TP ALL

TP1

TP 1 Méthodes de quadratures 1.

Sommaire

---

haut 1. 2. 3. 4. bas.

1. Numpy et les erreurs d'arrondi

Nous chargeons deux bibliothèques, *numpy* pour les calculs et *matplotlib.pyplot* pour les représentations graphiques. Nous modifions aussi les paramètres fixant la taille des figures et la taille du texte dans ces figures pour les agrandir (inutile si vous avez de bons yeux).

In [ ]:

```
import numpy as np
import matplotlib.pyplot as plt
plt.rcParams.update({'font.size': 18})
plt.rcParams.update({'figure.figsize' : (8, 6)})
```

Voici des exemples de leur utilisation. Les lignes suivantes affichent des valeurs approchées de e^2 et 2π.

In [ ]:

```
edeux, tau = np.exp(2), 2*np.pi
print("valeurs approchées du carré de e :", edeux)
print("et du double de pi :", tau)
```

Nous définissons les fonction $f_1 : x \mapsto e^{\sin x}$ et $f_2 : x \mapsto 1 + \cos x$ et en nous en faisons des représentations graphiques sur l'intervalle $]0, 1[$.

In [ ]:

```
def f1(x):
    return np.exp(np.sin(x))
f2 = lambda x: 1 + np.cos(x)

N=50
X=np.linspace(0,1,N + 1) #produit un tableau de N+1 points (0,h,2h,...,Nh=1)
f1X = f1(X) # renvoie le tableau construit en appliquant f1 à chaque élément de X
        # c'est possible car on a utilisé des fonctions numpy en définissant f1
f2X = f2(X)
plt.plot(X, f1X, 'b.', label="f1")
```

```
plt.plot(X, f2X, 'r-', label="f2")
plt.xlabel('x')
plt.ylabel('f(x)')
plt.legend()
plt.show()
```

Erreurs d'arrondis.

Les calculs avec numpy sont des calculs approchés. Par exemple, il n'est pas certain que les calculs $(e+\pi)\sin(1)$ **et** $e\sin(1)+\pi\sin(1)$ **donnent le même résultat**

In [ ]:

```
c1= (np.exp(1) + np.pi)*np.sin(1)
c2= np.exp(1)*np.sin(1) + np.pi*np.sin(1)
c1 - c2
```

Par défaut, les calculs se font en double précision, le plus petit nombre strictement plus grand que 1 **représentable dans ce cadre est** $1+\varepsilon$ **avec** ε **de l'ordre de** 10^{-15}**.**

En pratique, cela veut dire qu'on ne peut pas distinguer entre 1 **et** $1+x$ **si** $|x|<\varepsilon$**.**

Cette précision est largement suffisante dans la plupart des cas mais il faut faire attention : les erreurs peuvent se cumuler et se combiner jusqu'à polluer complètement un résultat.

Le cas le plus catastrophique est celui de la division par un petit nombre.

On considère l'évaluation de la fonction définie pour $x>0$ **par**

$$f(x):=\frac{1}{1-\sqrt{1-x^2}}.$$

Si on prend $x<\sqrt{\varepsilon}$**, l'ordinateur va remplacer** $1-x^2$ **par** 1 **et devra effectuer une division par** $1-\sqrt{1}=0$ **ce qui conduit à une erreur. Dans ce cas précis, on peut résoudre le problème en multipliant en haut et en bas par la quantité conjuguée** $1+\sqrt{1-x^2}$**, on a**

$$f(x)=\frac{1+\sqrt{1-x^2}}{x^2},$$

qui ne posera pas de problème de calcul (voir ci-dessous).

In [ ]:

```
f = lambda x:1/(1 - np.sqrt(1 - x**2))
x=1e-9
f(x)
```

In [ ]:

```
f = lambda x:(1 + np.sqrt(1 - x**2))/x**2
x=1e-9
f(x)
```

Il convient de garder en tête ces phénomènes quand on effectue des calculs approchés.

haut 1. 2. 3. 4. bas.

2. Représentation graphique de l'erreur pour l'étude des méthodes de calcul approché

En général, on dit qu'une approximation I_h **d'une quantité** I **est d'odre** p **par rapport au petit paramètre** $h>0$ **si il existe une constante** $C>0$ **telle que**

$$e_h := |I - I_h| \leq Ch^{p+1} \text{ pour } 0 < h < 1,$$

et que pour tout $q > p$, il n'existe pas de constante C' telle que

$$e_h \leq C' h^{q+1} \text{ pour } 0 < h < 1.$$

Très souvent, on a en fait une équivalence

$$e_h \overset{h \downarrow 0}{\sim} Ch^{p+1}.$$

Quand on fait une représentation graphique, il est difficile de distinguer à l'oeil nu entre (par exemple) les fonctions $h \mapsto h^2$ et $h \mapsto h^3$ sur l'intervalle $]0,1[$. Elles sont toutes deux convexes, croissantes avec une dérivée nulle en 0.

In [ ]:

```
h=np.linspace(1/100,1,100) # on prend h= 0.01, 0.02,
                           #           ..., 0.99, 1
plt.plot(h,h**2,label=r'$h\to h^2$')
plt.plot(h,h**3,label=r'$h\to h^3$')
plt.xlabel('h')
plt.legend()
plt.show()
```

Pour observer graphiquement l'ordre, on prend le logarithme de (1) et on obtient

$$\log(e_h) = (p+1)\log(h) + \log(C) + \eta(h) \quad \text{avec } \eta(h) \overset{h \downarrow 0}{\longrightarrow} 0.$$

Si on pose $t = \log(h)$ et $g(t) = \log(e_h)$, on a

$$g(t) = (p+1)t + \log(C) + \zeta(t) \quad \text{avec } \zeta(t) \overset{t \downarrow -\infty}{\longrightarrow} 0.$$

La fonction g admet donc pour asymptote la fonction affine $t \mapsto (p+1)t + \log(C)$ quand $t \to -\infty$. Le tracé de quelques points $(t_i, g(t_i))$ dans le plan permet de vérifier qu'ils sont approximativement alignés sur une droite.

In [ ]:

```
h=np.linspace(1/100,1,100) # on prend h= 0.01, 0.02,
                           #           ..., 0.99, 1
plt.plot(h,h**2,'go-',label=r'$h\to h^2$')
plt.plot(h,h**3,'bv-',label=r'$h\to h^3$')
plt.loglog()
plt.xlabel('h')
plt.legend()
plt.show()
```

In [ ]:

```
h=2**(-np.linspace(0,10,11))# on prend h= 1, 1/2, 1/4,
                            # ..., 1/1024
plt.plot(h, h**2, 'go-', label=r'$h\to h^2$')
plt.plot(h, h**3, 'bv-', label=r'$h\to h^3$')
plt.loglog()
plt.grid(True, which="both")
plt.legend()
plt.show()
```

[haut](#) [1.](#) [2.](#) [3.](#) [4.](#) [bas.](#)

3. Premières méthodes de quadrature

Nous prenons une fonction $f: [a,b] \to \mathbf{R}$
que nous supposons continue. Nous supposons que nous savons évaluer $f(x)$
(au moins de manière assez précise). Nous voulons calculer une bonne approximation de

(du moins de manière assez précise). Nous voulons calculer une bonne approximation de

$$I(f) := \int_a^b f(x)\,dx.$$

Pour cela, on se donne un entier $N \geq 1$
, on pose $h = (b-a)/N$
et on construit la discrétisation de $[a, b]$
définie par

$$x_i = a + ih, \qquad \text{pour } i = 0, \ldots, N.$$

Le nombre h
est appelé le ***pas de la discrétisation***.

On écrit alors $I(f)$
sous la forme

$$I(f) = \sum_{i=0}^{N-1} \int_{x_i}^{x_{i+1}} f(x)\,dx.$$

Quand f
est continue et $x_{i+1} - x_i$
est petit, $f(x)$
est proche de $f(y)$
pour $x, y \in [x_i, x_{i+1}]$
. On a donc

$$\int_{x_i}^{x_{i+1}} f(x)\,dx \simeq (x_{i+1} - x_i) f(y),$$

pour n'importe quel choix de y
dans $[x_i, x_{i+1}]$
.

En se basant sur cette idée, nous proposons d'étudier les méthodes suivantes pour approcher $I(f)$
: la méthode des rectangles à gauche, des rectangles à droite, des trapèzes et du point milieu. Avec ces méthodes on approche $\int_{x_i}^{x_{i+1}} f(x)\,dx$
par, respectivement,

$$I_i^{rg} := (x_{i+1} - x_i) f(x_i), \quad I_i^{rd} := (x_{i+1} - x_i) f(x_{i+1}), \quad I_i^{tr} := (x_{i+1} - x_i) \frac{f(x_i) + f(x_{i+1})}{2}, \quad I_i^{pm} := (x_{i+1} - x_i) f\left(\frac{x_i + x_{i+1}}{2}\right).$$

Pour chaque méthode $meth \in \{rg, rd, tr, pm\}$
, on note

$$J_h^{meth}(f) := \sum_{i=0}^{N-1} I_i^{meth}.$$

Pour évaluer ces méthodes, nous choisirons des fonctions f
pour lesquelles on a une formule explicite pour $I(f)$
. En pratique, on aura $I(f) = F(b) - F(a)$
où F
est une primitive de f
qu'on déterminera. L'erreur est

$$e_h^{meth}(f) := |J_h^{meth}(f) - I(f)|.$$

Exercice 1. Programmer les quatre fonctions ***quadrature_meth*** (où $meth \in \{rg, rd, tr, pm\}$
) qui prennent en donnée une fonction f

, qui prennent en donnée une fonction f
, deux réels a
, b
et un entier $N \geq 1$
et qui renvoient la quantité

$$J_h^{meth}(f),$$

avec $h = (b-a)/N$
. Dans la suite ces fonctions sont appelées *méthodes*.

Tester ces fonctions avec $f: x \in [0,1] \mapsto x^2 \in \mathbf{R}$
.

In [ ]:

```
### SOLUTION
def quadrature_rg0(f, a, b, N):
    x = np.linspace(a, b, N + 1)
    h=(b - a)/N
    S=0
    for  i in range(N):
        S+=h*f(x[i])
    return S

def quadrature_rg(f, a, b, N):
    x, h = np.linspace(a, b, N + 1), (b - a)/N
    return np.sum(f(x[:-1]))*h
```

In [ ]:

```
f =lambda x:x**2
a,b=0,1
N=10

print("Avec quadrature_rg0, I= ", quadrature_rg0(f,a,b,N))
print("Avec quadrature_rg, I= ", quadrature_rg(f,a,b,N))
print(quadrature_rg0(f,a,b,N)-quadrature_rg(f,a,b,N))
```

In [ ]:

```
def quadrature_rd(f, a, b, N):
    x, h = np.linspace(a, b, N + 1), (b - a)/N
    return np.sum(f(x[1:]))*h

def quadrature_tr(f, a, b, N):
    x, h = np.linspace(a, b, N + 1), (b - a)/N
    return np.sum((f(x[:-1])+f(x[1:]))/2)*h

def quadrature_pm(f, a, b, N):
    x, h = np.linspace(a, b, N + 1), (b - a)/N
    return np.sum(f((x[:-1]+x[1:])/2))*h
```

In [ ]:

```
Iex=1/3;
N=100
meths=[quadrature_rg,quadrature_rd,quadrature_tr,quadrature_pm]

for meth in meths:
    erreur=np.abs(Iex-meth(f,a,b,N))
    print("l'erreur  avec " + meth.__name__ + " vaut",erreur)
```

Exercice 2. **Programmer une fonction *calcul_erreurs* qui prend en entrée une fonction f**
, deux réel a
, b
, un autre réel I
, une méthode et un entier k
et qui renvoie deux tableaux de réels H

et E
de taille k
définis par

$$H[j] = (b-a)/2^{j+1} \quad \text{pour } j = 0, \ldots, k-1,$$

$$E[j] = \left| J^{meth}_{H[j]}(f) - I \right| \quad \text{pour } j = 0, \ldots, k-1.$$

Bien sûr, on utilisera cette fonction avec la valeur exacte $I = \int_a^b f$

.

In [ ]:

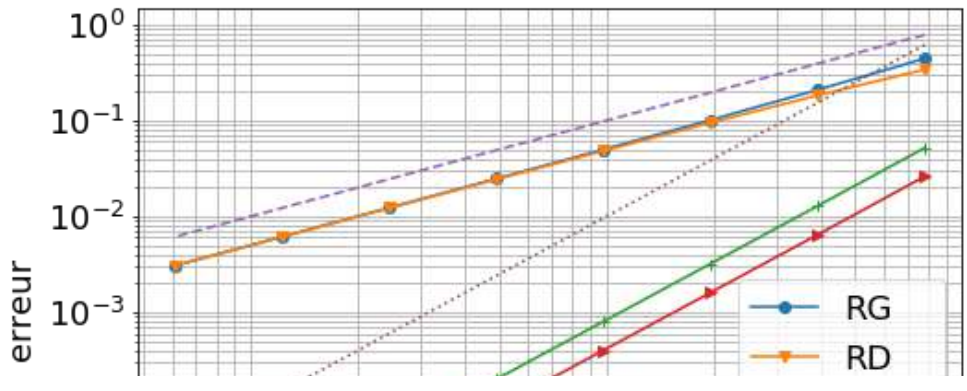
```
### SOLUTION
def calcul_erreurs(f, a, b, I, methode, k):
    H, E, N, h = np.zeros(k), np.zeros(k), 1, b - a
    for j in range(k):
        N *= 2
        h /= 2
        E[j]= np.abs(methode(f, a, b, N) - I)
        H[j]= h
    return H, E
```

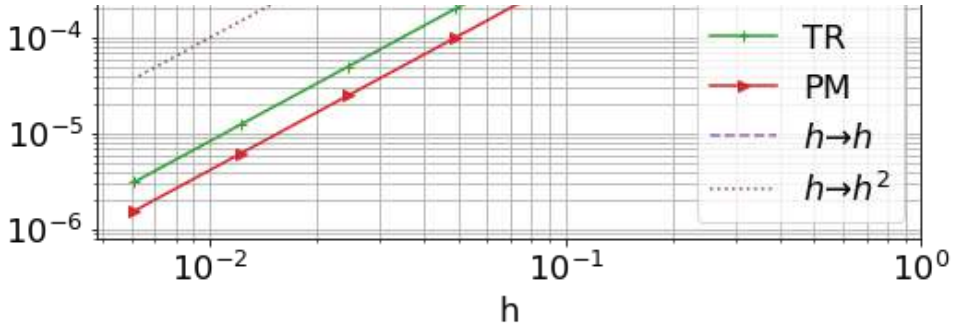
Exercice 3. Tester les quatre méthodes avec la fonction $f: x \mapsto \sin x$
et $[a, b] = [0, \pi/2]$
. On utilisera la fonction *calculs_erreurs*
et on fera une représentation de l'erreur en log
-log
en fonction du pas h
.

Qu'en déduisez vous sur l'ordre des méthodes ?

In [16]:

```
### SOLUTION
a, b, f, I = 0, .5*np.pi, lambda x: np.sin(x), 1
k = 8
meths=[quadrature_rg,quadrature_rd,quadrature_tr,quadrature_pm]
name=['RG','RD','TR','PM']
marker=['o-','v-','+-','>-']
j=0
for meth in meths:
    H, E = calcul_erreurs(f, a, b, I, meth, k)
    plt.plot(H, E, marker[j], label=name[j])
    j+=1
plt.plot(H,H,'--',label=r'$h\to h$')
plt.plot(H,H**2,':',label=r'$h\to h^2$')
plt.legend()
plt.xlabel("h")
plt.ylabel("erreur")
plt.loglog()
plt.grid(True, which="both", ls='-')
plt.show()
```





Solution : ...

[haut](#) [1.](#) [2.](#) [3.](#) [4.](#) [bas.](#)

4. Cas pathologiques

fonction peu régulières

Exercice 4. Tester les trois dernières méthodes avec $[a, b] = [0, 1]$
et les fonctions définies par:

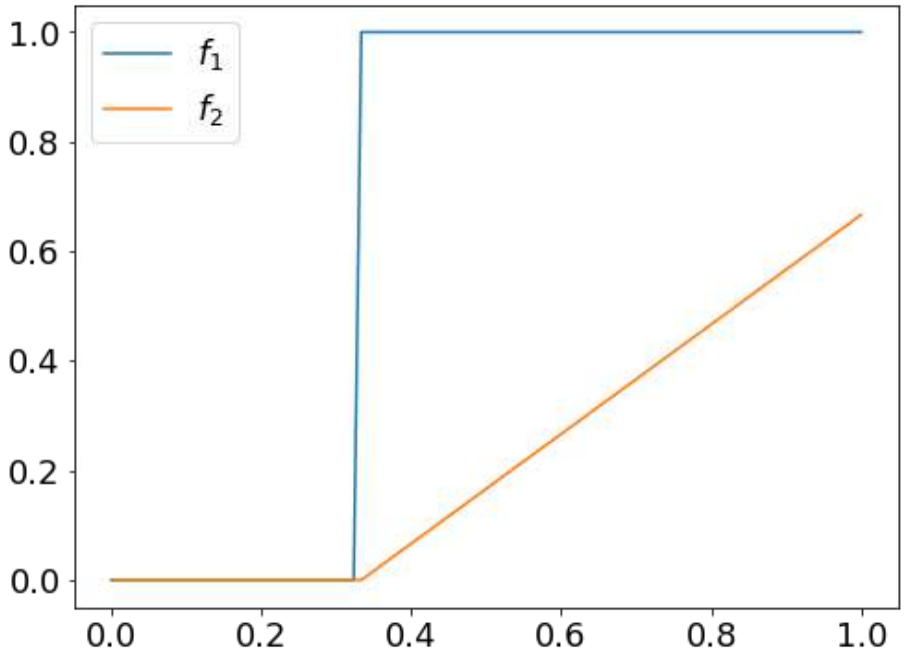
$$f_1(x) = \begin{cases} 0 & \text{si } x < 1/3, \\ 1 & \text{si } x \geq 1/3, \end{cases} \qquad f_2(x) = \begin{cases} 0 & \text{si } x < 1/3, \\ x - 1/3 & \text{si } x \geq 1/3. \end{cases}$$

(Les fonctions python correpondantes sont données ci-dessous.)

Quels sont les odres de convergence observés ?

In [17]:

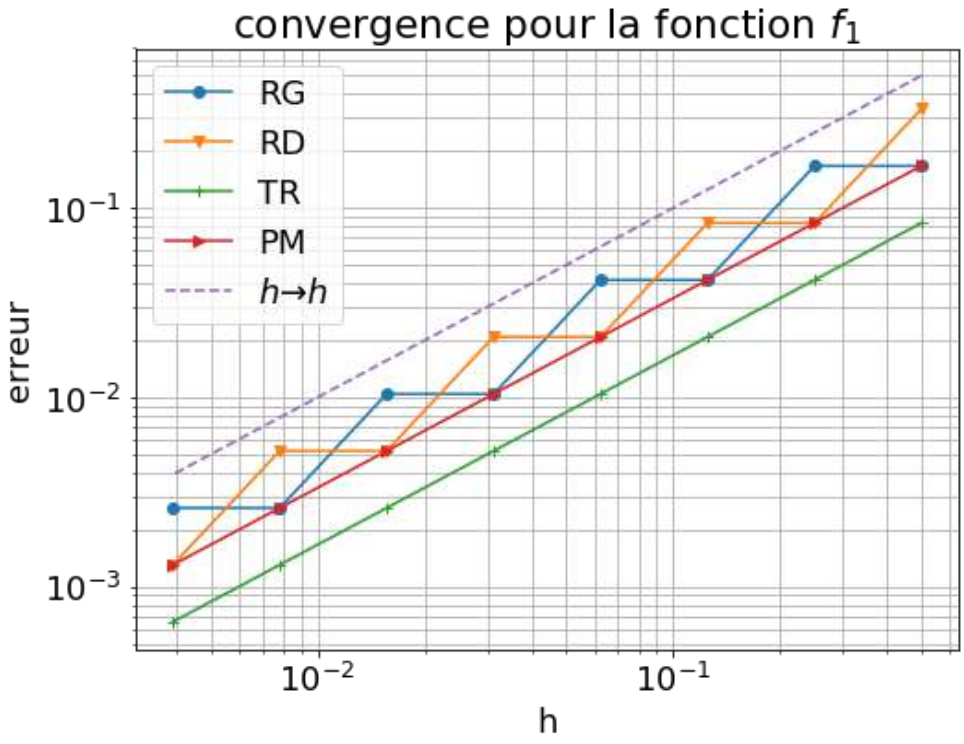
```
f1 = lambda x: np.where(x>=1/3, 1, 0)
f2 = lambda x: np.where(x>=1/3, x - 1/3, 0)

x=np.linspace(0,1,100)
plt.plot(x,f1(x),label=r'$f_1$')
plt.plot(x,f2(x),label=r'$f_2$')
plt.legend()
plt.show()
```

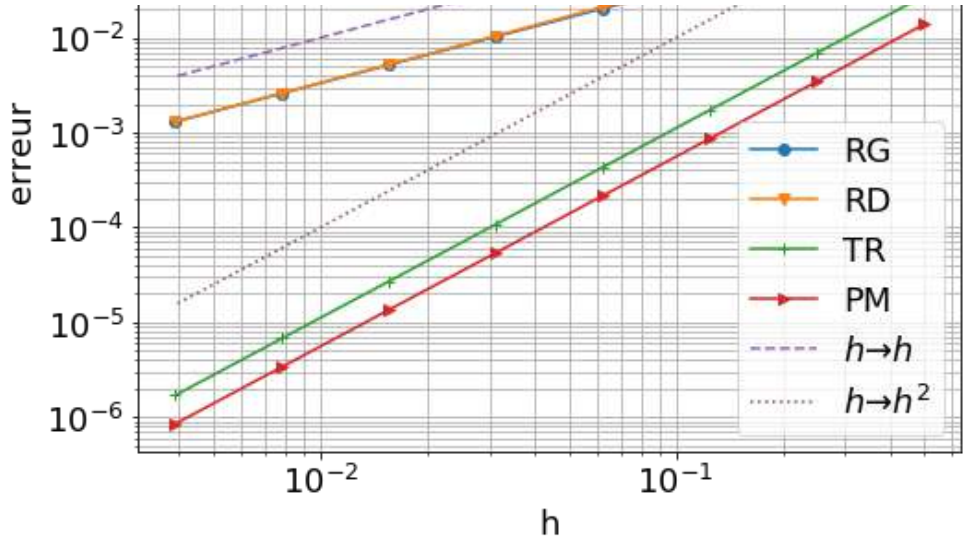


In [20]:

```
### SOLUTION
```

```
a, b = 0, 1
k = 8
I1, I2 = 2/3, 2/9
meths==[quadrature_rg,quadrature_rd,quadrature_tr,quadrature_pm]
name=['RG','RD','TR','PM']
marker=['o-','v-','+-','>-']
j=0

plt.figure(1)
for meth in meths:
    H, E = calcul_erreurs(f1, a, b, I1, meth, k)
    plt.plot(H, E, marker[j], label=name[j])
    j+=1
plt.plot(H,H,'--',label=r'$h\to h$')
plt.loglog()
plt.legend()
plt.xlabel("h")
plt.ylabel("erreur")
plt.grid(True, which="both", ls='-')
plt.title("convergence pour la fonction $f_1$")
plt.show()

plt.figure(2)
j=0
for meth in meths:
    H, E = calcul_erreurs(f2, a, b, I2, meth, k)
    plt.plot(H, E, marker[j], label=name[j])
    j+=1
plt.plot(H,H,'--',label=r'$h\to h$')
#plt.plot(H,H,'--',label='H')
plt.plot(H,H**2,':',label=r'$h\to h^2$')
#plt.plot(H,H**2,':',label='H2')
plt.loglog()
plt.legend()
plt.xlabel("h")
plt.ylabel("erreur")
plt.grid(True, which="both", ls='-')
plt.title("convergence pour la fonction $f_2$")
plt.show()
```





Solution : ...

fonctions périodiques régulières

On considère maintenant l'intégration d'une fonction périodique sur une de ces périodes. On prend tout d'abord la fonction de période 2π
et de classe C^∞
sur $\mathbf{R}$
,

$$f_{per}(x) = \cos^2(3(x-3)).$$

On a

$$I(f_{per}) := \int_0^{2\pi} \cos^2(3(x-3))\,dx = \pi.$$

(***Par*** 2π***-périodicité de*** $y \mapsto \cos y$
on a que

$$g(a) := \int_0^{2\pi} \cos^2(3x+a)\,dx$$

est indépendante de a***, en particulier, pour tout*** a***, on a***

$$2g(a) = g(a) + g\left(a - \frac{\pi}{2}\right) = \int_0^{2\pi} \left[\cos^2(3x+a) + \cos^2\left(3x + a - \frac{\pi}{2}\right)\right] dx.$$

Comme $\cos(y - \pi/2) = \sin y$
et $\cos^2 y + \sin^2 y = 1$
, ***on déduit*** $2g(a) = 2\pi$
soit $g(a) = \pi$
. ***En particulier,*** $g(-9) = I(f_{per}) = \pi.$
)

Exercice 5. Tester la méthode des rectangles à droite avec f_{per}
et $[a, b] = [0, 2\pi]$
(la fonction python ***fper*** **est définie ci-dessous). Quel est l'ordre de convergence observé ?**

In [ ]:

```
fper = lambda x: np.cos(3*(x-3))**2
```

In [ ]:

```
### SOLUTION
a, b = 0, 2*np.pi
I = np.pi
k = 5
H, E = calcul_erreurs(fper, a, b, I, quadrature_rd, k)
plt.loglog(H, E, 'o-', label="rd")
plt.legend()
plt.xlabel("h")
plt.ylabel("err")
plt.grid(True, which="both", ls='-')
plt.show()
```

Solution : (suite) La convergence est tellement rapide qu'on arrive à la précision machine dès $N = 4$. Dans ce cas (fonctions périodiques régulières), pour tout $p \geq 0$, il existe $C_p(f) > 0$ tel que_

$$|e_h| \leq C_p(f) h^{p+1}.$$

On dit que la convergence est d'ordre infini.

Exercice 6. Faire la même étude avec $[a, b] = [0, \pi/2]$ et la fonction $\pi/2$ -périodique définie sur $[0, \pi/2]$ par

$$f_{per2}(x) = \sin(x)(\pi/2 - x).$$

Cette fonction est continue sur $\mathbf{R}$ mais pas C^1 .

On pourra vérifier calmement que $I_{0,\pi/2}(f_{per2}) = \frac{\pi}{2} - 1$.

In [ ]:

```
### SOLUTION
fper2 = lambda x: np.sin(x)*(np.pi/2-x)
a, b, I = 0, np.pi/2, np.pi/2 - 1
k = 8
H, E = calcul_erreurs(fper2, a, b, I, quadrature_rd, k)
plt.loglog(H, E, 'o-', label="rd")
plt.loglog(H,H**2,'--',label=r'$h\to h^2$')
plt.legend()
plt.xlabel("h")
plt.ylabel("err")
plt.grid(True, which="both", ls='-')
plt.show()
```

Solution : ...

haut 1. 2. 3. 4. bas.

TP2

TP 2 Méthodes de quadratures 2 (méthodes d'ordre élevé)

Sommaire

Nous chargons les deux bibliothèques, *numpy* et *matplotlib.pyplot*.

In [1]:

```
import numpy as np
import matplotlib.pyplot as plt
plt.rcParams.update({'font.size': 18})
plt.rcParams.update({'figure.figsize' : (8, 6)})
```

Dans les deux premières parties, on étudie les formule de quadratures pour approcher

$$I(f) = \int_a^b f(s)\, ds.$$

On se donne une discrétisation de l'intervalle $[a, b]$**,**

$$x_0 = a < x_1 < \cdots < x_{N-1} < x_N = b.$$

On note $x = (x_0, x_1, \ldots, x_N)$**.**

Pour $i = 0, \ldots, N-1$**, on note** $c_i := (x_i + x_{i+1})/2$**,** $h_i = x_{i+1} - x_i$ **et on fait le changement de variable** $s = c_i + t h_i/2$**, pour le calcul de** $\int_{x_i}^{x_{i+1}} f(s)\, ds$ **et on obtient**

$$\int_{x_i}^{x_{i+1}} f(s)\, ds = \frac{h_i}{2} \int_{-1}^{1} f\Big(\varphi_i(t)\Big)\, dt,$$

où pour $i = 0, \ldots, N-1$**,** $\varphi_i(t) = c_i + (h_i/2)t$**.**

Pour approcher l'intégrale de -1 **à** 1 **dans le membre de droite, on utilise une formule de quadrature approchée:**

$$\int_{-1}^{1} g(s)\, ds \simeq J^{QE}(g) := \sum_{j=0}^{q-1} a_j g(\alpha_j),$$

où les $a_j \in \mathbf{R}$
et les $\alpha_j \in [-1, 1]$
caractérisent la méthode.

On dit que la méthode est d'ordre p
si (2)

est exacte pour les fonctions polynômiales de degrés inférieurs à p
et inexacte pour au moins un polynôme de degré $p+1$
.

La formule composée associée pour approcher $I(f)$
est donnée par la formule

$$J_x^{QC}(f) = \sum_{i=0}^{N-1} \frac{h_i}{2} J^{QE}(f \circ \varphi_i),$$

On utilisera des discrétisation régulières : $h_i = h = (b-a)/N$
et $x_i = a + ih$
pour $i = 0, \ldots, N$
.

haut 1. 2. 3. bas.

1. Méthodes d'ordres 1, 2, 3

Nous avons implémenté deux formules de quadratures élémentaires correspondant à deux méthodes vues au TP1 (rectangles à gauche et point milieu). Ci-dessous la fonction *Jmeth*
avec $meth \in \{rg, pm\}$
prend en donnée une fonction f
et deux réels c
et d
et renvoie le réel

$$\begin{cases} 2g(c) & \text{pour } meth = rg \\ 2g((c+d)/2) & \text{pour } meth = pm. \end{cases}$$

In [2]:

```
Jrg = lambda g, c, d: 2*g(c)
Jpm = lambda g, c, d: 2*g(.5*(c + d))
```

À ces deux méthodes de quadratures élémentaires, nous ajoutons une troisième de méthode, celle de Gauss à deux points:

$$J_{ga}^{QE}(g) := g\left(-\sqrt{3}/3\right) + g\left(\sqrt{3}/3\right).$$

In [3]:

```
def Jga(f, c, d):
    m, h, v = .5*(c+d), .5*(d - c), 1/np.sqrt(3)
    return f(m - h*v) + f(m + h*v)
```

Exercice 1. Programmer la fonction *quadrature*
qui prend en donnée une fonction f
, deux réls a
, b
, une méthode de quadrature *Jmeth*
(parmi *Jrg*
, *Jrd*
, *Jpm*
, *Jga*
) et un entier N
et qui renvoie le réel

$$J^{QC}(f) = \frac{h}{2} \sum_{i=0} J^{QE}(g_i),$$

où (avec les notations de (1)), $g_i(s) = f(c_i + sh/2)$

.

Tester cette fonction avec $f: [0,1] \to \mathbf{R}, x \mapsto x$
et $f: [0,\pi] \to \mathbf{R}, x \mapsto \sin x$
.

In [5]:

```
### SOLUTION
def quadrature(f, a, b, Jmeth, N):
    x,h = np.linspace(a, b, N + 1), (b-a)/N
    S = 0
    for i in range(N):
        S += Jmeth(f, x[i], x[i + 1])
    return S*h/2
```

In [6]:

```
### SOLUTION
# tests
# tests
N = 16
a, b, f = 0, 1, lambda x:x
Irg = quadrature(f, a, b, Jrg, N)
Ipm = quadrature(f, a, b, Jpm, N)
Iga = quadrature(f, a, b, Jga, N)
print(Irg, Ipm, Iga,"\n")

N = 16
a, b, f = 0, np.pi, lambda x: np.sin(x)
Irg = quadrature(f, a, b, Jrg, N)
Ipm = quadrature(f, a, b, Jpm, N)
Iga = quadrature(f, a, b, Jga, N)
print(Irg, Ipm, Iga)
```

```
0.46875 0.5 0.5

1.99357034377 2.00321637817 1.99999931103
```

Exercice 2. Modifier la fonction *calcul_erreurs*
du TP précédent pour étudier l'erreur des trois méthodes de quadrature précédente.

La fonction *calcul_erreurs*
prend en entrée une fonction f
, deux réels a
, b
, un réel I
, une méthode élémentaire *Jmeth*
un entier k
. Elle renvoie deux tableaux H
et E
de taille k
.

In [8]:

```
### SOLUTION
def calcul_erreurs(f, a, b, I, Jmeth, k):
    H, E, N, h = np.zeros(k), np.zeros(k), 1, b - a
    for j in range(k):
        N *= 2
        h /= 2
        E[j]= np.abs(quadrature(f, a, b, Jmeth, N) - I)
```

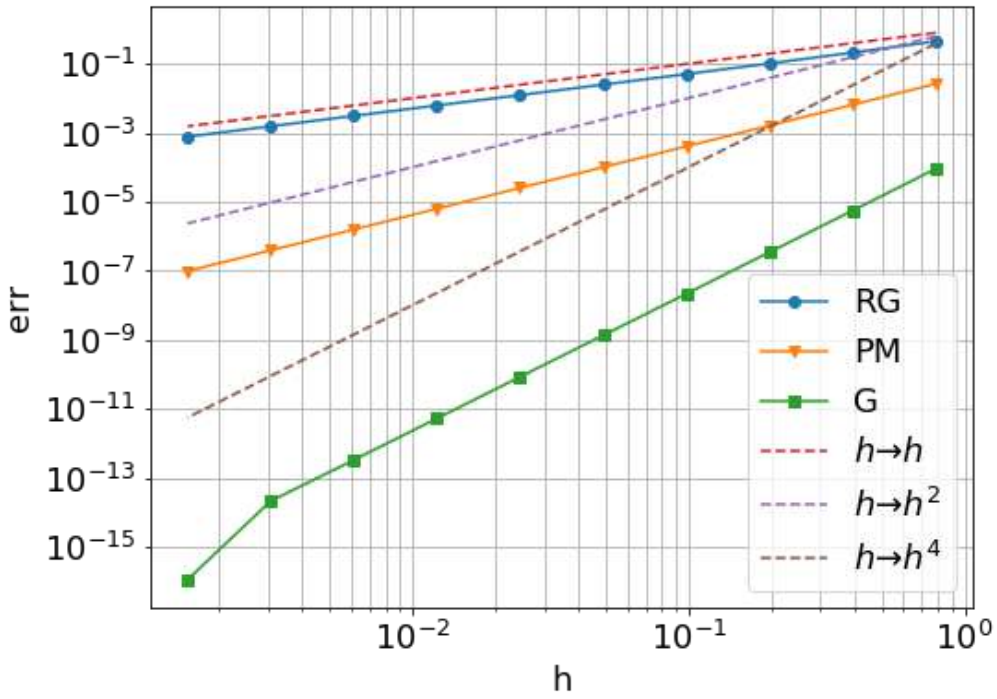
```
        H[j]= h
    return H, E
```

Exercice 3. Utiliser la fonction *calcul_erreurs*
pour observer l'ordre de convergence des méthodes ci-dessus. On testera la convergence en calculant des valeurs approchées de

$$I = \int_0^{\pi/2} \sin(x)\, dx = 1.$$

In [9]:

```
### SOLUTION
a, b, f, I = 0, np.pi/2, lambda x: np.sin(x), 1
k = 10
meths=[Jrg,Jpm,Jga]
name=['RG','PM','G']
marker=['o-','v-','s-']
j=0
for Jmeth in meths:
    H, E = calcul_erreurs(f, a, b, I, Jmeth, k)
    plt.loglog(H, E, marker[j], label=name[j])
    j+=1
plt.loglog(H,H,'--',label=r'$h\to h$')
plt.loglog(H,H**2,'--',label=r'$h\to h^2$')
plt.loglog(H,H**4,'--',label=r'$h\to h^4$')
plt.legend()
plt.xlabel("h")
plt.ylabel("err")
plt.grid(True, which="both", ls='-')
plt.show()
```



Solution : (suite) La méthode des rectangles à gauche est d'ordre 0 (pente 1), celle du point milieu d'ordre 1 (pente 2) et celle de Gauss à deux points d'ordre 3 (pente 4)._

haut 1. 2. 3. bas.

2. Une formule de quadrature d'ordre 5

Soit $\alpha \in]0,1[$
et $c = (c_0, c_1, c_2, c_3) \in \mathbf{R}^4$
. On considère la formule d'intégration numérique élémentaire :

$$J^{QE}_{c,\alpha}(f) = c_0 f(-1) + c_1 f(-\alpha) + c_2 f(\alpha) + c_3 f(1)$$

pour approcher $I(f) = \int_{-1}^{1} f(s)ds$

.

Exercice 4. Ecrire les conditions sur c_0

, c_1

, c_2

, c_3

pour que la formule $J^{a}_{c,\alpha}$

soit au moins d'ordre 3.

Montrer que dans ce cas, on a nécessairement $c_0 = c_3$

et $c_1 = c_2$

. En déduire les valeurs de c_0

, c_1

, c_2

, c_3

pour que $J^{QE}_{2,c,\alpha}$

soit au moins d'ordre 3.

On note J^{QE}_{α}

la formule de quadrature obtenue. Montrer qu'elle s'écrit :

$$J^{QE}_{\alpha}(f) = \frac{1-3\alpha^2}{3(1-\alpha^2)}(f(-1)+f(1)) + \frac{2}{3(1-\alpha^2)}(f(\alpha)+f(-\alpha)).$$

Solution : On écrit les conditions pour que la formule de quadrature soit exacte pour les fonctions polynômiales $x \mapsto x^d$

pour $d = 0, 1, 2, 3$

.

$$\begin{aligned} c_0 + c_1 + c_2 + c_3 &= 2 && (a)\\ -c_0 - c_1\alpha + c_2\alpha + c_3 &= 0 && (b)\\ c_0 + c_1\alpha^2 + c_2\alpha^2 + c_3 &= 2/3 && (c)\\ -c_0 - c_1\alpha^3 + c_2\alpha^3 + c_3 &= 0 && (d) \end{aligned}$$

(b)-(d) donne $c_1 = c_2$

et α^2

(b)-(d) donne $c_0 = c = 3$

. Ensuite (a) et (c) se récrivent

$$\begin{aligned} c_0 + c_1 &= 1 && (a')\\ c_0 + c_1\alpha^2 &= 1/3 && (c') \end{aligned}$$

(a')-(b') donne $c_1 = 2/(3(1-\alpha^2))$

et (c')-α^2

(a') donne $c_0 = (1-3\alpha^2)/(3(1-\alpha^2))$

. Au final

$$c_0 = c_3 = \frac{1-3\alpha^2}{3(1-\alpha^2)}, \qquad c_1 = c_2 = \frac{2}{3(1-\alpha^2)}.$$

On obtient bien la formule de quadrature élémentaire annoncée.

Exercice 5. **Montrer qu'il existe** $\alpha_0 \in]0,1[$
tel que la formule $J^{QE}_{\alpha_0}$
soit au moins d'ordre 5.

Calculer explicitement α_0
.

Solution : **La formule est au moins d'ordre 4 si et seulement si** $J^{QE}_{\alpha}(x \mapsto x^4) = 2/5$
. C'est équivalent à_

$$1 - 3\alpha^2 + 2\alpha^4 = (3/5)(1-\alpha^2) \;\Leftrightarrow\; 5\alpha^4 - 6\alpha^2 + 1 = 0$$

_Le polynôme $5X^2 - 6X + 1$
se factorise en $5(X-1)(X-1/5)$
. Sa seule racine dans l'intervalle $]0,1[$
est $X_0 = 1/5$
. On conclut que pour_

$$\alpha_0 = \sqrt{X_0} = \frac{\sqrt{5}}{5},$$

la formule est au moins d'ordre 4.

Finalement, par symétrie de la formule, on a toujours

$$J^{QE}_{\alpha}(x \mapsto f(-x)) = J^{QE}_{\alpha}(f).$$

On en déduit que pour
f ***impaire, on a***

$$J^{QE}_{\alpha}(f) = \int_{-1}^{1} f = 0.$$

_En particulier, la formule est aussi exacte pour $x \mapsto x^5$
. La formule est donc au moins d'ordre 5 pour le paramètre $\alpha = \alpha_0$
._

Exercice 6. **Déduire de** J^{QE}_{α}
une formule composée pour approcher $\int_a^b f(x)dx$
sur une subdivision régulière.

Que peut-on dire de la vitesse de convergence de la formule composée en fonction du pas de la subdivision h
?

Solution : **Dans le cas du calcul de l'intégrale d'une fonction régulière sur** $[a,b]$
, on s'attend à avoir une erreur proportionnelle à h^4
pour $\alpha \in]0,1[\setminus \{\alpha_0\}$
et proportionnelle à h^6
pour $\alpha = \alpha_0$
._

Exercice 7. **Reprendre les exercices 1, 2 et 3 avec cette nouvelle formule de quadrature. Vous devez implémenter de nouvelles fonctions** *quadrature2*
, *calcul_erreurs2*
qui dépendent du nouveau paramètre *alpha*
.

Retrouve-t-on numériquement les ordres de convergence attendus dans les cas $\alpha = \alpha_0$
et $\alpha = 1/2$
?

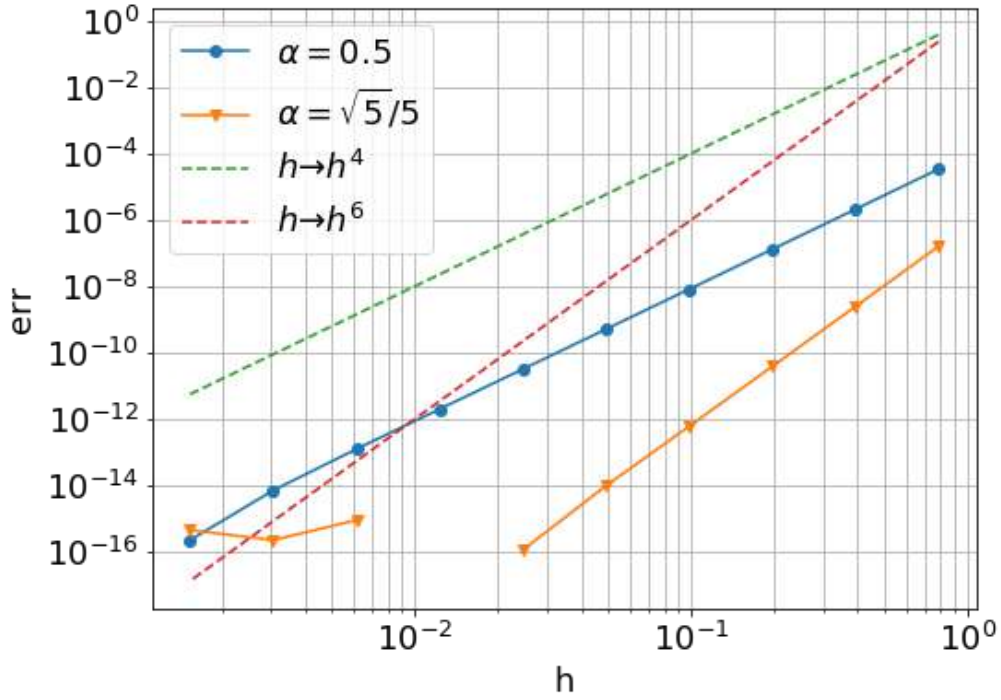
.

In [8]:

```
### SOLUTION (fonctions quadrature2 et calcul_erreurs2)
def Jformule(f, c, d,alpha):
    coeff1=(1-3*alpha**2)/3/(1-alpha**2)
    coeff2=2/3/(1-alpha**2)
    pointg=(c+d)/2 -alpha*(d-c)/2
    pointd=(c+d)/2 +alpha*(d-c)/2
    return coeff1*(f(c)+f(d))+coeff2*(f(pointg)+f(pointd))

def quadrature_2(f, a, b, alpha, N):
    x,h = np.linspace(a, b, N + 1), (b-a)/N
    S = 0
    for i in range(N):
        S += Jformule(f, x[i], x[i + 1],alpha)
    return S*h/2

def calcul_erreurs_2(f, a, b, I, alpha, k):
    H, E, N, h = np.zeros(k), np.zeros(k), 1, b - a
    for j in range(k):
        N *= 2
        h /= 2
        E[j]= np.abs(quadrature_2(f, a, b, alpha, N) - I)
        H[j]= h
    return H, E
```

In [9]:

```
### SOLUTION (calcul et affichage des courbes de convergence)
a, b, f, I = 0, np.pi/2, lambda x: np.sin(x), 1
k = 10
marker=['o-','v-']
name=[r'$\alpha=0.5$', r'$\alpha=\sqrt{5}/5$']
j=0
for alpha in [0.5, np.sqrt(5)/5]:
    H, E = calcul_erreurs_2(f, a, b, I, alpha, k)
    plt.loglog(H, E, marker[j], label=name[j])
    j+=1
plt.loglog(H,H**4,'--',label=r'$h\to h^4$')
plt.loglog(H,H**6,'--',label=r'$h\to h^6$')
plt.legend()
plt.xlabel("h")
plt.ylabel("err")
plt.grid(True, which="both", ls='-')
plt.show()
```



Solution : ...

haut 1. 2. 3. bas.

3. Intégrale de fonctions singulières en 0

On souhaite calculer de manière approchée des intégrales $I(f) = \int_0^b f(x)\,dx,$
pour $f:]0, b] \to \mathbf{R}$
de la forme

$$f(x) = \frac{g(x)}{\sqrt{x}}, \text{ avec } g \text{ de classe } C^\infty \text{ sur } [0, b] \text{ et } g(0) \neq 0.$$

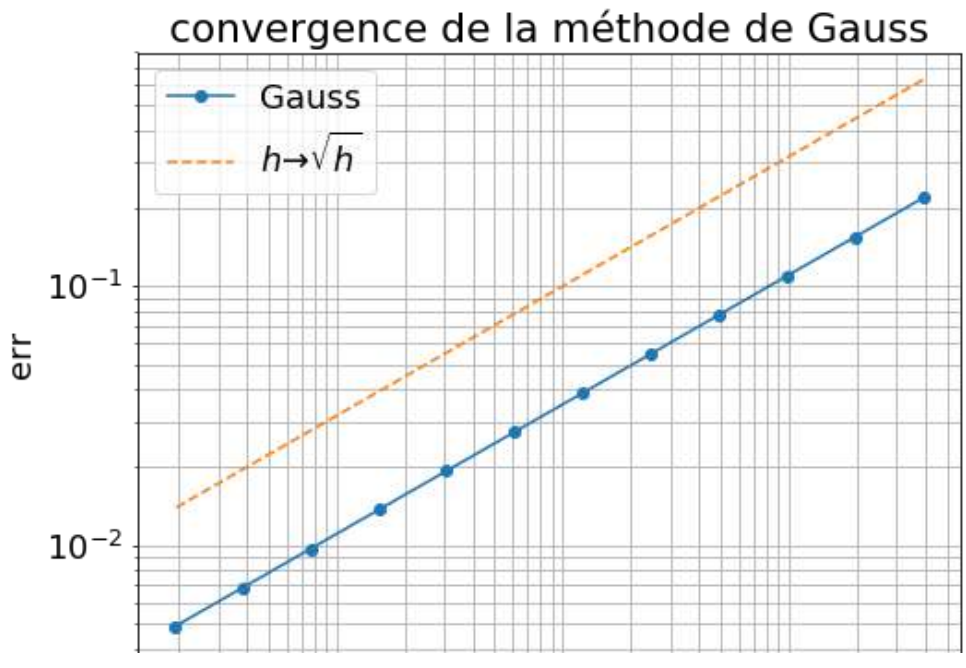
Exercice 8. Utilisez la méthode de Gauss de la partie 1 pour le calcul approché de

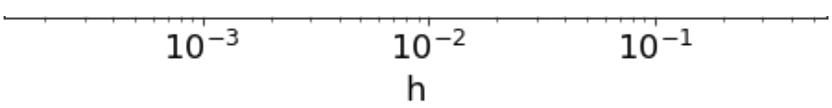
$$\int_0^{\pi/4} \frac{dt}{\sqrt{\tan t}} = \frac{\sqrt{2}}{4} \ln\left(3 + 2\sqrt{2}\right) + \frac{\sqrt{2}}{2} \left(\arctan(\sqrt{2} - 1) + \arctan(\sqrt{2} + 1)\right).$$

Quelle est la vitesse de convergence observée numériquement ?

In [15]:

```
b = np.pi/4
f = lambda t: 1/np.sqrt(np.tan(t))
s2 = np.sqrt(2)
I = s2/4*np.log(3 + 2*s2) + 1/s2*(np.arctan(s2 - 1) +np.arctan(s2 + 1))

### SOLUTION
k = 12
H, E = calcul_erreurs(f, a, b, I, Jga, k)
plt.loglog(H, E, 'o-',label='Gauss')
plt.title('convergence de la méthode de Gauss')
plt.xlabel("h")
plt.ylabel("err")
plt.grid(True, which="both", ls='-')

plt.loglog(H,np.sqrt(H),'--',label=r'$h\to \sqrt{h}$')
plt.legend()
plt.show()
```





Solution : (suite) On observe une convergence d'ordre $-1/2$ (pente $1/2$), ce qui est très lent.

Exercice 9. On suppose que la fonction f est du type (4). En faisant le changement de variable $x = t^2$, écrire $I(f) = \int_0^{\sqrt{b}} F_f(t)\,dt$ où la fonction F_f qui dépend de f est à déterminer. Quelle est la régularité de F_f ?

Exprimer $J_{ga}^{QC}(F_f)$ en fonction de f (voir formule (3)).

En déduire une méthode pour approcher $I(f)$ pour des fonctions f du type (4).

Solution :

$$\int_0^b f(x)\,dx = \int_0^b \frac{g(x)}{\sqrt{x}}\,dx \overset{x=t^2}{=} \int_0^{\sqrt{b}} 2g(t^2)\,dt = \int_0^{\sqrt{b}} 2tf(t^2)\,dt = \int_0^{\sqrt{b}} F_f(t)\,dt,$$

où on a posé $F_f(t) = 2tf(t^2) = 2g(t^2)$. La fonction F_f est de classe C^∞.

On pose $h = \sqrt{b}/N$, $y_i = (i+1/2)h$ et $r = \sqrt{3}/6$. On calcule,

$$\begin{aligned} J_{ga}^{QC}(F_f) &= \frac{h}{2}\sum_{i=0}^{N-1}\left(F_f(y_i - rh) + F_f(y_i + rh)\right) \\ &= h\sum_{i=0}^{N-1}\left((y_i - rh)f\left((y_i - rh)^2\right) + (y_i + rh)f\left((y_i + rh)^2\right)\right) \end{aligned}$$

Exercice 10. Mettre en oeuvre cette méthode avec la formule de quadrature de Gauss de la partie 1. Observez numériquement la vitesse de convergence avec le calcul de (5).

Comparer avec les résultats numérique de l'exercice 9.

In [11]:

```
### SOLUTION (fonction quadrature3)
def quadrature3(f,b,N):
    B=np.sqrt(b)
    F=lambda x:2*x*f(x**2)
    return quadrature(F,0,B,Jga,N)
```

In [12]:

```
### SOLUTION (tests de quadrature3)
Iapp = quadrature3(lambda x:1/np.sqrt(x), 1, 10)
print("test :", Iapp - 2)

Iapp = quadrature3(lambda t: 1/np.sqrt(np.tan(t)), np.pi/2, 1)
print("test :", Iapp - np.pi/np.sqrt(2))
```

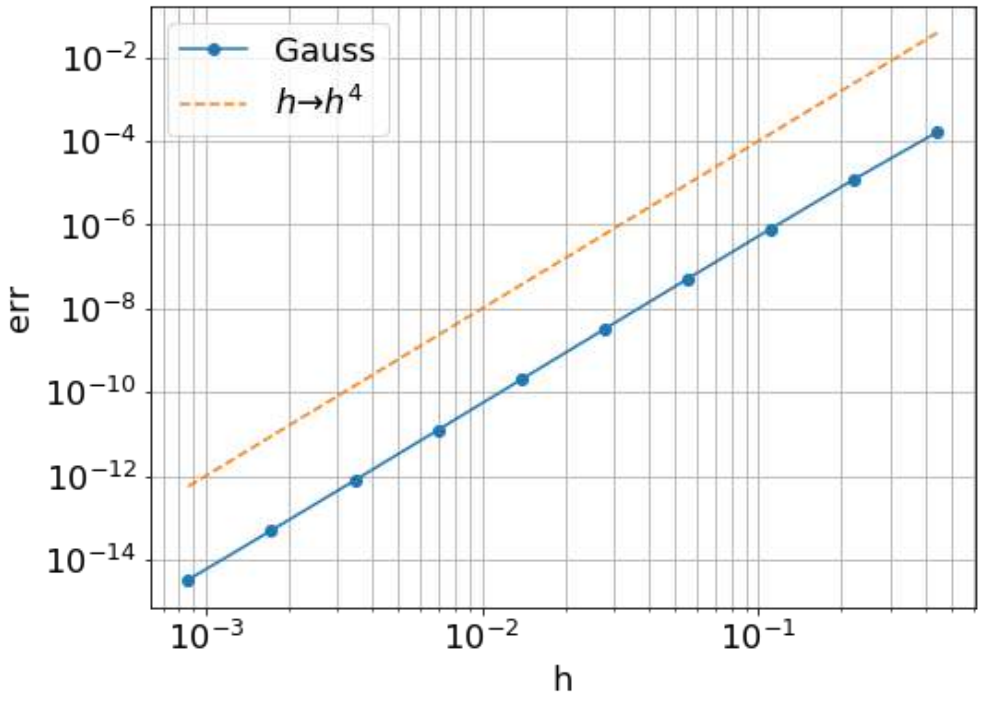
```
test : 0.0
test : 0.0486593575735
```

In [13]:

```
### SOLUTION (fonction calcul_erreurs3)
def calcul_erreurs3(f, b, I, k):
    H, E, N, h = np.zeros(k), np.zeros(k), 1, np.sqrt(b)
    for j in range(k):
        N *= 2
        h /= 2
        E[j]= np.abs(quadrature3(f, b, N) - I)
        H[j]= h
    return H, E
```

In [14]:

```
### SOLUTION (calcul et affichage de la courbe de convergence)
b = np.pi/4
f = lambda t: 1/np.sqrt(np.tan(t))
s2 = np.sqrt(2)
I = s2/4*np.log(3 + 2*s2) + 1/s2*(np.arctan(s2 - 1) +np.arctan(s2 + 1))
k = 10
H, E = calcul_erreurs3(f, b, I, k)
plt.loglog(H, E, 'o-',label="Gauss")
plt.loglog(H,H**4,'--',label=r'$h\to h^4$')
plt.legend()
plt.xlabel("h")
plt.ylabel("err")
plt.grid(True, which="both", ls='-')
plt.show()
```



Solution : (fin) On observe une convergence d'ordre 3
(pente 4
), ce qui est l'ordre de convergence attendu pour la méthode de Gauss à 2 points.

[haut](#) [1.](#) [2.](#) [3.](#) [bas.](#)

TP3

TP 3. Méthodes itératives

Sommaire

Nous chargons les deux bibliothèques, *numpy* et *matplotlib.pyplot*.

In [1]:

```
import numpy as np
import matplotlib.pyplot as plt
```

haut 1. 2. 3. bas.

1. Programmation d'une méthode itérative : principes fondamentaux

Les méthodes itératives dédiées au calcul approché d'un nombre $\alpha \in \mathbb{R}$ (qui pourra être le zéro d'une fonction f donnée) reposent sur la construction d'une suite $(u_n)_{n\in\mathbb{N}}$ qui vérifie

$$\lim_{n\to\infty} u_n = \alpha.$$

A l'aide d'une boucle for, on peut construire les éléments de la suite $(u_n)_{0\leq n\leq N}$ mais rien ne garantit que u_N, la dernière valeur calculée est "proche" de α.

Prenons l'exemple de la suite (S_n) introduite dans le cours :

$$S_n = \sum_{k=1}^{n} \frac{1}{k^2}$$

dont la limite est $\alpha = \frac{\pi^2}{6}$.

In [2]:

```
def suiteS(N):
    S = np.zeros(N)
    s = 1
    S[0]=1
    for k in range(1,N):
        s = s + 1/(k+1)**2
        S[k] = s
    return S
```

```
print(suiteS(10))
```

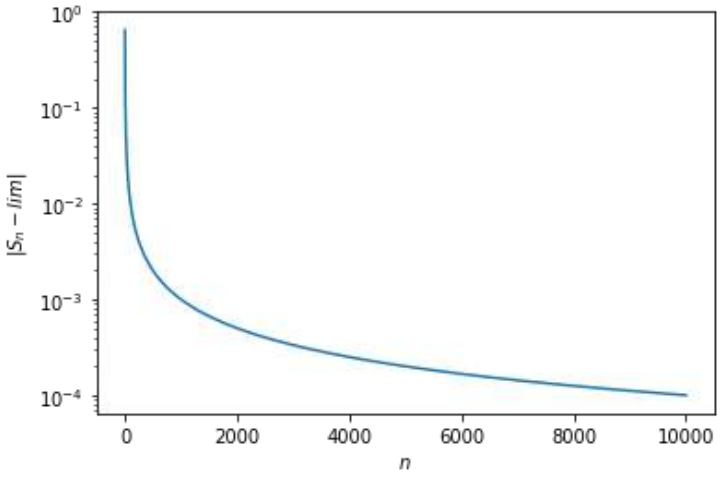
```
[ 1.          1.25        1.36111111  1.42361111  1.46361111  1.49138889
  1.51179705  1.52742205  1.53976773  1.54976773]
```

On a choisi ici de stocker tous les éléments de la suite. On verra par la suite qu'on ne garde en général que la dernière valeur.

On met en évidence avec le graphique suivant la lenteur de convergence de la suite $(S_n)_{n\geq 1}$.

In [3]:

```
N=10000
S = suiteS(N)
limite=np.pi**2/6
plt.plot(np.linspace(1,N,N), np.abs(S-limite))
plt.semilogy()
plt.xlabel('$n$')
plt.ylabel(r'$|S_n-lim|$')
plt.show()
```



Dans le but de calculer une "bonne" approximation de la limite α d'une suite $(u_n)_{n\in\mathbb{N}}$, on va calculer les éléments de la suite $(u_n)_{0\leq n\leq N}$, pour une valeur N déterminée par un citère d'arrêt. Ce critère doit garantir que u_N est proche de α.

On se donne pour cela une tolérance τ et on choisit un des critères d'arrêt suivants :

- $$|u_N - u_{N-1}| < \tau,$$
- $$\frac{|u_N - u_{N-1}|}{|u_N|} < \tau,$$
  (à privilégier si α est proche de 0)
- $$|f(u_N)| \leq \tau$$
  (si on cherche le zéro d'une fonction f).

On utilise alors une boucle while qui permet de calculer les itérés jusqu'à ce que le critère d'arrêt soit satisfait. Attention : dans une boucle while, on fixe toujours un nombre d'itérations maximal pour éviter les boucles "infinies".

Reprenons le calcul de la suite $(S_n)_{n\geq 1}$ avec la boucle while. On ne stocke plus que la dernière valeur calculée.

In [4]:

```
def suiteS_2(tau,nmax):
    S=1
    n=1
    testarret= True
    while testarret and n<nmax:
```

```
        Sprec=S
        n=n+1
        S+=1/(n**2)
        testarret= (np.abs(S-Sprec)>tau)
    return S,n

S,n=suiteS_2(1e-5,1e5)
print("S=",S)
print("n=",n)
print('|S-limite|=',np.abs(S-np.pi**2/6))
```

```
S= 1.6417844631526846
n= 317
|S-limite|= 0.00314960369554
```

Exercice 1. **Dans une fonction** $suiteHeron$**, programmer la suite**

$$\begin{cases} u_0 \in [0,1] \\ u_{n+1} = \frac{1}{2}\left(u_n + \frac{2}{u_n}\right) \end{cases}$$

La fonction $suiteHeron$ **prend en entrée la donnée initiale** u_0**, la tolérance** τ**, un nombre d'itérations maximal** $nmax$**. On pourra choisir le premier critère d'arrêt. En sortie, on donnera la dernière valeur de la suite calculée** u_N **et le nombre d'itérations effectué** N**.**

On choisit $u_0 = 1$ **et on fixe** $nmax = 20$**. Faire varier la tolérance (**$\tau = 10^{-k}$ **pour** $1 \leq k \leq 14$**). Représenter sur un premier graphique le nombre d'itérations effectué en fonction de** τ **et sur un second graphique l'erreur** $|u_N - \sqrt{2}|$ **en fonction de** τ**. Attention au choix des échelles pour vos graphiques.**

In [5]:

```
### SOLUTION
# definition de la fonction suiteHeron
def suiteHeron(u0,tau,nmax):
    u=u0
    testarret= True
    n=0
    while testarret and (n<nmax):
        uprec=u
        u=(u+2/u)/2
        testarret=(np.abs(u-uprec)>tau)
        n=n+1
    return u,n
```

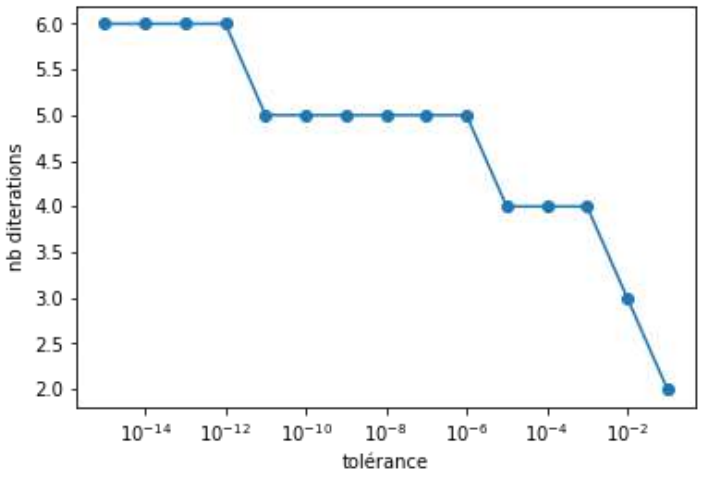
In [6]:

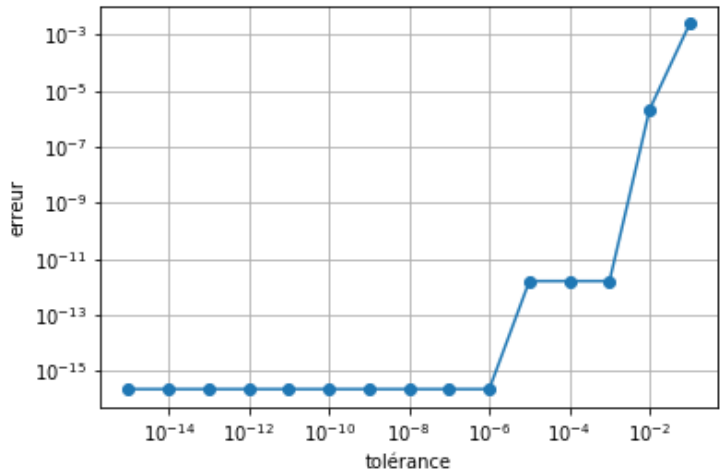
```
### SOLUTION (suite)
# Calcul du nombre d'itérations et de la dernière valeur
# de la suite pour différentes valeurs de tolérance

u0=1
nmax=100
Ntests=15
val_tau=10**(-np.linspace(1,Ntests,Ntests))
val_u= np.zeros(Ntests)
val_n= np.zeros(Ntests)

for j in range(Ntests):
    val_u[j], val_n[j] = suiteHeron(u0,val_tau[j],nmax)
```

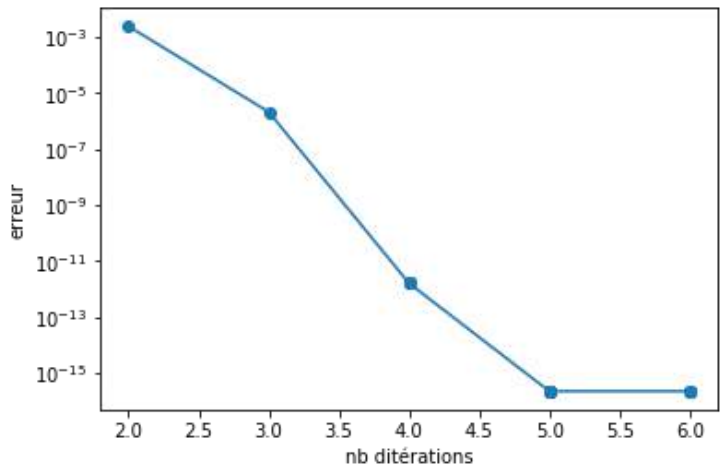
In [7]:

```
### SOLUTION (fin) Représentions graphiques
```

```
plt.figure(1)
plt.plot(val_tau,val_n,'o-')
plt.semilogx()
plt.xlabel('tolérance')
plt.ylabel('nb d''iterations')
plt.show()

plt.figure(2)
plt.plot(val_tau,np.abs(val_u-np.sqrt(2)),'o-')
plt.loglog()
plt.xlabel('tolérance')
plt.ylabel('erreur')
plt.grid()
plt.show()

plt.figure(3)
plt.plot(val_n,np.abs(val_u-np.sqrt(2)),'o-')
plt.semilogy()
plt.xlabel('nb d''itérations')
plt.ylabel('erreur')
plt.show()
```







Exercice 2. Soit $(u_n)_{n\geq 1}$ la suite telle que u_n est égal au demi-périmètre du polygone régulier à 2^n côtés inscrit dans le cercle unité.

1. Montrer que $u_n = 2^n \sin \frac{\pi}{2^n}$ pour tout $n \geq 1$.
2. Calculer $\lim_{n\to\infty} u_n$.
3. Pour $n \geq 1$, on pose $c_{n+1} = \frac{u_n}{u_{n+1}}$. Exprimez c_{n+1} en fonction de $\theta_{n+1} := \pi/2^{n+1}$. Que vaut c_1 défini par cette formule ? Donnez une expression de c_{n+1} en fonction de c_n.
4. Comment peut-on calculer les valeurs $(u_n)_{1\leq n\leq N}$ sans utiliser le nombre π (mais en utilisant néanmoins la fonction racine carrée) ?

Solution : _**1.** Posons $N = 2^n$. En utilisant le langage des nombres complexes, les sommets du polygones sont les points d'affixes $e^{i\alpha}e^{2ik\pi/N}$ pour $k = 0, \dots, N-1$ et un $\alpha \in \mathbb{R}$. La distance entre deux points successifs est $d_N = |e^{2i\pi/N} - 1|$. On calcule

$$\begin{aligned} d_N^2 &= (e^{2i\pi/N} - 1)(e^{-2i\pi/N} - 1) \\ &= 2(1 - \cos(2\pi/N)) = 4\sin^2(\pi/N). \end{aligned}$$

On en déduit $d_N = 2\sin(\pi/N)$. Le demi périmètre du polygone est donc $N\sin(\pi/N)$ et en rappelant $N = 2^n$, on a bien le résultat de l'énoncé._

(remarque : en faisant une figure on peut voir directement et sans calcul qu'une demi arête du polygone forme le côté opposé d'un triangle rectangle d'hypothenus 1 et d'angle adjacent π/N et a donc pour longueur $\sin(\pi/N)$)

_**2.** On a en posant $\varepsilon_n := \pi/2^n$,

$$u_n = \pi \frac{\sin \varepsilon_n}{\varepsilon_n}.$$

Comme $\varepsilon_n \downarrow 0$ quand $n \uparrow \infty$ et $\sin x \sim x$ au voisinage de $x = 0$, on en déduit

$$\lim_{n\uparrow\infty} u_n = \pi \lim_{x\downarrow 0} \frac{\sin x}{x} = \pi.$$

_

3. On définit pour $n \geq 1$***, $$ c*{n+1}:= \dfrac{u*n}{u*{n+1}}=\dfrac{\sin(\pi/2^n)}{2\sin(\pi/2^{n+1})}=\dfrac{\sin(\theta_n)}{2\sin(\theta_n/2)}, $$ où on a posé $\theta*n:=\pi/2^n$.***

_On rappelle les formules de trigonométrie

$$\begin{aligned} \sin\theta &= 2\sin(\theta/2)\cos(\theta/2), \\ \cos(\theta) &= 2\cos^2(\theta/2) - 1. \end{aligned}$$

En l'appliquant la première formule à θ_n et en simplifiant, on trouve

$$c_{n+1} = \cos(\theta_n/2) = \cos\theta_{n+1} = \cos(\pi/2^{n+1}).$$

En particulier,

$$c_1 = \cos(\pi/4) = 0.$$

En appliquant la seconde formule de trigonométrie ci-dessus (et en tenant compte du fait que $\theta_{n+1} \in [0, \pi/2] \;\Rightarrow\; \cos\theta_{n+1} \geq 0$) on obtient pour $n \geq 1$,_

$$c_{n+1} = \sqrt{\frac{1 + \cos(\theta_n)}{2}} = \sqrt{\frac{1 + c_n}{2}}.$$

_**4.** D'après la question précédente, on peut calculer la suite (u_n) en utilisant la relation de récurrence

$$\begin{cases} u_1 = 2, \quad c_1 = 0 \end{cases}$$

$$\begin{cases} c_{n+1} = \sqrt{\dfrac{1+c_n}{2}}, & u_{n+1} = \dfrac{u_n}{c_{n+1}} \end{cases} \quad \text{pour } n \geq 1.$$

_

Exercice 3. Écrire une fonction $suitecalculpi$ qui prend en entrée la tolérance τ, un nombre d'itérations maximal $nmax$. On pourra choisir le premier critère d'arrêt. En sortie, on donnera la dernière valeur de la suite calculée u_N et le nombre d'itérations effectué N.

Refaire les mêmes graphiques que dans l'exercice précédent. Qu'observe-t-on ? Peut-on démontrer le résultat observé ?

In [8]:

```
### SOLUTION
# definition de la fonction suitecalculpi

def suitecalculpi(tau,nmax):
    u=2
    c=0
    testarret=True
    n=0
    while testarret and (n<nmax):
        uprec=u
        c=np.sqrt((c + 1)/2)
        u=u/c
        testarret=(np.abs(u-uprec)>tau)
        n=n+1
    return u,n
```

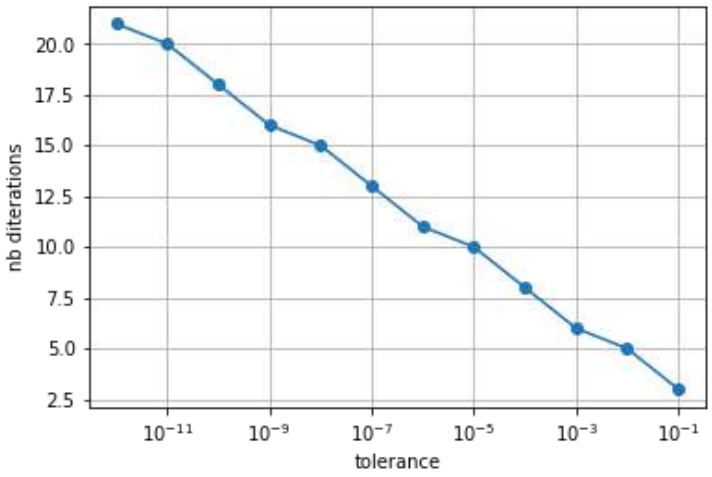
In [9]:

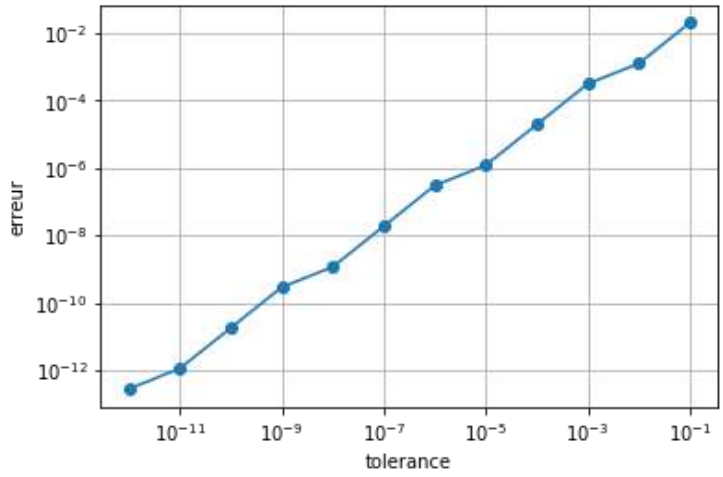
```
### SOLUTION (suite)
# Calcul du nombre d'itérations et de la dernière valeur
# de la suite pour différentes valeurs de tolérance

nmax=1e6
Ntests=12
val_tau=10**(-np.linspace(1,Ntests,Ntests))
val_u= np.zeros(Ntests)
val_n= np.zeros(Ntests)

for j in range(Ntests):
    val_u[j], val_n[j] = suitecalculpi(val_tau[j],nmax)
```

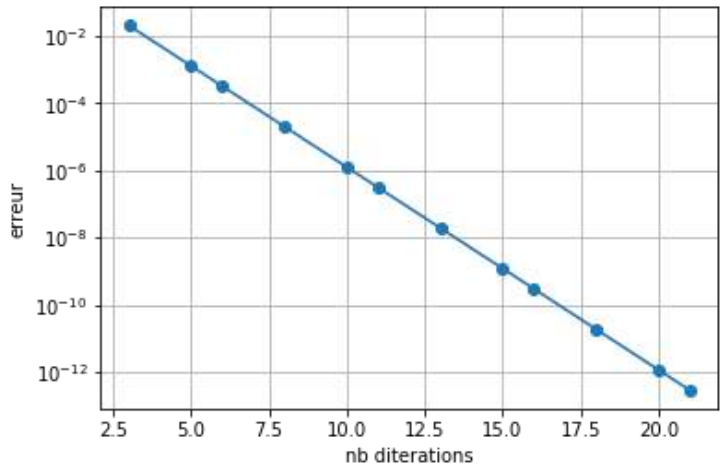
In [10]:

```
### SOLUTION (fin) Représentions graphiques

plt.figure(1)
plt.plot(val_tau,val_n,'o-')
plt.semilogx()
plt.xlabel('tolerance')
plt.ylabel('nb d''iterations')
plt.grid(True, which="both", ls='-')
plt.show()

plt.figure(2)
plt.plot(val_tau,np.abs(val_u-np.pi),'o-')
plt.loglog()
plt.xlabel('tolerance')
plt.ylabel('erreur')
plt.grid(True, which="both", ls='-')
plt.show()

plt.figure(3)
plt.plot(val_n,np.abs(val_u-np.pi),'o-')
plt.semilogy()
plt.xlabel('nb d''iterations')
plt.ylabel('erreur')
```

```
plt.grid(True, which="both", ls='-')
plt.show()
```







Solution : (suite) Estimation de l'erreur

_Utilisons l'inégalité classique

$$\begin{aligned} -\frac{x^3}{6} \\ \leq \sin x \\ -x \leq 0 \end{aligned}$$

et appliquons la à $x = \frac{\pi}{2^n}$. On obtient :

$$\begin{aligned} -2^n \\ (\frac{\pi}{2^n})^3 \\ \leq u_n \\ -\pi \leq 0, \end{aligned}$$

d'où

$$|u_n - \pi|$$

$$|u_n - \pi| \leq \frac{\pi^3}{4^n}.$$

Cela implique que

$$\log|u_n - \pi| \leq 3\log\pi - n\log 4.$$

–

haut 1. 2. 3. bas.

2. Méthode de dichotomie

On s'intéresse maintenant à des méthodes itératives qui permettent de calculer α, un zéro d'une fonction f donnée. Ces méthodes sont présentées dans le cours.

Exercice 4. La méthode de dichotomie permet d'approcher un zéro $\alpha \in [a,b]$ d'une fonction f continue sur $[a,b]$. L'existence d'un zéro est assurée dès que $f(a)f(b) < 0$. La suite $(x_n)_{n\geq 0}$ vérifie alors

$$|x_n - \alpha| \leq \frac{1}{2^n}|b-a|. \quad (*)$$

1. **Écrire une fonction $dichotomie$ qui a pour paramètres d'entrée f, a et b, la tolérance τ et un nombre d'itérations $nmax$. On choisira comme critère d'arrêt le troisième critère. La fonction renvoie l'approximation calculée ainsi que le nombre d'itérations.**
2. **Soit $f(x) = 4\sin(x) - x + 1$. Représenter graphiquement la fonction sur l'intervalle $[-5,5]$ puis utiliser la fonction $dichotomie$ pour déterminer une valeur approchée des différents zéros de la fonction f.**
3. **Soit $f(x) = x^2 - 2$, $a = 0$, $b = 2$. Utiliser la fonction programmée pour mettre en évidence le résultat $(*)$.**

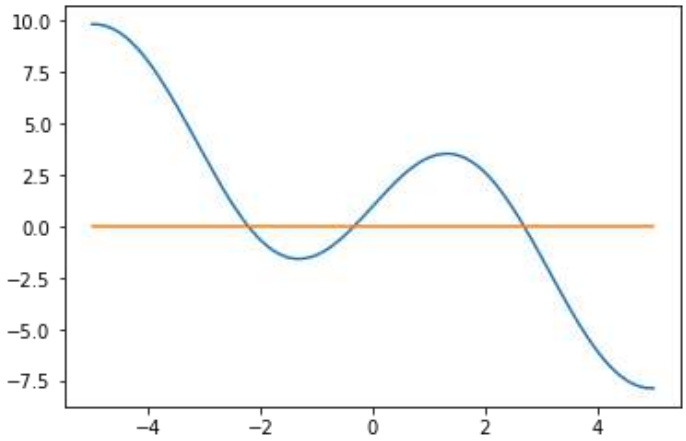
In [11]:

```
### SOLUTION 1.
# définition de la fonction dichotomie
def dichotomie(f,a,b,tau,nmax):
    if f(a)*f(b)>=0:
        return 'l''existence du zéro n''est pas garantie sur l''intervalle', []
    else:
        x=(a+b)/2
        testarret=(abs(f(x))>tau)
        n=0
        while testarret and n<nmax:
            if f(a)*f(x)<0:
                b=x
            else:
                a=x
            x=(a+b)/2
            n=n+1
            testarret=(abs(f(x))>tau)
    return x, n
```

In [12]:

```
### SOLUTION 2.
f = lambda x:4*np.sin(x)-x+1

plt.figure(1)
x=np.linspace(-5,5,500)
```

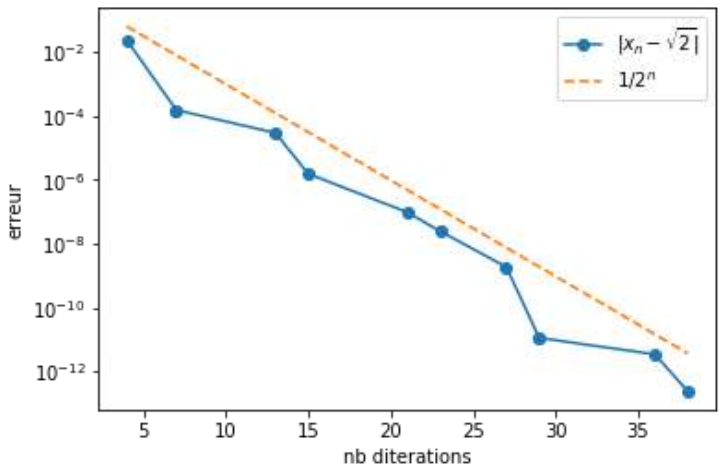
```
plt.plot(x,f(x),x,0*x)
plt.show()

x1,n1=dichotomie(f,-3,-1,1e-12,100)
x2,n2=dichotomie(f,-1,1,1e-12,100)
x3,n3=dichotomie(f,2,4,1e-12,100)

print("n1=",n1, ", x1=",x1,", f(w1)=", f(x1))
print("n2=",n2, ", x2=",x2,", f(w2)=", f(x2))
print("n3=",n3,", x3=",x3,", f(w3)=", f(x3))
```




```
n1= 40 , x1= -2.2100839440927302 , f(w1)= 2.34479102801e-13
n2= 41 , x2= -0.3421850529243784 , f(w2)= 2.209343819e-13
n3= 41 , x3= 2.70206137332616 , f(w3)= -5.52446977053e-13
```

In [13]:

```
### SOLUTION 3.
f= lambda x:x**2-2

nmax=100
Ntests=12
val_tau=10**-(np.linspace(1,Ntests,Ntests))
val_x=np.zeros(Ntests)
val_n=np.zeros(Ntests)

for j in range(Ntests):
    val_x[j], val_n[j]=dichotomie(f,0,2,val_tau[j],nmax)

plt.plot(val_n,np.abs(val_x-np.sqrt(2)),
         'o-',label=r'$|x_n-\sqrt{2}|$')
plt.plot(val_n, 1/(2**val_n),'--', label=r'$1/2^n$')
plt.semilogy()
plt.xlabel('nb d''iterations')
plt.ylabel('erreur')
plt.legend()
plt.show()
```



haut 1. 2. 3. bas.

3. Méthode de fausse position

Exercice 5. **La méthode de la fausse position permet d'approcher un zéro $\alpha \in [a,b]$ d'une fonction f continue sur $[a,b]$. L'existence d'un zéro est assurée dès que $f(a)f(b) < 0$ ET que la fonction est convexe ou concave sur l'intervalle $[a,b]$.**

1. **Ecrire une fonction $fausseposition$ qui a pour paramètres d'entrée f, a et b, la tolérance τ et un nombre d'itérations $nmax$. On choisira comme critère d'arrêt le troisième critère. La fonction renvoie l'approximation calculée ainsi que le nombre d'itérations.**
2. **Soit $f(x) = 4\sin$. Utiliser la fonction $fausseposition$ pour déterminer une valeur approchée des différents**
   $(x) - x + 1$
   zéros de la fonction f.
3. **Soit $f(x) = x^2$, $a = 0$, $b = 2$. Comparer l'efficacité de la méthode de la fausse position à celle de la**
   $-\,2$
   méthode de dichotomie.

In [14]:

```
### SOLUTION 1.
# définition de la fonction fausseposition
def fausseposition(f,a,b,tau,nmax):
    if f(a)*f(b)>=0:
        print("l'existence du zéro n''est pas garantie"
              + "sur l'intervalle")
        return a,-1
    else:
        x=a-f(a)*(b-a)/(f(b)-f(a))
        testarret=(abs(f(x))>tau)
        n=0
        while testarret==1 and n<nmax:
            if f(a)*f(x)<0:
                b=x
            else:
                a=x
            x=a-f(a)*(b-a)/(f(b)-f(a))
            n=n+1
            testarret=(abs(f(x))>tau)
    return x, n
```

In [15]:

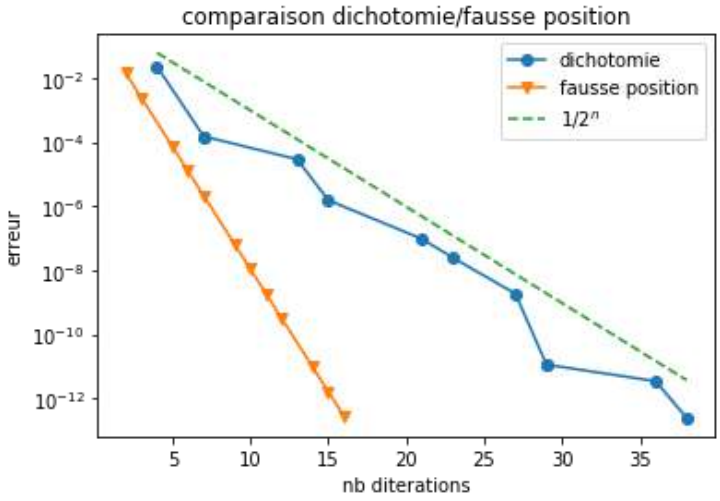
```
### SOLUTION 2.
f = lambda x:4*np.sin(x)-x+1

w1,n1=fausseposition(f,-3,-1,1e-12,100)
w2,n2=fausseposition(f,-1,1,1e-12,100)
w3,n3=fausseposition(f,2,4,1e-12,100)

print("n1=",n1, ", w1=",w1,", f(w1)=", f(x1))
print("n2=",n2, ", w2=",w2,", f(w2)=", f(x2))
print("n3=",n3,", w3=",w3,", f(w3)=", f(x3))
```

```
n1= 20 , w1= -2.21008394409 , f(w1)= 2.34479102801e-13
n2= 6 , w2= -0.342185052924 , f(w2)= 2.209343819e-13
n3= 6 , w3= 2.70206137333 , f(w3)= -5.52446977053e-13
```

In [16]:

```
### SOLUTION 3.
f= lambda x:x**2-2

nmax=100
Ntests=12
val_tau=10**-(np.linspace(1,Ntests,Ntests))
val_x=np.zeros(Ntests)
```

```
val_n=np.zeros(Ntests)
val_w=np.zeros(Ntests)
val_p=np.zeros(Ntests)

for j in range(Ntests):
    val_x[j], val_n[j]=dichotomie(f,0,2,val_tau[j],nmax)
    val_w[j], val_p[j]=fausseposition(f,0,2,val_tau[j],nmax)

plt.plot(val_n,np.abs(val_x-np.sqrt(2)),'o-',label='dichotomie')
plt.plot(val_p,np.abs(val_w-np.sqrt(2)),'v-',label='fausse position')
plt.plot(val_n, 1/(2**val_n),'--', label=r'$1/2^n$')
plt.semilogy()
plt.xlabel('nb d''iterations')
plt.ylabel('erreur')
plt.title('comparaison dichotomie/fausse position')
plt.legend()
plt.show()
```



haut 1. 2. 3. bas.

TP4

TP 4. Méthodes de point fixe et de Newton

Sommaire

Nous chargons les deux bibliothèques, *numpy* et *matplotlib.pyplot*.

In [17]:

```
import numpy as np
import matplotlib.pyplot as plt
import math
```

haut 1. 2. 3. bas.

1. Les méthodes de point fixe

Rappellons que pour la résolution d'une équation $f(x)=0$, la méthode de point fixe consiste en la construction d'une suite itérative $(u_n)_{n\in\mathbb{N}}$ définie par

$$\begin{cases} u_0 \text{ donné} \\ u_{n+1} = h(u_n), \\ \forall n \in \mathbb{N} \end{cases} \quad (1)$$

où $h(x)=x-\lambda f(x)$ avec $\lambda\in\mathbb{R}^*$. Si h est continue et si $\lim u_n=\alpha$, alors α est un point fixe de h (c-à.d. $h(\alpha)=\alpha$), donc un zéro de f (c-à.d. $f(\alpha)=0$).

Théorème 1. Soit $h : I\subset\mathbb{R} \to \mathbb{R}$ où I est un intervalle ouvert non vide. On suppose :

- $h\in C^1(I)$,
- h possède un point fixe $\alpha\in I$,
- $|h'(\alpha)|<1$.

Alors, il existe $\rho>0$ tel que $I_\rho := [\alpha-\rho,\alpha+\rho]\subset I$ tel que pour tout $u_0\in I_\rho$, la suite $(u_n)_{n\in\mathbb{N}}$ définie par (1) est convergente, de limite α et avec une vitesse de convergence au moins linéaire.

Dans le cas particulier où $h\in C^2(I)$ et $h'(\alpha)=0$, on a une convergence au moins quadratique. (

Dans le but de calculer une "bonne" approximation de la limite α, on va calculer les éléments de la suite $(u_n)_{0\leq n\leq N}$, pour une valeur N déterminée par un citère d'arrêt. Ce critère doit garantir que u_N est proche de α.

On se donne pour cela une tolérance τ et on choisit le critère d'arrêt suivant :

$$|f(u_N)| \leq \tau \tag{2}$$

On utilise alors une boucle while qui permet de calculer les itérés jusqu'à ce que le critère d'arrêt soit satisfait. Attention : dans une boucle while, on fixe toujours un nombre d'itérations maximal pour éviter les boucles "infinies".

Exercice 1. Écrire une fonction $pointfixe$ qui prend en entrée : une fonction f, un réel lam, un réel $u0$, un réel tau et un entier $nmax$ et qui renvoie le réel u_N et l'entier N calculés en appliquant la méthode du point fixe à la fonction $h(x) = x$ avec le critère d'arrêt (2) (et un nombre maximal d'itérations $nmax$).

$$-lamf$$
$$(x)$$

In [21]:

```
def pointfixe(f,lam,u0,tau,nmax):
    u=u0
    testarret= True
    n=0
    while testarret and (n<nmax):
        uprec=u
        u=u - lam*f(u)
        testarret=(np.abs(f(uprec))>tau)
        n=n+1
    return u,n
```

Exercice 2. On fixe $nmax = 40$.

1. Soit $f(x) = 4\sin$. Représenter graphiquement la fonction sur l'intervalle $[-5,5]$ puis utiliser la fonction

$$(x) - x + 1$$

$pointfixe$ pour déterminer une valeur approchée des différents zéros de la fonction f à une tolerance $\tau = 10^{-12}$. Soit α un zéro de f. Alors on prend $u_0 = \lfloor \alpha \rfloor$ (la partie entière de α) et $\lambda = signe.$

$$(\cos\alpha)/5$$

2. Soit $f(x) = x^2$. Utiliser la fonction $pointfixe$ avec $u_0 = 1$ et $\lambda = 1/2$ pour déterminer une valeur approchée

$$-2$$

du zéro $\sqrt{2}$. Faire varier la tolérance ($\tau = 10^{-k}$ pour $1 \leq k \leq 14$). Représenter sur un premier graphique le nombre d'itérations effectuées en fonction de τ et sur un second graphique l'erreur $|u_N - \sqrt{2}|$ en fonction de τ. Attention au choix des échelles pour vos graphiques.

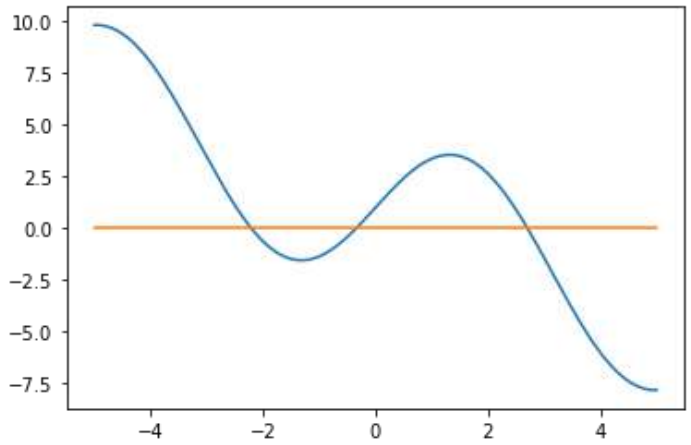
3. Pour la fonction de la question précédente, déterminer la valeur λ^* de λ pour laquelle la convergence est quadratique (voir (*) plus haut).

4. Comparer l'efficacité de la méthode du point fixe avec $\lambda = \lambda^*$ à celle de la méthode de la fausse position du TP3.

In [23]:

```
f = lambda x:(cos(alpha))/5

plt.figure(1)
x=np.linspace(-5,5,500)
plt.plot(x,f(x),x,0*x)
plt.show()

x1,n1=pointfixe(f,-1/5,-3,1e-12,40)
x2,n2=pointfixe(f,1/5,-1,1e-12,40)
x3,n3=pointfixe(f,-1/5,2,1e-12,40)

print("n1=",n1, ", x1=",x1,", f(x1)=", f(x1))
```

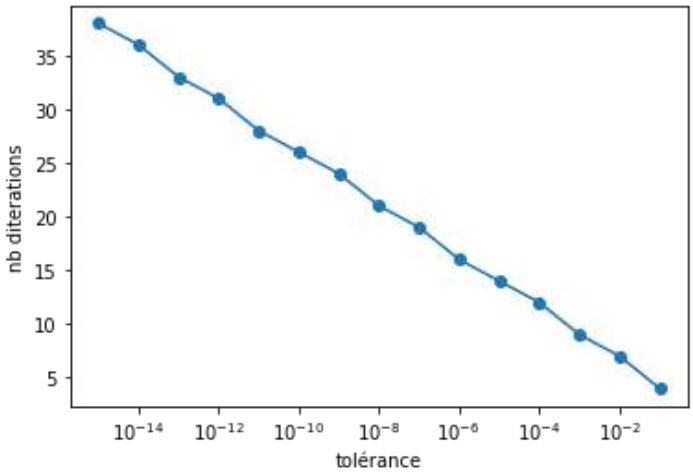
```
print("n2=",n2, ", x2=",x2,", f(x2)=", f(x2))
print("n3=",n3, ", x3=",x3,", f(x3)=", f(x3))
```

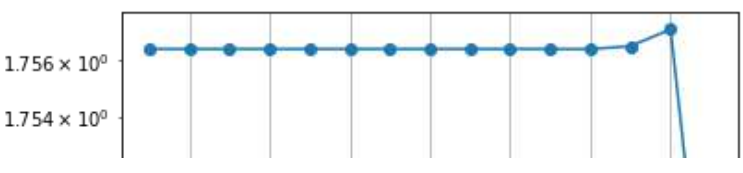



```
n1= 26 , x1= -2.210083944092707 , f(x1)= 1.567634910770721e-13
n2= 37 , x2= -0.34218505292458146 , f(x2)= -3.410605131648481e-13
n3= 13 , x3= 2.7020613733260306 , f(x3)= 4.463096558993129e-14
```

In [26]:

```
u0=1
nmax=100
Ntests=15
val_tau=10**(-np.linspace(1,Ntests,Ntests))
val_u= np.zeros(Ntests)
val_n= np.zeros(Ntests)

for j in range(Ntests):
    val_u[j], val_n[j] = pointfixe(f,1/2,1,val_tau[j],nmax)

### SOLUTION (fin) Représentions graphiques
plt.figure(1)
plt.plot(val_tau,val_n,'o-')
plt.semilogx()
plt.xlabel('tolérance')
plt.ylabel('nb d''iterations')
plt.show()

plt.figure(2)
plt.plot(val_tau,np.abs(val_u-np.sqrt(2)),'o-')
plt.loglog()
plt.xlabel('tolérance')
plt.ylabel('erreur')
plt.grid()
plt.show()
```





Solution 2.3 :

$h(x) = x - f(x)$, on a $f(x) = x^2 - 2$, d'où $f'(x) = 2x$

Ainsi $h'(x) = 1 - \lambda f'(x)$ $\Leftrightarrow h'(x) = 1 - \lambda 2x$

$$\lambda = \frac{1 - h'(x)}{2x}$$

Puis $h'(\alpha) = 0 \Leftrightarrow 1 - \lambda 2\alpha$ $\Leftrightarrow \lambda = \frac{1}{2}\alpha$

In [30]:

```
### SOLUTION 2.4, (début) fonction fausse position
f = lambda x:x**2-2
a=-5
b=5
tau=1e-10
nmax = 100
def fausseposition(f,a,b,tau,nmax):
    if f(a)*f(b)>=0:
        print("l'existence du zéro n''est pas garantie"
              + "sur l'intervalle")
        return a,-1
    else:
        x=a-f(a)*(b-a)/(f(b)-f(a))
        testarret=(abs(f(x))>tau)
        n=0
        while testarret==1 and n<nmax:
            if f(a)*f(x)<0:
                b=x
            else:
                a=x
            x=a-f(a)*(b-a)/(f(b)-f(a))
            n=n+1
            testarret=(abs(f(x))>tau)
    return x, n
fausseposition(f,a,b,tau,nmax)
```

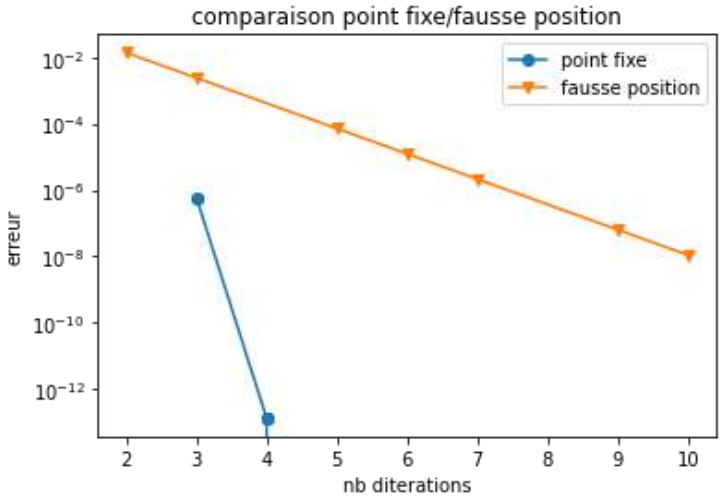
```
l'existence du zéro n''est pas garantiesur l'intervalle
```

Out[30]:

```
(-5, -1)
```

In [40]:

```
### SOLUTION 2.4 (suite) graphiques pour comparaison des méthodes
lam = 1/(2*np.sqrt(2))

f= lambda x:x**2-2
K=7
nmax=40
val_tau=np.asarray([10**(-k-1)for k in range(K)])
val_u_fp, val_n_fp = np.zeros(K),np.zeros(K)
val_u_pf, val_n_pf = np.zeros(K),np.zeros(K)

for k in range(K):
    tau = val_tau[k]
    val_u_fp[k],val_n_fp[k]=fausseposition(f,0,2,tau,nmax)
    val_u_pf[k],val_n_pf[k] = pointfixe(f,lam,1,tau,nmax)

plt.plot(val_n_pf, np.abs(val_u_pf - np.sqrt(2)),'o-',label='point fixe')
plt.semilogy(val_n_fp, np.abs(val_u_fp - np.sqrt(2)),'v-',label='fausse position')
plt.xlabel('nb d''iterations')
plt.ylabel('erreur')
plt.title('comparaison point fixe/fausse position')
plt.legend()
plt.show()
```



haut 1. 2. 3. bas.

2. Méthode de Newton

Exercice 3. La méthode de Newton consiste en la construction de la suite itérative $(u_n)_{n\in\mathbb{N}}$ définie par (1) avec

$$h(x) = x - \frac{f(x)}{f'(x)}.$$

Si $f'(\alpha) \neq 0$, on a bien $f(\alpha) = 0$ si et seulement si $h(\alpha) = \alpha$. Si de plus f est de classe C^2 au voisinage de α, alors $h'(\alpha) = 0$ et on aura une convergence au moins quadratique.

1. Ecrire une fonction $newton$ qui a pour paramètres d'entrée f, f' et u_0, la tolérance τ et un nombre d'itérations $nmax$ (on choisira le même critère d'arrêt que précedemment). La fonction renvoie l'approximation calculée ainsi que le nombre d'itérations. **Attention** : dans le cas où on aurait une division par 0 (c-à.d. s'il existe $k < nmax$ tel que $f'(u_k) = 0$), on arrête la boucle while en renvoyant le message <<Erreur : Division par 0>>.

2. Soit $f(x) = \sin(x)$. Utiliser la fonction $newton$ avec $u_0 = 3$ pour déterminer une valeur approchée du zéro π. Faire varier la tolérance ($\tau = 10^{-k}$ pour $1 \leq k \leq 14$). Représenter sur un premier graphique le nombre d'itérations effectuées en fonction de τ et sur un second graphique l'erreur $|u_N - \pi|$ en fonction de τ.

In [32]:

```
### SOLUTION 3.1 : fonction newton
```

In [33]:

```
### SOLUTION 3.2 : étude graphique de convergence
f= lambda x:np.sin(x)
df = lambda x:np.cos(x)
```

Exercice 4. On fixe desormais $nmax = 10$ et $\tau = 10^{-12}$. Si le point de départ u_0 est bien choisi, et si la convergence de la méthode de Newton est au moins quadratique, on s'attend à ce qu'il existe $n < nmax$ tel que u_n satisfait le critère d'arrêt $|f(u_n)| \leq \tau$. Pourquoi ce n'est pas le cas dans les exemples suivants ? Justifiez vos réponses et proposez des solutions pour améliorer la vitesse de convergence lorsque cela est possible.

$$f_1(x) = 1 - x^2, \quad \text{avec } u_0 = 0$$

$$f_2(x) = x^3 - 2x + 2, \quad \text{avec } u_0 = 1$$

$$f_3(x) = \sqrt[3]{x}, \quad \text{avec } u_0 = 1$$

$$f_4(x) = \begin{cases} 0 & \text{si } x = 0, \\ x + x^2 \sin(2/x) & \text{sinon,} \end{cases} \quad \text{avec } u_0 = 1$$

$$f_5(x) = x^2, \quad \text{avec } u_0 = 1$$

$$f_6(x) = x + x^{\frac{4}{3}}, \quad \text{avec } u_0 = 1$$

In [34]:

```
### SOLUTION 4 (calculs)
def newtonbavard(f,df,u,tau,nmax):
    n = 0
    fu, dfu = f(u), df(u)
    while (np.abs(fu)>tau) and n<nmax:
        if dfu==0:
            print("Erreur : Division par 0")
            break
        u = u - fu/dfu
        fu, dfu = f(u), df(u)
        n = n + 1
        print("u= {:.3e},   f(u)= {:.2e}".format(u,fu))
    return u, n

nmax = 10
tau= 10**(-12)
```

Solution 4 : (interprétations)

...

haut 1. 2. 3. bas.

3. Méthode de la sécante

Exercice 5. La méthode de Newton nécessite l'évaluation d'une dérivée ($f'(u_n)$) à chaque itération. L'idée de la méthode de la sécante est d'éviter cette évaluation en remplaçant $f'(u_n)$ par le taux d'accroissement :

$$\frac{f(u_n) - f(u_{n-1})}{u_n - u_{n-1}}.$$

La méthode de sécante consiste alors en la construction de la suite itérative $(u_n)_{n\in\mathbb{N}}$ définie par

$$\begin{cases} u_0, u_1 \text{ donnés} \\ u_{n+1} = h(u_n, u_{n-1}), \forall n \in \mathbb{N}^* \end{cases}$$

où

$$h(x,y) = x - f(x)\frac{x-y}{f(x)-f(y)}.$$

1. Ecrire une fonction $secante$ qui a pour paramètres d'entrée f, u_0 et u_1, la tolérance τ et un nombre d'itérations $nmax$. On choisira le-même critère d'arrêt que précédemment. La fonction renvoie l'approximation calculée ainsi que le nombre d'itérations. Dans le cas où on aurait une division par 0 (c-à.d. s'il existe $k < nmax$ tel que $f(u_k) = f(u_{k-1})$), on arrête la boucle while en renvoyant le message <<Erreur : Division par 0>>.

2. Soit $f(x) = \sin(x)$. Utiliser la fonction $newton$ avec $u_0 = 3$ et $u_1 = u_0 + 10^{-1}$ pour déterminer une valeur approchée du zéro π. Faire varier la tolérance ($\tau = 10^{-k}$ pour $1 \leq k \leq 14$). Représenter sur un premier graphique le nombre d'itérations effectué en fonction de τ et sur un second graphique l'erreur $|u_N - \pi|$ en fonction de τ.

3. Soit $f(x) = x^2 - 2$, $u_0 = 1$ et $u_1 = 1.1$. Comparer l'efficacité de la méthode de la sécante à celle de la méthode de Newton.

In [35]:

```
### SOLUTION 5.1, fonction secante
```

In [36]:

```
### SOLUTION 5.2, étude graphique de convergence
f= lambda x:np.sin(x)

nmax=10
```

In [37]:

```
### SOLUTION 5.3, comparaison newton/sécante
f= lambda x:x**2-2
df = lambda x:2*x

nmax=10
```

TP5

MNAP TP 5

Erreur

$e_n = \left| y_n - y_{ex}(t_n) \right|$
$e_{\Delta t} = \max \ e_n \quad \text{pour} \quad 0 \leqq n \leqq N$

La méthode est d'au moins ordre p si $\exists\, C$ tq

$$e_{\Delta t} \leqq C.\ \Delta t^{\,p}$$
$$\log(\, e_{\Delta t}\,) \leqq \log C + p.\ \log(\,\Delta t)$$

La méthode est d'ordre p si elle est au moins d'ordre p mais pas d'ordre q pour q > p.

Exercice 4 :

$\begin{cases} y' = a\,y - b\,y^2 \\ y(0) = y_0 \end{cases}$, montrer que : $y(\,t\,) = \dfrac{a\,y_0}{b\,y_0 + (a - b\,y_0)\,e^{-at}}$

Or $\quad y'(t\,) = \dfrac{a\,(a - b\,y_0)\,e^{-a\,t} \times a\,y_0}{\left(b\,y_0 + (a - b\,y_0)\,e^{-at}\right)^2}$

Vérifions que $\begin{cases} y' = a\,y - b\,y^2 \\ y(0) = y_0 \end{cases}$.

$\begin{cases} Y_{n+1} = f_{\Delta t\,,\,b,\,K}\,(\,Y_n) \\ Y_0 = y_0 \end{cases}$ et $\begin{cases} y' = b\,y\,(\,K - y\,) \\ y(0) = y_0 \end{cases}$ $(\,4\,)$ Problème de Cauchy.

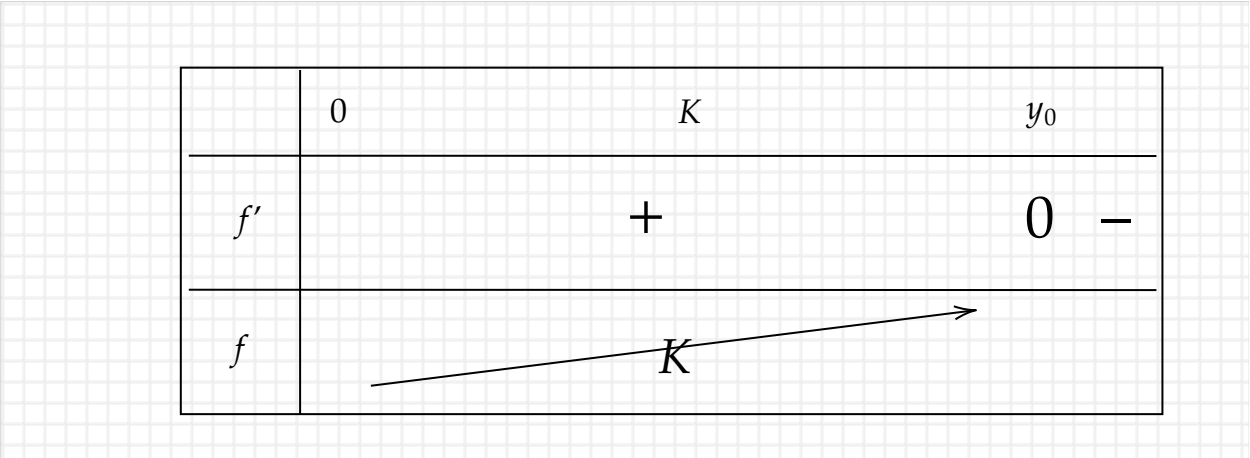
$$f'\,(\,y) = 1 + \Delta t\,.\,K.\ b - 2\,\Delta t - by$$

$$\cdot > 0 \quad si \ y < y_0$$
$$\cdot < 0 \quad si\, y > \ y_0$$

$$y_0 = 1 + \Delta t.K.b$$

$$K = \frac{a}{b}$$

$$y_0 \geq K \iff 2.\Delta t\ .\ K.\ b \leqq 1 + \Delta t\, K\, b$$
$$\iff \Delta t\, K\, b \leqq 1$$



f : bijection croissante $]0,\ K[$ *sur* $]0,\ K[$

Question 4.4)

$y_0 \in]0, K[\ ,\ y_1 = f(y_0) \in]0, K[$

De plus, $y_1 - y_0 = \Delta t.\ b.y_0(K - y_0) > 0$

Donc $0 < y_0 < y_1 < K$

Montrons par récurrence

$$0 < y_0 < y_1 < \ldots < y_n < y_{n+1} < y_{n+2} < K$$

• C'est vrai pour $n = 0$

Supposons vrai au rang n , on a :

$$y_{n+2} = f(y_{n+1}) \in]0, K[\quad car\ y_{n+1} \in]0, K[$$

$$y_{n+2} = f(y_{n+1}) > f(y_n) = y_{n+1} \ car\ f \nearrow \ et\ y_1 < y_{n+1}$$

Donc

$$0 < y_0 < y_1 < \ldots < y_n < y_{n+1} < y_{n+2} < K$$

$(y_n)_{n \in \mathbb{N}} \nearrow$, *majorée par* K .

Donc $\exists\, l \in]0, K[\ \ tq \ \ y_n \xrightarrow[n \to \infty]{} l$

⚠️dire que f est continue !

$$y_{n+1} = f(y_n)$$
$$l = f(l)$$

$$\begin{aligned} l \in]0, K[&\iff l + \Delta t \,.\, b.\, l\,(K - l) = l \\ &\iff l\,(K - l) = 0 \ \ car\ \Delta t\ et\ b\ \ sont\ positifs. \\ &\iff l = K \end{aligned}$$

TP 5. Résolution numérique d'équations différentielles

Sommaire

Nous chargons les deux bibliothèques, *numpy* et *matplotlib.pyplot*.

In [1]:

```
import numpy as np
import matplotlib.pyplot as plt
plt.rcParams.update({'font.size': 18})
plt.rcParams.update({'figure.figsize' : (8, 6)})
```

haut A. B. C. D. bas.

A. Premiers pas sur le modèle malthusien

Rappellons que la modèle de Malthus décrit l'évolution d'une population fermée dont les variations sont dues aux naissances et aux décès. Il conduit à une équation différentielle élémentaire :

$$\begin{cases} y' = ay, \\ y(0) = y_0. \end{cases} \quad (1)$$

Le paramètre a correspond à la différence entre le taux de natalité et le taux de mortalité et y_0 est la taille initiale de la population.

On rappelle également les méthodes numériques présentées dans le cours pour la résolution du problème de Cauchy :

$$\begin{cases} y' = F(y), \\ y(0) = y_0. \end{cases} \quad (2)$$

La fonction F décrit l'évolution et y_0 est la donnée initiale. Pour calculer une solution approchée sur un intervalle $[0, T]$, on se donne une subdivision :

$$t_n = n\Delta t \quad \forall 0 \leq n \leq N, \quad \text{avec } \Delta t = T/N.$$

Méthode d'Euler explicite

Méthode d'Euler explicite

$$\begin{cases} Y_{n+1} \\ = Y_n \\ + \Delta t F \\ (Y_n), \\ Y_0 = y_0 \\ . \end{cases}$$

Méthode d'Euler implicite

$$\begin{cases} Y_{n+1} \\ = Y_n \\ + \Delta t F \\ (Y_{n+1}), \\ Y_0 = y_0 \\ . \end{cases}$$

Méthode de Crank-Nicolson

$$\begin{cases} Y_{n+1} \\ = Y_n \\ + \frac{\Delta t}{2} \\ \Big(F(Y_n) \\ + F \\ (Y_{n+1}) \Big) \\ , \\ Y_0 = y_0 \\ . \end{cases}$$

Méthode de Heun

$$\begin{cases} Y_{n+1} \\ = Y_n \\ + \frac{\Delta t}{2} \\ \Big(F(Y_n) \\ + F \\ \big(Y_n \\ + \Delta t F \\ (Y_n) \big) \Big), \\ Y_0 = y_0 \\ . \end{cases}$$

Exercice 1.

1.1. Pour le modèle de Malthus (1), exprimer Y_{n+1} en fonction de Y_n pour les 4 méthodes proposées .

Méthode d'Euler explicite

$$\begin{cases} Y_{n+1} \\ = Y_n \\ (1 \\ + \Delta t a) \\ , \\ Y_0 = y_0 \\ . \end{cases}$$

Méthode d'Euler implicite

Méthode d'Euler implicite

$$\begin{cases} Y_{n+1} \\ = Y_n \\ + \Delta t a \\ \times Y_{n+1} \\ , \\ Y_{n+1} \\ = \\ (\frac{1}{1} \\ - a \Delta t \\ ) Y_n , \\ Y_0 = y_0 \\ . \end{cases}$$

Méthode de Crank-Nicolson

$$\begin{cases} Y_{n+1} \\ = Y_n \\ + \frac{\Delta t}{2} \\ \Big(a \\ \times Y_n \\ + a \\ \times Y_{n+1} \\ ) \Big) , \\ Y_{n+1} \\ 1 + a \\ \frac{. \Delta t}{2} \\ = \frac{}{1 - a} \\ \frac{. \Delta t}{2} \\ \times Y_n \\ Y_0 = y_0 \\ . \end{cases}$$

Méthode de Heun

$$\begin{cases} Y_{n+1} \\ = Y_n \\ + \frac{\Delta t}{2} \\ \Big(a \\ \times Y_n \\ + F \\ \big(Y_n \\ + \Delta t a \\ \times Y_n \big) \Big) \\ , \\ Y_{n+1} \\ = (1 \\ + \Delta t \\ . a \\ (\Delta t)^2 \end{cases}$$

$$\left\{ \begin{array}{l} (1 \\ + \frac{.a^2}{2}) \\ .Y_n \\ Y_0 = y_0 \\ . \end{array} \right.$$

1.2. Ecrire une fonction *EulerExplMalthus* qui, pour les paramètres d'entrée a, y_0, T et N, calcule la suite $(Y_n)_{0 \leq n \leq N}$.

In [2]:

```
### SOLUTION 1.2
def EulerExplMalthus(a, y0, T, N):
    dt = T / N
    c = (1+a*dt)      # coeff
    Y = np.zeros(N+1)  # derniere valeur du tableu Y
    y = y0
    Y[0] = y0

    for n in range(N):  # N étapes  ( de 0 à N-1)
        y = c*y
        Y[n+1]=y

    return Y
```

1.3. Ecrire de la même façon les fonctions *EulerImplMalthus*, *CNMalthus* et *HeunMalthus*.

In [3]:

```
### SOLUTION 1.3
def EulerImplMalthus(a, y0, T, N):
    dt = T / N
    c = 1/(1-a*dt)     # coeff
    Y = np.zeros(N+1)
    y = y0
    Y[0] = y0

    for n in range(N):  # N étapes  ( de 0 à N-1)
        y = c*y
        Y[n+1]=y

    return Y

def CNMalthus(a, y0, T, N):
    dt = T / N
    c = (1+(a*dt)/2)/(1-(a*dt)/2)      # coeff
    Y = np.zeros(N+1)  # derniere valeur du tableu Y
    y = y0
    Y[0] = y0

    for n in range(N):  # N étapes  ( de 0 à N-1)
        y = c*y
        Y[n+1]=y

    return Y

def HeunMalthus(a, y0, T, N):
    dt = T / N
    c = (1 + a*dt + ((a**2)*(dt)**2/2))     # coeff
    Y = np.zeros(N+1)  # derniere valeur du tableu Y
    y = y0
    Y[0] = y0

    for n in range(N):  # N étapes  ( de 0 à N-1)
        y = c*y
        Y[n+1]=y
```

```
    return Y
```

In [4]:

```
a, y0, T = 2, 1, 1
res = EulerExplMalthus(a,y0,T,N)

print(res)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-4-41ced28d2858> in <module>
      1 a, y0, T = 2, 1, 1
----> 2 res = EulerExplMalthus(a,y0,T,N)
      3
      4 print(res)

NameError: name 'N' is not defined
```

1.4. Pour $a = 2$ et $y_0 = 1$, représenter sur un même graphique la solution exacte et les solutions approchées données par chacune des méthodes sur $[0, 1]$. On pourra comparer les résultats obtenus pour $N = 5$ et $N = 50$. On pourra également tester le cas $a = -1$.

In [ ]:

```
### SOLUTION 1.4
# ========== CAS a=2
a, y0, T = 2, 1, 1

# ========== CAS N=5

# Solution exacte y(t)= exp(a*t)
t_fin = np.linspace(0, T, 6)
Yex = y0*np.exp(a*t_fin)

plt.plot(t_fin, Yex,'m', label="solution exacte")


# Solutions approchées par les différents schémas

YEE = EulerExplMalthus(a,y0,T,N)
YEI = EulerImplMalthus(a,y0,T,N)
YCN = CNMalthus(a,y0,T,N)
YHM = HeunMalthus(a,y0,T,N)



plt.plot(t_fin, YEE,'v',label="Euler explicite" )
plt.plot(t_fin, YEI,'y', label="Euler implicite")
plt.plot(t_fin, YCN,'b',label='Cranck Nicolson')
plt.plot(t_fin, YHM,'g', label="Heun")

#
plt.title("Cas $N=5$")
plt.legend()
plt.show()
```

In [ ]:

```
### SOLUTION 1.4
# ========== CAS a=2
a, y0, T = 2, 1, 1

# ========== CAS N=20

# Solution exacte y(t)= exp(a*t)
t_fin = np.linspace(0, T, 6)
Yex = y0*np.exp(a*t_fin)

plt.plot(t_fin, Yex,'m', label="solution exacte")
```

```
# Solutions approchées par les différents schémas

YEE = EulerExplMalthus(a,y0,T,N)
YEI = EulerImplMalthus(a,y0,T,N)
YCN = CNMalthus(a,y0,T,N)
YHM = HeunMalthus(a,y0,T,N)



plt.plot(t_fin, YEE,'v',label="Euler explicite" )
plt.plot(t_fin, YEI,'y', label="Euler implicite")
plt.plot(t_fin, YCN,'b',label='Cranck Nicolson')
plt.plot(t_fin, YHM,'g', label="Heun")

#
plt.title("Cas $N=20$")
plt.legend()
plt.show()
```

1.5. Ecrire un script qui calcule les erreurs globales entre solution exacte et solution approchée pour toutes les méthodes considérées pour différentes valeurs de N (soit Δt). On commencera par donner une valeur au paramètres : $a = 2$,$y_0 = 1$, $T = 1$ et $k = 7$; on prendra pour valeurs de N : $(N_j = 5 \times 2^j)_{0 \leq j \leq k}$.

In [ ]:

```
### SOLUTION 1.5
a, y0, T, k = 2, 1, 1, 7
N = 5
err_ee, err_ei= np.zeros(k+1), np.zeros(k+1)
err_cn, err_he = np.zeros(k+1), np.zeros(k + 1)
tab_dt=np.zeros(k+1)
meths = ["ee","ei","cn","he"]


for j in range(k+1):
    t = np.linspace(0, T, N+1)
    Yex = y0*np.exp(a*t)
    YEE = EulerExplMalthus(a,y0,T,N)
    YEI = EulerImplMalthus(a,y0,T,N)
    YCN = CNMalthus(a,y0,T,N)
    YHM = HeunMalthus(a,y0,T,N)
    err_ee[j], err_ei[j] = np.max(np.abs(YEE-Yex)), np.max(np.abs(YEI-Yex))
    err_cn[j], err_he[j] = np.max(np.abs(YCN-Yex)), np.max(np.abs(YHM-Yex))
    tab_dt[j] = T/N
    N = 2*N
```

1.6. Représenter graphiquement les erreurs en fonction du pas de temps. Quel est l'ordre observé pour chacune des méthodes ?

In [ ]:

```
plt.loglog(tab_dt, err_ee,'vm',label="erreur Euler explicite" )
plt.loglog(tab_dt, err_ei,'yv', label="erreur Euler implicite")

plt.title("Cas Erreur globale")
plt.legend()
plt.show()

plt.loglog(tab_dt, err_cn,'bv',label='erreur Cranck Nicolson')
plt.loglog(tab_dt, err_he,'gv', label="erreur Heun")

#
plt.title("Cas Erreur globale")
plt.legend()
plt.show()
```

Solution 1.6 (b)

La pente des droites des erreurs de Euler explicite et implicite vaut 1 donc l'ordre de convergence est de 1.

La pente des droites des erreurs de Cranck-Nicolson et Heun vaut 2 donc l'ordre de convergence est de 2.

Exercice 2.

2.1. Ecrire une fonction *EulerExpl* qui calcule la suite $(Y_n)_{0\leq n\leq N}$ donnée par la méthode d'Euler explicite pour le cas général (2). Les paramètres d'entrée sont la fonction F, la donné initiale y_0, l'instant final T et N.

2.2. Même question avec la méthode de Heun.

In [5]:

```
### SOLUTION 2.1 / 2.2
def EulerExpl(F, y0, T, N):
    dt = T / N


    Y = np.zeros(N+1)   # derniere valeur du tableu Y
    y = y0
    Y[0] = y0

    for n in range(N):   # N étapes   ( de 0 à N-1)
        y = y + dt*F(y)
        Y[n+1]=y

    return Y



def Heun(F, y0, T, N):
    dt = T / N


    Y = np.zeros(N+1)   # derniere valeur du tableu Y
    y = y0
    Y[0] = y0



    for n in range(N):   # N étapes   ( de 0 à N-1)
        temp = F(y)
        z = y + dt * temp
        y = y + (dt/2)*(temp+F(z))
        Y[n+1]=y

    return Y
```

2.3. Tester les fonctions sur le modèle malthusien en comparant avec les résultats de l'exercice 1.

In [ ]:

```
### SOLUTION 2.3
a, y0, T = 2, 1, 1
F = lambda x:a*x

N=20




YEEM = EulerExplMalthus(a,y0,T,N)
YEE = EulerExpl(F,y0,T,N)
YHM = HeunMalthus(a,y0,T,N)
YH = Heun(F,y0,T,N)
```

```
print(YEEM-YEE)
print()
print(YHM-YH)
print()
```

2.4. Ecrire une fonction *calculerreurs* qui calcule l'erreur globale entre solution exacte et solution approchée pour une méthode donnée. Les paramètres d'entrée seront F, y_0, la solution exacte Yex, T, k (on prendra toujours pour valeurs de N : $(N_j = 5 \times 2^j)_{0 \leq j \leq k}$) et la méthode *meth*.

In [ ]:

```
### SOLUTION 2.4
meths = ["EEM","HM","EEI","CNM"]

def calculerreurs(F, y0, Yex, T, k, meth):
    N=5
    err = np.zeros(k+1)
    Y = meth(F,y0,T,N)
        for j in range(k+1):
            err[j] = np.max(np.abs(Y-Yex))
            N = 2*N

    return err
```

haut A. B. C. D. bas.

B. Etude du modèle de croissance logistique

Le modèle de croissance logistique est un modèle de population qui prend en compte la compétition entre les individus pour l'accès aux ressources. Il s'écrit :

$$\begin{cases} y' = ay - by^2, \\ y(0) = y_0, \end{cases} \quad (3)$$

où a et b sont deux paramètres. On va supposer dans la suite que $y_0 \in [0, \frac{a}{b}]$

Exercice 3.

1. Montrer que

$$y(t) = \frac{ay_0}{by_0 + (a - by_0)e^{-at}}$$

est solution sur $\mathbb{R}$ de (3).

Solution 3.1

Mettez votre solution ici (ou sur une feuille de papier).

3.2. Calculer $K = \lim_{t \to +\infty} y(t)$ et montrer que le modèle (3) se réécrit

(t)

$$\begin{cases} y' \\ = by \\ (K \\ - y), \\ y(0) \\ = y_0, \end{cases}$$

$$(4)$$

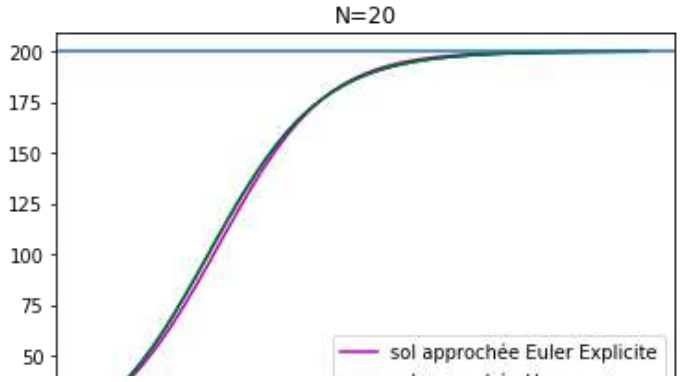
Solution 3.2

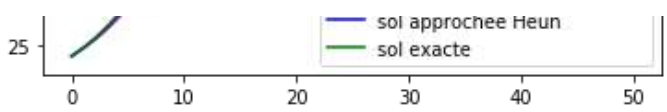
$$K = \lim_{t \to +\infty} y(t) = \frac{a}{b}$$

3.3. Utiliser les fonctions $EulerExpl$ et $Heun$ programmées dans l'exercice 2 pour calculer des solutions approchées du modèle de croissance logistique. Représenter sur un même graphique la solution exacte et les 2 solutions approchées. On pourra prendre $a = 0.2$, $b = 0.001$, $T = 50$. On pourra tester différentes valeurs de N et différentes valeurs de $y_0 \in [0, a/b]$.

In [25]:

```
### SOLUTION 3.3
# parametres du problème
a, b, T = .2, .001, 50
K = a/b
F = lambda y: b*y*(K - y)

# donnée initiale
y0 =.1*K

N = 50

YEE = EulerExpl(F,y0,T,N)

YH = Heun(F,y0,T,N)

t = np.linspace(0,T,N+1)
Yex = (a*y0)/(b*y0+(a-b*y0)*np.exp(-a*t))

plt.plot(t,YEE,'m',label="sol approchée Euler Explicite")
plt.plot(t,YH,'b',label="sol approchée Heun ")
plt.plot(t,YH,'g',label="sol exacte  ")

plt.axhline(y=a/b)

plt.title('N=20')
plt.legend()
plt.show()
```

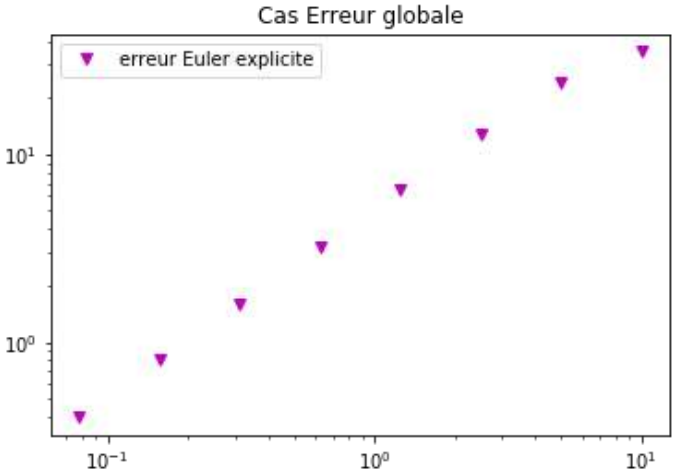


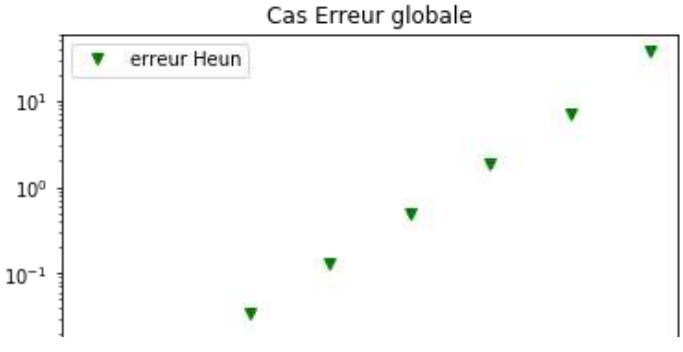


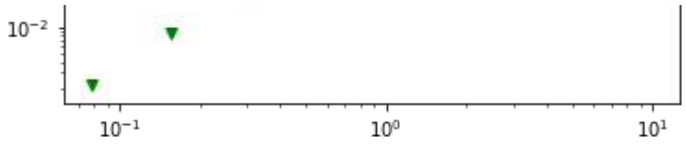
3.4. Mettre en évidence l'ordre de convergence des méthodes d'Euler explicite et Heun sur le modèle de croissance logistique.

In [36]:

```
### SOLUTION 3.4
# Nous procédons comme dans les questions 1.5, 1.6
# Paramètres caractérisant le modèle
a, b = .2, .001
k=7
N = 5
F = lambda y: b*y*(K - y)


err_ee, err_h= np.zeros(k+1), np.zeros(k+1)
tab_dt=np.zeros(k+1)
meths = ["ee","ei","cn","he"]

for j in range(k+1):
    t = np.linspace(0, T, N+1)
    Yex = (a*y0)/(b*y0+(a-b*y0)*np.exp(-a*t))
    YEE = EulerExpl(F,y0,T,N)
    YH = Heun(F,y0,T,N)
    err_ee[j], err_h[j] = np.max(np.abs(YEE-Yex)), np.max(np.abs(YH-Yex))
    tab_dt[j] = T/N
    N = 2*N

plt.loglog(tab_dt, err_ee,'vm',label="erreur Euler explicite" )
#plt.loglog(tab_dt, err_ei,'yv', label="erreur Euler implicite")
plt.title("Cas Erreur globale")
plt.legend()
plt.show()
#plt.loglog(tab_dt, err_cn,'bv',label='erreur Cranck Nicolson')
plt.loglog(tab_dt, err_h,'gv', label="erreur Heun")
#
plt.title("Cas Erreur globale")
plt.legend()
plt.show()
```







La pente des droites des erreurs de Euler explicite et implicite vaut 1 donc l'ordre de convergence est de 1. La pente des droites des erreurs de Cranck-Nicolson et Heun vaut 2 donc l'ordre de convergence est de 2.

Exercice 4. Etude de stabilité de la méthode d'Euler pour le modèle de croissance logistique (4).

4.1. Montrer que le schéma d'Euler explicite appliqué à (4) peut s'écrire sous la forme :

$$\begin{cases} Y_{n+1} \\ = f_{\Delta t,b,K} \\ (Y_n), \\ Y_0 = y_0, \end{cases}$$

où $f_{\Delta t,b,K}$ est une fonction de $\mathbb{R}$ dans $\mathbb{R}$ à définir.

Solution 4.1

Mettez votre solution ici (ou sur une feuille de papier).

4.2. Faire le tableau de variations de la fonction $f_{\Delta t,b,K}$.

Solution 4.2

Mettez votre solution ici (ou sur une feuille de papier).

4.3. Montrer que si $\Delta t \leq \frac{1}{bK}$, $f_{\Delta t,b,K}$ est strictement croissante de $[0,K]$ dans $[0,K]$.

Solution 4.3

Mettez votre solution ici (ou sur une feuille de papier).

4.4. Sous cette même condition et si $y_0 \in]0,$, en déduire que $(Y_n)_{n\geq 0}$ est strictement croissante et à valeurs dans $]0,K[$. Que vaut $\lim_{n\to\infty} Y_n$.
$K[$

Solution 4.4

Mettez votre solution ici (ou sur une feuille de papier).

haut A. B. C. D. bas.

C. Modèles de population à 2 espèces

Considérons maintenant un modèle décrivant l'évolution de deux populations : une population de proies $x(t)$ et une population de prédateurs $y(t)$. Il s'agit du modèle de Lotka-Voltera.

$$\begin{cases} x'(t) & = x(t)(a \\ & - by(t)), \\ y'(t) & = y(t)(-c \\ & + dx(t)), \\ x(0) & = x_0, \\ y(0) & = y_0. \end{cases} \qquad (4)$$

Les quatre paramètres a, b, c, d du modèle sont supposés strictement positifs. Les données initiales x_0, y_0

correspondant aux populations à l'instant 0 sont naturellement supposées positives.

Nous n'avons pas de solution explicite de (4) mais nous pouvons appliquer une méthode numérique pour calculer des solutions approchées. Ici nous allons appliquer le schéma de Heun. Notons

$$Y(t) = \begin{pmatrix} x(t) \\ y(t) \end{pmatrix}$$

et définissons la fonction $F : \mathbb{R}^2 \to \mathbb{R}^2$, par $F\begin{pmatrix} x \\ y \end{pmatrix} := \begin{pmatrix} x(a - by) \\ y(-c + dx) \end{pmatrix}$.

Le système (4) se récrit de manière plus compacte :

$$\begin{cases} Y'(t) &= F(Y(t)) \\ Y(0) &= Y_0. \end{cases}, \qquad (5)$$

avec $Y_0 := \begin{pmatrix} x_0 \\ y_0 \end{pmatrix}$.

Exercice 5.

5.1. Adaptez la fonction *Heun* à la résolution d'un problème à deux espèces. On appelera cette nouvelle fonction *Heun2*.

Indication :Il faut faire attention à la structure des données. Le vecteur *Y0* sera un tableau numpy de taille $(N+1) \times 2$. La solution sera renvoyée sera un tableau numpy de taille $(N+1) \times 2$. La commande pour créer un tableau numpy de taille $(N+1) \times 2$ est *Ysol = np.zeros([N + 1, 2])*.

In [41]:

```
### SOLUTION 5.1
def Heun2(F, Y0, T, N):
    dt = T / N
    Y = np.zeros([N+1,2])
    y = Y0
    Y[0,1] = Y0[1]
    Y[0,0] = Y0[0]
    # derniere valeur du tableu Y
    for n in range(N): # N étapes
        temp = F(y)
        z = y + dt * temp
        y = y + (dt/2)*(temp+F(z))
        Y[n+1]=y
    return Y
```

5.2. Appliquez la méthode de Heun à la résolution approchée du problème (5). On prendra les paramètres qui ont permis de réaliser les graphiques donnés dans le cours : $a = 0.2$, $b = 0.01$, $c = 0.8$, $d = 0.02$, $(x_0, y_0) = (40, 4)$ ou $(40, 10)$, avec $T = 20$.

In [43]:

```
### SOLUTION 5.2
a, b, c, d = 0.2, 0.01, 0.8, 0.02
x0, y0 = 40, 4
F = lambda Y:np.asarray([ Y[0]*(a - b*Y[1]) , Y[1]*(-c + d*Y[0]) ])

Y0 = np.asarray([x0,y0])
T = 20
N=20
```

```
Heun2(F, Y0, T, N)
```

Out[43]:

```
array([[40.        ,  4.        ],
       [46.912     ,  4.256     ],
       [54.71950697,  5.24281134],
       [62.90579812,  7.56142215],
       [69.99196528, 12.68158341],
       [72.4447664 , 23.6091964 ],
       [64.52832764, 42.88172967],
       [46.21822524, 59.64030538],
       [30.49385281, 55.23303209],
       [22.67962369, 40.92425065],
       [19.70177663, 27.93387285],
       [19.22914713, 18.63618537],
       [20.24767679, 12.53100229],
       [22.35504717,  8.67312138],
       [25.41144725,  6.29456334],
       [29.39819137,  4.88118185],
       [34.35397499,  4.12687233],
       [40.33256218,  3.88680354],
       [47.34896115,  4.1670198 ],
       [55.27123691,  5.1828011 ],
       [63.56534547,  7.56158543]])
```

haut A. B. C. D. bas.

D. Implémentation des schémas implicites (Euler implicite et Crank-Nicolson)

Exercice 6.

Les schémas d'Euler implicite et de Crank-Nicolson nécessitent la résolution d'une équation non linéaire à chaque itération. Etant donné Y_n, Y_{n+1} est défini comme le zéro d'une fonction G_n (i.e. $G_n(Y_{n+1}) = 0$) . Pour calculer Y_{n+1}, on va donc utiliser une méthode de Newton, initialisée avec la valeur connue Y_n.

6.1 Rappelons, le schéma donnant Y_{n+1} en fonction de Y_n dans la méthode d'Euler implicite :

$$Y_{n+1} = Y_n + \Delta t F(Y_{n+1}). \quad (6)$$

À partir du schéma (6), donnez une fonction G_n telle que l'équation $G_n(z) = 0$ permette de déterminer Y_{n+1}. Exprimez G_n et sa dérivée en fonction de Y_n, de F et F' et Δt.

6.2 Ecrire une fonction *EulerImpl* qui calcule la suite $(Y_n)_{0 \leq n \leq N}$ donnée par la méthode d'Euler implicite pour le cas général (2). Les paramètres d'entrée sont la fonction F, sa dérivée dF, la donné initiale y_0, l'instant final T et N. Les valeurs de la tolérance et du nombre maximal d'itérations pour la méthode de Newton seront fixées à l'intérieur de la fonction.

6.3. De même nous rappelons le schéma de Crank-Nicolson :

$$Y_{n+1} = Y_n + \frac{\Delta t}{2} (F(Y_n) + F(Y_{n+1})). \quad (7)$$

Reprenez les deux questions précédentes pour ce schéma.

Solution 6.1 / 6.3

(a) Pour la méthode d'Euler Implicite, ...

(b) Pour la méthode de Crank-Nicolson, ...

In [ ]:

```
### SOLUTION 6.2 / 6.3
def EulerImpl(F, dF, y0, T, N):
    pass

def CrankNicolson(F, dF, y0, T, N):
    pass
```

6.4 Tester les deux fonctions sur le modèle de croissance logistique : on tracera dans un premier temps les profils de solution approchée (en les comparant à la solution exacte) puis on élaborera les courbes d'erreur pour mettre en évidence l'ordre des 2 méthodes.

In [ ]:

```
### SOLUTION 6.4 (a) Profils de solutions approchés

# parametres du problème
a, b, T = .2, .001, 50
K =
F = lambda y: b*y*(K - y)
dF = lambda y: b*(K - 2*y)

# donnée initiale
y0 =.1*K

pass
```

In [ ]:

```
### SOLUTION 6.4 (b) Étude numérique de la convergence des méthodes d'Euler
#                   implicite et de Crank-Nicolson.

# Paramètres caractérisants le modèle
a, b, T = .2, .001, 50
K = a/b
F = lambda y: b*y*(K - y)
dF = lambda y: b*(K - 2*y)

# donnée initiale
y0 = .1*K

pass
```

haut A. B. C. D. bas.

TP 5 - CORRECTION

TP 4. Résolution numérique d'équations différentielles

Sommaire

Nous chargons les deux bibliothèques, *numpy* et *matplotlib.pyplot*.

In [ ]:

```
import numpy as np
import matplotlib.pyplot as plt
plt.rcParams.update({'font.size': 18})
plt.rcParams.update({'figure.figsize' : (8, 6)})
```

haut A. B. C. D. bas.

A. Premiers pas sur le modèle malthusien

Rappellons que la modèle de Malthus décrit l'évolution d'une population fermée dont les variations sont dues aux naissances et aux décès. Il conduit à une équation différentielle élémentaire :

$$\begin{cases} y' = ay, \\ y(0) = y_0. \end{cases} \qquad (1)$$

Le paramètre a correspond à la différence entre le taux de natalité et le taux de mortalité et y_0 est la taille initiale de la population.

On rappelle également les méthodes numériques présentées dans le cours pour la résolution du problème de Cauchy :

$$\begin{cases} y' = F(y), \\ y(0) = y_0. \end{cases} \qquad (2)$$

La fonction F décrit l'évolution et y_0 est la donnée initiale. Pour calculer une solution approchée sur un intervalle $[0, T]$, on se donne une subdivision :

$$t_n = n\Delta t \quad \forall 0 \leq n \leq N, \text{ avec } \Delta t = T/N.$$

Méthode d'Euler explicite

$$\begin{cases} Y_{n+1} = Y_n + \Delta t F(Y_n), \\ Y_0 = y_0. \end{cases}$$

Méthode d'Euler implicite

$$\begin{cases} Y_{n+1} = Y_n + \Delta t F(Y_{n+1}), \\ Y_0 = y_0. \end{cases}$$

Méthode de Crank-Nicolson

$$\begin{cases} Y_{n+1} = Y_n + \dfrac{\Delta t}{2}\left(F(Y_n) + F(Y_{n+1})\right), \\ Y_0 = y_0. \end{cases}$$

Méthode de Heun

$$\begin{cases} Y_{n+1} = Y_n + \dfrac{\Delta t}{2}\left(F(Y_n) + F\left(Y_n + \Delta t F(Y_n)\right)\right), \\ Y_0 = y_0. \end{cases}$$

Exercice 1.

1.1. Pour le modèle de Malthus (1), exprimer Y_{n+1}
en fonction de Y_n
pour les 4 méthodes proposées .

Solution 1.1. Dans le cas du problème de Malthus, on a $F(y) = ay$.

a) Méthode d'Euler explicite : la formule donne $\,y*** **{n+1} = y_n + \Delta t a y** ***n,\,$ soit $y_{n+1} = (1 + a\Delta t)y_n$.

b) Méthode d'Euler explicite : la formule donne $\,y **{n+1} = y*** n ***+ \Delta t a y*** **{n+1}\,, soit_**
\displaystyle y_{n+1} = \dfrac1{1 - a\Delta t}y_n.$

c) Méthode de Crank Nicolson : on a $\,y **{n+1} = y_n + (\Delta t/2) a (y*** ***n + y*** **{n+1}),\, soit_**
\displaystyle y_{n+1} = \dfrac{1 + a\Delta t/2}{1 - a\Delta t/2}y_n\,$,

d) Méthode de Heun : la formule donne $\,y*** **{n+1} = y_n + (\Delta t/2) (a y_n + a(y_n + \Delta t a y*** ***n))$, soit

$$y_{n+1} = \left(1 + a\Delta t + a^2(\Delta t)^2/2\right)y_n.$$

Remarque : Dans le cas des méthodes implicites (b) et (c), et pour une fonction F ***donnée, nous ne savons pas en général donner d'expression explicite de $y*** **{n+1}*** $en fonction de$ **y*** n . Le cas linéaire **F(y)=ay** est ici une exception rare. Dans le cas général, on sait seulement que **y{n+1}** est solution d'une équation du type **g(y)=0$ qu'on résout de manière approchée (à l'aide d'une méthode de point fixe pa exemple)._

1.2. Ecrire une fonction ***EulerExplMalthus*** **qui, pour les paramètres d'entrée** a
, y_0
, T
et N
, calcule la suite $(Y_n)_{0 \le n \le N}$
.

In [ ]:

```
### SOLUTION 1.2
def EulerExplMalthus(a, y0, T, N):
    Y = np.zeros(N+1)
    dt = T/N
    coef = 1 + a*dt
    Y[0], y = y0, y0
    for i in range(N):
        y = coef*y
        Y[i + 1] = y
    return Y
```

1.3. Ecrire de la même façon les fonctions *EulerImplMalthus*, *CNMalthus* et *HeunMalthus*.

In [ ]:

```
### SOLUTION 1.3
def EulerImplMalthus(a, y0, T, N):
    Y = np.zeros(N+1)
    dt = T/N
    coef = 1/(1 - a*dt)
    Y[0], y = y0, y0
    for i in range(N):
        y = coef*y
        Y[i + 1] = y
    return Y

def CNMalthus(a, y0, T, N):
    Y = np.zeros(N+1)
    dt = T/N
    coef = (1 + .5*a*dt)/(1 - .5*a*dt)
    Y[0], y = y0, y0
    for i in range(N):
        y = coef*y
        Y[i + 1] = y
    return Y

def HeunMalthus(a, y0, T, N):
    Y = np.zeros(N+1)
    dt = T/N
    coef = 1 + a*dt + .5*(a*dt)**2
    Y[0], y = y0, y0
    for i in range(N):
        y = coef*y
        Y[i + 1] = y
    return Y
```

1.4. Pour $a = 2$
et $y_0 = 1$
, représenter sur un même graphique la solution exacte et les solutions approchées données par chacune des méthodes sur $[0, 1]$
. On pourra comparer les résultats obtenus pour $N = 5$
et $N = 50$
. On pourra également tester le cas $a = -1$
.

In [ ]:

```
### SOLUTION 1.4
# ========== CAS a=2
a, y0, T = 2, 1, 1
#    Solution exacte y(t)= exp(a*t)
t_fin = np.linspace(0, T, 200)
Yex = y0*np.exp(a*t_fin)

#      ------ le cas N=5
N = 5
tn = np.linspace(0, T, N + 1)    # Vecteur des instants tn=n Delta t (n=0,...,N)
```

```
##             Caculs
Yee = EulerExplMalthus(a, y0, T, N)
Yei = EulerImplMalthus(a, y0, T, N)
Ycn = CNMalthus(a, y0, T, N)
Yhe = HeunMalthus(a, y0, T, N)

#               Representation graphique
plt.plot(t_fin, Yex,'r', label="solution exacte")
plt.plot(tn, Yee,'mo--', label="Euler explicite")
plt.plot(tn, Yei, 'bo--', label="Euler implicite")
plt.plot(tn, Ycn, 'go--', label="Crank-Nicolson")
plt.plot(tn, Yhe, 'yo--', label="Heun")
plt.legend()
plt.title("a=2, N=5")
plt.show()

#      ------ le cas N=50
N = 50
tn = np.linspace(0, T, N + 1)   # Vecteur des instants tn=n Delta t (n=0,...,N)

##             Caculs
Yee = EulerExplMalthus(a, y0, T, N)
Yei = EulerImplMalthus(a, y0, T, N)
Ycn = CNMalthus(a, y0, T, N)
Yhe = HeunMalthus(a, y0, T, N)

#             Representation graphique
plt.plot(t_fin, Yex,'r', label="solution exacte")
plt.plot(tn, Yee,'m.--', label="Euler explicite")
plt.plot(tn, Yei, 'b.--', label="Euler implicite")
plt.plot(tn, Ycn, 'g.--', label="Crank-Nicolson")
plt.plot(tn, Yhe, 'y.--', label="Heun")
plt.legend()
plt.title("a=2, N=50")
plt.show()

# ============  CAS a=-1
a, y0, T = -1, 1, 1
# Solution exacte y(t)= exp(a*t)
t_fin = np.linspace(0, T, 200)
Yex = y0*np.exp(a*t_fin)

#     ------- le cas N=5
N = 5
tn = np.linspace(0, T, N + 1)   # Vecteur des instants tn=n Delta t (n=0,...,N)

#             Caculs
Yee = EulerExplMalthus(a, y0, T, N)
Yei = EulerImplMalthus(a, y0, T, N)
Ycn = CNMalthus(a, y0, T, N)
Yhe = HeunMalthus(a, y0, T, N)

#           Representation graphique
plt.plot(t_fin, Yex,'r', label="solution exacte")
plt.plot(tn, Yee,'mo--', label="Euler explicite")
plt.plot(tn, Yei, 'bo--', label="Euler implicite")
plt.plot(tn, Ycn, 'go--', label="Crank-Nicolson")
plt.plot(tn, Yhe, 'yo--', label="Heun")
plt.legend()
plt.title("a=-1, N=5")
plt.show()

#      ------- le cas N=50
N = 50
tn = np.linspace(0, T, N + 1)   # Vecteur des instants tn=n Delta t (n=0,...,N)

#               Caculs
Yee = EulerExplMalthus(a, y0, T, N)
Yei = EulerImplMalthus(a, y0, T, N)
Ycn = CNMalthus(a, y0, T, N)
Yhe = HeunMalthus(a, y0, T, N)
```

```
#                Representation graphique
plt.plot(t_fin, Yex,'r', label="solution exacte")
plt.plot(tn, Yee,'m.--', label="Euler explicite")
plt.plot(tn, Yei, 'b.--', label="Euler implicite")
plt.plot(tn, Ycn, 'g.--', label="Crank-Nicolson")
plt.plot(tn, Yhe, 'y.--', label="Heun")
plt.legend()
plt.title("a=-1, N=50")
plt.show()
```

1.5. Ecrire un script qui calcule les erreurs globales entre solution exacte et solution approchée pour toutes les méthodes considérées pour différentes valeurs de N (soit Δt). On commencera par donner une valeur au paramètres : $a = 2$,$y_0 = 1$, $T = 1$ et $k = 7$; on prendra pour valeurs de N : $(N_j = 5 \times 2^j)_{0 \leq j \leq k}$.

In [ ]:

```
### SOLUTION 1.5
a, y0, T, k = 2, 1, 1, 7
N = 5
# initialisation des tableaux d'erreur et de pas de temps
err_ee, err_ei= np.zeros(k+1), np.zeros(k+1)
err_cn, err_he = np.zeros(k+1), np.zeros(k + 1)
tab_dt = np.zeros(k+1)
for j in range(k + 1):
    tn = np.linspace(0,T , N + 1) # vecteur des points de la grille de discrétisation
    Yex = y0*np.exp(a*tn)         # solution exacte en ces points
    ## Calcul des solution approchées
    Yee = EulerExplMalthus(a, y0, T, N)
    Yei = EulerImplMalthus(a, y0, T, N)
    Ycn = CNMalthus(a, y0, T, N)
    Yhe = HeunMalthus(a, y0, T, N)
    ## Calcul des erreurs
    err_ee[j] = np.max(np.abs(Yee - Yex))
    err_ei[j] = np.max(np.abs(Yei - Yex))
    err_cn[j] = np.max(np.abs(Ycn - Yex))
    err_he[j] = np.max(np.abs(Yhe - Yex))
    ## Stokage su pas de temps
    tab_dt[j] = T/N
    N = 2*N
```

1.6. Représenter graphiquement les erreurs en fonction du pas de temps. Quel est l'ordre observé pour chacune des méthodes ?

In [ ]:

```
### SOLUTION 1.6 (a)

# Représentenation en échelle logarithmique de l'erreur
# en fonction de Delta t pour les méthodes d'Euler
plt.loglog(tab_dt, err_ee,'mo--', label="Euler explicite")
plt.loglog(tab_dt, err_ei, 'bo--', label="Euler implicite")
plt.legend()
plt.title("Erreurs en fonction du pas de temps")
plt.show()

# Représentenation en échelle logarithmique de l'erreur
# en fonction de Delta t pour les méthodes de Crank-Nicolson et Heun
plt.loglog(tab_dt, err_cn, 'go--', label="Crank-Nicolson")
plt.loglog(tab_dt, err_he, 'yo--', label="Heun")
plt.legend()
```

```
plt.title("Erreurs en fonction du pas de temps")
plt.show()
```

Solution 1.6 (b)

On observe une erreur d'ordre 1 (pente 1) pour les méthodes d'Euler explicite et d'Euler implicite.

On observe une erreur d'ordre 2 (pente 2) pour les méthodes de Crank-Nicolson et de Heun.

Exercice 2.

2.1. Ecrire une fonction *EulerExpl* qui calcule la suite $(Y_n)_{0\leq n\leq N}$ donnée par la méthode d'Euler explicite pour le cas général (2). Les paramètres d'entrée sont la fonction F , la donné initiale y_0 , l'instant final T et N .

2.2. Même question avec la méthode de Heun.

In [ ]:

```
### SOLUTION 2.1 / 2.2
def EulerExpl(F, y0, T, N):
    Y =np.zeros(N + 1)
    dt = T/N
    Y[0], y= y0, y0
    for i in range(N):
        y = y + dt*F(y)
        Y[i + 1] = y
    return Y

def Heun(F, y0, T, N):
    Y =np.zeros(N + 1)
    dt = T/N
    Y[0], y= y0, y0
    for i in range(N):
        Fy = F(y)
        y = y + .5*dt*(Fy + F(y + dt*Fy))
        Y[i + 1] = y
    return Y
```

2.3. Tester les fonctions sur le modèle malthusien en comparant avec les résultats de l'exercice 1.

In [ ]:

```
### SOLUTION 2.3
a, y0, T = 2, 1, 1
F = lambda x:a*x
N=30
Yee1=EulerExplMalthus(a, y0, T, N)
Yee2=EulerExpl(F, y0, T, N)
print("différence pour la méthode d'Euler explicte :", Yee1 - Yee2, "\n")

Yhe1=HeunMalthus(a, y0, T, N)
Yhe2=Heun(F, y0, T, N)
print("différence pour la méthode de Heun :", Yhe1 - Yhe2)

print("""\nLes différences observées sont négligeables.
Elles sont dues aux troncatures dans les calculs (erreurs d'arrondis).
""")
```

2.4. Ecrire une fonction *calculerreurs* qui calcule l'erreur globale entre solution exacte et solution approchée pour une méthode donnée. Les paramètres d'entrée seront F , y_0 , la solution exacte Yex T

, k
(on prendra toujours pour valeurs de N
: $(N_j = 5 \times 2^j)_{0 \le j \le k}$
) et la méthode *meth*.

In [ ]:

```
### SOLUTION 2.4
def calculerreurs(F, y0, Yex, T, k, meth):
    err = np.zeros(k + 1)
    N = 5
    for j in range(k + 1):
        Yapp = meth(F, y0, T, N)
        err[j] = np.max(np.abs(Yapp - Yex))
        N = 2*N
    return err
```

haut A. B. C. D. bas.

B. Etude du modèle de croissance logistique

Le modèle de croissance logistique est un modèle de population qui prend en compte la compétition entre les individus pour l'accès aux ressources. Il s'écrit :

$$\begin{cases} y' = ay - by^2, \\ y(0) = y_0, \end{cases} \qquad (3)$$

où a
et b
sont deux paramètres. On va supposer dans la suite que
$y_0 \in [0, \frac{a}{b}]$.

Exercice 3.

1. Montrer que

$$y(t) = \frac{ay_0}{by_0 + (a - by_0)e^{-at}}$$

est solution sur R
de (3).

Solution 3.1

On prend la solution proposée : $y(t) = \frac{ay_0}{by_0 + (a - by_0)e^{-at}}$.

Commençons par vérifier que la condition initiale est satisfaite. On a

$$y(0) = \frac{ay_0}{by_0 + (a - by_0)e^0} = \frac{ay_0}{by_0 + (a - by_0)} = y_0.$$

On vérifie ensuite que

y ***est solution de l'edo. On calcule d'abord :*** $y'(t) = \frac{a^2 y_0 (a - by_0)e^{-at}}{\left(by_0 + (a - by_0)e^{-at}\right)^2}$.

On soustrait en suite

$ay(t)$ **et on réduit au même dénominateur :**

$$y'(t) - ay(t) = \frac{a^2 y_0 \left[(a - by_0)e^{-at} - \left(by_0 + (a - by_0)e^{-at} \right) \right]}{\left(by_0 + (a - by_0)e^{-at} \right)^2}.$$

_Les termes $(a - by_0)e^{-at}$
se simplifient au numérateur et on arrive à

$$y'(t) - ay(t) = \frac{-ba^2 y_0^2}{\left(by_0 + (a - by_0)e^{-at} \right)^2} = -b(y(t))^2.$$

Ainsi y
est bien solution de (3)._

3.2. Calculer $K = \lim_{t \to +\infty} y(t)$
et montrer que le modèle (3) se réécrit

$$\begin{cases} y' = by(K - y), \\ y(0) = y_0, \end{cases} \qquad (4)$$

Solution 3.2

_On passe à la limite dans la formule $y(t) = \frac{ay_0}{by_0 + (a - by_0)e^{-at}}$.
Comme $a > 0$
, on a $\lim_{t \to +\infty} e^{-at} = 0$
. D'où_

$$K := \lim_{t \to +\infty} y(t) = \frac{a}{b}.$$

On peut alors écrire $ay - by^2 = bKy - by^2 = by(K - y)$
et (3) peut se mettre sous la forme (4).

3.3. Utiliser les fonctions *EulerExpl*
et *Heun*
programmées dans l'exercice 2 pour calculer des solutions approchées du modèle de croissance logistique. Représenter sur un même graphique la solution exacte et les 2 solutions approchées. On pourra prendre $a = 0.2$
, $b = 0.001$
, $T = 50$
. On pourra tester différentes valeurs de N
et différentes valeurs de $y_0 \in [0, a/b]$
.

In [ ]:

```
### SOLUTION 3.3
# parametres du problème
a, b, T = .2, .001, 50
K=a/b
F = lambda y: b*y*(K - y)

# donnée initiale
y0 = .5*K

# -----  y0 =.5*K, N = 10 -----
```

```
# paramètres de le discrétisation
N=10
tn = np.linspace(0, T, N + 1)

# solution exacte aux points de la grille
Yex = a*y0/(b*y0 + (a - b*y0)*np.exp(-a*tn))

# Calcul des solutions approchées
Yee = EulerExpl(F, y0, T, N)
Yhe = Heun(F, y0, T, N)

# Représentation graphique des solutions (exacte et approchées
pretitre = "Solutions exacte du modèle de croissance logistique"

plt.plot(tn, Yex,'ro', label="Solution Exacte")
plt.plot(tn, Yee,'go--', label="Euler explicite")
plt.plot(tn, Yhe, 'bo--', label="Heun")
plt.legend()
plt.title(pretitre + ", y0={:.0f} (K={:.0f}), N={:d}".format(y0,K,N))
plt.show()

# -----  y0 =.5*K, N = 20 -----
# paramètres de le discrétisation
N=20
tn = np.linspace(0, T, N + 1)

# solution exacte aux points de la grille
Yex = a*y0/(b*y0 + (a - b*y0)*np.exp(-a*tn))

# Calcul des solutions approchées
Yee = EulerExpl(F, y0, T, N)
Yhe = Heun(F, y0, T, N)

# Représentation graphique des solutions (exacte et approchées)
plt.plot(tn, Yex,'r', label="Solution Exacte")
plt.plot(tn, Yee,'g.--', label="Euler explicite")
plt.plot(tn, Yhe, 'b.--', label="Heun")
plt.legend()
plt.title(pretitre + ", y0={:.0f} (K={:.0f}), N={:d}".format(y0,K,N))
plt.show()

# -----  y0 =.5*K, N = 40 -----
# paramètres de le discrétisation
N=40
tn = np.linspace(0, T, N + 1)

# solution exacte aux points de la grille
Yex = a*y0/(b*y0 + (a - b*y0)*np.exp(-a*tn))

# Calcul des solutions approchées
Yee = EulerExpl(F, y0, T, N)
Yhe = Heun(F, y0, T, N)

# Représentation graphique des solutions (exacte et approchées)
plt.plot(tn, Yex,'r.', label="Solution Exacte")
plt.plot(tn, Yee,'g.--', label="Euler explicite")
plt.plot(tn, Yhe, 'b--', label="Heun")
plt.legend()
plt.title(pretitre + ", y0={:.0f} (K={:.0f}), N={:d}".format(y0,K,N))
plt.show()

# ========== y0=.05*K ==================
# donnée initiale
y0 =.05*K

# -----  y0 =.05*a/b, N = 10 -----
# paramètres de le discrétisation
N=10
tn = np.linspace(0, T, N + 1)

# solution exacte aux points de la grille
Yex = a*y0/(b*y0 + (a - b*y0)*np.exp(-a*tn))
```

```
# Calcul des solutions approchées
Yee = EulerExpl(F, y0, T, N)
Yhe = Heun(F, y0, T, N)

# Représentation graphique des solutions (exacte et approchées)
plt.plot(tn, Yex,'ro', label="Solution Exacte")
plt.plot(tn, Yee,'go--', label="Euler explicite")
plt.plot(tn, Yhe, 'bo--', label="Heun")
plt.legend()
plt.title(pretitre + ", y0={:.0f} (K={:.0f}), N={:d}".format(y0,K,N))
plt.show()

# -----  y0 =.05*a/b, N = 20 -----
# paramètres de le discrétisation
N=20
tn = np.linspace(0, T, N + 1)

# solution exacte aux points de la grille
Yex = a*y0/(b*y0 + (a - b*y0)*np.exp(-a*tn))

# Calcul des solutions approchées
Yee = EulerExpl(F, y0, T, N)
Yhe = Heun(F, y0, T, N)

# Représentation graphique des solutions (exacte et approchées)
plt.plot(tn, Yex,'r', label="Solution Exacte")
plt.plot(tn, Yee,'g.--', label="Euler explicite")
plt.plot(tn, Yhe, 'b.--', label="Heun")
plt.legend()
plt.title(pretitre + ", y0={:.0f} (K={:.0f}), N={:d}".format(y0,K,N))
plt.show()

# -----  y0 =.05*a/b, N = 40 -----
# paramètres de le discrétisation
N=40
tn = np.linspace(0, T, N + 1)

# solution exacte aux points de la grille
Yex = a*y0/(b*y0 + (a - b*y0)*np.exp(-a*tn))

# Calcul des solutions approchées
Yee = EulerExpl(F, y0, T, N)
Yhe = Heun(F, y0, T, N)

# Représentation graphique des solutions (exacte et approchées)
plt.plot(tn, Yex,'r.', label="Solution Exacte")
plt.plot(tn, Yee,'g.--', label="Euler explicite")
plt.plot(tn, Yhe, 'b--', label="Heun")
plt.legend()
plt.title(pretitre + ", y0={:.0f} (K={:.0f}), N={:d}".format(y0,K,N))
plt.show()
```

3.4. Mettre en évidence l'ordre de convergence des méthodes d'Euler explicite et Heun sur le modèle de croissance logistique.

In [ ]:

```
### SOLUTION 3.4
# Nous procédons comme dans les questions 1.5, 1.6
# Paramètres caractérisants le modèle
a, b = .2, .001
K = a/b
F = lambda y: b*y*(K - y)

# Durée d'intérêt et donnée initiale
T, y0 = .05*K, 50

k, N = 7, 5
# initialisation des tableaux d'erreur et de pas de temps
```

```
err_ee, err_he, tab_dt = np.zeros(k+1), np.zeros(k + 1), np.zeros(k + 1)
for j in range(k + 1):
    tn = np.linspace(0,T , N + 1)        # points de la grille de discrétisation
    Yex = a*y0/(b*y0 + (a - b*y0)*np.exp(-a*tn)) # solution exacte en ces points
    ## Calcul des solution approchées
    Yee = EulerExpl(F, y0, T, N)
    Yhe = Heun(F, y0, T, N)
    ## Calcul des erreurs
    err_ee[j] = np.max(np.abs(Yee - Yex))
    err_he[j] = np.max(np.abs(Yhe - Yex))
    ## Stokage du pas de temps
    tab_dt[j] = T/N
    N = 2*N


# Représenttnation en échelle logarithmique de l'erreur
# en fonction de Delta t pour la méthode d'Euler explicite
plt.loglog(tab_dt, err_ee,'mo--', label="Euler explicite")
plt.legend()
plt.title("Erreurs en fonction du pas de temps")
plt.show()

# Représentenation en échelle logarithmique de l'erreur
# en fonction de Delta t pour la méthode de Heun
plt.loglog(tab_dt, err_he, 'yo--', label="Heun")
plt.legend()
plt.title("Erreurs en fonction du pas de temps")
plt.show()

print("""
        On observe à nouveau un odre de convergence de 1 pour la méthode
        d'Euler explicite et de 2 pour la méthode de Heun.""")
```

Exercice 4. **Etude de stabilité de la méthode d'Euler pour le modèle de croissance logistique (4).**

4.1. **Montrer que le schéma d'Euler explicite appliqué à (4) peut s'écrire sous la forme :**

$$\begin{cases} Y_{n+1} = f_{\Delta t,\, b,\, K}(Y_n), \\ Y_0 = y_0, \end{cases}$$

où $f_{\Delta t,\, b,\, K}$
est une fonction de R
dans R
à définir.

Solution 4.1

On écrit le shéma au pas de temps
$n \geq 0$ ***$$ y{n+1} = y_n + \Delta t F(y_n) = y_n + \Delta t by_n(K - y_n) = y_n(1+\Delta t b(K - y n)) = f{\Delta t,b,K}(y n), $$ où on pose***

$$f_{\Delta t,\, b,\, K}(y) := y(1 + \Delta t b(K - y)).$$

4.2. **Faire le tableau de variations de la fonction** $f_{\Delta t,\, b,\, K}$

.

Solution 4.2

***La fonction $f{\Delta t,b,K}** estdeclasse*
C^\infty. *Calculonssadérivée.*
Pour $y \in$ R, $\quad f'_{\Delta t,\, b,\, K}(y) = 1 + \Delta t K b - 2\Delta t b y.$
Onvoitque
$f'_{\Delta t,\, b,\, K}(y) = 0 \iff y = y_*$
avec

avec

$y_* := \frac{1 + \Delta t K b}{2 \Delta t b} > 0.$

De plus_

$f'_{\Delta t, b, K}(y) > 0 \quad \text{pour } y < y_* \qquad \text{et} \qquad f'_{\Delta t, b, K}(y) < 0 \quad \text{pour } y > y_*.$

$

La fonction $f{\Delta t,b,K}*estdoncsrictementcroissantesur*
]-\infty,y**]***
*puissrictementdécroissantesur****[y*,+\infty,[$. On a aussi_***

$$\lim_{y \to -\infty} f_{\Delta t, b, K}(y) = -\infty, \qquad \max_{\mathbb{R}} f_{\Delta t, b, K} = f_{\Delta t, b, K}(y_*) = \frac{(1 + \Delta t K b)^2}{4 \Delta t b}, \qquad \lim_{y \to +\infty} f_{\Delta t, b, K}(y) = -\infty.$$

4.3. Montrer que si $\Delta t \leq \frac{1}{bK}$
, $f_{\Delta t, b, K}$
est strictement croissante de $[0, K]$
dans $[0, K]$
.

Solution 4.3

Supposons
$\Delta t \leq \frac{1}{bK}$***,***
soit :
$\Delta t K b \leq 1$***. On alors $$ y = \dfrac{1 + \Delta t K b }{2\Delta t b}\ge \dfrac1{\Delta t b}\ge K. $$ Comme $f {\Delta t,b,K}***
*eststrictementcroissantesur****[0,y]****, onendéduitqu*′ *elleeststrictementcroissantesur*
[0,K]$._

De plus $f{\Delta t,b,K}([0,K])=[f{\Delta t,b,K}(0),f{\Delta t,b,K}(K)]*car*
f{\Delta t,b,K}*
*estcontinueetcroissantesur****[0,K]$.***

On calcule ensuite $f{\Delta t,b,K}(0)=0*et*
f{\Delta t,b,K}(K)=K$.*

Conclusion : si
$\Delta t \leq \frac{1}{bK}$***, $f{\Delta t,b,K}****eststrictementcroissantede*
[0,K]*sur*
[0,K]$._

4.4. Sous cette même condition et si $y_0 \in]0, K[$
, en déduire que $(Y_n)_{n \geq 0}$
est strictement croissante et à valeurs dans $]0, K[$
. Que vaut $\lim_{n \to \infty} Y_n$
.

Solution 4.4

_(a) On montre par récurrence sur $n \geq 0$
que $y_n \in]0, K]$
et $y_{n+1} > y_n$
._

_Initialisation. Pour $n = 0$
, on a $y_0 \in]0, K]$
et

$$y_1 = f_{\Delta t, b, K}(y_0) > by_0(K - y_0) > 0\$,$$

car $b > 0$
,$y > 0$
et $K - y_0 > 0$
._

_Hérédité. Soit $n \geq 0$
et supposons $y_n \in]0, K[$
et $y_{n+1} > y_n$
. On a d'abord $y_{n+1} = f_{\Delta t, b, K}(y_n).$
Comme d'après la question précédente $]0, K[$
est stable par $f_{\Delta t, b, K}$
et que $y_n \in]0, K[$
par hypothèse de récurrence, on a bien_

$$y_{n+1} = f_{\Delta t, b, K}(y_n) \in]0, K[.$$

De plus, $\,y{n+2} - y{n+1} = f{\Delta t,b,K}(y{n+1}) - f{\Delta t,b,K}(yn) >0
, car ***f{\Delta t,b,K}*** *eststrictementcroissantesur*
]0,K[*et*
y_{n+1}>y*n$ par hypothèse de récurrence. On a bien aussi :*

$$y_{n+2} > y_{n+1}.$$

_Conclusion. Par principe de récurrence, $(y_n) \subset]0, K[$
pour $n \geq 0$
et la suite (y_n)
est strictement croissante._

haut **A.** **B.** **C.** **D.** **bas.**

C. Modèles de population à 2 espèces

Considérons maintenant un modèle décrivant l'évolution de deux populations : une population de proies $x(t)$
et une population de prédateurs $y(t)$
. Il s'agit du modèle de Lotka-Voltera.

$$\begin{cases} x'(t) &= x(t)(a - by(t)), \\ y'(t) &= y(t)(-c + dx(t)), \\ x(0) &= x_0, \\ y(0) &= y_0. \end{cases} \qquad (4)$$

Les quatre paramètres a
, b
, c
, d
du modèle sont supposés strictement positifs. Les données initiales x_0
, y_0
correspondant aux populations à l'instant 0
sont naturellement supposées positives.

Nous n'avons pas de solution explicite de (4) mais nous pouvons appliquer une méthode numérique pour calculer des solutions approchées. Ici nous allons appliquer le schéma de Heun. Notons

$Y(t) = \begin{pmatrix} x(t) \\ y(t) \end{pmatrix}$ et définissons la fonction $F: \mathrm{R}^2 \to \mathrm{R}^2$, par $F\begin{pmatrix} x \\ y \end{pmatrix} := \begin{pmatrix} x(a - by) \\ y(-c + dx) \end{pmatrix}$.

Le système (4) se récrit de manière plus compacte :

$$\begin{cases} Y'(t) &= F(Y(t)), \\ Y(0) &= Y_0. \end{cases} \qquad (5)$$

avec $Y_0 := \begin{pmatrix} x_0 \\ y_0 \end{pmatrix}$

.

Exercice 5.

5.1. Adaptez la fonction *Heun* à la résolution d'un problème à deux espèces. On appelera cette nouvelle fonction *Heun2*.

***Indication* :Il faut faire attention à la structure des données. Le vecteur *Y0* sera un tableau numpy de taille 2. La solution sera renvoyée sera un tableau numpy de taille** $(N+1) \times 2$
. La commande pour créer un tableau numpy de taille $(N+1) \times 2$
est *Ysol = np.zeros([N + 1, 2])*.

In [ ]:

```
### SOLUTION 5.1
def Heun2(F, y0, T, N):
    Y =np.zeros([N + 1, 2])
    dt = T/N
    Y[0], y= y0, y0
    for i in range(N):
        Fy = F(y)
        y = y + .5*dt*(Fy + F(y + dt*Fy))
        Y[i + 1] = y
    return Y
```

5.2. Appliquez la méthode de Heun à la résolution approchée du problème (5). On prendra les paramètres qui ont permis de réaliser les graphiques donnés dans le cours : $a = 0.2$
, $b = 0.01$
, $c = 0.8$
, $d = 0.02$
, $(x_0, y_0) = (40, 4)$
ou $(40, 10)$
, avec $T = 20$
.

In [ ]:

```
### SOLUTION 5.2
a, b, c, d = 0.2, 0.01, 0.8, 0.02
x0, y0 = 40, 4
F = lambda Y:np.asarray([ Y[0]*(a - b*Y[1]) , Y[1]*(-c + d*Y[0]) ])

Y0 = np.asarray([x0,y0])
T = 40

for N in [40,80,160]:
    Ysol = Heun2(F,Y0,T,N)
    tn = np.linspace(0, T, N + 1)
    plt.plot(tn,Ysol[:,0], '.', label="N={}".format(N))
    #plt.plot(tn,Ysol[:,1], 'o', label="N={}".format(N))
    plt.title("Solutions approchées d'une population, N = {}".format(N))
    plt.legend()
```

```
    #plt.show()
plt.show()

for N in [160, 320, 640]:
    Ysol = Heun2(F,Y0,T,N)
    tn = np.linspace(0, T, N + 1)
    plt.plot(tn,Ysol[:,0], '.', label="proies")
    plt.plot(tn,Ysol[:,1], '.', label="prédateurs")
    plt.title("Solutions approchées des deux populations".format(N))
    plt.legend()
    #plt.show()
plt.show()
```

haut A. B. C. D. bas.

D. Implémentation des schémas implicites (Euler implicite et Crank-Nicolson)

Exercice 6.

Les schémas d'Euler implicite et de Crank-Nicolson nécessitent la résolution d'une équation non linéaire à chaque itération. Etant donné Y_n
, Y_{n+1}
est défini comme le zéro d'une fonction G_n
(i.e. $G_n(Y_{n+1}) = 0$
). Pour calculer Y_{n+1}
, on va donc utiliser une méthode de Newton, initialisée avec la valeur connue Y_n
.

6.1 Rappelons, le schéma donnant Y_{n+1}
en fonction de Y_n
dans la méthode d'Euler implicite :

$$Y_{n+1} = Y_n + \Delta t F(Y_{n+1}). \qquad (6)$$

À partir du schéma (6), donnez une fonction G_n
telle que l'équation $G_{n+1}(z) = 0$
permette de déterminer Y_{n+1}
. Exprimez G_n
et sa dérivée en fonction de Y_n
, de F
et F'
.

6.2 Ecrire une fonction *EulerImpl* qui calcule la suite $(Y_n)_{0 \le n \le N}$
donnée par la méthode d'Euler implicite pour le cas général (2). Les paramètres d'entrée sont la fonction F
, sa dérivée dF
, la donné initiale y_0
, l'instant final T
et N
. Les valeurs de la tolérance et du nombre maximal d'itérations pour la méthode de Newton seront fixées à l'intérieur de la fonction.

6.3. De même nous rappelons le schéma de Crank-Nicolson :

$$Y_{n+1} = Y_n + \frac{\Delta t}{2}(F(Y_n) + F(Y_{n+1})). \qquad (7)$$

Reprenez les deux questions précédentes pour ce schéma.

Solution 6.1 / 6.3

(a) Pour la méthode d'Euler Implicite, un choix naturel est $G_n(z) := z - Y_n - \Delta t F(z)$

.

Sa dérivée est $G_n'(z) = 1 - \Delta t F'(z)$

.

(b) Pour la méthode de Crank-Nicolson, on prend $G_n(z) := z - Y_n - \frac{\Delta t}{2}(F(Y_n) + F(z))$

. $

On a $G_n'(z) = 1 - \frac{\Delta t}{2} F'(z)$

.

In [ ]:

```
### SOLUTION 6.2 / 6.3
def EulerImpl(F, dF, y0, T, N):
    Y =np.zeros(N + 1)
    dt = T/N
    Y[0], y= y0, y0
    tau_nw = 1e-12   # tolérance pour la méthode de Newton
    nmax_nw = 10      # nombre maximal d'itérations pour la méthode de Newton
    for n in range(N):
        # Initialisation de la méthode de Newton
        z = y
        Gz = - dt*F(z)
        niter_nw = 0
        # itérations de Newton
        while np.abs(Gz)>tau_nw and niter_nw < nmax_nw:
            z = z - Gz/(1 - dt*dF(z))
            niter_nw += 1
            Gz = z - y - dt*F(z)
        # Attribution et sockage de y_n+1
        y = z
        Y[n + 1] = y
    return Y

def CrankNicolson(F, dF, y0, T, N):
    Y =np.zeros(N + 1)
    dt = T/N
    Y[0], y= y0, y0
    tau_nw = 1e-12
    nmax_nw = 10
    ds = dt/2
    for n in range(N):
        # Initialisation de la méthode de Newton
        z = y
        q = - y - ds*F(y)
        Gz = z + q - ds*F(z)
        niter_nw = 0
        # itérations de Newton
        while np.abs(Gz)>tau_nw and niter_nw < nmax_nw:
            z = z - Gz/(1 - ds*dF(z))
            niter_nw += 1
            Gz = z + q - ds*F(z)
        # Attribution et stockage de y_n+1
        y = z
        Y[n + 1] = y
    return Y
```

6.4 Tester les deux fonctions sur le modèle de croissance logistique : on tracera dans un premier temps les profils de solution approchée (en les comparant à la solution exacte) puis on élaborera les courbes d'erreur pour mettre en évidence l'ordre des 2 méthodes.

In [ ]:

```
### SOLUTION 6.4 (a) Profils de solutions approchés
```

```
# parametres du problème
a, b, T = .2, .001, 50
K=a/b
F = lambda y: b*y*(K - y)
dF = lambda y: b*(K - 2*y)

# donnée initiale
y0 =.5*K

# -----  y0 =.5*K, N = 10 -----
# paramètres de le discrétisation
N=10
tn = np.linspace(0, T, N + 1)

# solution exacte aux points de la grille
Yex = a*y0/(b*y0 + (a - b*y0)*np.exp(-a*tn))

# Calcul des solutions approchées
Yei = EulerImpl(F, dF, y0, T, N)
Ycn = CrankNicolson(F, dF, y0, T, N)

# Représentation graphique des solutions (exacte et approchées
pretitre = "Solutions exacte du modèle de croissance logistique"

plt.plot(tn, Yex,'ro', label="Solution Exacte")
plt.plot(tn, Yei,'go--', label="Euler implicite")
plt.plot(tn, Ycn, 'bo--', label="Crank-Nicolson")
plt.legend()
plt.title(pretitre + ", y0={:.0f} (K={:.0f}), N={:d}".format(y0,K,N))
plt.show()

# -----  y0 =.5*a/b, N = 20 -----
# paramètres de le discrétisation
N=20
tn = np.linspace(0, T, N + 1)

# solution exacte aux points de la grille
Yex = a*y0/(b*y0 + (a - b*y0)*np.exp(-a*tn))

# Calcul des solutions approchées
Yei = EulerImpl(F, dF, y0, T, N)
Ycn = CrankNicolson(F, dF, y0, T, N)

# Représentation graphique des solutions (exacte et approchées)
plt.plot(tn, Yex,'r', label="Solution Exacte")
plt.plot(tn, Yei,'g.--', label="Euler implicite")
plt.plot(tn, Ycn, 'b.--', label="Crank-Nicolson")
plt.legend()
plt.title(pretitre + ", y0={:.0f} (K={:.0f}), N={:d}".format(y0,K,N))
plt.show()

# -----  y0 =.5*a/b, N = 40 -----
# paramètres de le discrétisation
N=40
tn = np.linspace(0, T, N + 1)

# solution exacte aux points de la grille
Yex = a*y0/(b*y0 + (a - b*y0)*np.exp(-a*tn))

# Calcul des solutions approchées
Yei = EulerImpl(F, dF, y0, T, N)
Ycn = CrankNicolson(F, dF, y0, T, N)

# Représentation graphique des solutions (exacte et approchées)
plt.plot(tn, Yex,'r.', label="Solution Exacte")
plt.plot(tn, Yei,'g.--', label="Euler implicite")
plt.plot(tn, Ycn, 'b--', label="Crank-Nicolson")
plt.legend()
plt.title(pretitre + ", y0={:.0f} (K={:.0f}), N={:d}".format(y0,K,N))
plt.show()
```

```
# ========== y0=.05*K ==================
# donnée initiale
y0 =.05*K

# -----  y0 =.05*a/b, N = 10 -----
# paramètres de le discrétisation
N=10
tn = np.linspace(0, T, N + 1)

# solution exacte aux points de la grille
Yex = a*y0/(b*y0 + (a - b*y0)*np.exp(-a*tn))

# Calcul des solutions approchées
Yei = EulerImpl(F, dF, y0, T, N)
Ycn = CrankNicolson(F, dF, y0, T, N)

# Représentation graphique des solutions (exacte et approchées)
plt.plot(tn, Yex,'ro', label="Solution Exacte")
plt.plot(tn, Yei,'go--', label="Euler implicite")
plt.plot(tn, Ycn, 'bo--', label="Crank-Nicolson")
plt.legend()
plt.title(pretitre + ", y0={:.0f} (K={:.0f}), N={:d}".format(y0,K,N))
plt.show()

# -----  y0 =.05*a/b, N = 20 -----
# paramètres de le discrétisation
N=20
tn = np.linspace(0, T, N + 1)

# solution exacte aux points de la grille
Yex = a*y0/(b*y0 + (a - b*y0)*np.exp(-a*tn))

# Calcul des solutions approchées
Yei = EulerImpl(F, dF, y0, T, N)
Ycn = CrankNicolson(F, dF, y0, T, N)

# Représentation graphique des solutions (exacte et approchées)
plt.plot(tn, Yex,'r', label="Solution Exacte")
plt.plot(tn, Yei,'g.--', label="Euler implicite")
plt.plot(tn, Ycn, 'b.--', label="Crank-Nicolson")
plt.legend()
plt.title(pretitre + ", y0={:.0f} (K={:.0f}), N={:d}".format(y0,K,N))
plt.show()

# -----  y0 =.05*a/b, N = 40 -----
# paramètres de le discrétisation
N=40
tn = np.linspace(0, T, N + 1)

# solution exacte aux points de la grille
Yex = a*y0/(b*y0 + (a - b*y0)*np.exp(-a*tn))

# Calcul des solutions approchées
Yei = EulerImpl(F, dF, y0, T, N)
Ycn = CrankNicolson(F, dF, y0, T, N)

# Représentation graphique des solutions (exacte et approchées)
plt.plot(tn, Yex,'r.', label="Solution Exacte")
plt.plot(tn, Yei,'g.--', label="Euler implicite")
plt.plot(tn, Ycn, 'b--', label="Crank-Nicolson")
plt.legend()
plt.title(pretitre + ", y0={:.0f} (K={:.0f}), N={:d}".format(y0,K,N))
plt.show()
```

In [ ]:

```
### SOLUTION 6.4 (b) Étude numérique de la convergence des méthodes d'Euler
#                   implicite et de Crank-Nicolson.

# Nous procédons comme dans les questions 1.5, 1.6 et 3.4
```

```
# Paramètres caractérisants le modèle
a, b, T = .2, .001, 50
K = a/b
F = lambda y: b*y*(K - y)
dF = lambda y: b*(K - 2*y)

# Donnée initiale
y0 = .1*K

k, N = 7, 5
# initialisation des tableaux d'erreur et de pas de temps
err_ei, err_cn, tab_dt = np.zeros(k+1), np.zeros(k + 1), np.zeros(k + 1)
for j in range(k + 1):
    tn = np.linspace(0,T , N + 1)        # points de la grille de discrétisation
    Yex = a*y0/(b*y0 + (a - b*y0)*np.exp(-a*tn)) # solution exacte en ces points
    ## Calcul des solution approchées
    Yei = EulerImpl(F, dF, y0, T, N)
    Ycn = CrankNicolson(F, dF, y0, T, N)
    ## Calcul des erreurs
    err_ei[j] = np.max(np.abs(Yei - Yex))
    err_cn[j] = np.max(np.abs(Ycn - Yex))
    ## Stockage du pas de temps
    tab_dt[j] = T/N
    N = 2*N

# Représentation en échelle logarithmique de l'erreur
# en fonction de Delta t pour la méthode d'Euler implicite
plt.loglog(tab_dt, err_ei,'mo--', label="Euler implicite")
plt.legend()
plt.title("Erreurs en fonction du pas de temps")
plt.show()

# Représentation en échelle logarithmique de l'erreur
# en fonction de Delta t pour la méthode de Crank-Nicolson
plt.loglog(tab_dt, err_cn, 'yo--', label="Crank-Nicolson")
plt.legend()
plt.title("Erreurs en fonction du pas de temps")
plt.show()

print("""
        On observe un odre de convergence de 1 pour la méthode
        d'Euler explicite et de 2 pour la méthode de Crank-Nicolson.""")
```

haut A. B. C. D. bas.