Equadif

December 17, 2020

1 Résolution numérique des équations différentielles

Marc Lorenzi

15 décembre 2020

[1]:
```
import matplotlib.pyplot as plt
import math
```

[2]:
```
plt.rcParams['figure.figsize'] = (10, 10)
```

Dans tout ce notebook les fonctions seront définies sur un segment $[a, b]$ de $\mathbb{R}$, à valeurs dans $\mathbb{R}$ ou parfois dans $\mathbb{R}^p$ $(p \geq 2)$.

Notations :

- On note la « variable » par la lettre t.
- Lorsque la fonction est à valeurs réelles, on la note par une lettre minuscule du genre x, u, v, etc. Par exemple, si l'on résout l'équation différentielle $x' = x + t$, on notera une solution de l'équation par la lettre x, l'image du réel t par la fonction x est notée $x(t)$, etc.
- Les fonctions à valeurs dans $\mathbb{R}^p$ sont des fonctions à valeurs vectorielles, on les notera de préférence par des lettres majuscules, $X, Z, etc.$

Les fonctions seront aussi régulières que nécessaire.

1.1 1. Introduction

1.1.1 1.1 Équation d'ordre 1 : l'idée générale

Soit

$$(\mathcal{E}) \quad x' = F(t, x)$$

une équation différentielle linéaire d'ordre 1 « résolue en x' ». Soit $x_0 \in \mathbb{R}$. Soient $a, b \in \mathbb{R}$, $a < b$. Sous certaines conditions de régularité sur la fonction F que je n'expliciterai pas ici, il existe une unique solution $x : [a, b] \to \mathbb{R}$ de l'équation différentielle sur le segment $[a, b]$ telle que $x(a) = x_0$.

Supposons connue une **méthode** qui nous permet, pour $t \in [a, b]$ et h « petit », de calculer à partir d'une valeur approchée de $x(t)$, une valeur approchée de $x(t + h)$. Si l'on connaît $x(a)$, une simple

boucle permet alors de calculer une valeur approchée de x en les points d'une subdivision de $[a, b]$ de pas h, en appliquant itérativement cette méthode.

Tout réside dans le sens du mot « approchée ». Quelle est la précision de **l'approximation** ? Tout dépend évidemment de la **méthode**. Nous pouvons déjà écrire une fonction générale de résolution approchée d'une équation différentielle d'ordre 1 …

La fonction `dsolve` prend en paramètres :

- Une fonction F, qui est le « F » de l'équation différentielle à résoudre.
- Deux réels a et b qui sont les bornes de l'intervalle sur lequel on résout.
- Un réel x_0, la valeur initiale de notre solution en a.
- Une entier n qui est le nombre de points de $[a, b]$ en lesquels nous voulons une valeur approchée de la solution.
- Une fonction `method`, qui est … la **méthode** permettant de calculer $x(t + h)$ à partir de $x(t)$, connaissant $x(t)$, F, t et h. Un appel à `method(F, t, x, h)` renvoie une valeur approchée de $x(t + h)$ lorsque le paramètre `x` est une valeur approchée de $x(t)$.

La fonction `dsolve` renvoie un couple (ts, xs) où ts est la liste des points d'une subdivision régulière de $[a, b]$ de pas $\frac{b-a}{n}$ et xs est la liste des valeurs approchées de la solution x aux points de la subdivision.

Remarque : les professionnels ne disent pas « méthode » mais **schéma**. Étant un amateur, nous continuerons à employer le mot « méthode ».

[3]:
```
def subdivision(a, b, n):
    h = (b - a) / n
    return [a + k * h for k in range(n + 1)]
```

[4]:
```
def dsolve(F, a, b, x0, n, method):
    h = (b - a) / n
    ts = subdivision(a, b, n)
    xs = [x0]
    x = x0
    for k in range(n):
        t = ts[k]
        x = method(F, t, x, h)
        xs.append(x)
    return (ts, xs)
```

Testons .. avec une méthode absurde :-), juste pour voir si la fonction fonctionne. Résolvons $x' = x$ sur $[0, 1]$ avec $x(0) = 1$ et une subdivision de 10 points.

[5]:
```
F = lambda t, x: -x
method = lambda F, t, x, h: 0
print(dsolve(F, 0, 1, 1, 10, method))
```

```
([0.0, 0.1, 0.2, 0.30000000000000004, 0.4, 0.5, 0.6000000000000001,
0.7000000000000001, 0.8, 0.9, 1.0], [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

La solution cherchée est l'exponentielle. Évidemment la liste $[1, 0, 0, \ldots, 0]$ est une approximation

assez effroyable des valeurs de $x \mapsto e^x$ aux points $0, \frac{1}{10}, \frac{2}{10}, \ldots, 1$. Mais on s'en doutait. On a évidemment compris que tout réside dans la **méthode**.

1.1.2 1.2 Équation d'ordre 2

Nous allons bientôt voir pourquoi il est utile de considérer des équations différentielles

$$X' = G(t, X)$$

où $X : [a, b] \to \mathbb{R}^p$ est une fonction à valeurs **vectorielles** dans $\mathbb{R}^p$. Une telle fonction X est définie par une égalité du type

$$X(t) = (x_1(t), \ldots, x_p(t))$$

où les $x_i : [a, b] \to \mathbb{R}$ sont des fonctions à valeurs réelles. Dériver X, c'est tout simplement dériver chacune des fonctions x_i :

$$X'(t) = (x_1'(t), \ldots, x_p'(t))$$

S'intéresser à des fonctions à valeurs vectorielles c'est juste pour faire *tendance* ? Non, pas du tout ! Considérons une équation différentielle d'ordre 2,

$$x'' = F(t, x, x')$$

Posons $X = (x, x')$. Posons également

$$G(t, (u, v)) = (v, F(t, u, v))$$

Alors

$$\begin{array}{rcl} X' & = & (x', x'') \\ & = & (x', F(t, x, x')) \\ & = & G(t, (x, x')) \\ & = & G(t, X) \end{array}$$

Bref, résoudre l'équation différentielle d'ordre 2, $x'' = F(t, x, x')$ revient à résoudre l'équation vectorielle d'ordre 1 :

$$X' = G(t, X)$$

Bien sûr, ceci peut se généraliser à des équations différentielles d'ordre 3, 4, etc. Par exemple, pour une équation d'ordre 3 on prendrait $X = (x, x', x'')$.

La fonction `dsolve2` ci-dessous permet de "résoudre" des équations d'ordre 2. Ici, x_0 et x_1 sont respectivement les valeurs de la fonction x et de sa dérivée x' au point a. Les autres paramètres ont la même signification que pour la fonction `dsolve`.

[6]:
```
def dsolve2(F, a, b, x0, x1, n, method):
    X0 = [x0, x1]
    G = lambda t, X: (X[1], F(t, X[0], X[1]))
    ts, Xs = dsolve(G, a, b, X0, n, method)
    return (ts, [X[0] for X in Xs])
```

Le seul souci que pour l'instant on ne peut résoudre aucune équation vu qu'il nous manque des méthodes. Mais c'est pour ça qu'on est en prépa, pour acquérir des méthodes, non ? Nous allons en voir deux :

- La **méthode d'Euler** est très facile à implémenter, intuitivement claire, … et très inefficace.
- La **méthode de Runge-Kutta** est un peu plus délicate à écrire, pas du tout intuitive … et terriblement efficace.

1.2 2. Zone de repos …

Nous aurons besoin sous peu d'additionner des vecteurs de $\mathbb{R}^p$ et de les multiplier par un réel. Nous représenterons ces vecteurs en Python par des listes de longueur p. Écrivons des fonctions d'addition et de multiplication par un réel.

[7]:
```
def mul(X, mu):
    return [x * mu for x in X]
```

[8]:
```
mul([1, 3, 2], 5)
```

[8]:
```
[5, 15, 10]
```

Question : pourquoi `mu` et pas `lambda` ???

[9]:
```
def add(X1, X2):
    return [X1[k] + X2[k] for k in range(len(X1)) ]
```

[10]:
```
add([1, 2, 6], [3, 2, 4])
```

[10]:
```
[4, 4, 10]
```

1.3 3. Euler

1.3.1 3.1 La méthode, enfin

Considérons l'équation différentielle

$$x' = F(t, x)$$

Soit x une fonction solution de cette équation sur un intervalle I. Soit $t \in I$. Pour h réel tel que $x + h \in I$, on a

$$x(t+h) = x(t) + \int_t^{t+h} x'(u)\,du = x(t) + \int_t^{t+h} F(u, x(u))\,du$$

Si h est « petit », une approximation très crue de l'intégrale est

$$\int_t^{t+h} F(u, x(u))\,du \simeq hF(t, x(t))$$

On peut donc espérer (il faut être optimistes) que

$$x(t+h) \simeq x(t) + hF(t, x(t))$$

Je n'irai pas plus avant dans la théorie, ce n'est pas le but de ce notebook. Mais nous tenons là une méthode. Est-elle efficace ?

[11]:
```
def euler_method(F, t, x, h):
    dx = F(t, x) * h
    return x + dx
```

1.3.2 3.2 Itération

[12]:
```
def euler(F, a, b, x0, n):
    return dsolve(F, a, b, x0, n, euler_method)
```

Résolvons l'équation $y' = y$ sur $[0, 2]$ avec une valeur initiale de 1 en 0. La solution exacte est bien sûr la fonction exponentielle.

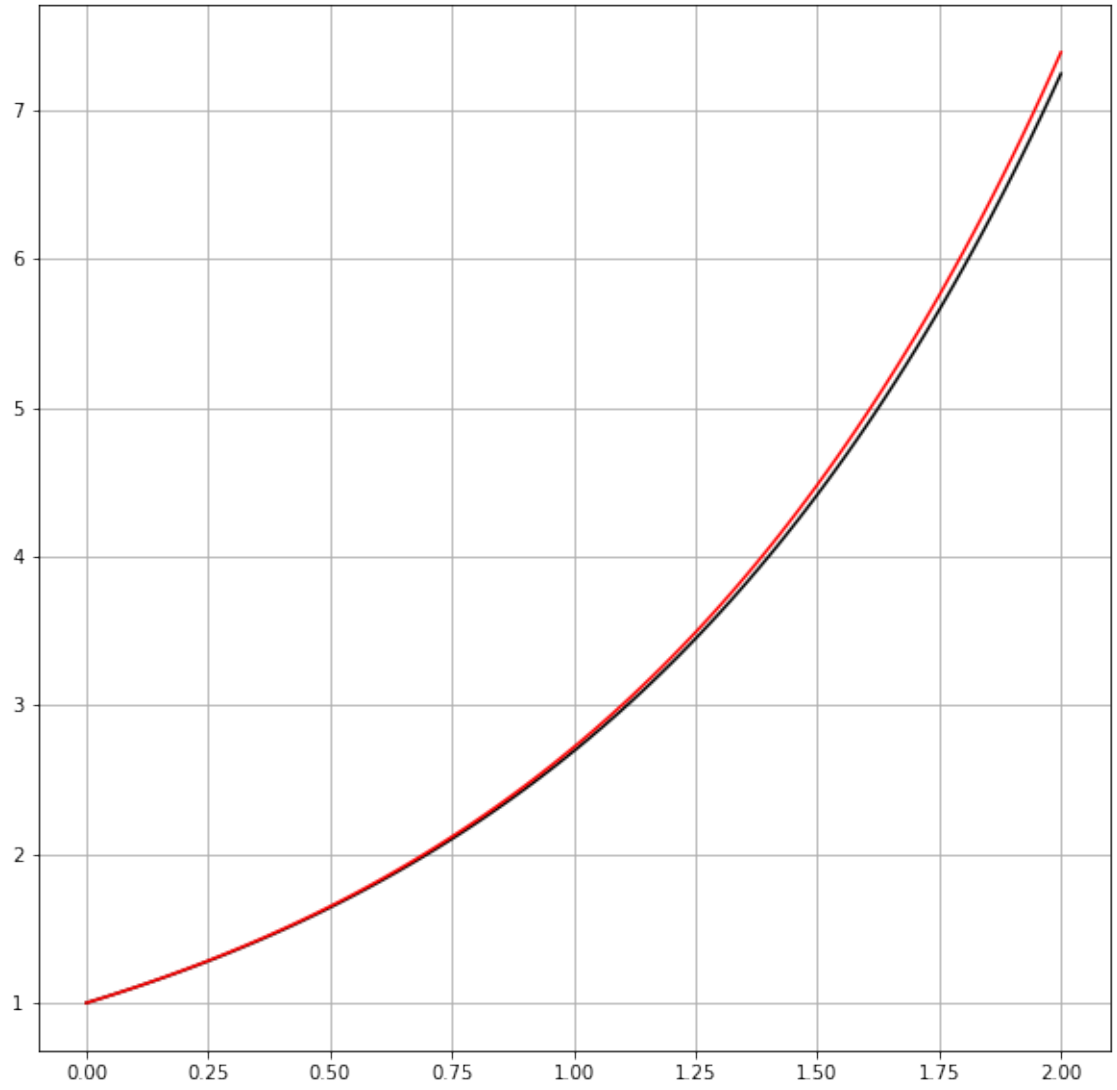
[13]:
```
ts, xs = euler(lambda t, x:x, 0, 2, 1, 100)
print(xs)
```

```
[1, 1.02, 1.0404, 1.061208, 1.08243216, 1.1040808032, 1.126162419264,
1.14868566764928, 1.1716593810022657, 1.195092568622311, 1.2189944199947573,
1.2433743083946525, 1.2682417945625455, 1.2936066304537963, 1.3194787630628722,
1.3458683383241297, 1.3727857050906123, 1.4002414191924246, 1.4282462475762732,
1.4568111725277986, 1.4859473959783545, 1.5156663438979217, 1.5459796707758802,
1.5768992641913977, 1.6084372494752257, 1.6406059944647302, 1.6734181143540248,
1.7068864766411052, 1.7410242061739272, 1.7758446902974057, 1.8113615841033537,
1.8475888157854208, 1.8845405921011291, 1.9222314039431516, 1.9606760320220147,
1.999889552662455, 2.039887343715704, 2.080685090590018, 2.1222987924018186,
2.164744768249855, 2.2080396636148523, 2.252200456887149, 2.297244466024892,
2.3431893553453897, 2.3900531424522975, 2.4378542053013437, 2.4866112894073704,
2.536343515195518, 2.587070385499428, 2.638811793209417, 2.691588029073605,
2.7454197896550774, 2.800328185448179, 2.8563347491571425, 2.9134614441402853,
```

```
2.971730673023091, 3.031165286483553, 3.091788592213224, 3.1536243640574884,
3.216696851338638, 3.2810307883654106, 3.346651404132719, 3.4135844322153734,
3.481856120859681, 3.5514932432768744, 3.622523108142412, 3.6949735703052604,
3.7688730417113656, 3.844250502545593, 3.921135512596505, 3.9995582228484348,
4.079549387305404, 4.161140375051512, 4.244363182552543, 4.329250446203593,
4.415835455127665, 4.504152164230218, 4.5942352075148225, 4.686119911665119,
4.779842309898421, 4.875439156096389, 4.972947939218317, 5.072406898002683,
5.173855035962736, 5.277332136681991, 5.382878779415631, 5.490536355003944,
5.600347082104023, 5.712354023746104, 5.826601104221027, 5.943133126305447,
6.061995788831556, 6.183235704608188, 6.3069004187003515, 6.433038427074359,
6.561699195615846, 6.692933179528163, 6.826791843118726, 6.9633276799811,
7.102594233580723, 7.244646118252337]
```

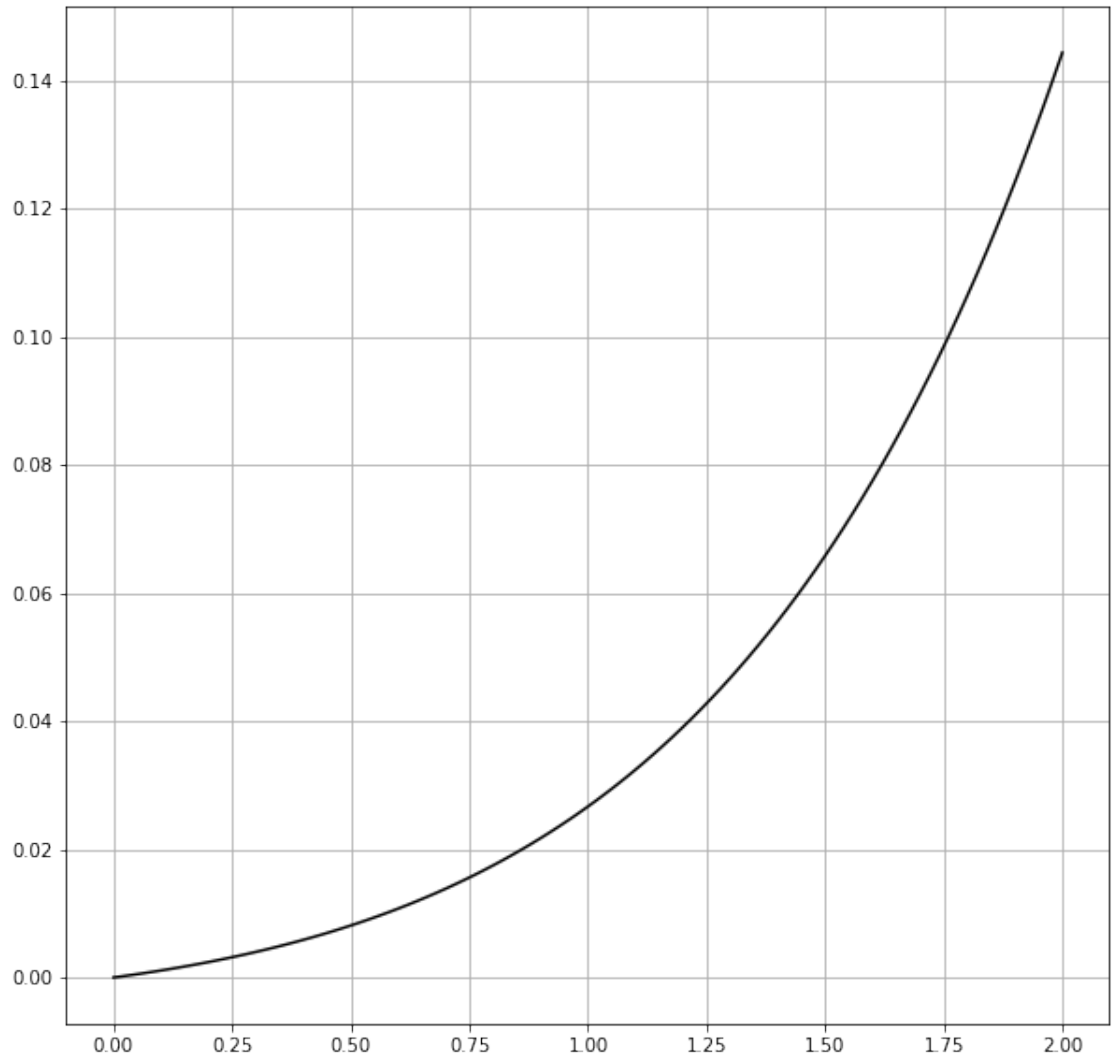
Traçons sur un même graphe la solution approchée et la solution exacte.

[14]:
```
ts, xs = euler(lambda t, x:x, 0, 2, 1, 100)
plt.plot(ts, xs, 'k')
plt.plot(ts, [math.exp(t) for t in ts], 'r')
plt.grid()
plt.show()
```



Quelle est l'erreur commise ?

[15]:
```
err = [abs(xs[k] - math.exp(ts[k])) for k in range(len(ts))]
plt.plot(ts, err, 'k')
plt.grid()
plt.show()
```



Une erreur de 0.14 à la dernière étape, ce n'est pas très bon, c'est le moins qu'on puisse dire. On peut montrer que la méthode d'Euler est ce que l'on appelle une **méthode d'ordre 1** : si on appelle E_k l'erreur commise à l'étape k, on a pour tout k, **SI** h est assez petit,

$$E_k \leq Ch$$

où C est une « constante » et h est le pas de la méthode (le dernier paramètre de `euler_method`). Cette « constante » C dépend en fait de tout sauf de h. Elle dépend en particulier exponentiellement de la longueur de l'intervalle où l'on résout l'équation différentielle.

1.3.3 2.3 Fonctions à valeurs vectorielles

On recopie ici ce que l'on a fait pour les fonctions à valeurs réelles, sauf que l'on met des `mul` et des `add` à la place des multiplications et des additions.

[16]:
```
def eulerV_method(G, t, X, h):
    dX = mul(G(t, X), h)
    return add(X, dX)
```

[17]:
```
def eulerV(G, a, b, X0, n):
    return dsolve(G, a, b, X0, n, eulerV_method)
```

Testons sur un petit exemple sans prétention. Considérons le **système différentiel**

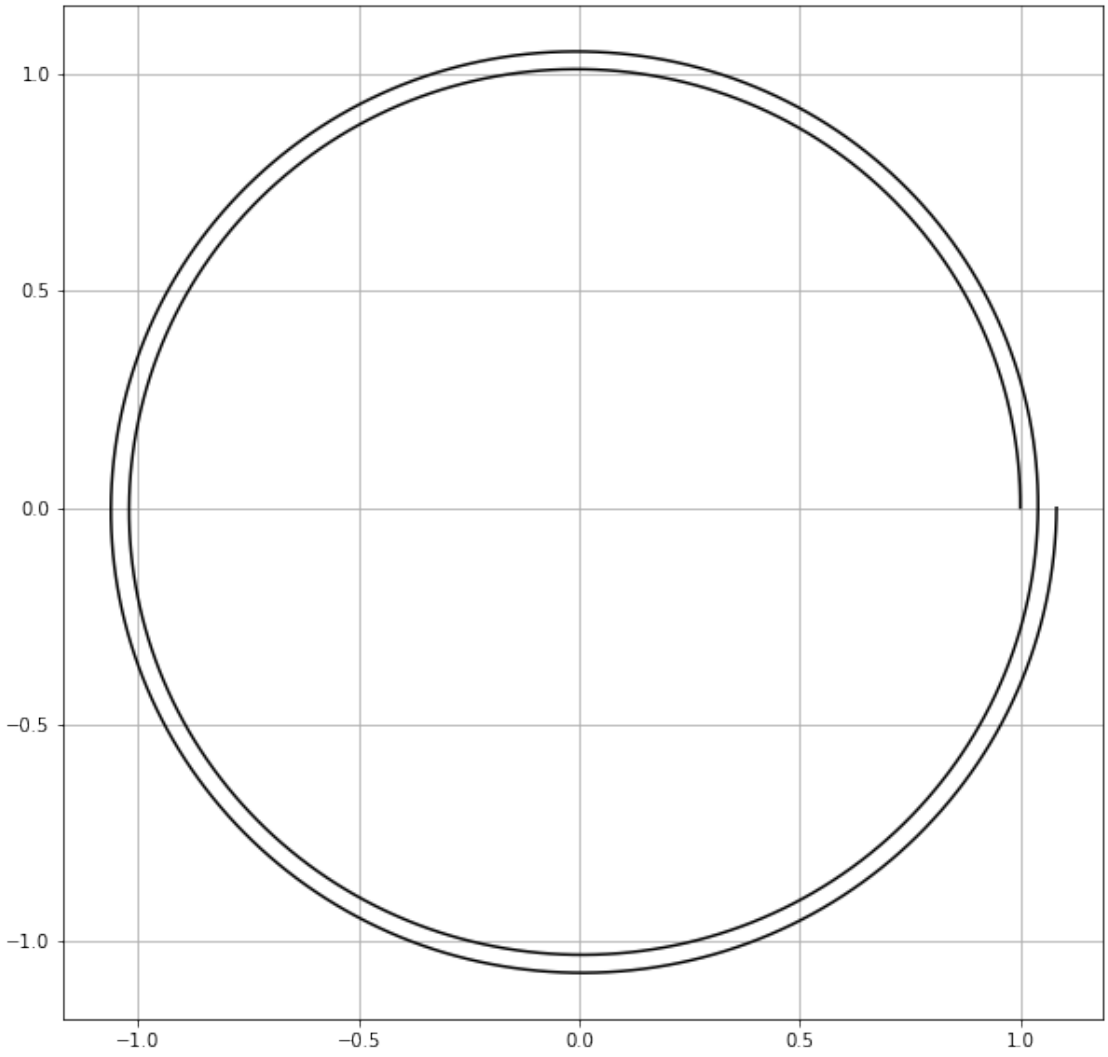
$$(u', v') = (-v, u)$$

où u et v sont deux fonctions inconnues de la variable t. Prenons les conditions initiales $u(0) = 1$, $v(0) = 0$. Que valent u et v ? Clairement,

$$u(t) = \cos t,\ v(t) = \sin t$$

est une solution, et on peut montrer que c'est la seule. Si l'on trace la **courbe paramétrée** $t \mapsto (u(t), v(t))$ dans le plan, on obtient … le cercle unité. Enfin si tout va bien.

[18]:
```
def G(t, X): return [-X[1], X[0]]
ts, Xs = eulerV(G, 0, 4 * math.pi, [1, 0], 1000)
us = [X[0] for X in Xs]
vs = [X[1] for X in Xs]
plt.plot(us, vs, 'k')
plt.grid()
plt.show()
```



On voit ici que l'erreur en $O(h)$ est fatale. La périodicité des solutions fait éclater l'erreur au grand jour, notre cercle ne se referme pas. La méthode d'Euler est trop approximative pour permettre d'obtenir des valeurs approchées fiables des solutions d'une équation différentielle.

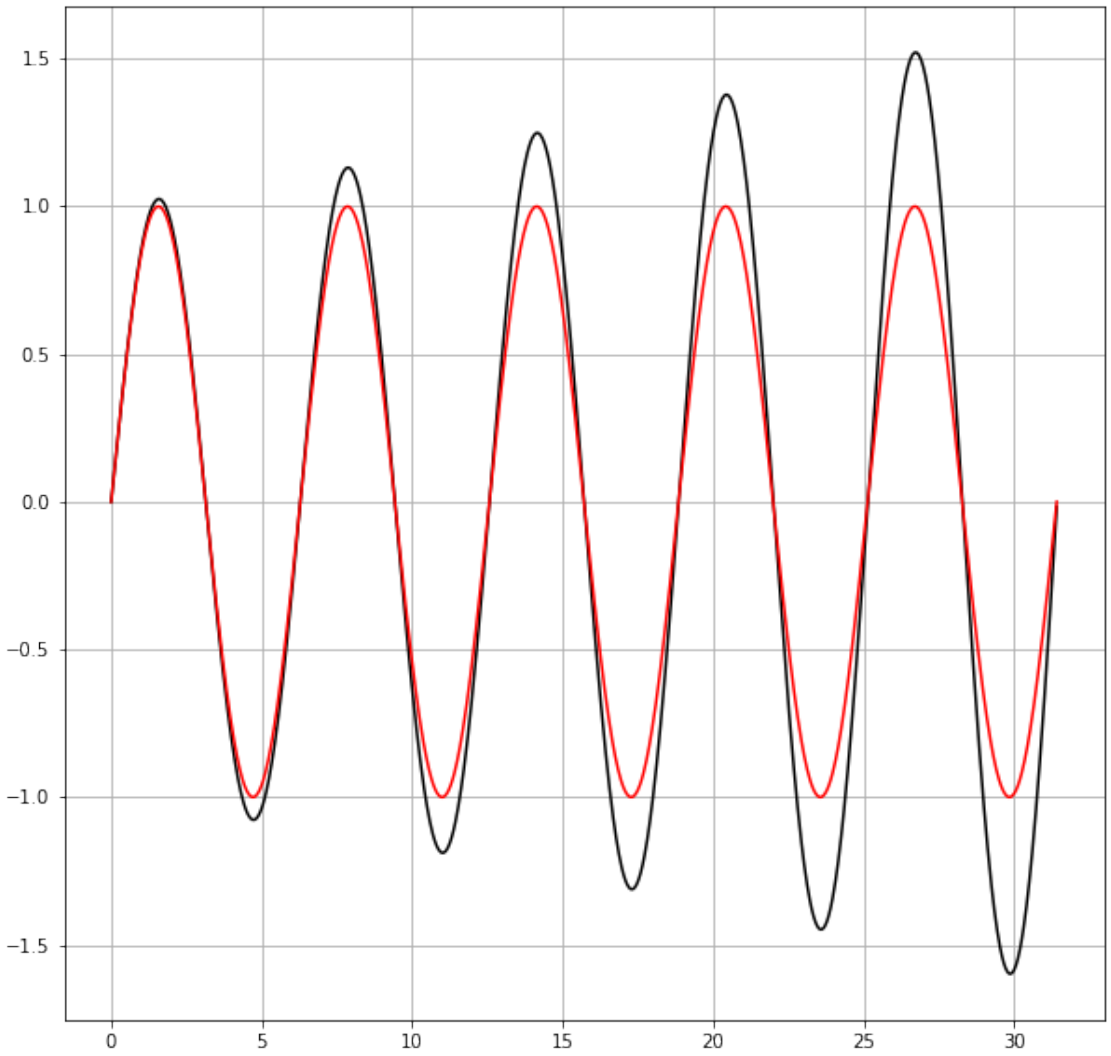
1.3.4 2.4 Équation différentielle d'ordre 2

On a compris, Euler pas bien. Mais enfonçons le clou en résolvant l'équation $y'' + y = 0$.

[19]:
```
def euler2(F, a, b, x0, x1, n):
    return dsolve2(F, a, b, x0, x1, n, eulerV_method)
```

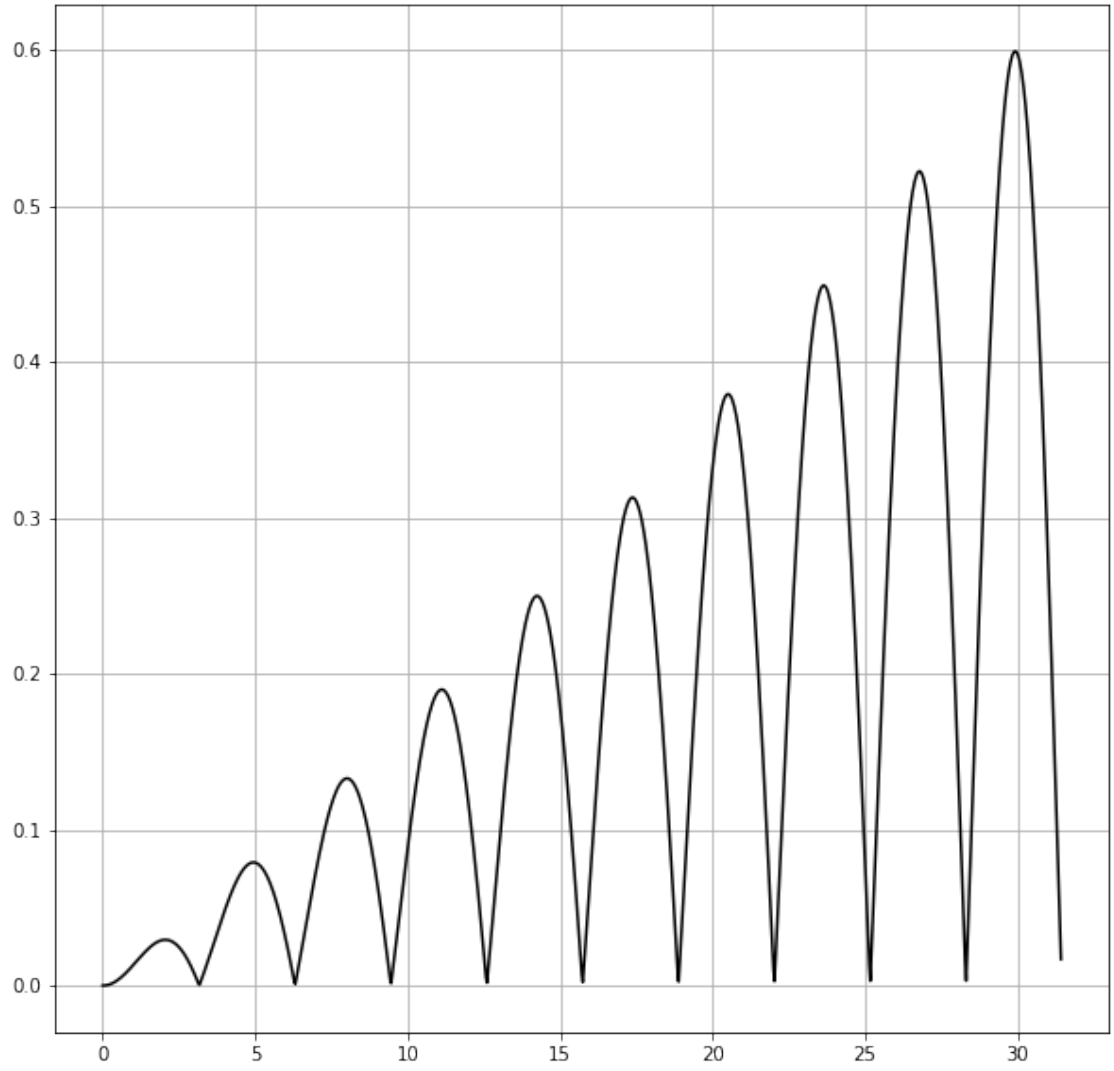
[20]:
```
ts, xs = euler2(lambda t, x, x1: -x, 0, 10 * math.pi, 0, 1, 1000)
plt.plot(ts, xs, 'k')
plt.plot(ts, [math.sin(t) for t in ts], 'r')
```

```
plt.grid()
plt.show()
```



En rouge, la vraie solution, qui est la fonction sinus. En noir, la solution approchée. Dessinons le graphe de l'erreur commise.

[21]:
```
err = [abs(xs[k] - math.sin(ts[k])) for k in range(len(ts))]
plt.plot(ts, err, 'k')
plt.grid()
plt.show()
```



La méthode d'Euler permet d'obtenir rapidement des valeurs approchées de solutions d'équations différentielles, mais ces valeurs approchées sont de qualité plus que médiocre.

Peut-on faire mieux ? Oui, et quitte à faire mieux, allons directement à **ce qui se fait de mieux**.

1.4 3. « La » méthode de Runge-Kutta

1.4.1 3.1 Équation d'ordre 1

Il existe en fait **des** méthodes de Runge-Kutta. Nous allons nous intéresser à la méthode dite d'ordre 4. Considérant toujours notre équation différentielle $x' = F(t, x)$, comment passer d'une valeur approchée de $x(t)$ à une valeur approchée de $x(t + h)$?

On calcule tout d'abord successivement les 4 nombres :

$$\left\{\begin{array}{lcl} k_0 &=& hF(t,x) \\ k_1 &=& hF(t+\frac{h}{2}, x+\frac{k_0}{2}) \\ k_2 &=& hF(t+\frac{h}{2}, x+\frac{k_1}{2}) \\ k_3 &=& hF(t+h, x+k_2) \end{array}\right.$$

On pose ensuite

$$\delta x = \frac{1}{6}(k_0 + 2k_1 + 2k_2 + k_3)$$

La méthode que nous examinons ici consiste à prendre

$$x(t+h) \simeq x + \delta x$$

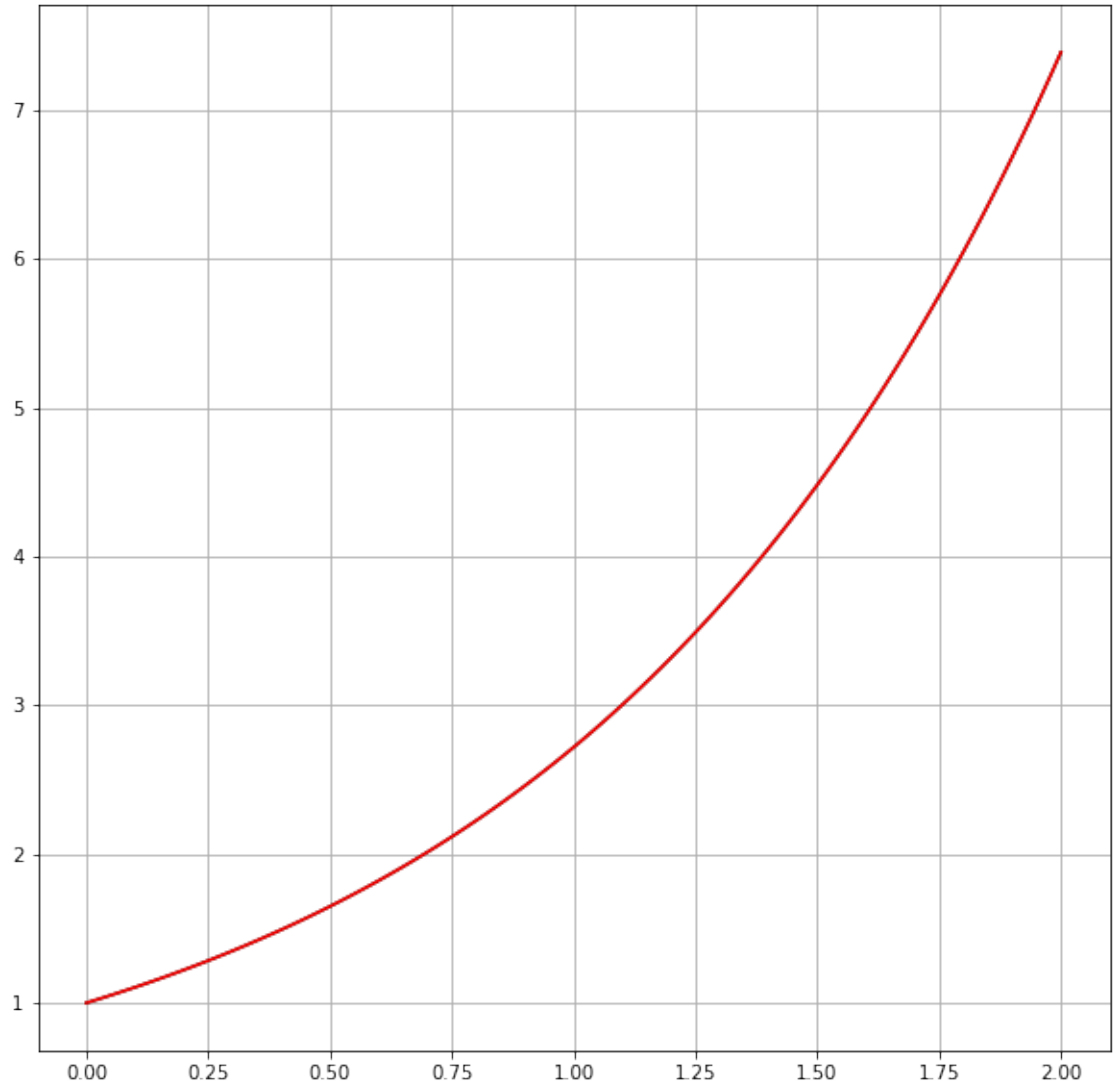
Je ne donnerai ici aucune justification concernant l'excellence de cette méthode.

[22]:
```
def rk4_method(F, t, x, h):
    k0 = h * F(t, x)
    k1 = h * F(t + 0.5 * h, x + 0.5 * k0)
    k2 = h * F(t + 0.5 * h, x + 0.5 * k1)
    k3 = h * F(t + h, x + k2)
    dx = (1 / 6) * (k0 + 2 * k1 + 2 * k2 + k3)
    return x + dx
```

[23]:
```
def rk4(F, a, b, x0, n):
    return dsolve(F, a, b, x0, n, rk4_method)
```

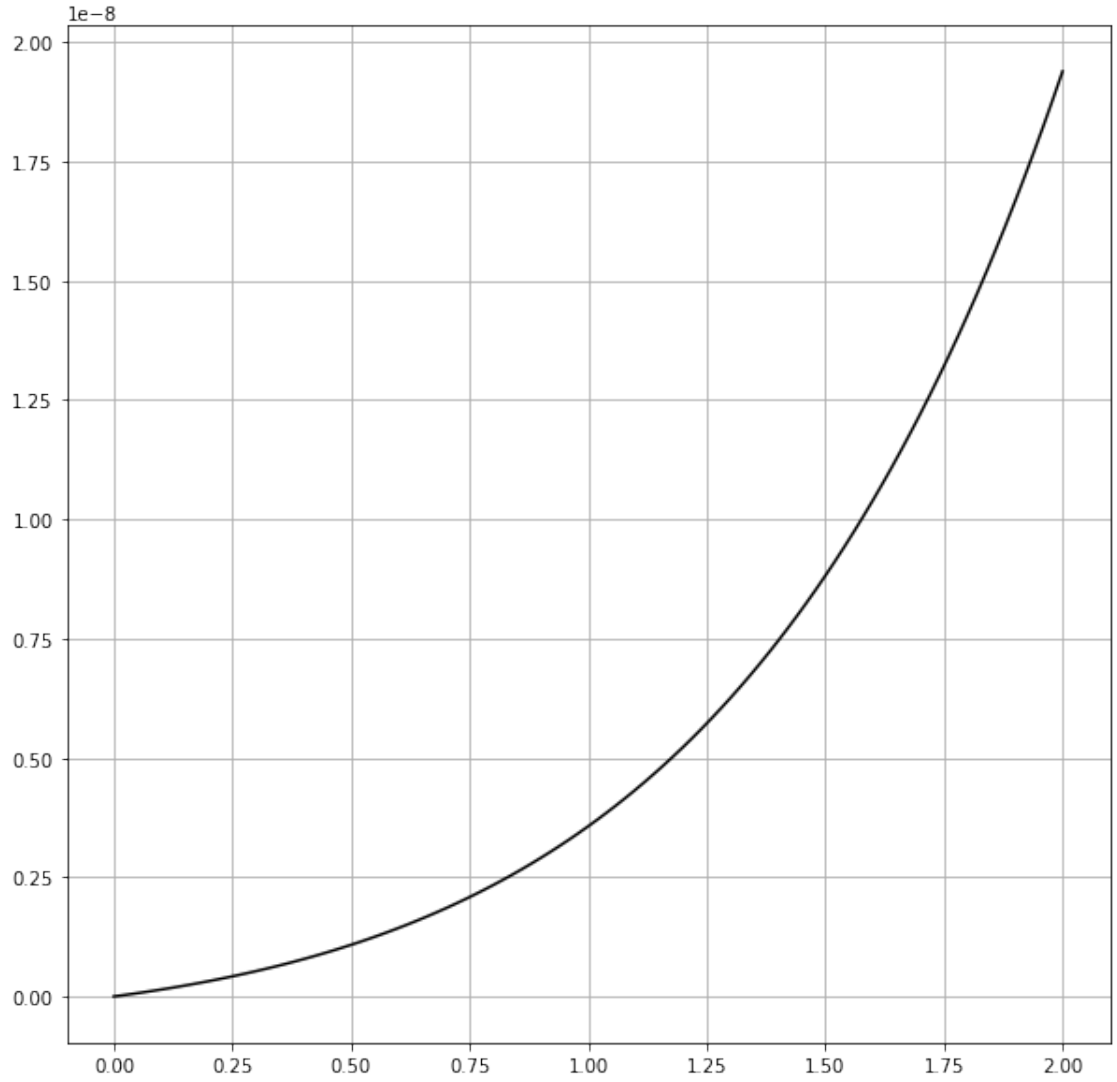
Reprenons l'exemple de l'exponentielle.

[24]:
```
ts, xs = rk4(lambda t, x:x, 0, 2, 1, 100)
plt.plot(ts, xs, 'k')
plt.plot(ts, [math.exp(t) for t in ts], 'r')
plt.grid()
plt.show()
```



Tiens, on ne voit qu'une courbe ? Où est la courbe noire de la solution approchée ? Eh bien elle est **sous** la courbe rouge, bien cachée … tiendrait-on une bonne méthode ?

[25]:
```
err = [abs(xs[k] - math.exp(ts[k])) for k in range(len(ts))]
plt.plot(ts, err, 'k')
plt.grid()
plt.show()
```



Remarquez le 10^{-8} en haut à gauche du graphique : l'erreur maximale commise n'est pas 2, mais 2×10^{-8}. L'approximation obtenue est excellente.

1.4.2 3.2 Équation vectorielle

Poursuivons nos tests. On réécrit la méthode pour des fonctions à valeurs vectorielles. Ce n'est qu'une réécriture avec des `add` et des `mul` de la fonction `rk4_method`.

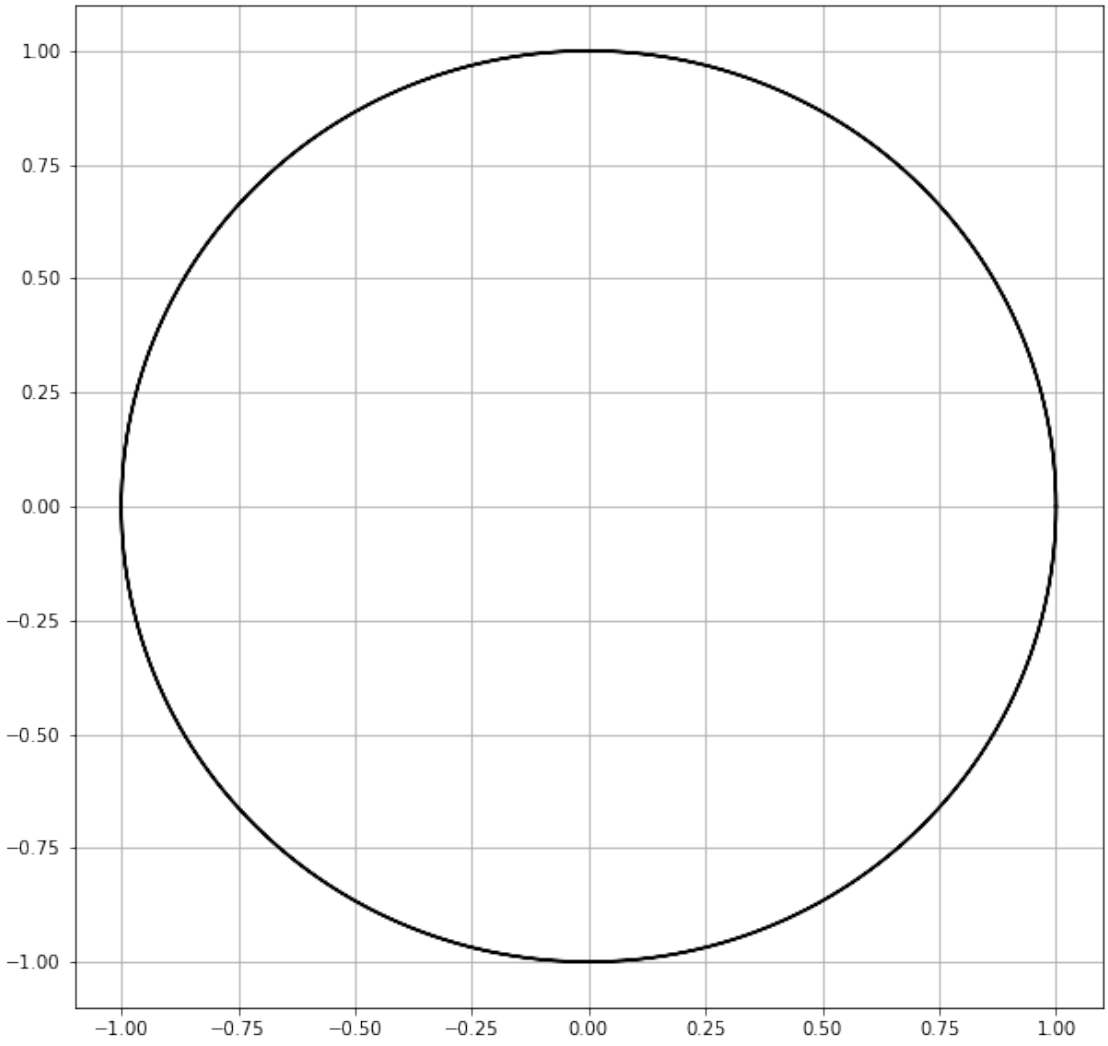
[26]:
```
def rk4V_method(G, t, X, h):
    K0 = mul(G(t, X), h)
    K1 = mul(G(t + 0.5 * h, add(X, mul(K0, 0.5))), h)
    K2 = mul(G(t + 0.5 * h, add(X, mul(K1, 0.5))), h)
    K3 = mul(G(t + h, add(X, K2)), h)
```

```
    dX = mul(add(add(K0, K3), mul(add(K1, K2), 2)), 1/6)
    return add(X, dX)
```

[27]:
```
def rk4V(G, a, b, X0, n):
    return dsolve(G, a, b, X0, n, rk4V_method)
```

Retentons le tracé du cercle.

[28]:
```
def G(t, X): return [-X[1], X[0]]
ts, Xs = rk4V(G, 0, 4 * math.pi, [1, 0], 1000)
us = [X[0] for X in Xs]
vs = [X[1] for X in Xs]
plt.plot(us, vs, 'k')
plt.grid()
plt.show()
```



Cette fois-ci rien ne dépasse. Le cercle se referme.

1.4.3 3.3 Équation d'ordre 2

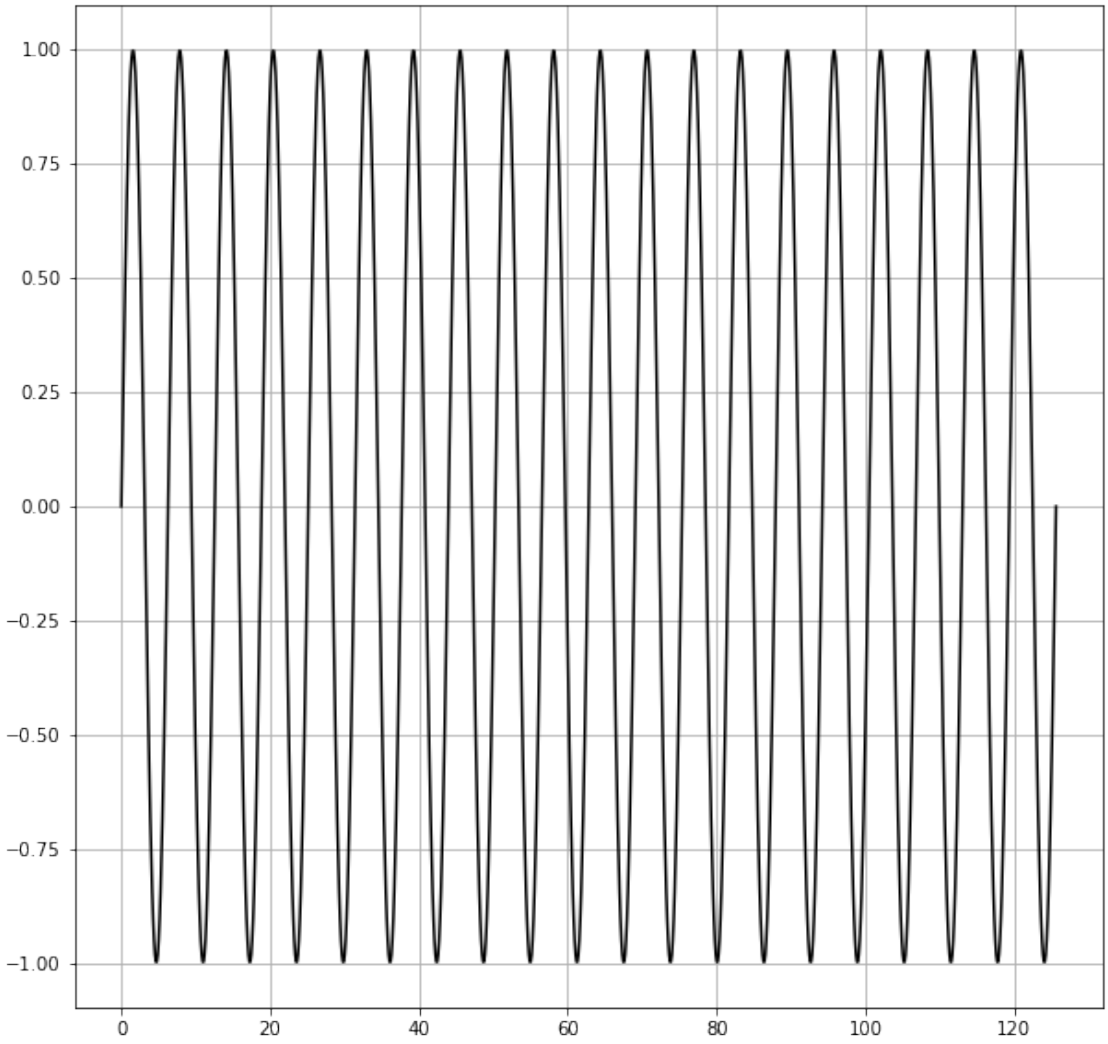
Rien à dire, copier-coller de ce qui a été fait pour la méthode d'Euler.

[29]:
```
def rk42(F, a, b, x0, x1, n):
    return dsolve2(F, a, b, x0, x1, n, rk4V_method)
```

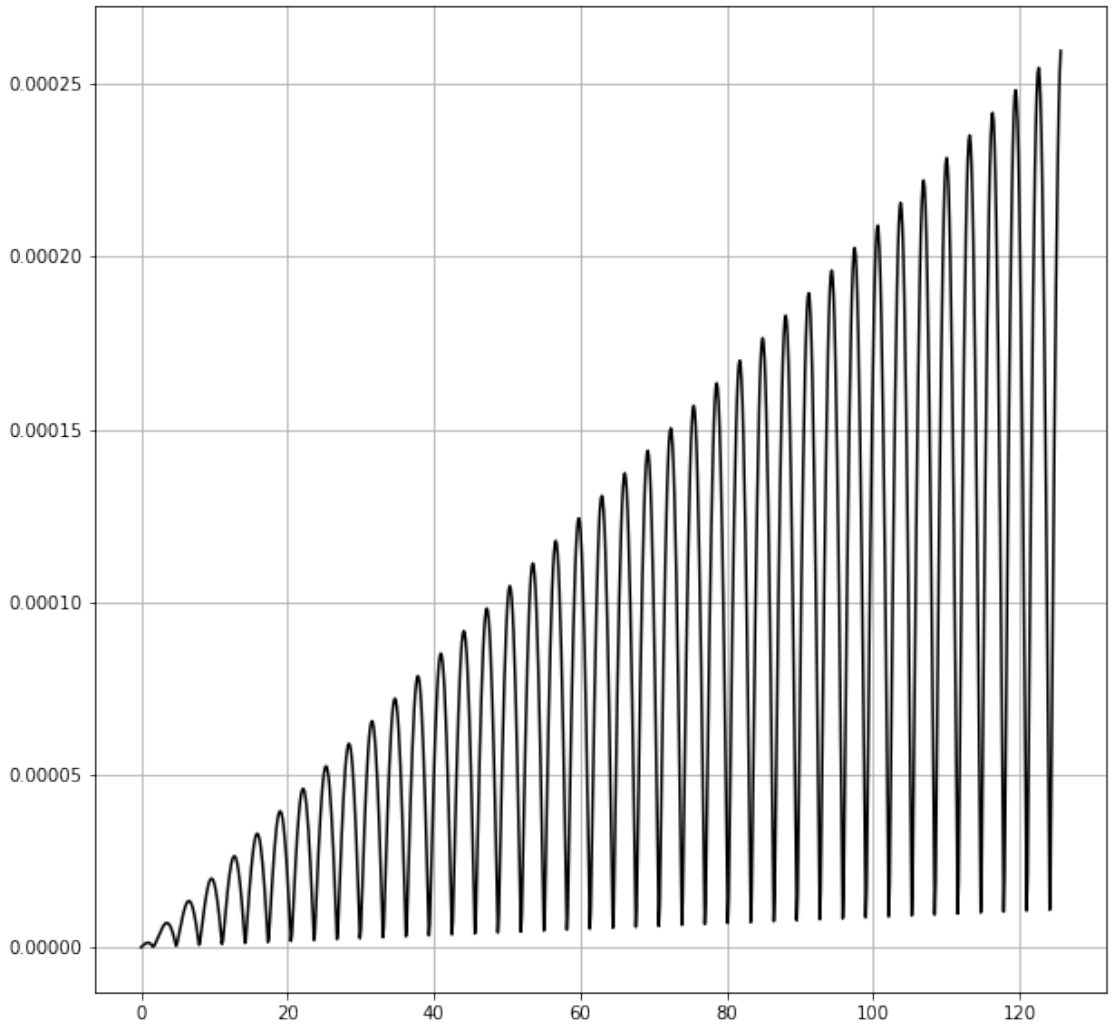
Pour bien voir l'efficacité de la méthode, on intègre l'équation différentielle sur 20 périodes du sinus, avec 1000 pas.

[30]:
```
ts, xs = rk42(lambda t, x, x1: -x, 0, 40 * math.pi, 0, 1, 1000)
```

[31]:
```
plt.plot(ts, xs, 'k')
plt.grid()
plt.show()
```



[32]:
```
err = [abs(xs[k] - math.sin(ts[k])) for k in range(len(ts))]
plt.plot(ts, err, 'k')
plt.grid()
plt.show()
```



L'erreur maximale est donc environ 2×10^{-4}. Bien qu'il existe des méthodes encore meilleures, la méthode de Runge-Kutta d'ordre 4 est une méthode à envisager sérieusement lorsqu'on a à faire des simulations numériques.

[ ]: