Chapitre $\underline{1010}_2$
Équations différentielles

17 octobre 2016

1 Cadre : le problème de Cauchy

Nous nous intéresserons au probème suivant, dit *de Cauchy* :

$$y' = F(y, t), \tag{1}$$

avec la condition initiale $y(t_0) = y_0$,

où :

— y est l'inconnue ;

— y est une fonction dérivable définie sur un intervalle I de $\mathbb{R}$;

— y et F sont à valeurs dans $E = \mathbb{R}^n$ ou $E = \mathbb{C}^n$ ($n \in \mathbb{N}^*$) ;

— F est définie sur $E \times I$;

— $t_0 \in I$ et $y_0 \in E$.

Remarque 1.0.1. En mathématiques et en physique, l'habitude est plutôt de considérer les équations différentielles sous la forme $y' = F(t, y)$ et non $y' = F(y, t)$.
Mais en `Python`, la fonction `odeint`, que nous verrons en fin de chapitre, utilise l'écriture $y' = F(y, t)$. Par souci de simplicité, nous nous conformerons à cette écriture dans ce cours.

2 Notion de solution

On appelle *solution* du problème précédent tout couple (y, J) où :

— J est un sous-intervalle de I contenant t_0

— $y : J \to E$ est une fonction dérivable vérifiant $y(t_0) = y_0$ et

$$\forall t \in J \quad y'(t) = F(y(t), t).$$

On dit qu'une telle solution est *maximale* s'il est impossible de prolonger y en une solution sur un intervalle strictement plus grand que J.

3 Premier exemple : équation linéaire

Une solution de l'équation (1) avec la condition initiale $y(1) = 2\mathrm{e}$ où

$$\begin{array}{rccc} F: & \mathbb{R}\times\mathbb{R} & \to & \mathbb{R} \\ & (y,t) & \mapsto & y \end{array}$$

est

$$\begin{array}{rcl} [0,1] & \to & \mathbb{R}, \\ t & \mapsto & 2\mathrm{e}^t, \end{array}$$

mais cette solution n'est pas maximale car elle est prolongeable (par exemple) par

$$\begin{array}{rcl} [0,17] & \to & \mathbb{R}, \\ t & \mapsto & 2\mathrm{e}^t. \end{array}$$

La seule solution maximale est dans ce cas :

$$\begin{array}{rcl} \mathbb{R} & \to & \mathbb{R}, \\ t & \mapsto & 2\mathrm{e}^t. \end{array}$$

4 Second exemple : équation non linéaire

On cherche à résoudre sur $\mathbb{R}$ l'équation

$$y' = \frac{3}{7}y^3$$

avec la condition initiale $y(0) = \frac{1}{6}$:

- il n'y a pas de solution définie sur $\mathbb{R}$ tout entier (même en changeant la condition initiale, sauf pour une condition initiale $y(t_0) = 0$) ;
- il y a une unique solution maximale : $t \mapsto \sqrt{\frac{7}{252-6t}}$;
- cette solution maximale est définie sur $]-\infty, 42[$.

5 Théorème de Cauchy-Lipschitz

Sous des hypothèses raisonnables sur F : pour chaque problème de Cauchy, il existe une unique solution maximale.

Attention : l'intervalle sur lequel une solution maximale est définie peut dépendre de la condition initiale.

6 En pratique

Bien souvent, on considère en pratique le cas $I = [a, b]$ (où $a, b \in \mathbb{R}$ avec $a < b$) et les solutions seront définies sur I tout entier.

On se placera dans ce cadre pour la suite de ce cours. On supposera que la condition initiale est donnée en a et on notera y_0 la valeur initiale de y en a. On note y l'unique solution maximale du problème de Cauchy.

7 La méthode d'Euler

Nous avons vu en mathématiques que la résolution d'une équation différentielle linéaire du premier ordre se ramenait au calcul d'une primitive. Mais il existe des fonctions pour lesquelles nous ne savons pas calculer de primitive. Nous sommes donc incapables de résoudre ces équations de manière exacte. Quand il s'agit d'équations différentielles d'ordre supérieur ou non linéaires, la situation est encore moins favorable. Que faire s'il nous faut vraiment une solution à une telle équation différentielle ? Une possibilité est d'essayer d'obtenir une *approximation numérique* d'une solution exacte. La méthode d'Euler est la méthode classique la plus simple pour faire cela. Le principe assez élémentaire de cette méthode est le suivant :

— On choisit un *pas* $h = \frac{b-a}{n}$, où $n \in \mathbb{N}^*$.

— On subdivise l'intervalle $[a, b]$ en n morceaux, en posant $t_k = a + kh$ pour chaque $k \in [\![0, n]\!]$ ($t_0 = a$, $t_n = b$).

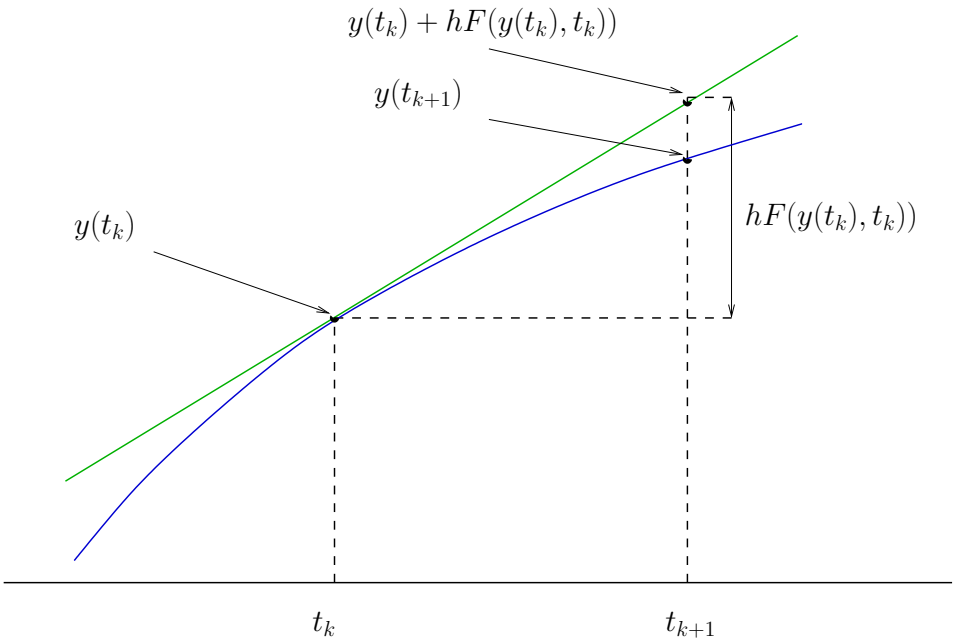
— On sait (par le théorème des accroissements finis) que pour chaque $k \in [\![0, n[\![$, il existe $c_k \in [t_k, t_{k+1}]$ tel que

$$\begin{aligned} y(t_{k+1}) - y(t_k) &= (t_{k+1} - t_k)y'(c_k) \\ &= hF(y(c_k), c_k) \\ &\approx hF(y(t_k), t_k). \end{aligned}$$

Ceci conduit à poser pour $k \in [\![0, n[\![$

$$y_{k+1} = y_k + hF(y_k, t_k)$$

On espère alors que l'on aura $y_k \approx y(t_k)$ pour tout $k \in [\![1, n]\!]$.



Par récurrence, on construit ainsi, de proche en proche, les réels y_0, y_1, y_2, $\cdots$, y_n qui sont des approximations des $y(t_k)$. Chaque approximation y_{k+1} est construite à partir de l'approximation précédente, y_k.
La ligne brisée joignant les points (t_k, y_k) est le graphe d'une fonction, qui est elle-même une approximation de la solution exacte.

8 Premier exemple, une équation linéaire

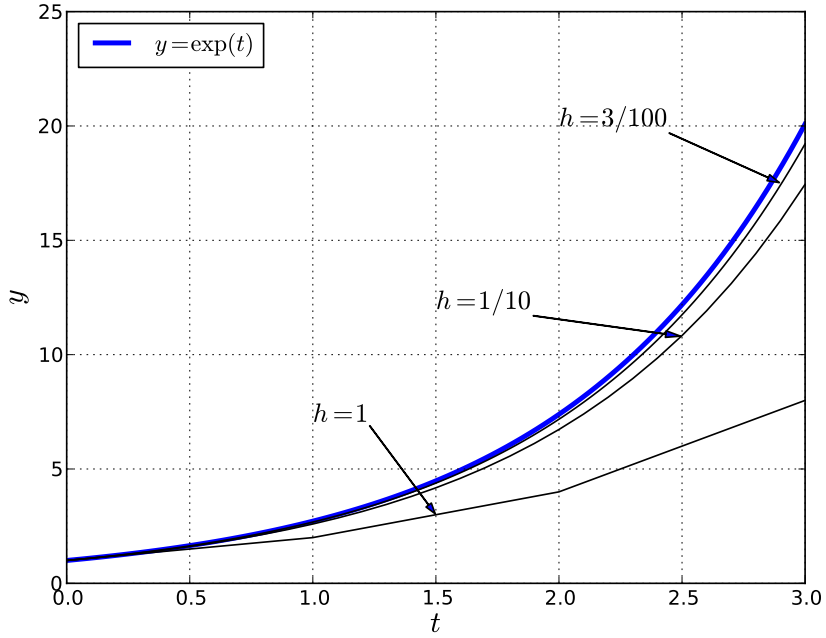
On cherche à approcher la solution (maximale) sur $[0, 3]$ de

$$y' = y$$

avec condition initiale $y(0) = 1$.

Remarque 8.0.2. Vous savez bien entendu que c'est $t \mapsto \mathrm{e}^t$.

Numériquement, on regarde successivement les approximations obtenues par la méthode d'Euler avec les paramètres $n = 3$, $n = 30$ et $n = 100$, soit des pas respectivement de 1, $1/10$ et de $3/100$.



9 Quelques remarques

On peut conjecturer que

1. quand le pas diminue, l'approximation s'améliore ;
2. la méthode d'Euler ne corrige pas les erreurs d'approximation mais au contraire les propage en les augmentant.

En fait :

— Le premier point est vrai mathématiquement.

— Le second dépend des équations considérées.

Informatiquement :

— Si le pas est grand, l'approximation est mauvaise ;

— Si le pas est petit, elle est longue à calculer ;

— S'il est vraiment trop petit, les erreurs d'arrondis causent en plus d'autres problèmes.

10 Mise en œuvre

```
def euler(F, a, b, y0, h):
    """Solution de y'=F(y,t) sur [a,b], y(a) = y0, pas h"""
    y = y0
```

```
    t = a
    les_y = [y0] # la liste des valeurs renvoyées
    les_t = [a]
    while t+h <= b:
        # Variant : floor((b-t)/h)
        # Invariant : au tour k, les_y = [y_0,...,y_k], les_t = [t_0,...,t_k]
        y += h * F(y, t)
        les_y.append(y)
        t += h
        les_t.append(t)
    return les_t, les_y
```

Remarque 10.0.3. L'instruction `y += x` incrémente `y` de `x`. Naïvement, elle devrait correspondre à `y = y+x`, mais nous verrons bientôt que ce n'est pas le cas.

Remarque 10.0.4. La variante écrite au dessus correspond au cas où le pas h nous est donné. Si le nombre de subdivisions n nous est donné à la place, on pourra préférer écrire la fonction avec une boucle `for`, en utilisant la fonction `linspace` du module `numpy`.

11 Second exemple : une autre équation linéaire

Problème de cinétique chimique : on s'intéresse à une réaction chimique faisant disparaître un réactif A d'une solution.

On suppose que la vitesse de disparition de A est proportionnelle à sa concentration (réaction d'ordre 1). La concentration de A suit donc l'équation

$$\frac{d[A]}{dt} = -\alpha[A],$$

où α est une constante s'exprimant en s^{-1} (la valeur $\frac{\ln 2}{\alpha}$ est un temps appelé temps de demi-réaction).

L'équation est bien de la forme

$$[A]' = F([A], t) \qquad \text{où } F : (y, t) \mapsto -\alpha y.$$

On suppose $\alpha = 1\text{s}^{-1}$ et, au temps $t = 0$, $[A] = 1\text{mol/l}$. On veut regarder l'évolution jusqu'à $t = 6\text{s}$.

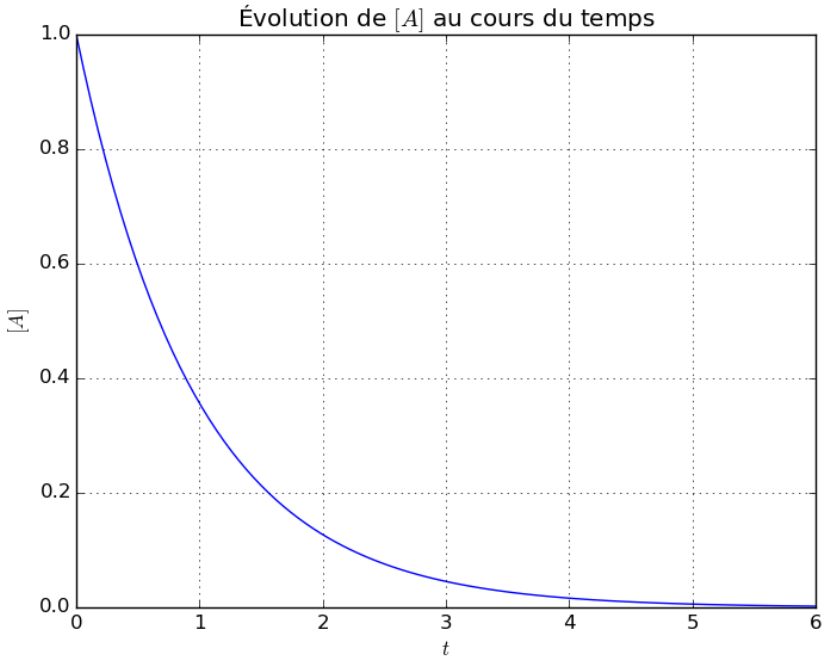
```
alpha = 1 # s ** -1
A0 = 1 # mol/l
t0, t1 = 0, 6 # s
n = 100
h = float(t1 - t0) / n
def F(y, t):
    return -alpha * y
les_t, les_y = euler(F, t0, t1, A0, h)
```

Remarque 11.0.5. Les pythonistes pourront préférer définir `F` comme suit.

```
F = lambda y, t: -alpha * y
```

12 Représentation de graphes

```
import matplotlib.pyplot as pl

pl.plot(les_t, les_y)
pl.grid()
pl.title('Évolution de $[A]$ au cours du temps')
pl.xlabel('$t$')
pl.ylabel('$[A]$')
pl.savefig('graphe-A.png') # pl.show() pour afficher
pl.clf() # pour effacer le graphe
```



13 Exemple en dimension 3

Problème de cinétique chimique : avec trois réactifs, A, B et C. A se transforme en B qui se transforme en C (réactions d'ordre 1). Les concentrations de A, B et C suivent

donc les équations suivantes.

$$\frac{\mathrm{d}[A]}{\mathrm{d}t} = -\alpha[A]$$
$$\frac{\mathrm{d}[B]}{dt} = \alpha[A] - \beta[B]$$
$$\frac{\mathrm{d}[C]}{\mathrm{d}t} = \beta[B]$$

On supposera $\alpha = 1\mathrm{s}^{-1}$, $\beta = 10\mathrm{s}^{-1}$ et au temps $t = 0$, $[A] = 1\mathrm{mol/l}$, $[B] = [C] = 0$. On veut regarder l'évolution jusqu'à $t = 6\mathrm{s}$.

On applique le même principe que précédemment : la méthode d'Euler, **mais** avec une fonction F qui rendra des éléments de $\mathbb{R}^3$ et non plus de $\mathbb{R}$. En effet, on a

$$\frac{\mathrm{d}}{\mathrm{d}t}\begin{pmatrix}[A]\\[B]\\[C]\end{pmatrix} = \begin{pmatrix}-\alpha[A]\\ \alpha[A]-\beta[B]\\ \beta[B]\end{pmatrix} = F\begin{pmatrix}[A]\\[B]\\[C]\end{pmatrix},$$

avec

$$F : \begin{pmatrix}a\\b\\c\end{pmatrix} \longmapsto \begin{pmatrix}-\alpha a\\ \alpha a-\beta b\\ \beta b\end{pmatrix}.$$

Remarque 13.0.6. Problème technique : l'addition sur les triplets ne donne pas ce qu'on veut :

```
>>> (1, 2, 3) + (4, 5, 6)
(1, 2, 3, 4, 5, 6)
```

Heureusement : la bibliothèque de calcul numérique `numpy` fournit un type `array` de tableau sur lequel les opérations usuelles sont appliquées terme à terme :

```
>>> from numpy import array
>>> array([1, 2, 3]) + array([4, 5, 6])
array([5, 7, 9])
>>> array([1, 2, 3]) * array([4, 5, 6])
array([ 4, 10, 18])
```

```
from numpy import array
alpha, beta = 1, 10 # s ** -1
A0, B0, C0 = 1, 0, 0 # mol / l
t0, t1 = 0, 6 # s
n = 3
h = float(t1 - t0) / n
def F(y, t):
    A, B, C = y
    return array([-alpha*A, alpha*A-beta*B, beta*B])
les_t, les_y = \
    euler(F, t0, t1, array([A0, B0, C0]), h)
```

Essayons ceci. Après calcul on obtient :
les_t = [0, 2.0, 4.0, 6.0]
les_y = [array([-1, 762, -760]), array([-1, 762, -760]), array([-1, 762, -760]), array([-1, 762, -760])]
Petit problème : cela ne fonctionne pas. Après l'exécution, tous les éléments de `les_y` sont égaux. Que s'est-il passé ? ? ?
Revoyons le code de la fonction `euler` :

```
def euler(F, a, b, y0, h):
    """Solution de y'=F(y,t) sur [a,b], y(a) = y0, pas h"""
    y = y0
    t = a
    les_y = [y0] # la liste des valeurs renvoyées
    les_t = [a]
    while t+h <= b:
        # Variant : floor((b-t)/h)
        # Invariant : au tour k, les_y = [y_0,...,y_k], les_t = [t_0,...,t_k]
        y += h * F(y, t)
        les_y.append(y)
        t += h
        les_t.append(t)
    return les_t, les_y
```

Un point important : les tableaux sont modifiables (on dit parfois *mutables*), mais pas les entiers. Conséquence : les instructions

```
>>> x = array([1, 2, 3])
>>> y = x
>>> y += array([4, 5, 6])
>>> print(x,y)
[5 7 9] [5 7 9]
```

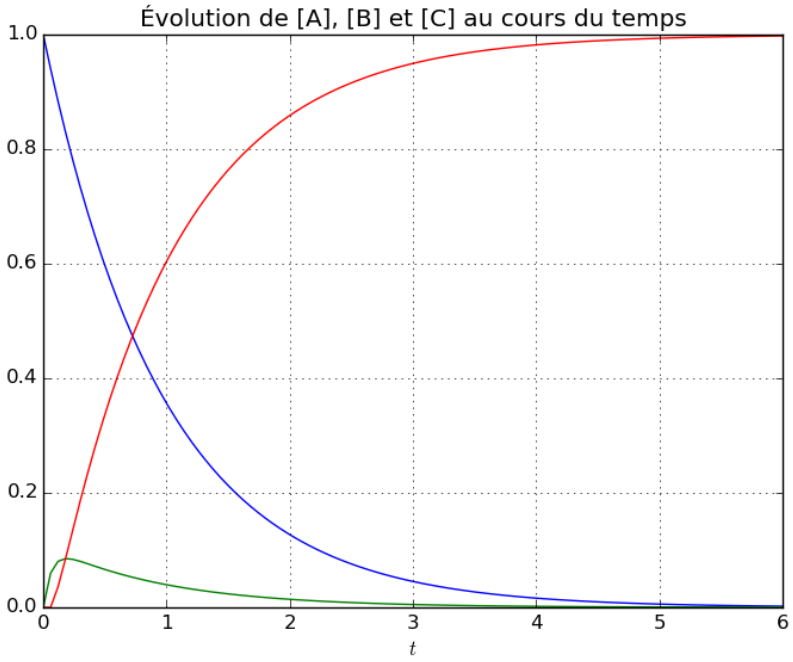
et les instructions

```
>>> x = array([1, 2, 3])
>>> y = x
>>> y = y + array([4, 5, 6])
>>> print(x,y)
[1 2 3] [5 7 9]
```

ne sont pas équivalentes. L'erreur se corrige rapidement :

```
def euler_vectoriel(F, a, b, y0, h):
    """Solution de y'=F(y,t) sur [a,b], y(a) = y0, pas h"""
    y = y0
```

```
    t = a
    les_y = [y0] # la liste des valeurs renvoyées
    les_t = [a]
    while t+h <= b:
        # Variant : floor((b-t)/h)
        # Invariant : au tour k, les_y = [y_0,...,y_k], les_t = [t_0,...,t_k]
        y = y + h * F(y, t) # surtout pas += !
        les_y.append(y)
        t += h
        les_t.append(t)
    return les_t, les_y
```



14 Utilisation de scipy

Nous ne sommes pas les seuls à vouloir résoudre numériquement des équations différentielles, donc il doit déjà exister des implantations pour ça.

Intérêt de réutiliser une implantation existante :

— gain de temps (pas besoin de la reprogrammer) ;

— bugs connus (trouvés par d'autres) ;

— problèmes de performance connus (remarqués par d'autres).

De plus on peut espérer :

— que les bugs ont été corrigés ;

— que les performances ont été optimisées.

Dans le cas du logiciel libre :

— c'est souvent le cas ;

— sinon, on peut réparer le logiciel soi-même (ou le faire réparer).

La bibliothèque `scipy` propose un grand nombre de méthodes de calcul numérique. En particulier, `odeint` :

— est une fonction de la bibliothèque `scipy.integrate` ;

— résout numériquement des EDO ;

— utilise une méthode plus raffinée que celle d'Euler ;

— choisit elle-même le pas à utiliser.

On l'utilise comme suite.

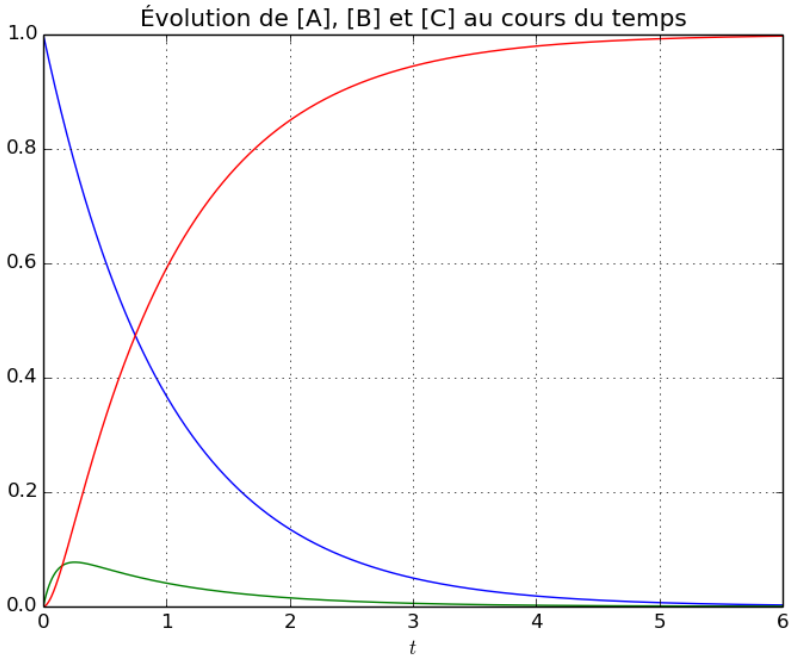
```
from scipy.integrate import odeint
les_y = odeint(F, y0, les_t)
```

Attention :

— On ne donne pas les extrémités de l'intervalle mais les points où on veut des valeurs (`y0` : condition initiale en `les_t[0]`)

Exemple :

```
from numpy import array, linspace
from scipy.integrate import odeint
alpha, beta = 1, 10 # s ** -1
A0, B0, C0 = 1, 0, 0 # mol / l
t0, t1 = 0, 6 # s
n = 1000
h = float(t1 - t0) / n

def F(y, t):
  A, B, C = y
  return array([-alpha*A, alpha*A - beta*B, beta*B])

les_t = linspace(0, 6, n + 1) # nb points

les_y = odeint(F, array([A0, B0, C0]), les_t)
```



15 Gérer des EDO d'ordre 2, 3, ...

Considérons l'équation d'un amortisseur soumis à une excitation de pulsation ω :

$$my'' = -cy' - ky + \alpha \cos \omega t$$

Ce n'est pas une équation d'ordre 1. Comment la gérer ?

On introduit

$$Y : t \longmapsto \begin{pmatrix} y(t) \\ y'(t) \end{pmatrix}.$$

on a alors

$$\begin{aligned} Y'(t) &= \begin{pmatrix} y'(t) \\ y''(t) \end{pmatrix} \\ &= \begin{pmatrix} y'(t) \\ (-cy'(t) - ky(t) + \alpha \cos \omega t)m^{-1} \end{pmatrix} \\ &= F(Y(t), t) \end{aligned}$$

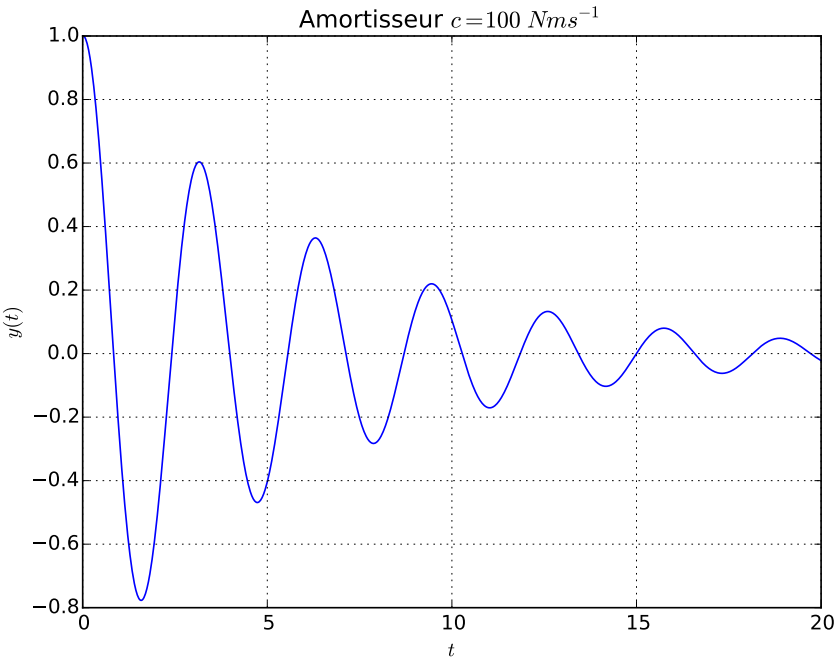
où

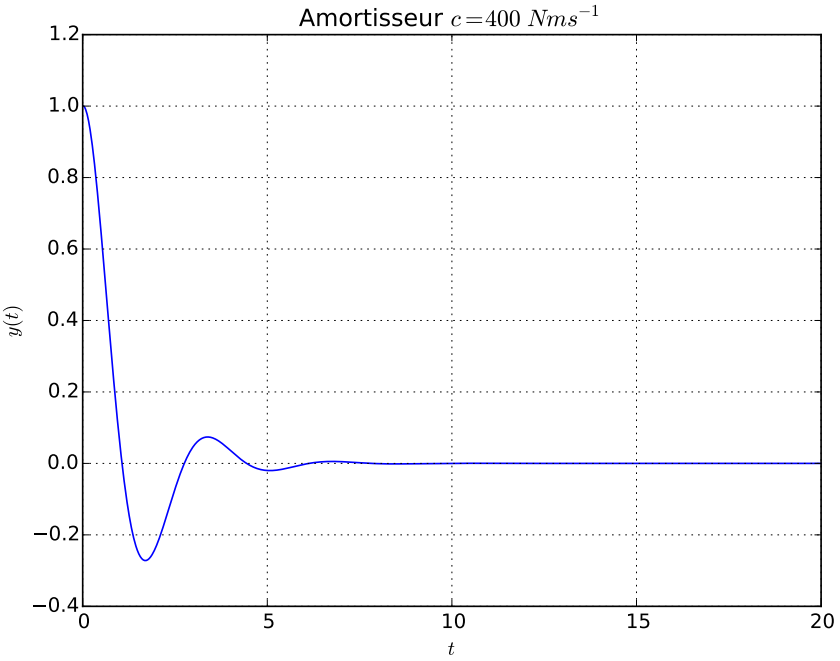
$$F \ : \ \left(\begin{pmatrix} a \\ b \end{pmatrix}, t \right) \longmapsto (b, (-cb - ka + \alpha \cos \omega t)m^{-1}).$$

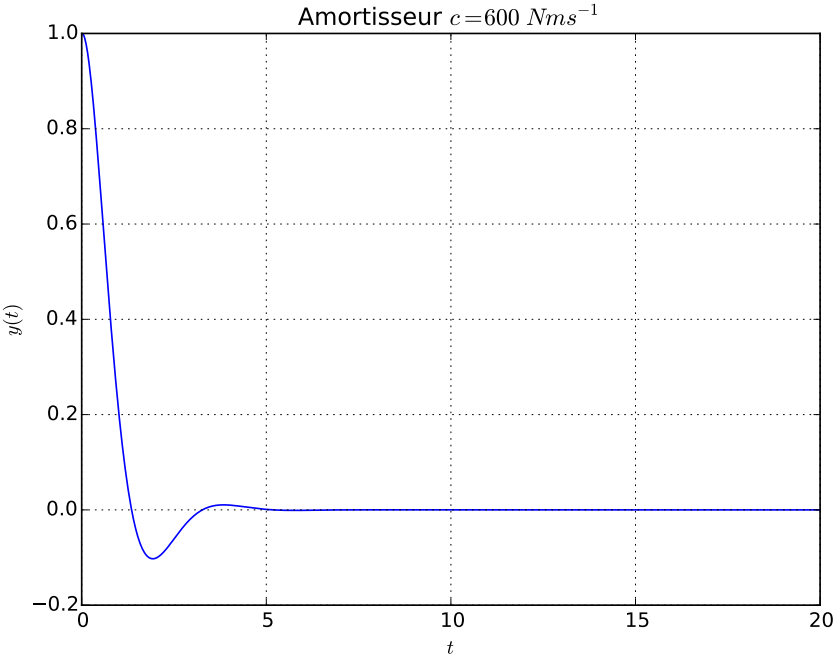
On s'est ramené à une équation d'ordre 1 !

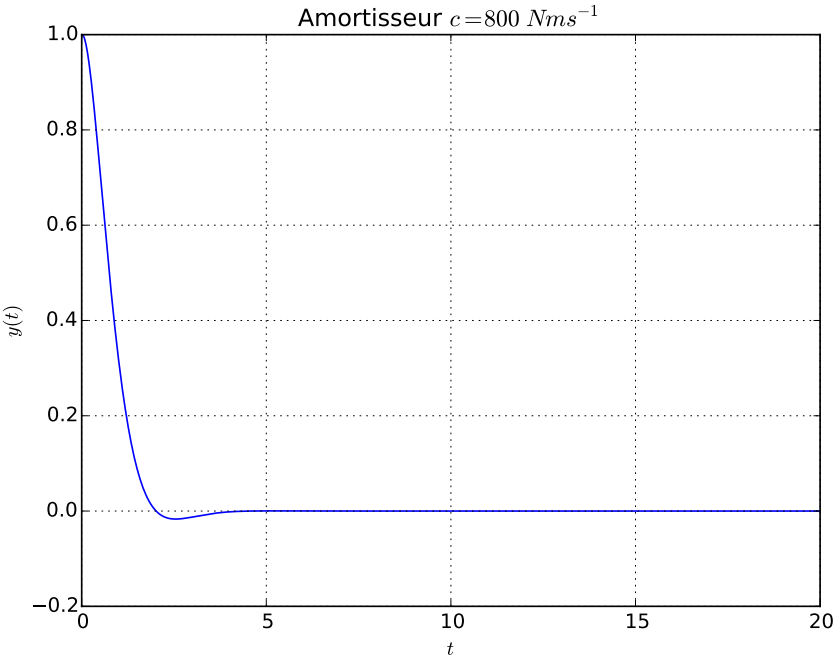
Résolution :

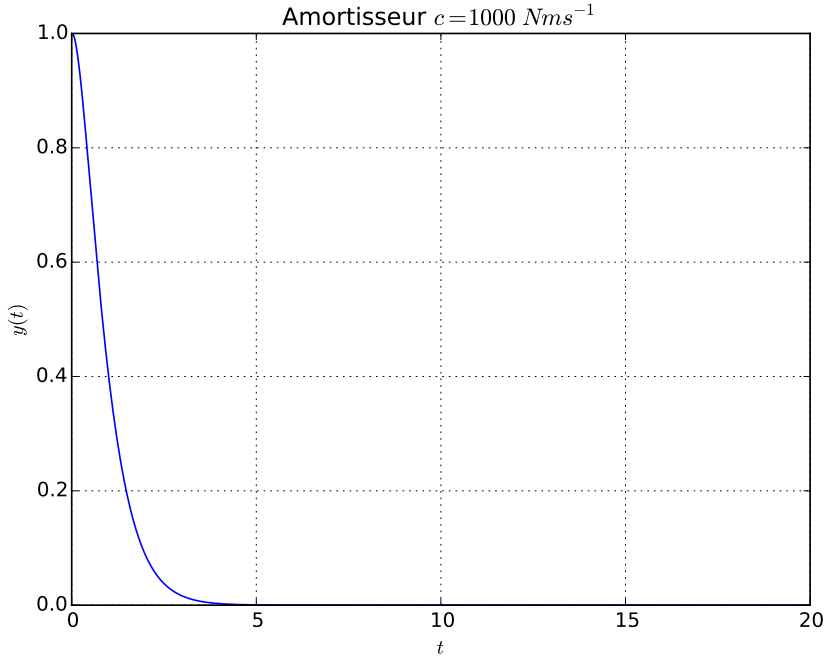
```
>>> m = 250 # kg
>>> omega = 1 # s ** (-1)
>>> alpha = 0 # N
>>> k = 1000 # N / m
>>> c = 500 # N * m * s**(-1)
>>> y0 = 1 # m
>>> yp0 = 0 # m . s**(-1)
>>> t0 = 0
>>> t1 = 20 # s
>>> n = 1000
>>> h = float(t1 - t0) / n

>>> def F(Y, t) :
...   y, yp = Y
...   ypp = (-c*yp - k*y + alpha*cos(omega*t)) / m
...   return array([yp, ypp])
... les_t, les_y = euler_vectoriel(F, t0, t1,
>>>   array([y0, yp0]), h)
```

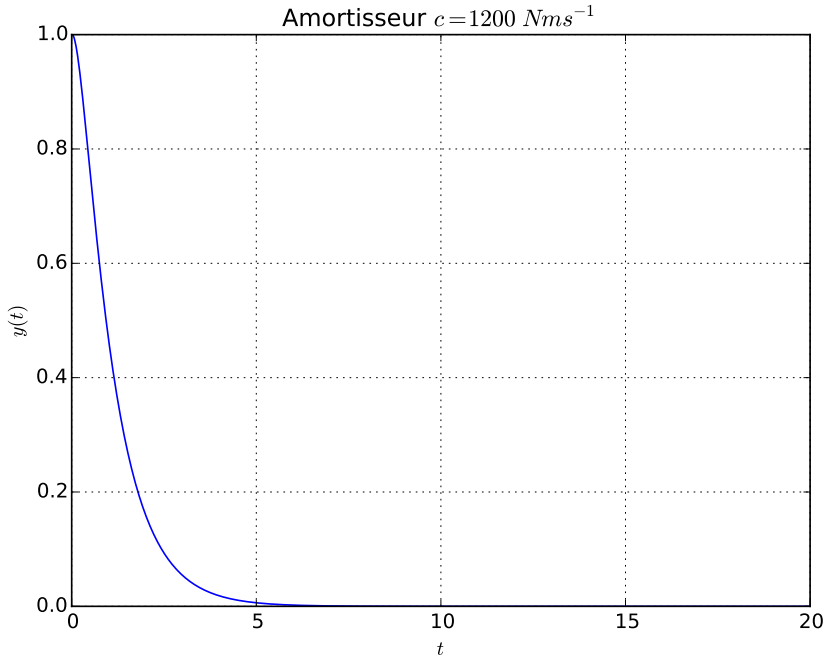


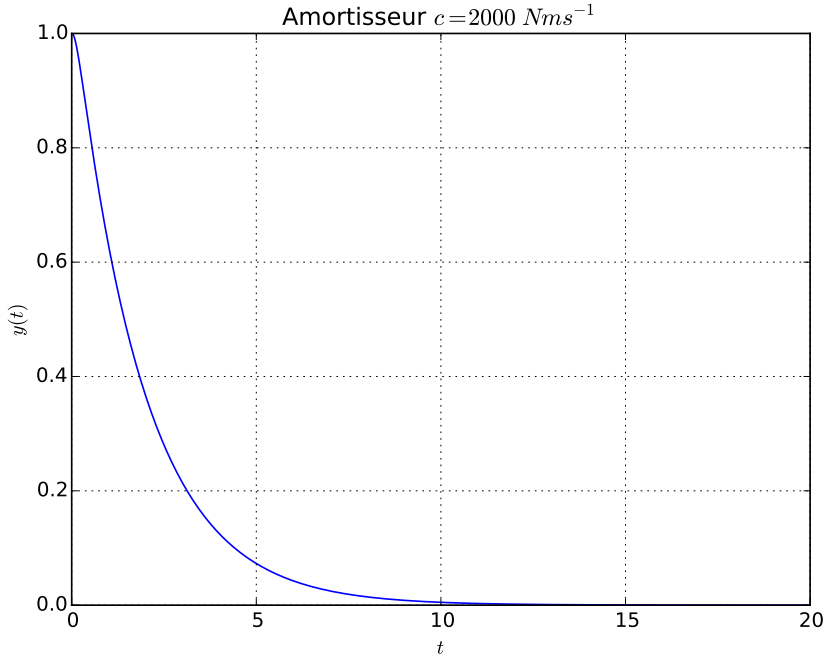

Amortisseur $c = 400\ Nms^{-1}$
1.2
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
$y(t)$
0
5
10
15
20
t


Amortisseur $c = 600\ Nms^{-1}$
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
$y(t)$
0
5
10
15
20
t


Amortisseur $c = 800\ Nms^{-1}$
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
$y(t)$
0
5
10
15
20
t


Amortisseur $c = 1000\ Nms^{-1}$
1.0
0.8
0.6
0.4
0.2
0.0
$y(t)$
0
5
10
15
20
t

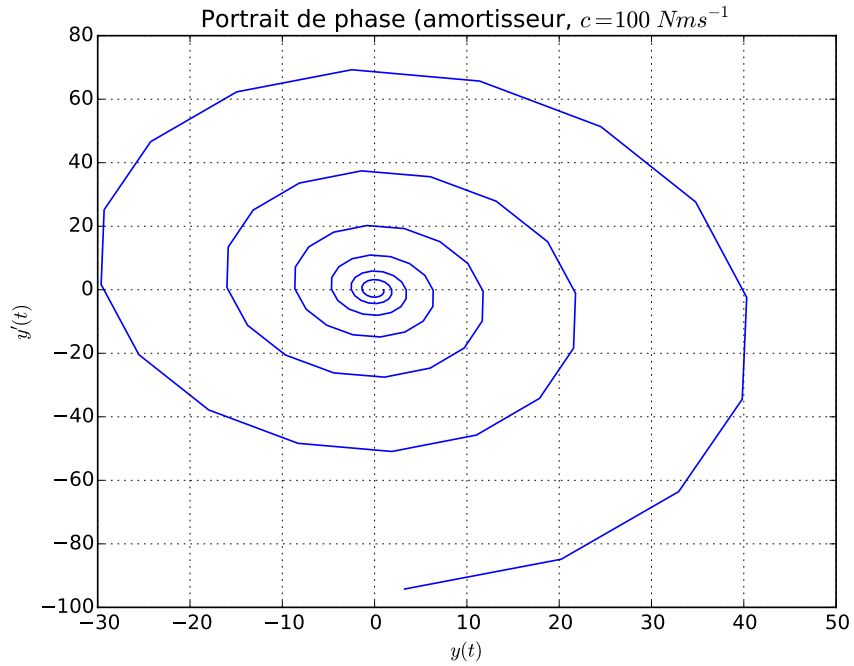




16 Portraits de phase

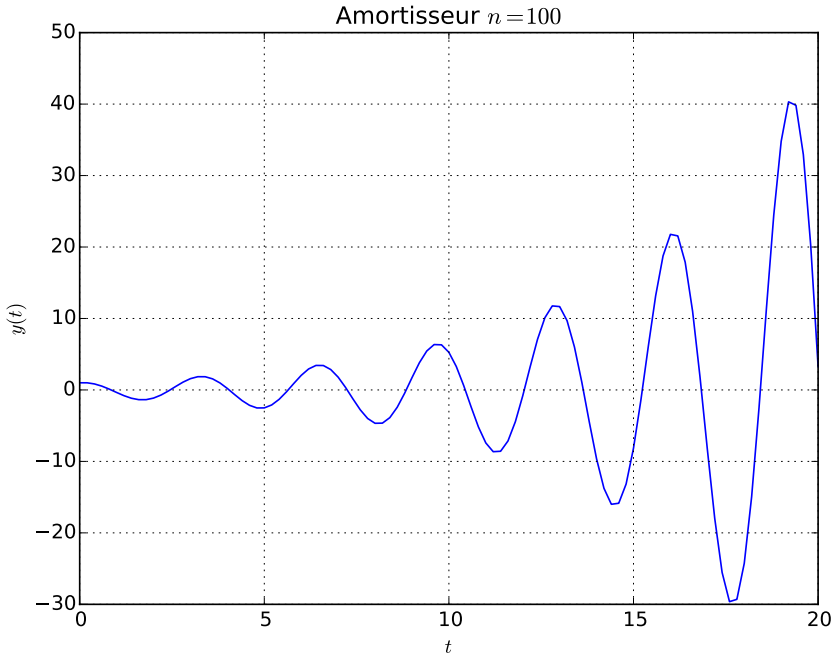
On porte sur le graphe les couples $(y(t), y'(t))$:

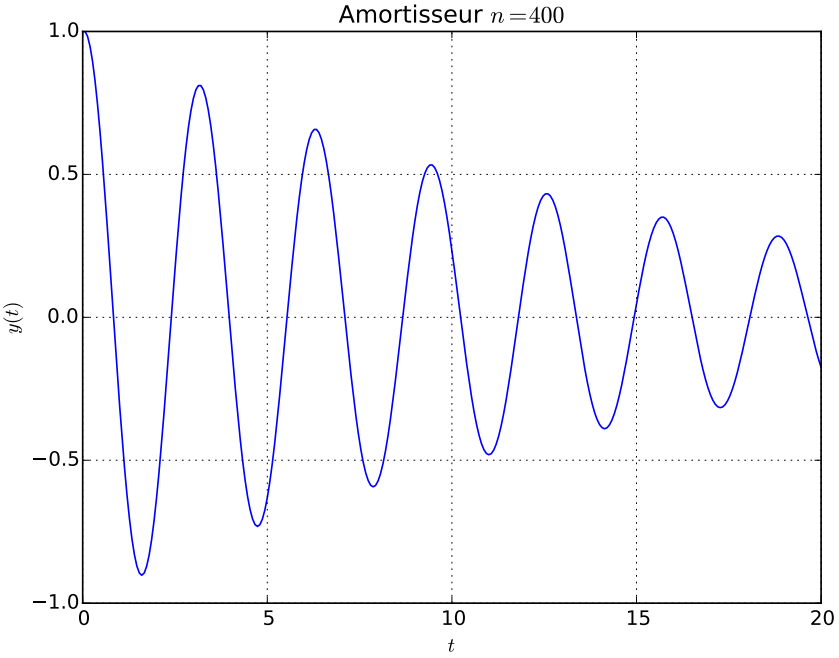
```
>>> def phase(legende, nomfichier) :
...     _, les_y = euler_vectoriel(F, t0, t1,
...         array([y0, yp0]), h)
...     pl.clf()
...     pl.xlabel('$y(t)$')
...     pl.ylabel("$y'(t)$")
...     pl.grid()
...     pl.plot(array(les_y)[:, 0], array(les_y)[:, 1])
...     pl.title(legende)
...     pl.savefig(nomfichier)
```

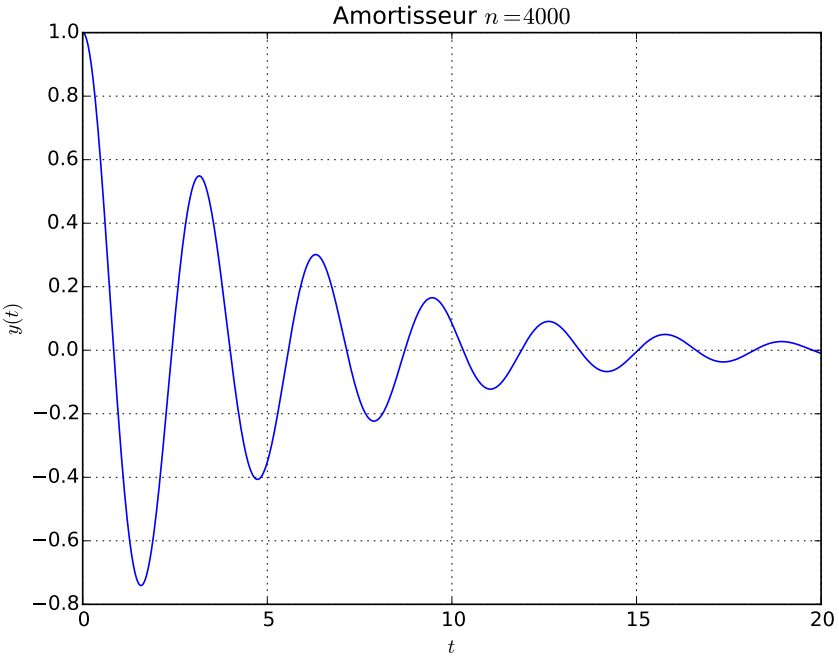


17 Influence du pas de discrétisation

On reprend l'exemple de l'amortisseur, avec $c = 100$ et avec différentes valeurs du pas de discrétisation $h = (t_1 - t_0)/n$:


Amortisseur $n=100$
50
40
30
20
10
0
−10
−20
−30
0
5
10
15
20
$y(t)$
t


Amortisseur $n=400$
1.0
0.5
0.0
−0.5
−1.0
0
5
10
15
20
$y(t)$
t



Le premier graphe est clairement délirant sur le plan physique ...
Plusieurs facteurs contribuent à l'erreur :

— des *erreurs de méthode* (la méthode d'Euler n'est pas parfaite)

— des *erreurs de calcul* (les flottants ne sont pas les réels)

Pour les erreurs de méthode, on dit qu'il y a *convergence* si :

1. la somme $e(h) = \sum_{k=0}^{n} |y(t_k) - y_k|$ des petites erreurs commises à chaque étape tend vers 0 quand h tend vers 0 ($e(h)$ est appelée *erreur de consistance relative à la solution* y).
2. et certaines conditions de *stabilité* de la méthode employée sont respectées (essentiellement, si l'on commet une petite erreur à chaque étape du calcul des y_k, menant à de nouvelles approximations $\tilde{y}_k$, alors si l'erreur de consistance relative aux y_k est petite, l'erreur entre les y_k et les $\tilde{y}_k$ doit être petite aussi).

On dit qu'une méthode est d'*ordre* p s'il existe une constante $K > 0$ telle que $e(h) \leq Kh^p$, autrement dit l'erreur de consistance est un $O(h^p)$.
La méthode d'Euler est une méthode d'ordre 1.

18 Autres méthodes

18.1 Méthode d'Euler implicite

La méthode d'Euler présentée jusqu'ici est dite *explicite* : y_{k+1} ne dépend que y_k et t_k.

$$y_{k+1} = y_k + hF(y_k, t_k)$$

Mais en reprenant les approximations qui ont conduit à ce schéma, nous pouvons aussi bien écrire :

$$y_{k+1} = y_k + hF(y_{k+1}, t_k)$$

ce qui mène à la méthode d'Euler *implicite*.

Elle a un gros inconvénient : pour trouver y_{k+1}, il faut résoudre une équation (numériquement).

Mais elle a un avantage : elle est souvent plus stable.

18.2 Méthode de Heun

Si $y_n = y(t_n)$, il existe $c \in]t_n, t_{n+1}[$ tel que

$$y(t_{n+1}) = y_n + hy'(c)$$

Tout le problème est d'estimer $y'(c)$.

On connaît l'idée de la méthode d'Euler explicite :

$$y'(c) \approx y'(t_n)$$

Donc on pose $\boxed{k_1 = F(y_n, t_n)}$.

Puis

$$\boxed{y_{n+1} = y_n + hk_1}$$

(erreur locale en $O(h^2)$)

L'idée de la méthode de Heun est la suivante :

$$y'(c) \approx \frac{y'(t_n) + y'(t_{n+1})}{2}$$

On pose $\boxed{k_1 = F(y_n, t_n)}$. Alors $k_1 = y'(t_n)$.

On estime $y'(t_{n+1})$ en utilisant $y'(t_{n+1}) = F(y(t_{n+1}), t_{n+1})$.

Pour cela, on estime $y(t_{n+1})$ par la méthode d'Euler explicite.

On pose donc : $\boxed{k_2 = F(y_n + hk_1, t_n + h)}$ (erreur en $O(h^2)$). Puis

$$\boxed{y_{n+1} = y_n + h\left(\frac{k_1 + k_2}{2}\right)}$$

On peut alors montrer que l'erreur locale est en $O(h^3)$.

18.3 Méthode de Runge-Kutta

Pour évaluer $y'(c)$, on calcule

$$\frac{k_1 + 2k_2 + 2k_3 + k_4}{6}$$

où k_1, k_2, k_3 et k_4 sont des évaluations des pentes :

k_1 pente en y_n

k_2 pente évaluée en $t_n + \frac{h}{2}$ en utilisant k_1 pour estimer $y(t_n + \frac{h}{2})$

k_3 pente évaluée en $y_n + \frac{h}{2}$ en utilisant k_2 pour estimer $y(t_n + \frac{h}{2})$

k_4 pente évaluée en $t_n + h$ en utilisant k_3 pour estimer $y(t_n + h)$.

$$\begin{aligned}
k_1 &= F(y_n, t_n) \\
k_2 &= F(y_n + \frac{h}{2}k_1, t_n + \frac{h}{2}) \\
k_3 &= F(y_n + \frac{h}{2}k_2, t_n + \frac{h}{2}) \\
k_4 &= F(y_n + hk_3, t_n + h) \\
y_{n+1} &= y_n + h\left(\frac{k_1 + 2k_2 + 2k_3 + k_4}{6}\right)
\end{aligned}$$

On peut alors montrer que l'erreur locale est en $O(h^5)$.
Il s'agit d'une méthode d'ordre de convergence 4.
Elle est facile à programmer et donc très populaire.