

# L1 - Calcul Scientifique



Youssef Chahir  
[youssef.chahir@unicaen.fr](mailto:youssef.chahir@unicaen.fr)

Université de Caen Normandie

1

## Chapitre V : Interpolation et Ajustement des courbes

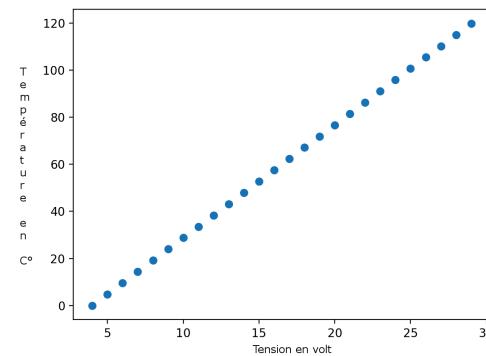
### Introduction

- Pour plusieurs applications, nous utilisons des trajectoires i.e. des chemins qui représentent le parcours à suivre par un objet.
- Toutes les caractéristiques d'une scène peuvent varier dans le temps :
  - un point de référence sur un objet de la scène,
  - la position de la caméra,
  - la couleur d'un objet,
  - la position d'une source lumineuse,
  - etc.
- Les données numériques sont généralement fournies sous la forme de tableaux

$x_0$	$x_1$	$x_2$	$\dots$	$x_n$
$y_0$	$y_1$	$y_2$	$\dots$	$y_n$

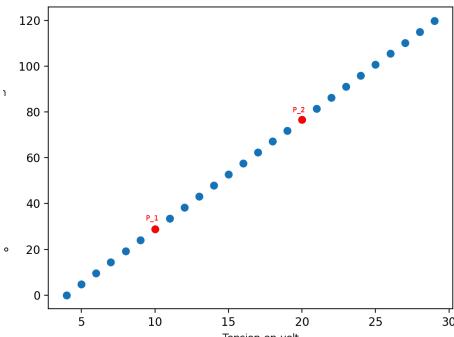
### Différents scénarios

- 1er scénario:
  - Toutes les mesures sont parfaitement alignées. Il suffit de trouver l'équation de la droite correspondant aux mesures.



## Trouver équation de la droite

- Pour calculer l'équation de la droite, il suffit d'avoir seulement deux points
- Par exemple pour notre exemple P(10 ; 28.76) P.(20 ; 76.70).
- Pour une droite de la forme :  $y = mx + p$



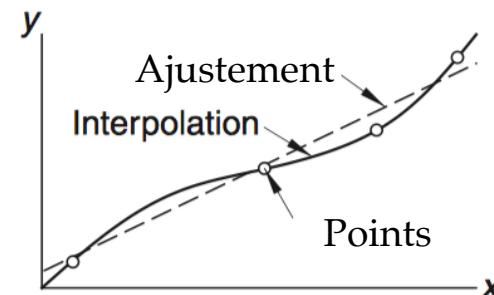
$$m = \frac{y_{P_1} - y_{P_2}}{x_{P_1} - x_{P_2}} = \frac{76.70 - 28.76}{20 - 10} = 4.794$$

$$b = y_{P_1} - mx_{P_1} = 28.76 - 4.794 \times 10 = -19.179999$$

6

## Interpolation ou Ajustement ?

- Étant donné  $n+1$  points  $(x_i, y_i)$  avec  $i=0 \dots n$ , on cherche à estimer  $y(x)$  ?
- Nous distinguons :
  - L'interpolation** : qui doit passer par les points
  - L'ajustement** de courbe qui considère que les données sont bruitées et que la courbe ne doit pas nécessairement passer par les points

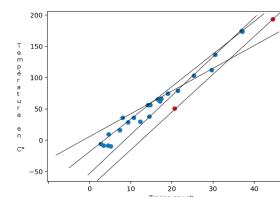
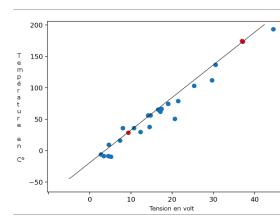
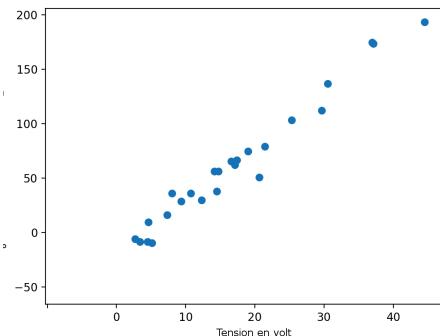


8

## Différents scénarios

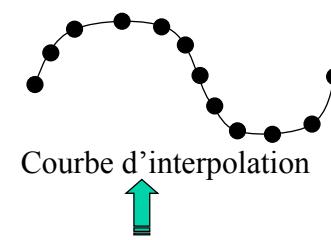
- 2nd scénario:

- Mesures réelles : Les imprécisions de la mesure, les variabilités électroniques... font que les points réellement mesurés ne sont pas parfaitement alignés.
- Plusieurs droites différentes sont possibles.

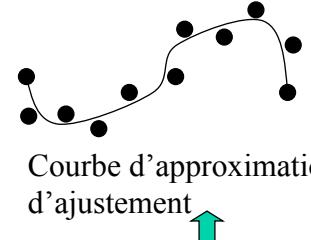


7

## Interpolation ou Approximation ?



Courbe d'interpolation  
Positions effectives par lesquelles la courbe doit passer.



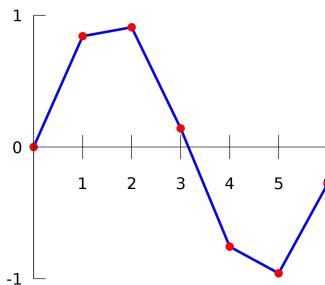
Courbe d'approximation ou d'ajustement  
Échantillon de points rapprochés de la courbe

9

# Interpolation linéaire

- [https://fr.wikipedia.org/wiki/Interpolation\\_linéaire](https://fr.wikipedia.org/wiki/Interpolation_linéaire)

- Interpolation linéaire : méthode la plus simple pour estimer la valeur prise par une fonction continue entre 2 points
- Fonction affine de la forme  $f(x) = m \cdot x + b$  passant par les deux points déterminés.
- Emploi systématique lorsque l'on ne disposait que de tables numériques pour le calcul.



Les points **rouges** correspondent aux points  $(x_k, y_k)$ , et la courbe **bleue** représente la fonction d'interpolation, composée de segments de droite.

10

# Interpolation polynomiale : méthode de Lagrange

- Méthode d'interpolation consistant à faire passer un polynôme de degré  $n$  qui passe par les  $n+1$  points.

$$P_n(x) = \sum_{i=0}^n y_i \ell_i(x)$$

$$\ell_i(x) = \frac{x - x_0}{x_i - x_0} \cdot \frac{x - x_1}{x_i - x_1} \cdots \frac{x - x_{i-1}}{x_i - x_{i-1}} \cdot \frac{x - x_{i+1}}{x_i - x_{i+1}} \cdots \frac{x - x_n}{x_i - x_n}$$

$$= \prod_{\substack{j=0 \\ j \neq i}}^n \frac{x - x_j}{x_i - x_j}, \quad i = 0, 1, \dots, n$$

# Interpolation linéaire

- Approcher la fonction  $f$  par la fonction affine  $\bar{f}$  telle que :

$$\bar{f}(x_a) = y_a \quad \text{et} \quad \bar{f}(x_b) = y_b$$

- Résolution d'un système de 2 équations à 2 inconnues :

$$\bar{f}(x) = \frac{x_b - x}{x_b - x_a} y_a + \frac{x - x_a}{x_b - x_a} y_b$$

12

# Interpolation polynomiale : méthode de Lagrange

- Les termes  $\ell_i(x)$  sont appelés fonctions cardinales
- Cas  $n = 1$  (2 points) :

$$P_1(x) = y_0 \ell_0(x) + y_1 \ell_1(x)$$

$$\ell_0(x) = \frac{x - x_1}{x_0 - x_1}$$

- Cas  $n = 2$  :

$$P_2(x) = y_0 \ell_0(x) + y_1 \ell_1(x) + y_2 \ell_2(x) \quad \ell_0(x) = \frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)}$$

$$\ell_1(x) = \frac{(x - x_0)(x - x_2)}{(x_1 - x_0)(x_1 - x_2)}$$

$$\ell_2(x) = \frac{(x - x_0)(x - x_1)}{(x_2 - x_0)(x_2 - x_1)}$$

13

# Propriétés

$$l_i(x) = \frac{(x - x_0)(x - x_1)\dots(x - x_{i-1})(x - x_{i+1})\dots(x - x_n)}{(x_i - x_0)(x_i - x_1)\dots(x_i - x_{i-1})(x_i - x_{i+1})\dots(x_i - x_n)}.$$

- Le numérateur est un polynôme de degré n
- Le dénominateur est une constante :
- i)  $l_i$  est un polynôme de degré n
- ii)  $l_i(x_k) = 0$  si  $i \neq k$  et  $0 \leq k \leq n$
- iii)  $l_i(x_i) = 1$ .

- Propriété des fonctions cardinales :

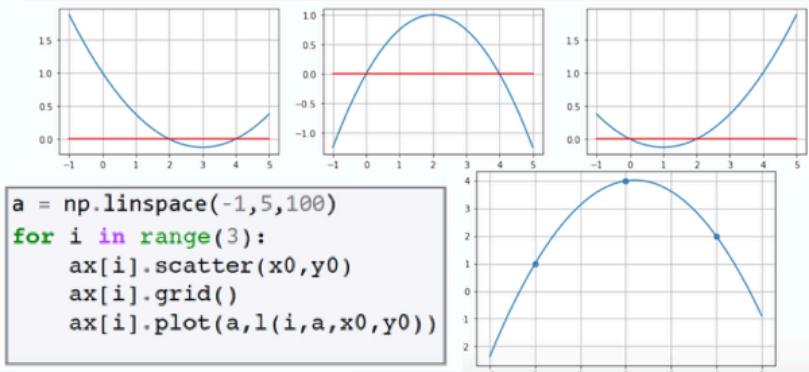
$$\ell_i(x_j) = \begin{cases} 0 & \text{if } i \neq j \\ 1 & \text{if } i = j \end{cases} = \delta_{ij}$$

- Long à calculer

14

## Interpolation polynomiale : méthode de Lagrange (suite)

```
def l(i,x,x0,y0) :
    x = np.reshape(x,(100,1))
    n = x0.shape[0]
    x0 = np.reshape(x0,(1,n))
    ind = np.ones(n, bool)
    ind[i] = False
    return y0[i]*np.prod(x-x0[:,ind],axis=1)/np.prod(x0[:,i]-x0[:,ind]).T
```



16

# Interpolation : méthode de Newton

- Polynôme de la forme :

$$P_n(x) = a_0 + (x - x_0)a_1 + (x - x_0)(x - x_1)a_2 + \dots + (x - x_0)(x - x_1)\dots(x - x_{n-1})a_{n-1}$$

- Exemple  $n=3$  :

$$P_3(x) = a_0 + (x - x_0)a_1 + (x - x_0)(x - x_1)a_2 + (x - x_0)(x - x_1)(x - x_2)a_3$$

$$= a_0 + (x - x_0) \{ a_1 + (x - x_1) [a_2 + (x - x_2)a_3] \}$$

$\overbrace{\hspace{10em}}$   $P_1$   
 $\overbrace{\hspace{10em}}$   $P_2$   
 $P_0(x) = a_3 \overbrace{\hspace{10em}}$   $P_3$

$$P_1(x) = a_2 + (x - x_2)P_0(x)$$

$$P_2(x) = a_1 + (x - x_1)P_1(x)$$

$$P_3(x) = a_0 + (x - x_0)P_2(x)$$

- Pour n arbitraire, cela donne :

$$P_0(x) = a_n \quad P_k(x) = a_{n-k} + (x - x_{n-k})P_{k-1}(x), \quad k = 1, 2, \dots, n$$

- Évaluation efficace au moyen d'une structure itérative (boucle)

## Interpolation : méthode de Newton (suite)

- Estimation des paramètres :

$$y_0 = a_0$$

$$y_1 = a_0 + (x_1 - x_0)a_1$$

$$y_2 = a_0 + (x_2 - x_0)a_1 + (x_2 - x_0)(x_2 - x_1)a_2$$

⋮

$$y_n = a_0 + (x_n - x_0)a_1 + \dots + (x_n - x_0)(x_n - x_1)\dots(x_n - x_{n-1})a_n$$

- En posant :  $n_j(x) = \prod_{0 \leq i < j} (x - x_i) \quad j = 0, \dots, k$

- On a  $\sum_{j=0}^i a_j n_j(x_i) = y_i \quad i = 0, \dots, k,$

## Interpolation : méthode de Newton (suite)

- Estimation des paramètres (suite)

$$\begin{pmatrix} 1 & & & & 0 \\ 1 & x_1 - x_0 & & & \\ 1 & x_2 - x_0 & (x_2 - x_0)(x_2 - x_1) & & \\ \vdots & \vdots & \ddots & & \\ 1 & x_k - x_0 & \dots & \dots & \prod_{j=0}^{k-1} (x_k - x_j) \end{pmatrix} \begin{pmatrix} a_0 \\ \vdots \\ a_k \end{pmatrix} = \begin{pmatrix} y_0 \\ \vdots \\ y_k \end{pmatrix}.$$

$$\nabla y_i = \frac{y_i - y_0}{x_i - x_0}, \quad i = 1, 2, \dots, n$$

$$\nabla^2 y_i = \frac{\nabla y_i - \nabla y_1}{x_i - x_1}, \quad i = 2, 3, \dots, n$$

$$\nabla^3 y_i = \frac{\nabla^2 y_i - \nabla^2 y_2}{x_i - x_2}, \quad i = 3, 4, \dots, n$$

⋮

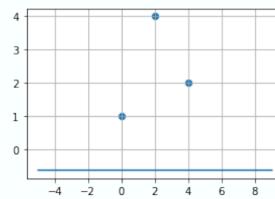
$$\nabla^n y_n = \frac{\nabla^{n-1} y_n - \nabla^{n-1} y_{n-1}}{x_n - x_{n-1}}$$

- $1 \times a_0 + (x_1 - x_0)a_1 = y_1$   
D'où :  $\nabla y_1 = a_1 = -\frac{x_1 - x_0}{y_1 - y_0}$
- $1 \times a_0 + (x_1 - x_0)a_1 + (x_1 - x_0)(x_2 - x_0)a_2 = \dots$   
 $\nabla y_2 = \frac{y_2 - 1 \times a_0 + (x_1 - x_0)a_1}{(x_1 - x_0)(x_2 - x_0)}$   
D'où :
- $= \frac{\nabla y_2 - \nabla y_1}{x_2 - x_0}$

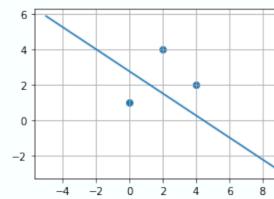
- D'où :  $a_0 = y_0 \quad a_1 = \nabla y_1 \quad a_2 = \nabla^2 y_2 \quad \dots \quad a_n = \nabla^n y_n$

## Interpolation : méthode de Newton (suite)

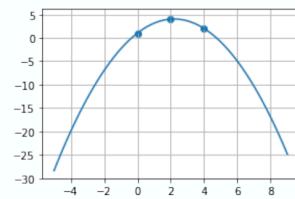
```
f,ax = plt.subplots(1,x0.shape[0],figsize=(15,3))
u = np.linspace(x0[0]-5,x0[-1]+5,100)
for i in range(x0.shape[0]):
    ax[i].grid()
    ax[i].plot(u,P(u,i,a,x0,y0))
    ax[i].scatter(x0,y0)
```



P0(x)



P1(x)



P2(x)

## Interpolation polynomiale : méthode de Neville

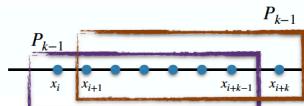
- Méthode de Newton nécessite de calculer tous les coefficients avant de pouvoir évaluer la fonction
- Intéressant si beaucoup d'interpolations pour mêmes données
- Si besoin d'interpoler 1 seul point, peu intéressant -> Neville
- Soit  $P_k[x_i, x_{i+1}, \dots, x_{i+k}]$  un polynôme de degré  $k$  passant par  $k+1$  points  $(x_i, y_i)$ .
- Pour un point :  $P_0[x_i] = y_i$
- Pour 2 points :  $P_1[x_i, x_{i+1}] = \frac{(x - x_{i+1})P_0[x_i] + (x_i - x)P_0[x_{i+1}]}{x_i - x_{i+1}}$

passe bien par les deux points

- On peut montrer récursivement que

$$P_k[x_i, x_{i+1}, \dots, x_{i+k}]$$

$$= \frac{(x - x_{i+k})P_{k-1}[x_i, x_{i+1}, \dots, x_{i+k-1}] + (x_i - x)P_{k-1}[x_{i+1}, x_{i+2}, \dots, x_{i+k}]}{x_i - x_{i+k}}$$



est un polynôme de degré  $n$  passant par tous points

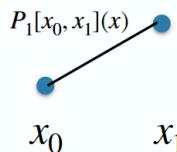
```
def d(j,i,x0,y0):
    if j==1:
        print(i)
        return (y0[i]-y0[0])/(x0[i]-x0[0])
    else:
        return (d(j-1,i,x0,y0)-d(j-1,j-1,x0,y0))/(x0[i]-x0[j-1])

n = x0.shape[0]
a = np.zeros(n)
a[0] = y0[0]

for i in range(1,n):
    a[i] = d(i,i,x0,y0)
```

```
def P(x,i,a,x0,y0):
    n = x0.shape[0]
    if i==0:
        return np.ones_like(x) * a[n-i-1]
    else :
        return np.ones_like(x) * a[n-i-1] + (x-x0[n-i-1])*P(x,i-1,a,x0,y0)
```

## Interpolation polynomiale : méthode de Neville



$\rightarrow_0$  : interpolation avec 1 seul point :

$$P_0[x_i](x) = y_i$$

$\rightarrow_1$  : interpolation avec 2 points :

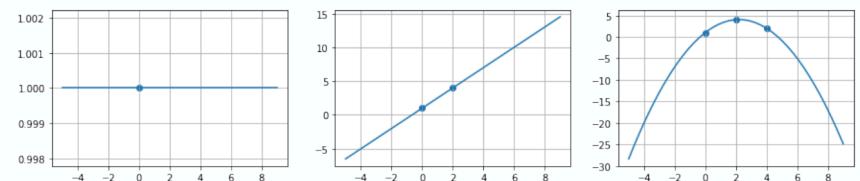
$$P_1[x_0, x_1](x) = \frac{(x - x_0)P_0[x_0](x) + (x_1 - x)P_0[x_1](x)}{x_0 - x_1}$$

(passe bien par les deux points)

## Interpolation polynomiale : méthode de Neville

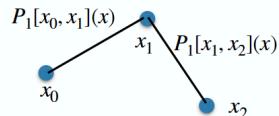
```
def P(x,x0,y0):
    if x0.shape[0]==1:
        return y0[0]*np.ones_like(x)
    else:
        return ((x-x0[-1])*P(x,x0[0:-1],y0[0:-1])+(x0[0]*x)*P(x,x0[1:],y0[1:]))/(x0[0]-x0[-1])

f,ax = plt.subplots(1,x0.shape[0],figsize=(15,3))
x = np.linspace(x0[0]-5,x0[-1]+5,100)
for i in range(x0.shape[0]):
    ax[i].grid()
    ax[i].plot(x,P(x,x0[0:i+1],y0[0:i+1]))
    ax[i].scatter(x0[0:i+1],y0[0:i+1])
```



## Limitation des interpolations polynomiales

$\rightarrow_2$  : interpolation avec 2 points :



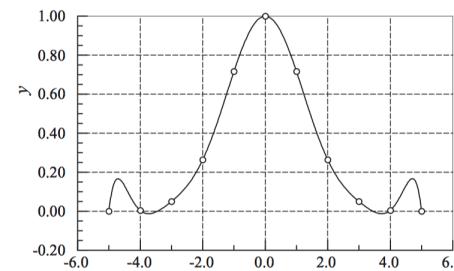
$$P_2[x_0, x_1, x_2](x) = \frac{(x - x_0)P_1[x_1, x_2](x) + (x_2 - x)P_1[x_0, x_1](x)}{x_0 - x_2}$$

Vérification que P2 passe par les 3 points :

$$P_2[x_0, x_1, x_2](x_0) = \frac{(x_2 - x_0)P_1[x_0, x_1](x_0)}{x_0 - x_2} = y_0$$

$$\begin{aligned} P_2[x_0, x_1, x_2](x_1) &= \frac{(x_1 - x_0)P_1[x_1, x_2](x_1) + (x_2 - x_1)P_1[x_0, x_1](x_1)}{x_0 - x_2} \\ &= \frac{(x_1 - x_0)y_1 + (x_2 - x_1)y_1}{x_0 - x_2} = v. \end{aligned}$$

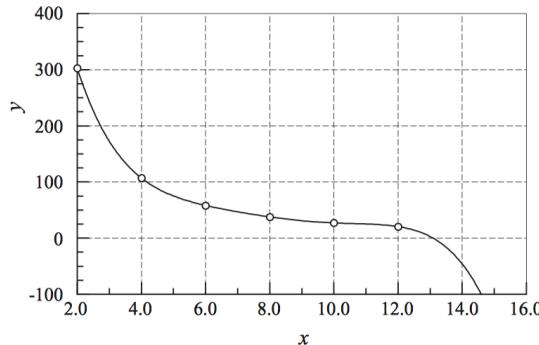
- Donne de bons résultats pour un petit nombre de points (moins que 6)
- Dès qu'il y a beaucoup de points, risques d'oscillations :



- L'extrapolation (prédition en dehors du domaine défini par les points) est dangereuse

## Limitation des interpolations polynomiales (suite)

- L'extrapolation (prédiction en dehors du domaine défini par les points) est dangereuse



29

## Interpolation par des splines cubiques (suite)

- Méthode de résolution : interpolation par n fonctions d'interpolation (chacune sur 1 intervalle)
- on commence par forcer les dérivées secondes à être égales entre deux fonctions pour le même point et on Interpole entre

$$f''_{i,i+1}(x) = k_i \ell_i(x) + k_{i+1} \ell_{i+1}(x)$$

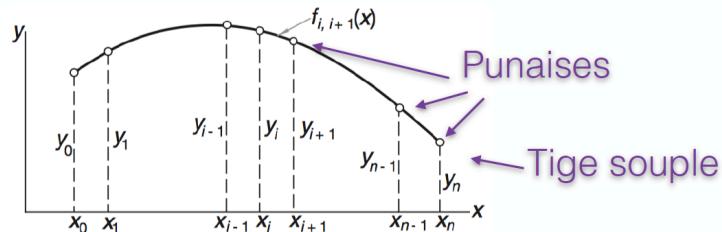
il s'agit d'une interpolation de Lagrange sur les valeurs de  $k_i$  (valeur de la dérivée seconde au point i)

$$\ell_i(x) = \frac{x - x_{i+1}}{x_i - x_{i+1}} \quad \ell_{i+1}(x) = \frac{x - x_i}{x_{i+1} - x_i}$$

31

## Interpolation par des splines cubiques

- Dès qu'il y a beaucoup de points, les splines cubiques sont préférables aux polynômes (moins d'oscillations)
- Connexion élastique entre les points :



- Polynômes de degré 3,  $f(x) = ax^2 + bx + c$  tels que  $f''_{i-1,i}(x_i) = f''_{i,i+1}(x_i) = k_i$  c'est-à-dire interpolation par morceau avec égalité des dérivées secondes entre les morceaux.
- Extrémités :  $k_0 = k_n = 0$
- On recherche également une continuité des pentes :  $f'_{i-1,i}(x_i) = f'_{i,i+1}(x_i)$

## Interpolation par des splines cubiques (suite)

- En intégrant 2 fois l'expression :

$$f''_{i,i+1}(x) = \frac{k_i(x - x_{i+1}) - k_{i+1}(x - x_i)}{x_i - x_{i+1}}$$

- Nous obtenons :

$$f_{i-1,i+1}(x) = \frac{k_i(x - x_{i+1})^3 - k_{i+1}(x - x_i)^3}{6(x_i - x_{i+1})} + A(x - x_{i+1}) - B(x - x_i)$$

remarque : les constantes d'intégration sont normalement  $Ax+B$ , mais on peut les écrire sous la forme ci-dessus sans que cela ne change rien au résultat

32

## Interpolation par des splines cubiques (suite)

- On part de (cf transparent précédent) :

$$f_{i,i+1}(x) = \frac{k_i(x - x_{i+1})^3 - k_{i+1}(x - x_i)^3}{6(x_i - x_{i+1})} + A(x - x_{i+1}) - B(x - x_i)$$

- En imposant  $f_{i,i+1}(x_i) = y_i$  on obtient :  $\frac{k_i(x_i - x_{i+1})^3}{6(x_i - x_{i+1})} + A(x_i - x_{i+1}) = y_i$   
d'où :  $A = \frac{y_i}{x_i - x_{i+1}} - \frac{k_i}{6}(x_i - x_{i+1})$
- de même  $f_{i,i+1}(x_{i+1}) = y_{i+1}$

donne :  $B = \frac{y_{i+1}}{x_i - x_{i+1}} - \frac{k_{i+1}}{6}(x_i - x_{i+1})$

Donc en connaissant les valeurs des  $k_i$  on peut trouver  $f_{i,i+1}(x)$   
en remplaçant A et B dans l'équation du haut

33

## Interpolation par des splines cubiques (suite)

- Nous obtenons ainsi :

$$\begin{aligned} f_{i,i+1}(x) &= \frac{k_i}{6} \left[ \frac{(x - x_{i+1})^3}{x_i - x_{i+1}} - (x - x_{i+1})(x_i - x_{i+1}) \right] \\ &\quad - \frac{k_{i+1}}{6} \left[ \frac{(x - x_i)^3}{x_i - x_{i+1}} - (x - x_i)(x_i - x_{i+1}) \right] \\ &\quad + \frac{y_i(x - x_{i+1}) - y_{i+1}(x - x_i)}{x_i - x_{i+1}} \end{aligned}$$

- Pour trouver les  $k_i$ , on suppose que les intervalles sont fixes :

$$x_{i-1} - x_i = x_i - x_{i+1} = -h$$

- En prenant les valeurs des fonctions aux points  $x_0, x_1, \dots$ , etc., on obtient un système d'équations :

$$k_{i-1} + 4k_i + k_{i+1} = \frac{6}{h^2}(y_{i-1} - 2y_i + y_{i+1}), \quad i = 1, 2, \dots, n-1$$

- En résolvant ce système, on trouve les valeurs des  $k_i$

34

## Interpolation avec scipy

- 

## Le package ‘interpolation’ de scipy

### Interpolation (scipy.interpolate)¶

Sub-package for objects used in interpolation.

As listed below, this sub-package contains spline functions and classes, one-dimensional and multi-dimensional (univariate and multivariate) interpolation classes, Lagrange and Taylor polynomial interpolators, and wrappers for [FITPACK](#) and [DFITPACK](#) functions.

#### Univariate interpolation

<code>Interp1d(x, y[, kind, axis, copy, ...])</code>	Interpolate a 1-D function.
<code>BarycentricInterpolator([xi[, yi[, axis]])</code>	The interpolating polynomial for a set of points.
<code>KroghInterpolator(xi, yi[, axis])</code>	Interpolating polynomial for a set of points.
<code>PchipInterpolator(x, y[, axis, extrapolate])</code>	PCHIP 1-d monotonic cubic interpolation.
<code>barycentric_interpolate(xi, yi, xl[, axis])</code>	Convenience function for polynomial interpolation.
<code>krogh_interpolate(xi, yi, xl[, der, axis])</code>	Convenience function for polynomial interpolation.
<code>pchip_interpolate(xi, yi, xl[, der, axis])</code>	Convenience function for pchip interpolation.
<code>Akima1DInterpolator(x, y[, axis])</code>	Akima interpolator
<code>CubicSpline(x, y[, axis, bc_type, extrapolate])</code>	Cubic spline data interpolator.
<code>PPoly(c, xl[, extrapolate, axis])</code>	Piecewise polynomial in terms of coefficients and breakpoints
<code>BPoly(c, xl[, extrapolate, axis])</code>	Piecewise polynomial in terms of coefficients and breakpoints.

#### Multivariate interpolation

Unstructured data:

<code>griddata(points, values, xi[, method, ...])</code>	Interpolate unstructured D-dimensional data.
<code>LinearNDInterpolator(points, values[, ...])</code>	Piecewise linear interpolant in N dimensions.
<code>NearestNDInterpolator(x, y)</code>	Nearest-neighbour interpolation in N dimensions.
<code>CloughTocher2DInterpolator(points, values[, tol])</code>	Piecewise cubic, C1 smooth, curvature-minimizing interpolant in 2D.
<code>Rbf(*args)</code>	A class for radial basis function approximation/interpolation of n-dimensional scattered data.
<code>interp2d(x, y, z[, kind, copy, ...])</code>	Interpolate over a 2-D grid.

36

## Le package Optimization and root finding (scipy.optimize)

### Optimization

#### Local Optimization

`minimize(fun, x0[, args, method, jac, hess, ...])` Minimization of scalar function of one or more variables.

`minimize_scalar(fun[, bracket, bounds, ...])` Minimization of scalar function of one variable.

`OptimizeResult` Represents the optimization result.

`OptimizeWarning`

The `minimize` function supports the following methods:

- `minimize(method='Nelder-Mead')`
- `minimize(method='Powell')`
- `minimize(method='CG')`
- `minimize(method='BFGS')`
- `minimize(method='Newton-CG')`
- `minimize(method='L-BFGS-B')`
- `minimize(method='TNC')`
- `minimize(method='COBYLA')`
- `minimize(method='SLSQP')`
- `minimize(method='dogleg')`
- `minimize(method='trust-ncg')`
- `minimize(method='trust-krylov')`
- `minimize(method='trust-exact')`

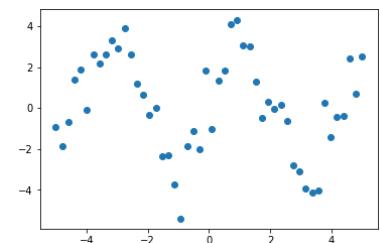
The `minimize_scalar` function supports the following methods:

- `minimize_scalar(method='brent')`
- `minimize_scalar(method='bounded')`
- `minimize_scalar(method='golden')`

## Ajustement de courbes

## Exemple

- Nous enregistrons un son composé d'une fréquence unique et nous cherchons à connaître cette fréquence.
- Le signal enregistré doit avoir la forme d'une sinusoïde dont on va chercher à estimer la fréquence
- L'enregistrement est bruité :



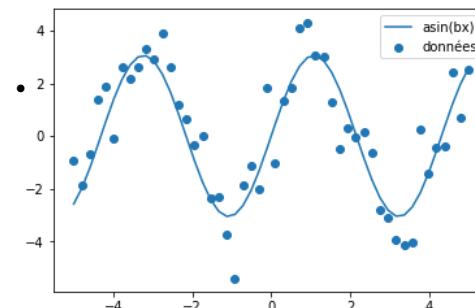
- Comment faire passer "au mieux" une sinusoïde par ces points ?

37

39

## Exemple (suite)

- Nous écrivons le modèle de fonction à ajuster sous la forme :  $y = a \sin(bx)$
- Nous disposons d'un ensemble de points et cherchons à estimer les paramètres  $a$  et  $b$  tels que le modèle passe au mieux par les points  $a, b = \underset{a,b}{\operatorname{argmin}} \sum_i (y_i - a \sin(bx_i))^2$



40

## Exemple (suite)

```

from scipy import optimize

def test_func(x, a, b):
    return a * np.sin(b * x)

params, params_covariance = optimize.curve_fit(test_func,
x_data, y_data, p0=[2, 2])

print(params)

plt.figure(figsize=(6, 4))
plt.scatter(x_data, y_data, label='données')
plt.plot(x_data, test_func(x_data, params[0], params[1]),
label='asin(bx)')

plt.show()

```

## Ajustement de courbes au sens des moindres carrés (suite)

- Les termes  $r_i = y_i - f(x_i)$  sont appelés "résidus".
- La fonction  $f$  est généralement choisie comme une combinaison linéaire de fonctions :  

$$f(x) = a_0 f_0(x) + a_1 f_1(x) + \cdots + a_m f_m(x)$$
comme par exemple des polynômes :  $f_0(x) = 1$ ,  $f_1(x) = x$ ,  $f_2(x) = x^2$
- L'écart de la courbe / points est mesuré par l'écart-type :

$$\sigma = \sqrt{\frac{S}{n-m}}$$

où  $n$  est le nombre de points et  $m$  le nombre de paramètres.  
RQ : si  $m=n$  interpolation, donc pas d'écart

43

## Ajustement de courbes au sens des moindres carrés (suite)

- Supposons disposer de points de mesure contenant du bruit (par exemple en raison de bruit dans les mesures)
- On cherche une fonction paramétrique passant au mieux par les données (mais pas forcément en chaque point).
- Notons :  $f(x) = f(x; a_0, a_1, \dots, a_m)$   
cette fonction (par exemple un polynôme)
- La nature de cette fonction dépend en général de la connaissance que nous avons des données (c'est un modèle des données)
- Que veut dire "passant au mieux par les données" ?
- On peut adopter l'ajustement au sens des moindres carrés par minimisation de :  $S(a_0, a_1, \dots, a_m) = \sum_{i=0}^n [y_i - f(x_i)]^2$
- Valeurs optimales obtenues pour :

$$\frac{\partial S}{\partial a_k} = 0, \quad k = 0, 1, \dots, m$$

42

- Cas de l'ajustement d'une droite  $f(x) = a + bx$

$$S(a, b) = \sum_{i=0}^n [y_i - f(x_i)]^2 = \sum_{i=0}^n (y_i - a - bx_i)^2$$

Dérivation :

$$\frac{\partial S}{\partial a} = \sum_{i=0}^n -2(y_i - a - bx_i) = 2 \left[ a(n+1) + b \sum_{i=0}^n x_i - \sum_{i=0}^n y_i \right] = 0$$

$$\frac{\partial S}{\partial b} = \sum_{i=0}^n -2(y_i - a - bx_i)x_i = 2 \left( a \sum_{i=0}^n x_i + b \sum_{i=0}^n x_i^2 - \sum_{i=0}^n x_i y_i \right) = 0$$

En divisant par  $2(n+1)$  et réarrangeant :

$$a + \bar{x}b = \bar{y} \quad \bar{x}a + \left( \frac{1}{n+1} \sum_{i=0}^n x_i^2 \right) b = \frac{1}{n+1} \sum_{i=0}^n x_i y_i$$

Avec :

$$\bar{x} = \frac{1}{n+1} \sum_{i=0}^n x_i \quad \bar{y} = \frac{1}{n+1} \sum_{i=0}^n y_i$$

D'où :

$$a = \frac{\bar{y} \sum x_i^2 - \bar{x} \sum x_i y_i}{\sum x_i^2 - n \bar{x}^2} \quad b = \frac{\sum x_i y_i - \bar{x} \sum y_i}{\sum x_i^2 - n \bar{x}^2}$$

44

## Ajustement de courbes au sens des moindres carrés (suite)

- Cas des formes linéaires :

$$f(x) = a_0 f_0(x) + a_1 f_1(x) + \dots + a_m f_m(x) = \sum_{j=0}^m a_j f_j(x)$$

où  $f$  est une fonction fixée appelée fonction de base. On a :

$$S = \sum_{i=0}^n \left[ y_i - \sum_{j=0}^m a_j f_j(x_i) \right]^2 \quad \frac{\partial S}{\partial a_j} = 2 \sum_{i=0}^n \left( y_i - \sum_{k=0}^m a_k f_k(x_i) \right) f_j(x_i) = 0$$

ce qui donne après dérivation :

$$\sum_{j=0}^m \left[ \sum_{i=0}^n f_j(x_i) f_k(x_i) \right] a_j = \sum_{i=0}^n f_k(x_i) y_i, \quad k = 0, 1, \dots, m$$

ou encore sous forme matricielle :

$$\mathbf{Aa} = \mathbf{b}$$

$$A_{kj} = \sum_{i=0}^n f_j(x_i) f_k(x_i) \quad b_k = \sum_{i=0}^n f_k(x_i) y_i$$

admettant la solution :  $\mathbf{a} = (\mathbf{A}'\mathbf{A})^{-1}\mathbf{b}$  où  $\mathbf{A}'$  représente la transposée de  $\mathbf{A}$

45

## Ajustement dans le cas général

Dans le cas général, le problème peut s'écrire comme la minimisation de la fonction

$$S(w) = \sum_i (y_i - f(x_i, w))^2$$

La méthode d'optimisation par descente de gradient est une méthode itérative consiste à écrire :

$$w^n = w^{n-1} - \eta \frac{\partial S(w^{n-1})}{\partial w} \quad \text{où } \eta \text{ est un paramètre permettant de contrôler la vitesse de la convergence}$$

## Ajustement de courbes au sens des moindres carrés (suite)

Dans le cas des polynômes, cela donne :

$$f(x) = \sum_{j=0}^m a_j x^j$$

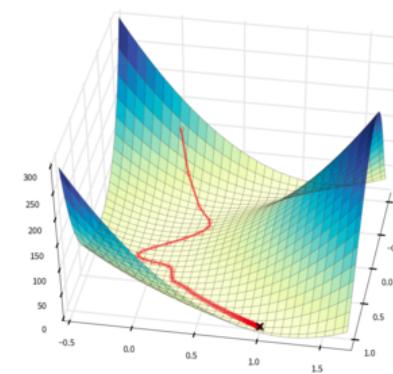
soit :

$$A_{kj} = \sum_{i=0}^n x_i^{j+k} \quad b_k = \sum_{i=0}^n x_i^k y_i$$

et donc :

$$\mathbf{A} = \begin{bmatrix} n & \sum x_i & \sum x_i^2 & \dots & \sum x_i^m \\ \sum x_i & \sum x_i^2 & \sum x_i^3 & \dots & \sum x_i^{m+1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \sum x_i^{m-1} & \sum x_i^m & \sum x_i^{m+1} & \dots & \sum x_i^{2m} \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} \sum y_i \\ \sum x_i y_i \\ \vdots \\ \sum x_i^m y_i \end{bmatrix}$$

## Descente de gradient



47

48

# Retour sur notre exemple

- $y = a \sin(bx)$
- $s = \sum (y_i - a \cos(bx))^2$
- Calcul des dérivées par rapport à a et b :

```
>>> import sympy
>>> x,y,a,b=sympy.symbols('x,y,a,b')
>>> s = (y-a*sympy.sin(b*x))**2
>>> print(sympy.diff(s,a))
>>> print(sympy.diff(s,b))

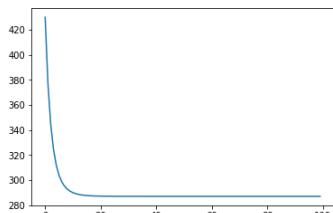
-2*(y-a*sin(b*x))*sin(b*x)
-2*a*x*(y-a*sin(b*x))*cos(b*x)
```

- Optimisation :

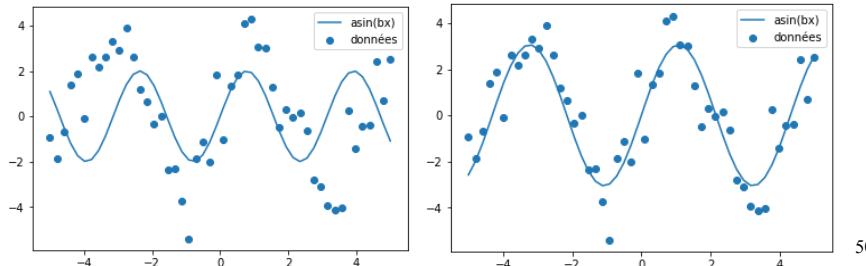
```
for i in range(100):
    a = a + 1*np.sum(y_data-test_func(x_data, a, b)*np.sin(b*x_data))
    b = b + 1*np.sum((y_data-test_func(x_data, a, b))*a*x_data*np.cos(x_data*b))
```

# Retour sur notre exemple

- Evolution de l'erreur



- Fonction avant et après optimisation



## Ajustement avec scipy

## scipy.optimize.curve\_fit

`scipy.optimize.curve_fit(f, xdata, ydata, p0=None, sigma=None, absolute_sigma=False, check_finite=True, bounds=(-inf, inf), method='None', jac=None, **kwargs)` [\[source\]](#)

Use non-linear least squares to fit a function, f, to data.

Assumes `ydata = f(xdata, *params) + eps`

Parameters:

`f : callable`  
The model function, `f(x, ...)`. It must take the independent variable as the first argument and the parameters to fit as separate remaining arguments.

`xdata : An M-length sequence or an (k,M)-shaped array for functions with k predictors`  
The independent variable where the data is measured.

`ydata : M-length sequence`

The dependent data — nominally `f(xdata, ...)`

`p0 : None, scalar, or N-length sequence, optional`

Initial guess for the parameters. If None, then the initial values will all be 1 (if the number of parameters for the function can be determined using introspection, otherwise a `ValueError` is raised).

`sigma : None or M-length sequence or MxM array, optional`

Determines the uncertainty in `ydata`. If we define residuals as `r = ydata - f(xdata, *p0)`, then the interpretation of `sigma` depends on its number of dimensions:

- A 1-d `sigma` should contain values of standard deviations of errors in `ydata`. In this case, the optimized function is `chisq = sum((r / sigma) ** 2)`.
- A 2-d `sigma` should contain the covariance matrix of errors in `ydata`. In this case, the optimized function is `chisq = r.T @ inv(sigma) @ r`.

*New in version 0.19.*

`None` (default) is equivalent of 1-d `sigma` filled with ones.

`absolute_sigma : bool, optional`

If True, `sigma` is used in an absolute sense and the estimated parameter covariance `pcov` reflects these absolute values.

If False, only the relative magnitudes of the `sigma` values matter. The returned parameter covariance matrix `pcov` is based on scaling `sigma` by a constant factor. This constant is set by demanding that the reduced `chisq` for the optimal parameters `popt` when using the scaled `sigma` equals unity. In other words, `pcov` is scaled to match the sample variance of the residuals after the fit. Mathematically,

# scipy.interpolate.interp1d

## scipy.interpolate.interp1d¶

```
class scipy.interpolate.interp1d(x, y, kind='linear', axis=1, copy=True, bounds_error=None, fill_value=np.nan,  
assume_sorted=False)
```

[source]

Interpolate a 1-D function.

*x* and *y* are arrays of values used to approximate some function *f*: *y* = *f*(*x*). This class returns a function whose call method uses interpolation to find the value of new points.

Note that calling `interp1d` with NaNs present in input values results in undefined behaviour.

**Parameters:**

- x* : (N,) array\_like  
A 1-D array of real values.
- y* : (..., N, ...) array\_like  
A N-D array of real values. The length of *y* along the interpolation axis must be equal to the length of *x*.
- kind* : str or int, optional  
Specifies the kind of interpolation as a string ('linear', 'nearest', 'zero', 'slinear', 'quadratic', 'cubic' where 'zero', 'slinear', 'quadratic' and 'cubic' refer to a spline interpolation of zeroth, first, second or third order) or as an integer specifying the order of the spline interpolator to use. Default is 'linear'.
- axis* : int, optional  
Specifies the axis of *y* along which to interpolate. Interpolation defaults to the last axis of *y*.
- copy* : bool, optional  
If True, the class makes internal copies of *x* and *y*, if False, references to *x* and *y* are used. The default is to copy.
- bounds\_error* : bool, optional  
If True, a ValueError is raised any time interpolation is attempted on a value outside of the range of *x* (where extrapolation is necessary). If False, out of bounds values are assigned *fill\_value*. By default, an error is raised unless *fill\_value*="extrapolate".
- fill\_value* : array-like or (array-like, array-like) or "extrapolate", optional
  - If a ndarray (or float), this value will be used to fill in for requested points outside of the data range. If not provided, then the default is NaN. The array-like must broadcast properly to the dimensions of the non-interpolation axes.
  - If a two-element tuple, then the first element is used as a fill value for *x\_new* < *x*[0] and the second element is used for *x* > *x*[-1]. Anything that is not a 2-element tuple is a list or ndarray, regard-

53

# scipy.interpolate.CubicSpline

```
class scipy.interpolate.CubicSpline(x, y, axis=0, bc_type='not-a-knot', extrapolate=None)
```

[source]

Cubic spline data interpolator.

Interpolate data with a piecewise cubic polynomial which is twice continuously differentiable [R85]. The result is represented as a PPoly instance with breakpoints matching the given data.

**Parameters:**

- x* : array\_like, shape (n)  
1-d array containing values of the independent variable. Values must be real, finite and in strictly increasing order.
- y* : array\_like  
Array containing values of the dependent variable. It can have arbitrary number of dimensions, but the length along *axis* (see below) must match the length of *x*. Values must be finite.
- axis* : int, optional  
Axis along which *y* is assumed to be varying. Meaning that for *x*[*i*] the corresponding values are *np.take(y, i, axis=axis)*. Default is 0.
- bc\_type* : string or 2-tuple, optional  
Boundary condition type. Two additional equations, given by the boundary conditions, are required to determine all coefficients of polynomials on each segment [R86]. If *bc\_type* is a string, then the specified condition will be applied at both ends of a spline. Available conditions are:
  - 'not-a-knot' (default): The first and second segment at a curve end are the same polynomial. It is a good default when there is no information on boundary conditions.
  - 'periodic': The interpolated functions is assumed to be periodic of period *x*[-1] - *x*[0]. The first and last value of *y* must be identical: *y*[0] == *y*[-1]. This boundary condition will result in *y'*[0] == *y'*[-1] and *y''*[0] == *y''*[-1].
  - 'clamped': The first derivative at curve ends are zero. Assuming a 1D *y*, *bc\_type*=((1, 0, 0), (1, 0, 0)) is the same condition.
  - 'natural': The second derivative at curve ends are zero. Assuming a 1D *y*, *bc\_type*=((2, 0, 0), (2, 0, 0)) is the same condition.If *bc\_type* is a 2-tuple, the first and the second value will be applied at the curve start and end respectively. The tuple values can be one of the previously mentioned strings (except 'periodic') or a tuple (*order*, *deriv\_values*) allowing to specify arbitrary derivatives at curve ends:
  - *order*: the derivative order, 1 or 2.
  - *deriv\_value*: array\_like containing derivative values, shape must be the same as *y*, excluding *axis* dimension. For example, if *y* is 1D, then *deriv\_value* must be a scalar. If *y* is 3D with the shape (n0, n1, n2) and *axis*=2, then *deriv\_value* must be 2D and have the shape (n0, n1).
- extrapolate* : (bool, 'periodic', None), optional  
If bool, determines whether to extrapolate to-out-of-bounds points based on first and last intervals, or to return NaNs. If 'periodic', periodic extrapolation is used. If None (default), *extrapolate* is set to 'periodic' for *bc\_type='periodic'* and to True otherwise.

54