Derivee

April 25, 2019

1 Approcher une dérivée

Marc Lorenzi
25 avril 2019

```
In [1]: import matplotlib.pyplot as plt
        import math
        from sympy import *
        init_printing()
```

```
In [2]: plt.rcParams['figure.figsize'] = (12, 8)
```

Nous allons dans ce notebook

- Fabriquer des formules permettant d'approcher la dérivée d'une fonction f en un point x.
- Estimer l'erreur commise dans ces formules.
- Minimiser l'erreur commise.

Hormis dans la dernière partie, nous nous intéresserons uniquement à des dérivées d'ordre 1. Nous verrons tout à la fin que la généralisation aux dérivées d'ordre supérieur ne présente aucune difficulté.

Au travail !

1.1 1. Taux d'accroissement

1.1.1 1.1 L'approximation "naïve"

Tout le monde le sait, pour approcher la dérivée de la fonction f au point x il n'y a qu'à prendre la quantité

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

où h est réel "petit" ? Ce que tout le monde ne sait peut-être pas, c'est que cette approximation n'est pas très bonne. Pourquoi ?

Voici tout d'abord la fonction `approx_deriv`. Elle prend en paramètres une fonction f, un réel x et un réel $h \neq 0$ et renvoie le quotient dont nous venons de parler.

```
In [3]: def approx_deriv(f, x, h):
            return (f(x + h) - f(x)) / h
```

Remarque : Dans tout le notebook je considérerai $f(x) = e^x$, en $x = 1$. Toutes les dérivées exactes de f en 1 valent évidemment e. Il vous est fortement conseillé de tester sur d'autres fonctions et d'autres points !

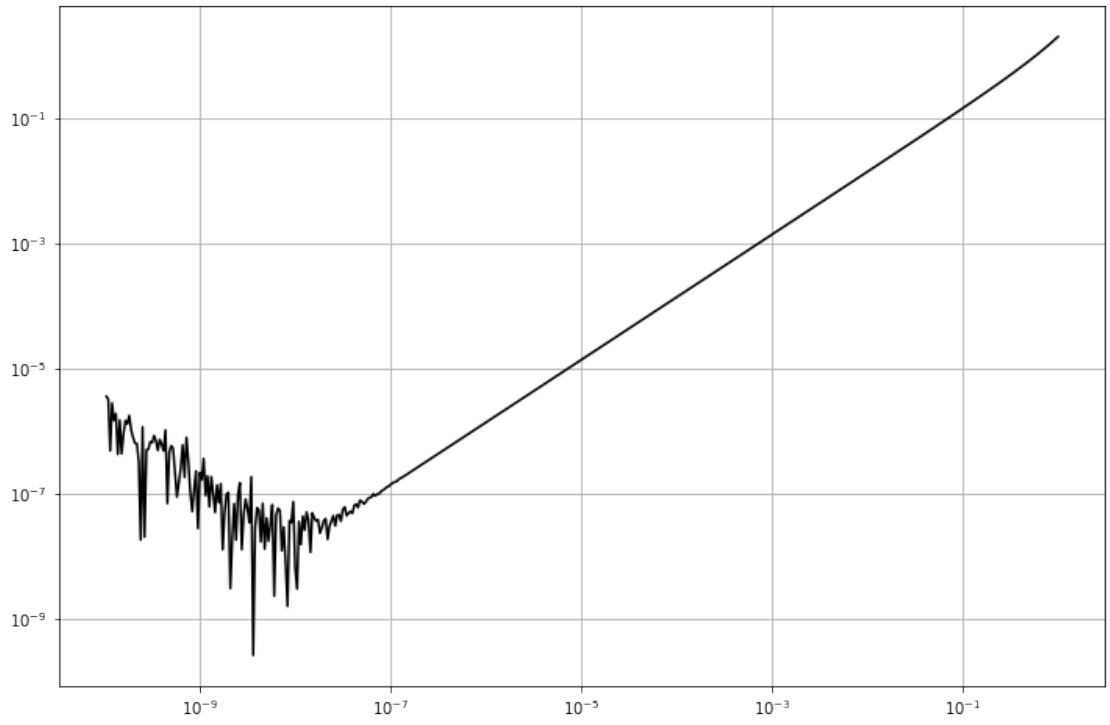
Ci-dessous, le graphe de l'erreur commise en fonction de h. La fonction `tracer_approx` prend en paramètres

- une fonction f
- Un réel x
- Une fonction f_1 qui est censée être la fonction dérivée de f
- Une fonction `fonc_approx` qui est du genre de notre fonction `approx_deriv`

La fonction trace l'erreur commise en fonction de h, en coordonnées logarithmiques.

```
In [6]: def tracer_approx(f, x, f1, fonc_approx):
            hs = [10. ** (-k / 50) for k in range(500)]
            ds = [abs(fonc_approx(f, x, h) - f1(x)) for h in hs]
            plt.loglog(hs, ds, 'k')
            plt.grid()
```

```
In [7]: tracer_approx(math.exp, 1, math.exp, approx_deriv)
```



Bigre ! Que voit-on ? En lisant le dessin de droite à gauche, on voit que lorsque h diminue, l'erreur commise diminue. Ceci n'est pas une surprise. Sauf que lorsque h devient plus petit que, environ, 10^{-8}, l'erreur se met à fluctuer de façon "imprévisible" et, globalement, à ré-augmenter. Que se passe-t-il ?

1.1.2 1.2 Taylor-Lagrange

Nous allons utiliser dans tout ce document une version du théorème de Taylor-Lagrange que voici.

Théorème : Soit $x \in \mathbb{R}$. Soit f une fonction définie dans un voisinage $V = [x - \alpha, x + \alpha]$ ($\alpha > 0$) de x. Soit $n \in \mathbb{N}$. On suppose que

- f est de classe $\mathcal{C}^n$ sur V.
- f est $n + 1$ fois dérivable sur V.

Soit $h > 0$ assez petit pour que $x + h \in V$. Alors il existe $c \in]x, x + h[$ tel que

$$f(x+h) = \sum_{k=0}^{n} \frac{x^k}{k!} f^{(k)}(x) + \frac{h^{n+1}}{(n+1)!} f^{(n+1)}(c)$$

Soit f une fonction de classe $\mathcal{C}^2$ dans un voisinage du réel x. Soit $h > 0$ assez petit pour que $x + h$ soit dans ce voisinage. La formule de Taylor-Lagrange appliquée avec $n = 1$ nous dit que

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2} f''(c)$$

où $x < c < x + h$.
Ainsi,

$$\frac{f(x+h) - f(x)}{h} = f'(x) + \frac{h}{2} f''(c)$$

1.1.3 1.3 Erreurs d'arrondi

En réalité, notre machine calcule avec une précision limitée. Le calcul de $f(x + h)$ renvoie $f(x + h) + \varepsilon_1$ où $\varepsilon_1 \in \mathbb{R}$ est une erreur d'arrondi (que l'on espère petite). De même, le calcul de $f(x)$ renvoie en réalité $f(x) + \varepsilon_2$. Ainsi, notre approximation de $f'(x)$ est plutôt

$$\frac{f(x+h) + \varepsilon_1 - f(x) - \varepsilon_2}{h} = f'(x) + \frac{h}{2} f''(c) + \frac{\varepsilon_1 - \varepsilon_2}{h}$$

et donc

$$\left| f'(x) - \frac{f(x+h) + \varepsilon_1 - f(x) - \varepsilon_2}{h} \right| \leq \frac{1}{2} M_2 h + \frac{|\varepsilon_1 - \varepsilon_2|}{h}$$

où M_2 est un majorant de $|f''|$, par exemple le maximum de $|f''|$ sur le segment $[x, x + h]$. Ce maximum existe car nous avons supposé f de classe $\mathcal{C}^2$.

1.1.4 1.4 Minimisation de l'erreur

Supposons que ε_1 et ε_2 sont inférieurs en valeur absolue à un certain $\varepsilon > 0$. L'erreur commise est donc inférieure à

$$\varphi(h) = \frac{1}{2} M_2 h + 2\frac{\varepsilon}{h}$$

On a

$$\varphi'(h) = \frac{1}{2}M_2 - 2\frac{\varepsilon}{h^2}$$

et on constate que φ passe par un minimum pour

$$h = \sqrt{\frac{4\varepsilon}{M_2}}$$

La valeur minimale de φ est alors, par un petit calcul,

$$\frac{3}{2}\sqrt{M_2\varepsilon}$$

Exercice : Faites le petit calcul.

Conséquence : Sauf coup de chance (erreurs d'arrondi magiquement petites) on ne peut pas obtenir la valeur de $f'(x)$ avec une précision meilleure que quelque chose de l'ordre de $\sqrt{\varepsilon}$.

Remarque : La norme IEEE754 décrit la façon dont sont codés les différents formats de réels. Le langage Python utilise le format **double**, dans lequel la "précision" des nombres est de 52 bits (je n'entre pas dans les détails ici). Une bonne valeur pour ε est donc sans doute

$$\varepsilon = 2^{-52} > 10^{-16}$$

Comme $\sqrt{\varepsilon} > 10^{-8}$, nous ne pouvons donc pas compter, sauf miracle, sur plus de 8 chiffres exacts après la virgule !

Remarque : Pour ceux d'entre-vous qui voudraient en savoir plus, le module `sys` contient un objet `float_info` qui contient tous les renseignement que l'on peut désirer sur la structure des `float` en Python.

Voici donc le ϵ que nous prendrons dans le notebook :

```
In [8]: 2 ** (-52)
```

```
Out[8]:
```

$$2.220446049250313e-16$$

Et voici la "précision" des flottants en Python :

```
In [9]: 52 * math.log(2, 10)
```

```
Out[9]:
```

$$15.65355977452702$$

Qu'obtenons-nous comme valeurs numériques du h optimal et comme majorant de l'erreur minimale ?

```
In [10]: def h_optimal(eps, M):
             return math.sqrt(4 * eps / M)
```

```
In [14]: h0 = h_optimal(2 ** (-52), math.exp(1))
         print(h0)
```

```
1.8076022258777422e-08
```

```
In [12]: def erreur_minimale(eps, M):
             return 3 / 2 * math.sqrt(M * eps)
```

```
In [13]: erreur_minimale(2 ** (-52), math.exp(1))
```

Out[13]:

$$3.685179212764192e-08$$

Et voici l'approximation de la dérivée obtenue en prenant $h = h_0$, le h théoriquement optimal.

```
In [15]: approx_deriv(math.exp, 1, h0) - math.exp(1)
```

Out[15]:

$$3.613141119629404e-08$$

1.1.5 1.5 Tracés

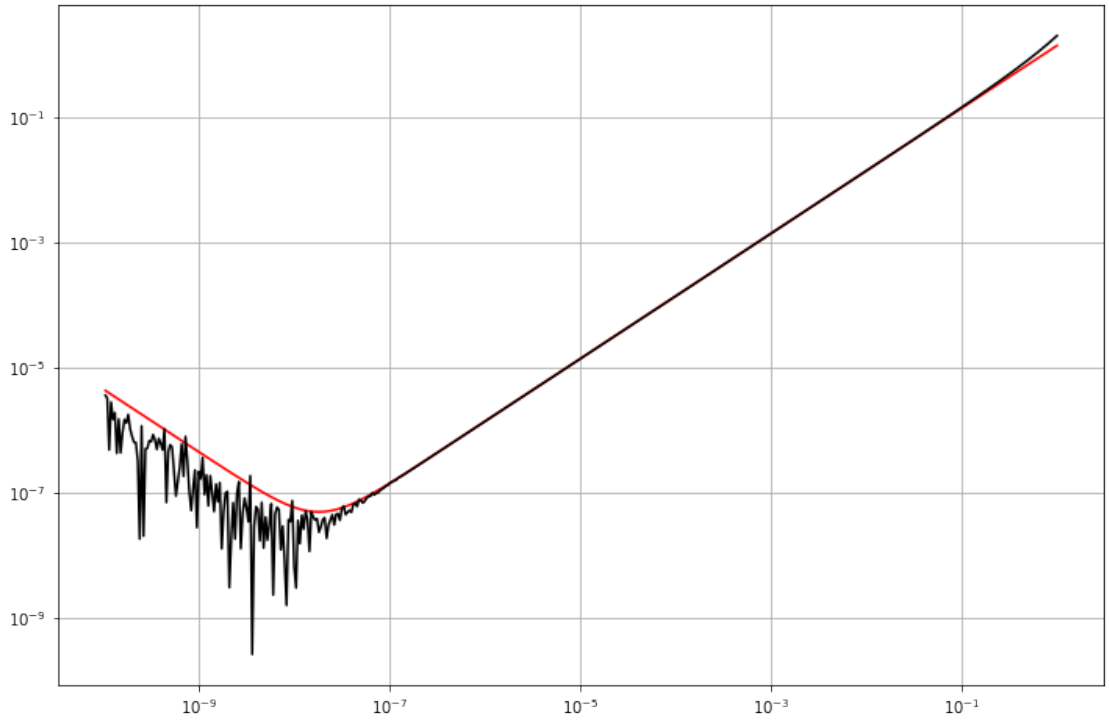
Traçons sur un même graphe l'erreur réelle et l'erreur théorique, pour la dérivée de l'exponentielle en 1.

Remarquez la valeur $M_2 = e$. En réalité $M_2 = e^{1+h} > e$, il faudrait prendre un peu plus que e, mais allons-y comme ça !

```
In [16]: def erreur_theorique(h, eps, M):
             return h * M / 2 + 2 * eps / h
```

```
In [17]: def tracer_approx2(f, x, f1, fonc_approx, fonc_erreur, M):
             hs = [10. ** (-k / 50) for k in range(500)]
             ds = [abs(fonc_approx(f, x, h) - f1(x)) for h in hs]
             es1 = [fonc_erreur(h, 2 ** (-52), M) for h in hs]
             plt.loglog(hs, es1, 'r')
             plt.loglog(hs, ds, 'k')
             plt.grid()
```

```
In [18]: tracer_approx2(math.exp, 1, math.exp, approx_deriv, erreur_theorique, math.exp(1))
```



Pour h petit, c'est le terme $\frac{2\epsilon}{h}$ qui est prépondérant. Pour h grand, au contraire, c'est le terme $\frac{1}{2}M_2 h$ qui domine. Remarquez, pour h voisin de 1, la courbe noire l'égèrement au-dessus de la courbe rouge. C'est dû à notre sous-estimation de la valeur de M_2 !

Exercice : Reprenez la cellule ci-dessus avec

- $f(x) = x^3$ en $x = 1$
- $f(x) = \arctan x$ en $x = 1$
- $f(x) = \ln x$ en $x = 2$
- $f(x) = \sin x$ en $x = \frac{\pi}{4}$

Des surprises ?

1.2 2. Une meilleure approximation

1.2.1 2.1 Combinons ...

Peut-on obtenir de meilleures approximations de $f'(x)$? Oui, on peut. Supposons que f est de classe $\mathcal{C}^3$ au voisinage de x. Soit $h > 0$ assez petit. On a, par la formule de Taylor-Lagrange,

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{6}f'''(c_1)$$

et

$$f(x-h) = f(x) - hf'(x) + \frac{h^2}{2}f''(x) - \frac{h^3}{6}f'''(c_2)$$

où $x < c_1, c_2 < x + h$. On en déduit que

$$\frac{f(x+h) - f(x-h)}{2h} = f'(x) + \frac{h^2}{12}(f'''(c_1) + f'''(c_2))$$

```
In [20]: def approx_deriv2(f, x, h):
             return (f(x + h) - f(x - h)) / (2 * h)
```

1.2.2 2.2 L'erreur commise

Faisons comme dans le paragraphe précédent et remplaçons $f(x+h)$ par $f(x+h)+\varepsilon_1$ et $f(x-h)$ par $f(x-h)+\varepsilon_2$. Nous obtenons

$$\frac{f(x+h) + \varepsilon_1 - f(x-h) - \varepsilon_2}{2h} = f'(x) + \frac{h^2}{12}(f'''(c_1) + f'''(c_2)) + \frac{\varepsilon_1 - \varepsilon_2}{2h}$$

Ainsi,

$$|f'(x) - \frac{f(x+h) + \varepsilon_1 - f(x-h) - \varepsilon_2}{2h}| \leq \frac{h^2}{6}M_3 + \frac{\varepsilon}{h}$$

où M_3 est un majorant de $|f'''|$ et $\varepsilon > 0$ est un majorant de $|\varepsilon_1|$ et $|\varepsilon_2|$. Posons

$$\varphi(h) = \frac{h^2}{6}M_3 + \frac{\varepsilon}{h}$$

On a

$$\varphi'(h) = \frac{h}{3}M_3 - \frac{\varepsilon}{h^2}$$

Cette fois-ci, le minimum de φ est atteint pour

$$h = \left(\frac{3\varepsilon}{M_3}\right)^{1/3}$$

et la valeur du minimum est

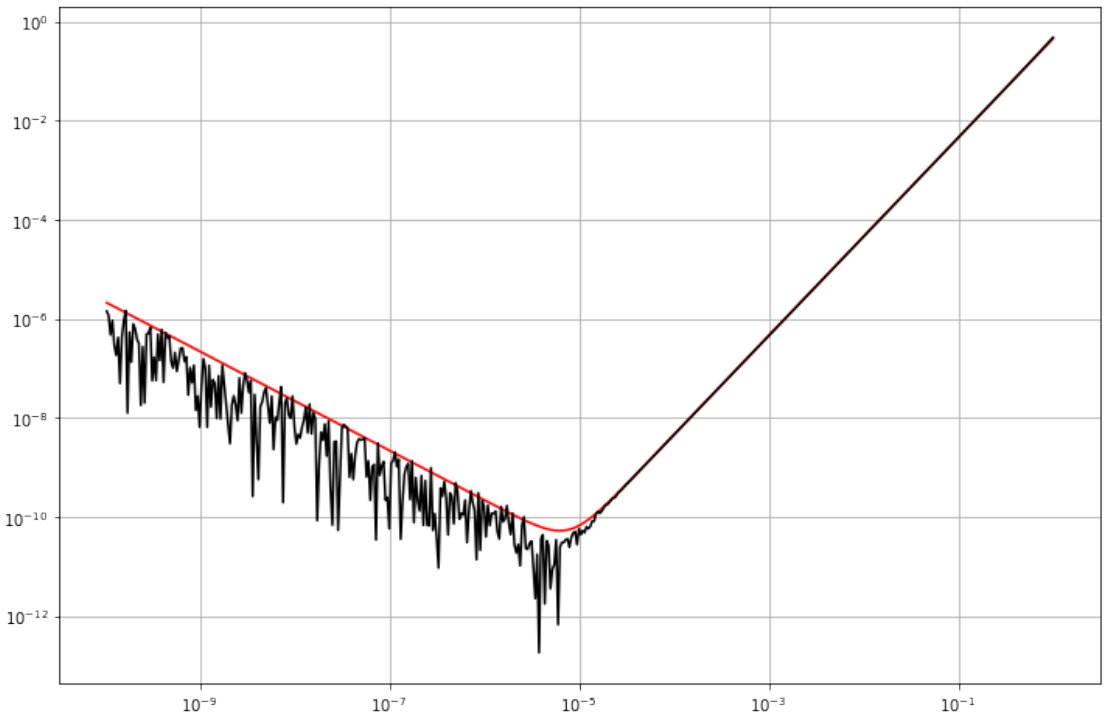
$$\frac{1}{2}3^{2/3}M^{1/3}\varepsilon^{2/3}$$

1.2.3 2.3 Tracés

Traçons, comme dans le paragraphe précédent.

```
In [21]: def erreur_theorique2(h, eps, M):
             return h ** 2 * M / 6 + eps / h
```

```
In [22]: tracer_approx2(math.exp, 1, math.exp, approx_deriv2, erreur_theorique2, math.exp(1))
```



Cette fois-ci, le h optimal est de l'ordre de 10^{-5}, et l'erreur minimale de l'ordre de 10^{-11}.

```
In [23]: def h_optimal2(eps, M):
             return (3 * eps / M) ** (1 / 3)

In [24]: print(h_optimal2(2 ** (-52), math.exp(1)))

6.257809421151901e-06


In [25]: def erreur_minimale2(eps, M):
             return 1/ 3 * 3 ** (2 / 3) * M ** (1 / 3) * eps ** (2 / 3)

In [26]: erreur_minimale2(2 ** (-52), math.exp(1))

Out[26]:
```

$$3.548280076643163e-11$$

Nous avons gagné 3 chiffres significatifs par rapport à la méthode naïve ! Peut-on faire mieux ? Oui.

1.3 3. Une infinité de formules d'approximation

Mais pourquoi une infinité ? Deux ça ne suffit pas :-) ?

1.3.1 3.1 Taylor-Lagrange, encore

Soit f une fonction de classe $\mathcal{C}^{n+1}$ au voisinage du réel x. Soit $h > 0$. Évaluons $f(x + j_0 h)$, $f(x + (j_0 + 1)h), \ldots, f(x + (j_0 + n)h)$. Posons dans ce qui suit $n_j = j_0 + j$.

$$f(x + n_j h) = \sum_{k=0}^{n} \frac{n_j^k h^k}{k!} f^{(k)}(x) + \frac{n_j^{n+1} h^{n+1}}{(n+1)!} f^{(n+1)}(c_j)$$

Combinons ensuite toutes ces valeurs. Pour cela, donnons nous $n + 1$ réels λ_j, $j = 0, \ldots, n$. On a

$$\sum_{j=0}^{n} \lambda_j f(x + n_j h) = \sum_{k=0}^{n} a_k \frac{h^k}{k!} f^{(k)}(x) + \frac{a_{n+1} h^{n+1}}{(n+1)!}$$

où, pour $k = 0, 1, \ldots, n$,

$$a_k = \sum_{j=0}^{n} n_j^k \lambda_j$$

et

$$a_{n+1} = \sum_{j=0}^{n} n_j^{n+1} \lambda_j f^{(n+1)}(c_j)$$

Est-il possible de choisir les λ_j pour que $a_0 = 0$, $a_1 = 1$, $a_2 = \ldots = a_n = 0$? Admettons pour l'instant que cela est possible. On obtient alors

$$\sum_{j=0}^{n} \lambda_j f(x + n_j h) = h f'(x) + \frac{a_{n+1} h^{n+1}}{(n+1)!}$$

Et, donc, la promesse d'une formule nous donnant une valeur approchée de $f'(x)$!

Remarque : Dans tout ce qui suit je prendrai toujours en exemple des n_j symétriques par rapport à 0. Par exemple :

- $j_0 = -1$, $n = 2$: les n_j sont $-1, 0, 1$.
- $j_0 = -2$, $n = 4$: les n_j sont $-2, -1, 0, 1, 2$.
- $j_0 = -3$, $n = 6$: les n_j sont $-3, -2, -1, 0, 1, 2, 3$.
- etc.

Cela n'a rien d'obligatoire, rien ne vous empêche de faire des tests différents de ceux-ci !

1.3.2 3.2 Vandermonde !

Nous avons repoussé à plus tard dans le paragraphe précédent le problème suivant : le système

$$(S) \quad a_k = \sum_{j=0}^{n} n_j^k \lambda_j$$

où $0 \leq k \leq n$ a-t-il une solution $(\lambda_0, \ldots, \lambda_n)$? Nous avons là un système de $n + 1$ équations à $n + 1$ inconnues dont la matrice A est la matrice dont les coefficients sont les n_j^k, $0 \leq k \leq n, 0 \leq j \leq n$ (k est l'indice des lignes, j est celui des colonnes). Cette matrice est bien connue, c'est une

matrice de Vandermonde. Et comme les n_j sont des nombres distincts, la théorie nous dit que A est inversible. En d'autres termes (S) est un système de Cramer. D'où la

Proposition : Le système (S) a une unique solution

1.3.3 3.3 Résolvons le système

La fonction `vandermonde` prend en paramètre une liste s de réels et renvoie la matrice de Vandermonde associée.

```
In [27]: def vandermonde(s):
             n = len(s)
             A = [[x ** k for x in s] for k in range(n)]
             return Matrix(A)

In [28]: vandermonde([-2, -1, 0, 1, 2])
```

Out[28]:

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ -2 & -1 & 0 & 1 & 2 \\ 4 & 1 & 0 & 1 & 4 \\ -8 & -1 & 0 & 1 & 8 \\ 16 & 1 & 0 & 1 & 16 \end{bmatrix}$$

Il est maintenant facile de résoudre notre système, qui est de la forme $A\Lambda = B$. Son unique solution est

$$\Lambda = A^{-1}B$$

La fonction `coefs` prend en paramètres les entiers j_0 et n dont nous parlons depuis un certain temps et renvoie la liste des coefficients $\lambda_j, j = 0, \ldots, n$. Évidemment, `sympy` est notre ami :-).

```
In [29]: def coefs(j0, n):
             s = [j0 + j for j in range(n + 1)]
             A = vandermonde(s)
             B = (n + 1) * [0]
             B[1] = 1
             B = Matrix(B)
             return A ** (-1) * B
```

Remarque : le calcul des coefficients λ_j nécessite l'inversion *exacte* d'une matrice de taille $n+1$. Il nous faudra donc être raisonnables sur la valeur de n. En pratique, les temps de calcul deviennent prohibitifs lorsque n atteint 20.

1.3.4 3.4 Exemples

Trois exemples ? Reprenons tout d'abord l'approximation naïve du début du notebook,

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

Ici, $j_0 = 0$ et $n = 1$.

```
In [30]: C = coefs(0, 1)
         C
```

Out[30]:

$$\begin{bmatrix} -1 \\ 1 \end{bmatrix}$$

Nous trouvons effectivement $\lambda_0 = -1$ et $\lambda_1 = 1$.
Prenons maintenant l'approximation un peu moins naïve

$$f'(x) \simeq \frac{f(x+h) - f(x-h)}{2h}$$

Ici, $j_0 = -1$ et $n = 2$.

```
In [31]: C = coefs(-1, 2)
         C
```

Out[31]:

$$\begin{bmatrix} -\frac{1}{2} \\ 0 \\ \frac{1}{2} \end{bmatrix}$$

Prenons maintenant $j_0 = -2$ et $n = 4$.

```
In [32]: C = coefs(-2, 4)
         C
```

Out[32]:

$$\begin{bmatrix} \frac{1}{12} \\ -\frac{2}{3} \\ 0 \\ \frac{2}{3} \\ -\frac{1}{12} \end{bmatrix}$$

Je vous laisse écrire la formule correspondante. Ou alors attendez un peu, Python va le faire pour nous.

1.3.5 3.5 La "formule"

Voici notre nouvelle fonction pour approcher une dérivée. La fonction `approx_deriv3` prend en paramètres

- Une fonction f.
- Un réel x.
- Une liste C de coefficients.
- Un entier j_0.
- Un réel $h > 0$.

Elle renvoie l'approximation

$$f'(x) \simeq \frac{1}{h}\sum_{j=0}^{n} \lambda_j f(x + n_j h)$$

où $n+1$ est la longueur de C et $n_j = j_0 + j$.

```
In [33]: def approx_deriv3(f, x, C, j0, h):
             s = 0
             n = len(C) - 1
             for j in range(0, n + 1):
                 s += C[j] * f(x + h * (j0 + j))
             return s / h
```

```
In [34]: approx_deriv3(math.exp, 1, coefs(-2, 4), -2, 1e-3)
```

Out[34]:

$$2.71828182845851$$

Voulez-vous quelque chose de plus "formel" ? Voici la "formule d'approximation".

```
In [35]: f = Function('f')
         x, h = symbols('x h')
```

```
In [36]: simplify(approx_deriv3(f, x, coefs(-2, 4), -2, h))
```

Out[36]:

$$\frac{f(-2h+x) - 8f(-h+x) + 8f(h+x) - f(2h+x)}{12h}$$

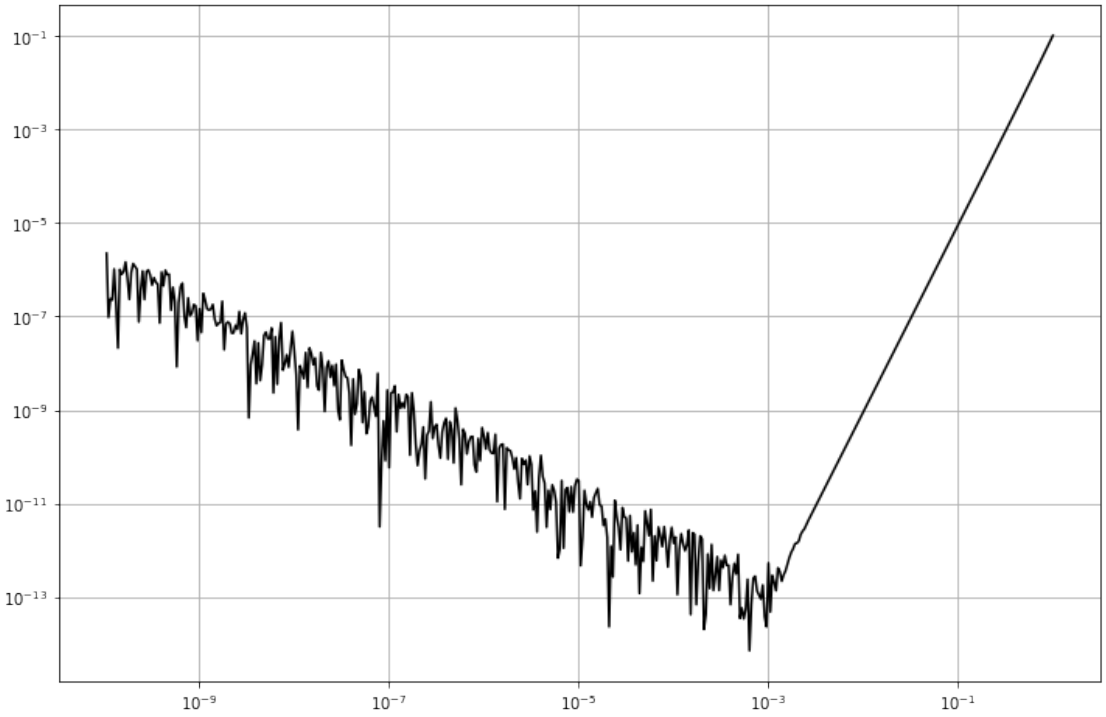
Nous avons donc

$$f'(x) \simeq \frac{f(x-2h) - 8f(x-h) + 8f(x+h) - f(x+2h)}{12h}$$

1.3.6 3.6 L'erreur commise

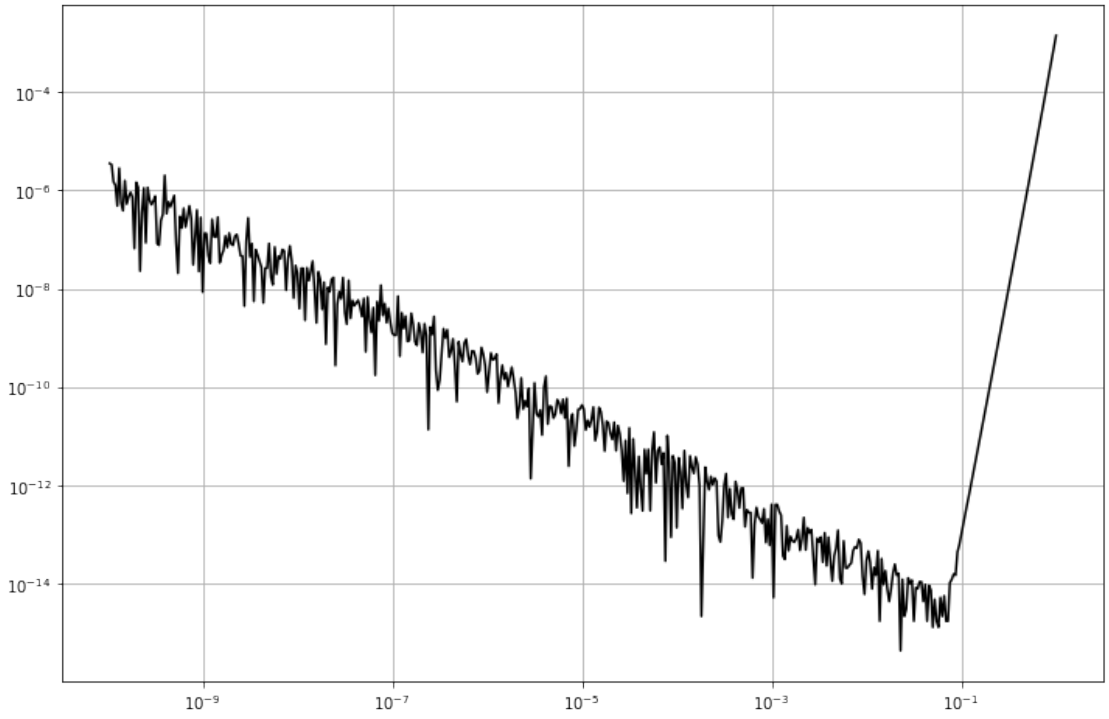
3.6.1 Quelques tests Reprenons les représentations graphiques de l'erreur que nous avions faites au début du notebook. Si nous prenons le troisième exemple vu ci-dessous, voici ce que nous obtenons.

```
In [37]: j0 = -2
         n = 4
         C = coefs(j0, n)
         tracer_approx(math.exp, 1, math.exp, lambda f, x, h: approx_deriv3(f, x, C, j0, h))
```



L'erreur minimale est obtenue pour $h \simeq 10^{-3}$ et elle est de l'ordre de 10^{-13}. Encore deux chiffres significatifs gagnés ! Pleins d'espoir, tentons $j_0 = -5$ et $n = 10$.

```
In [38]: j0 = -5
         n = 10
         C = coefs(j0, n)
         tracer_approx(math.exp, 1, math.exp, lambda f, x, h: approx_deriv3(f, x, C, j0, h))
```



Une erreur minimale inférieure à 10^{-14}, obtenue pour $h \simeq \frac{1}{10}$ … Sachant que Python travaille avec un peu plus de 15 chiffres significatifs, nous sommes à un chiffre du mieux possible !

3.6.2 La théorie Rappelons la formule trouvée plus haut :

$$\sum_{j=0}^{n} \lambda_j f(x + n_j h) = h f'(x) + \frac{a_{n+1} h^{n+1}}{(n+1)!}$$

Si nous tenons compte des erreurs d'arrondis dues aux calculs en virgule flottante, il nous faut remplacer $f(x + n_j h)$ par $f(x + n_j h) + \varepsilon_j$, où $\varepsilon_j \in \mathbb{R}$ est la fameuse erreur. Ainsi,

$$\sum_{j=0}^{n} \lambda_j f(x + n_j h) + \sum_{j=0}^{n} \lambda_j \varepsilon_j = h f'(x) + \frac{a_{n+1} h^{n}}{(n+1)!}$$

Si l'on suppose tous les ε_j inférieurs, en valeur absolue, à un réel $\varepsilon > 0$, il vient

$$\left| f'(x) - \frac{1}{h} \sum_{j=0}^{n} \lambda_j f(x + n_j h) \right| \leq \frac{\varepsilon}{h} \sum_{j=0}^{n} |\lambda_j| + \frac{|a_{n+1}| h^n}{(n+1)!}$$

Rappelons-nous que

$$a_{n+1} = \sum_{j=0}^{n} n_j^{n+1} \lambda_j f^{(n+1)}(c_j)$$

On a donc

$$|a_{n+1}| \leq M_{n+1} \sum_{j=0}^{n} |n_j|^{n+1} |\lambda_j|$$

où M_{n+1} est un majorant de $|f^{(n+1)}|$.

Notation : Pour $k \in \mathbb{N}$, notons $S_k = \sum_{j=0}^{n} |\lambda_j| |n_j|^k$. La fonction `sum_abs` ci-dessous calcule les sommes S_k. On lui passe évidemment en paramètre la liste C des coefficients λ_j, ainsi que l'entier j_0.

```
In [39]: def sum_abs(C, j0, k):
             s = 0
             n = len(C) - 1
             for j in range(0, n + 1):
                 nj = j0 + j
                 s += abs(C[j]) * abs(nj ** k)
             return s
```

```
In [40]: C = coefs(-2, 4)
         C, [sum_abs(C, -2, k) for k in range(6)]
```

Out[40]:

$$\left(\begin{bmatrix} \frac{1}{12} \\ -\frac{2}{3} \\ 0 \\ \frac{2}{3} \\ -\frac{1}{12} \end{bmatrix}, \quad \begin{bmatrix} \frac{3}{2}, & \frac{5}{3}, & 2, & \frac{8}{3}, & 4, & \frac{20}{3} \end{bmatrix} \right)$$

Avec nos nouvelles notations nous avons donc

$$\left| f'(x) - \frac{1}{h} \sum_{j=0}^{n} \lambda_j f(x + n_j h) \right| \leq \frac{\varepsilon S_0}{h} + \frac{M_{n+1} S_{n+1} h^n}{(n+1)!}$$

Posons

$$\varphi(h) = \frac{\varepsilon S_0}{h} + \frac{M_{n+1} S_{n+1} h^n}{(n+1)!}$$

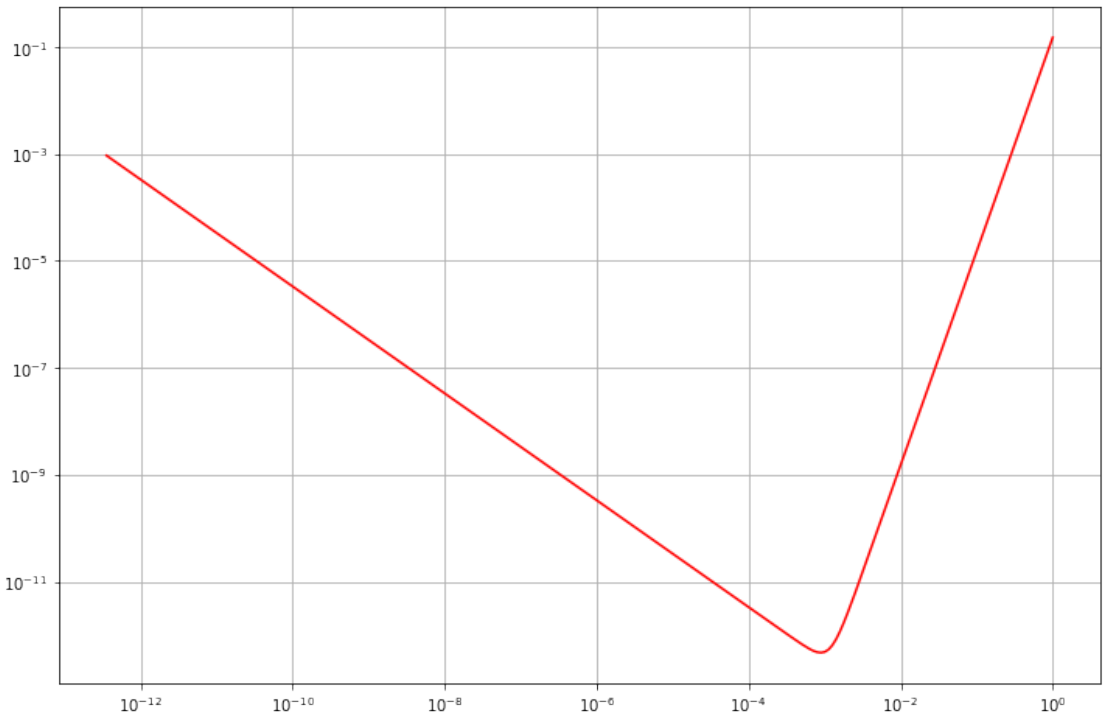
Un peu de courage, nous y sommes presque (non, c'est pas vrai :-)).

```
In [41]: def fact(n):
             p = 1
             for k in range(1, n + 1): p *= k
             return p
```

```
In [42]: def erreur_theorique3(C, j0, h, eps, M):
             n = len(C) - 1
             S0 = sum_abs(C, j0, 0)
             S1 = sum_abs(C, j0, n + 1)
             return eps * S0 / h + M * S1 * h ** n / fact(n + 1)
```

Voici le graphe de l'erreur théorique.

```
In [43]: C = coefs(-2, 4)
         j0 = -2
         hs = [10. ** (-k / 20) for k in range(250)]
         es1 = [erreur_theorique3(C, j0, h, 2 ** (-52), math.exp(1)) for h in hs]
         plt.loglog(hs, es1, 'r')
         plt.grid()
```



Une étude de φ comme celles que nous avons déjà faites plus haut montre que le minimum de φ est atteint pour

$$h_0 = \left(\frac{\varepsilon S_0 (n+1)!}{n M_{n+1} S_{n+1}}\right)^{1/(n+1)}$$

et que

$$\varphi(h_0) = \varepsilon S_0 \left(1 + \frac{1}{n}\right) \left(\frac{n M_{n+1} S_{n+1}}{\varepsilon S_0 (n+1)!}\right)^{1/(n+1)}$$

Exercice : Faites les calculs ! Non, ce n'est pas difficile.

```
In [44]: def h_optimal3(C, j0, eps, M):
             n = len(C) - 1
             S0 = sum_abs(C, j0, 0)
             S1 = sum_abs(C, j0, n + 1)
             return (eps * S0 * fact(n + 1) / (n * M * S1)) ** (1 / (n + 1))
```

```
In [45]: C = coefs(-2, 4)
         h_optimal3(C, -2, 2 ** (-52), math.exp(1))
```

Out[45]:

$$0.000887748410302645$$

```
In [46]: def erreur_minimale3(C, j0, eps, M):
             n = len(C) - 1
             S0 = sum_abs(C, j0, 0)
             S1 = sum_abs(C, j0, n + 1)
             return eps * S0 * (1 + 1 / n) * ((n * M * S1) / (eps * S0 * fact(n + 1))) ** (1 / (
```

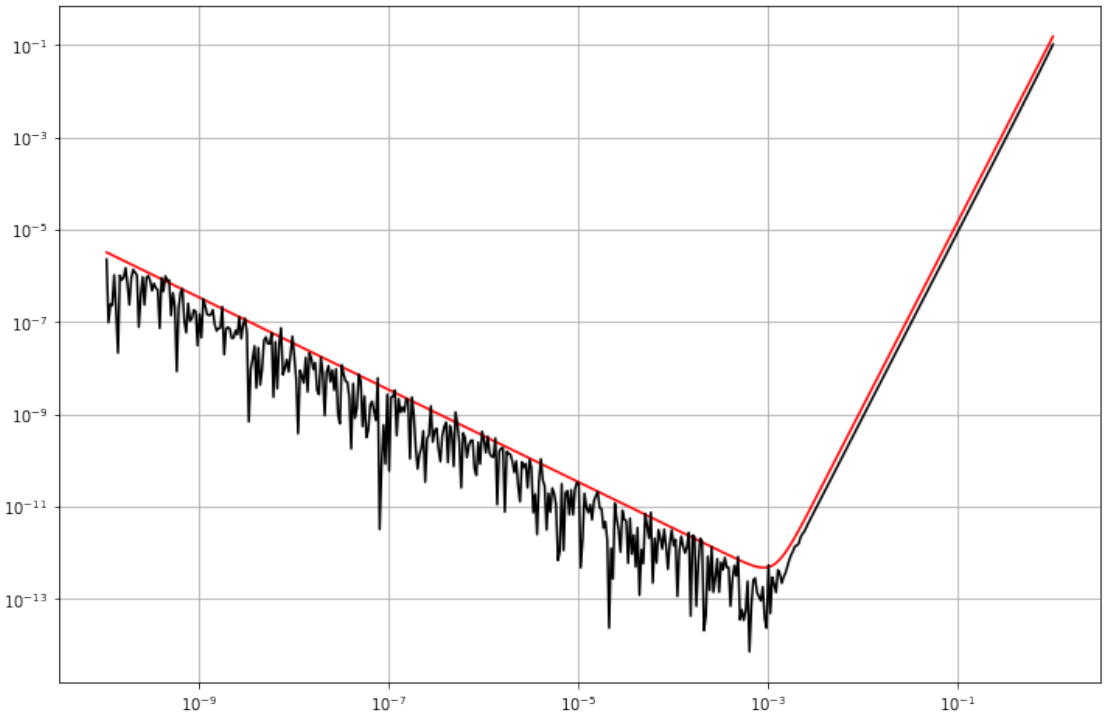
```
In [47]: C = coefs(-2, 4)
         erreur_minimale3(C, -2, 2 ** (-52), math.exp(1))
```

Out[47]:

$$4.68977054087318 \cdot 10^{-13}$$

Superposons comme d'habitude l'erreur réelle et son estimation théorique :

```
In [48]: j0 = -2
         n = 4
         C = coefs(j0, n)
         tracer_approx2(math.exp, 1, math.exp,
                        lambda f, x, h: approx_deriv3(f, x, C, j0, h),
                        lambda h, eps, M:erreur_theorique3(C, j0, h, eps, M),
                        math.exp(1))
```



Et tentons pour finir une formule faisant intervenir 19 points !

```
In [49]: C = coefs(-9, 18)
         erreur_minimale3(C, -2, 2 ** (-52), math.exp(1))
```

Out[49] :

$$6.65052132890785 \cdot 10^{-15}$$

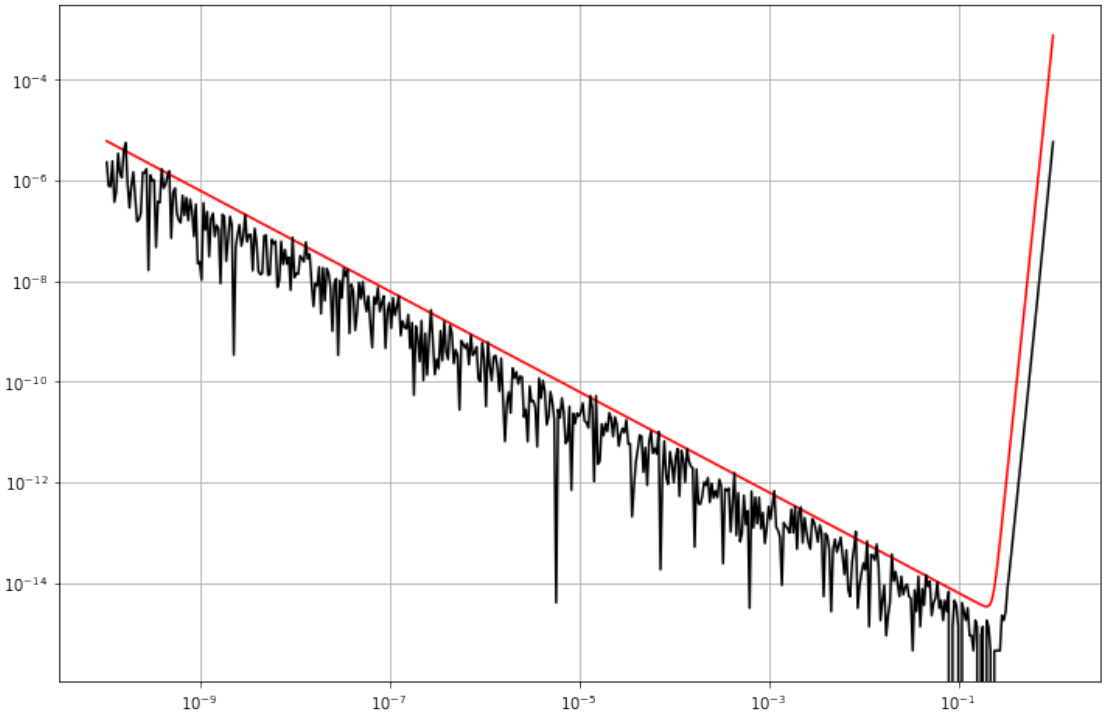
Juste pour rire, voici la formule :

```
In [50]: simplify(approx_deriv3(f, x, C, -9, h))
```

Out[50] :

$$-28f(-9h+x)+567f(-8h+x)-5508f(-7h+x)+34272f(-6h+x)-154224f(-5h+x)+539784f(-4h+$$

```
In [51]: j0 = -9
         n = 18
         #C = coefs(j0, n)
         tracer_approx2(math.exp, 1, math.exp,
                        lambda f, x, h: approx_deriv3(f, x, C, j0, h),
                        lambda h, eps, M:erreur_theorique3(C, j0, h, eps, M),
                        math.exp(1))
```



1.4 4. Dérivées d'ordre supérieur

Pour ne pas faire fuir un éventuel lecteur, je ne reprendrai pas dans cette section les calculs d'erreurs effectués dans les sections précédentes. Nous allons juste voir que notre méthode permet de calculer des valeurs approchées de dérivées d'ordre 2, 3, etc.

Rappelons-nous la formule

$$\sum_{j=0}^{n} \lambda_j f(x + n_j h) = \sum_{k=0}^{n} a_k \frac{h^k}{k!} f^{(k)}(x) + \frac{a_{n+1} h^{n+1}}{(n+1)!}$$

où, pour $k = 0, 1, \ldots, n$,

$$a_k = \sum_{j=0}^{n} n_j^k \lambda_j$$

et

$$a_{n+1} = \sum_{j=0}^{n} n_j^{n+1} \lambda_j f^{(n+1)}(c_j)$$

Nous avions alors décidé de choisir les λ_j pour que $a_1 = 1$ et $a_0 = a_2 = \ldots = a_n = 0$.

Soit maintenant $p \in [0, n]$. Nous pouvons choisir les λ_j pour que $a_p = 1$ et tous les autres a_k soient nuls, $0 \leq k \leq n$. On obtient alors

$$\sum_{j=0}^{n} \lambda_j f(x + n_j h) = \frac{h^p}{p!} f^{(p)}(x) + \frac{a_{n+1} h^{n+1}}{(n+1)!}$$

Une valeur approchée de $f^{(p)}(x)$ est donc

$$f^{(p)}(x) \simeq \frac{p!}{h^p} \sum_{j=0}^{n} \lambda_j f(x + n_j h)$$

avec une erreur commise que nous n'étudierons pas. Pas parce que c'est difficile mais parce que c'est exactement comme nous avons déjà fait. Laissé au lecteur, sans état d'âme !

```
In [52]: def coefs2(j0, n, p):
             s = [j0 + j for j in range(n + 1)]
             A = vandermonde(s)
             B = (n + 1) * [0]
             B[p] = 1
             B = Matrix(B)
             return A ** (-1) * B

In [53]: def approx_deriv4(f, x, p, C, j0, h):
             s = 0
             n = len(C) - 1
             for j in range(0, n + 1):
                 s += C[j] * f(x + h * (j0 + j))
             return s * fact(p) / h ** p
```

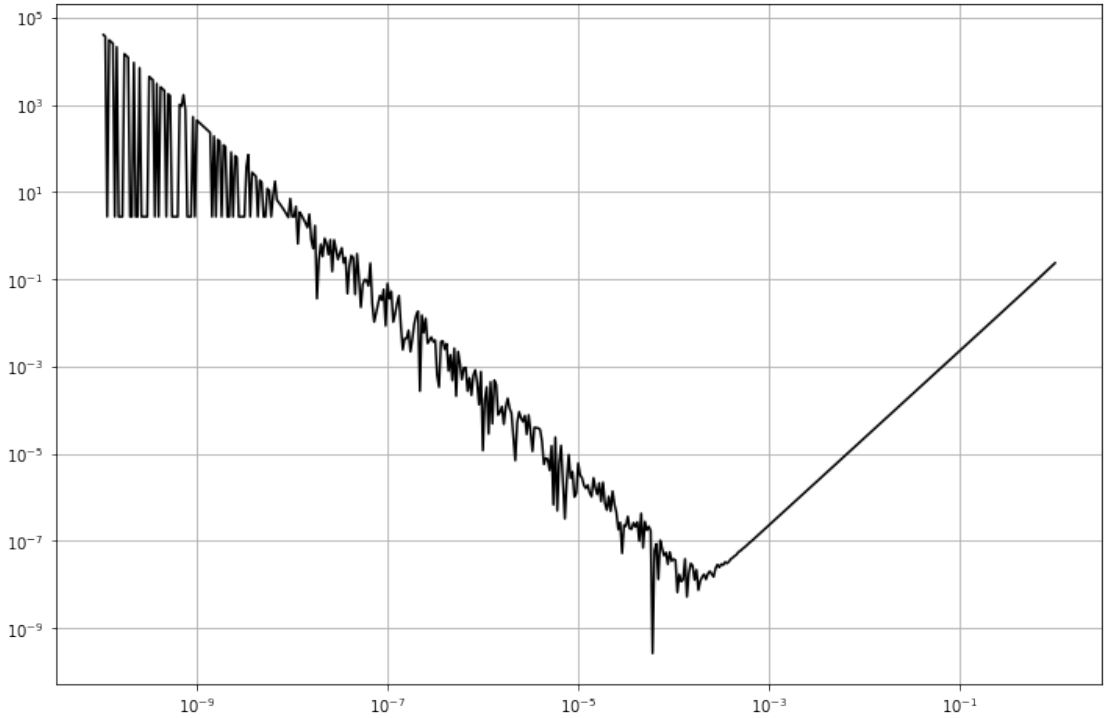
Voici par exemple une formule pour calculer une dérivée seconde.

In [54]:
```
simplify(approx_deriv4(f, x, 2, coefs2(-1, 2, 2), -1, h))
```

Out[54]:

$$\frac{-2f(x)+f(-h+x)+f(h+x)}{h^2}$$

In [55]:
```
j0 = -1
n = 2
p = 2
C = coefs2(j0, n, p)
tracer_approx(math.exp, 1, math.exp, lambda f, x, h: approx_deriv4(f, x, p, C, j0, h))
```



L'erreur minimale est de l'ordre de 10^{-7}. Approcher une dérivée d'ordre supérieur est-il une tâche délicate ?

Exercice : Mettez ci-dessus j_0 à -8 et n à 16. Moralité : tout n'est pas perdu :-).

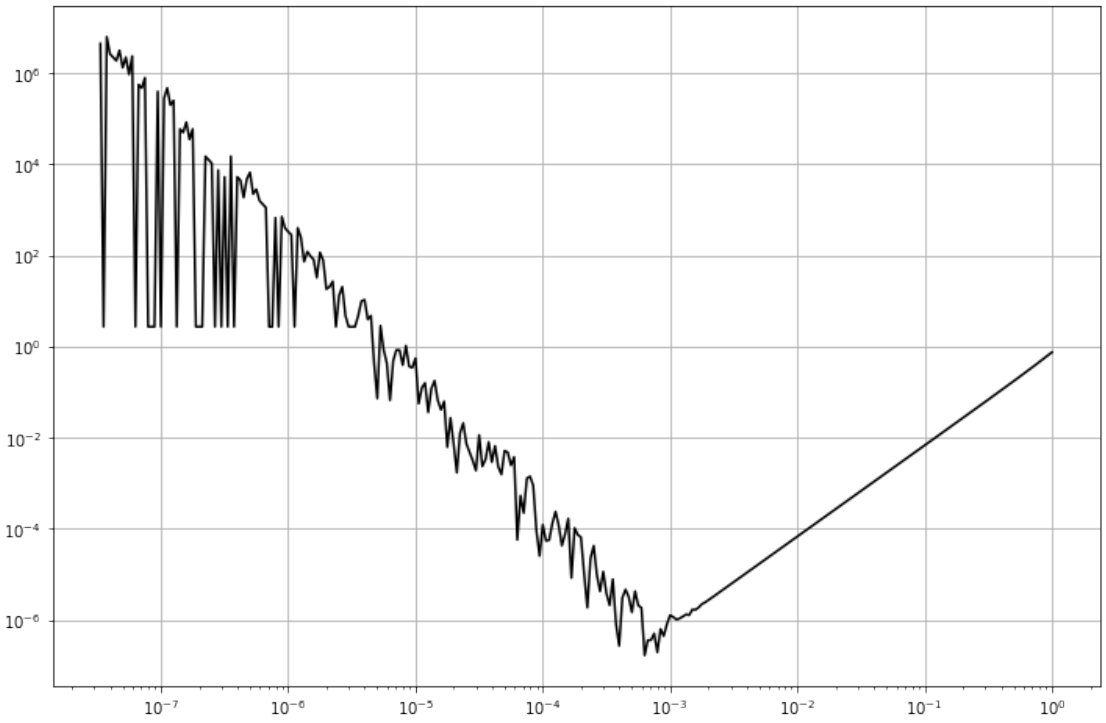
On tente une dérivée d'ordre 3 ?

In [56]:
```
factor(approx_deriv4(f, x, 3, coefs2(-2, 4, 3), -2, h))
```

Out[56]:

$$-\frac{f(-2h+x)-2f(-h+x)+2f(h+x)-f(2h+x)}{2h^3}$$

```
In [57]: j0 = -2
         n = 4
         p = 3
         C = coefs2(j0, n, p)
         hs = [10. ** (-k / 40) for k in range(300)]
         ds = [abs(approx_deriv4(math.exp, 1, 3, C, j0, h) - math.exp(1)) for h in hs]
         plt.loglog(hs, ds, 'k')
         plt.grid()
```



1.5 5. Et maintenant ?

Maintenant une question essentielle se pose : avez-vous appris quelque chose en lisant ce notebook ? Il y a une excellente façon d'en être sûrs :

1. Calculez un majorant de l'erreur théorique, $\varphi(h)$, commise dans le paragraphe 4 lors du calcul de $f^{(p)}(x)$. Cette erreur dépend de paramètres que nous appellerons sans plus de détails M_{n+1}, ε et p. Sans compter les coefficients λ_k, bien entendu. Pour cela inspirez-vous du paragraphe 3. Les calculs sont totalement analogues. Allez, dans un élan de bonté, je vous donne la réponse. Vous devriez trouver

$$\varphi(h) = \frac{p!S_{n+1}M_{n+1}}{(n+1)!}h^{n+1-p} + \frac{p!S_0\varepsilon}{h^p}$$

2. Calculez la valeur h_0 du paramètre h qui minimise φ. Vous trouvez

$$h_0 = \left(\frac{p(n+1)!S_0\varepsilon}{(n+1-p)S_{n+1}M_{n+1}} \right)^{\frac{1}{n+1}}$$

3. Calculez $\varphi(h_0)$. Si vous avez été soigneux, vous avez évidemment trouvé

$$\frac{p!S_0\varepsilon(n+1)}{n+1-p} \left(\frac{(n+1-p)S_{n+1}M_{n+1}}{p(n+1)!S_0\varepsilon} \right)^{\frac{p}{n+1}}$$

4. Tracez sur un même graphe l'erreur théorique en rouge et l'erreur réelle en noir, et admirez.

Avoir la possibilité de **visualiser** des résultats est un avantage immense. La confrontation entre théorie et pratique permet de se convaincre que nos raisonnements, nos calculs, ne sont pas entâchés (de trop) d'erreurs, que nos théories sont pertinentes. Si vous voulez un petit encouragement pour faire le travail demandé, peut-être faudra-t-il que je vous avoue l'inavouable ? Ce que le scientifique ne dira pas (car, c'est bien connu, le scientifique ne se trompe jamais) ?

Avant d'obtenir les splendides et parfaits graphiques de ce notebook j'ai dû recommencer plusieurs fois tous mes calculs :-).

In [ ]: