RESOLUTION APPROCHEE D'UNE EQUATION DIFFERENTIELLE

De nombreux phénomènes physiques se modélisent à l'aide d'équations différentielles pour lesquelles on ne dispose pas de méthode analytique pour obtenir l'expression des solutions exactes. On souhaite alors obtenir une solution "approchée" du problème pour en déduire un comportement qualitatif.

Pour cela, on utilise des méthodes de résolution approchée

I) Méthode d'Euler

Principe de la méthode d'Euler

On veut trouver une solution "approchée" de la solution d'un problème de Cauchy écrit sous la forme : $y' = F(y,t)$ et $y(t_0) = y_0$.

On veut évaluer de proche en proche une valeur approchée de $y(t)$ en t_0, $t_1 = t_0 + h$, $t_2 = t_0 + 2h$, $t_k = t_0 + kh$... où $h \in \mathbb{R}_+^*$ est appelé pas de la méthode.

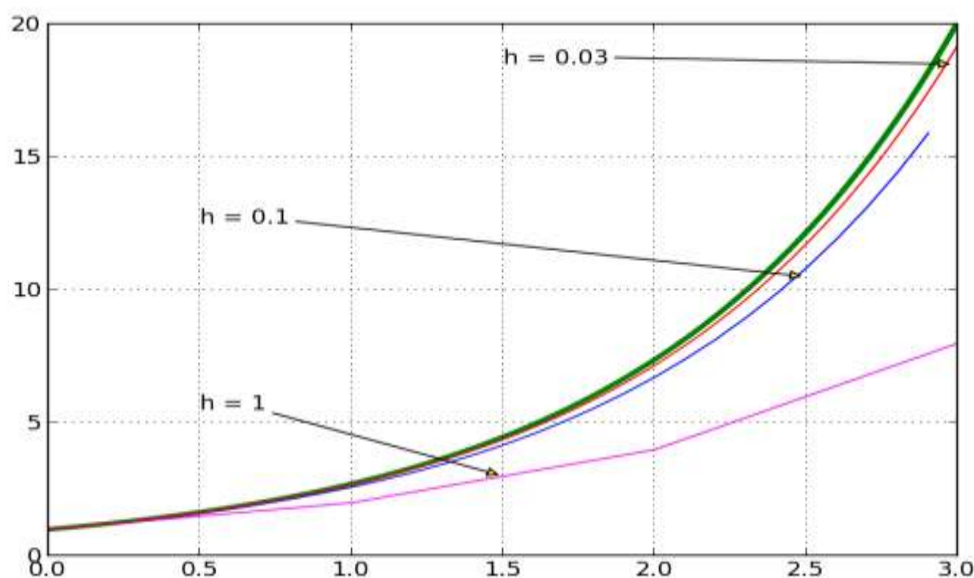
Comme dans la "vraie" solution, la tangente au point de coordonnées $(t, y(t))$ est de pente $F(y(t),t)$, la méthode d'Euler consiste à écrire que la tangente au point (t_k , y_k) est de pente $F(y_k, t_k)$.

On doit alors étudier la suite de premier terme y_0 et définie par la relation de récurrence : $\forall k \in \mathbb{N}$, $y_{k+1} = y_k + h\,F(y_k , t_k)$

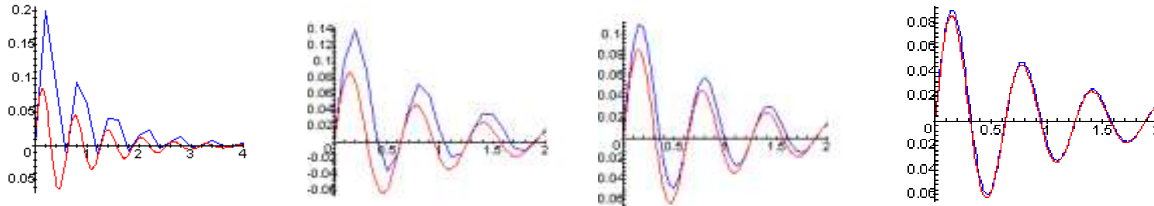
On relie alors les points (t_k, y_k) et (t_{k+1}, y_{k+1}) pour obtenir une courbe "approchée" de la courbe de la vraie solution.

En principe, plus le pas est petit plus on est proche de la solution exacte mais alors pour avoir la valeur en un point il faut calculer plus de valeurs et on a alors un risque de cumuler des erreurs d'arrondi.

Exemple 1 Avec l'équation : $y' = y$ avec la condition initiale $y(0) = y_0 = 1$ et les pas 1., 0.1 et 0.03, on obtient les courbes suivantes… On voit que la solution approchée s'écarte de la vraie solution (ici exp) lorsque t croit, mais que l'écart est de plus en plus petit avec h



Exemple 2 Avec l'équation : $y' + y = e^{-t}\cos(10t)$, la condition initiale $y_0 = 0$ et les pas respectifs h = 0.2, h = 0.1, h = 0.05 et h = 0.01, on obtient les courbes suivantes



Rem : La méthode d'Euler est dite d'ordre 1 car l'erreur commise en b par la méthode d'Euler en partant de a avec un pas h, est en $O(h^1)$. D'autres méthodes existent, comme par exemple celles de Runge-Kutta qui généralisent la méthode d'Euler et qui sont d'ordre 1, 2 ou 4 (erreur en $O(h^1)$, $O(h^2)$, $O(h^4)$) : c'est la méthode RK4.

Algorithme de la méthode d'Euler

Données : F, a, b, y_0, h
t ← a
y ← y_0
Liste_t = [t]
Liste_y = [y]
tant que t + h <= b **faire**
  y ← y + h * F(y, t)
  Liste_y ← Liste_y + [y]
  t ← t + h
  Liste_t ← Liste_t + [t]
Résultat : Liste_t , Liste_y

Programme Python

```
def euler (F, a, b, y0, h) :
    y = y0
    t = a
    Liste_t = [a]
    Liste_y = [y0]
    while t + h <= b :
        y = y + h*F(y,t)          ou, uniquement si y est scalaire : y += h*F(y,t)
        Liste_y.append(y)         ou :  Liste_y = Liste_y + [y]  (plus lent)
        t = t + h                 ou : t += h
        Liste_t.append(t)
    return ( Liste_t , Liste_y)
```

Terminaison, correction et complexité

Si on part bien de F continue, $a < b$ et $h > 0$ (et supérieure à la précision machine), il suffit, en notant $\lfloor x \rfloor$ la partie entière de x, de $\lfloor (b - a) / h \rfloor$ itérations pour obtenir la condition du test " $t + h > b$ " vérifiée : **la terminaison** de l'algorithme est assurée.

Pour ce qui est de la **correction**, la convergence de la suite vers la solution exacte dépend de la fonction F. Un théorème d'analyse numérique affirme que si F est de classe $\mathscr{C}^1$ et "uniformément lipschitzienne selon la seconde variable", alors la méthode est d'ordre 1 et donc lorsque le pas tend vers 0, la différence entre la vrai solution et la solution approchée tend vers 0.

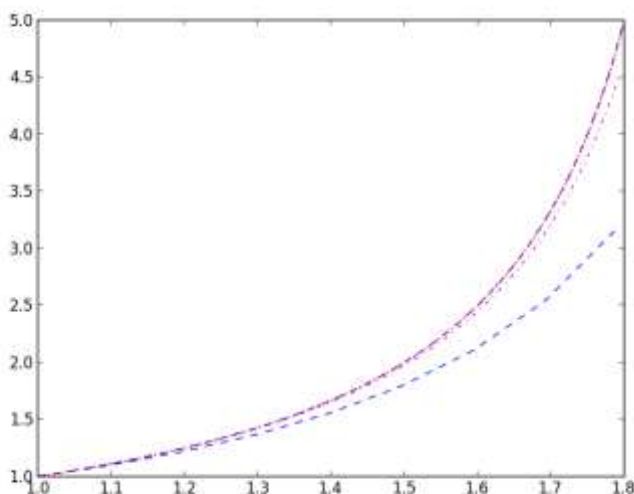
Pour ce qui est de la **complexité**, on voit que, si $n = \lfloor (b - a) / h \rfloor$, on effectue n calcul de F(t,y), 2n additions, n multiplications et 2n concaténations de listes. De plus, pour ce qui est de la complexité en mémoire, on a 2 variables numériques (t et y) mais également 2 variables de type list qui contiennent chacune n termes… De plus l'ajout d'un terme dans une liste longue peut avoir un coût en temps non négligeable… surtout si on effectue cet ajout un nombre important de fois….

A titre d'exemple voici quelques résultats de test de la méthode d'Euler pour différents pas (ainsi que différentes versions du programme Python). On travaille avec l'équation différentielle $y' = y$ et $y(0) = 1$ et on teste l'erreur en t = 3. (on connait la valeur de la solution exacte en 3. puisqu'il s'agit de exp(3))

pas	10^0	10^{-1}	10^{-2}	10^{-3}	10^{-4}	10^{-5}	10^{-6}	10^{-7}
Erreur	12.08	4.22	0.29	3.10^{-2}	5.10^{-3}	5.10^{-4}	5.10^{-5}	3.10^{-6}
version Liste_y = Liste_y + [y]	1.8×10^{-5} sec	5.8×10^{-5} sec	1.5×10^{-3} sec	4.1×10^{-2} sec	5.14 sec	non calculé (> 200 sec)	non calculé	non calculé
version Liste_y += [y]	1.6×10^{-5} sec	3.8×10^{-5} sec	5.7×10^{-4} sec	3.1×10^{-3} sec	2.5×10^{-2} sec	2.5×10^{-1} sec	2.49 sec	25.5 sec
version Liste_y.append(y)	2.1×10^{-5} sec	1.3×10^{-4} sec	5.8×10^{-4} sec	3.3×10^{-3} sec	2.3×10^{-2} sec	2.2×10^{-1} sec	2.23 sec	21.8 sec
version sans calcul de Liste_t	2.4×10^{-5} sec	3.4×10^{-5} sec	3.8×10^{-4} sec	2.6×10^{-3} sec	1.8×10^{-2} sec	1.8×10^{-1} sec	1.72 sec	16.3 sec

On constate que la différence entre la solution exacte et la solution approchée est de l'ordre de grandeur de h (majorée environ par 50 h) et que la durée d'exécution est proportionnelle au nombre d'itérations effectuées à l'exception notable de la version "Liste_y = Liste_y + [y] " dans laquelle la concaténation d'une liste de plus de 1000 termes et d'une liste d'un terme prend un temps important.

Autre exemple avec l'équation différentielle : $y' = y^2$ et la condition initiale $y(1) = 1$
On a tracé dans le graphe suivant, les solutions approchées pour les pas $h = 10^{-n}$ pour n entre 1 et 5



II) Equations scalaires d'ordre supérieur

Equation vectorielle et méthode d'Euler

En fait la méthode d'Euler s'applique également aux équations du type Y' = F(Y, t) avec Y fonction d'un intervalle de $\mathbb{R}$ vers $\mathbb{R}^n$ avec $n \geqslant 2$. On peut ainsi ramener une équation différentielle d'ordre 2, 3.. en une équation différentielle de degré 1. On procède à la vectorisation de l'équation.

Par exemple, supposons que l'on veuille résoudre le probléme de Cauchy (PC) suivant :

$$(PC) \qquad y^{(3)} + 2\,y'' - 6\,y' + y^2 = \exp(t) \qquad \text{et} \qquad y(0) = a_0\,,\ y'(0) = b_0 \ \text{ et }\ y''(0) = c_0.$$

On pose Z la fonction de $\mathbb{R}$ vers $\mathbb{R}^3$ définie par : $Z(t) = (y''(t), y'(t), y(t))$. Alors, le problème de Cauchy devient : $Z' = F(t, Z)$ avec la condition initiale $Z(0) = (a_0, b_0, c_0)$ où F est la fonction de $\mathbb{R}^4$ vers $\mathbb{R}^3$ définie par $F(\alpha, \beta, \chi, t) = (\exp(t) - 2\,\alpha + 6\,\beta - \chi^2 , \alpha, \beta)$

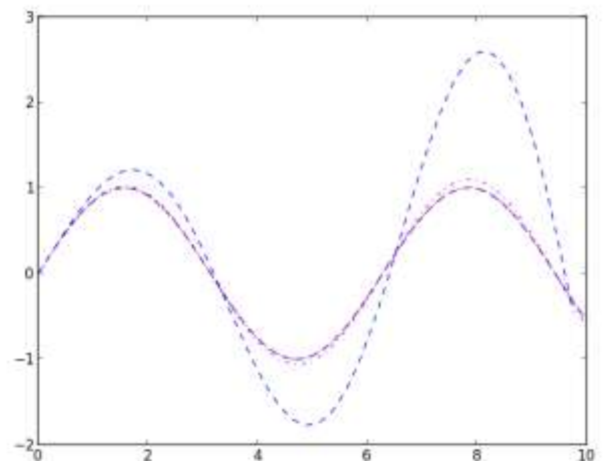
Attention cependant : lorsqu'à la première itération, on effectue y = y + h*F(y,t) on veut une addition de vecteurs et non une concaténation. Si y et h*F(y,t) sont deux listes, y + h * F(y,t) est pourtant une concaténation. Il faut utiliser le type array de la bibliothèque numpy

Programme Python

```
def Eulervectoriel (F, a, b, y0, h) :
    y = y0
    t = a
    Liste_t = [a]
    Liste_y = [y0]
    while t + h <= b :
        y = y + h*F(y,t)
        Liste_y.append(y)                ou :  Liste_y = Liste_y + [y]  (plus lent)
        t = t + h                        ou : t += h
        Liste_t.append(t)
    return ( Liste_t , Liste_y)
```

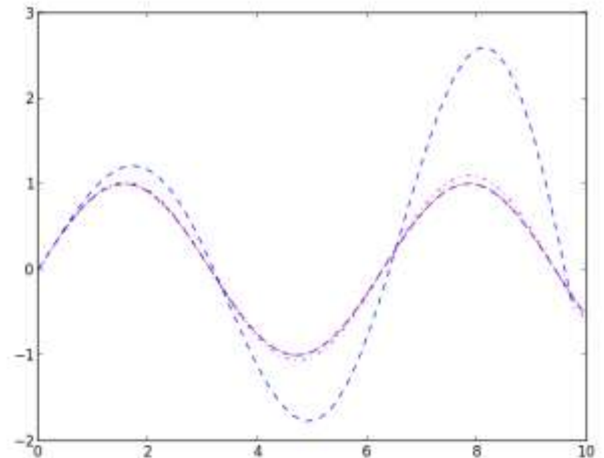
Exemple d'utilisation 1

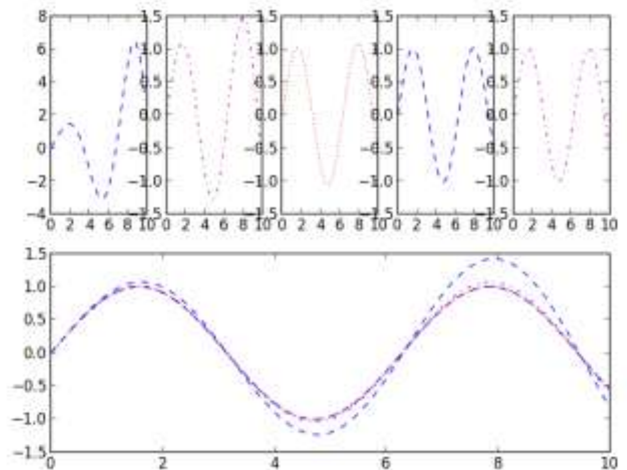
```
import numpy as np
import matplotlib.pyplot as plt
style = ['r:', 'b--', 'm-.']
F = lambda y,t: np.array([-y[1], y[0]])
plt.ion()
y0 = np.array([1,0])
for n in range(1,6):
    h = (10**(-n))/0.41
    res = Eulervectoriel(F, 0, 10, y0, h)
    abscisses = res[0]
    ordonnees = [res[1][k][1]
                        for k in range( len( res[1] ) ) ]
    plt.plot(abscisses, ordonnees, style[n%3])
    plt.draw()

plt.ioff()
plt.show()
```



Exemple d'utilisation 2

```
import numpy as np
import matplotlib.pyplot as plt
style = ['r:', 'b--', 'm-.']
F = lambda y,t: np.array([-y[1], y[0]])

plt.ion()

y0 = np.array([1,0])
sousgraphe = [251, 252, 253, 254, 255]

for n in range(1,6):
    plt.subplot( sousgraphe[n-1])
    h = (5**(-n))/0.41
    res = Eulervectoriel(F, 0, 10, y0, h)
    abscisses = res[0]
    ordonnees = [res[1][k][1] for k in range( len( res[1] ) ) ]
    plt.plot(abscisses, ordonnees, style[n%3])
    plt.draw()

plt.subplot(212)

for n in range(1,6):
    h = (5**(-n))/2.21
    res = Eulervectoriel(F, 0, 10, y0, h)
    abscisses = res[0]
    ordonnees = [res[1][k][1]
                 for k in range( len ( res[1] ) ) ]
    plt.plot(abscisses, ordonnees, style[n%3])
    plt.draw()

plt.ioff()
plt.show()
```





III) Utilisation de scipy

Les procédures précédentes ont évidemment été implémentées et améliorées dans les bibliothèques logicielles scientifiques Scipy. De plus la méthode d'Euler a elle-même été améliorée en une méthode de Runge-Kutta : par exemple la méthode RK4 .

Une méthode numérique d'intégration des équations différentielles est implémentée dans la fonction **scipy.optimize.odeint**

Cette fonction odeint s'utilise, pour résoudre l'équation y'(t) = F(y(t), t), sous la forme odeint(F, y0, t) où t est un tableau de temps de type array…

```
>>> from scipy.integrate import odeint
>>> odeint(lambda y,t : y , 1, [0, 0.5, 1.])
array([[ 1.        ],       [ 1.64872127],       [ 2.71828191]])
```