielle de $X \mapsto X^{-1}$

Différentiation à pas complexe

TODO

Méthode de Newton

Revenir sur la preuve du théorème des fonctions implicites mais sous sous une hypothèse C^2, montrer qu'il n'est pas nécessaire de modifier la méthode de Newton.

Inversion Locale

TODO: exemple où l'on complète le jacobien pour pouvoir appliquer le TIL.

Projet Numérique

Idées pour poursuivre l'introduction du moteur de diff auto:

— gérer fct retournant des constantes
— compléter les opérateurs arithmétiques, fcts usuelles, etc.
— gérer le control flow (important et pas dur si un peu guidé !)
— adapter le code pour faire du backward diff (à évaluer), avec pointeurs vers articles introductifs.
— diff d'ordre deux ? Compliqué, 2 show-stoppers potentiels (différentiation "lazy" et nodes nestés).

Faire un projet privé et une document de tests (public) pour permettre la vérification que ça marche ? Demander résultat comme un fichier autodiff.py + notebook mise en oeuvre ou notebook générant autodiff.py ? Quoi qu'il en soit: code et doc et accès sur github. Ce qui est fait en cours déjà fourni (sous quelle forme ? fichier, notebook, etc ?). Oui, avec jupyter nbconvert, ça ne pose pas de pb. Intégration doctest/notebook ? Bof, non, on gère ça "normalement", en dehors, avec le truc comme un doc markdown.

Applications (avec algo type IFT par exemple) ? En plus ? Eventuellement en utilisant un "vrai" autodiff pour ne pas être bloqué par des étapes précédentes non réussies ?