TP 21 équ. différentielles : méthode d'Euler

L'objectif du TP est de résoudre numériquement des équations différentielles du premier ordre. Pour ce faire, on va utiliser la méthode d'Euler, sur laquelle sont basées la plupart des méthodes usuelles de résolution. La méthode d'Euler est basée sur le résultat suivant : si f est de classe $\mathcal{C}^1$ sur un intervalle I, alors

$$\forall t \in I \ , \ \lim_{h\to 0} \frac{f(t+h)-f(t)}{h} = f'(t) \ .$$

On reformule ceci en écrivant que si $t \in I$ et si h est assez proche de 0 (avec $t+h \in I$) on a

$$f(t+h) \approx f(t) + hf'(t) \ .$$

Cela revient à remplacer localement (autour de t) la courbe de f par la tangente à la courbe en t : $f(t)+hf'(t)$ est l'ordonnée du point d'abscisse $t+h$ sur la tangente à la courbe en t. **Un choix d'approximation de $f(t+h)$ comme celui-ci est appelé un schéma de résolution numérique.** Nous verrons un autre schéma de résolution un peu plus loin. Considérons une équation différentielle (pas forcément linéaire) du premier ordre.

$$\forall x \in I \ , \ y'(x) = F(x, y(x)) \ ,$$

où $F : \mathbf{R}^2 \to \mathbf{R}$ est une fonction régulière (disons au moins de classe $\mathcal{C}^1$). Voici les exemples que nous traiterons dans ce TP :

(1) $y' = y$ (2) $y' + \ln(x)y = \sqrt{x}y^2$ (3) $y' = y(1-y)$ (4) $y' = \ln(y)$

Dans ces quatre cas, les fonctions F considérées sont $(x,y) \mapsto y$, $(x,y) \mapsto y(1-y)$, $(x,y) \mapsto \sqrt{x}y^2 - \ln(x)y$ et $(x,y) \mapsto \ln(y)$. Fonctions qui sont bien de classe $\mathcal{C}^1$ sur leurs intervalles de définition. Notez que la dernière est définie sur $\mathbf{R} \times \mathbf{R}_+^*$, ce qui impose que toute solution doit être positive strictement, et que la deuxième est définie sur $\mathbf{R}_+^* \times \mathbf{R}$, ce qui impose $I \subset \mathbf{R}_+^*$.

Appliquons la méthode d'Euler pour résoudre l'équation différentielle : si h est assez petit on aura

$$y(t+h) \approx y(t) + hy'(t) = y(t) + hF(y(t))$$

Autrement dit connaissant la valeur $y(t)$ de la solution en t, on aura une approximation de la valeur de la solution en $t+h$ en calculant $y(t)+hF(y(t))$. Bien entendu cette approximation sera d'autant meilleure que h sera petit. La qualité de cette approximation dépend aussi de caractéristiques de l'équation différentielle elle-même, mais ceci dépasse le cadre de notre étude. Supposons donc que l'on cherche à résoudre $y' = F(x, y(x))$ sur un intervalle $[a;b]$, avec un pas de temps h. Une équation différentielle du premier ordre a une unique solution vérifiant une condition initiale $y(x_0) = y_0$. On fournit donc un $x_0 \in [a;b]$ à notre programme, accompagné de la valeur correspondante y_0. Ensuite on progresse de x_0 à b en avançant par pas de h

$$y(x_0) \to y(x_0+h) \to y(x_0+2h) \to \cdots$$

(on s'arrête dès que l'on dépasse b) et on repart ensuite de t_0 pour progresser vers a par pas de $-h$:

$$\cdots \leftarrow y(x_0-2h) \leftarrow y(x_0-h) \leftarrow y(x_0)$$

(on s'arrête dès que l'on dépasse a). Le programme renverra alors une liste contenant toutes les listes $[x_0+kh, y(x_0+kh)]$ obtenues au cours de cette procédure.

Héritage de classe

On commence par créer une classe ODE (ordinary differential equation) comme suit :

```
class ODE():
    def __init__(self, F, h):
        self.F = F
        self.h = h

    def cond_ini(self, x0, y0):
        self.liste = [[x0,y0]]
```

1. Expliquez ce que font les deux méthodes `__init__` et `cond_ini`
2. Expliquez ce que contiennent les attributs F, h et *liste*.
3. Nous allons ajouter une méthode `resolution`, qui étant donné un schéma de résolution (ici la méthode d'Euler), renvoie la liste des couples $[x, y(x)]$ calculés. Ajoutez cette méthode à la classe ODE :

```
def resolution(self, a, b):
    self.indice = -1
    while self.liste[-1][0] <= b:
        self.liste.append(self.iteration())
    self.h = - self.h
    self.indice = 0
    while self.liste[0][0] >= a:
        self.liste.insert(0,self.iteration())
    return self.liste
```

   Expliquez chaque ligne de ce code.

4. Cette méthode `resolution` nécessite une autre méthode : la méthode `iteration`, que nous n'avons pas encore codée ! Si l'on veut pouvoir implémenter plusieurs schémas de résolution numérique (la méthode d'Euler en étant un), il serait pratique de pouvoir facilement passer d'un schéma à un autre sans avoir à modifier notre code à chaque fois. Nous allons utiliser l'héritage de classe, typique de la programmation orientée objet. On définit une nouvelle classe, mais qui héritera des attributs et méthodes de la classe ODE :

```
class Euler(ODE):
    def iteration(self):
        F, h = self.F, self.h
        [x, y] = self.liste[self.indice]
        return [x + h, y + h * F(x, y)]
```

   On voit que pour que tout fonctionne, les attributs F et h de la classe manipulée par la méthode itération doivent déjà être initialisés, ainsi que *liste* et *indice*. On peut vérifier dans notre code précédent que lors d'un appel à `iteration`, ces attributs sont déjà définis.

5. Il est temps de résoudre notre première équation différentielle. Codez en Python (en dehors des classes ODE et Euler) la fonction F correspondant au premier exemple proposé dans l'introduction. Nous connaissons l'ensemble des solutions de cette équation : ce sont les $x \mapsto Ce^x$. Si on donne la condition initiale $y(x_0) = y_0$, on aura alors $y_0 = Ce^{x_0}$, ce qui détermine la constante $C = y_0 e^{-x_0}$. Nous connaissons donc l'unique solution associée à la condition initiale $[x_0, y_0]$, ce qui permettra de comparer la théorie et notre résolution.

   Pour lancer la résolution de l'équation différentielle, on commence par fixer des valeurs de a, b et h. Puis on créée un objet `approx` de type `Euler` à l'aide de la commande `Euler(F, h)`. On fixe les conditions initiales souhaitées avec la méthode `cond_ini`. Enfin on créée une liste `liste_approx` contenant les résultats de la méthode `resolution`, avec la commande
   `liste_approx = approx.resolution(a, b)`.

   Faites toutes ces étapes et vérifiez que vous obtenez une liste cohérente de résultats.

6. Avec matplotlib tracez la courbe du résultat et comparez avec la courbe théorique (ce n'est pas si facile car il faut extraire de la liste de liste `liste_approx` deux listes : la liste des abscisses et la liste des ordonnées).

   Vous pouvez utiliser `plt.plot(abscisses, ordonnees, 'b+')` pour que matplotlib fasse des symboles + bleus au lieu d'une ligne continue. Les autres styles possibles sont par exemple 'ro' pour des cercles rouges ou encore 'g-' pour une ligne verte.

7. Pour profiter de l'héritage de classe on peut coder une deuxième méthode, plus précise, la méthode de Runge-Kutta d'ordre 4. On créée une nouvelle classe `RK4` comme suit :

```python
class RK4(ODE):
    def iteration(self):
        F, h = self.F, self.h
        [x, y] = self.liste[self.indice]
        k1 = h * F(x, y)
        k2 = h * F(x + h/2, y + k1/2)
        k3 = h * F(x + h/2, y + k2/2)
        k4 = h * F(x + h, y + k3)
        return [x + h, y + (k1 + 2*k2 + 2*k3 + k4) / 6]
```

Elle aussi héritera des attributs et méthodes de `ODE`, tout en définissant d'une autre manière la méthode `iteration`.

8. Tentez de résoudre l'équation (3) avec la méthode de Runge-Kutta. Il faut tester des jeux de paramètres a, b, h, x_0, y_0.
9. On appelle **équilibre d'une équation différentielle** une solution constante de cette équation différentielle. Trouvez par le calcul deux équilibres de l'équation (3). Confirmez numériquement ce résultat.
10. À l'aide d'une boucle for dans laquelle vous ajouterez successivement des courbes à un même graphique, affichez un ensemble de solutions de l'équation (3) pour lesquelles y_0 prend une dizaine de valeurs réparties entre 0 et 1. On choisira $a = 0$, $b = 5$, $h = 0.01$ et $x_0 = 0$.

Formulaire :

Vous aurez besoin des commandes suivantes (dans le désordre)

```
### LES MODULES USUELS
from math import *
import numpy as np
import matplotlib.pyplot as plt
import numpy.random as rd

### LES VARIABLES ALEATOIRES
rd.randint(a,b,N)
rd.random(N)
rd.binomial(n,p,N)
rd.geometric(p,N)
rd.poisson(lambda,N)

## LES LISTES
np.linspace(a,b,n)
for compteur in liste :
for compteur in range(a,b):
[ f(compteur) for compteur in liste]
[ f(compteur) for compteur in range(a,b)]

### COMMANDES CLASSIQUES
while condition:
def nom_fonction( variables ) :
return resultat
print( 'message', variable )
if condition :
else :
log(x)
plt.plot( abscisses , ordonnees )

### CLASSES EN PYTHON
class nom_de_classe :
class nom_de_classe(classe_d_heritage):
def __init__(self, autres_parametres):
def __str__(self):

### MATRICES EN PYTHON
np.array([[...],...,[...]])
A.shape
np.zeros([p,n])
np.eye(n)
np.ones([p,n])
A.dot(B)
np.dot(A,B)
A.transpose()
A.all()
```