Analyse Numérique

Simone Deparis

June 4, 2023

Contents

1 Equations non-linéaires

Objectif : trouver les zéros (ou racines) d'une fonction $f : [a, b] \to \mathbf{R}$:

$$\alpha \in [a, b] \ : \ f(\alpha) = 0$$

3.1 Dichotomie

(Enlevé pour l'examen)

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

1.1 Méthode de Newton

Soit $f : \mathbb{R} \to \mathbb{R}$ une fonction différentiable.

Soit $x^{(0)}$ un point donné. On considère l'équation de la droite $y(x)$ qui passe par le point $(x^{(k)}, f(x^{(k)}))$ et qui a comme pente $f'(x^{(k)})$,

$$y(x) = f'(x^{(k)})(x - x^{(k)}) + f(x^{(k)}).$$

On définit $x^{(k+1)}$ comme étant le point où cette droite intersecte l'axe x, c'est-à-dire $y(x^{(k+1)}) = 0$. On en déduit que :

$$x^{(k+1)} = x^{(k)} - \frac{f(x^{(k)})}{f'(x^{(k)})}, \quad k = 0, 1, 2 \ldots .$$

[2]:
```
# importing libraries used in this book
import numpy as np
import matplotlib.pyplot as plt
```

Exercice (5, Série 1) On cherche les zéros de la fonction

$$f(x) = \frac{1}{2} \sin\left(\frac{\pi x}{2}\right) + 1 - x \; .$$

1. Vérifiez qu'il y a au moins un zéro α dans l'intervalle $[0, 2]$.
2. Ecrivez la méthode de Newton pour trouver le zéro α de la fonction $f(x)$ et calculez la première itération à partir de la valeur initiale $x^{(0)} = 1$.
3. Calculez les zéros α de la fonction f avec la méthode de Newton (fonction `newton` que vous devrez écrire)

```
def  Newton( F, dF, x0, tol, nmax ) :
    # NEWTON Find the zeros of a nonlinear equations.
    # NEWTON(F,DF,X0,TOL,NMAX) tries to find the zero X of the
    # continuous and differentiable function F nearest to X0 using
    # the Newton method. DF is a function which take X and return the derivative of F.
    # If the search fails an error message is displayed.
    #
    # returns the value of the
    # residual R in X,the number of iterations N required for computing X and
    # INC the increments computed by Newton.

    return x, r, n, inc
```

Choisissez $x^{(0)} = 1$ comme point de départ pour la méthode et utilisez une tolérance $tol = 10^{-4}$ sur la valeur absolue de l'incrément entre deux itérations successives $|x^{(k+1)} - x^{(k)}|$.

Dans le cas de la méthode de Newton, l'incrément est une bonne approximation de l'erreur.

[3]:
```
def  Newton( F, dF, x0, tol, nmax ) :
    '''
    NEWTON Find the zeros of a nonlinear equations.
    NEWTON(F,DF,X0,TOL,NMAX) tries to find the zero X of the
    continuous and differentiable function F nearest to X0 using
    the Newton method. DF is a function which take X and return the derivative␣
 ↪of F.
    If the search fails an error message is displayed.

    Outputs : [x, r, n, inc, x_sequence]
```

```
    x : the approximated root of the function
    r : the absolute value of the residualin X
    n : the number of iterations N required for computing X and
    inc : the increments computed by Newton.
    x_sequence : the sequence computed by Newton
    '''

    # Initial values
    n = 0
    xk = x0

    # initialisation of loop components
    # increments (in abs value) at each iteration
    inc = []
    # in case we wish to plot the sequence
    x = [x0]

    # diff : last increment,
    diff = tol + 1 # initially set larger then tolerance

    # Loop until tolerance is reached
    while ( diff >= tol and n <= nmax ) :
        # Newton iteration
        deltax  = F(xk) / dF(xk)
        xk1 = xk - deltax

        # increments
        diff = np.abs(deltax)
        inc.append (diff)

        # prepare the next loop
        n = n + 1
        xk = xk1
        x.append(xk)

    # Final residual
    rk1 = np.abs(F(xk1))

    # Warning if not converged
    if n > nmax :
        print('Newton stopped without converging to the desired tolerance ')
        print('because the maximum number of iterations was reached')

    return xk1, rk1, n, inc, np.array(x)
```

[4]:
```
f  = lambda x : 0.5*np.sin(np.pi*x/2)+1-x
df = lambda x : 0.25*np.pi*np.cos(np.pi*x/2)-1
```

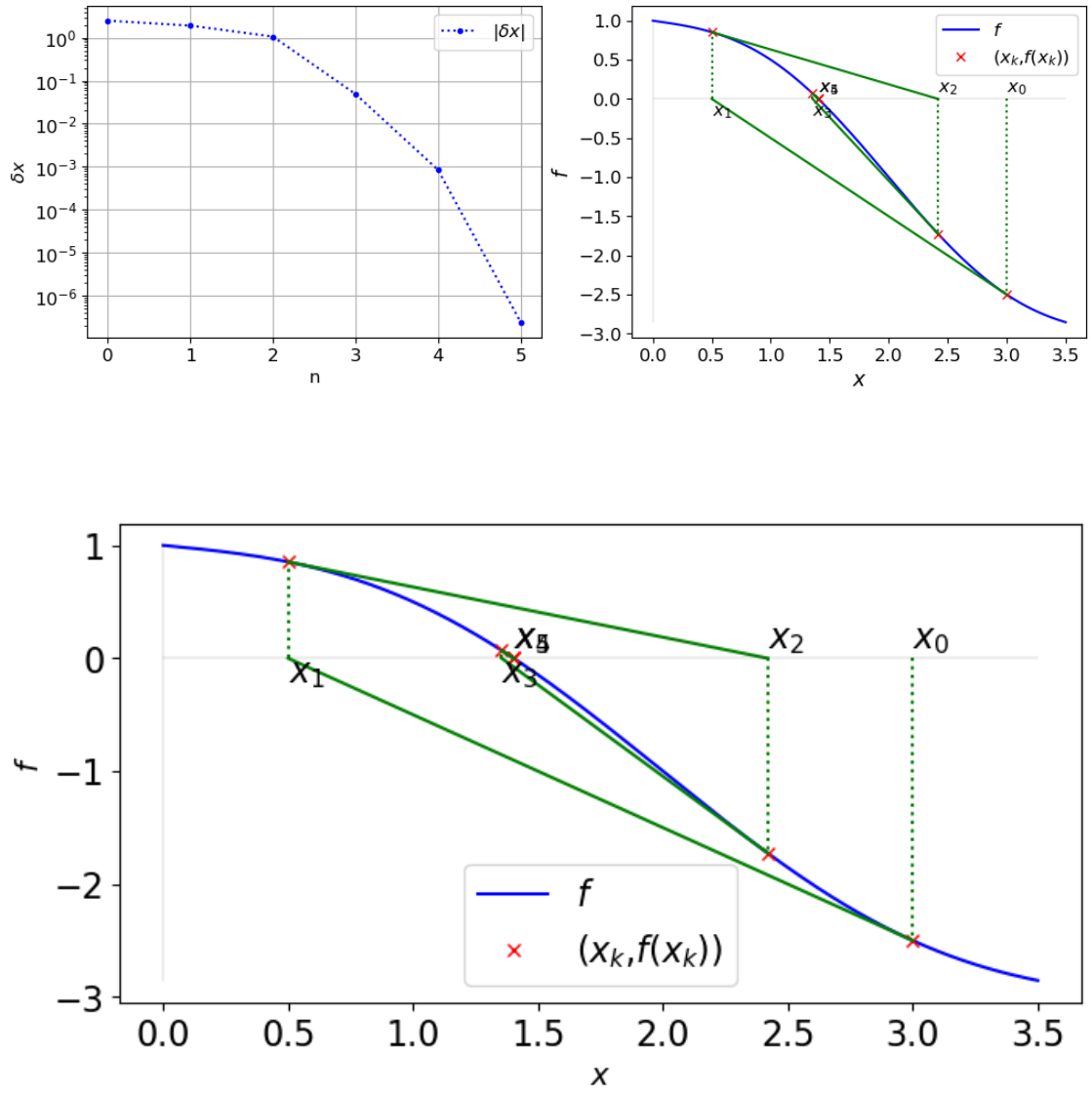
```
x0   = 1
tol  = 1e-4
nmax = 10

zero, residual, niter, inc, x = Newton(f, df, x0, tol, nmax)

print(f'The zero computed is {zero:1.4f}')
print(f'Newton stoppedconverged in {niter} iterations');
print(f'with a residual of {residual:1.4e}.\n');
```

```
The zero computed is 1.4031
Newton stoppedconverged in 4 iterations
with a residual of 3.3120e-12.
```

[5]:
```
from NonLinearEquationsLib import plotNewtonIterations
```

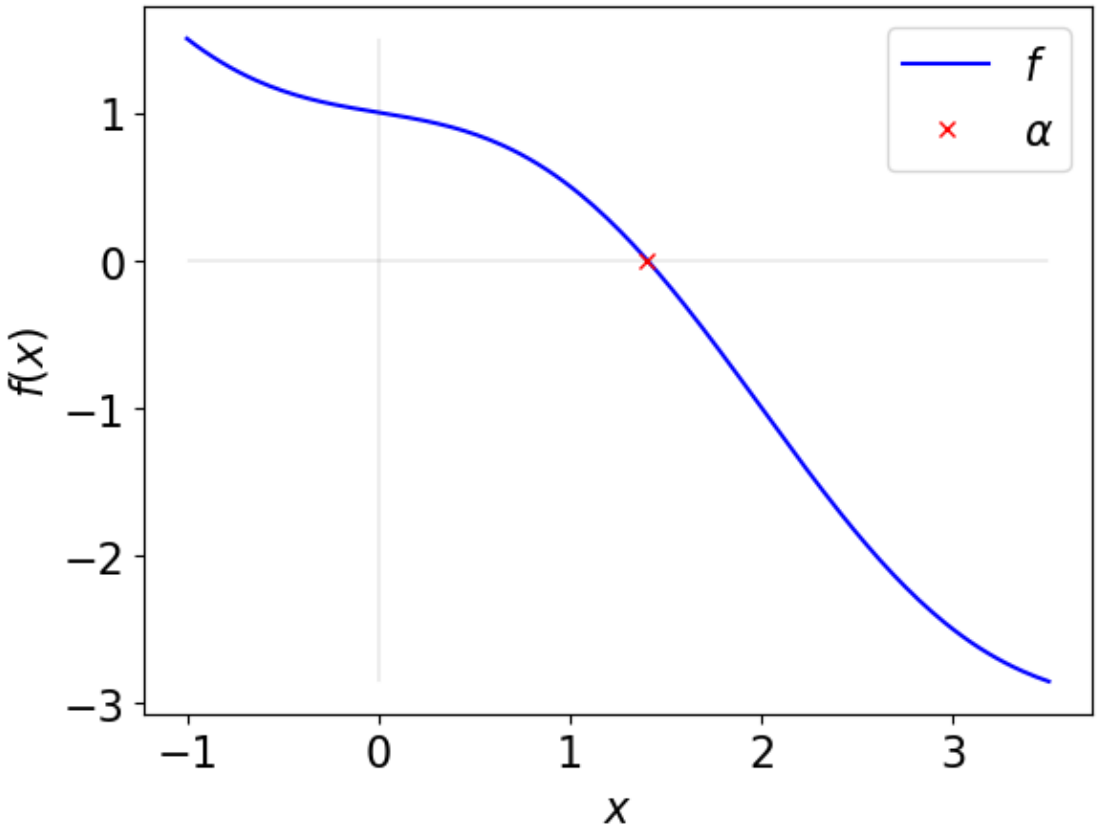
[6]:
```
[a,b] = [0,3.5]
x0   = 3
zero, residual, niter, inc, x = Newton(f, df, x0, tol, nmax)

# Rezise plots, which are usually too small
plt.figure(figsize=(12, 4))
plt.rcParams.update({'font.size': 12})

# Subplot 1 over 2, 1st one
plt.subplot(121)

#plt.plot(range(MaxIterations), RelativeError, 'b:.')
plt.plot(range(niter), inc, 'b:.')

plt.xlabel('n'); plt.ylabel('$\\delta x$');
plt.grid(True)
#plt.xscale('log')
plt.yscale('log')
plt.legend(['$|\\delta x|$'])


# Subplot 1 over 2, 2nd one
plt.subplot(122)

plotNewtonIterations (a,b,f,x,200)

plt.show()

# Rezise plots, which are usually too small
plt.figure(figsize=(8, 4))
```

```
plt.rcParams.update({'font.size': 16})

plotNewtonIterations (a,b,f,x,200)
# plt.savefig('Newton-iterations.png', dpi=600)
plt.show()
```



[7]:
```
[a,b] = [-1,3.5]
z = np.linspace(a,b,200)
plt.plot(z,f(z), 'b-', x[6],f(x[6]), 'rx')
plt.ylabel('$f(x)$'); plt.xlabel('$x$');

# Plot the x,y-axis
```

```
plt.plot([a,b], [0,0], 'k-',linewidth=0.1)
plt.plot([0,0], [np.min(f(z)),np.max(f(z))], 'k-',linewidth=0.1)

plt.legend(['$f$','$\\alpha$'])
# plt.savefig('Newton-fx-alpha.png', dpi=600)

plt.show()
```



[ ]:

1.2 Methode de point fixe

1.2.1 Exercice (1, Série 3)

On considère les méthodes de point fixe $x^{(n+1)} = g_i(x^{(n)}) \quad (i = 1, 2, 3)$ avec:

$$g_1(x^{(n)}) = \frac{1}{2}e^{x^{(n)}/2}, \qquad g_2(x^{(n)}) = -\frac{1}{2}e^{x^{(n)}/2}, \qquad g_3(x^{(n)}) = 2\ln(2x^{(n)}),$$

dont les fonctions d'itération $g_i(x)$ sont visualisées sur la figure plus bas

1. Pour chaque point fixe $\bar{x}$ de la fonction d'itération g_i $(i = 1, 2, 3)$, on suppose d'avoir choisi une valeur initiale $x^{(0)}$ proche de $\bar{x}$. Etudiez si la méthode converge vers $\bar{x}$.
2. Pour chaque fonction d'itération g_i, déterminez graphiquement pour quelles valeurs initiales $x^{(0)}$ la méthode de point fixe correspondante converge et vers quel point fixe.
3. Montrez que si $\bar{x}$ est un point fixe de la fonction g_i $(i = 1, 2, 3)$, alors il est aussi un zéro de la fonction $f(x) = e^x - 4x^2$ (dont le comportement est tracé sur la dernière figure).
4. Comment peut-on calculer les zéros de f?

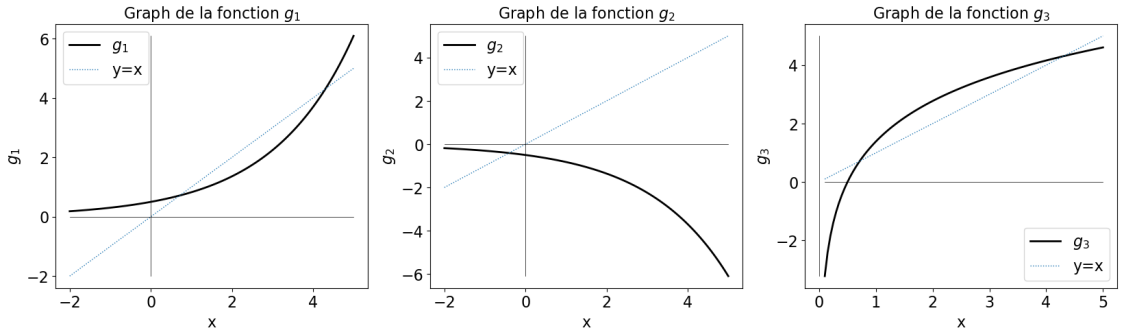
L'exercice 2 est à faire sur papier, ici une indication par ordinateur

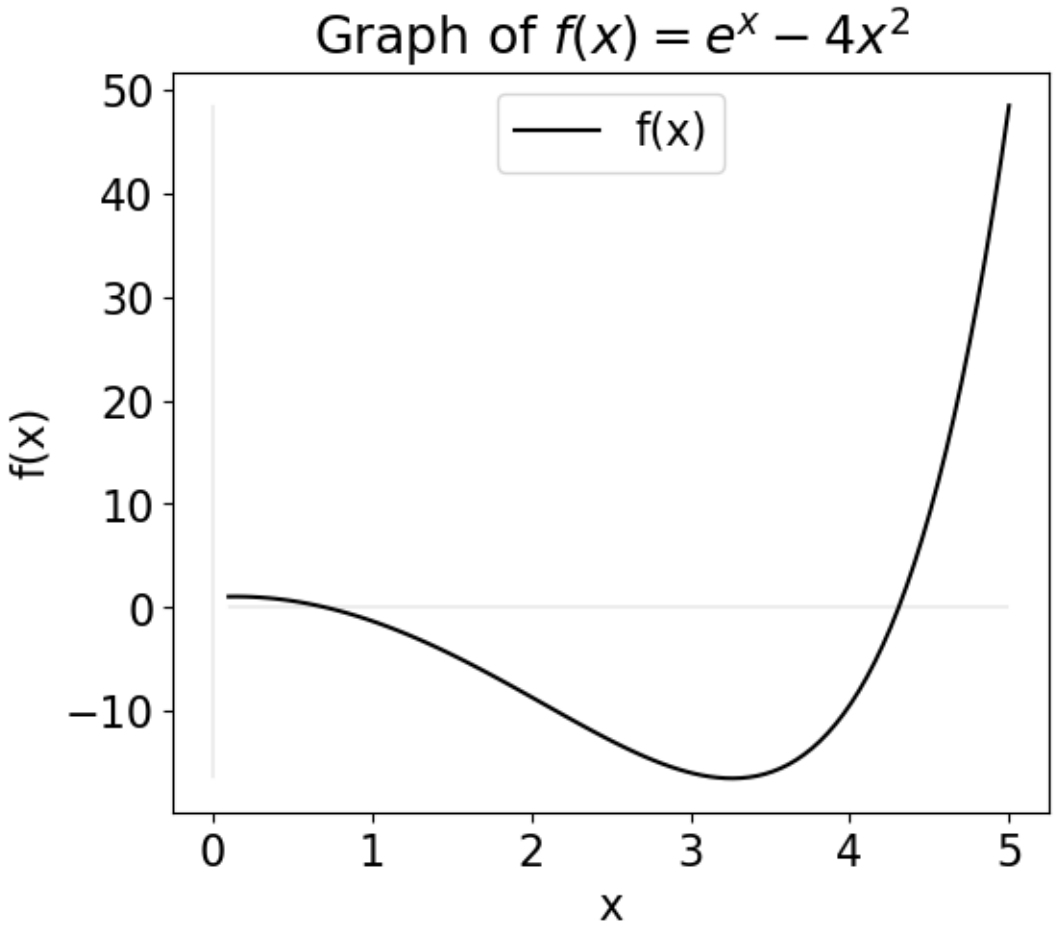
[8]:
```
# importing libraries used in this book
import numpy as np
import matplotlib.pyplot as plt
```

[9]:
```
from NonLinearEquationsLib import plotPhi, FixedPoint, plotPhiIterations
```

[10]:
```
plt.rcParams['figure.figsize'] = [20, 5]

plt.subplot(1,3,1)
[a,b] = [-2,5]
phi1 = lambda x : np.exp(x/2)/2
plotPhi(a,b,phi1,'$g_1$')

plt.subplot(1,3,2)
[a,b] = [-2,5]
phi2 = lambda x : - np.exp(x/2)/2
plotPhi(a,b,phi2,'$g_2$')

plt.subplot(1,3,3)
[a,b] = [1e-1,5]
phi3 = lambda x : 2*np.log(2*x)
plotPhi(a,b,phi3,'$g_3$')

plt.show()

# Graph of the fonction $f(x)=e^x-4x^2$
N = 100
z = np.linspace(a,b,N)
f = lambda x : np.exp(x) - 4*x*x

plt.subplot(1,3,1)
plt.plot(z,f(z),'k-')

plt.xlabel('x'); plt.ylabel('f(x)');
# Plot the x,y-axis
plt.plot([a,b], [0,0], 'k-',linewidth=0.1)
```

```
plt.plot([0,0], [np.min(f(z)),np.max(f(z))], 'k-',linewidth=0.1)
plt.legend(['f(x)'])
plt.title('Graph of $f(x)=e^x-4x^2$')

plt.show()
```



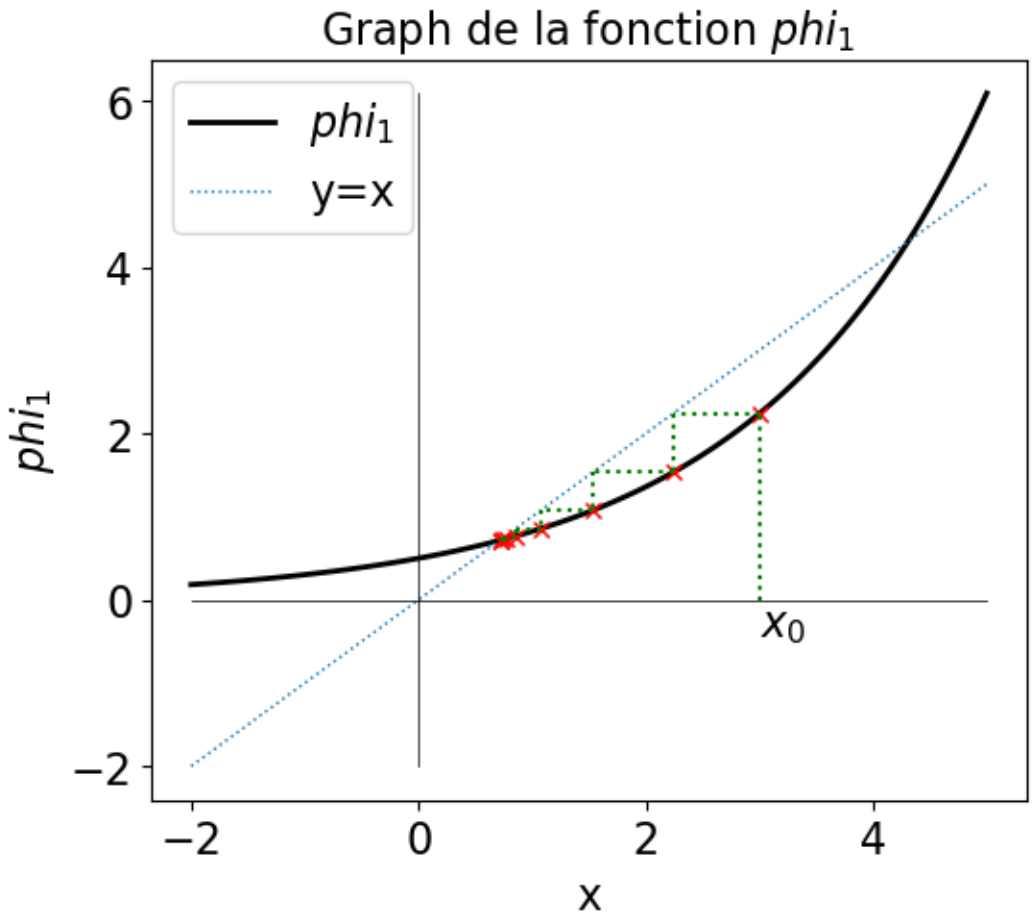


Partie 1 *Pour chaque point fixe $\bar{x}$ de la fonction d'itération g_i $(i = 1, 2, 3)$, on suppose choisir une valeur initiale $x^{(0)}$ proche de $\bar{x}$. Etudiez si la méthode converge vers $\bar{x}$.*

On va utiliser la fonction `FixedPoint` qui se trouve dans `NonLinearEquationsLib.py`

```
def  FixedPoint( phi, x0, a, b tol, nmax ) :
    '''
    FixedPoint Find the fixed point of a function by iterative iterations
    FixedPoint( PHI,X0,a,b, TOL,NMAX) tries to find the fixedpoint X of the a
    continuous function PHI nearest to X0 using
    the fixed point iterations method.
    [a,b] : if the iterations exit the interval, the method stops
    If the search fails an error message is displayed.

    Outputs : [x, r, n, inc, x_sequence]
    x : the approximated fixed point of the function
    r : the absolute value of the residual in X : |phi(x) - x|
    n : the number of iterations N required for computing X and
    x_sequence : the sequence computed by Newton


    ...
    return xk1, rk1, n, np.array(x)
    '''
```

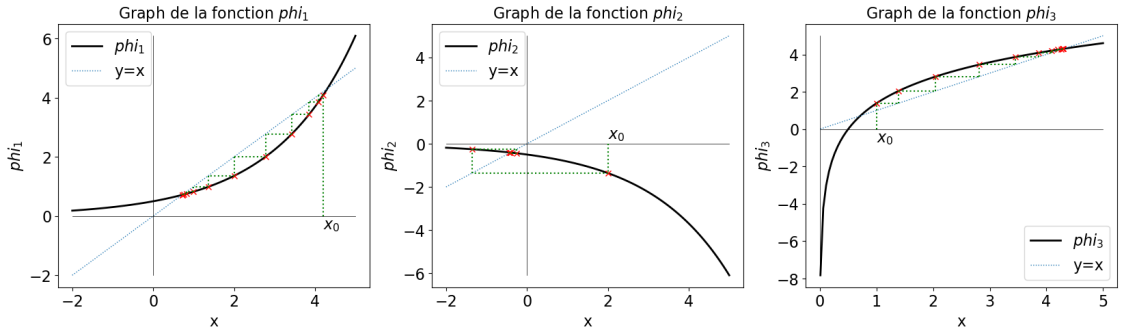
[11]:
```
tol  = 1e-2
nmax = 10

# Choose fonction phi
[a,b] = [-2,5]
phi = phi1
label = '$phi_1$'

# Initial Point
x0   = 3
zero, residual, niter, x = FixedPoint(phi, x0, a,b, tol, nmax)

plt.subplot(131)
plotPhi (a,b,phi,label)
plt.plot(x,phi(x), 'rx')

# plot the graphical interpretation of the Fixed Point method
plotPhiIterations(x)

plt.show()
```



[12]:
```
tol  = 1e-2
nmax = 10

# Choose fonction phi
intervals = [ [-2,5] , [-2,5] , [1e-2,5] ]
phiFunctions = [phi1, phi2, phi3]
labels = ['$phi_1$', '$phi_2$', '$phi_3$']
# Initial Points
initialPoints   = [4.2,2,1]

alpha = np.empty(3)

for k in range(3) :
    phi = phiFunctions[k]; x0 = initialPoints[k]
    a = intervals[k][0]; b = intervals[k][1]
    label = labels[k]
```

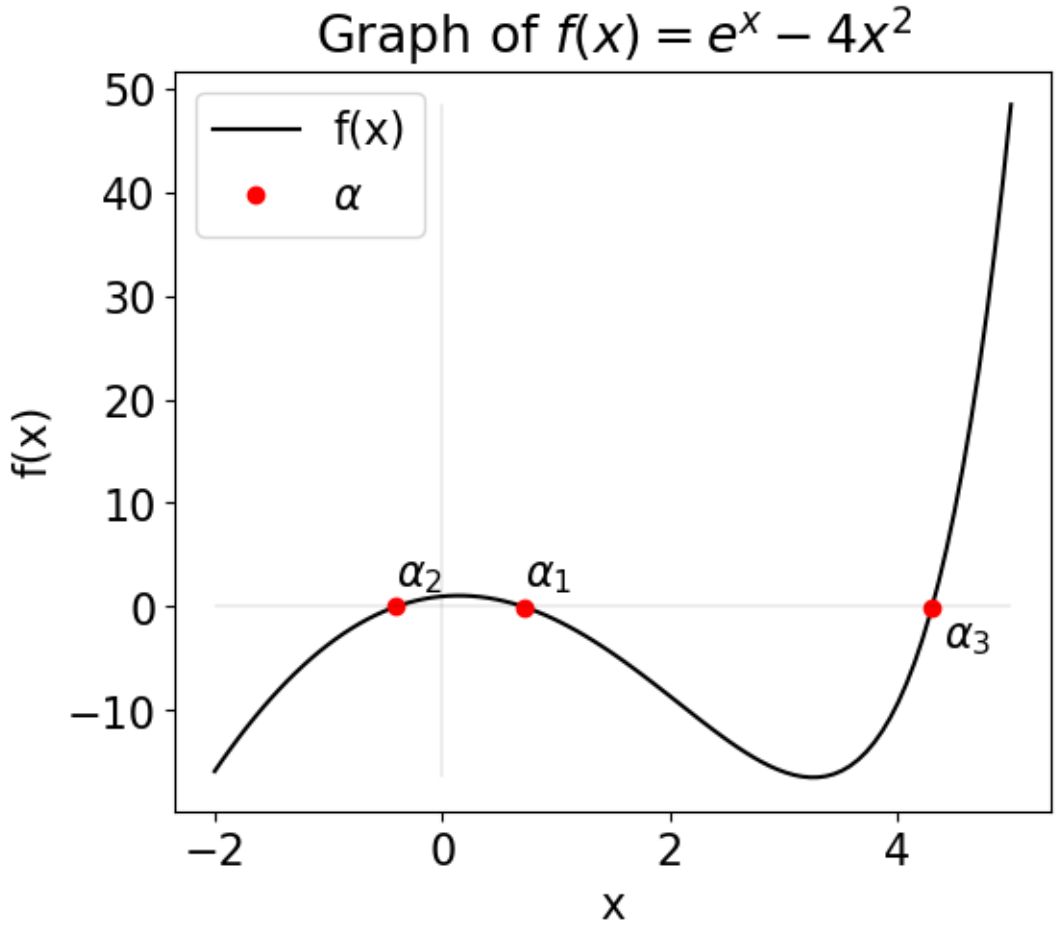
```
    alpha[k], residual, niter, x = FixedPoint(phi, x0, a,b, tol, nmax)

    plt.subplot(1,3,k+1)

    plotPhi (a,b,phi,label)
    plt.plot(x,phi(x), 'rx')

    # plot the graphical interpretation of the Fixed Point method
    plotPhiIterations(x)



plt.show()
```

```
FixexPoint stopped without converging to the desired tolerance
because the maximum number of iterations was reached
FixexPoint stopped without converging to the desired tolerance
because the maximum number of iterations was reached
```



[13]:
```
# Graph of the fonction $f(x)=e^x-4x^2$
N = 100
a,b = [-2,5]
z = np.linspace(a,b,N)
f = lambda x : np.exp(x) - 4*x*x

plt.subplot(1,3,1)
plt.plot(z,f(z),'k-')

# Solutions found:
plt.plot(alpha,f(alpha),'ro')
plt.annotate("$\\alpha_1$", (alpha[0], 2))
plt.annotate("$\\alpha_2$", (alpha[1], 2))
plt.annotate("$\\alpha_3$", (alpha[2]+0.1, -4))

plt.xlabel('x'); plt.ylabel('f(x)');
```

```
# Plot the x,y-axis
plt.plot([a,b], [0,0], 'k-',linewidth=0.1)
plt.plot([0,0], [np.min(f(z)),np.max(f(z))], 'k-',linewidth=0.1)
plt.legend(['f(x)','$\\alpha$'])
plt.title('Graph of $f(x)=e^x-4x^2$')

plt.show()
```



1.2.2 Critères d'arrêt

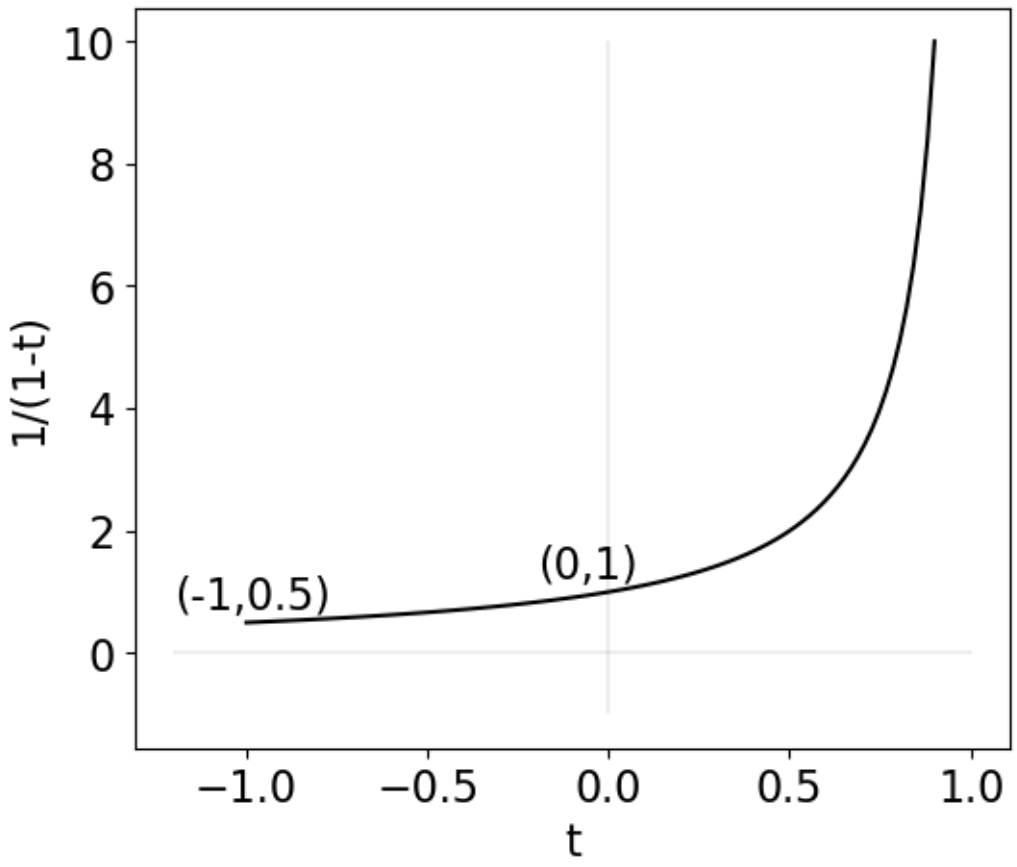
On a l'estimation

$$e^{(k)} = \frac{1}{\left(1-\phi'(\xi^{(k)})\right)}(x^{(k+1)} - x^{(k)}). = \gamma(\phi'(\xi^{(k)}))(x^{(k+1)} - x^{(k)})$$

On cherche à obtenir $|e^{(k)}| \approx \epsilon$ (une tolérance choisie).

On trace un graphe de la fonction $\gamma(t) = \frac{1}{1-t}$

[14]:
```
# Graph of the fonction $f(t)=1/(1-t)$
N = 100
a,b = [-1,0.9]
z = np.linspace(a,b,N)
f = lambda x : 1/(1-x)

plt.subplot(1,3,1)
plt.plot(z,f(z),'k-')

plt.annotate("(-1,0.5)", (-1.2, 0.7))

plt.annotate("(0,1)", (-0.2, 1.2))


plt.plot([-1.2,1], [0,0], 'k-',linewidth=0.1)
plt.plot([0,0], [-1,10], 'k-',linewidth=0.1)


plt.xlabel('t'); plt.ylabel('1/(1-t)');
# Plot the x,y-axis

plt.show()
```



2 Interpolation et approximation de données

2.1 Position du problème

2.1.1 Interpolation de données

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

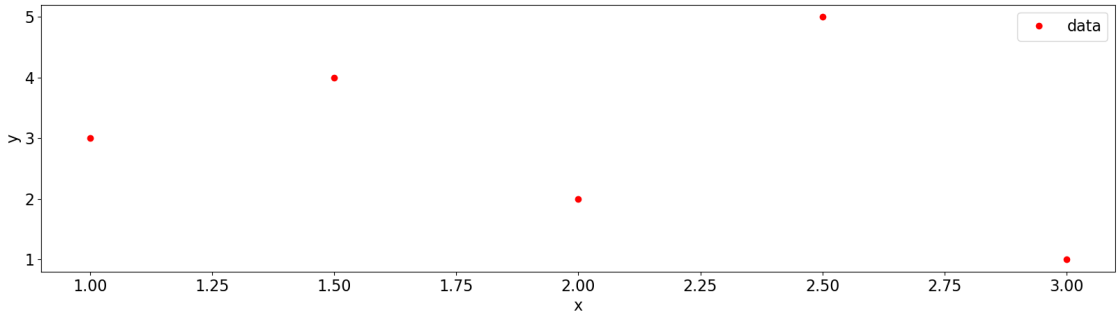
$$p(x_j) = y_j \qquad \text{pour } 0 \leq j \leq n.$$

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

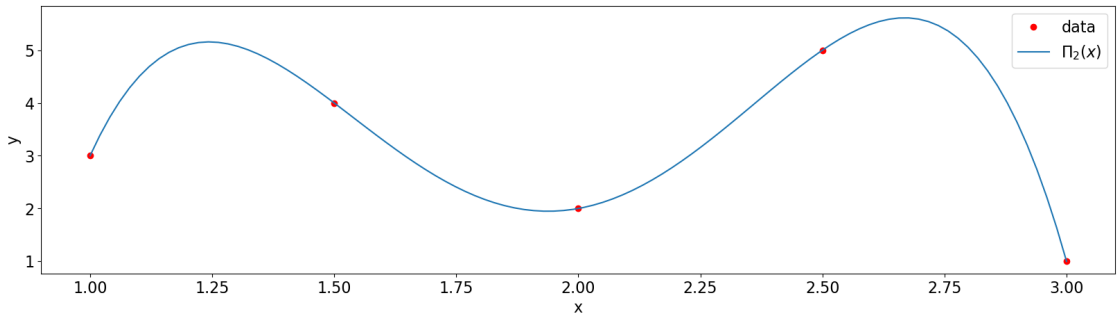
[15]:
```
# importing libraries used in this book
import numpy as np
import matplotlib.pyplot as plt
```

[16]:
```
# 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, $j = 0, \ldots, n$.

[17]:
```
# 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()
```



2.1.2 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,... x_n.

Exemple Soient

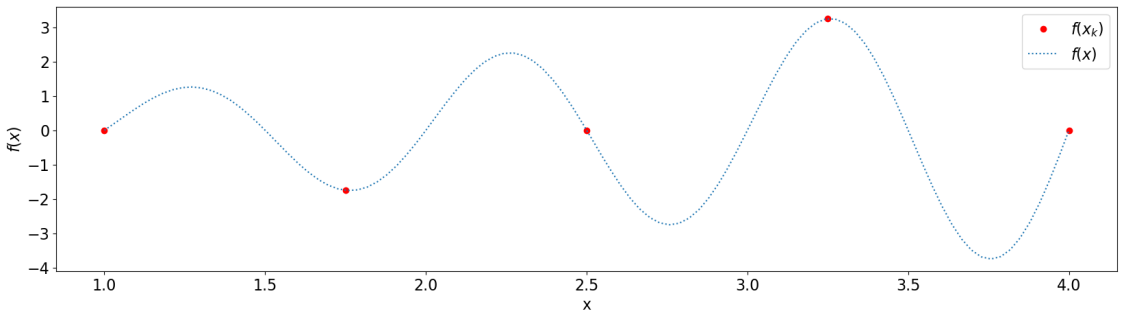
$$x_1 = 1, x_2 = 1.75, x_3 = 2.5, x_4 = 3.25, x_5 = 4$$

les points d'interpolation et

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

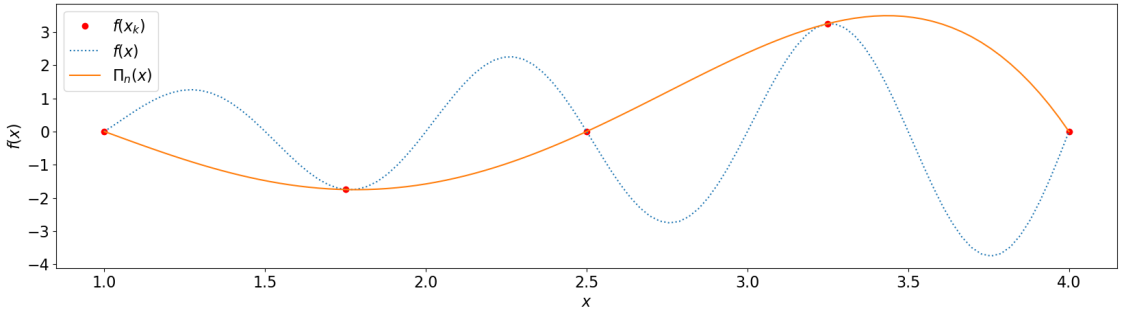
On cherche l'interpolant $\Pi_n f$ de degré $n = 4$

[18]:
```
# 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()
```



[19]:
```
# 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()
```



2.1.3 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

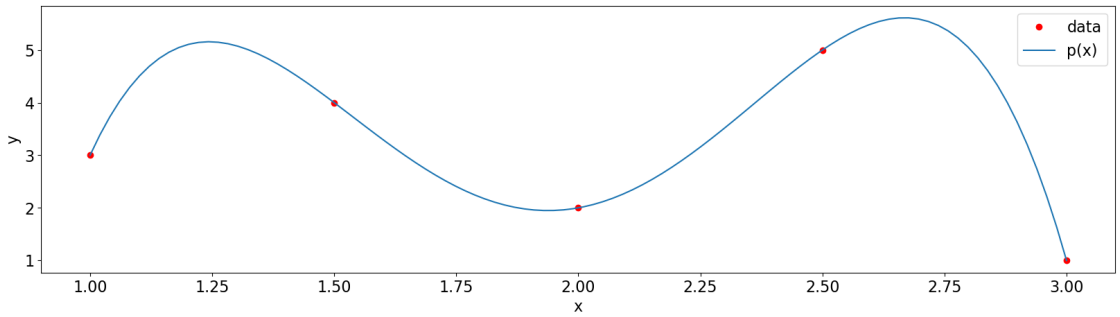
x_k	y_k
1	3
1.5	4
2	2
2.5	5
3	1

[20]:
```
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]
```

[21]:
```
# 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()
```



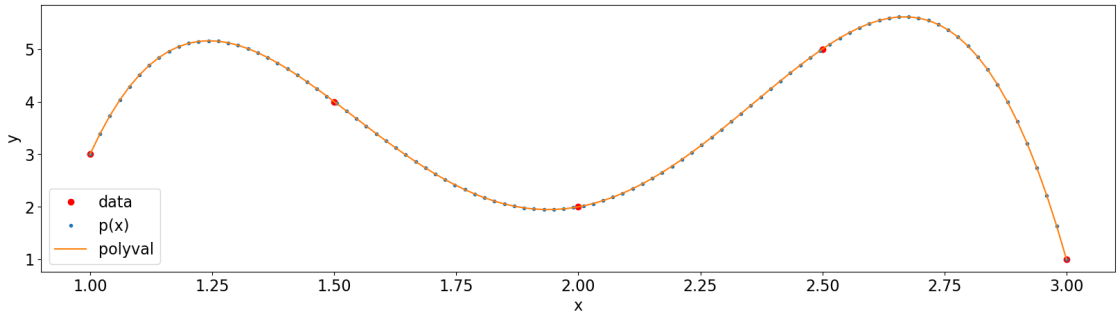
2.1.4 Alternatives : polyfit et polyval

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 !

[22]:
```
# 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()
```



[ ]:

2.2 Interpolation de Lagrange

2.2.1 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

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

[23]:
```
# 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. $\varphi_k(x_j) = \delta_{jk}, \qquad k, j = 0, \ldots, n$

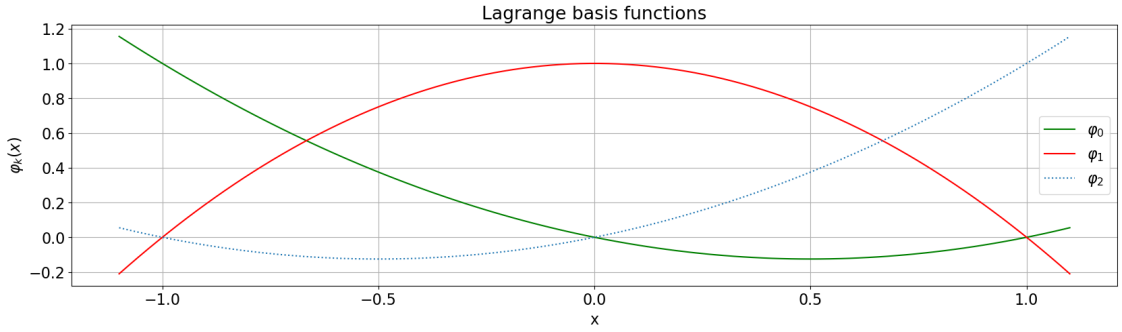
[24]:
```
# Defining the Lagrange basis functions
def phi(x,k,z):
    # the input variables are:
    # x : the interpolatory points
    # k : which basis function
    # z : where to evaluate the function


    # 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

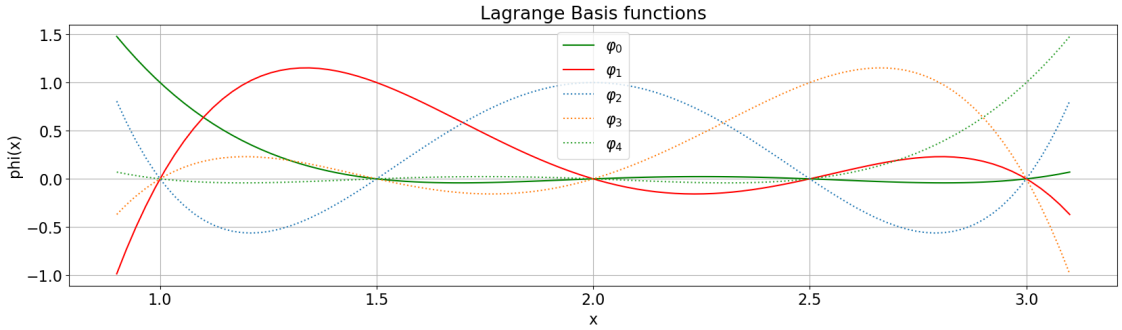
$$\begin{aligned}
\varphi_0(x) &= \frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)} = \frac{1}{2}x(x-1), \\
\varphi_1(x) &= \frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)} = -(x+1)(x-1), \\
\varphi_2(x) &= \frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)} = \frac{1}{2}x(x+1).
\end{aligned}$$

[25]:
```
# 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 $\varphi_k(x_j)$

[26]:
```
# 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()
```



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

2.3 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

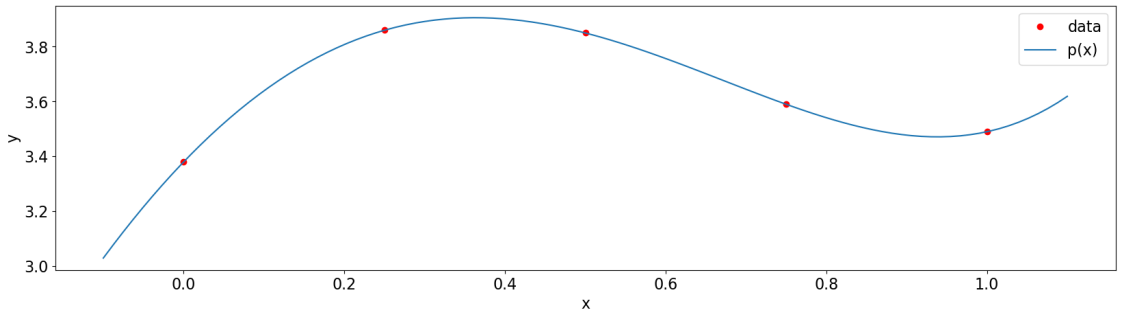
$$\Pi_n(z) = y_0\varphi_0 + ... + y_n\varphi_n$$

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

from InterpolationLib import LagrangeBasis as phi
```

[28]:
```
# 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
```

[29]:
```
# 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()
```



2.3.1 Erreur d'interpolation

Soient x_0, x_1, …, 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 ?

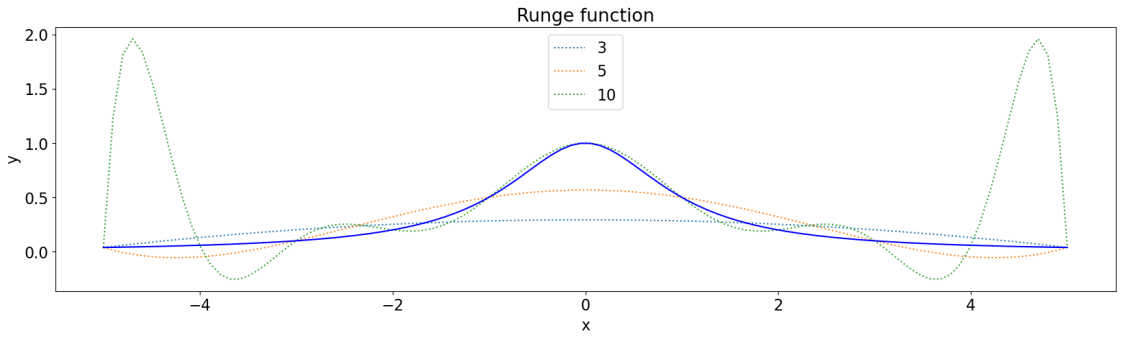
Comportement pour n grand: eg, la fonction de Runge. Le fait que

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

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.

[30]:
```
# 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.

[31]:
```
# 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
```

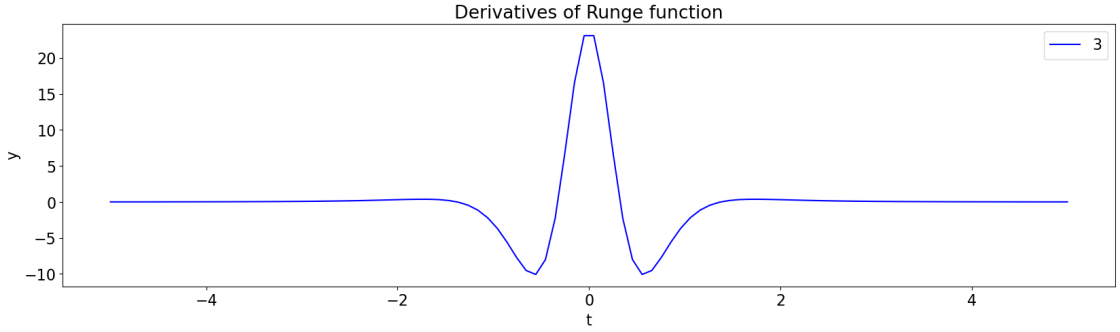
Pour définir une fonction qui accepte un array de valeur, il faut utiliser les lignes suivantes.

Ensuite on peut aussi dessiner le graphe ...

[32]:
```
# 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)
```

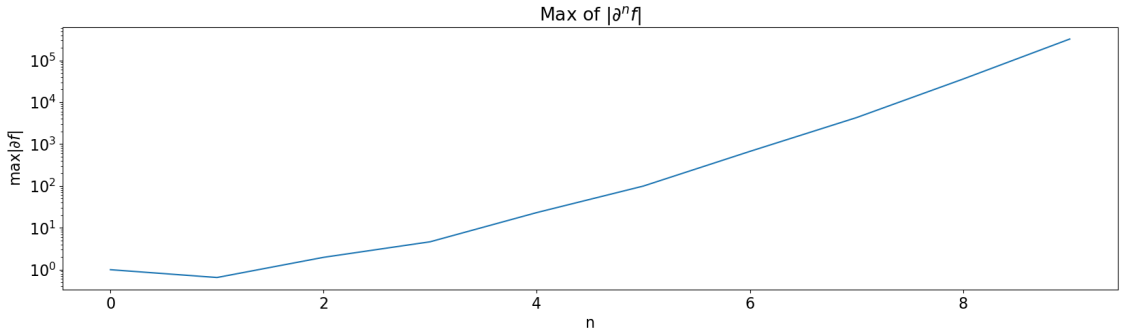
```
# the derivative can be set with k (not very elegant...)
k = 4
print(diff_f(4.5))

# 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.

[33]:
```
# 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()
```



2.3.2 Interpolation de Chebyshev

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

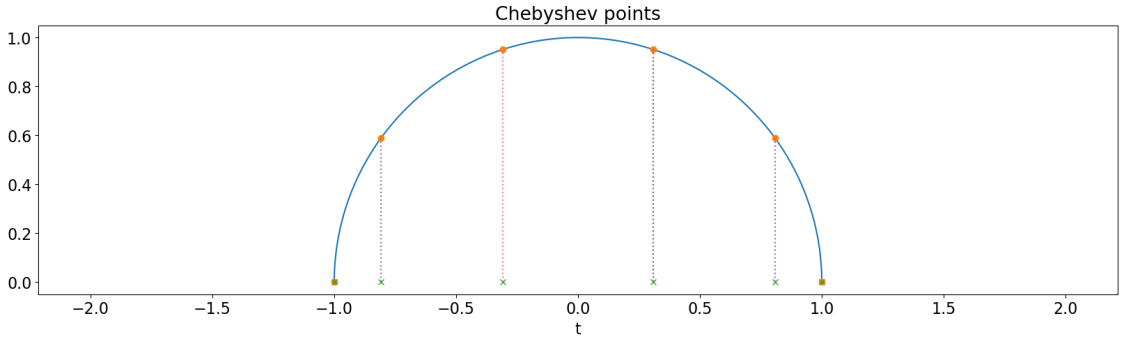
$$\hat{x}_i = -\cos(\pi i/n) \in [-1, 1]$$

les points de Chebyshev et on définit

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

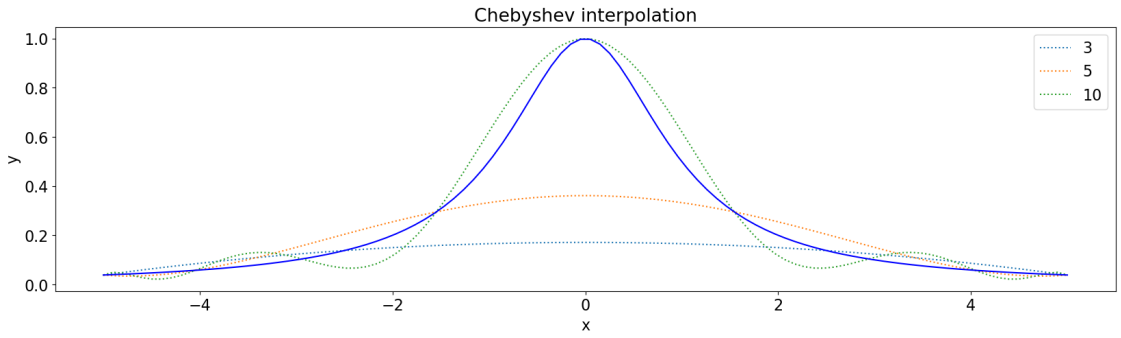
pour un intervalle arbitraire $[a, b]$. Pour une fonction continue $f \in C^1([a, b])$, le polynôme d'interpolation $\Pi_n f$ de degré n aux noeuds $\{x_i, i = 0, \ldots, n\}$ converge uniformément vers f quand $n \to \infty$.

[34]:
```
# 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.

[35]:
```
# 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()
```



[ ]:

2.4 Interpolation par intervalles

2.4.1 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 $i = 0, ..., N$

[36]:
```
# 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

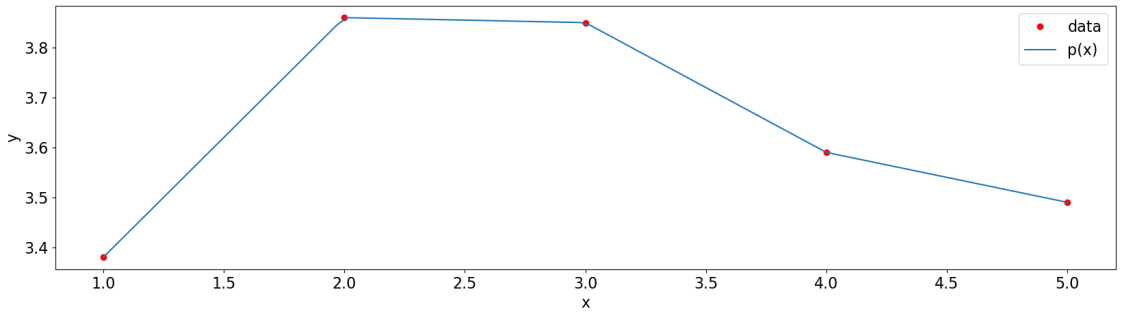
```
def PiecewiseLinearInterpolation(a,b,N,f,z):
    # the input variables are:
    # a,b : x[0] = a, x[n] = b
    # f : the corresponding data at the points x
    # z : where to evaluate the function
```

2.4.2 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

[37]:
```
# 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()
```

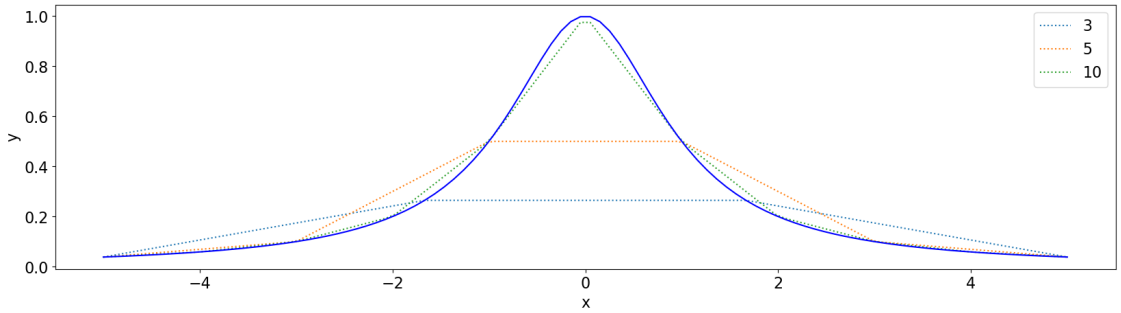


2.4.3 Exercice

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

[38]:
```
# 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()
```



2.4.4 Erreur d'interpolation linéaire par morceaux

Théorème (Enlevé pourl'examen)

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

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

2.4.5 Interpolation quadratique par morceaux

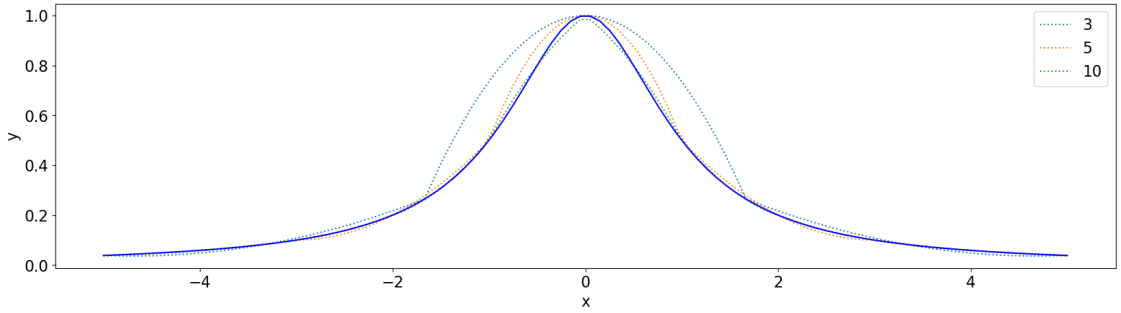
Soit $f : [a, b] \to R$ continue et $a = x_0 < \ldots < x_n = b$. Sur chaque sous-intervalle $[x_i, x_{i+1}]$, on fait une interpolation de **degré 2** avec les **3 noeuds** $x_i, x_{i+\frac{1}{2}}, x_{i+1}$, où $x_{i+\frac{1}{2}}$ est le milieu de $[x_i, x_{i+1}]$. Sur chaque sous-intervalle $I_i =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

[39]:
```
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.

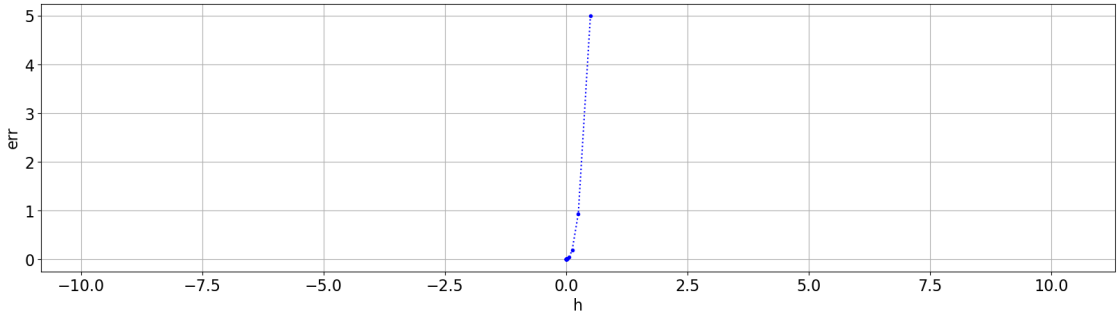
Admettons que l'erreur converge quadratiquement vers 0 par rapport à H, e.g

`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 ?

[40]:
```
## 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()
```



2.5 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`

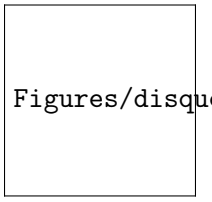
Pour construire cette matrice, vous pouvez utiliser la fonction

```
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

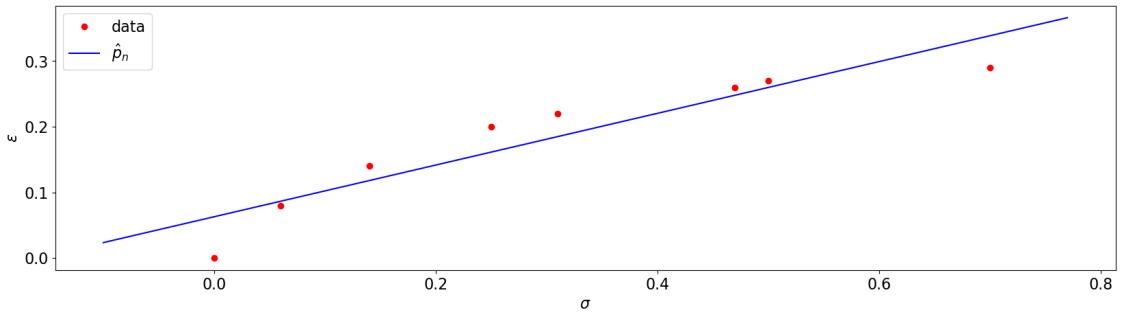
```
sigma = np.array([0.00, 0.06, 0.14, 0.25, 0.31, 0.47, 0.50, 0.70]);
epsilon = np.array([0.00, 0.08, 0.14, 0.20, 0.22, 0.26, 0.27, 0.29]);
```

[41]:
```
# importing libraries used in this book
import numpy as np
import matplotlib.pyplot as plt

from InterpolationLib import VandermondeMatrix
```

[42]:
```
# 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]
```

[43]:
```
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()
```



Exercice Depuis https://hsso.ch/fr/2012/b/14 on a téléchargé les données de la population suisse dans le fichier `Data/PopulationSuisse.csv`.

Approximez l'évolution de la population avec un polynôme de degré $n = 1, 2, 3, 7$.

Ensuite faites l'hypothèse de croissance exponentielle de la population, c'est-à-dire $p(x) = e^{a_1 x + a_0}$ où a_0 et a_1 sont des paramètres. Comment utiliser l'approximation polynômiale dans ce contexte ?

[44]:
```
# Data of the population has been dowloaded from https://hsso.ch/fr/2012/b/14
# into the file Data/PopulationSuisse.csv
import pandas as pd

# Read data from file 'PopulationSuisse.csv'
data = pd.read_csv("Data/PopulationSuisse.csv")
# Preview the first 5 lines of the loaded data
data.head()
```

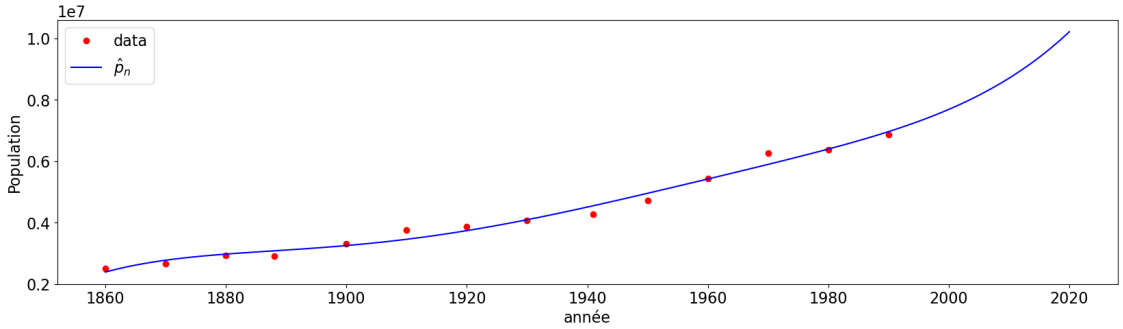
[44]:

	Année	Population
0	1860	2506784
1	1870	2654394
2	1880	2924702
3	1888	2917754
4	1900	3315443

[45]:
```
x = data['Année'].to_numpy()
y = data['Population'].to_numpy()

m = 9
B = VandermondeMatrix(x,m)

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

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

```
The coefficients a_0, ..., a_n are
[2506784 2654394 2924702 2917754 3315443 3753293 3880320 4066400 4265703
 4714992 5429061 6269783 6365960 6873687]
```

[46]:
```
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(x[0], 2020, 100)

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



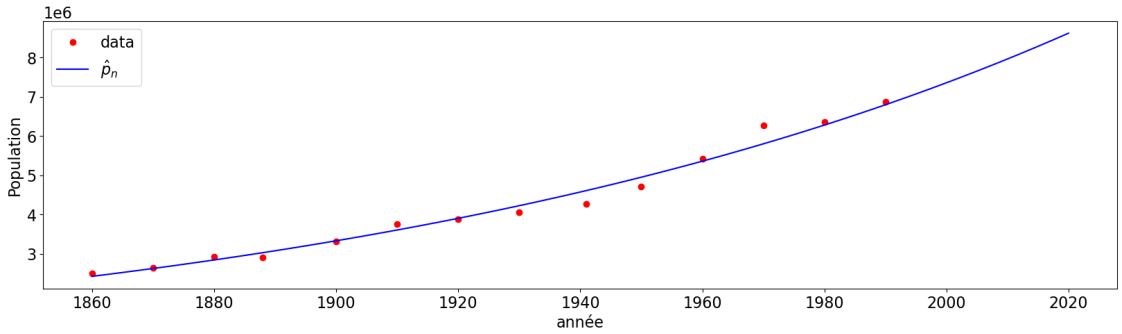
On assume une croissance exponentielle : $population(x) = C * e^{a_1 x} = e^{a_o + a_1 x}$

En d'autres termes: $\log(population)(x) = a_o + a_1 x$

[47]:
```
#

x = data['Année'].to_numpy()
y = np.log(data['Population'].to_numpy())

m = 1
B = VandermondeMatrix(x,m)

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

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

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

[48]:
```
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()
```



[ ]:

3 Intégration numérique

3.1 3.1 Formules de quadrature sur $[-1, 1]$

(Enlevé pour l'examen)

3.2 Intégration Numérique : Exercices

[49]:
```
# importing libraries used in this book
import numpy as np
import matplotlib.pyplot as plt
```

3.2.1 Exercice Série 6, Ex 5 : Formule de Simpson

On considère la fonction $f : [a,b] \to \mathbb{R}$ dans $C^0([a,b])$; on est intéressé à approcher l'intégrale $I(f) = \int_a^b f(x)\,dx$.

1. Ecrivez une fonction `Simpson` qui implémente la formule composite de Simpson pour l'approximation de l'intégrale ci-dessus. Utilisez la structure suivante:

```
def Simpson( a, b, N, f ) :
    # [a,b] : inteval
    # N : number of subintervals
    # f : fonction to integrate using the Simpson rule
```

2. Testez ensuite pour quels monômes $f(x) = x^d$ cette formule intègre exactement la fonction, pour $d = 0, 1, ...$ sur l'intervalle $[1, 4]$, avec $N = 1$ et ensuite $N = 10$.

3. Vérifiez numériquement pour quelques polynômes que la fonction ainsi écrite est linéaire en f pour $N = 10$

Suggestion: En utilisant une fonction `lambda` il est possible de décider le paramètre d d'un monôme à un moment ultérieur:

```
monomial = lambda x :  x**d
```

Partie 1

[50]:
```
# This function just provides the Simpson quadrature rule.
def Simpson( a, b, N, f ) :
    # [a,b] : inteval
    # N : number of subintervals
    # f : fonction to integrate using the trapezoidal rule

    M=3

    nodes = np.array([-1, 0, 1])
    weights = np.array([1./3., 4./3., 1./3.])

    # size of the subintervals
    H = (b - a) / N
    # points defining intervals
    x = np.linspace(a,b,N+1)

    Lh = 0;

    z = np.zeros(M);

    for k in range(N) :
        # left of the subinterval, also first quadrature point
        z[0] = x[k]
        # right of the subinterval, also third quadrature point
        z[2] = x[k+1]
        # mid point, , also second quadrature point
```

```
        z[1] = (x[k] + x[k+1])/2
        # can also be computed as
        z = (x[k] + x[k+1])/2 + nodes*(x[k+1] - x[k])/2

        # local quadrature:
        Jgk = weights[0] * f(z[0]) + weights[1] * f(z[1]) + weights[2] * f(z[2])
        # or as a single sum
        Jgk = sum(weights * f(z))

        Lh = Lh + Jgk

    # approximate integral
    return H/2 * Lh
```

Partie 2

$$\int_1^4 1dx = 3$$

$$\int_1^4 x^d dx = \frac{1}{d+1} x^{d+1}|_1^4 = \frac{4^{d+1}-1}{d+1}$$

[51]:
```
# Checking simpson fonction
a = 1; b = 4;

# with lambda fonctions, it is possible to determine a parameter (here d)
# at a later moment
monomial = lambda x :  x**d

# recording for which degrees the integral s exact (up to epsilon)
exactDegree = -1
epsilon = 1e-12
```

[52]:
```
N = 1
for d in range(7) :
    intsim = Simpson( a, b, N, monomial )
    intExact = (4**(d+1) - 1)/(d+1)
    print(f'Simpson on x^{d} : {intsim:.6f} - {intExact:.6f}  = {intsim-intExact:
 ↪.6e}')
    if np.abs(intsim-intExact) < epsilon :
        exactDegree = d

print(f'Simpson with N = {N} is exact up to degree {exactDegree}')
```

```
Simpson on x^0 : 3.000000 - 3.000000  = -4.440892e-16
Simpson on x^1 : 7.500000 - 7.500000  = 0.000000e+00
Simpson on x^2 : 21.000000 - 21.000000  = 0.000000e+00
Simpson on x^3 : 63.750000 - 63.750000  = 0.000000e+00
Simpson on x^4 : 206.625000 - 204.600000  = 2.025000e+00
```

```
Simpson on x^5 : 707.812500 - 682.500000  = 2.531250e+01
Simpson on x^6 : 2536.781250 - 2340.428571  = 1.963527e+02
Simpson with N = 1 is exact up to degree 3
```

[53]:
```
N = 10
exactDegree = -1

for d in range(7) :
    intsim = Simpson( a, b, N, monomial )
    intExact = (4**(d+1) - 1)/(d+1)
    print(f'Simpson on x^{d} : {intsim:.6f} - {intExact:.6f}  = {intsim-intExact:
 ↪.6e}')
    if np.abs(intsim-intExact) < epsilon :
        exactDegree = d

print(f'Simpson with N = {N} is exact up to degree {exactDegree}')
```

```
Simpson on x^0 : 3.000000 - 3.000000  = -4.440892e-16
Simpson on x^1 : 7.500000 - 7.500000  = 0.000000e+00
Simpson on x^2 : 21.000000 - 21.000000  = 0.000000e+00
Simpson on x^3 : 63.750000 - 63.750000  = -1.421085e-14
Simpson on x^4 : 204.600202 - 204.600000  = 2.025000e-04
Simpson on x^5 : 682.502531 - 682.500000  = 2.531250e-03
Simpson on x^6 : 2340.449818 - 2340.428571  = 2.124623e-02
Simpson with N = 10 is exact up to degree 3
```

Partie 3

$$\int_1^4 1dx = 3$$

$$\int_1^4 x^d dx = \frac{1}{d+1} x^{d+1}|_1^4 = \frac{4^{d+1}-1}{d+1}$$

[54]:
```
maxD = 5;
N = 10

p = lambda x :  np.polyval(coefs,x)

# pre-computing integrls of monomials up to degree maxD
intMono = np.zeros(maxD+1)
for d in range(maxD+1) :
    intMono[d] = Simpson( a, b, N, monomial )
```

[55]:
```
# generating random coefficients
coefs = np.random.rand(maxD+1)

# evaluating Simpson on the polynomial :
intPoly = Simpson(a,b,N, p)
```

```
# computing integral by linearity. Remeber that in polyval, the order of the␣
 ↪coefficients is opposite !
intSum = 0
for d in range(maxD+1) :
    intSum = intSum + coefs[maxD-d]*intMono[d]

print(f'Simpson on a_{d} + a_{d-1} x + ... + a_0x^{d} by linearity : \n'
      f'\t {intPoly:.6f} - {intSum:.6f}  = {intPoly-intSum:.6e}')
```

```
Simpson on a_5 + a_4 x + ... + a_0x^5 by linearity :
        262.343120 - 262.343120  = 0.000000e+00
```

[ ]:

3.2.2 Exercice Série 6, Ex 6 : Degré d'exactitude d'une formule de quadrature

On considère la fonction $f : [a,b] \to \mathbb{R}$ dans $C^0([a,b])$; on est intéressé à approcher l'intégrale $I(f) = \int_a^b f(x)\,dx$.

Plus précisément on prend $f(x) = \sin(\frac{7}{2}x) + e^x - 1$ avec $a = 0$ et $b = 1$ ($f \in C^\infty([a,b])$) et on peut calculer $I(f) = \frac{2}{7}(1 - \cos(\frac{7}{2})) + e - 2$.

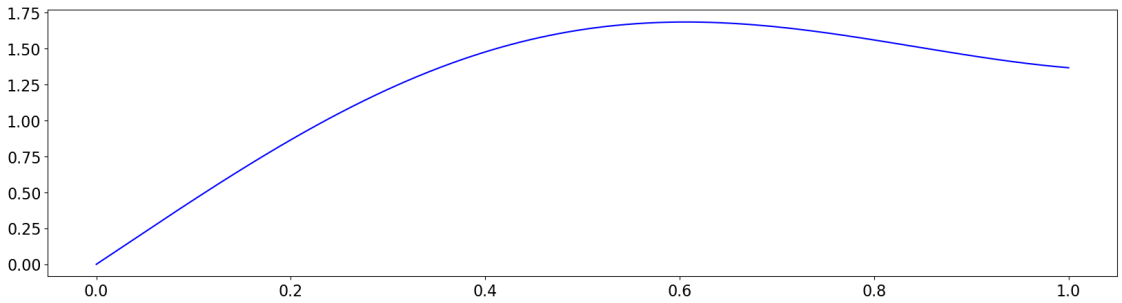
1. Calculez une approximation de l'intégrale en utilisant les formules du rectangle, du trapèze et de Simpson *simples*, c-à-d avec un seul intervalle.
2. Calculez une approximation de l'intégrale en utilisant les fonctions `Midpoint`, `Trapezoidal` et `Simpson` (déjà codées). avec $N = 10$ sous-intervalles de même taille. On notera les valeurs approchées de l'intégrale $I^c_{mp}(f)$, $I^c_t(f)$, and $I^c_s(f)$, respectivement.
3. Répétez le point 2. avec $N = 2^k$ pour $k = 2, \ldots, 7$ et calculez les erreurs $E^c_{mp}(f) := |I(f) - I^c_{mp}(f)|$, $E^c_t(f) := |I(f) - I^c_t(f)|$, et $E^c_s(f) := |I(f) - I^c_s(f)|$. Dessinez les erreurs en fonction de $H = (b-a)/N$ sur une échelle logarithmique sur les deux axes. Quel est l'ordre de convergence de ces méthodes ? Est-ce en accord avec la théorie ? Motivez votre réponse.
4. On prend maintenant $f(x) = x^d$, $a = 0$ et $b = 1$, avec $d \in \mathbb{N}$. L'intégrale de f vaut $I(f) = 1/(d+1)$. Vérifiez numériquement les degrés d'exactitude de chacune des formules de quadrature du point 1. Pour cela, il faut choisir plusieurs valeurs de $d = 0, 1, 2, \ldots$. Motivez votre réponse.

[56]:
```
import matplotlib.pyplot as plt
import numpy as np

from IntegrationLib import *
```

Partie 1

[57]:
```
f = lambda x :  np.sin(7/2*x) + np.exp(x) - 1

a = 0; b = 1
Iexact = 2/7*(1-np.cos(7/2) ) + np.exp(1) - 2
```

```
x = np.linspace(a,b,1000)
y=f(x)

plt.plot(x, y, 'b')
plt.show()
```



[58]:
```
intMD   = (b-a) * f( (a+b)/2 )
intTrap = (b-a) * ( f(a)+f(b) )/2
intSimp = (b-a)/6 * ( f(a) + 4*f((a+b)/2) + f(b) )

print(f'exact \t rect \t\t trap \t\t Simpson')
print(f'{Iexact:.4f} \t {intMD:.4f} \t {intTrap:.4f} \t {intSimp:.4f}')
```

```
exact    rect            trap            Simpson
1.2716   1.6327          0.6837          1.3164
```

Partie 2

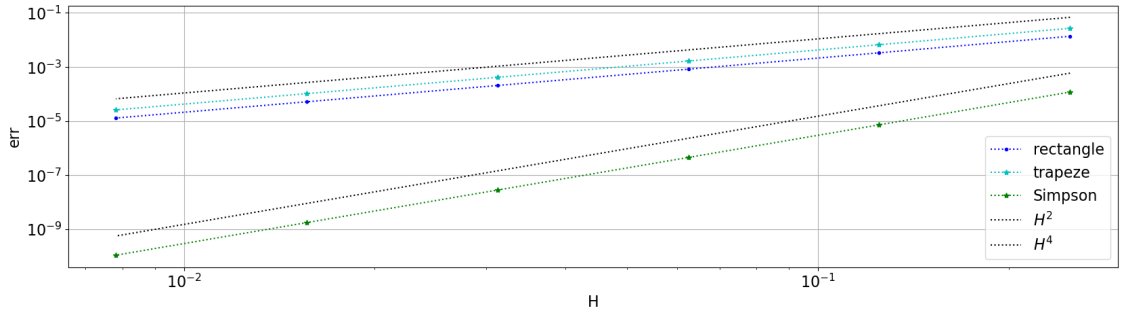
[59]:
```
N = 10
intMD   = Midpoint(a,b,N,f)
intTrap = Trapezoidal(a,b,N,f)
intSimp = Simpson(a,b,N,f)

print(f'exact \t rect \t\t trap \t\t Simpson')
print(f'{Iexact:.4f} \t {intMD:.4f} \t {intTrap:.4f} \t {intSimp:.4f}')
```

```
exact    rect            trap            Simpson
1.2716   1.2737          1.2673          1.2716
```

Partie 3

[60]:
```
errmp = []
errtrap = []
errSim = []

N = 2**np.linspace(2,7,6).astype(int)
```

```
for n in N :
    errmp.append( np.abs( Midpoint( a, b, n, f) - Iexact ) )
    errtrap.append( np.abs( Trapezoidal( a, b, n, f) - Iexact ) )
    errSim.append( np.abs( Simpson( a, b, n, f) - Iexact ) )

H = (b-a)/N
plt.plot(H, errmp, 'b:.',  H, errtrap, 'c:*', H, errSim, 'g:*')

plt.plot(H, H**2 * (errmp[0]/H[0]**2)*5, 'k:', H, H**4 * (errSim[0]/H[0]**4)*5,␣
↪'k:')

plt.legend(['rectangle', 'trapeze', 'Simpson', '$H^2$', '$H^4$'])
plt.xlabel('H'); plt.ylabel('err');

plt.xscale('log')
plt.yscale('log')
plt.grid(True)
```



[Pardon d'écrire en anglais..]

We recall that the errors $E^c_{mp}(f) = |e^c_{mp}(f)|$, $E^c_t(f) = |e^c_t(f)|$, and $E^c_s(f) = |e^c_s(f)|$ read, for a sufficiently regular function $f(x)$:

$$e^c_{mp}(f) = I(f) - I^c_{mp}(f) = \frac{b-a}{24} H^2 f''(\xi), \qquad \text{for some } \xi \in [a,b], \qquad \text{if } f \in C^2([a,b]),$$

$$e^c_t(f) = I(f) - I^c_t(f) = -\frac{b-a}{12} H^2 f''(\eta), \qquad \text{for some } \eta \in [a,b], \qquad \text{if } f \in C^2([a,b]),$$

$$e^c_s(f) = I(f) - I^c_s(f) = -\frac{b-a}{16 \cdot 180} H^4 f^{(4)}(\zeta), \qquad \text{for some } \zeta \in [a,b], \qquad \text{if } f \in C^4([a,b]).$$

Since in this case $f(x) \in C^\infty([a,b])$, we expect the orders of accuracy (convergence orders of the errors) to be equal to 2, 2, and 4 for the composite midpoint, trapezoidal, and Simpson quadrature formulas, respectively. This is confirmed by the figure above, where we can observe that the plots of the errors $E_{mp}(f)$ and $E_t(f)$ vs. H are, in log-log scale, parallel to the line representing the curve (H, H^2), thus indicating the order of accuracy (convergence order) 2 for the composite midpoint and trapezoidal quadrature formulas. Similarly, the plot of the error $E_s(f)$ is, in log-log scale, parallel to

the line representing the curve (H, H^4), from which we deduce the order of accuracy (convergence order) 4 for the composite Simpson quadrature formula.

We notice that the orders of accuracy could be deduced by computing the errors for two different values of H, say H_1 and H_2; for example for a generic composite quadrature formula we obtain the corresponding errors $E^c_{H_1}(f)$ and $E^c_{H_2}(f)$. If we assume that the error $E^c_H(f)$ can be expressed as $E^c_H(f) = C\,H^\alpha$, with $\alpha > 0$ and C a positive constant independent of H and α, we can estimate the order of accuracy (convergence order) of the quadrature formula as $\alpha = \log_\beta \left(E^c_{H_2}(f)/E^c_{H_1}(f)\right) / \log_\beta (H_2/H_1)$, for any $\beta > 1$ and H_1 and H_2 "sufficiently small''. For the composite midpoint, trapezoidal, and Simpson quadrature formulas, we use the following commands, for which the results ($\alpha = 2.01$, 2.01, and 4.01, respectively) confirm the expected orders of accuracy (convergence orders).

[61]:
```
slopeMP = ( np.log(errmp[-1]) - np.log(errmp[0]) ) / ( np.log(H[-1]) - np.
 ↪log(H[0]) )
slopeTrap = ( np.log(errtrap[-1]) - np.log(errtrap[0]) ) / ( np.log(H[-1]) - np.
 ↪log(H[0]) )
slopeSim = ( np.log(errSim[-1]) - np.log(errSim[0]) ) / ( np.log(H[-1]) - np.
 ↪log(H[0]) )

print(f'La convergence numérique est environ de ')
print(f'Rectangle : {slopeMP:.2f}')
print(f'Trapeze : {slopeTrap:.2f}')
print(f'Simpson : {slopeSim:.2f}')
```

```
La convergence numérique est environ de
Rectangle : 2.01
Trapeze : 2.01
Simpson : 4.01
```

Partie 4 The function $f(x) = x^d$ is a polynomial of degree d for $d \in \mathbb{N}$. The simple midpoint, trapezoidal, and Simpson quadrature formulas possesses degree of exactness equal to 1, 1, and 3, respectively.

[62]:
```
N = 1
# with lambda fonctions, it is possible to determine a parameter (here d)
# at a later moment
monomial = lambda x :  x**d

# recording for which degrees the integral s exact (up to epsilon)
exactDegree = -1
epsilon = 1e-12

for d in range(7) :
    intsim = Midpoint( a, b, N, monomial )
    intExact = 1/(d+1)
    print(f'Midpoint on x^{d} : {intsim:.6f} - {intExact:.6f}  =␣
 ↪{intsim-intExact:.6e}')
```

```
    if np.abs(intsim-intExact) < epsilon :
        exactDegree = d

print(f'Midpoint is exact up to degree {exactDegree}')
```

```
Midpoint on x^0 : 1.000000 - 1.000000  = 0.000000e+00
Midpoint on x^1 : 0.500000 - 0.500000  = 0.000000e+00
Midpoint on x^2 : 0.250000 - 0.333333  = -8.333333e-02
Midpoint on x^3 : 0.125000 - 0.250000  = -1.250000e-01
Midpoint on x^4 : 0.062500 - 0.200000  = -1.375000e-01
Midpoint on x^5 : 0.031250 - 0.166667  = -1.354167e-01
Midpoint on x^6 : 0.015625 - 0.142857  = -1.272321e-01
Midpoint is exact up to degree 1
```

[63]:
```
for d in range(7) :
    intsim = Trapezoidal( a, b, N, monomial )
    intExact = 1/(d+1)
    print(f'Trapezoidal on x^{d} : {intsim:.6f} - {intExact:.6f}  =␣
 ↪{intsim-intExact:.6e}')
    if np.abs(intsim-intExact) < epsilon :
        exactDegree = d

print(f'Trapezoidal is exact up to degree {exactDegree}')
```

```
Trapezoidal on x^0 : 1.000000 - 1.000000  = 0.000000e+00
Trapezoidal on x^1 : 0.500000 - 0.500000  = 0.000000e+00
Trapezoidal on x^2 : 0.500000 - 0.333333  = 1.666667e-01
Trapezoidal on x^3 : 0.500000 - 0.250000  = 2.500000e-01
Trapezoidal on x^4 : 0.500000 - 0.200000  = 3.000000e-01
Trapezoidal on x^5 : 0.500000 - 0.166667  = 3.333333e-01
Trapezoidal on x^6 : 0.500000 - 0.142857  = 3.571429e-01
Trapezoidal is exact up to degree 1
```

[64]:
```
for d in range(7) :
    intsim = Simpson( a, b, N, monomial )
    intExact = 1/(d+1)
    print(f'Simpson on x^{d} : {intsim:.6f} - {intExact:.6f}  = {intsim-intExact:
 ↪.6e}')
    if np.abs(intsim-intExact) < epsilon :
        exactDegree = d

print(f'Simpson is exact up to degree {exactDegree}')
```

```
Simpson on x^0 : 1.000000 - 1.000000  = 0.000000e+00
Simpson on x^1 : 0.500000 - 0.500000  = 0.000000e+00
Simpson on x^2 : 0.333333 - 0.333333  = 0.000000e+00
Simpson on x^3 : 0.250000 - 0.250000  = 0.000000e+00
Simpson on x^4 : 0.208333 - 0.200000  = 8.333333e-03
Simpson on x^5 : 0.187500 - 0.166667  = 2.083333e-02
```

```
Simpson on x^6 : 0.177083 - 0.142857  = 3.422619e-02
Simpson is exact up to degree 3
```

De ces simulations nous vérifions que les monômes $f(x) = x^d$ sont intégrées exactement pour les degrées $d = 0$ et $d = 1$ dans le cas des formules du rectangle et du trapèze, et pour $d = 0, 1, 2, 3$ dans le cas de la formule de Simpson.

[ ]:

3.2.3 Exercice Série 6, Ex 7 : Convergence pour fonction non-lisse

[65]:
```
import matplotlib.pyplot as plt
import numpy as np

from IntegrationLib import Midpoint
from IntegrationLib import Trapezoidal
from IntegrationLib import Simpson
```

On considère, sur l'intervalle $[-1, 1]$, la fonction f suivante:

$$f(x) = \begin{cases} e^x & \text{si } x \leq 0 \\ 1 & \text{si } x > 0 \end{cases}$$

On peut définir une telle fonction en utilisant la commande

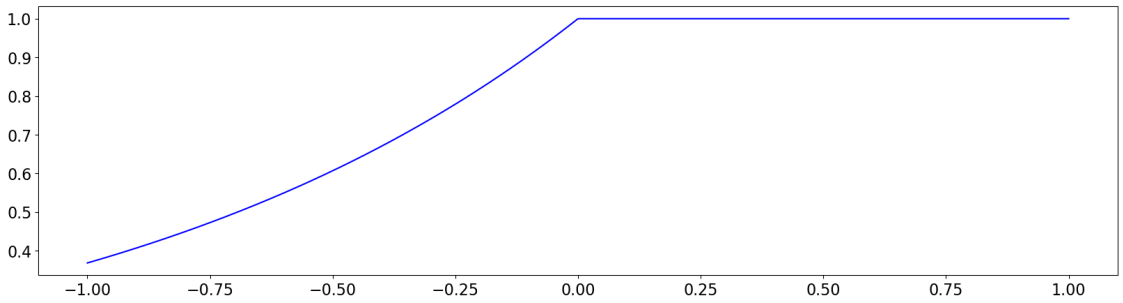
```
f = lambda x :  np.exp(x)*(x<=0) + (1)*(x>0)
```

1. Utilisez 1000 points équirépartis dans l'intervalle $[-1, 1]$ pour afficher la fonction f (utilisez la commande `axis` pour recadrer l'image).
2. On s'intéresse à présent à l'intégrale $I = \int_{-1}^{1} f(x)\ dx$. On peut calculer la valeur de I analytiquement et on trouve $I = 2 - \frac{1}{e} \cong 1.6321$. Calculez des valeurs approchées de I en considérant les formules du point milieu, du trapèze et de Simpson avec $N = 1, 9, 99, 999$ où N est le nombre de sous-intervalles de formules composites. Utilisez les fonctions `midpoint`, `trapezoidal` et `simpson` (déjà codées).
3. Reportez les erreurs calculées au point (b) dans un graphe montrant l'erreur en fonction de H avec des échelles logarithmiques.
4. Estimez, à partir des graphes obtenus au point précédent, l'ordre de chacune des méthodes. Comparez-les avec les ordres donnés au cours. Y a-t-il des différences? Pourquoi? (Regardez les dérivées de f).
5. Pourquoi obtient-on de bien meilleurs résultats pour la méthode de Simpson avec un nombre pair d'intervalles qu'avec un nombre impair (essayez avec $N = 99$ et ensuite $N = 100$)?
6. Refaites l'exercice avec $N = 2, 10, 100, 1000$.

Partie 1

[66]:
```
f = lambda x :  np.exp(x)*(x<=0) + (1)*(x>0)

a = -1; b = 1
```

```
x = np.linspace(a,b,1000)
y=f(x)

plt.plot(x, y, 'b')
plt.show()
```



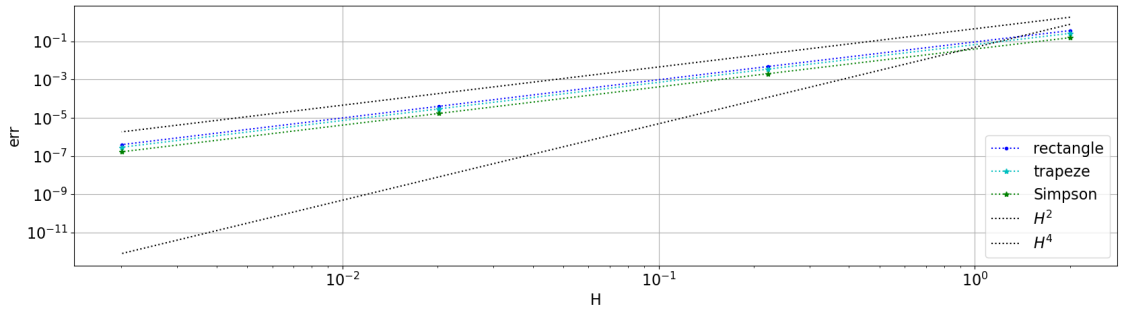
Partie 2 On calcule tout d'abord la valeur exacte de l'intégrale, puis on calcule les erreurs, que l'on stocke dans des vecteurs:

[67]:
```
Iexact = 2 - np.exp(-1)

errmp = []
errtrap = []
errSim = []

N = [1,9,99,999]
#N = [2,10,100,1000]

for i in range(4) :
    errmp.append( np.abs( Midpoint( a, b, N[i], f) - Iexact ) )
    errtrap.append( np.abs( Trapezoidal( a, b, N[i], f) - Iexact ) )
    errSim.append( np.abs( Simpson( a, b, N[i], f) - Iexact ) )
```

Partie 3

[68]:
```
H = 2./np.array(N)
plt.plot(H, errmp, 'b:.',  H, errtrap, 'c:*', H, errSim, 'g:*')

plt.plot(H, H**2 * (errmp[0]/H[0]**2)*5, 'k:', H, H**4 * (errSim[0]/H[0]**4)*5,␣
 ↪'k:')

plt.legend(['rectangle', 'trapeze', 'Simpson', '$H^2$', '$H^4$'])
plt.xlabel('H'); plt.ylabel('err');

plt.xscale('log')
```

```
plt.yscale('log')
plt.grid(True)
plt.show()
```



Les graphes pour toutes les méthodes sont des droites. On remarque que lorsque H est divisé par 10, les erreurs sont environ divisées par 100. On peut donc supposer que les ordres sont 2 par rapport à H. Pour le confirmer, on peut ajouter sur le graphe la pente représentant l'ordre 2 et vérifier qu'elle est parallèle aux droites d'erreur.

Partie 4

[69]:
```
slopeMP = ( np.log(errmp[-1]) - np.log(errmp[0]) ) / ( np.log(H[-1]) - np.
 ↪log(H[0]) )
slopeTrap = ( np.log(errtrap[-1]) - np.log(errtrap[0]) ) / ( np.log(H[-1]) - np.
 ↪log(H[0]) )
slopeSim = ( np.log(errSim[-1]) - np.log(errSim[0]) ) / ( np.log(H[-1]) - np.
 ↪log(H[0]) )

print(f'La convergence numérique est environ de ')
print(f'Rectangle : {slopeMP:.2f}')
print(f'Trapeze : {slopeTrap:.2f}')
print(f'Simpson : {slopeSim:.2f}')
```

```
La convergence numérique est environ de
Rectangle : 1.99
Trapeze : 1.99
Simpson : 1.99
```

Les graphiques sont parallèles à la droite H^2 et donc l'ordre est 2 pour toutes les méthodes. Pour la méthode du point milieu et du trapèze, c'est l'ordre auquel on s'attend d'après la théorie. Par contre, pour Simpson, on pouvait s'attendre à un ordre 4. Le problème vient du fait que la fonction n'est pas très régulière, puisque $f \in C^0([-1,1])$ mais $f \notin C^1([-1,1])$ puisque sa dérivée n'est pas continue en $x = 0$. Dans la théorie, on demande $f \in C^4([-1,1])$ pour assurer l'ordre 4. On obtient tout de même l'ordre 2 car, mis à part en $x = 0$, la fonction f est très régulière.

Partie 5 Si on regarde l'erreur pour la méthode de Simpson avec $N = 99$ et $N = 100$ sous-intervalles

[70]:
```
print (np.abs( Simpson( a, b, 99, f) - Iexact ) )
print (np.abs( Simpson( a, b, 100, f) - Iexact ) )
```

```
1.7004959483424287e-05
3.5117464491918327e-11
```

On obtient des valeurs d'environ $1.70\ 10^{-5}$ et $3.51\ 10^{-11}$. Les erreurs sont très différentes ! Voilà pourquoi:

- si N est pair, le point $x = 0$ est un point x_i (pour $i = N/2$) exactement entre deux sous-intervalles. La fonction f est régulière à droite et à gauche, en particulier, on retrouve l'ordre 4 par rapport à H de chaque côté.
- si N est impair, le point $x = 0$ est au milieu d'un sous-intervalle. La fonction f n'est pas régulière dans cet intervalle, ce qui donnt un ordre de convergence plus petit.

Partie 6 Il faut relancer les parties 2 et 3 avec `N= [2,10,100,1000]` à la place de `N = [1,9,99,999]`

4 Résolution de systèmes linéaires

4.1 Méthodes Directes

D'abord quelques exemples d'utilisations de la factorisation LU et de Choleski, ensuite on passe aux methodes itéreatives

4.1.1 Exercice 1 série 8

On considère le système linéaire $A\mathbf{x} = \mathbf{b}$ où :

$$A = \begin{pmatrix} 3 & 6 & 7 \\ 1 & 1 & 4 \\ 2 & 4 & 8 \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} 4 \\ 5 \\ 6 \end{pmatrix}.$$

1. Calculez la factorisation LU de la matrice A avec Python à l'aide du code ci-dessous.
2. Résolvez le système linéaire $A\mathbf{x} = \mathbf{b}$ en utilisant la factorisation trouvée au point précédent (Ne plus utiliser Python. Mais ici on va le faire)
3. Calculez le déterminant de la matrice A en utilisant sa factorisation LU.

[71]:
```
# importing libraries used in this notebook
import numpy as np
import matplotlib.pyplot as plt

# scipy.linalg.lu : LU decomposition
from scipy.linalg import lu
# scipy.linalg.cholesky : Cholesky decomposition
from scipy.linalg import cholesky
```

```
import pprint

np.set_printoptions(precision=4, suppress=True, linewidth=120)
```

Partie 1

[72]:
```
A = np.array([[3, 6, 7],
              [1, 1, 4],
              [2, 4, 8] ])

# LU factorisation with pivoting
P, L, U = lu(A)

print("A = P L U")
pprint.pprint(P.dot(L.dot(U)) )

print("P:")
pprint.pprint(P)

print ("L:")
pprint.pprint(L)

print ("U:")
pprint.pprint(U)
```

```
A = P L U
array([[3., 6., 7.],
       [1., 1., 4.],
       [2., 4., 8.]])
P:
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
L:
array([[ 1.    ,  0.    ,  0.    ],
       [ 0.3333,  1.    ,  0.    ],
       [ 0.6667, -0.    ,  1.    ]])
U:
array([[ 3.    ,  6.    ,  7.    ],
       [ 0.    , -1.    ,  1.6667],
       [ 0.    ,  0.    ,  3.3333]])
```

Partie 2 Si P est l'identité, $A = LU$.

1. résolvez pour $\mathbf{y}$ tel que $L\mathbf{y} = \mathbf{b}$ par substitution progressive
2. résolvez pour $\mathbf{x}$ tel que $U\mathbf{x} = \mathbf{y}$ par substitution retrograde

Si P n'est pas l'identité, $A = PLU$.

Attention, une matrice de permutation est une matrice orthogonale car les colonnes sont orthonormées. Donc $P^{-1} = P^T$. Du coup : $P^T A = LU$ et

$$A\mathbf{x} = \mathbf{b} \Leftrightarrow P^T A\mathbf{x} = P^T\mathbf{b} \Leftrightarrow LU\mathbf{x} = P^T\mathbf{b}.$$

Alors il faut modifier les calculs précédants comme suit:

1. résolvez pour $\mathbf{y}$ tel que $L\mathbf{y} = P^T\mathbf{b}$ par substitution progressive
2. résolvez pour $\mathbf{x}$ tel que $U\mathbf{x} = \mathbf{y}$ par substitution retrograde

[73]:
```python
def subst_progressive(L, b):
    """
    substitution progressive: résout y, L*y = b
    Input:
    - L: matrice carrée nxn, triangulaire inférieure
    - b: vector de dimension n
    Output:
    - y: vector de dimension n
    """

    # Initialisation de la solution
    y = np.zeros(L.shape[1])

    # La première ligne de Ly = b est L_{11} y_1 = b_1
    # Ensuite pour la ligne k on connait y_1, ..., y_{k-1} et elle s'écrit
    # L_{kk} y_k = b_k - ( L_{k1} y_1 + ... L_{kk-1} y_{k-1} )
    for k in range(L.shape[0]):
        # sum_k est ( L_{k1} y_1 + ... L_{kk-1} y_{k-1} )
        sum_k = 0
        for j in range(k):
            sum_k += L[k,j]*y[j]
        y[k] = 1/L[k,k]*(b[k]-sum_k)
    return y

def subst_retrograde(U, y):
    """
    substitution retrograde: résout pour x, U*x = y
    Input:
    - U: matrice carrée nxn, triangulaire supérieure
    - y: vector de dimension n
    Output:
    - x: vector de dimension n
    """

    # Initialisation de la solution
    x = np.zeros(U.shape[1])

    # La dernière ligne de Yx = y est U_{nn} x_n = y_n
    # Ensuite pour la ligne k on connait x_n, ..., x_{k+1} et elle s'écrit
```

```
    # U_{kk} x_k = y_k - ( U_{kk+1} x_{k+1} + ... U_{kn} y_{n} )
    for k in reversed(range(U.shape[0])):
        # sum_k est ( U_{kk+1} x_{k+1} + ... U_{kn} y_{n} )
        sum_k = 0
        for j in range(k+1, U.shape[0]):
            sum_k += U[k,j]*x[j]
        x[k] = 1/U[k,k]*(y[k]-sum_k)
    return x
```

[74]:
```
b = np.array([4, 5, 6])
Ptb = P.T.dot(b)

y = subst_progressive(L, Ptb)
print("y =", y)

x = subst_retrograde(U, y)
print("x =", x)

# check the residual of the equation
print("residual =",b - A.dot(x))
```

```
y = [4.     3.6667 3.3333]
x = [ 3. -2.  1.]
residual = [0. 0. 0.]
```

Partie 3

$$\det A = \det P \det L \det U$$

[75]:
```
# scipy.linalg.det : determinant
from scipy.linalg import det

det(P)*det(L)*det(U)
```

[75]:
-10.0

4.1.2 Critère de Sylvester

Les mineurs principaux d'une matrice $A \in \mathbb{R}^{n\times n}$ sont les déterminants des matrices $A_p = (a_{i,j})_{1\leq i,j\leq p}$, $p = 1, ..., n$.

Critère de Sylvester: une matrice symétrique $A \in \mathbb{R}^{n\times n}$ est définie positive si et seulement si les mineurs principaux de A sont tous positifs.

4.1.3 Exercice 2 série 8

On considère le système linéaire $A\mathbf{x} = \mathbf{b}$ où

$$A = \begin{pmatrix} \varepsilon & 1 & 2 \\ 1 & 3 & 1 \\ 2 & 1 & 3 \end{pmatrix}.$$

1. Déterminez pour quelles valeurs du paramètre réel $\varepsilon \in \mathbb{R}$, la matrice A est symétrique définie positive.
2. Soit maintenant $\varepsilon = 0$. On veut résoudre le système $A\mathbf{x} = \mathbf{b}$ par une méthode directe; quelle factorisation de la matrice A envisageriez-vous? Justifiez votre réponse.
3. En considérant $\varepsilon = 2$, vérifier que dans ce cas la matrice A est définie positive et calculer sa factorisation de Cholesky $A = LL^T$.
4. En supposant que $\mathbf{b} = (1, 1, 1)^T$, résolvez le système linéaire $A\mathbf{x} = \mathbf{b}$ en utilisant la factorisation de Cholesky calculée au point c).

Référence Python pour la factorisation de Cholesky : scipy.linalg.cholesky

Partie 1 En appliquant le critère de Sylvester, il suffit d'imposer

$$\begin{cases} \varepsilon > 0, \\ \det \begin{pmatrix} \varepsilon & 1 \\ 1 & 3 \end{pmatrix} = 3\varepsilon - 1 > 0, \\ \det A = 8\varepsilon - 11 > 0, \end{cases} \qquad \Rightarrow \quad \varepsilon > \frac{11}{8}.$$

Partie 2 Si $\varepsilon = 0$ la matrice A est sym'etrique, mais elle n'est pas d'efinie positive; donc on ne peut pas calculer la factorisation de Cholesky. On utilisera la m'ethode d' 'elimination de Gauss avec changement de pivot, puisque $a_{11} = 0$; par exemple, on peut consid'erer la matrice de permutation P par lignes:

$$P = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

On peut facilement voir que $A = PLU$ avec

$$L = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 2 & -5 & 1 \end{pmatrix} \qquad \text{et} \qquad U = \begin{pmatrix} 1 & 3 & 1 \\ 0 & 1 & 2 \\ 0 & 0 & 11 \end{pmatrix}.$$

Partie 3 Si $\varepsilon = 2$, la matrice A est symétrique définie positive. Ici on va utiliser $A = LL^T$. Les éléments de la matrice L de la factorisation de Cholesky de A sont:

$$\begin{aligned} l_{11} &= \sqrt{a_{11}} = \sqrt{2} \\ l_{21} &= \frac{1}{l_{11}} \cdot a_{21} = \frac{1}{\sqrt{2}} \\ l_{22} &= \sqrt{a_{22} - l_{21}^2} = \sqrt{\frac{5}{2}} \\ l_{31} &= \frac{1}{l_{11}} \cdot a_{31} = \sqrt{2} \\ l_{32} &= \frac{1}{l_{22}} \cdot (a_{32} - l_{31} l_{21}) = 0 \\ l_{33} &= \sqrt{a_{33} - (l_{31}^2 + l_{32}^2)} = 1 \end{aligned}$$

c'est-à-dire:

$$L = \begin{pmatrix} \sqrt{2} & 0 & 0 \\ \frac{1}{\sqrt{2}} & \sqrt{\frac{5}{2}} & 0 \\ \sqrt{2} & 0 & 1 \end{pmatrix}.$$

[76]:
```
# scipy.linalg.eigh : eigenvalues for symmetric matric
from scipy.linalg import eigh
# scipy.linalg.eig : eigenvalues of a matrix
from scipy.linalg import eig
```

[77]:
```
epsilon = 2
A = np.array([[epsilon, 1, 2],
              [1, 3, 1],
              [2, 1, 3] ])

# Is A symmetric ?
print(f'max(abs(A-AT)) = {np.max(np.abs(A-A.T))} ')

# What are the eigenvlues of $A$ ? (usually, use eig, but here A is symmetric,␣
 ↪we can use eigh )
lk, v = eigh(A)
print(f'The eigenvalues of A are {lk}')

# Cholesky factorisation: lower : return lower-triangular matrix, A = L L^T
L = cholesky(A, lower=True)
print(f"\n A = L^T L = {L.dot(L.T)}\n")

print (f"L = {L}")
```

```
max(abs(A-AT)) = 0
The eigenvalues of A are [0.4242 2.1873 5.3885]

 A = L^T L = [[2. 1. 2.]
```

```
 [1. 3. 1.]
 [2. 1. 3.]]

L = [[1.4142 0.     0.    ]
 [0.7071 1.5811 0.    ]
 [1.4142 0.     1.    ]]
```

Partie 4

[78]:
```
b = np.array([1,1, 1])

y = np.linalg.solve(L,b)
x = np.linalg.solve(L.T,y)

print(x)

# check the residual of the equation
print(b - A.dot(x))
```

```
[0.4 0.2 0. ]
[ 0.  0. -0.]
```

4.1.4 Problèmes de précision (Exercice 3 série 8)

Les erreurs d'arrondis peuvent causer des différences importantes entre la solution calculée par la méthode d'élimination de Gauss (MEG) et la solution exacte. Cela arrive si le *conditionnement* de la matrice du système est très grand.

La matrice de Hilbert de taille $n \times n$ est une matrice symétrique définie par

$$a_{ij} = \frac{1}{i+j-1}, \qquad i,j = 1,\dots,n$$

On peut construire une matrice de Hilbert de taille n quelconque en utilisant la commande `A = scipy.linalg.hilbert(n)`. Par exemple, pour $n = 4$, on a:

$$A = \begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} \\ \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} \end{bmatrix}$$

On considère les systèmes linéaires $A_n \mathbf{x}_n = \mathbf{b}_n$ ou A_n est la matrice de Hilbert de taille n avec $n = 4, 6, 8, 10, 12, 14, \dots, 20$ tandis que $\mathbf{b}_n$ est choisi de sorte que la solution exacte soit $\mathbf{x}_n = (1, 1, \cdots, 1)^T$.

1. Pour chaque n, calculez le conditionnement de la matrice
2. Résolvez le système linéaire par la factorisation LU et notez $\vec{x}_n^{LU}$ la solution calculée.
3. Dessinez le graphique avec le conditionnement obtenu ainsi que l'erreur rélative $\|\mathbf{x}_n - \mathbf{x}_n^{LU}\|/\|\mathbf{x}_n\|$ (où $\|\cdot\|$ est la norme euclidienne d'un vecteur, $\|\mathbf{x}\| = \sqrt{\mathbf{x}^T \cdot \mathbf{x}}$). Utilisez une échelle logarithmique pour l'axe y.

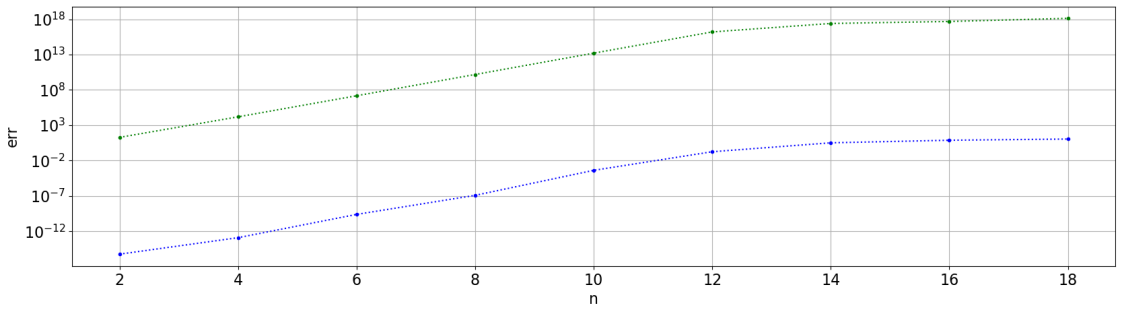
4. Sur le même graphique, reportez le conditionnement de la matrice A, `np.linalg.cond(A)`

Répétez la même chose avec la factorisation de Cholesky `L = cholesky(A, lower=True)` pour $n = 4, 6, 8, 10, 12$. Que se passe-t-il si $n = 14$?

[79]:
```
from scipy.linalg import hilbert
```

[80]:
```
Nrange = range(2,20,2)
err = []
cond = []

for n in Nrange :
    A = hilbert(n)
    P, L, U = lu(A)

    x = np.ones([n,1])

    b = A.dot(x)

    y = np.linalg.solve(L,P.T.dot(b))
    xLU = np.linalg.solve(U,y)

    err.append( np.linalg.norm(x-xLU) / np.linalg.norm(x) )
    cond.append( np.linalg.cond(A) )
```

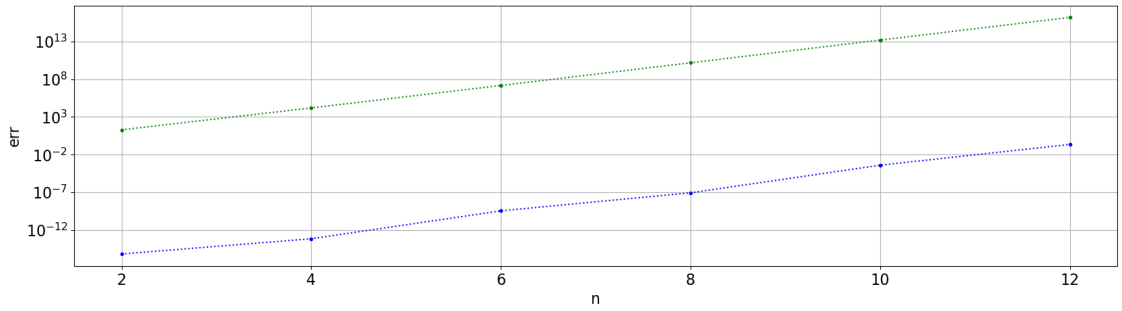
[81]:
```
plt.plot(Nrange, err, 'b:.',Nrange, cond, 'g:.')

plt.xlabel('n'); plt.ylabel('err');
# plt.xscale('log')
plt.yscale('log')
plt.grid(True)
plt.show()
```



[82]:
```
Nrange = range(2,13,2)
err = []
cond = []
```

```
for n in Nrange :
    A = hilbert(n)
    L = cholesky(A, lower=True)

    x = np.ones([n,1])

    b = A.dot(x)

    y = np.linalg.solve(L,b)
    xCho = np.linalg.solve(L.T,y)

    err.append( np.linalg.norm(x-xCho) / np.linalg.norm(x) )
    cond.append( np.linalg.cond(A) )
```

[83]:
```
plt.plot(Nrange, err, 'b:.',Nrange, cond, 'g:.')

plt.xlabel('n'); plt.ylabel('err');
# plt.xscale('log')
plt.yscale('log')
plt.grid(True)
plt.show()
```



[84]:
```
n = 14
A = hilbert(n)
# L = cholesky(A, lower=True)
lk, v = eigh(A)

print(f'lk[0] = {lk[0]:0.5e} , the matrix is not positive definite. Let''s␣
 ↪verify with the first eigenvector')
```

```
lk[0] = -9.13148e-18 , the matrix is not positive definite. Lets verify with the
first eigenvector
```

La première valeur propre est négative, cela signifie que $(\mathbf{v}, A\mathbf{v}) = (\mathbf{v}, \lambda\mathbf{v}) = \lambda(\mathbf{v}, \mathbf{v}) = \lambda\|\mathbf{v}\|^2 < 0$, où $\mathbf{v}$ est le vecteur propre associé à cette valeur propre λ négative.

Vérifions:

[85]:
```
print ( A.dot(v[0]).T.dot(v[0]) )
```

```
0.007555533559357492
```

Pourquoi ? En vérité, le calcul des valeurs et vecteurs propres de A a aussi un problème d'arrondis à cause du conditionnement. On peut en effet vérifier que le rapport entre les composantes de $\mathbf{v}$ et $A\mathbf{v}$ n'est ni égale à λ, ni à une autre constante :

[86]:
```
print(v[0]/A.dot(v[0]) )
```

```
[ -0.      -0.      -0.       0.0001  -0.0008   0.0052  -0.0292   0.1421
-0.6044  -2.2364   7.0956  18.675  -37.1359
  40.0708]
```

[ ]:
```

```

[ ]:
```

```

4.2 Méthodes itératives

[87]:
```
# importing libraries used in this notebook
import numpy as np
import matplotlib.pyplot as plt


# scipy.linalg.eig : eigenvalues of a  matrix
from scipy.linalg import eig
# scipy.linalg.lu : LU decomposition
from scipy.linalg import lu
# scipy.linalg.cholesky : Cholesky decomposition
from scipy.linalg import cholesky
# scipy.linalg.hilbert : Hilbert matrix
from scipy.linalg import hilbert

np.set_printoptions(precision=4, suppress=True, linewidth=120)
```

4.3 Méthode de Richardson

A et $\mathbf{b}$ donnés; on cherche à approximer la solution $\mathbf{x}$ de $A\mathbf{x} = \mathbf{b}$.

Soit $\mathbf{x}^{(0)}$ donné, $\mathbf{r}^{(0)} = \mathbf{b} - A\mathbf{x}^{(0)}$ pour $k = 0, 1, 2, \ldots$:

$$\begin{aligned} \text{trouvez } \mathbf{z}^{(k)} \text{ tel que} \quad & P\mathbf{z}^{(k)} = \mathbf{r}^{(k)} \\ \text{choisissez } \alpha_k \quad & \\ & \mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k \mathbf{z}^{(k)} \\ & \mathbf{r}^{(k+1)} = \mathbf{r}^{(k)} - \alpha_k A\mathbf{z}^{(k)}. \end{aligned}$$

4.3.1 Exemple 1

On considère la matrice A et le terme de droite **b** suivants

$$A = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix} \quad b = \begin{pmatrix} 1 \\ -1 \\ 1 \\ -1 \end{pmatrix}.$$

4.3.2 Méthode de Jacobi

La diagonale de A est

$$D = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 6 & 0 & 0 \\ 0 & 0 & 11 & 0 \\ 0 & 0 & 0 & 16 \end{pmatrix};$$

Prenond $\alpha = 1$ et $P = D$. La méthode de Richardson dans ce cas correspond à la méthode de Jacobi et s'écrit comme suit :

[88]:
```
A = np.array([[1,2,3,4],
              [5,6,7,8],
              [9,10,11,12],
              [13,14,15,16]])

b = np.array([1,-1,1,-1])

# Diagonale de A
P = np.diag(np.diag(A))
alpha = 1

# Initialisation, par exemple
k=0
xk = np.array([1,0,0,0])
# résidu
rk = b- A.dot(xk)
```

[89]:
```
# Une itération de Richardson

# résidu préconditionné :
zk = np.linalg.solve(P,rk)

# correction de la solution, x(k+1) :
xk1 = xk + alpha*zk
# résidu
rk = b- A.dot(xk1)

# Préparer la prochaine itération
xk = xk1
```

```
print (xk)
```

```
[ 1.     -1.     -0.7273 -0.875 ]
```

Si on execute plusieurs fois ces itérations, la méthode ne converge pas. Pourquoi ?

Dans ce cas, la matrice d'itération de la méthode de Richardson est égale à

$$B = D^{-1}(D - A) = I - D^{-1}A = \begin{pmatrix} 0 & -2 & -3 & -4 \\ -5/6 & 0 & -7/6 & -4/3 \\ -9/11 & -10/11 & 0 & -12/11 \\ -13/16 & -14/16 & -15/16 & 0 \end{pmatrix}.$$

[90]:
```
# Jacobi
# the first diag extracts the diagonal of A, the second one builds a diagonal␣
 ↪matrix starting from a vector
D = np.diag(np.diag(A))

# efficiently computing the Jacobi iteration matrix without explicitely␣
 ↪computing the inverse of D
Bj = np.linalg.solve(D,(D-A))

# What are the eigenvlues of Bj ?
lk, v = eig(Bj)
print(f'The eigenvalues of Bj are {lk}\n')
print(f'The spectral radius of Bj is {np.max(np.abs(lk)):.2f} > 1, therefore the␣
 ↪Jacobi method does not converge')
```

```
The eigenvalues of Bj are [-3.9362+0.j  1.9362+0.j  1.    +0.j  1.    +0.j]

The spectral radius of Bj is 3.94 > 1, therefore the Jacobi method does not
converge
```

4.3.3 Méthode de Gauss-Seidel

Prenond $\alpha = 1$ et P la partie triangulaire inférieure de A avec la diagonale. La méthode de Richardson dans ce cas correspond à la méthode de Gauss-Seidel et s'écrit comme suit :

[91]:
```
# Gauss-Seidel
# Return a copy of an array with elements above the k-th diagonal set to zero.
Pgs =  np.tril(A, k=0)

# efficiently computing the Gauss-Siedel iteration matrix without explicitely␣
 ↪computing the inverse of D
Bgs = np.linalg.solve(Pgs,(Pgs-A))

# What are the eigenvlues of Bgs ?
lk, v = eig(Bgs)
print(f'The eigenvalues of Bgs are {lk}\n')
```

```
print(f'The spectral radius of Bgs is {np.max(np.abs(lk)):.2f} > 1, therefore␣
 ↪the Gauss-Seidel method does not converge')
```

```
The eigenvalues of Bgs are [0.    +0.j 2.0682+0.j 1.    +0.j 1.    +0.j]

The spectral radius of Bgs is 2.07 > 1, therefore the Gauss-Seidel method does
not converge
```

La méthode de Gauss-Seidel appliqué à cette matrice ne converge pas.

4.3.4 Exercice

Réprenez ces deux méthodes avec la matrice

$$A = \begin{pmatrix} 3 & 2 & 1 & 0 \\ 3 & 4 & 3 & 2 \\ 3 & 2 & 5 & 2 \\ 1 & 2 & 3 & 4 \end{pmatrix}.$$

```
A = np.array([[3,2,1,0],[3,4,3,2],[3,2,5,2],[1,2,3,4]])
```

1. Vérifiez si Jacobi et Gauss-Seidel convergent à l'aide de la matrice d'itérations
2. Calculez les premier 10 itérations de ces méthodes.

4.4 Exemple 2

Considerez la matrice A et le vecteur b suivants

$$A = \begin{pmatrix} 3 & 2 & 1 \\ 1 & 4 & 1 \\ 2 & 4 & 8 \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} 4 \\ 5 \\ 6 \end{pmatrix}.$$

Exprimez (sans calculer) P, B, et $\rho(B)$ dans le cas de Jacobi et Gauss-Seidel.

[92]:
```
A = np.array([[3, 2, 1],
              [1, 4, 1],
              [2, 4, 8] ])
```

[93]:
```
# Jacobi
# first diag extract the diagonal of A, the second builds a diagonal matrix␣
 ↪starting from a vector
D = np.diag(np.diag(A))

Bj = np.linalg.solve(D,(D-A))
# What are the eigenvlues of $Bj$ ?
lk, v = eig(Bj)
print(f'The eigenvalues of Bj are {lk}\n')
print(f'The spectral radius of Bj is {np.max(np.abs(lk)):.2f} < 1, therefore␣
 ↪Jacobi converges')
```

```
The eigenvalues of Bj are [-0.7026+0.j  0.4205+0.j  0.2821+0.j]

The spectral radius of Bj is 0.70 < 1, therefore Jacobi converges
```

[94]:
```
# Gauss-Seidel
# Return a copy of an array with elements above the k-th diagonal zeroed.
Pgs =  np.tril(A,k=0)

Bgs = np.linalg.solve(Pgs,(Pgs-A))
# What are the eigenvlues of $Bgs$ ?
lk, v = eig(Bgs)
print(f'The eigenvalues of Bgs are {lk} whose moduli are {np.absolute(lk)} \n')
print(f'The spectral radius of Bgs is {np.max(np.absolute(lk)):.2f} < 1,␣
 ↪therefore Gauss-Seidel converges')
```

```
The eigenvalues of Bgs are [0.    +0.j     0.1667+0.1179j 0.1667-0.1179j] whose
moduli are [0.     0.2041 0.2041]

The spectral radius of Bgs is 0.20 < 1, therefore Gauss-Seidel converges
```

4.5 Exemple 3 - Jacobi et Gauss-Seidel avec relaxation

Nous avons vu que la méthode de Jacobi appliquée à la matrice

$$A = \begin{pmatrix} 3 & 2 & 1 & 0 \\ 3 & 4 & 3 & 2 \\ 3 & 2 & 5 & 2 \\ 1 & 2 & 3 & 4 \end{pmatrix}$$

ne converge pas, alors que celle de Gauss-Seidel converge.

Nous allons utiliser la méthode de Richardson stationnaire avec un paramètre de relaxation α constant pour voir si on peut améliorer la convergence.

On va utiliser la méthode de Richardson stationnaire avec un paramètre de relaxation α constant pour voir si on peut améliorer la convergence.

Partie 1 - Matrices d'itérations La matrice d'itérations pour la méthode de Richardson est $B(\alpha_k) = I - \alpha_k P^{-1} A$. En particulier, pour α constant, nous pouvons faire un graphique du rayon spectral $\rho(B(\alpha))$ dans les cas d'un préconditionneur égale à la diagonale de A (similaire à Jacobi) ou au triangle inférieur de A (similaire à Gauss-Seidel):

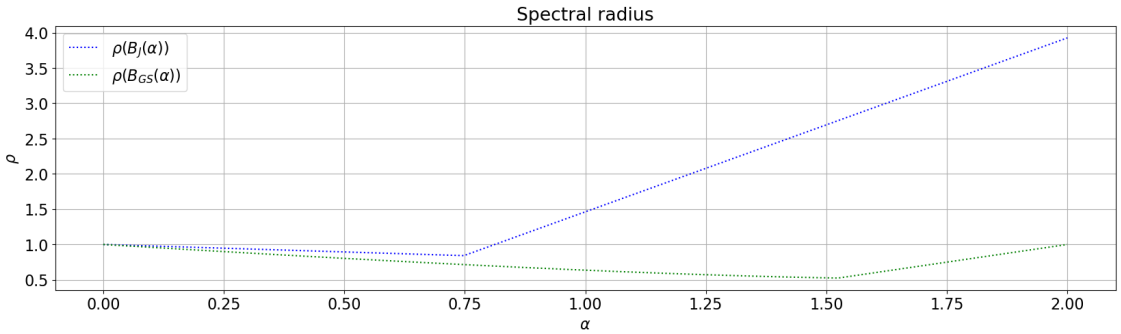
Voici comment voir que $B(\alpha_k) = I - \alpha_k P^{-1} A$:

Pour Richardson préconditionné nous avons que

$$\begin{aligned} Px^{(k+1)} &= Px^{(k)} + \alpha_k (b - Ax^{(k)}) \\ Px^{(k+1)} &= \alpha_k b + (P - \alpha_k A) x^{(k)} \\ x^{(k+1)} &= \alpha_k P^{-1} b + P^{-1}(P - \alpha_k A) x^{(k)} \end{aligned}$$

On reconnait la matric d'itération en $B(\alpha_k) = P^{-1}(P - \alpha_k A) = I - \alpha_k P^{-1} A)$

[95]:
```
A = np.array([[3,2,1,0],
              [3,4,3,2],
              [3,2,5,2],
              [1,2,3,4]])
D = np.diag(np.diag(A))
Pgs =  np.tril(A,k=0)

Alphas = np.linspace(0, 2, 3001)

rhoBj  = []
rhoBgs = []

for alpha in Alphas :
    Bgs = np.linalg.solve(Pgs,(Pgs-alpha*A))
    Bj = np.linalg.solve(D,(D-alpha*A))

    lk, v = eig(Bj)
    rhoBj.append( np.max(np.abs(lk)) )

    lk, v = eig(Bgs)
    rhoBgs.append( np.max(np.abs(lk)) )

plt.plot(Alphas, rhoBj, 'b:', label=r'$\rho(B_J(\alpha))$')
plt.plot(Alphas, rhoBgs, 'g:', label=r'$\rho(B_{GS}(\alpha))$')

plt.xlabel(r'$\alpha$'); plt.ylabel(r'$\rho$');
plt.title('Spectral radius')
plt.grid(True)
plt.legend(['$\\rho(B_J(\\alpha))$','$\\rho(B_{GS}(\\alpha))$'])
plt.show()
```



On constate que:

1. En utilisant $P = D$, Richardson converge pour $\alpha \lessapprox 0.8$ and the optimal one is for $\alpha \approx 0.75$

2. En utilisant P égal au triangle inférieur de A, Richardson converge pour $\alpha \lessapprox 2$; le paramètre optimale est $\alpha \approx 1.5$

Partie 2 - Implémentation de la Méthode de Richardson Définissez une fonction Python qui implémente la méthode de Richardson stationnaire avec la structure suivante:

```
def Richardson(A, b, x0, P, alpha, maxIterations, tolerance) :
    # Stationary Richardson method to approximate the solution of Ax=b
    # INPUT          :
    # x0             : initial guess
    # P              : preconditioner
    # alpha          : constant relaxation parameter
    # maxIterations  : maximum number of iterations
    # tolerance      : tolerance for relative residual
    # OUTPUT         :
    # xk             : approximate solution to the linear system
    # rk             : vector of relative norm of the residuals
```

La méthode de Richardson, comme toute méthode iterative a besoin d'un critère d'arrêt. Dans ce cas, le plus simple est de poser les critères suivants:

1. On fixe le nombre maximal d'itérations
2. Le résidu relatif est plus petit qu'une tolérance ϵ :

$$\frac{\|\mathbf{b} - A\mathbf{x}^{(k)}\|}{\|\mathbf{b}\|} < \epsilon$$

On s'arrête dès que l'un des ces critières est rempli.

[96]:
```
def Richardson(A, b, x0, P, alpha, maxIterations, tolerance) :
    # Stationary Richardson method to approximate the solution of Ax=b
    # INPUT          :
    # x0             : initial guess
    # P              : preconditioner
    # alpha          : constant relaxation parameter
    # maxIterations  : maximum number of iterations
    # tolerance      : tolerance for relative residual
    # OUTPUT         :
    # xk             : approximate solution to the linear system
    # rk             : vector of relative norm of the residuals

    # we do not keep track of all the sequence, just the last two entries
    xk = x0
    rk = b - A.dot(xk)

    RelativeResidualNorm = []
    for k in range(maxIterations) :

        zk = np.linalg.solve(P,rk)
        xk = xk + alpha*zk
```

```
        rk = b - A.dot(xk)

        # you can verify that this is equivalent to
        # rk = rk - alpha*A.dot(zk)
        RelativeResidualNorm.append(np.linalg.norm(rk)/np.linalg.norm(b))
        if ( RelativeResidualNorm[-1] < tolerance ) :
            print(f'Richardson converged in {k+1} iterations with a relative␣
↪residual of {RelativeResidualNorm[-1]:1.3e}')
            return xk, RelativeResidualNorm

    print(f'Richardson did not converge in {maxIterations} iterations, the␣
↪relative residual is {np.linalg.norm(rk)/np.linalg.norm(b):1.3e}')
    return xk, RelativeResidualNorm
```

Partie 3 - Utilisation de la Méthode de Richardson Utilisez la fonction `Richardson` pour approcher la solution de $A\mathbf{x} = \mathbf{b}$ pour $\mathbf{b} = (0, 1, -1, 1)^T$ avec une tolérance sur le résidu relatif de 10^{-6} et le vecteur nul comme point initial et un nombre maximum d'itérations de 200

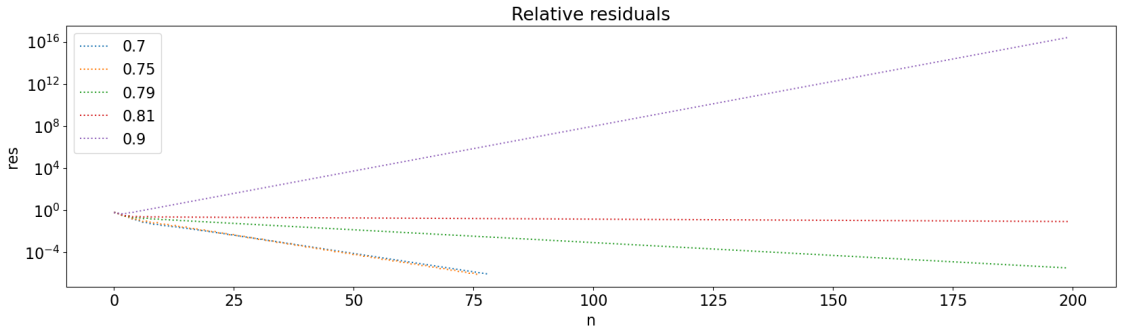
1. En utilisant $P = D$, avec : $\alpha = 0.7, 0.75, 0.79, 0.81, 0.9$
2. En utilisant P égal au triangle inférieur de A, avec : $\alpha = 1, 1.5, 1.95, 2.05$

Comment interprétez-vous ces résultats ?

[97]:
```
b = np.array([0,1,-1,1])
# Define the initial guess
x0 = 0*b # trick to have the right size for x0

tolerance = 1e-6
maxIter = 200
```

[98]:
```
# Jacobi-like preconditioner
P = np.diag(np.diag(A))
legend = []

for alpha in [0.7, 0.75, 0.79,0.81, 0.9] :
    x,relRes = Richardson(A,b,x0,P,alpha,maxIter,tolerance)
    print(relRes[-1])
    plt.plot( relRes, ':')
    legend.append(str(alpha))

plt.legend(legend)

plt.xlabel('n'); plt.ylabel('res');
plt.title('Relative residuals')
plt.yscale('log')
plt.show()
```

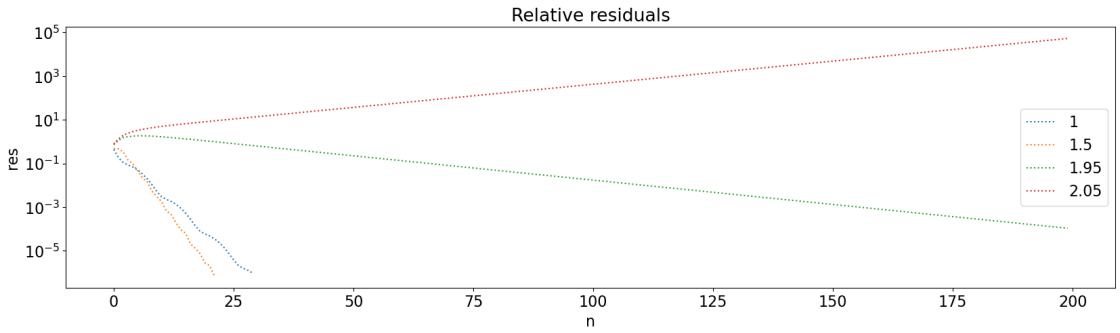
```
Richardson converged in 79 iterations with a relative residual of 8.984e-07
8.984498815385382e-07
Richardson converged in 77 iterations with a relative residual of 7.593e-07
7.592562065358359e-07
Richardson did not converge in 200 iterations, the relative residual is
3.379e-06
3.378918951868571e-06
Richardson did not converge in 200 iterations, the relative residual is
8.699e-02
0.08698843985290847
Richardson did not converge in 200 iterations, the relative residual is
2.576e+16
2.575559592693911e+16
```



[99]:
```
# Gauss-Seidel-like preconditioner
P = np.tril(A,k=0)
legend = []

for alpha in [1, 1.5, 1.95,2.05] :
    x,relRes = Richardson(A,b,x0,P,alpha,maxIter,tolerance)
    print(relRes[-1])
    plt.plot( relRes, ':')
    legend.append(str(alpha))

plt.legend(legend)

plt.xlabel('n'); plt.ylabel('res');
plt.title('Relative residuals')
plt.yscale('log')
plt.show()
```

```
Richardson converged in 30 iterations with a relative residual of 9.521e-07
9.520753485686893e-07
Richardson converged in 22 iterations with a relative residual of 7.144e-07
7.144137957248418e-07
```

```
Richardson did not converge in 200 iterations, the relative residual is
1.071e-04
0.00010708766426366904
Richardson did not converge in 200 iterations, the relative residual is
5.283e+04
52829.70703107408
```



Partie 4 - Implémentation de la Méthode du Gradient Préconditionné Définissez une fonction Python qui implémente la méthode de Gradient Préconditionné et qui a la structure

```
def PrecGradient(A, b, x0, P, maxIterations, tolerance) :
    # Preconditioned Gradient method to approximate the solution of Ax=b
    # INPUT          :
    # x0             : initial guess
    # P              : preconditioner
    # maxIterations  : maximum number of iterations
    # tolerance      : tolerance for relative residual
    # OUTPUTS        :
    # xk             : approximate solution to the linear system
    # rk             : vector of relative norm of the residuals
```

[100]:
```
def PrecGradient(A, b, x0, P, maxIterations, tolerance) :
    # Preconditioned Gradient method to approximate the solution of Ax=b
    # INPUT          :
    # x0             : initial guess
    # P              : preconditioner
    # maxIterations  : maximum number of iterations
    # tolerance      : tolerance for relative residual
    # OUTPUTS        :
    # xk             : approximate solution to the linear system
    # rk             : vector of relative norm of the residuals

    # we do not keep track of all the sequence, just the last two entries
    xk = x0
    rk = b - A.dot(xk)
```

```
    RelativeResidualNorm = []
    for k in range(maxIterations) :

        zk = np.linalg.solve(P,rk)
        Azk = A.dot(zk)

        alphak = zk.dot(rk) / zk.dot(Azk)

        xk = xk + alphak*zk

        rk = b - A.dot(xk)
        # you can verify that this is equivalent to
        # rk = rk - alphak*A.dot(zk)
        RelativeResidualNorm.append(np.linalg.norm(rk)/np.linalg.norm(b))
        if ( RelativeResidualNorm[-1] < tolerance ) :
            print(f'Gradient converged in {k+1} iterations with a relative␣
↪residual of {RelativeResidualNorm[-1]:1.3e}')
            return xk, RelativeResidualNorm

    print(f'Graident did not converge in {maxIterations} iterations, the␣
↪relative residual is {np.linalg.norm(rk)/np.linalg.norm(b):1.3e}')
    return xk, RelativeResidualNorm
```

Partie 5 - Implémentation de la Méthode du Gradient Conjugué Préconditionné
Définissez une fonction Python qui implémente la méthode du Gradient Conjugué Préconditionné avec la structure suivante:

```
def PrecConjugateGradient(A, b, x0, P, maxIterations, tolerance) :
    # Preconditionate Conjugate Gradient method to approximate the solution of Ax=b
    # INPUT          :
    # x0             : initial guess
    # P              : preconditioner
    # maxIterations  : maximum number of iterations
    # tolerance      : tolerance for relative residual
    # OUTPUTS        :
    # xk             : approximate solution to the linear system
    # rk             : vector of relative norm of the residuals
```

[101]:
```
def PrecConjugateGradient(A, b, x0, P, maxIterations, tolerance) :
    # Preconditionate Conjugate Gradient method to approximate the solution of␣
↪Ax=b
    # INPUT          :
    # x0             : initial guess
    # P              : preconditioner
    # maxIterations  : maximum number of iterations
```

```
    # tolerance     : tolerance for relative residual
    # OUTPUTS       :
    # xk            : approximate solution to the linear system
    # rk            : vector of relative norm of the residuals

    # we do not keep track of all the sequence, just the last two entries
    xk = x0
    rk = b - A.dot(xk)
    zk = np.linalg.solve(P,rk)
    pk = zk

    RelativeResidualNorm = []
    for k in range(maxIterations) :

        Apk = A.dot(pk)
        alphak = pk.dot(rk) / pk.dot(Apk)

        xk = xk + alphak*pk

        rk = rk - alphak*Apk
        # you can verify that this is equivalent to
        # rk = b - A.dot(xk)

        zk = np.linalg.solve(P,rk)

        betak = Apk.dot(zk) / pk.dot(Apk)
        pk = zk - betak*pk


        RelativeResidualNorm.append(np.linalg.norm(rk)/np.linalg.norm(b))
        if ( RelativeResidualNorm[-1] < tolerance ) :
            print(f'Conjugate Gradient converged in {k+1} iterations with a␣
↪relative residual of {RelativeResidualNorm[-1]:1.3e}')
            return xk, RelativeResidualNorm

    print(f'Conjugate Gradient did not converge in {maxIterations} iterations,␣
↪the relative residual is {np.linalg.norm(rk)/np.linalg.norm(b):1.3e}')
    return xk, RelativeResidualNorm
```

Partie 6 - Utilisation de la Méthode du Gradient Preconditioné et du Gradient Conjugué Préconditionné Utilisez les fonctions `PrecGradient` et `PrecConjugateGradient` pour approcher la solution de $A\mathbf{x} = \mathbf{b}$ pour $\mathbf{b} = (0, 1, -1, 1)^T$ avec une tolérance sur le résidu relatif de 10^{-6} et le vecteur nul comme point initial, et un nombre maximum d'itérations de 200

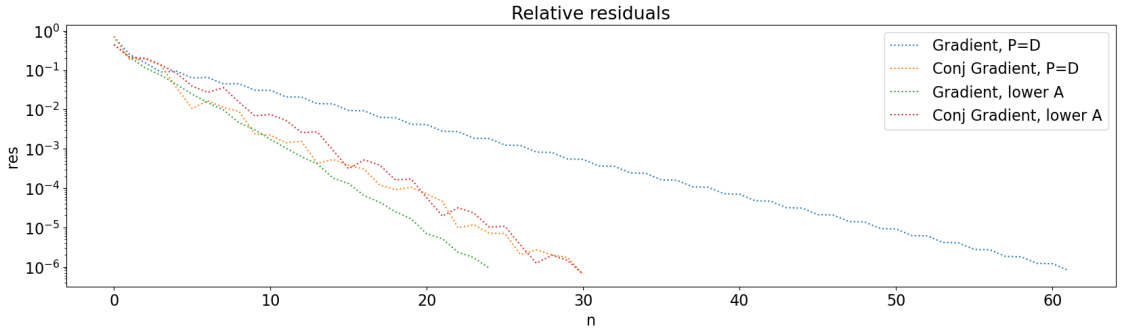
1. En utilisant $P = D$
2. En utilisant P égal au triangle inférieur de A

Comment interprétez-vous ces résultats ?

[102]:
```
legend = []

# Jacobi-like precondtioner
P = np.diag(np.diag(A))

x,relRes = PrecGradient(A,b,x0,P,maxIter,tolerance)
print(relRes[-1])
plt.plot( relRes, ':')
legend.append('Gradient, P=D')

x,relRes = PrecConjugateGradient(A,b,x0,P,maxIter,tolerance)
print(relRes[-1])
plt.plot( relRes, ':')
legend.append('Conj Gradient, P=D')

# Gauss-Seidel-like precondtioner
P = np.tril(A,k=0)

x,relRes = PrecGradient(A,b,x0,P,maxIter,tolerance)
print(relRes[-1])
plt.plot( relRes, ':')
legend.append('Gradient, lower A')

x,relRes = PrecConjugateGradient(A,b,x0,P,maxIter,tolerance)
print(relRes[-1])
plt.plot( relRes, ':')
legend.append('Conj Gradient, lower A')

plt.legend(legend)

plt.xlabel('n'); plt.ylabel('res');
plt.title('Relative residuals')
plt.yscale('log')
plt.show()
```

```
Gradient converged in 62 iterations with a relative residual of 8.146e-07
8.145842032913786e-07
Conjugate Gradient converged in 31 iterations with a relative residual of
6.323e-07
6.323181670011519e-07
Gradient converged in 25 iterations with a relative residual of 9.177e-07
9.177277863663321e-07
Conjugate Gradient converged in 31 iterations with a relative residual of
6.529e-07
```

```
6.529042663207426e-07
```



4.6 Autres Exemples

Répétez les expériences numériques ci-dessous pour approcher les solutions de

$$\begin{pmatrix} 2 & 1 \\ 1 & 3 \end{pmatrix} \mathbf{x} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad , \quad \begin{pmatrix} 2 & 2 \\ -1 & 3 \end{pmatrix} \mathbf{x} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad \text{et} \quad \begin{pmatrix} 5 & 7 \\ 7 & 10 \end{pmatrix} \mathbf{x} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}.$$

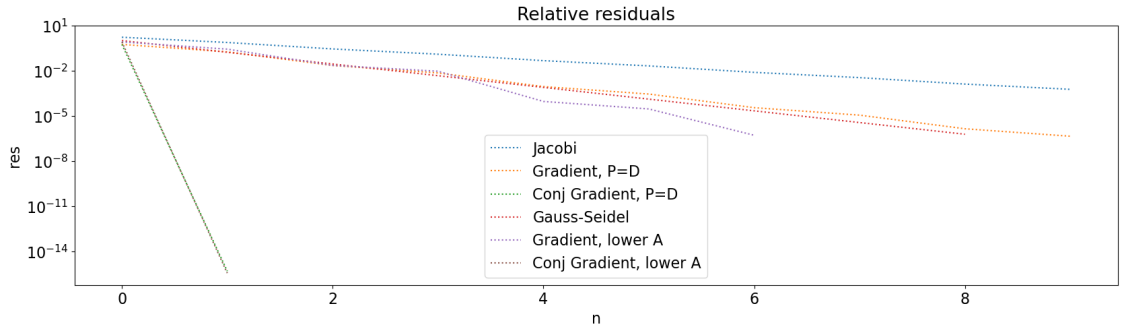
Pour quelles matrices et méthodes vous attendez-vous à une convergence ? Pour la dernière matrice, le résultat n'est pas celui attendu, pourquoi ?

[103]:
```
b = np.array([1,0])
# Define the initial guess
x0 = np.array([1,1])

tolerance = 1e-6
maxIter = 10
A = np.array([[2,1],[1,3]])
# A = np.array([[2,2],[-1,3]])
# A = np.array([[5,7],[7,10]])

legend = []

# To complete: similarly to previous example:
# 1) Jacobi, Gauss-Seidel (simple)
# 2) PrecGradient and PrecConjGradient, with
# both Jacobi-like and Gauss-Seidel-like preconditioning

# 1) Jacobi-like precondtioner
P = np.diag(np.diag(A))

x,relRes = Richardson(A,b,x0,P,1,maxIter,tolerance)
print(relRes[-1])
plt.plot( relRes, ':')
```

```
legend.append('Jacobi')

x,relRes = PrecGradient(A,b,x0,P,maxIter,tolerance)
print(relRes[-1])
plt.plot( relRes, ':')
legend.append('Gradient, P=D')

x,relRes = PrecConjugateGradient(A,b,x0,P,maxIter,tolerance)
print(relRes[-1])
plt.plot( relRes, ':')
legend.append('Conj Gradient, P=D')

# Gauss-Seidel-like precondtioner
P = np.tril(A,k=0)

x,relRes = Richardson(A,b,x0,P,1,maxIter,tolerance)
print(relRes[-1])
plt.plot( relRes, ':')
legend.append('Gauss-Seidel')

x,relRes = PrecGradient(A,b,x0,P,maxIter,tolerance)
print(relRes[-1])
plt.plot( relRes, ':')
legend.append('Gradient, lower A')

x,relRes = PrecConjugateGradient(A,b,x0,P,maxIter,tolerance)
print(relRes[-1])
plt.plot( relRes, ':')
legend.append('Conj Gradient, lower A')

plt.legend(legend)

plt.xlabel('n'); plt.ylabel('res');
plt.title('Relative residuals')
plt.yscale('log')
plt.show()
```

```
Richardson did not converge in 10 iterations, the relative residual is 5.751e-04
0.0005751203645832735
Gradient converged in 10 iterations with a relative residual of 4.401e-07
4.40060404871371e-07
Conjugate Gradient converged in 2 iterations with a relative residual of
4.710e-16
4.710277376051325e-16
Richardson converged in 9 iterations with a relative residual of 5.954e-07
5.953741807340762e-07
Gradient converged in 7 iterations with a relative residual of 5.098e-07
5.098498742819304e-07
```

```
Conjugate Gradient converged in 2 iterations with a relative residual of
3.511e-16
3.510833468576701e-16
```



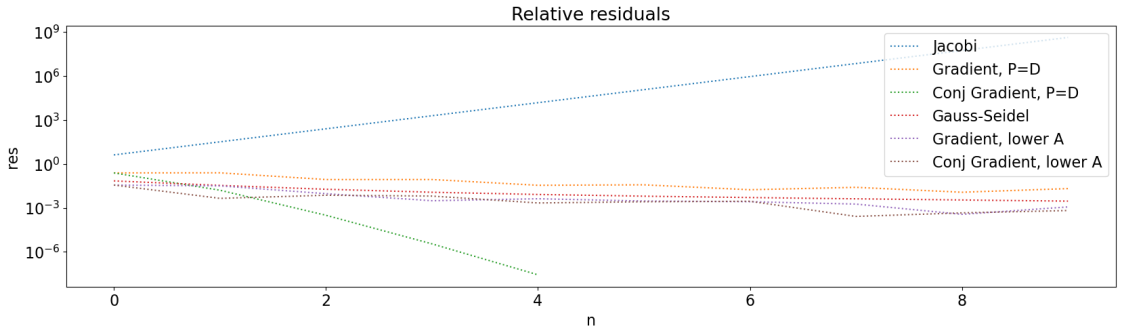
La matrice de Hilbert Peut-on utiliser une méthode de Richardson pour résoudre le problème du mauvais conditionnement de la matrice de Hilbert ?

[104]:
```
n = 10
A = hilbert(n)
xexact = np.ones(n)
b = A.dot(xexact)

# Define the initial guess
x0 = b

tolerance = 1e-6
maxIter = 10

legend = []

# To complete: similarly to previous example:
# 1) Jacobi, Gauss-Seidel (simple)
# 2) PrecGradient and PrecConjGradient, with
# both Jacobi-like and Gauss-Seidel-like preconditioning

# 1) Jacobi-like precondtioner
P = np.diag(np.diag(A))

x,relRes = Richardson(A,b,x0,P,1,maxIter,tolerance)
print(f'The relatove residual and absolute error are : \t {relRes[-1]:1.3e} \t␣
 ↪{np.linalg.norm(x-xexact):1.3e}\n')
plt.plot( relRes, ':')
legend.append('Jacobi')

x,relRes = PrecGradient(A,b,x0,P,maxIter,tolerance)
```

```
print(f'The relatove residual and absolute error are : \t {relRes[-1]:1.3e} \t␣
 ↪{np.linalg.norm(x-xexact):1.3e}\n')
plt.plot( relRes, ':')
legend.append('Gradient, P=D')

x,relRes = PrecConjugateGradient(A,b,x0,P,maxIter,tolerance)
print(f'The relatove residual and absolute error are : \t {relRes[-1]:1.3e} \t␣
 ↪{np.linalg.norm(x-xexact):1.3e}\n')
plt.plot( relRes, ':')
legend.append('Conj Gradient, P=D')

# Gauss-Seidel-like precondtioner
P = np.tril(A,k=0)

x,relRes = Richardson(A,b,x0,P,1,maxIter,tolerance)
print(f'The relatove residual and absolute error are : \t {relRes[-1]:1.3e} \t␣
 ↪{np.linalg.norm(x-xexact):1.3e}\n')
plt.plot( relRes, ':')
legend.append('Gauss-Seidel')

x,relRes = PrecGradient(A,b,x0,P,maxIter,tolerance)
print(f'The relatove residual and absolute error are : \t {relRes[-1]:1.3e} \t␣
 ↪{np.linalg.norm(x-xexact):1.3e}\n')
plt.plot( relRes, ':')
legend.append('Gradient, lower A')

x,relRes = PrecConjugateGradient(A,b,x0,P,maxIter,tolerance)
print(f'The relatove residual and absolute error are : \t {relRes[-1]:1.3e} \t␣
 ↪{np.linalg.norm(x-xexact):1.3e}\n')
plt.plot( relRes, ':')
legend.append('Conj Gradient, lower A')

plt.legend(legend)

plt.xlabel('n'); plt.ylabel('res');
plt.title('Relative residuals')
plt.yscale('log')
plt.show()
```

```
Richardson did not converge in 10 iterations, the relative residual is 4.305e+08
The relatove residual and absolute error are :   4.305e+08       1.847e+09

Graident did not converge in 10 iterations, the relative residual is 2.066e-02
The relatove residual and absolute error are :   2.066e-02       8.555e-01

Conjugate Gradient converged in 5 iterations with a relative residual of
2.692e-08
```

```
The relatove residual and absolute error are :   2.692e-08       2.280e-02

Richardson did not converge in 10 iterations, the relative residual is 2.883e-03
The relatove residual and absolute error are :   2.883e-03       4.243e-01

Graident did not converge in 10 iterations, the relative residual is 1.158e-03
The relatove residual and absolute error are :   1.158e-03       4.149e-01

Conjugate Gradient did not converge in 10 iterations, the relative residual is
6.569e-04
The relatove residual and absolute error are :   6.569e-04       4.205e-01
```



[ ]:

4.7 Compléments

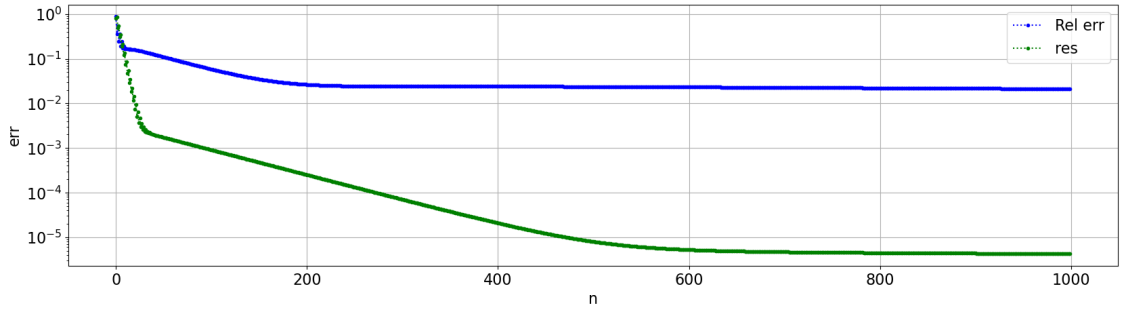
[105]:
```
# importing libraries used in this book
import numpy as np
import matplotlib.pyplot as plt


# scipy.linalg.lu : LU decomposition
from scipy.linalg import lu
# scipy.linalg.cholesky : Cholesky decomposition
from scipy.linalg import cholesky
from scipy.linalg import hilbert

from IterativeMethodsLib import *
```

Dans le cas d'une matrice mal conditionnée, on peut essayer d'utiliser la factorisation LU ou de Cholesky comme préconditionneur. Mais cela ne marche pas forcément ...

[106]:
```
n = 4
A = hilbert(n)
```

```
D = np.diag(np.diag(A))

MaxIterations = 1000

x = np.ones([n,1])
b = A.dot(x)
alpha = 0.5

# we do not keep track of all the sequence, just the last two entries
xk = b
rk = b - A.dot(xk)

RelativeError = []
residualNorm = []

# We rewrite Richardson for the sake of the example:
for k in range(MaxIterations) :

    zk = np.linalg.solve(D,rk)

    xk = xk + alpha*zk
    rk = rk - alpha*A.dot(zk)
    #rk = b - A.dot(xk)


    RelativeError.append( np.linalg.norm(x-xk) / np.linalg.norm(x) )
    residualNorm.append( np.linalg.norm(rk) )
```

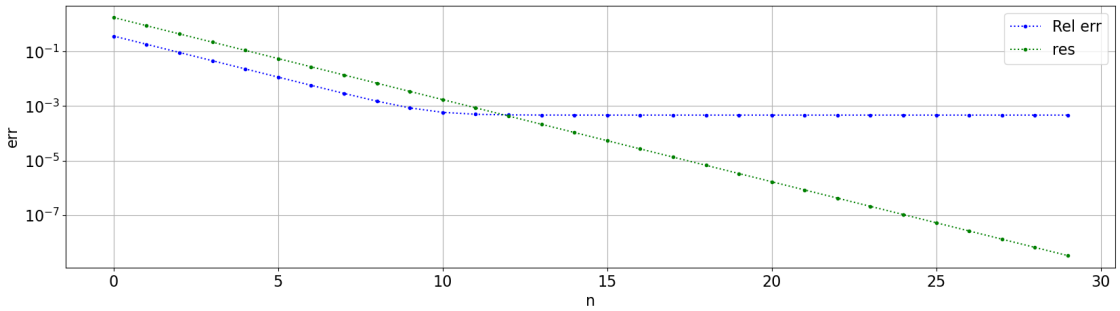
[107]:
```
#plt.plot(range(MaxIterations), RelativeError, 'b:.')
plt.plot(range(MaxIterations), RelativeError, 'b:.',range(MaxIterations),␣
 ↪residualNorm, 'g:.')

plt.xlabel('n'); plt.ylabel('err');
# plt.xscale('log')
plt.yscale('log')
plt.grid(True)
plt.legend(['Rel err','res'])
plt.show()
```



[108]:
```
n = 10
A = hilbert(n)

# epsilon = 2
#A = np.array([[epsilon, 1, 2],
#               [1, 3, 1],
#               [2, 1, 3] ])

# B = np.diag(np.diag(A, k=1),k=1) + np.diag(np.diag(A, k=0),k=0) + np.diag(np.
 ↪diag(A, k=-1),k=-1)
# L = cholesky(B, lower=True)
P, L, U = lu(A)

MaxIterations = 30

x = np.ones([n,1])
b = A.dot(x)
alpha = 0.5

# we do not keep track of all the sequence, just the last two entries
xk = b
rk = b - A.dot(xk)

RelativeError = []
residualNorm = []

for k in range(MaxIterations) :

    # y  = np.linalg.solve(L,rk)
    # zk = np.linalg.solve(L.T,y)

    y  = np.linalg.solve(L,P.T.dot(rk))
    zk = np.linalg.solve(U,y)

    xk = xk + alpha*zk
```

```
    rk = rk - alpha*A.dot(zk)
    #rk = b - A.dot(xk)


    RelativeError.append( np.linalg.norm(x-xk) / np.linalg.norm(x) )
    residualNorm.append( np.linalg.norm(rk) )
```

[109]:
```
#plt.plot(range(MaxIterations), RelativeError, 'b:.')
plt.plot(range(MaxIterations), RelativeError, 'b:.',range(MaxIterations),␣
↪residualNorm, 'g:.')

plt.xlabel('n'); plt.ylabel('err');
# plt.xscale('log')
plt.yscale('log')
plt.grid(True)
plt.legend(['Rel err','res'])
plt.show()
```



[ ]:

4.8 Problèmes d'arrondis

Voyons comment l'ordre dans la somme des ligne d'une matrice de Hilbert change le résultat. Pour rappel, la matrice de Hilbert de taille $n \times n$ est une matrice symétrique définie par

$$a_{ij} = \frac{1}{i+j-1}, \qquad i,j = 1,\dots,n$$

donc la somme d'une ligne s_i est

$$s_i = \sum_{j=1}^{n} \frac{1}{i+j-1}, \qquad i = 1,\dots,n$$

On peut la calculer de manière exacte

[110]:
```
import fractions
import numpy as np
from scipy.linalg import hilbert
```

[111]:
```
def SumHilbertLines (n) :
    s = []
    for i in range(n) :
        s.append(fractions.Fraction(0,1))
        for j in range(n) :
            s[i] = s[i] + fractions.Fraction(1,i+j-1+2)
        # print (f's[{i}] = {s[i]}')
    return s
```

Differents méthodes pour faire la somme :

[112]:
```
def regularSum(A) :
    return np.sum(A,axis=1)

def dotSum(A) :
    n = A.shape[0]
    x = np.ones(n)
    return A.dot(x)

def onebyoneSum(A) :
    n = A.shape[0]
    s = np.zeros(n)
    for i in range(n) :
        for j in range(n) :
            s[i] = s[i] + A[i,j]

    return s



##
def kahan_sum(a, axis=0):
    '''Kahan summation of the numpy array along an axis.
    '''
    s = np.zeros(a.shape[:axis] + a.shape[axis+1:])
    c = np.zeros(s.shape)
    for i in range(a.shape[axis]):
        # https://stackoverflow.com/a/42817610/353337
        y = a[(slice(None),) * axis + (i,)] - c
        t = s + y
        c = (t - s) - y
        s = t.copy()
    return s
```

```
def kahanSum(A) :
## Kahan Sum

    n = A.shape[0]
    b = np.zeros(n)
    for k in range(n) :
        b[k] = kahan_sum(A[k,:])
    return b

##

def kahanSortedSum(A) :
    n = A.shape[0]
    b = np.zeros(n)
    for k in range(n) :
        ind = np.argsort(np.abs(A[k,:]), axis=0)
        b[k] = kahan_sum(A[k,ind])

    return b

##

def sortedSum(A) :
    n = A.shape[0]
    b = np.zeros(n)
    for k in range(n) :
        ind = np.argsort(np.abs(A[k,:]), axis=0)
        b[k] = np.sum(A[k,ind])

    return b
```

[113]:
```
n = 1000
s = SumHilbertLines (n)

A = hilbert(n)

s_r = regularSum(A)
s_d = dotSum(A)
s_o = onebyoneSum(A)
s_s = sortedSum(A)
s_kr = kahanSum(A)
s_ks = kahanSortedSum(A)
```

[114]:
```
print(np.max(np.abs(s - s_r)))
print(np.max(np.abs(s - s_d)))
print(np.max(np.abs(s - s_o)))
```

```
print(np.max(np.abs(s - s_s)))
print(np.max(np.abs(s - s_kr)))
print(np.max(np.abs(s - s_ks)))
```

```
8.881784197001252e-16
1.7763568394002505e-15
9.769962616701378e-15
8.881784197001252e-16
8.881784197001252e-16
8.881784197001252e-16
```

[ ]:

[ ]:

5 Dérivée numérique

Soit $f : [a, b] \to \mathbf{R}$, de classe C^1 et $x_0 \in [a, b]$. La dérivée $f'(x_o)$ est donnée par

$$\begin{aligned} f'(x_0) &= \lim_{h\to 0^+} \frac{f(x_0+h)-f(x_0)}{h}, \\ &= \lim_{h\to 0^+} \frac{f(x_0)-f(x_0-h)}{h}, \\ &= \lim_{h\to 0} \frac{f(x_0+h)-f(x_0-h)}{2h}. \end{aligned}$$

Soient $x_0 \in [a, b]$, (Dy) une approximation de $f'(x_0)$ et (D^2y) une approximation de $f''(x_0)$.

On appelle

- **différence finie progressive** l'approximation

$$(Dy)^P = \frac{f(x_0+h)-f(x_0)}{h}$$

- **différence finie rétrograde** l'approximation

$$(Dy)^R = \frac{f(x_0)-f(x_0-h)}{h}$$

- **différence finie centrée** l'approximation

$$(Dy)^C = \frac{f(x_0+\frac{1}{2}h)-f(x_0-\frac{1}{2}h)}{h}$$

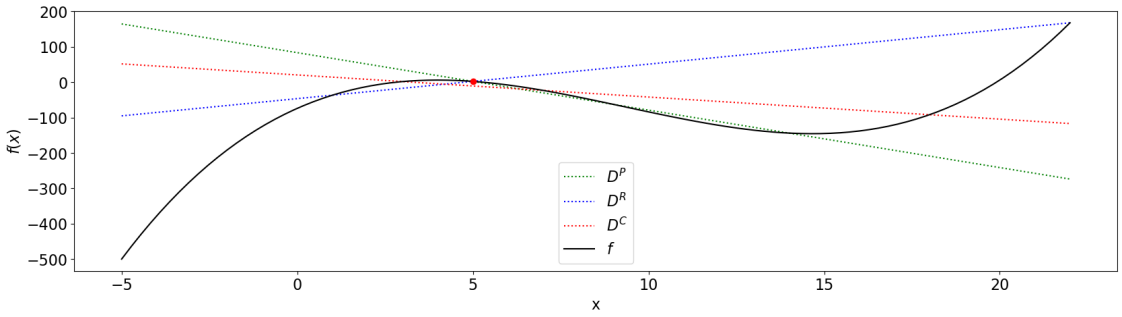
- **différence finie centrée d'ordre 2** l'approximation

$$(D^2y)^C = \frac{f(x_0+h)-2f(x_0)+f(x_0-h)}{h^2}$$

[115]:
```
# importing libraries used in this book
import numpy as np
import matplotlib.pyplot as plt
```

Les différences finies progressive, rétrograde et centrée approchent la dérivée de la fonction au le point x_0.

[116]:
```
# Define a function and a point x0
f = lambda x : 0.25*(x-4)**3 - 4*(x-4)**2 +6
x0 = 5

# Choosing finite difference size
h = 4


# Defininig first order finite differences
DP = ( f(x0 + h) - f(x0) ) / h
DR = ( f(x0) - f(x0-h) ) / h
DC = ( f(x0 + h/2) - f(x0-h/2) ) / h

# defining the straight lines with slope equal to the FDs
dfp = lambda x : f(x0) + DP*(x-x0)
dfr = lambda x : f(x0) + DR*(x-x0)
dfc = lambda x : f(x0-h/2) + DC*(x-x0+h/2)

# Drowing points
z = np.linspace(-5, 22, 100)

plt.plot(z, dfp(z), 'g:', z, dfr(z), 'b:', z, dfc(z), 'r:' )
plt.plot(z, f(z), 'k', x0,f(x0),'ro')

plt.xlabel('x'); plt.ylabel('$f(x)$');
plt.legend(['$D^P$', '$D^R$', '$D^C$', '$f$'])
plt.show()
```

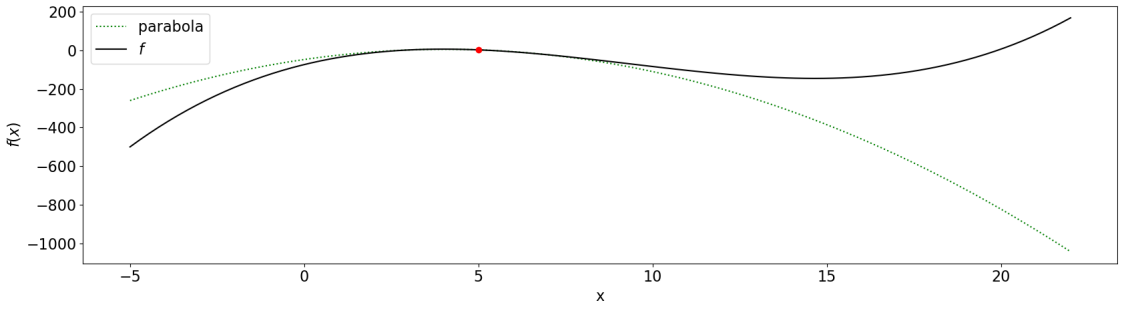


La différence finie d'ordre 2 approche la deuxième dérivée de la fonction au le point x_0.

Pour le voir graphiquement, on peut dessiner une fonction quadratique qui a la forme

$$p_2(x) = f(x_0) + (Dy)^C\,(x - x_0) + \frac{1}{2}(D^2y)\,(x - x_0)^2$$

[117]:
```
# Define a function and a point x0
f = lambda x : 0.25*(x-4)**3 - 4*(x-4)**2 +6
x0 = 5

# Choosing finite difference size
h = 4


# Defininig first order centered finite differences
DC = ( f(x0 + h/2) - f(x0-h/2) ) / h

# Defininig second order centered finite differences
D2 = ( f(x0 + h) - 2* f(x0) + f(x0-h) ) / (h**2)


# defining the parabola going through (x_0, y_0)
parabola = lambda x : f(x0) + DC*(x-x0) + 0.5 * D2 * (x-x0)**2

# Drowing points
z = np.linspace(-5, 22, 100)

plt.plot(z, parabola(z), 'g:' )
plt.plot(z, f(z), 'k', x0,f(x0),'ro')

plt.xlabel('x'); plt.ylabel('$f(x)$');
plt.legend(['parabola', '$f$'])
plt.show()
```



5.1 Exercice

Il s'agit de vérifier numériquement que

$$\left|f'(x_0) - (Dy)^P\right| = O(h^1).$$

On pose $x_0 = 1$, $f(x) = \sin(x)\ \forall x \in \mathbb{R}$

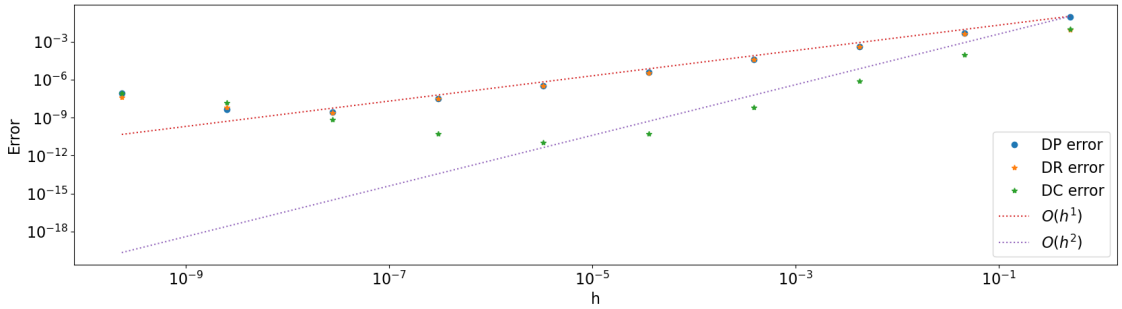
Calculez l'erreur commise en utilisant la différence finie progressive.

Quelle est la valeur de l'erreur pour h=0.1?

Ensuite vérifiez numériquement l'ordre pour $(Dy)^R$ et $(Dy)^C$.

[118]:
```
def DPerror(f,df, x0,h) :
    # formule de differences finies
    DP = ( f(x0 + h) - f(x0) ) / h
    err = abs(DP-df(x0));
    return err

def DRerror(f,df, x0,h) :
    # formule de differences finies
    DR = ( f(x0) - f(x0-h) ) / h
    err = abs(DR-df(x0));
    return err

def DCerror(f,df, x0,h) :
    # formule de differences finies
    DC = ( f(x0 + h/2) - f(x0-h/2) ) / h
    # Similarly, it is possible to define
    # DC = ( f(x0 + h) - f(x0-h) ) / (2*h)
    err = abs(DC-df(x0));
    return err


# This function just provides a line to compare the convergence in a log-log␣
 ↪plot.
from FiniteDifferenceLib import sampleConvergence
# sampleConvergence(h,order,ref) :
#   computes a pseudo order of convergence on h
#   ref is a reference maximum value


x0 = 0.2

h = 2**np.linspace(-32,-1,10)
plt.plot(h, DPerror(np.sin, np.cos, x0, h) , 'o' )
plt.plot(h, DRerror(np.sin, np.cos, x0, h) , '*' )
plt.plot(h, DCerror(np.sin, np.cos, x0, h) , '*' )


plt.plot(h, sampleConvergence(h, 1, 1e-1) , ':',
```

```
        h, sampleConvergence(h, 2, 1e-1) , ':')

plt.xlabel('h'); plt.ylabel('Error');
plt.legend(['DP error','DR error', 'DC error', '$O(h^1)$', '$O(h^2)$'])
plt.xscale('log')
plt.yscale('log')
plt.show()
```



5.2 Exercice

Il s'agit de vérifier numériquement que

$$\left|f''(x_0) - (D^2 y)\right| = O(h^2).$$

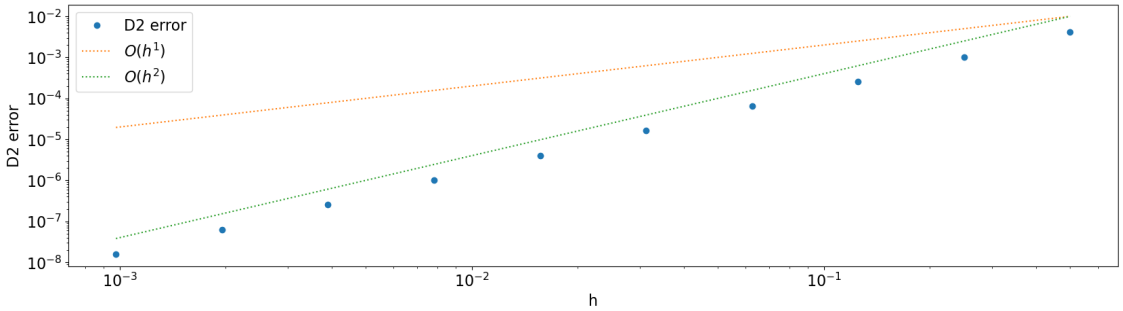
On pose $x_0 = 1$, $f(x) = \sin(x)\ \forall x \in \mathbb{R}$

Calculez l'erreur commise par ce schéma.

Que vaut l'erreur pour h=0.1?

[119]:
```
def D2error(f,df, x0,h) :
    # formule de differences finies
    D2 = ( f(x0 + h) - 2* f(x0) + f(x0-h) ) / (h**2)
    err = abs(D2-df(x0));
    # print(' x0 %e h %e erreur %e \n',x0,h,err)

    return err

x0 = 0.2

df = lambda x : -np.sin(x)

h = 2**np.linspace(-10,-1,10)
plt.plot(h, D2error(np.sin, df, x0, h) , 'o' )
plt.plot(h, sampleConvergence(h, 1, 1e-2) , ':',
         h, sampleConvergence(h, 2, 1e-2) , ':')
```

```
plt.xlabel('h'); plt.ylabel('D2 error');
plt.legend(['D2 error', '$O(h^1)$', '$O(h^2)$'])
plt.xscale('log')
plt.yscale('log')
plt.show()
```



5.3 Exercice

Il s'agit de vérifier numériquement que

$$\left| f'(x_0) - \frac{3f(x_0) - 4f(x_0 - h) + f(x_0 - 2h)}{2h} \right| = O(h^2).$$

Cette formule de différences finies est à l'origine du schéma "BDF2'' pour résoudre numériquement des équations différentielles (Chapitre 9 du livre).

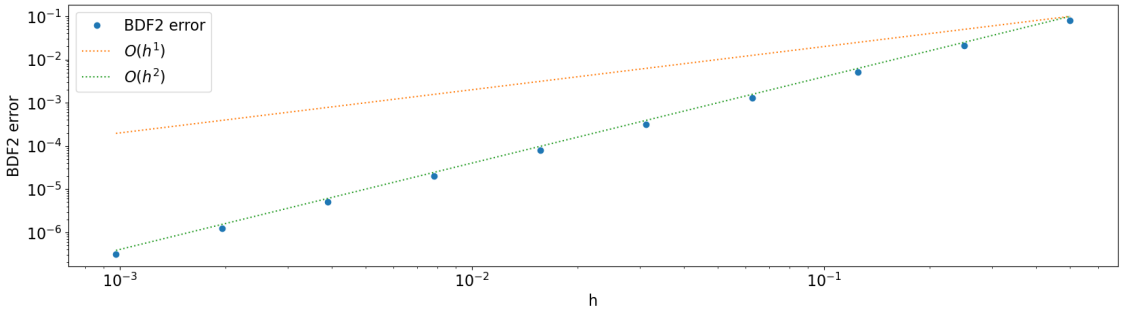
On pose $x_0 = 1$, $f(x) = \sin(x)$ $\forall x \in \mathbb{R}$

Calculez l'erreur commise par ce schéma.

Que vaut l'erreur pour h=0.1?

[120]:
```
def bdf2error(f,df, x0,h) :
    # formule de differences finies BDF2
    diff = (3*f(x0)-4*f(x0-h)+f(x0-2*h))/(2*h);
    err = abs(diff-df(x0));
    # print(' x0 %e h %e erreur %e \n',x0,h,err)

    return err

x0 = 0.2

h = 2**np.linspace(-10,-1,10)
plt.plot(h, bdf2error(np.sin, np.cos, x0, h) , 'o' )
plt.plot(h, sampleConvergence(h, 1, 1e-1) , ':',
        h, sampleConvergence(h, 2, 1e-1) , ':')
```

```
plt.xlabel('h'); plt.ylabel('BDF2 error');
plt.legend(['BDF2 error', '$O(h^1)$', '$O(h^2)$'])
plt.xscale('log')
plt.yscale('log')
plt.show()
```



6 Equations Différentielles Ordinaires

6.1 Problème de Cauchy

$f : \mathbb{R}_+ \times \mathbb{R} \to \mathbb{R}$ continue, $y_0 \in$ donné. On cherche $y : t \in I \subset_+ \to y(t) \in$ qui satisfait le problème suivant

$$\begin{cases} y'(t) = f(t, y(t)) & \forall t \in I \\ y(t_0) = y_0 \end{cases}$$

où $y'(t) = \dfrac{dy(t)}{dt}$.

Exemple Écrivez la discretisation par la méthode d'Euler progressive et rétrograde du problème de Cauchy

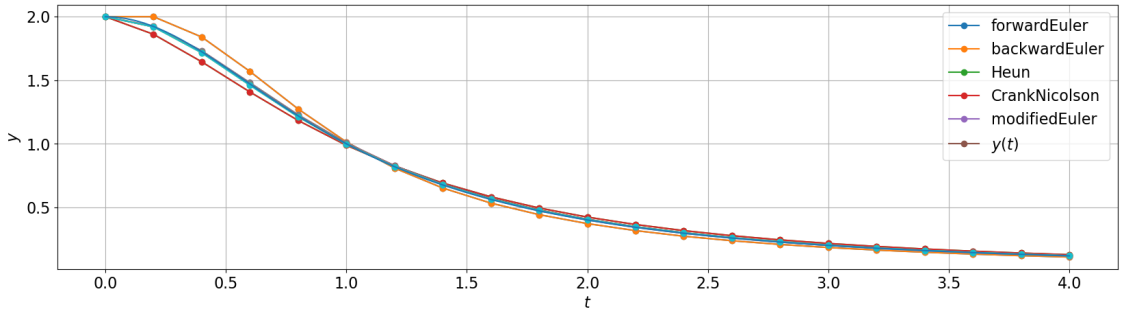
$$\begin{cases} y'(t) = -t\, y(t)^2 & \forall t \in [0, 4] \\ y(t_0) = 2 \end{cases}$$

La solution de ce problème est $y(t) = \frac{2}{1+t^2}$ Avec les méthodes de Euler Proressive et Rétrograde, Heun, Crank-Nicolson, et Euler modifié.

[121]:
```
# importing libraries used in this book
import numpy as np
import matplotlib.pyplot as plt

from OrdinaryDifferentialEquationsLib import␣
↪forwardEuler,backwardEuler,Heun,CrankNicolson,modifiedEuler
```

[122]:
```
f = lambda t,x :   -t*x**2
y0 = 2; tspan=[0, 4]
Nh = 20
```

```
for method in [forwardEuler,backwardEuler,Heun,CrankNicolson,modifiedEuler] :

    t, y = method(f, tspan, y0, Nh)
    plt.plot(t, y,'o-')
    plt.plot(t, y,'o-')

y = lambda t : 2/(1+t**2)
t = np.linspace(tspan[0],tspan[1],100)
plt.plot(t, y(t),'-')
# labels, title, legend
plt.xlabel('$t$'); plt.ylabel('$y$')
plt.
 ↪legend(['forwardEuler','backwardEuler','Heun','CrankNicolson','modifiedEuler','$y(t)$'])
plt.grid(True)
plt.show()
```



6.2 Euler Progressif

Ecrivez une fonction `forwardEuler` qui approche la solution du problème

$$y'(t) = f(t, y(t)),\ t \in (T_0, T_f), \quad y(0) = y_0,$$

en utilisant la méthode d'Euler progressif. L'entête de la fonction doit être la suivante:

```
def forwardEuler( fun, interval, y0, N ) :
    #  FORWARDEULER Solve differential equations using the forward Euler method.
    #  [T, U] = FORWARDEULER( FUN, INTERVAL, Y0, N ), with INTERVAL = [T0, TF],
    #  integrates the system of differential equations y'=f(t, y) from time T0
    #  to time TF, with initial condition Y0, using the forward Euler method on
    #  an equispaced grid of N intervals. Function FUN(T, Y) must return
    #  a column vector corresponding to f(t, y). Each row in the solution
    #  array U corresponds to a time returned in the column vector T.
```

Une fois écrite la fonction `forwardEuler`, utilisez les commandes suivantes pour calculer la solution:

```
f = lambda t,x : ( 2/15*x*(1-x/1000) ) # C=2/15 et B = 1000
tsp = [0,100]
```

```
y0 = 100; Nh = 25;
t25,u25 = forwardEuler(f,tsp,y0,Nh)

plt.plot(t25,u25,'o')
```

Approchez la solution de l'équation différentielle

$$y'(t) = \frac{2\,y}{15}(1 - y/1000),\ t \in (0, 100), \quad y(0) = 100,$$

avec $N_h = 25$.

Que se passe-t-il avec $N_h = 7$? Discutez les résultats obtenus.

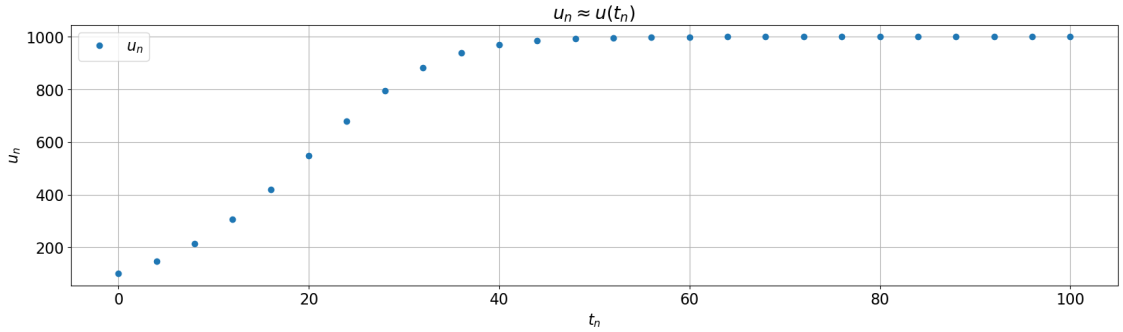
[123]:
```
# importing libraries used in this book
import numpy as np
import matplotlib.pyplot as plt
```

[124]:
```
def forwardEuler( fun, interval, y0, N ) :
    #  FORWARDEULER Solve differential equations using the forward Euler method.
    #  [T, U] = FORWARDEULER( FUN, INTERVAL, Y0, N ), with INTERVAL = [T0, TF],
    #  integrates the system of differential equations y'=f(t, y) from time T0
    #  to time TF, with initial condition Y0, using the forward Euler method on
    #  an equispaced grid of N intervals. Function FUN(T, Y) must return
    #  a column vector corresponding to f(t, y). Each row in the solution
    #  array U corresponds to a time returned in the column vector T.

    # time step
    h = ( interval[1] - interval[0] ) / N

    # time snapshots
    t = np.linspace( interval[0], interval[1], N+1 )

    # initialize the solution vector
    u = np.zeros(N+1)
    u[0] = y0

    # time loop (n=0,...,n, but array indeces in Matlab start at 1)
    for n in range(N) :
        u[n+1] = u[n] + h * fun( t[n], u[n] )

    return t, u
```

Nous pouvons alors résoudre l'exercice avec les commandes:

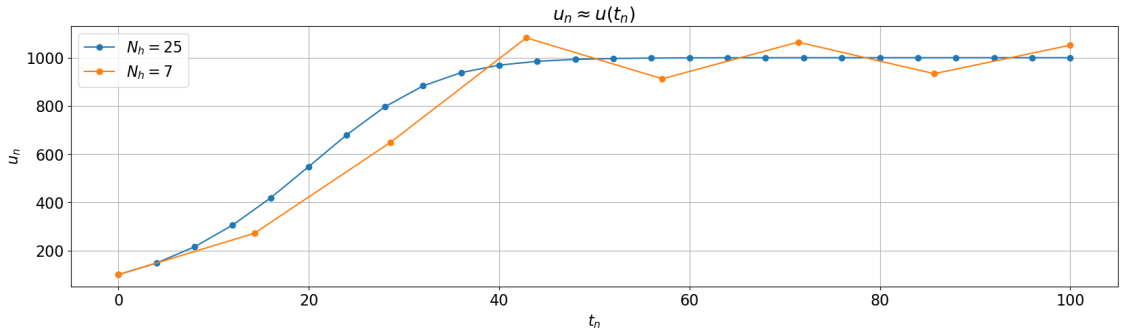
[125]:
```
f = lambda t,x : ( 2/15*x*(1-x/1000) ) # C=2/15 et B = 1000
tsp = [0,100]
y0 = 100; Nh = 25;
t25,u25 = forwardEuler(f,tsp,y0,Nh)

plt.plot(t25,u25,'o')
```

```
# labels, title, legend
plt.xlabel('$t_n$'); plt.ylabel('$u_n$'); #plt.title('data')
plt.legend(['$u_n$'])
plt.title('$u_n\\approx u(t_n)$')
plt.grid(True)
plt.show()
```



Pour résoudre le problème avec $N_h = 7$ et tracer le graphe de la solution, on utilise les commandes suivantes:

[126]:
```
Nh = 7
t7,u7 = forwardEuler(f,tsp,y0,Nh);

plt.plot(t25,u25,'o-')
plt.plot(t7,u7,'o-')

# labels, title, legend
plt.xlabel('$t_n$'); plt.ylabel('$u_n$'); #plt.title('data')
plt.legend(['$N_h=25$','$N_h=7$'])
plt.title('$u_n\\approx u(t_n)$')
plt.grid(True)
plt.show()
```



Solutions de l'équation différentielle $y'(t) = \frac{2y}{15}(1 - y/1000),\ t \in (0, 100), \quad y(0) = 100$ *pour diverses valeurs de* N_h.

On voit que la solution avec $N_h = 7$ est oscillante et ne tend pas vers 1000 lorsque $t \to \infty$: en effet, la condition de stabilité sur le pas de discretisation N_h n'est pas satisfaite.

[ ]:

6.3 Euler Retrograde

Ecrivez une fonction Matlab `backwardEuler` qui approche la solution du problème

$$y'(t) = f(t, y(t)),\ t \in (T_0, T_f), \quad y(0) = y_0,$$

en utilisant la méthode d'Euler progressif. L'entête de la fonction doit être la suivante:

```
def backwardEuler( fun, interval, y0, N ) :
    #  BACKWARDEULER Solve differential equations using the backward Euler method.
    #  [T, U] = BACKWARDEULER( FUN, INTERVAL, Y0, N ), with INTERVAL = [T0, TF],
    #  integrates the system of differential equations y'=f(t, y) from time T0
    #  to time TF, with initial condition Y0, using the backward Euler method on
    #  an equispaced grid of N intervals. Function FUN(T, Y) must return
    #  a column vector corresponding to f(t, y). Each row in the solution
    #  array U corresponds to a time returned in the column vector T.
    from scipy.optimize import fsolve
```

Vous disposez de la fonction `scipy.optimize.fsolve` pour résoudre une équation non-linéaire (`fsolve` cache plusieurs méthodes de resolution, comme Newton ou le point fixe).

Exemple d'utilisation de `fsolve`: on peut chercher le zéro de `F` près de `x0` en utilisant le code suivant:

```
from scipy.optimize import fsolve
a = 5; h = 0.1;
F = lambda x :  a + np.sin(x) - h*x;

x0 = a+h
zero = fsolve(F, x0)

print(f'F({zero[0]:5.2f}) = {F(zero)[0]:4.1e}')
```

Approchez la solution de l'équation différentielle

$$y'(t) = \frac{2\,y}{15}(1 - y/1000),\ t \in (0, 100), \quad y(0) = 100,$$

avec $N_h = 25$. Que se passe-t-il avec $N_h = 7$? Discuter les résultats obtenus.

[127]:
```
# importing libraries used in this book
import numpy as np
import matplotlib.pyplot as plt
```

[128]:
```
from scipy.optimize import fsolve
a = 5; h = 0.1;
F = lambda x :  a + np.sin(x) - h*x;

x0 = a+h
zero = fsolve(F, x0)

print(f'F({zero[0]:5.2f}) = {F(zero)[0]:4.1e}')
```

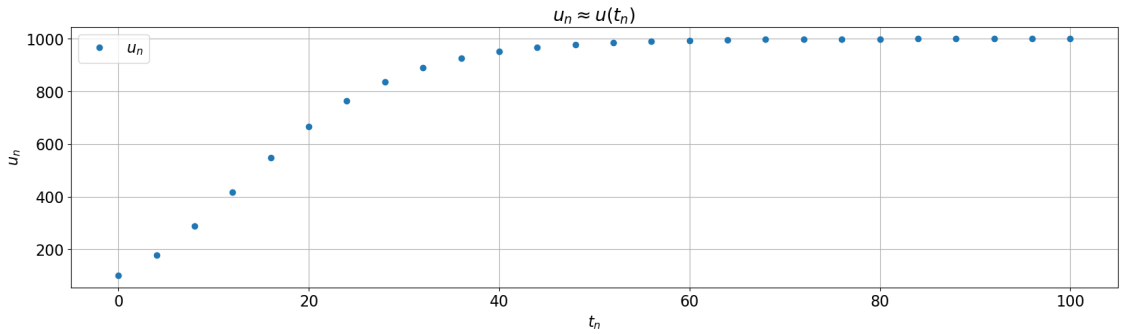
```
F(41.80) = -8.7e-13
```

[129]:
```
def backwardEuler( fun, interval, y0, N ) :
    #  BACKWARDEULER Solve differential equations using the backward Euler␣
 ↪method.
    #  [T, U] = BACKWARDEULER( FUN, INTERVAL, Y0, N ), with INTERVAL = [T0, TF],
    #  integrates the system of differential equations y'=f(t, y) from time T0
    #  to time TF, with initial condition Y0, using the backward Euler method on
    #  an equispaced grid of N intervals. Function FUN(T, Y) must return
    #  a column vector corresponding to f(t, y). Each row in the solution
    #  array U corresponds to a time returned in the column vector T.
    from scipy.optimize import fsolve

    # time step
    h = ( interval[1] - interval[0] ) / N

    # time snapshots
    t = np.linspace( interval[0], interval[1], N+1 )

    # initialize the solution vector
    u = np.zeros(N+1)
    u[0] = y0

    # time loop (n=0,...,n, but array indeces in Matlab start at 1)
    for n in range(N) :
        # non-linear function
        F = lambda x : u[n] + h * fun(t[n+1],x) - x
        # solve the non-linear equation using the built-in matlab function␣
 ↪"fsolve"
        # to compute u[n+1]
        u[n+1] = fsolve( F, u[n]);
        # u[n+1] = fsolve( F, u[n]+ h * fun( t[n], u[n] ));

        # NOTE:
        # in the call of fsolve, a more accurate initial guess is obtained
        # by replacing u[n] with the forward euler method:
        # u[n+1] = fsolve( F, u(n) + h * fun( t(n), u(n) ), options );
```

```
    return t, u
```

Nous pouvons alors résoudre l'exercice avec les commandes:

[130]:
```
f = lambda t,x : ( 2/15*x*(1-x/1000) ) # C=2/15 et B = 1000
tsp = [0,100]
y0 = 100; Nh = 25;
t25,u25 = backwardEuler(f,tsp,y0,Nh)

plt.plot(t25,u25,'o')

# labels, title, legend
plt.xlabel('$t_n$'); plt.ylabel('$u_n$'); #plt.title('data')
plt.legend(['$u_n$'])
plt.title('$u_n\\approx u(t_n)$')
plt.grid(True)
plt.show()
```

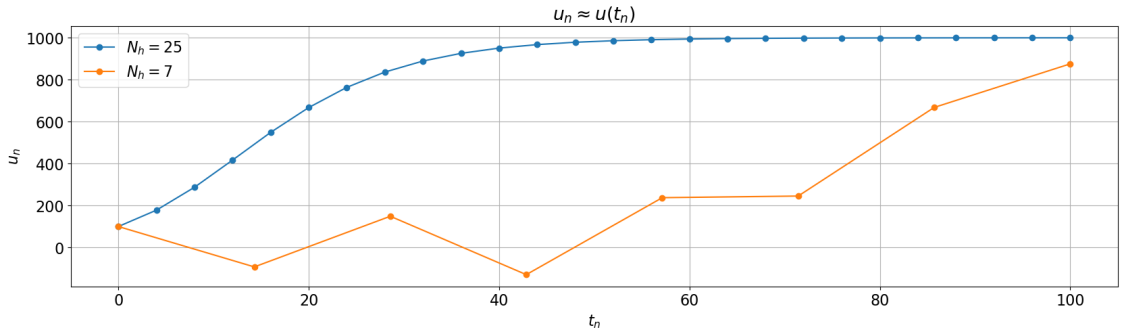


Pour résoudre le problème avec $N_h = 7$ et tracer le graphe de la solution, on utilise les commandes suivantes:

[131]:
```
Nh = 7
t7,u7 = backwardEuler(f,tsp,y0,Nh);

plt.plot(t25,u25,'o-')
plt.plot(t7,u7,'o-')

# labels, title, legend
plt.xlabel('$t_n$'); plt.ylabel('$u_n$'); #plt.title('data')
plt.legend(['$N_h=25$','$N_h=7$'])
plt.title('$u_n\\approx u(t_n)$')
plt.grid(True)
plt.show()
```

```
/Users/simone/opt/anaconda3/lib/python3.7/site-
packages/scipy/optimize/minpack.py:175: RuntimeWarning: The iteration is not
```

```
making good progress, as measured by the
  improvement from the last five Jacobian evaluations.
  warnings.warn(msg, RuntimeWarning)
/Users/simone/opt/anaconda3/lib/python3.7/site-
packages/scipy/optimize/minpack.py:175: RuntimeWarning: The iteration is not
making good progress, as measured by the
  improvement from the last ten iterations.
  warnings.warn(msg, RuntimeWarning)
```



Solutions de l'équation différentielle $y'(t) = \frac{2y}{15}(1 - y/1000)$, $t \in (0, 100)$, $\quad y(0) = 100$ *pour diverses valeurs de* N_h.

On voit que `fsolve` n'arrive pas à resoudre l'équation non-linéaire pour $N_h = 7$. Il faut chercher un meilleur `x0`.

Dans `backwardEuler`, replacez le `u[n]`

```
u[n+1] = fsolve( F, u[n]);
```

par la solution correspondante à la méthode d'Euler progressive, i.e.

```
u[n+1] = fsolve( F, u[n]+ h * fun( t[n], u[n] ));
```

Maintenant, pas seulement `fsolve` trouve une solution, mais en plus l'approximation de l'EDO ne présente pas d'oscillations.

[ ]:

[ ]:

6.4 Stabilité

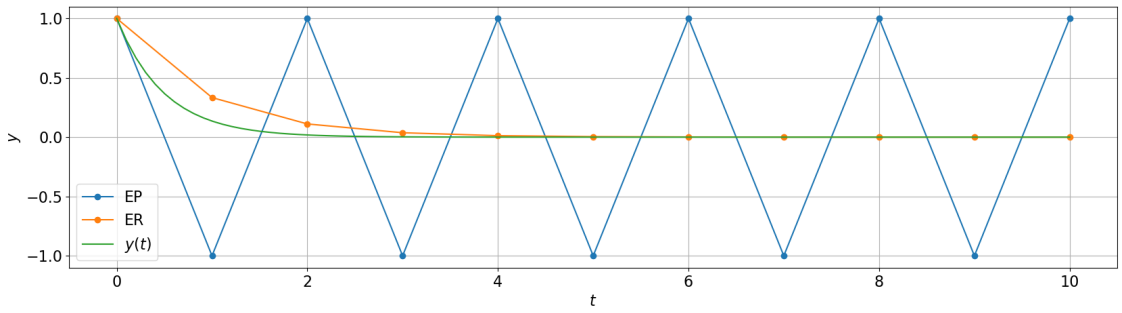
On considère le problème de Cauchy

$$\begin{cases} y'(t) = -2y(t), \quad t > 0 \\ y(0) = 1 \end{cases}$$

La solution exacte de ce problème est $y(t) = e^{-2t}$

Résolvez ce problème par les méthodes d'Euler Progressive et Rétrograde sur l'intervalle $[0, 10]$ avec un pas de temps $h = 0.9$ et 1.1.

[132]:
```
l = -2
f = lambda t,x :   l*x
y0 = 1; tspan=[0, 10]
h = 1.1; Nh = np.ceil((tspan[1] - tspan[0])/h).astype(int)
t_EP, y_EP = forwardEuler(f, tspan, y0, Nh)
t_ER, y_ER = backwardEuler(f, tspan, y0, Nh)

plt.plot(t_EP, y_EP,'o-')
plt.plot(t_ER, y_ER,'o-')

y = lambda t : np.exp(l*t)
t = np.linspace(tspan[0],tspan[1],100)
plt.plot(t, y(t),'-')
# labels, title, legend
plt.xlabel('$t$'); plt.ylabel('$y$')
plt.legend(['EP','ER','$y(t)$'])
plt.grid(True)
plt.show()
```



6.5 Convergence

On considère le problème de Cauchy

$$\begin{cases} y'(t) = -y(0.1 - \cos(t)), \quad t > 0 \\ y(0) = 1 \end{cases}$$

Resolvez ce problème par les méthodes d'Euler progressive et de Heun sur l'intervalle $[0, 12]$ avec un pas de temps $h = 0.4$.

La solution exacte est $y(t) = e^{-0.1t+\sin(t)}$. On remarque que la solution obtenue par la méthode de Heun est beaucoup plus précise que celle d'Euler progressive. Par ailleurs, on peut voir que si on réduit le pas de temps, la solution obtenue par la méthode d'Euler progressive s'approche de la solution exacte.

[133]:
```
f = lambda t,y : (np.cos(t) - 0.1)*y

tspan = [0,12]; y0 = 1;
h = 0.4; Nh = np.ceil((tspan[1] - tspan[0])/h).astype(int)

t_EP, y_EP = forwardEuler(f, tspan, y0, Nh)
t_H, y_H = Heun(f, tspan, y0, Nh)

plt.plot(t_EP, y_EP,'o-')
plt.plot(t_H, y_H,'o-')

y = lambda t : np.exp(-0.1*t+np.sin(t))
t = np.linspace(tspan[0],tspan[1],100)
plt.plot(t, y(t),'-')
# labels, title, legend
plt.xlabel('$t$'); plt.ylabel('$y$')
plt.legend(['EP','Heun','$y(t)$'])
plt.grid(True)
plt.show()
print("Figure: Approximation par le méthodes de Euler Retrograde et Heun.")
```

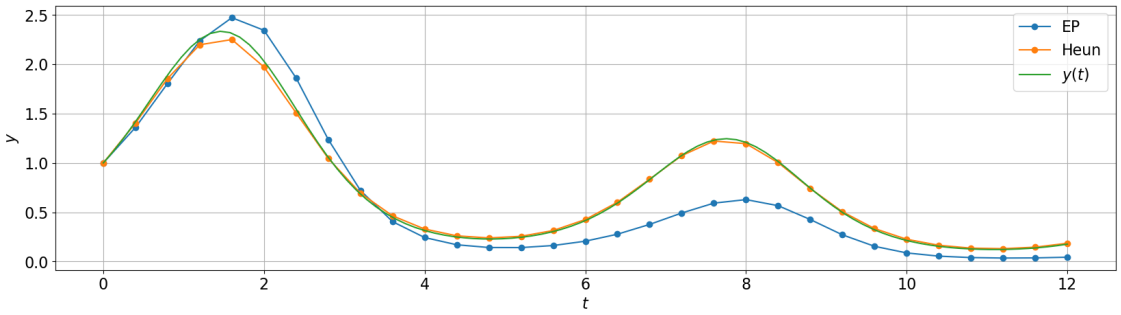


Figure: Approximation par le méthodes de Euler Retrograde et Heun.

[134]:
```
tspan = [0,12]; y0 = 1;
NhRange = [30, 50, 100, 500]
for Nh in NhRange :
    t, y = forwardEuler(f,tspan,y0,Nh)
    plt.plot(t, y,'-')

yt = lambda t : np.exp(-0.1*t+np.sin(t))
t = np.linspace(tspan[0],tspan[1],100)
plt.plot(t, yt(t),':')
# labels, title, legend
plt.xlabel('$t_n$'); plt.ylabel('$u_n$')
plt.legend(NhRange+['$y(t)$'])
```

```
plt.title('$u_n\\approx u(t_n)$')
plt.grid(True)
plt.show()
print("Figure: Solutions obtenues par la méthode d'Euler progressive pour␣
 ↪différents pas de temps.")
```

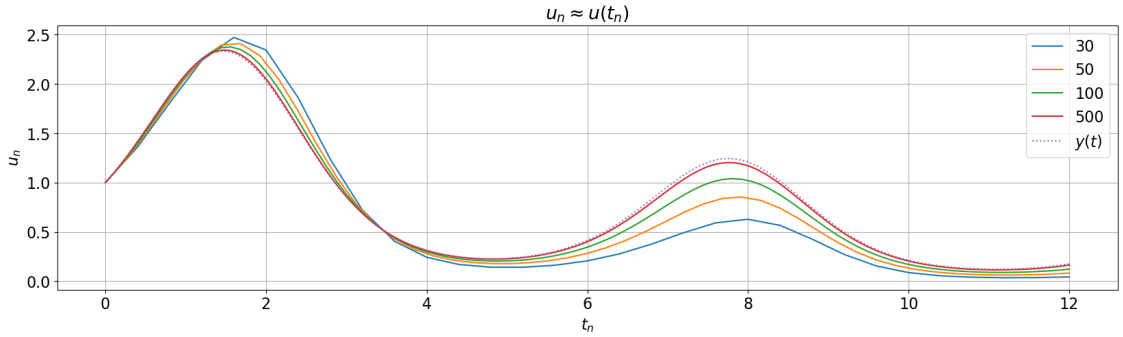


Figure: Solutions obtenues par la méthode d'Euler progressive pour différents pas de temps.

On veut, maintenant, estimer l'ordre de convergence de ces deux méthodes. Pour cela, on résout le problème avec différents pas de temps et on compare les résultats obtenus à l'instant $t = 6$ avec la solution exacte.

[135]:
```
tspan=[0,6]
NhRange = [30, 50, 100, 500]
errEP = []
errH = []
# Solution at end time
y6 = yt(tspan[1])
for Nh in NhRange :
    # Forward Euler
    t, y = backwardEuler(f,tspan,y0,Nh)
    # Error at the end of the simulation
    errEP.append( np.abs(y6 - y[-1] ) )

    # Heun
    [t, y] = Heun(f, tspan, y0, Nh);
    # Error at the end of the simulation
    errH.append( np.abs(y6 - y[-1] ) )

h = (tspan[1] - tspan[0])/np.array(NhRange)
plt.loglog(h,errEP,'o-b',h,errH,'o-r')
plt.loglog(h,h*(errEP[0]/h[0]),':',h,(h**2*(errH[0]/h[0]**2)),':')
plt.xlabel('$h$'); plt.ylabel('$|y(6)-u_{N_h}|$')
plt.legend(['EP','Heun','$h$','$h^2$'])
```

```
plt.title('Decay of the error')
plt.grid(True)
plt.show()

print("Figure: Erreurs en échelle logarithmique commises par les méthodes" +
      " d'Euler progressive et de Heun dans le calcul de y(6).")
```

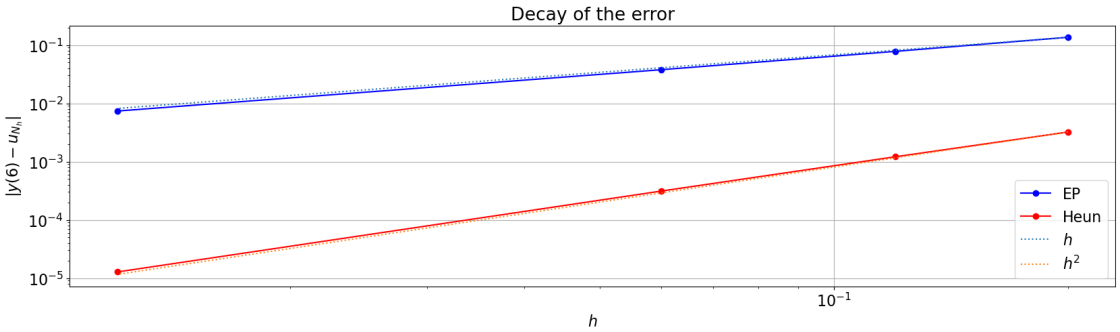


Figure: Erreurs en échelle logarithmique commises par les méthodes d'Euler progressive et de Heun dans le calcul de y(6).

La figure montre, en échelle logarithmique, les erreurs commises par les deux méthodes en fonction de h. On voit bien que la méthode d'Euler progressive converge à l'ordre 1 tandis que celle de Heun à l'ordre 2.

[ ]:

6.6 Stabilité

On considère le problème de Cauchy

$$\begin{cases} y'(t) = -2y(t), \quad t > 0 \\ y(0) = 1 \end{cases}$$

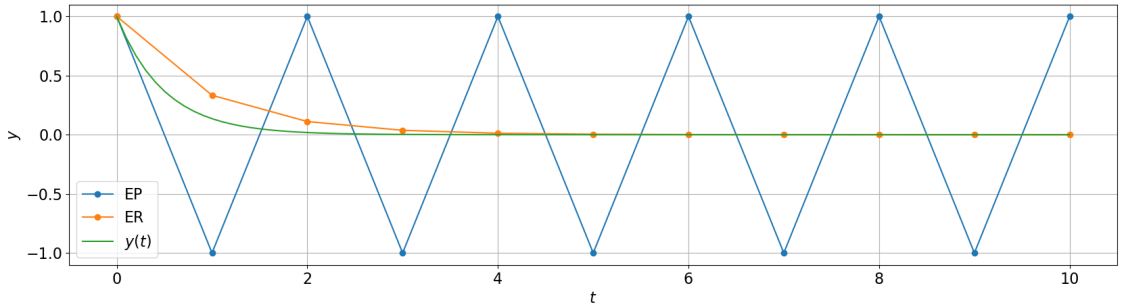
La solution exacte de ce problème est $y(t) = e^{-2t}$

Résolvez ce problème par les méthodes d'Euler Progressive et Rétrograde sur l'intervalle $[0, 10]$ avec un pas de temps $h = 0.9$ et 1.1.

[136]:
```
# importing libraries used in this book
import numpy as np
import matplotlib.pyplot as plt

from OrdinaryDifferentialEquationsLib import␣
 ↪forwardEuler,backwardEuler,Heun,CrankNicolson,modifiedEuler
```

[137]:
```
l = -2
f = lambda t,x :   l*x
```

```
y0 = 1; tspan=[0, 10]
h = 1.1; Nh = np.ceil((tspan[1] - tspan[0])/h).astype(int)
t_EP, y_EP = forwardEuler(f, tspan, y0, Nh)
t_ER, y_ER = backwardEuler(f, tspan, y0, Nh)

plt.plot(t_EP, y_EP,'o-')
plt.plot(t_ER, y_ER,'o-')

y = lambda t : np.exp(l*t)
t = np.linspace(tspan[0],tspan[1],100)
plt.plot(t, y(t),'-')
# labels, title, legend
plt.xlabel('$t$'); plt.ylabel('$y$')
plt.legend(['EP','ER','$y(t)$'])
plt.grid(True)
plt.show()
```



[ ]: