Applications I : Ajustement et optimisation

Ajustement

Description du problème :

Soient

- $(t_1, y_1) \dots (t_n, y_n)$ des données, p. ex. des données expérimentales,
- $f(t; \beta_1, \dots, \beta_p)$ une fonction (un "modèle") qui dépend de certains paramètres $\beta_1, \dots, \beta_p$.

On cherche les valeurs des paramètres telles que la fonction f correspondante décrit le mieux les données :

$$f(t_i; \vec{\beta}) \approx y_i \qquad \forall\, i\,.$$

Ajustement

Exemple : La trajectoire d'un objet en chute libre est donnée par une fonction quadratique f du temps t,

$$f(t; \beta_1, \beta_2, \beta_3) = \beta_1 + \beta_2\, t + \beta_3\, t^2 \stackrel{!}{=} y(t)$$

Ici

- $\beta_1 = y_0$ est la hauteur initiale à $t = 0$,
- $\beta_2 = v_0$ est la vitesse initiale,
- $\beta_3 = -g/2$ avec g l'accélération gravitationnelle.

Lors d'une expérience, on mesure

t[s]	y [m]
0	1
0.31	3
0.59	4
1.02	4
1.32	3
1.74	0

Comment en obtient-on les valeurs numériques de y_0, v_0 et g ?

Méthode des moindres carrés

Méthode standard : Méthode des moindres carrés.

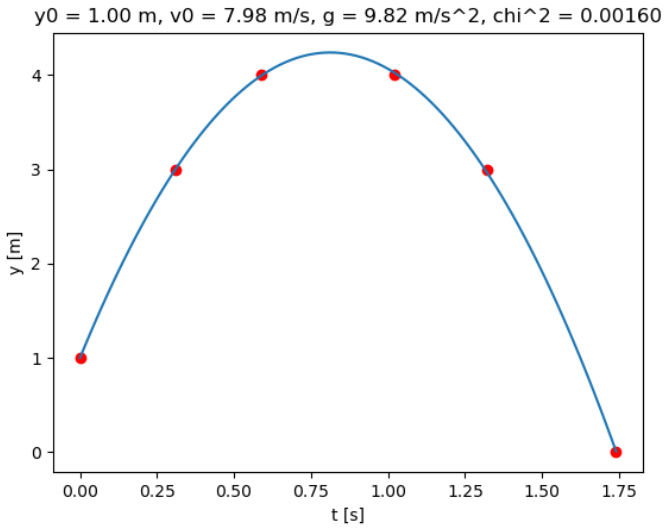
On définit le résidu r_i du point de données (t_i, y_i) par

$$r_i(\vec{\beta}) = f(t_i; \vec{\beta}) - y_i$$

et on cherche les valeurs des paramètres $\beta_1 \dots \beta_p$ telles que χ^2, défini par

$$\chi^2(\vec{\beta}) \equiv \sum_{i=1}^{n} r_i^2(\vec{\beta})$$

est minimisé. Ces valeurs donnent le meilleur ajustement des paramètres aux données.



Méthode des moindres carrés

Généralisation pour incertitudes variables

Si les données sont de la forme $(t_i, y_i \pm \sigma_i)$ avec des σ_i différents, alors on minimise plutôt

$$\chi^2(\vec{\beta}) = \sum_{i=1}^{n} \left(\frac{r_i(\vec{\beta})}{\sigma_i} \right)^2 , \qquad r_i(\vec{\beta}) = f(t_i; \vec{\beta}) - y_i \, .$$

Ainsi, les points de données avec grandes incertitudes σ_i contribuent à l'ajustement avec un poids moins important.

Régression linéaire

Problème numérique : Comment minimiser χ^2 ?

Dans l'exemple de la chute libre, la fonction f dépend des paramètres β_1, β_2, β_3 linéairement,

$$f(t;\,\vec{\beta}) = \beta_1 + \beta_2\, t + \beta_3\, t^2\,.$$

Problème de régression linéaire : on cherche β_1, β_2, β_3 tels que

$$y_i \approx f(t_i;\vec{\beta}) \;\Leftrightarrow\; A\vec{\beta} \approx \vec{y}$$

où

$$A = \begin{pmatrix} 1 & x_1(t_1) & x_2(t_1) \\ 1 & x_1(t_2) & x_2(t_2) \\ 1 & x_1(t_3) & x_2(t_3) \\ \vdots & \vdots & \vdots \\ 1 & x_1(t_6) & x_2(t_6) \end{pmatrix} \qquad \text{avec } x_1(t) = t, \quad x_2(t) = t^2\,.$$

Le système linéaire $A\vec{\beta} = \vec{y}$ est surdéterminé (6 équations pour seulement 3 inconnues ; χ^2 strictement positif). Le meilleur ajustement est donné par la solution d' un système linéaire de seulement 3 équations :

$$\boxed{A^T A\vec{\beta} = A^T\vec{y}}\,.$$

Preuve :

On souhaite minimiser $\vec{r}^{\,2} = (A\vec{\beta} - \vec{y})^2$ par rapport à $\vec{\beta}$, alors on cherche le $\vec{\beta}$ où le gradient s'annulle :

$$\frac{\partial}{\partial \beta_i} \left(A\vec{\beta} - \vec{y} \right)^2 = 0 \, .$$

Explicitement :

$$\begin{aligned} \frac{\partial}{\partial \beta_i} \left(A\vec{\beta} - \vec{y} \right)^2 &= \frac{\partial}{\partial \beta_i} \sum_{ajk} \left(A_{aj}\beta_j - y_a \right) \left(A_{ak}\beta_k - y_a \right) \\ &= \sum_{ak} A_{ai} A_{ak} \beta_k + \sum_{aj} A_{aj} \beta_j A_{ai} - \sum_a y_a A_{ai} - \sum_a A_{ai} y_a \\ &= 2(A^T A \vec{\beta})_i - 2(A^T \vec{y})_i \end{aligned}$$

ce qui s'annulle si $\vec{\beta}$ vérifie

$$A^T A \vec{\beta} = A^T \vec{y} \, .$$

Régression linéaire

```
import numpy as np
import numpy.linalg as la

# Les données:
t = np.array([0., .31, .59, 1.02, 1.32, 1.74])
y = np.array([1., 3., 4., 4., 3., 0.])

def x1(t):          # les variables prédicteur
    return t
def x2(t):
    return t**2

A = np.ones((6, 3)) # la matrice de coefficients
A[:, 1] = x1(t)     # (2ème colonne)
A[:, 2] = x2(t)     # (3ème colonne)

# Résoudre le système linéaire pour trouver les paramètres:
beta = la.solve(A.T @ A, A.T @ y)
y0, v0, g = beta[0], beta[1], -2 * beta[2]
```

Régression linéaire

Calculer χ^2 et tracer le résultat :

```
def f(t):
    return y0 + v0 * t - g/2 * t**2

r = f(t) - y          # les résidus
chi2 = np.sum(r**2)

import matplotlib.pyplot as plt
tpoints = np.linspace(0, 1.74, 100)
plt.plot(t, y, 'ro') # tracer les données
plt.plot(tpoints, f(tpoints)) # tracer la courbe théorique
plt.xlabel("t [s]")
plt.ylabel("y [m]")
plt.show()
```

Régression non linéaire

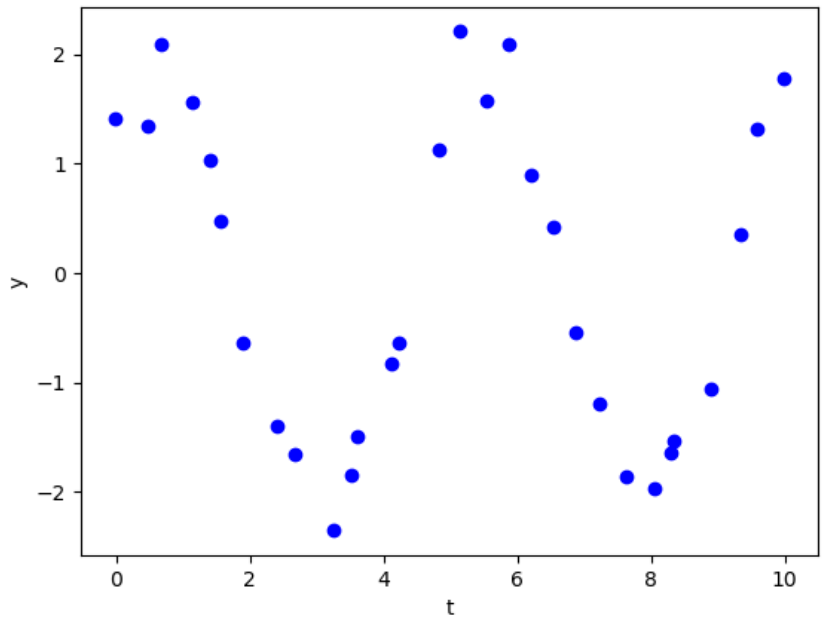
Dans l'exemple avant, la fonction d'ajustement $f(t;\,\vec{\beta})$ dépendait des paramètres $\beta_1 \ldots \beta_p$ linéairement.

Si la dépendence est plus compliquée, la minimisation de χ^2 devient plus difficile.

Exemple : Ajuster une fonction sinusoïdale,

$$f(t;\,\beta_1, \beta_2, \beta_3) = \beta_1\,\sin\left(\beta_2 t + \beta_3\right)$$

où β_1 est l'amplitude, β_2 la fréquence, β_3 la phase à $t = 0$.



Régression non linéaire : L'algorithme de Gauss-Newton

On cherche un minimum de $\chi^2(\vec{\beta})$.

- Méthode de Newton (rappel) : pour trouver un zéro de $g(x)$, on itère

$$x \leftarrow x - \frac{g(x)}{g'(x)}$$

- Pour trouver un point critique (potentiellement un minimum) de $g(x)$, on cherche un zéro de $g'(x)$: on itère

$$x \leftarrow x - \frac{g'(x)}{g''(x)}$$

- Généralisation à plusieurs variables : soit $g(\vec{\beta})$ une fonction de p variables $\vec{\beta}$. Pour trouver un point critique, itérer

$$\vec{\beta} \leftarrow \vec{\beta} - H^{-1}(\vec{\beta})\vec{\nabla} g(\vec{\beta})$$

où la matrice hesséenne $H(\vec{\beta})$ est

$$H = \begin{pmatrix} \frac{\partial^2 g}{\partial \beta_1^2} & \frac{\partial^2 g}{\partial \beta_1 \partial \beta_2} & \frac{\partial^2 g}{\partial \beta_1 \partial \beta_3} & \cdots & \frac{\partial^2 g}{\partial \beta_1 \partial \beta_p} \\ \frac{\partial^2 g}{\partial \beta_2 \partial \beta_1} & \frac{\partial^2 g}{\partial \beta_2^2} & \frac{\partial^2 g}{\partial \beta_2 \partial \beta_3} & \cdots & \frac{\partial^2 g}{\partial \beta_2 \partial \beta_p} \\ \vdots & & & & \vdots \\ \frac{\partial^2 g}{\partial \beta_p \partial \beta_1} & \frac{\partial^2 g}{\partial \beta_p \partial \beta_2} & \frac{\partial^2 g}{\partial \beta_p \partial \beta_3} & \cdots & \frac{\partial^2 g}{\partial \beta_p^2} \end{pmatrix}$$

Régression non linéaire : L'algorithme de Gauss-Newton

L'algorithme de Gauss-Newton évite le calcul de H en exploitant le fait que la fonction à minimiser est une somme de carrés :

$$\chi^2(\beta_1 \dots \beta_p) = r_1(\vec{\beta})^2 + r_2(\vec{\beta})^2 + \dots + r_n(\vec{\beta})^2 = \vec{r} \cdot \vec{r}$$

avec les n résidus

$$r_i(\vec{\beta}) = f(t_i; \vec{\beta}) - y_i$$

Alors

$$\vec{\nabla}\chi^2 = \begin{pmatrix} 2\,r_1 \frac{\partial r_1}{\partial \beta_1} + 2\,r_2 \frac{\partial r_2}{\partial \beta_1} + \dots + 2\,r_n \frac{\partial r_n}{\partial \beta_1} \\ 2\,r_1 \frac{\partial r_1}{\partial \beta_2} + 2\,r_2 \frac{\partial r_2}{\partial \beta_2} + \dots + 2\,r_n \frac{\partial r_n}{\partial \beta_2} \\ \vdots \\ 2\,r_1 \frac{\partial r_1}{\partial \beta_p} + 2\,r_2 \frac{\partial r_2}{\partial \beta_p} + \dots + 2\,r_n \frac{\partial r_n}{\partial \beta_p} \end{pmatrix} = 2\,\vec{r}\,J$$

Ici J est la matrice jacobienne $n \times p$

$$J_{ij} = \frac{\partial r_i}{\partial \beta_j}\,.$$

De plus,

$$H_{ij} = \frac{\partial}{\partial \beta_i}\left(\nabla\chi^2\right)_j = \frac{\partial}{\partial \beta_i}\left(2\,\vec{r}\,J\right)_j = 2\left(J^T\,J\right)_{ij} + 2\vec{r}\cdot\frac{\partial^2 \vec{r}}{\partial \beta_i \partial \beta_j} \approx 2\left(J^T\,J\right)_{ij}$$

(en supposant que les termes $\vec{r}\cdot\frac{\partial^2 \vec{r}}{\partial \beta_i \partial \beta_j}$ sont négligeables).

Regression non linéaire : L'algorithme de Gauss-Newton

On a trouvé

$$\vec{\nabla}\chi^2 = 2\,\vec{r}\,J$$

et

$$H \approx 2\left(J^T\,J\right)$$

où $\vec{r}$ est le vecteur à n composantes des résidus (fonctions des paramètres β_i)

$$r_i(\vec{\beta}) = f(t_i;\vec{\beta}) - y_i$$

et J est la matrice jacobienne $n \times p$ de $\vec{r}$ par rapport à $\vec{\beta}$,

$$J = \frac{\partial \vec{r}}{\partial \vec{\beta}} = \frac{\partial \vec{f}}{\partial \vec{\beta}}\,.$$

L'itération de la méthode de Newton pour trouver un point critique de $\chi^2 = \vec{r}\cdot\vec{r}$ devient

$$\boxed{\vec{\beta} \leftarrow \vec{\beta} - (J^T J)^{-1}\vec{r}\,J}$$

Méthode de Gauss-Newton.

Regression non linéaire : L'algorithme de Gauss-Newton

En pratique :

- Calculer analytiquement les dérivées $\frac{\partial f}{\partial \beta_j}$.
- Commencer avec un ensemble de paramètres $\vec{\beta}$ au choix.
- Mettre à jour les $r_i = f(t_i; \vec{\beta}) - y_i$ et les $J_{ij} = \frac{\partial f}{\partial \beta_j}(t_i; \vec{\beta})$.
- Remplacer $\vec{\beta} \leftarrow \vec{\beta} - (J^T J)^{-1} \vec{r}\, J$.

  Équivalent, à préférer en pratique (car plus stable) : calculer le nouveau $\vec{\beta}$ avec la solution d'un système linéaire, trouvée p.ex. par la méthode de Gauss :

$$\vec{\beta}_{\text{nouveau}} = \vec{\beta}_{\text{ancien}} + \vec{\delta}\,, \qquad \vec{\delta} = \text{ (solution de } J^T J\, \vec{\delta} = -\vec{r}\, J)\,.$$

- Itérer ces dernières deux étapes jusqu'à la convergence.
  S'arrêter lorsque $||\vec{\delta}|| < \epsilon$.

Régression non linéaire : L'algorithme de Gauss-Newton

On supposera que les fonctions $\frac{\partial f}{\partial \beta_j}$ sont données dans une liste `gradf` :

```
import numpy as np
import gauss   # pour la fonction gauss() du cours

def gauss_newton(t, y, f, gradf, beta0, epsilon=1.E-4):
  beta = np.copy(beta0)       # les paramètres à ajuster
  delta = np.ones(len(beta)) # diff. entre deux itérations

  while np.sqrt(np.sum(delta**2)) > epsilon:
    r = f(t, beta) - y      # les résidus
    J = np.array([df(t, beta) for df in gradf]).T # matr. J
    delta = gauss.gauss(J.T @ J, - r @ J) # sol. du système
    beta += delta

  chi2 = np.sum(r**2)         # chi^2 après minimisation
  return beta, chi2
```

Algorithme de Gauss-Newton : Exemple d'application

Par exemple, pour la fonction sinusoïdale ci-dessus :

$$f(t;\vec{\beta}) = \beta_1 \sin(\beta_2 t + \beta_3)$$

et donc

$$\frac{\partial f}{\partial \beta_1}(t;\vec{\beta}) = \sin(\beta_2 t + \beta_3), \quad \frac{\partial f}{\partial \beta_2}(t;\vec{\beta}) = \beta_1 \cos(\beta_2 t + \beta_3)\, t,$$
$$\frac{\partial f}{\partial \beta_3}(t;\vec{\beta}) = \beta_1 \cos(\beta_2 t + \beta_3).$$

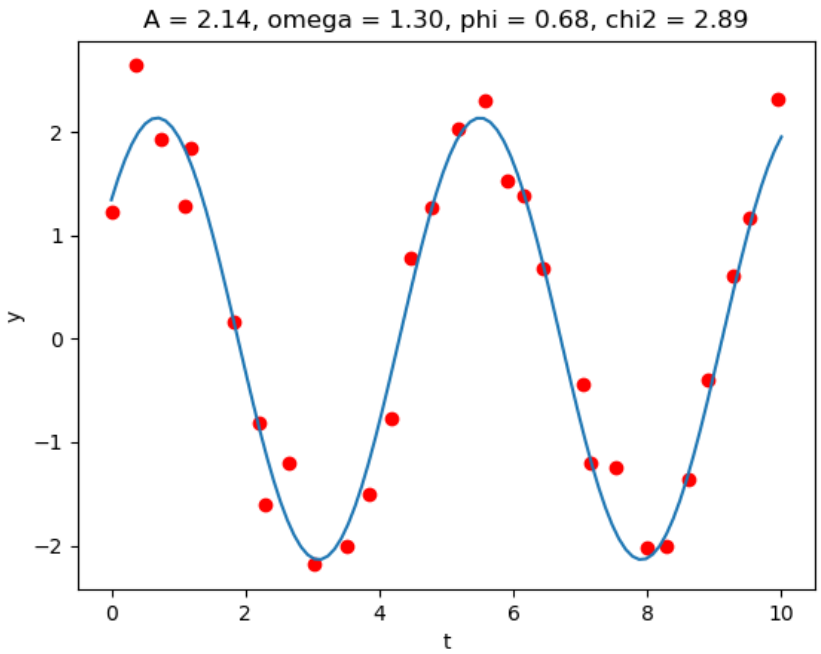
```
# la fonction modèle
def f(t, beta):
    return beta[0] * np.sin(beta[1] * t + beta[2])

# ses dérivées partielles par rapport aux paramètres
df = [lambda t, beta: np.sin(beta[1]*t + beta[2]),
      lambda t, beta: beta[0]*np.cos(beta[1]*t + beta[2])*t,
      lambda t, beta: beta[0]*np.cos(beta[1]*t + beta[2])]

data = np.loadtxt('noisysin.txt').T  # les données
beta, chi2 = gauss_newton(data[0], data[1], f, df,
                          np.array([2., 1., 0.]))
```

Algorithme de Gauss-Newton : Exemple d'application

$$f(t;\, A, \omega, \phi) = A\, \sin\left(\omega t + \phi\right)$$



Régression non linéaire : Algorithme de Levenberg-Marquardt

Faiblesse de la méthode de Gauss-Newton : si on commence avec une première estimation trop imprécise, alors l'itération ne ne trouvera pas le minimum.

Ce problème est amélioré dans une modification de l'algorithme menant à la méthode de Levenberg-Marquardt.

Gauss-Newton :

$$\vec{\beta}_{\text{nouveau}} = \vec{\beta}_{\text{ancien}} + \vec{\delta}\,, \qquad J^T J\vec{\delta} = -\vec{r}J$$

.

Levenberg-Marquardt :

$$\vec{\beta}_{\text{nouveau}} = \vec{\beta}_{\text{ancien}} + \vec{\delta}\,, \qquad \left(J^T J + \lambda \mathbb{1}\right)\vec{\delta} = -\vec{r}J$$

.

où $\lambda \geq 0$ est un paramètre dit d'amortissement, à adapter à chaque itération.

- Si $\lambda \to 0$, la méthode s'approche à celle de Gauss-Newton.
- Pour des grands λ, on s'approche à la méthode du gradient : la variation $\vec{\delta}$ suit la direction de la plus forte pente $-\vec{r}J \propto -\vec{\nabla}\chi^2$.

Régression non linéaire : Algorithme de Levenberg-Marquardt

Algorithme :

- Choisir une assez petite valeur initiale de λ (disons 10^{-4}).
- Faire tourner l'algorithme modifié de Gauss-Newton en remplaçant $J^T J \rightarrow J^T J + \lambda \mathbb{1}$.
- A chaque itération, calculer χ^2.
  - Si χ^2 a grandi par rapport à l'itération précédente, retourner à l'ancien $\vec{\beta}$ et refaire avec $\lambda \leftarrow 10\,\lambda$.
  - Si χ^2 a diminué, garder le nouveau $\vec{\beta}$ et continuer avec $\lambda \leftarrow \lambda/10$.
- S'arrêter dès que χ^2 diminue par moins que $\approx 10^{-2}$ entre deux itérations.

Propriétés :

- Par rapport à Gauss-Newton, convergence légèrement moins rapide mais plus stable.
- Beaucoup de variations et d'optimisations existent.

Optimisation

Soit f : $E \subset \mathbb{R}^n \to \mathbb{R}$ une fonction ; on cherche un minimum (local) de f.

Généralise l'ajustement où la fonction à minimiser était χ^2.

Deux classes de méthodes :

1. Méthodes nécessitant l'évaluation des dérivées de f (en supposant qu'elles existent et soient facilement calculables, au moins approximativement)
2. Méthodes ne nécessitant que le calcul des valeurs de f

Optimisation : Méthode de Newton dans n dimensions

Exemple d'une méthode avec dérivées : la méthode de Newton n-dimensionnelle.

Comme avant : un minimum de f est un zéro de $\vec{\nabla} f$; on cherche alors une solution $\vec{x}$ de

$$\vec{\nabla} f(\vec{x}) = \vec{0}$$

par l'itération

$$\vec{x} \leftarrow \vec{x} - H^{-1}(\vec{x}) \vec{\nabla} f(\vec{x})$$

avec H la matrice hesséenne des dérivées secondes.

Les éléments de $\vec{\nabla} f$ et de H peuvent s'approximer par des différences finies,

$$\frac{\partial f}{\partial x_i}(\vec{x}) = \lim_{h \to 0} \frac{f(\vec{x} + h\vec{e}_i) - f(\vec{x} - h\vec{e}_i)}{2h},$$

$$\frac{\partial^2 f}{\partial x_i \partial x_j}(\vec{x}) = \lim_{h \to 0} \frac{f(\vec{x} + h(\vec{e}_i + \vec{e}_j)) + f(\vec{x} - h(\vec{e}_i + \vec{e}_j)) - f(\vec{x} + h(\vec{e}_i - \vec{e}_j)) - f(\vec{x} - h(\vec{e}_i - \vec{e}_j))}{4\,h^2}$$

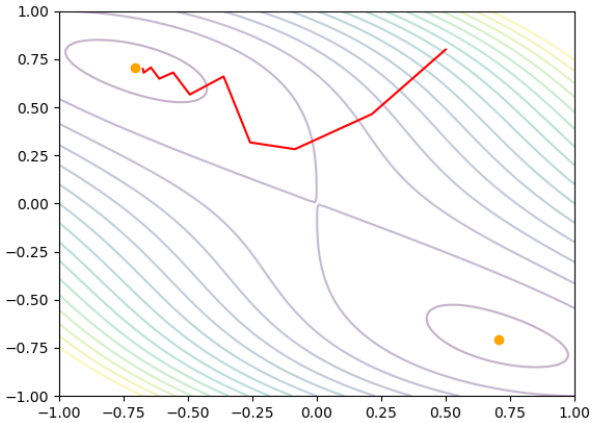
- Problème de stabilité : L'algorithme a tendance à ne pas converger si les valeurs initiales ne sont pas déjà assez proches du minimum.
- Problème de coût : Parfois l'évaluation de f nécessite un lourd calcul numérique $\Rightarrow$ intérêt de minimiser le nombre d'appels de fonction à f. Le calcul de H coûte cher.

Optimisation : Méthode du gradient

Idée : Descendre dans la direction de la plus grande pente,

$$\vec{x} \leftarrow \vec{x} - \lambda \vec{\nabla} f(\vec{x}) \, .$$

Ici le paramètre $\lambda > 0$ peut varier (on réduira λ quand on est proche du minimum).

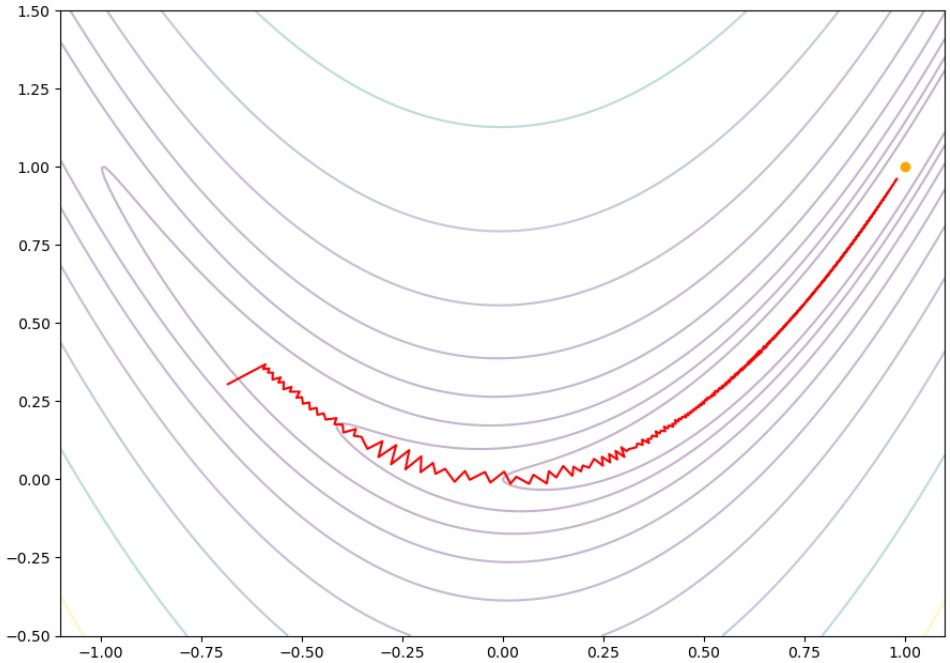


Algorithme :

1. Commencer avec une première estimation $\vec{x} = \vec{x}_0$ et une valeur de départ $\lambda = \lambda_0$.
2. Numériquement calculer $\vec{\nabla} f(\vec{x})$.
3. Si $f(\vec{x} - \lambda \vec{\nabla} f(\vec{x})) > f(\vec{x})$, réessayer avec $\lambda \leftarrow \lambda/2$.
   Si $f(\vec{x} - \lambda \vec{\nabla} f(\vec{x})) < f(\vec{x})$, remplacer $\vec{x} \leftarrow \vec{x} - \lambda \vec{\nabla} f(\vec{x})$ et tester convergence.
4. Si pas de convergence, continuer à partir de (2) avec $\lambda \leftarrow 2\,\lambda$.

Optimisation : Méthode du gradient

Problème : Convergence très mauvaise si le minimum est dans un "fond de vallée allongée". Exemple :



Après 2500 itérations on n'a toujours pas trouvé le minimum au point orange, le chemin de descente zigzague et ne s'y approche que très lentement. Par un choix optimal de λ (minimiser la fonction le long de la direction du gradient) on pourrait légèrement améliorer la vitesse de convergence, sans pourtant complètement résoudre ce problème.

Optimisation : Méthode du gradient conjugué

Mieux : Au lieu de suivre la direction du gradient à chaque itération, suivre un chemin de directions $\vec{h}_i$ tel que, autant que possible,

$$\vec{h}_i \cdot H_f \cdot \vec{h}_j = 0 \qquad \forall\, j < i\,.$$

Il se trouve qu'on peut construire les $\vec{h}_i$ approximativement sans connaître H_f. Méthode du gradient conjugué.

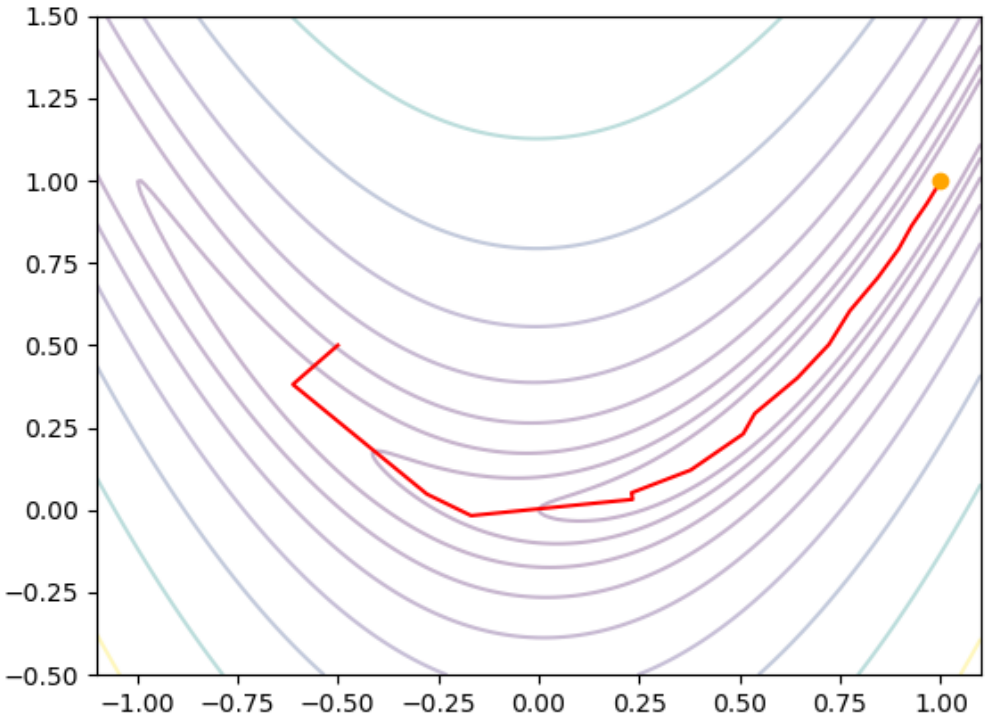
Algorithme :

1. Commencer à $\vec{x} = \vec{x}_0$. Calculer $\vec{\nabla} f(\vec{x}_0)$ et poser $\vec{h}_0 = \vec{g}_0 = -\vec{\nabla} f(\vec{x}_0)$.
2. Minimiser f le long de la droite $\vec{x}_0 + \lambda \vec{h}_0$ par rapport à λ (p.ex. par la méthode de Newton 1-dimensionnelle), ce qui donne $\vec{x}_1$.
   (Ce premier pas = un pas de la méthode du gradient avec λ optimal.)
3. Calculer $\vec{g}_i = -\vec{\nabla} f(\vec{x}_i)$. Poser

   $$\gamma_i = \max\left(\frac{(\vec{g}_i - \vec{g}_{i-1}) \cdot \vec{g}_i}{\vec{g}_{i-1} \cdot \vec{g}_{i-1}},\ 0 \right) \qquad \text{“préscription de Polak-Ribière”}$$

   et $\vec{h}_i = \vec{g}_i + \gamma_i \vec{h}_{i-1}$.
4. Minimiser f le long de la droite $\vec{x}_i + \lambda \vec{h}_i$, ce qui donne $\vec{x}_{i+1}$.
5. Itérer à partir de (3) jusqu'à la convergence.

Optimisation : Méthode du gradient conjugué

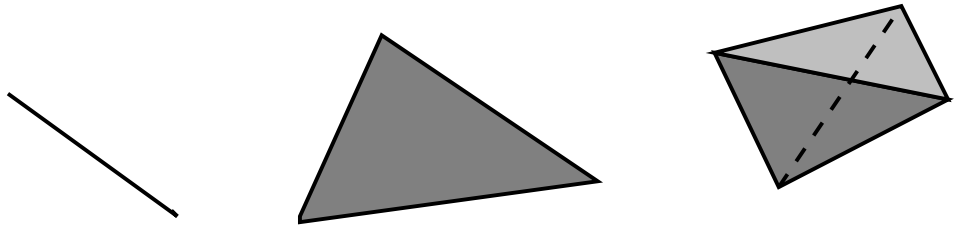


(20 itérations pour trouver le minimum à une précision de 10^{-5})

Optimisation sans dérivées : la méthode de Nelder-Mead

Un simplexe en n dimensions est donné par $n+1$ points non dégénérés. Par exemple :

- en $n = 1$, deux points non dégénérés forment un segment de droite,
- en $n = 2$, trois points non dégénérés forment un triangle,
- en $n = 3$, quatre points non dégénérés forment un tétraèdre.



Optimisation sans dérivées : la méthode de Nelder-Mead

Le point de départ de la méthode de Nelder-Mead est un simplexe que l'on transforme de façon répétée afin de trouver un minimum de la fonction f :

- réflexion d'un point par rapport à la face opposé,
- réflexion et expansion,
- contraction dans une direction,
- rétrécissement du simplexe entier.

Optimisation sans dérivées : la méthode de Nelder-Mead

Algorithme :

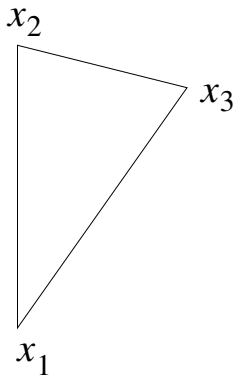
1. Commencer avec un n-simplexe $\vec{x}_1 \dots \vec{x}_{n+1}$.
2. Tester convergence. Trier les points de façon que $f(\vec{x}_1) \leq \dots \leq f(\vec{x}_n) \leq f(\vec{x}_{n+1})$.
3. Calculer $\vec{x}_0$, le barycentre des points $\vec{x}_1 \dots \vec{x}_n$:

$$\vec{x}_0 = \frac{1}{n} \sum_{k=1}^{n} \vec{x}_k \, .$$

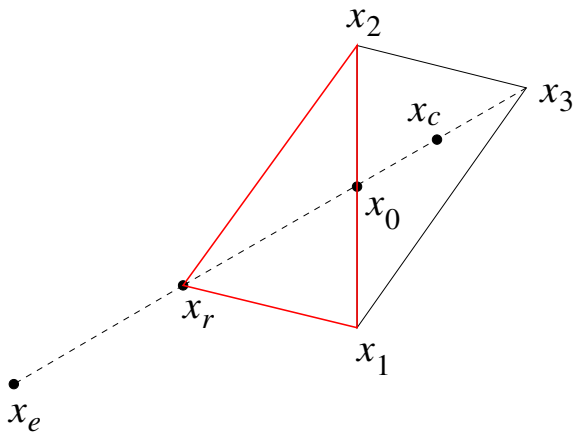
4. Calculer $\vec{x}_r$, obtenu par réflexion de $\vec{x}_{n+1}$ par rapport à $\vec{x}_0$: $\vec{x}_r = 2\vec{x}_0 - \vec{x}_{n+1}$.
5. Si $f(\vec{x}_1) \leq f(\vec{x}_r) \leq f(\vec{x}_n)$, réflexion : $\vec{x}_{n+1} \leftarrow \vec{x}_r$.
6. Sinon, si $f(\vec{x}_r) < f(\vec{x}_1)$, calculer $\vec{x}_e = 2\vec{x}_r - \vec{x}_0$.
   - Si $f(\vec{x}_e) < f(\vec{x}_r)$, réflexion et expansion : $\vec{x}_{n+1} \leftarrow \vec{x}_e$.
   - Sinon, réflexion : $\vec{x}_{n+1} \leftarrow \vec{x}_r$.
7. Sinon, calculer $\vec{x}_c = \frac{1}{2}\vec{x}_0 + \frac{1}{2}\vec{x}_{n+1}$.
   - Si $f(\vec{x}_c) < f(\vec{x}_{n+1})$, contraction : $\vec{x}_{n+1} \leftarrow \vec{x}_c$.
   - Sinon, rétrécissement : $\vec{x}_i, \leftarrow \frac{1}{2}\vec{x}_1 + \frac{1}{2}\vec{x}_i$ $(i = 2 \dots n+1)$.
8. Itérer à partir de (2).

Optimisation sans dérivées : la méthode de Nelder-Mead

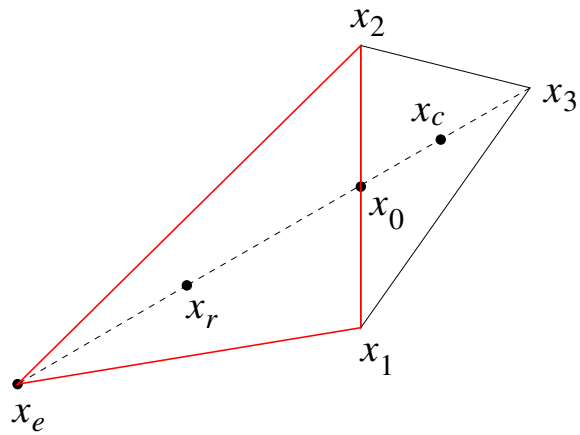
Simplexe de départ :



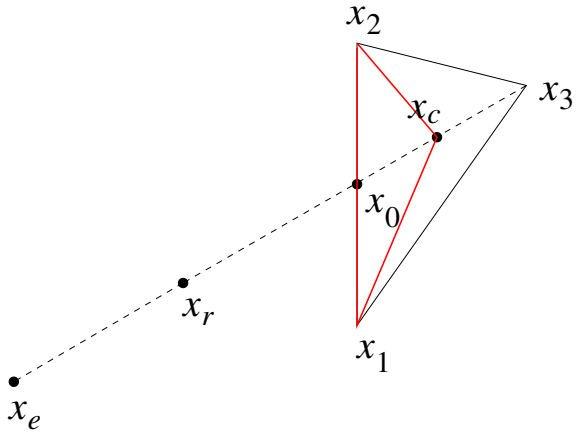
Réflexion :



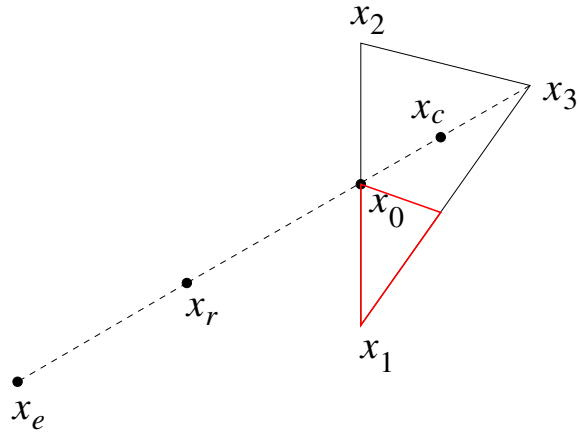
Réflexion + expansion :



Contraction :



Rétrécissement :



Applications II : Problèmes de Sturm-Liouville

Dans ce chapitre

Python

- Aperçu de la bibliothèque SciPy pour la programmation scientifique avec Python
- Le mode interactif de Python avec l'interface Jupyter

SciPy

La bibliothèque `SciPy` (`http://www.scipy.org`) est une extension de NumPy qui fournit de la fonctionnalité supplémentaire pour la programmation scientifique et en particulier pour la physique numérique.

Une grande partie des routines de SciPy prend appui sur les bibliothèques de logiciels d'analyse numérique du site `Netlib` (`http://www.netlib.org/`) dont la plupart est implémentée en `C` ou `FORTRAN`. Pour cette raison les fonctions de SciPy sont typiquement beaucoup plus efficaces qu'une fonction équivalente entièrement implémentée en Python. De plus, les bibliothèques de `Netlib` ont été testés et debougés pendant des dizaines d'années, alors elles sont relativement fiables.

SciPy

On a déjà rencontré la sous-bibliothèque `scipy.linalg` contenant des routines d'algèbre linéaire numérique. D'autres modules sont :

- `scipy.constants` qui contient les valeurs numériques de nombreuses constantes physiques
- `scipy.special` qui d'définit des fonctions speciales comme les fonctions d'Airy, de Bessel (et leurs primitives et dérivées), des polynômes orthogonaux, la fonction Gamma et les fonctions liées, les fonctions elliptiques, hypergéométriques ...
- `scipy.integrate` pour le calcul numérique des intégrales des fonctions numériques, ainsi que pour la solution numérique des systèmes d'équations différentielles avec des conditions initiales
- `scipy.optimize` pour la maximisation ou minimisation numérique des fonctions à plusieurs variables et pour la recherche des zéros
- `scipy.sparse` pour des méthodes du calcul matriciel et l'algèbre linéaire optimisées pour les matrices creuses

SciPy

- `scipy.interpolate` pour l'interpolation entre des points de données discrètes
- `scipy.fftpack` pour le calcul des transformations de Fourier
- `scipy.signal` pour des méthodes de traitement de signal
- `scipy.stats` pour des méthodes de statistique
- . . . et de nombreux autres.

Python en mode interactif avec Jupyter

Ici on ne va pas systématiquement discuter toute la fonctionnalité de SciPy. Nous allons plutôt découvrir quelques aspects sélectionnés de l'utilisation de SciPy avec un interface interactif : le notebook graphique `Jupyter`.

- Pour travailler avec les fichiers d'exemples, télécharger les trois fichiers `LennardJones.ipynb`, `Membrane.ipynb` et `Corde.ipynb` de la page `Moodle` du cours.
- Pour lancer l'interface : dans une console, entrer `jupyter notebook`. Sélectionner p.ex. le fichier `LennardJones.ipynb`.
- Le contenu du fichier est divisé en cellules : il y a des cellules de text avec des explications et des cellules de code source. Pour exécuter le code dans une cellule, la marquer et entrer `[Shift] + [Enter]`.