Analyse Numérique
Interpolation

Simone Deparis

March 11, 2021

Contents

1 Interpolation et approximation de données

1.1 Position du problème

1.1.1 Interpolation de données

Soit $n \geq 0$ un nombre entier. Etant donnés $(n+1)$ noeuds distincts $x_0, x_1, \ldots x_n$ et $(n+1)$ valeurs $y_0, y_1, \ldots y_n$, on cherche un polynôme p de degré n, tel que

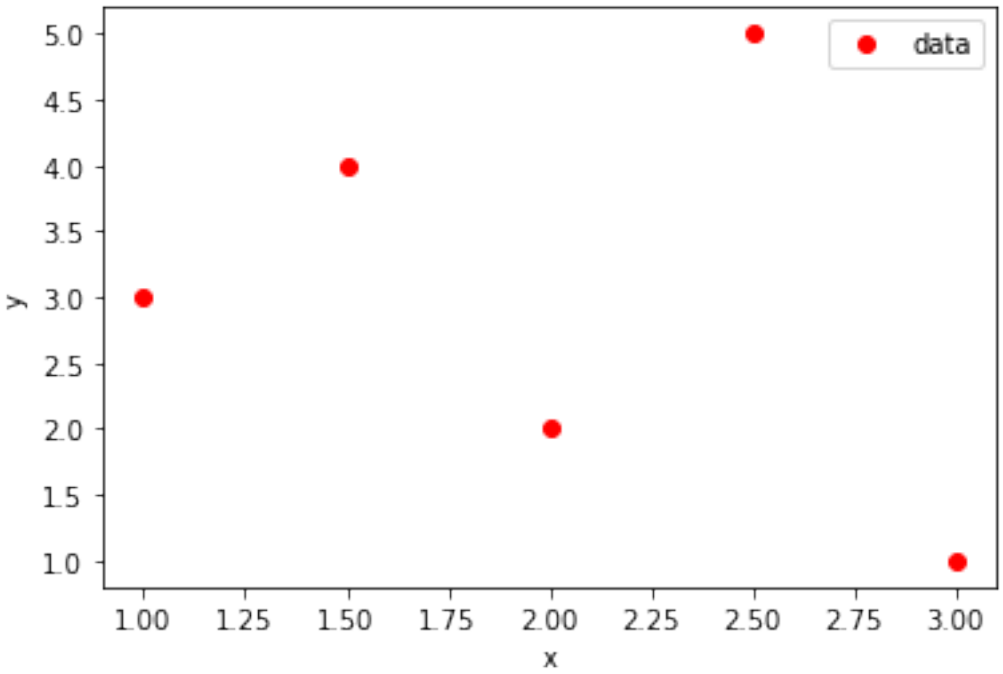
$$p(x_j) = y_j \qquad \text{pour } 0 \leq j \leq n.$$

Exemple On cherche le polynôme Π_n de degré $n = 4$ tel que $\Pi_n(x_j) = y_j, j = 1, ..., 5$ avec les données suivantes

x_k	y_k
1	3
1.5	4
2	2
2.5	5
3	1

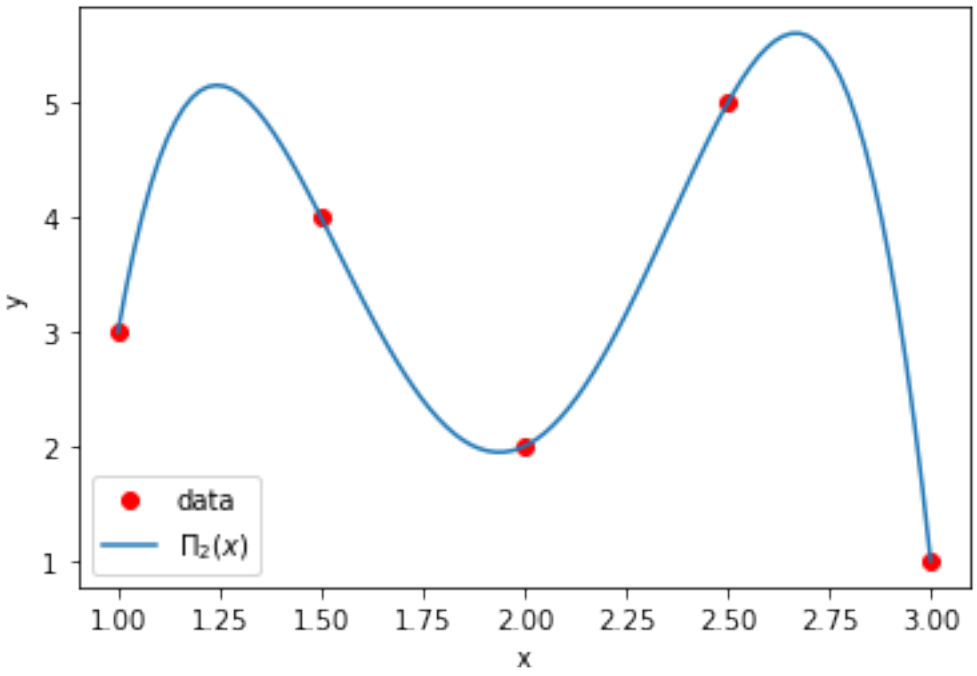
[1]:
```
# importing libraries used in this book
import numpy as np
import matplotlib.pyplot as plt
```

[2]:
```
# Some data given: x=1, 1.5, 2, 2.5, 3 and y = 3,4,2,5,1
x = np.linspace(1, 3, 5) # equivalent to np.array([ 1, 1.5, 2, 2.5, 3 ])
y = np.array([3, 4, 2, 5, 1])

# Plot the points using matplotlib
plt.plot(x, y, 'ro')

plt.xlabel('x'); plt.ylabel('y'); #plt.title('data')
plt.legend(['data'])
plt.show()
```



Si ce polynôme existe, on le note $p = \Pi_n$. On appelle Π_n le polynôme d'interpolation des valeurs y_j aux noeuds x_j, $j = 0, \ldots, n$.

[3]:
```
# Plot the interpolating function

# Defining the polynomial function
def p(x):
    # coefficients of the interpolating polynomial
    a = np.array([-140.0, 343.0, -872./3.,  104.0,  -40./3.])

    # value of the polynomial in all the points t
    return  a[0] + a[1]*x + a[2]*(x**2) + a[3]*(x**3) + a[4]*(x**4)


# points used to plot the graph
z = np.linspace(1, 3, 100)

plt.plot(x, y, 'ro', z, p(z))
plt.xlabel('x'); plt.ylabel('y'); #plt.title('data')
plt.legend(['data','$\Pi_2(x)$'])
plt.show()
```



1.1.2 Interpolation de fonctions

Soit $f \in C^0(I)$ et $x_0, \ldots, x_n \in I$. Si on prend

$$y_j = f(x_j), \quad 0 \leq j \leq n,$$

alors le polynôme d'interpolation $\Pi_n(x)$ est noté $\Pi_n f(x)$ et est appelé l'interpolant de f aux noeuds $x_0, \ldots x_n$.

Exemple Soient

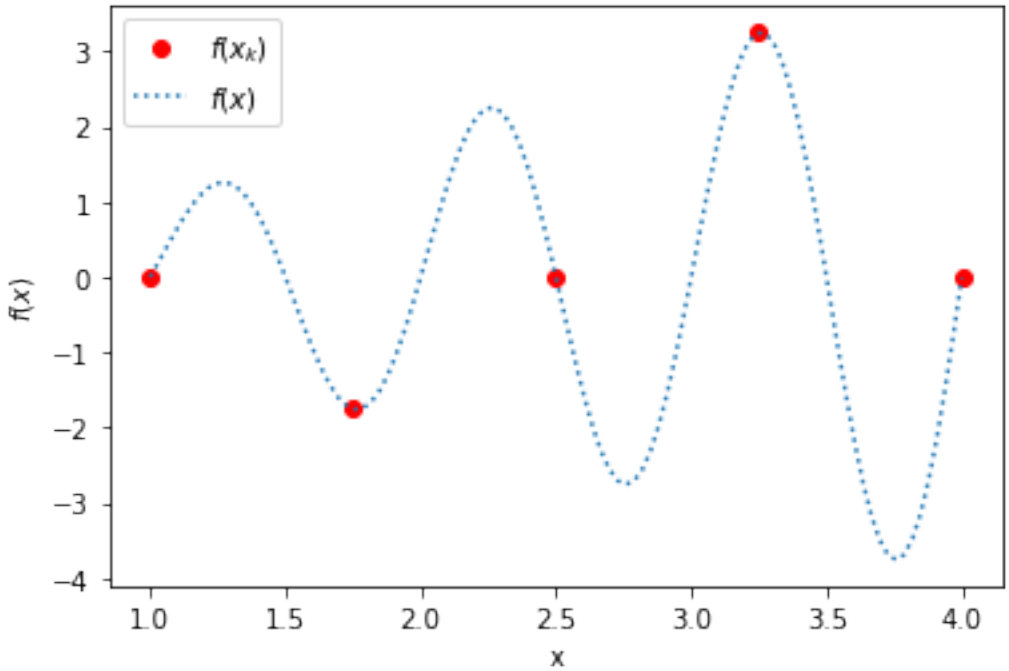
$$x_1 = 1, x_2 = 1.75, x_3 = 2.5, x_4 = 3.25, x_5 = 4$$

les points d'interpolation et

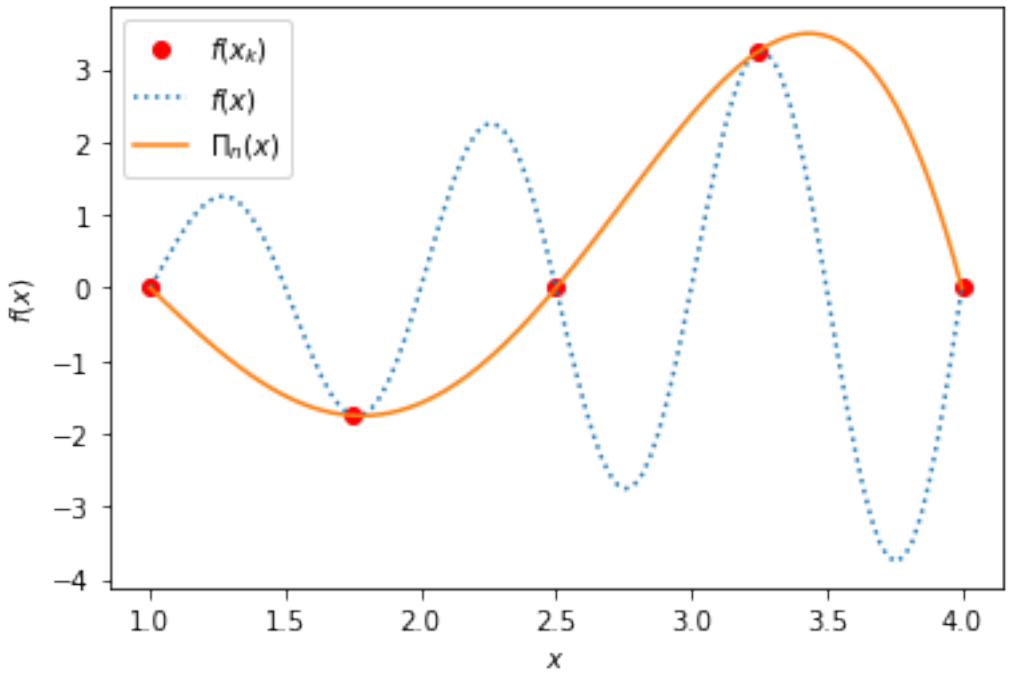
$$f(x) = x \sin(2\pi x).$$

On cherche l'interpolant $\Pi_n f$ de degré $n = 4$

[4]:
```
# defining the fonction that we want to interpolate
def f(x):
    return x*np.sin(x*2.*np.pi)

# The interpolation must occour at points x=1, 1.75, 2.5, 3.25, 4
x = np.linspace(1, 4, 5)

# points used to plot the graph
z = np.linspace(1, 4, 100)

plt.plot(x, f(x), 'ro', z, f(z),':')

# labels, title, legend
plt.xlabel('x'); plt.ylabel('$f(x)$'); #plt.title('data')
plt.legend(['$f(x_k)$','$f(x)$'])
plt.show()
```



[5]:
```
# Plot the interpolating function

# Defining the polynomial function
def p(x):
    # coefficients of the interpolating polynomial
    a = np.array([0,       7.9012,      -13.037,       5.9259,     -0.79012])

    # value of the polynomial in all the points x
    return  a[0] + a[1]*x + a[2]*(x**2) + a[3]*(x**3) + a[4]*(x**4)


# points where to evaluate the polynomial
z = np.linspace(1, 4, 100)

plt.plot(x, f(x), 'ro', z, f(z),':', z, p(z))
plt.xlabel('$x$'); plt.ylabel('$f(x)$'); #plt.title('data')
plt.legend(['$f(x_k)$','$f(x)$','$\Pi_n(x)$'])

plt.show()
```



1.1.3 Matrice de Vandermonde

Il est possible d'écrire un système d'équations et de trouver les coefficients de manière directe. Ce n'est pas toujours la meilleure solution.

Nous cherchons les coefficients du polynôme $p(x) = a_0 + a_1 x + ... + a_n x^n$ qui satisfont les $(n+1)$ équations $p(x_k) = y_k, k = 0, ..., n$, c'est-à-dire

$$a_0 + a_1 x_k + ... + a_n x_k^n = y_k, k = 0, ..., n$$

Ce système s'écrit sous forme matricielle

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ \vdots & & & & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^n \end{pmatrix} \begin{pmatrix} a_0 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} y_0 \\ \vdots \\ y_n \end{pmatrix}$$

Pour construire cette matrice, vous pouvez utiliser la fonction

```
# Defining the mxn Vandermonde matrix
def VandermondeMatrix(x):
    # Input
    # x : +1 array with interpolation nodes
    # Output
    # Matrix of Vandermonde of size n x n
```

que vous pouvez importer avec la commande

```
from InterpolationLib import VandermondeMatrix
```

Exemple On cherche les coefficients du polynôme d'interpolation de degré $n = 4$ des valeurs suivantes

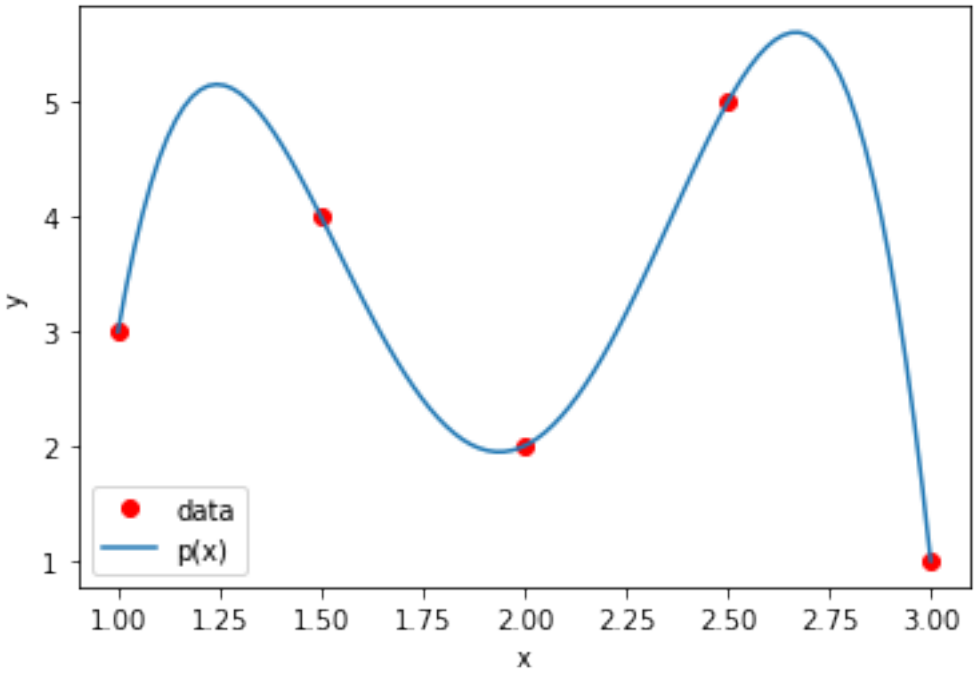
x_k	y_k
1	3
1.5	4
2	2
2.5	5
3	1

[6]:
```
from InterpolationLib import VandermondeMatrix

# Some data given: x=1, 1.5, 2, 2.5, 3 and y = 3,4,2,5,1
x = np.linspace(1, 3, 5)
y = np.array([3, 4, 2, 5, 1])
n = x.size - 1

A = VandermondeMatrix(x)
# print(A)

# compute coefficients
a = np.linalg.solve(A, y) # Resouds Ax = b avec b=y et rends x
```

```
# print the coefficients on screen
print('The coefficients a_0, ..., a_n are')
print(a)
```

```
The coefficients a_0, ..., a_n are
[-140.          343.         -290.66666667  104.          -13.33333333]
```

[7]:
```
# Now we can define the polynomial
p = lambda x : a[0] + a[1]*x + a[2]*(x**2) + a[3]*(x**3) + a[4]*(x**4)

# points used to plot the graph
z = np.linspace(1, 3, 100)

plt.plot(x, y, 'ro', z, p(z))
plt.xlabel('x'); plt.ylabel('y'); #plt.title('data')
plt.legend(['data','p(x)'])
plt.show()
```



1.1.4 Alternatives : polyfit et polyval

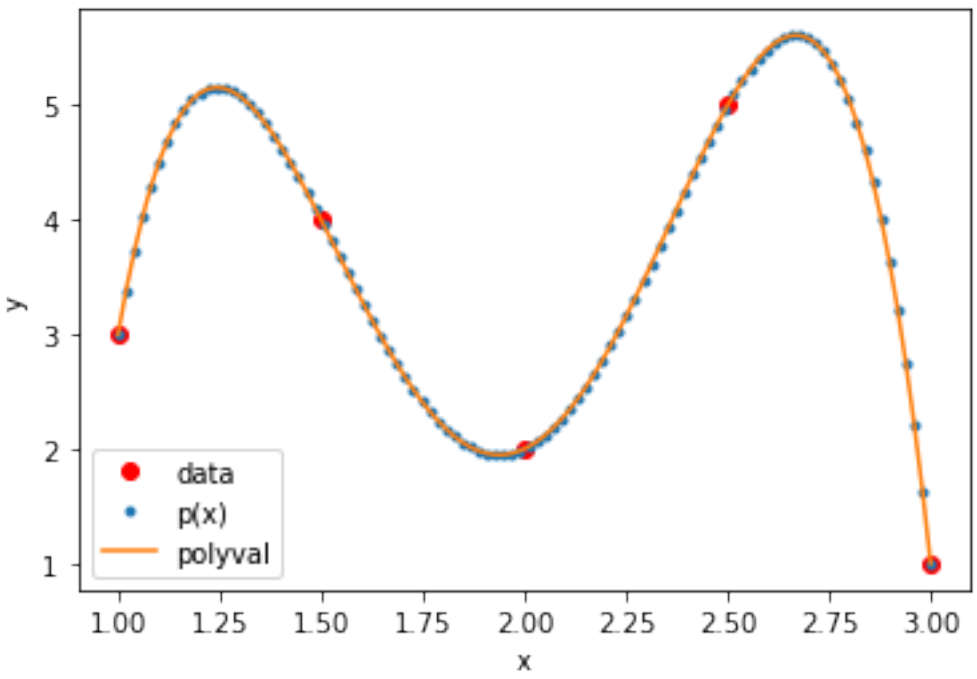
Les fonctions `polyfit` et `polyval` de `numpy` font essentiellement la même chose que les paragraphes ci-dessous. Plus tard nous verrons des méthodes plus performantes.

`a = numpy.polyfit(x, y, n, ... ) :`

- input : x, y les données à interpoler, n le degré du polynôme recherché

- output : les coefficients du polynôme, *dans l'ordre inverse* de ce que nous avons vu !

[8]:
```
# Some data given: x=1, 1.5, 2, 2.5, 3 and y = 3,4,2,5,1
x = np.linspace(1, 3, 5)
y = np.array([3, 4, 2, 5, 1])
n = x.size - 1

a = np.polyfit(x,y,n)

# Now we can define the polynomial, with coeffs in the reverse order !
p = lambda x : a[4] + a[3]*x + a[2]*(x**2) + a[1]*(x**3) + a[0]*(x**4)

# We can also use polyval instead !
# np.polyval(a,x)

# points used to plot the graph
z = np.linspace(1, 3, 100)

plt.plot(x, y, 'ro', z, p(z), '.', z, np.polyval(a,z))
plt.xlabel('x'); plt.ylabel('y'); #plt.title('data')
plt.legend(['data','p(x)','polyval'])
plt.show()
```



[ ]:

1.2 Interpolation de Lagrange

1.2.1 Base de Lagrange

On considère les polynômes φ_k, $k = 0, \ldots, n$ de degré n tels que

$$\varphi_k(x_j) = \delta_{jk}, \qquad k, j = 0, \ldots, n,$$

où $\delta_{jk} = 1$ si $j = k$ et $\delta_{jk} = 0$ si $j \neq k$. Explicitement, on a

$$\varphi_k(x) = \prod_{j=0, j\neq k}^{n} \frac{(x - x_j)}{(x_k - x_j)}.$$

[9]:
```
# importing libraries used in this book
import numpy as np
import matplotlib.pyplot as plt
```

Exercice (théorique) Vérifiez que

1. $B = \{\varphi_k, k = 0, \ldots, n\}$ est une base de $P_n(\mathbb{R})$
2. Chaque polynôme φ_k est de degré n
3. $\varphi_k(x_j) = \delta_{jk}, \qquad k, j = 0, \ldots, n$

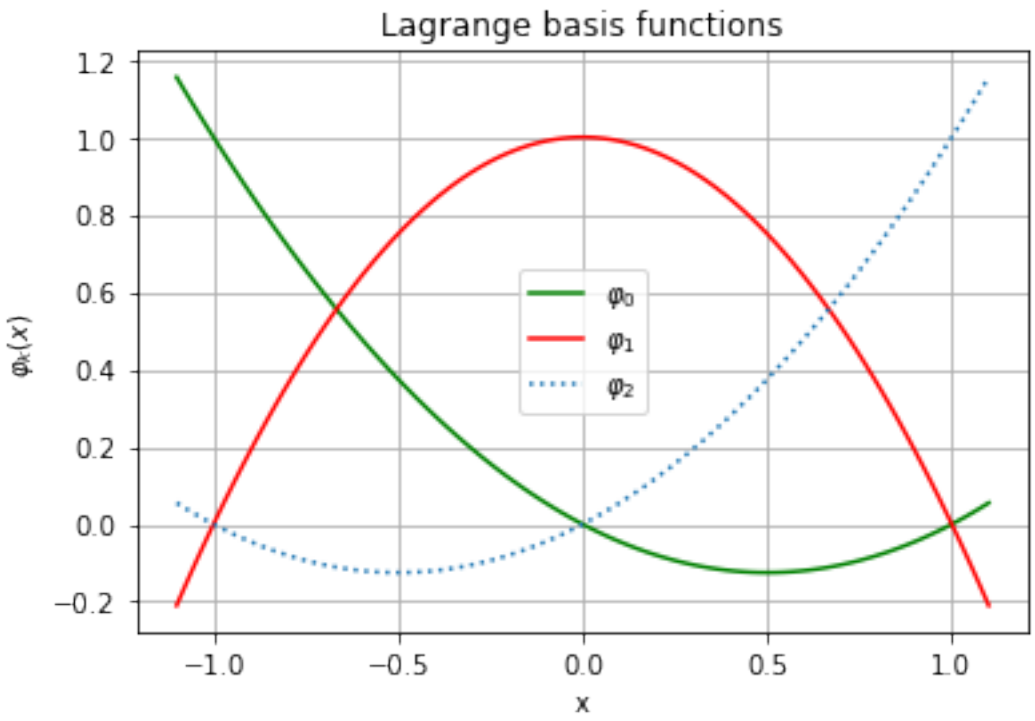
[10]:
```
# Defining the Lagrange basis functions
def phi(x,k,z):
    # the input variables are:
    # x : the interpolatory points
    # k : which basis function
    # z : where to evaluate the function


    # careful, there are n+1 interpolation points!
    n = x.size - 1

    # init result to one, of same type and size as z
    result = np.zeros_like(z) + 1

    # first few checks on k:
    if (type(k) != int) or (x.size < 1) or (k > n) or (k < 0):
        raise ValueError('Lagrange basis needs a positive integer k, smaller␣
 ↪than the size of x')

    # loop on n to compute the product
    for j in range(0,n+1) :
        if (j == k) :
            continue
        if (x[k] == x[j]) :
```

```
            raise ValueError('Lagrange basis: all the interpolation points need␣
↪to be distinct')

        result = result * (z - x[j]) / (x[k] - x[j])

    return result
```

Exemple Pour $n = 2$, $x_0 = -1$, $x_1 = 0$, $x_2 = 1$, les polynômes de la base de Lagrange sont

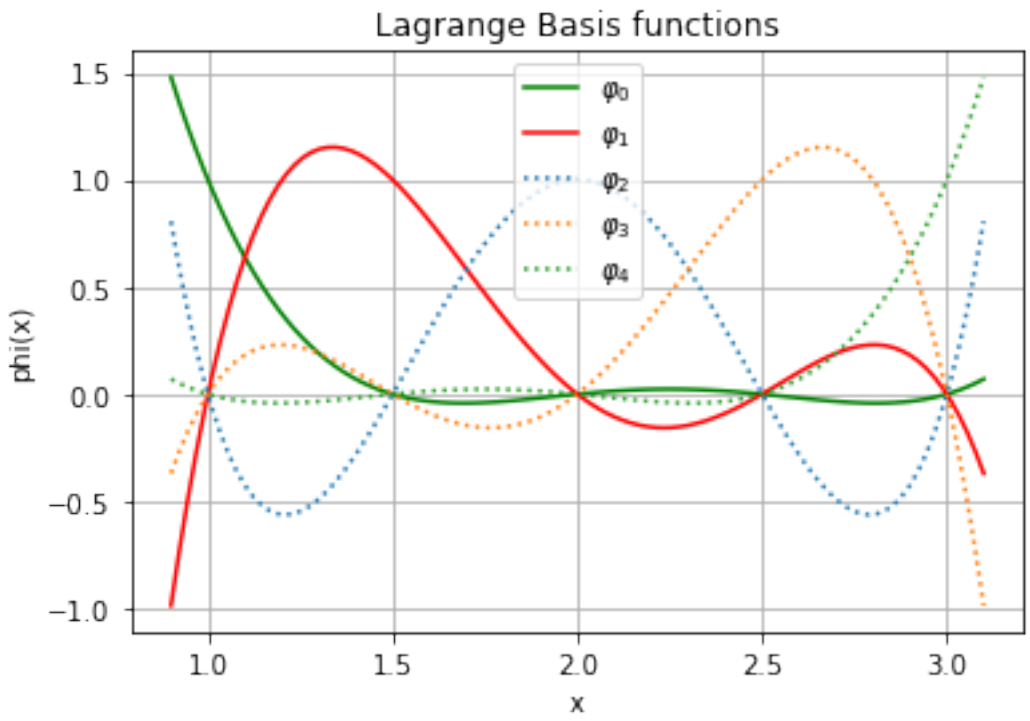
$$\begin{aligned}
\varphi_0(x) &= \frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)} = \frac{1}{2}x(x-1),\\
\varphi_1(x) &= \frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)} = -(x+1)(x-1),\\
\varphi_2(x) &= \frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)} = \frac{1}{2}x(x+1).
\end{aligned}$$

[11]:
```
# plot the Lagrange Basis functions
x = np.linspace(-1., 1, 3)
z = np.linspace(-1.1, 1.1, 100)

plt.plot(z, phi(x,0,z), 'g', z, phi(x,1,z), 'r', z, phi(x,2,z),':')

plt.xlabel('x'); plt.ylabel('$\\varphi_{k}(x)$'); plt.title('Lagrange basis␣
↪functions')
plt.legend(['$\\varphi_{0}$','$\\varphi_{1}$','$\\varphi_{2}$'])
plt.grid(True)
plt.show()
```



Exercice Visualisez la base de Lagrange associée aux points $x_k = 1, 1.5, 2, 2.5, 3$. Evaluez sur le graphique les valeurs de $\varphi_k(x_j)$

[12]:
```
# plot the Lagrange Basis functions
x = np.linspace(1., 3, 5)
z = np.linspace(0.9, 3.1, 100)

plt.plot(z, phi(x,0,z), 'g', z, phi(x,1,z), 'r', z, phi(x,2,z),':', z,␣
 ↪phi(x,3,z),':', z, phi(x,4,z),':')

plt.xlabel('x'); plt.ylabel('phi(x)'); plt.title('Lagrange Basis functions')
plt.legend(['$\\varphi_0$','$\\varphi_1$','$\\varphi_2$',
            '$\\varphi_3$','$\\varphi_4$'])
plt.grid(True)
plt.show()
```



1.2.2 Polynôme d'interpolation

Le polynôme d'interpolation Π_n des valeurs y_j aux noeuds x_j, $j = 0, \ldots, n$, s'écrit

$$\Pi_n(x) = \sum_{k=0}^{n} y_k \varphi_k(x), \tag{1}$$

car il vérifie $\Pi_n(x_j) = \sum_{k=0}^{n} y_k \varphi_k(x_j) = y_j$.

1.3 Interpolation d'une fonction continue

Soit $f : [a, b] \to R$ continue et $x_0, \ldots, x_n \in [a, b]$ des noeuds distincts. Le polynôme d'interpolation $\Pi_n(x)$ est noté $\Pi_n f(x)$ et est appelé l'interpolant de f aux noeuds $x_0, \ldots, x_n$.

Si on prend

$$y_k = f(x_k), k = 0, ..., n,$$

alors on aura

$$\Pi_n f(x) = \sum_{k=0}^{n} f(x_k) \varphi_k(x).$$

Exercice Ecrivez une fonction Python qui a la définition suivante, en utilisant la fonction `phi` définie plus haut. Ecrivez aussi un petit test sur la base de l'exercice précédent.

```
# Lagrange Interpolation of data (x,y), evaluated at ordinate(s) z
def LagrangePi(x,y,z):
    # the input variables are:
    # x : the interpolatory points
    # y : the corresponding data at the points x
    # z : where to evaluate the function
```

Utilisez le fait que $\{\varphi_k, k = 0, ..., n\}$ est une base des polynômes de degré $\leq n$ et que le vecteur y représente les coordonnées du polynôme d'interpolation recherché par rapport à cette base, c'est-à-dire

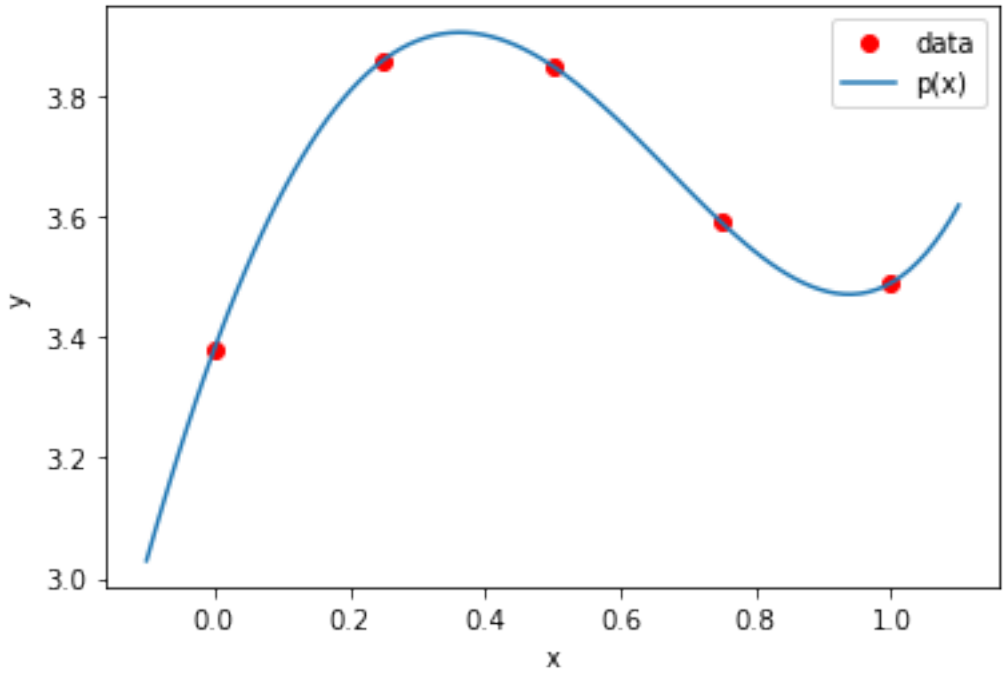
$$\Pi_n(z) = y_0\varphi_0 + ... + y_n\varphi_n$$

[13]:
```
# importing libraries used in this book
import numpy as np
import matplotlib.pyplot as plt

from InterpolationLib import LagrangeBasis as phi
```

[14]:
```
# Lagrange Interpolation of data (x,y), evaluated at ordinate(s) z
def LagrangePi(x,y,z):
    # the input variables are:
    # x : the interpolatory points
    # y : the corresponding data at the points x
    # z : where to evaluate the function

    # {phi(x,k,.), k=0,...,n} is a basis of the polynomials of degree n
    # y represents the coordinates of the interpolating polynomial with respect␣
 ↪to this basis.
    # Therefore LagrangePi(x,y,.) = y[0] phi(x,0,.) + ... + y[n] phi(x,n,.)

    # careful, there are n+1 basis functions!
    n = x.size - 1

    # init result to zero, of same type and size as z
    result = np.zeros_like(z)

     # loop on n to compute the product
    for k in range(0,n+1) :
        result = result + y[k] * phi(x,k,z)

    return result
```

[15]:
```
# vecteur des points d'interpolation
x = np.linspace(0, 1, 5)

# vecteur des valeurs
```

```
y = np.array([3.38, 3.86, 3.85, 3.59, 3.49])

z = np.linspace(-0.1, 1.1, 100)
plt.plot(x, y, 'ro', z, LagrangePi(x,y,z))

plt.xlabel('x'); plt.ylabel('y');
plt.legend(['data','p(x)'])
plt.show()
```



1.3.1 Erreur d'interpolation

Soient $x_0, x_1, \ldots, x_n$, $(n+1)$ nœuds *équirépartis* dans $I = [a, b]$ et soit $f \in C^{n+1}(I)$. Alors

$$\max_{x \in I} |f(x) - \Pi_n f(x)| \leq \frac{1}{2(n+1)} \left(\frac{b-a}{n}\right)^{n+1} \max_{x \in I} |f^{(n+1)}(x)|. \tag{2}$$

On remarque que l'erreur d'interpolation dépend de la dérivée $(n+1)$-ième de f.

Exercice On considère les points d'interpolation

$$x_0 = 1, x_1 = 1.75, x_2 = 2.5, x_3 = 3.25, x_4 = 4$$

et la fonction

$$f(x) = x \sin(2\pi x)$$

.

1. Calculez la base de Lagrange associée à ces points. D'abord sur papier, ensuite utilisez Python pour en dessiner le graphique.
2. Calculez le polynôme d'interpolation Π_n à l'aide de la base de Lagrange. D'abord sur papier, ensuite avec Python.
3. Quelle est l'erreur théorique d'interpolation ?

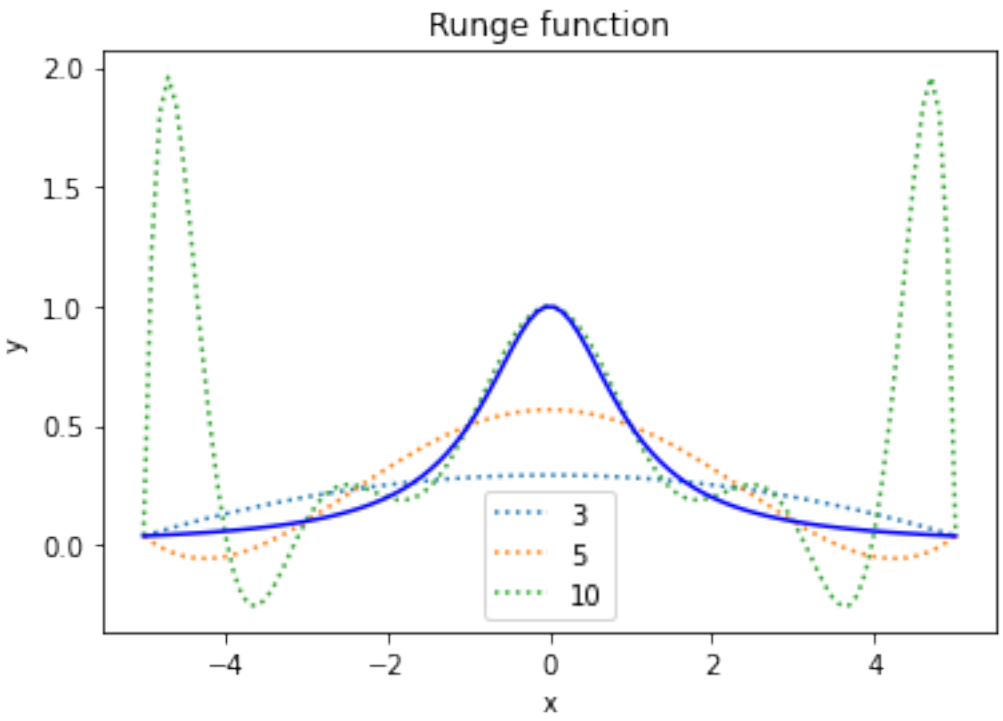
Comportement pour n grand: eg, la fonction de Runge. Le fait que

$$\lim_{n\to\infty} \frac{1}{2(n+1)} \left(\frac{b-a}{n}\right)^{n+1} = 0$$

n'implique pas forcément que $\max_{x\in I} |E_n f(x)|$ tende vers zéro quand $n \to \infty$.

Soit $f(x) = \dfrac{1}{1+x^2}$, $x \in [-5,5]$. Si on l'interpole dans des noeuds équirépartis, l'interpolant présente des oscillations au voisinage des extrémités de l'intervalle.

[16]:
```
# Runge fonction
f = lambda x : 1./(1+x**2)

# Values of N to use
Nrange = [3,5,10]
# plotting points
z = np.linspace(-5, 5, 100)

for n in Nrange :

    x = np.linspace(-5,5,n+1)
    y = f(x);

    plt.plot(z, LagrangePi(x,y,z), ':')

plt.plot(z,f(z), 'b')
plt.xlabel('x'); plt.ylabel('y'); plt.title('Runge function')
plt.legend(Nrange)
plt.show()
```



sympy est une librairie pour le calcul symbolique en Python. Nous allons l'utiliser pour étudier le comportement de l'erreur d'interpolation de la fonction de Runge.

[17]:
```
# Using symbolic python to compute derivatives
import sympy as sp

# define x as symbol
x = sp.symbols('x')

# define Runge function
f = 1/(1+x**2)

# pretty print the 5th derivative of f
f5 = sp.diff(f, x,5)
# display f5
sp.init_printing(use_unicode=True)
display(f5)


# evalf can be used to compute the value of a function at a given point
print('5th derivative evaluated at 3 :')
print( f5.evalf(subs={x: 3})  )
```

$$\frac{240x\left(-\frac{16x^4}{(x^2+1)^2}+\frac{16x^2}{x^2+1}-3\right)}{(x^2+1)^4}$$

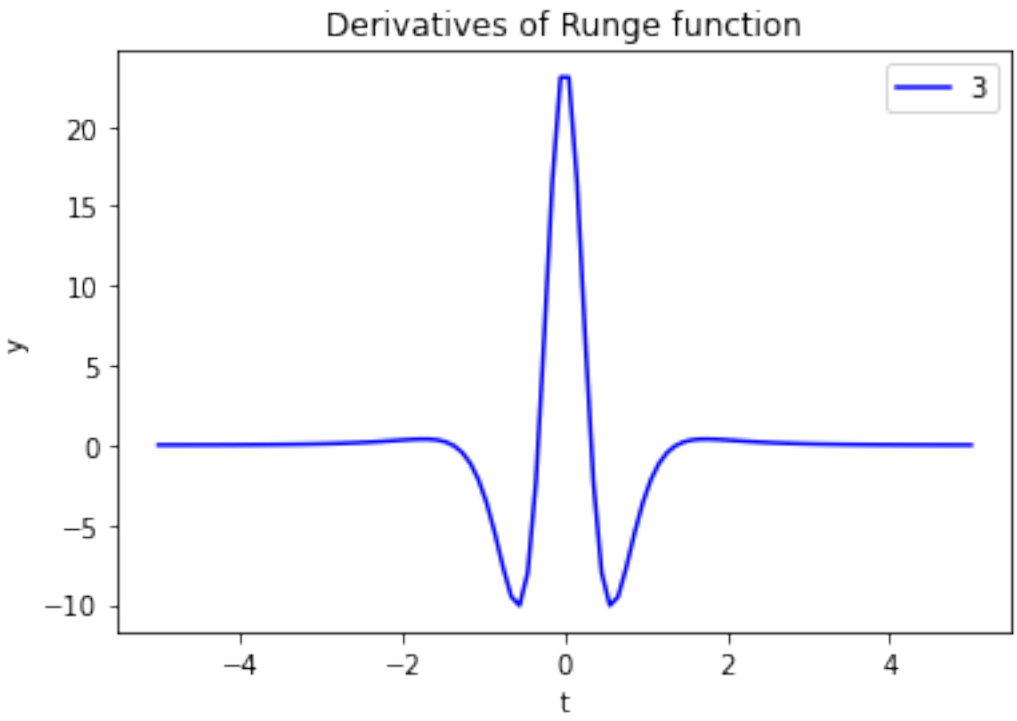
```
5th derivative evaluated at 3 :
-0.112320000000000
```

Pour définir une fonction qui accepte un array de valeur, il faut utiliser les lignes suivantes.

Ensuite on peut aussi dessiner le graphe ...

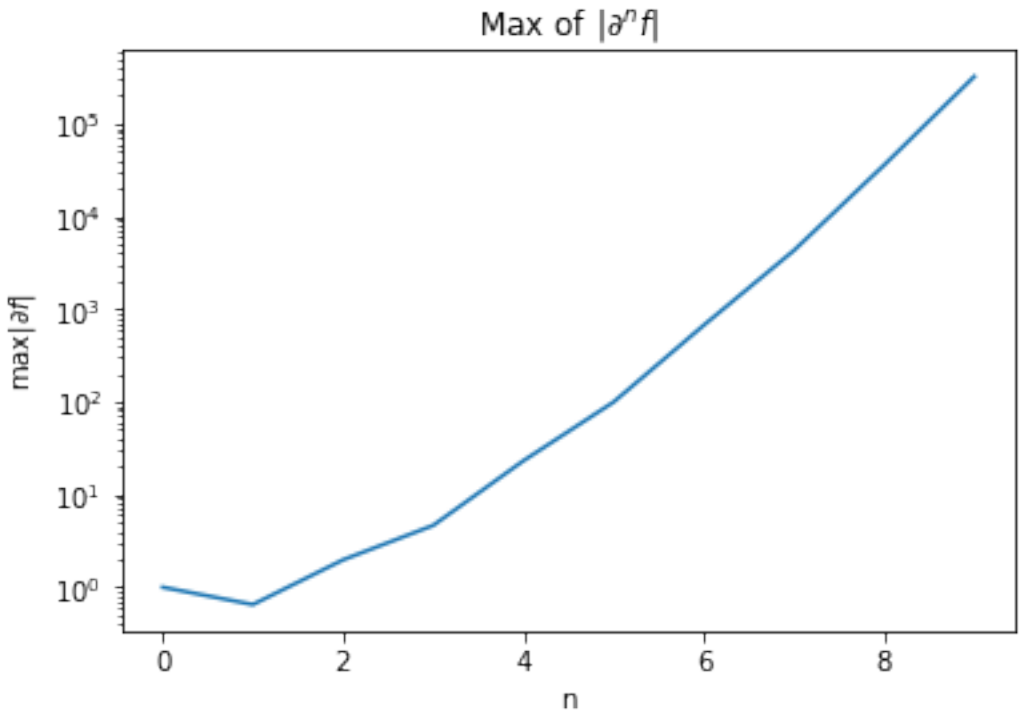
[18]:
```
# to evaluate a function at many given points, we need the following trick
diff_f_func = lambda t: float(sp.diff(f,x,k).evalf(subs={x: t}))
diff_f = np.vectorize(diff_f_func)

# the derivative can be set with k (not very elegant...)
k = 4
print(diff_f(4.5))

# plotting points
z = np.linspace(-5, 5, 100)
# plot the derivative between -5 and 5

plt.plot(z,diff_f(z), 'b')
plt.xlabel('t'); plt.ylabel('y'); plt.title('Derivatives of Runge function')
plt.legend(Nrange)
plt.show()
```

```
0.0102402235718104
```



... ou évaluer le maximum pour plusieurs n et voir le comportement de max $|f^{(n)}|$ en fonction de n.

[19]:
```
# Plot max(abs(fn)) in the range -5,5
z = np.linspace(-5, 5, 100)

Nmax = 10
maxValFn = np.zeros(Nmax)
for k in range(Nmax):
    maxValFn[k] = np.max(np.abs(diff_f(z)))

plt.plot(range(10), maxValFn)

plt.yscale('log')
plt.xlabel('n'); plt.ylabel('$\max|\partial f|$');
plt.title('Max of $|\partial^n f|$');
plt.show()
```



1.3.2 Interpolation de Chebyshev

Pour chaque entier positif $n \geq 1$, pour $i = 0, \ldots n$, on note

$$\hat{x}_i = -\cos(\pi i/n) \in [-1, 1]$$

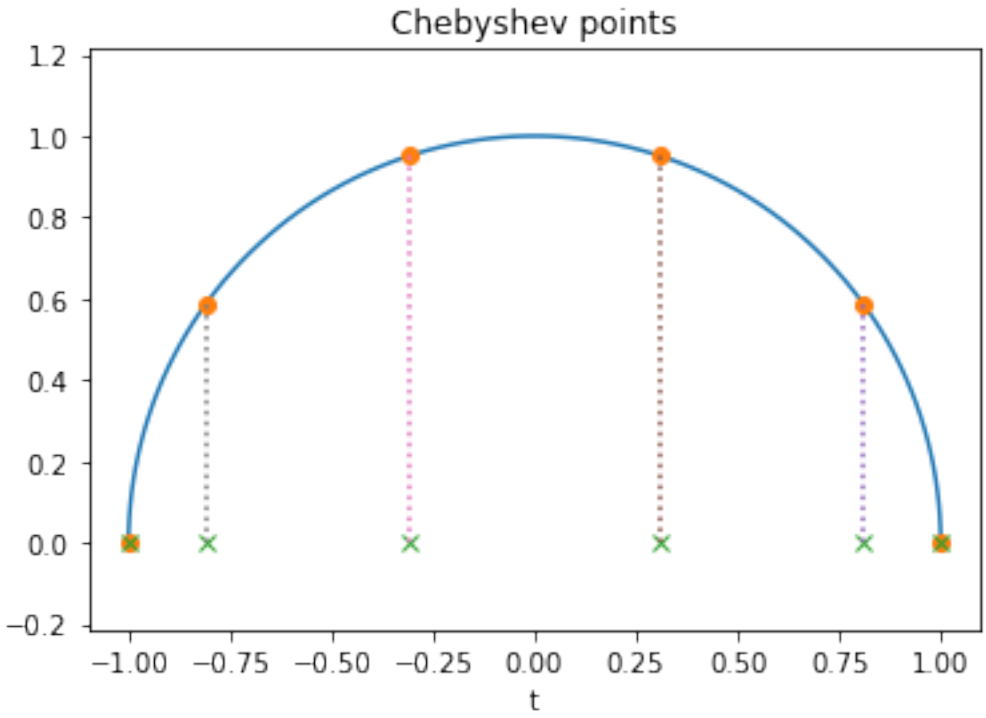
les points de Chebyshev et on définit

$$x_i = \frac{a+b}{2} + \frac{b-a}{2}\hat{x}_i \in [a, b],$$

pour un intervalle arbitraire $[a, b]$. Pour une fonction continue $f \in C^1([a, b])$, le polynôme d'interpolation $\Pi_n f$ de degré n aux noeuds $\{x_i, i = 0, \ldots, n\}$ converge uniformément vers f quand $n \to \infty$.

[20]:
```
# Chebichev points on the interval [-1,1]

z = np.linspace(0,1, 100)
plt.plot(np.cos(np.pi*z), np.sin(np.pi*z))

n =5
z = np.linspace(0,1, n+1)
plt.plot(np.cos(np.pi*z), np.sin(np.pi*z), 'o')
plt.plot(np.cos(np.pi*z), 0*z, 'x')
```

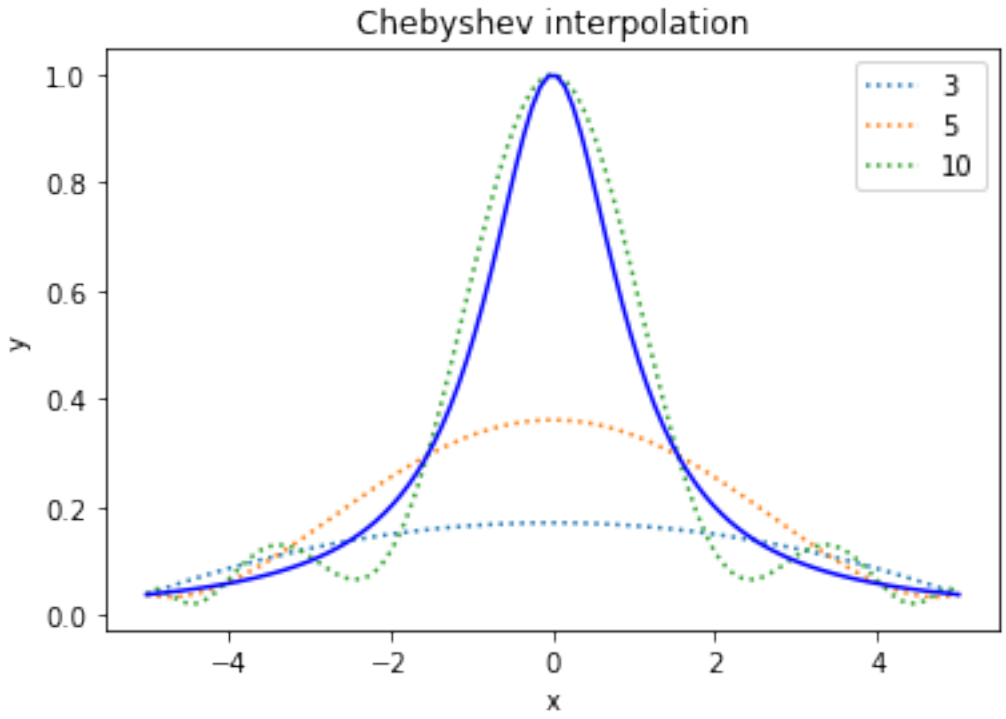
```
for k in range(0,n+1) :
    plt.plot([np.cos(np.pi*z[k]),np.cos(np.pi*z[k])],[0,np.sin(np.pi*z[k])],':')

plt.axis('equal')

plt.xlabel('t');
plt.title('Chebyshev points')
plt.show()
```



Exemple On reprend le même exemple mais on interpole la fonction de Runge dans les points de Chebyshev. La figure montre les polynômes de Chebyshev de degrés $n = 5$ et $n = 10$. On remarque que les oscillations diminuent lorsqu'on augmente le degré du polynôme.

[21]:
```
# Runge fonction
f = lambda x : 1./(1+x**2)

# Values of N to use
Nrange = [3,5,10]
# plotting points
[a,b] = [-5,5]
z = np.linspace(a,b, 100)

for n in Nrange :
```

```
    # Chebyshev points on [-1,1]
    hx = -np.cos(np.pi*np.linspace(0,n,n+1)/n)
    # mapped to [a,b]
    x =(a+b)/2 + (b-a)/2*hx

    y = f(x);

    plt.plot(z, LagrangePi(x,y,z), ':')

plt.plot(z,f(z), 'b')
plt.xlabel('x'); plt.ylabel('y'); plt.title('Chebyshev interpolation')
plt.legend(Nrange)
plt.show()
```



[ ]:

1.4 Interpolation par intervalles

1.4.1 Interpolation linéaire par morceaux

Soit $f : [a, b] \to R$ continue et $a = x_0 < \ldots < x_n = b$.

On choisit une partition de $[a, b]$ en N sous-intervalles de la forme $[x_i, x_{i+1}], i = 0, ..., N - 1]$. Sur

chaque sous-intervalle, on fait une interpolation de degré 1 avec les 2 noeuds x_i, x_{i+1}. Sur chaque sous-intervalle $I_i = [x_i, x_{i+1}]$, on interpole $f_{|I_i}$ par un polynôme de degré 1. Le polynôme par morceaux (polynôme composite) qu'on obtient est noté $\Pi_1^H f(x)$ et on a:

$$\Pi_1^H f(x) = f(x_i) + \frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i}(x - x_i) \quad \text{pour } x \in [x_i, x_{i+1}]$$

Dans le cas de données y, cela s'écrit

$$\Pi_1^H(x) = y_i + \frac{y_{i+1} - y_i}{x_{i+1} - x_i}(x - x_i) \quad \text{pour } x \in [x_i, x_{i+1}]$$

Le choix le plus simple est le suivant :

- on pose $H = \frac{b-a}{N}$
- ensuite $x_i = a + iH$ pour $i = 0, ..., N$

[22]:
```
# importing libraries used in this book
import numpy as np
import matplotlib.pyplot as plt

from InterpolationLib import PiecewiseLinearInterpolation as  PiH1
```

La fonction `PiH1` implémente l'interpolation linéaire par morceaux pour des points équidistribués

```
def PiecewiseLinearInterpolation(a,b,N,f,z):
    # the input variables are:
    # a,b : x[0] = a, x[n] = b
    # f : the corresponding data at the points x
    # z : where to evaluate the function
```

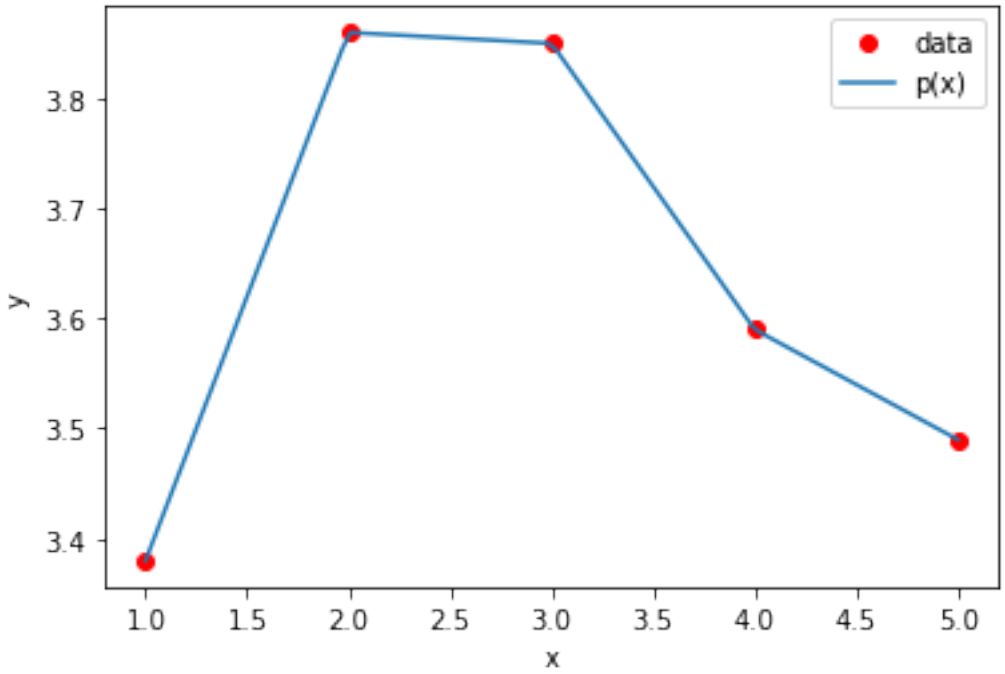
1.4.2 Exercice

Soient $x_k, k = 0, ..., 4$ des points équidistribués sur l'intervalle $[1, 5]$ et $y_k = (3.38, 3.86, 3.85, 3.59, 3.49)$ les valeurs d'une fonction en ces points.

- Dessinez le graphe de l'interpolateur par morceaux de cette fonction
- Calculez numériquement (sur papier) la valeur de $\Pi_1^H(4.5)$ et vérifiez le résultat sur le graphique

[23]:
```
# intervalle d'interpolation
a = 1; b = 5

# vecteur des valeurs aux points equidistribué
y = np.array([3.38, 3.86, 3.85, 3.59, 3.49])
N = y.size-1
x = np.linspace(a,b,N+1)

z = np.linspace(a, b, 100)
plt.plot(x, y, 'ro', z, PiH1(a,b,N,y,z) )
```

```
plt.xlabel('x'); plt.ylabel('y'); #plt.title('data')
plt.legend(['data','p(x)'])
plt.show()
```

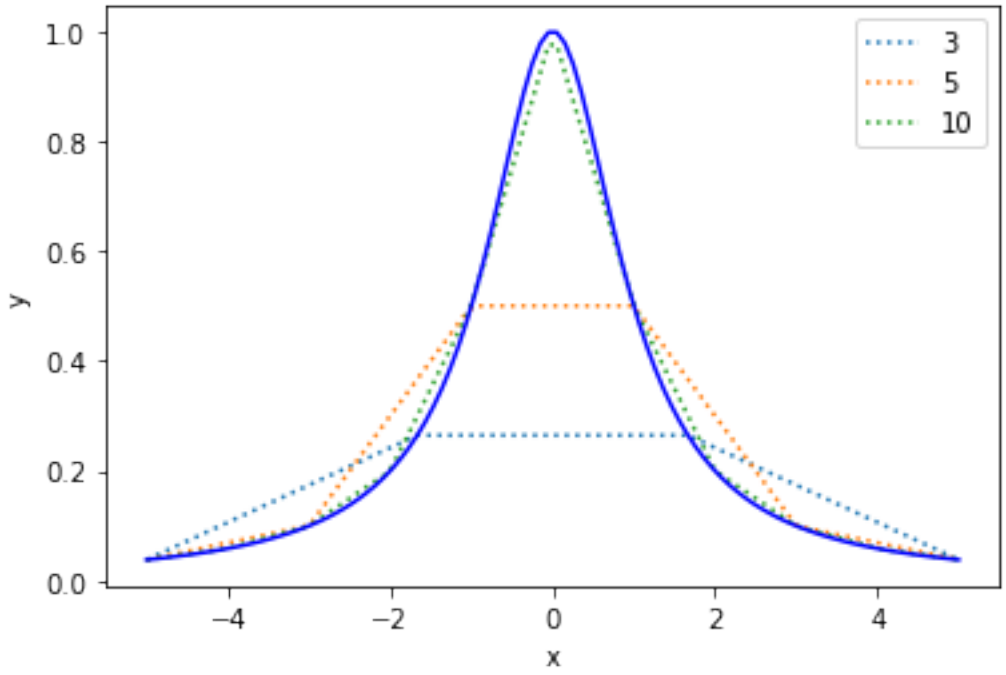


1.4.3 Exercice

Dessinez le graphe de la fonction de Runge et l'interpolateur linéaire par morceaux sur l'intervalle $[-5,5]$ pour $N = 3,5,10$

[24]:
```
# interval and function
a = -5; b = 5
f = lambda x : 1/(1+x**2)

# Values of N to use
Nrange = [3,5,10]
# plotting points
z = np.linspace(-5, 5, 100)

for N in Nrange :
    plt.plot(z, PiH1(a,b,N,f,z), ':')

plt.plot(z, f(z), 'b')
plt.xlabel('x'); plt.ylabel('y'); #plt.title('data')
plt.legend(Nrange)
```

```
plt.show()
```



1.4.4 Erreur d'interpolation linéaire par morceaux

Théorème Soient $f \in C^2([a,b])$, $H = \frac{b-a}{N}$, $x_i = a + iH$ pour $i = 0, ..., N$.

Soit $E_1^H f(x) = f(x) - \Pi_1^H f(x)$, alors

$$\max_{x \in I} \mid E_1^H f(x) \mid \leq \frac{H^2}{4} \max_{x \in I} |f''(x)|.$$

Preuve D'après la formule, sur chaque intervalle $I_i = [x_i, x_{i+1}]$, on a

$$\max_{x \in [x_i, x_{i+1}]} \mid E_1^H f(x) \mid \leq \frac{H^2}{2(1+1)} \max_{x \in I_i} \mid f''(x) \mid .$$

Remarque On peut aussi montrer que, si l'on utilise un polynôme de degré n (≥ 1) et si l'on dénote $E_n^H f(x) = f(x) - \Pi_n^H f(x)$, dans chaque sous-intervalle I_i, on trouve

$$\max_{x \in I} \mid E_n^H f(x) \mid \leq \frac{H^{n+1}}{2(n+1)} \max_{x \in I} |f^{(n+1)}(x)| .$$

1.4.5 Interpolation quadratique par morceaux

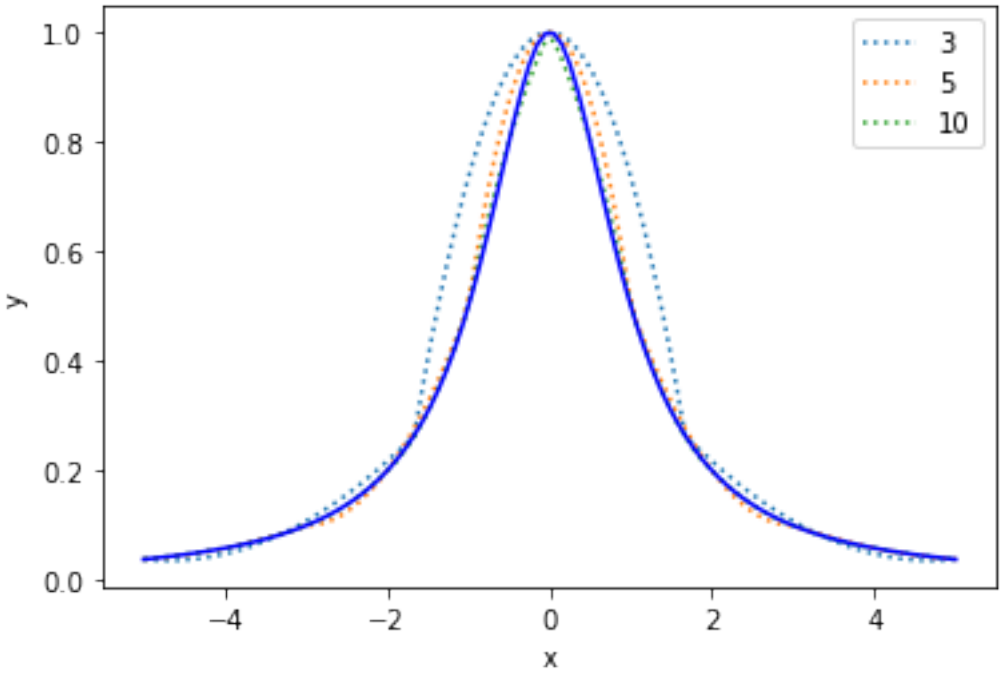
Soit $f : [a,b] \to R$ continue et $a = x_0 < \ldots < x_n = b$. Sur chaque sous-intervalle $[x_i, x_{i+1}]$, on fait une interpolation de **degré 2** avec les **3 noeuds** $x_i, x_{i+\frac{1}{2}}, x_{i+1}$, où $x_{i+\frac{1}{2}}$ est le milieu de $[x_i, x_{i+1}]$. Sur

chaque sous-intervalle $I_i = [x_i, x_{i+1}]$, on interpole $f_{|I_i}$ par un polynôme de degré 2. Le polynôme par morceaux (polynôme composite) qu'on obtient est noté $\Pi_2^H f(x)$ et on a:

$$\Pi_2^H f(x) = f(x_i)\varphi_0^{(i)}(x) + f(x_{i+\frac{1}{2}})\varphi_1^{(i)}(x) + f(x_{i+1})\varphi_2^{(i)}(x) \quad \text{pour } x \in [x_i, x_{i+1}]$$

où $\varphi_0^{(i)}(x), \varphi_1^{(i)}(x), \varphi_0^{(i)}(x)$ sont les polynômes de la base de Lagrange associés aux noeuds $(x_i, x_{i+\frac{1}{2}}, x_{i+1})$.

[25]:
```
from InterpolationLib import PiecewiseQuadraticInterpolation as  PiH2

# intervalle et fonction
a = -5; b = 5
f = lambda x : 1/(1+x**2)

# Values of N to use
Nrange = [3,5,10]
# plotting points
z = np.linspace(-5, 5, 100)

for N in Nrange :
    plt.plot(z, PiH2(a,b,N,f,z), ':')

plt.plot(z, f(z), 'b')
plt.xlabel('x'); plt.ylabel('y'); #plt.title('data')
plt.legend(Nrange)
plt.show()
```



Exercice Calculez l'erreur d'interpolation de la fonction de Runge dans l'intervalle $[-5,5]$ quand on utilise l'interpolation linéaire par morceaux

1. Théoriquement
2. Faites un graphique qui montre la convergence en fonction de $H = \frac{b-a}{N} = \frac{10}{N}$
3. Refaites le même exercice pour l'interpolation quadratique par morceaux

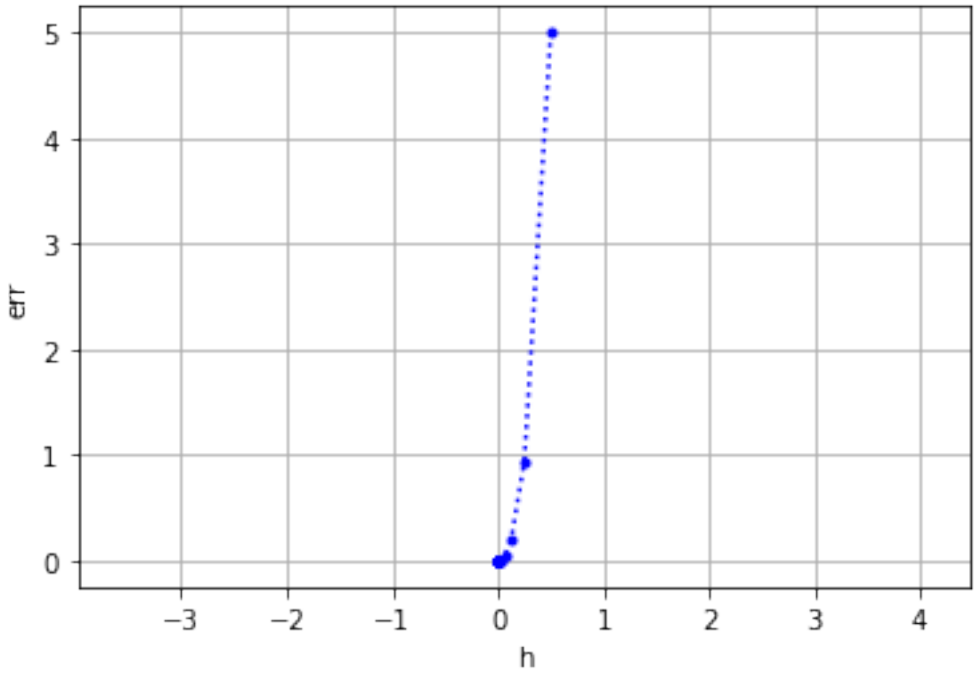
Remarque On s'attend à ce que l'erreur soit quadratique en H. Pour le voir il faut utiliser des axes logarithmiques dans les deux directions.

Admettons que l'erreur converge quadratiquement vers 0 par rapport à H, e.g

```
err = lambda H : 3*H**2 + 2*H**3;
```

Comment est le graphique de cette fonction dans l'intervalle $[10^{-6},1]$? Que se passe-t-il si on change les axes avec `plt.xscale('log')` et `plt.yscale('log')` ? Quelle est la pente de `err` dans ce système ?

[26]:
```
## Assume the error is quadratic
err = lambda H : 10*H**2 + 20*H**3;

# take h has powers of 2: 2^(-20),2^(-19), ..., 2^(-1)
h = np.power(2,np.linspace(-20, -1, 20, endpoint=True) )

plt.plot(h, err(h), 'b:.')

plt.xlabel('h'); plt.ylabel('err');

# plt.xscale('log')
# plt.yscale('log')

plt.grid(True)
# try with and wothout equal axis
plt.axis('equal')
plt.show()
```



[ ]:

1.5 Approximation au sens des moindres carrés

Soit $m \geq 0$ un nombre entier. Etant donnés $(n+1)$ points ~~distincts~~ x_0, x_1,... x_n et $(n+1)$ valeurs y_0, y_1,... y_n, on cherche un polynôme p de degré $m < n$, tel que

$$\sum_{j=0}^{n} |y_j - p(x_j)|^2 \qquad \text{soit le plus petit possible.}$$

On appelle **polynôme aux moindres carrés de degré** m le polynôme $\hat{p}_n(x)$ de degré m tel que

$$\sum_{j=0}^{n} |y_j - \hat{p}_n(x_j)|^2 \leq \sum_{j=0}^{n} |y_j - p_n(x_j)|^2 \qquad \forall p_m(x) \in \mathbb{P}_m \tag{3}$$

Nous cherchons les coefficients du polynôme $p(x) = a_0 + a_1 x + ... + a_m x^m$ qui satisfait *au mieux* les $(n+1)$ équations $p(x_k) = y_k, k = 0, ..., n$, c'est-à-dire

$$a_0 + a_1 x_k + ... + a_m x_k^m = y_k, k = 0, ..., n$$

Ce système s'écrit sous forme matricielle

$$\begin{pmatrix} 1 & x_0 & \cdots & x_0^m \\ \vdots & & & \vdots \\ 1 & x_n & \cdots & x_n^m \end{pmatrix} \begin{pmatrix} a_0 \\ \vdots \\ a_m \end{pmatrix} = \begin{pmatrix} y_0 \\ \vdots \\ y_n \end{pmatrix}$$

Puisque $m < n$, on ne peut pas résoudre ce système de façon classique.

Il faut le résoudre au sens des moindres carrés, en considérant:

$$B^T B\mathbf{a} = B^T \mathbf{y}$$

Où B est la matrice $(m+1) \times (n+1)$ du système, $\mathbf{a} \in \mathbf{R}^{n+1}$ le vecteur des inconnues et $\mathbf{y} \in \mathbf{R}^{m+1}$ le vecteur des données.

Ce système linéaire est dit **système d'équations normales**. On peut montrer que les équations normales sont équivalentes au problème de minimisation.

Remarque: *La résolution de ce système demande parfois des méthodes plus avancées que l'élimination de Gauss, comme la factorisation QR. Pour l'instant on va se contenter d'utiliser* `np.linalg.solve`

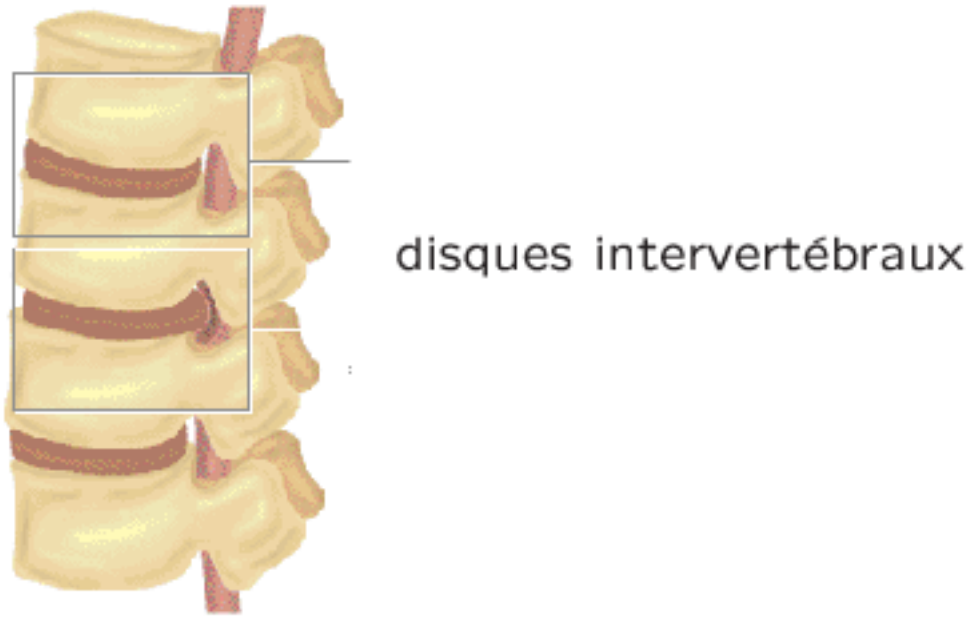
Pour construire cette matrice, vous pouvez utiliser la fonction

```
def VandermondeMatrix(x, m=0):
    # Input
    # x : +1 array with interpolation nodes
    # m : degree of the polynomial. If empty, chooses m=size(x)-1
    # Output
    # Matrix of Vandermonde of size m x n
```

que vous pouvez importer avec la commande

```
from InterpolationLib import VandermondeMatrix
```

Exemple On considère un test mécanique pour établir le lien entre contraintes ($MPa = 100N/cm^2$) et déformations relatives (cm/cm) d'un échantillon de tissu biologique (disque intervertébral, selon P. Komarek, Ch. 2 de *Biomechanics of Clinical Aspects of Biomedicine*, 1993, J. Valenta ed., Elsevier).



On cherche à approximer au sens des moindres carrés avec un polynôme $\hat{p}_n$ de degré $n = 1, 2, 3$. Les mesures effectuées sont les suivantes

```
sigma = np.array([0.00, 0.06, 0.14, 0.25, 0.31, 0.47, 0.50, 0.70]);
epsilon = np.array([0.00, 0.08, 0.14, 0.20, 0.22, 0.26, 0.27, 0.29]);
```

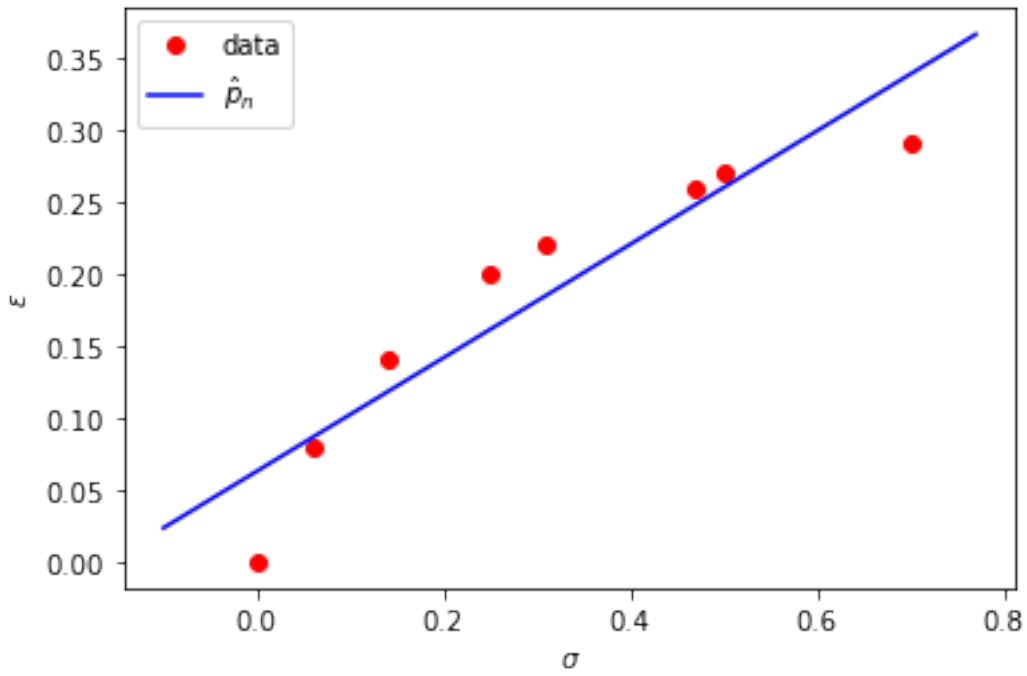
[27]:
```
# importing libraries used in this book
import numpy as np
import matplotlib.pyplot as plt

from InterpolationLib import VandermondeMatrix
```

[28]:
```
# data given:
sigma = np.array([0.00, 0.06, 0.14, 0.25, 0.31, 0.47, 0.50, 0.70]);
epsilon = np.array([0.00, 0.08, 0.14, 0.20, 0.22, 0.26, 0.27, 0.29]);

# degree of the polynomial
m = 1;

B = VandermondeMatrix(sigma,m)

# print(B)

# compute coefficients
a = np.linalg.solve( B.T.dot(B), B.T.dot(epsilon))
```

```
# print the coefficients on screen
print('The coefficients a_0, ..., a_m are')
print(a)
```

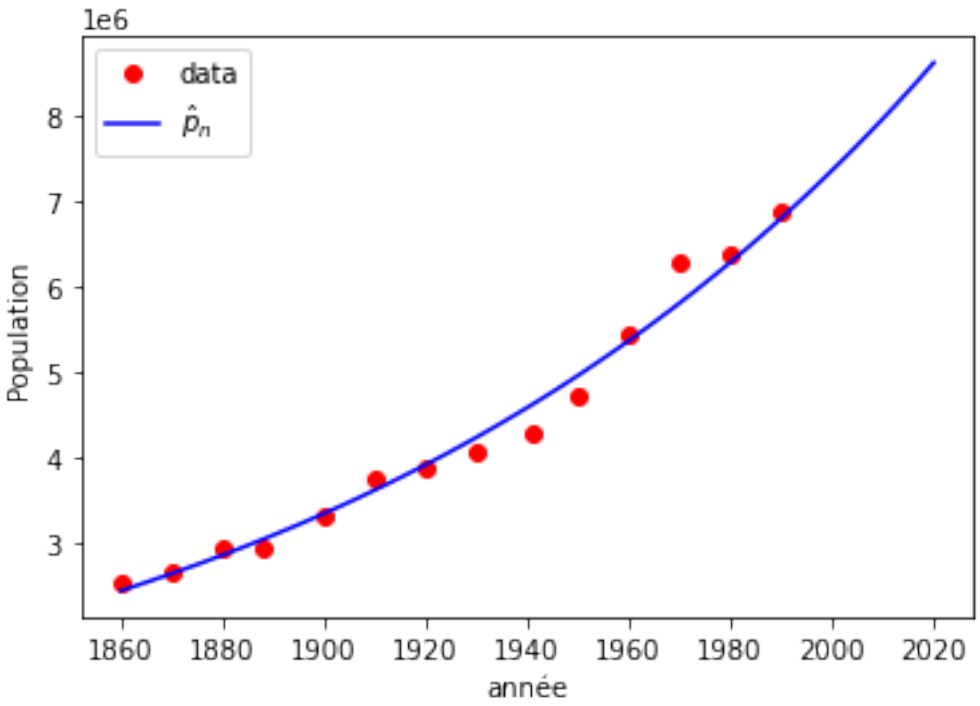
```
The coefficients a_0, ..., a_m are
[0.06288442 0.39379615]
```

[29]:
```
def polynomial(a,x):
    m = a.size-1
    # \hat p = a_0 + a_1 x + ... + a_n x^m
    # is equal to the scalar product between the vectors a and (1, x, ..., x^m) :
    return np.power( np.tile(x, (m+1, 1)).T , np.linspace(0,m,m+1)).dot(a)


# points used to plot the graph, slightly larger than data
z = np.linspace(sigma[0]-0.1, sigma[-1]*1.1, 100)

plt.plot(sigma, epsilon, 'ro', z, polynomial(a,z),'b')
plt.xlabel('$\sigma$'); plt.ylabel('$\epsilon$');
plt.legend(['data','$\hat p_n$'])
plt.show()
```



```
The coefficients a_0, ..., a_n are
[-0.01356599  0.00791249]
```

[34]:
```
def expPolynomial(a,x):
    m = a.size-1
    # \hat p = a_0 + a_1 x + ... + a_n x^m
    # is equal to the scalar product between the vectors a and (1, x, ..., x^m) :
    return np.exp(np.power( np.tile(x, (m+1, 1)).T , np.linspace(0,m,m+1)).
 ↪dot(a))

# points used to plot the graph, slightly larger than data
z = np.linspace(x[0], 2020, 100)

plt.plot(x, np.exp(y), 'ro', z, expPolynomial(a,z),'b')
plt.xlabel('année'); plt.ylabel('Population');
plt.legend(['data','$\hat p_n$'])
plt.show()
```



[ ]: