TD numérique pour la physique n°1 – ED d'ordre 1 (méthode d'Euler)

Fichier à compléter : `TD 1 Euler ordre 1.py`, à récupérer sur le site de physique.

Théorie

L'objectif de la résolution numérique d'équations différentielles est d'obtenir, pour une liste $[t_k]$ de dates souhaitées, la liste correspondante des valeurs de la fonction inconnue du temps $[y_k]$.

Naturellement, toute méthode de résolution conduit à des erreurs : on obtiendra en réalité $[\widetilde{y_k}]$, liste des estimations des valeurs de la fonction à ces dates.

On peut introduire la liste des erreurs commises $[\varepsilon_k]=[\widetilde{y_k}-y_k]$, mais la valeur vraie y_k est bien sûr inconnue : dans le TD, on travaillera avec des ED qu'on sait résoudre analytiquement afin de déterminer les erreurs $[\varepsilon_k]$, ce qui nous permettra de juger la méthode de résolution utilisée.

Une équation différentielle d'ordre 1, linéaire ou non, indépendante du temps ou non, peut se mettre sous la forme canonique suivante, adaptée à la programmation :

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

Q1. Déterminer la fonction fondamentale de l'équadiff *F* dans les cas suivants :

- Tension de charge *u* d'un condensateur dans un circuit RC soumis à une tension constante E.
- Envoi d'une tension alternative $e(t)=E\cos\omega t$ à l'entrée d'un filtre passe bas d'ordre 1 de fonction de transfert $\underline{H}=\dfrac{1}{1+j\omega\tau}$;
- Évolution de la vitesse *v* d'un corps en mouvement vertical vers le bas dans le champ de pesanteur, soumis à des frottements turbulents de coefficient β.

La donnée de l'équation différentielle permet donc de calculer la dérivée de la fonction à toute date, connaissant sa valeur à cette date.

Comme la condition initiale $y_0=y(t_0)$ est connue, toutes les méthodes de résolution fonctionnent de proche en proche :

- utilisation de la fonction *F* pour obtenir la (ou certaines) dérivée de la fonction.
- utilisation de la (ou des) dérivée calculée pour estimer la valeur suivante $\widetilde{y}_1$ à la date t_1
- etc.

La **méthode d'Euler** est la plus intuitive et la plus simple : comme on a choisi, pour avoir une bonne précision, des dates proches les unes des autres, on peut considérer que la fonction inconnue est presque affine entre deux dates successives, donc utiliser l'**approximation affine de la tangente**.

On en déduit l'estimation de la valeur suivante, à partir de la précédente :

$\widetilde{y}_1=y_0+y'(t_0)(t_1-t_0)=y_0+F(y_0,t_0)(t_1-t_0)$ pour la première itération (y_0 est une valeur vraie), puis, dans le cas général, pour les valeurs suivantes :

$$\boxed{\text{Euler :}\quad \widetilde{y}_{k+1}=\widetilde{y}_k+F(\widetilde{y}_k,t_k)(t_{k+1}-t_k)}$$

Q2. Faire un schéma rapide d'une fonction non affine quelconque passant par le point $(t_k ; y_k)$ supposé exact, tracer sa tangente en t_k, placer une date t_{k+1} après t_k, et marquer l'estimation suivante $\widetilde{y}_{k+1}$ ainsi que l'erreur commise ε_{k+1}.

On voit que la méthode d'Euler introduit une erreur, d'autant plus grande que la concavité de la fonction, donc sa dérivée seconde, est importante.

Q3. On donne l'équation différentielle suivante, $\tau y' + y = 0$; où τ, constante de temps, est strictement positive.

Sans la résoudre, obtenir la dérivée temporelle seconde y'' en fonction de y et de τ.

Conclure sur l'écart entre la solution obtenue par la méthode d'Euler et la fonction vraie, en supposant que la fonction y reste toujours strictement positive.

Pour corriger ce problème, il existe d'autres méthodes de résolution (Bac+3) qui calculent plusieurs dérivées : par exemple en t_k, en t_{k+1} (en estimant la valeur $\widetilde{y}_{k+1}$ une première fois par Euler), ainsi qu'à la date milieu de t_k et t_{k+1}, puis font une moyenne pondérée des dérivées obtenues pour estimer au mieux $\widetilde{y}_{k+1}$: l'évolution de la dérivée autour du point considéré permet d'estimer la dérivée seconde de la fonction et de la corriger.

C'est la cas de la méthode utilisée de base par le solveur Python, dans la fonction `odeint`, appelée méthode de Runge-Kutta d'ordre 4 : RK4 (on démontre que l'erreur ε_{k+1} est alors $o\left((t_{k+1}-t_k)^4\right)$).

Description du TD

On va implémenter les différentes méthodes sur l'exemple de l'équation différentielle $\tau y' + y = \ldots$ avec $\tau = 1$ et la condition initiale $y_0 = 1$.

A) tout d'abord avec un second membre nul : décharge d'un condensateur, typiquement ;

B) ensuite avec un second membre sinusoïdal $e(t) = E\cos\omega t$ (avec $E = 1\,(\mathrm{V})$ et $\omega = 8\,(\mathrm{rad/s})$).

Q4. Obtenir la solution exacte de l'équation homogène, avec la CI donnée.

Questions 5 à 11. Voir fichier Python à compléter. Pour la Q8, voir les exigibles ci-dessous.

Q12. Sauvegarder le fichier sous `TD 1 Euler ordre 1 hom.py`, puis renommer le fichier initial en `TD 1 Euler ordre 1 cos.py`

Modifier l'ensemble du code de ce nouveau fichier pour travailler sur l'ED $\tau y' + y = E\cos\omega t$ avec les valeurs numériques données dans la description.

On tracera à nouveau les deux fonctions obtenues par Euler et par RK4 (`odeint`), avec un nombre de points limité pour voir les défauts, ainsi que la solution exacte (en utilisant les valeurs numériques) :

$$y(t) = \left(y_0 - \frac{E}{\sqrt{1+(\omega\tau)^2}} \cos \mathrm{Arctan}(\omega\tau) \right) \exp\left(-\frac{t}{\tau}\right) + \frac{E}{\sqrt{1+(\omega\tau)^2}} \cos\left(\omega t - \mathrm{Arctan}(\omega\tau)\right)$$

Dans le module `math`, la fonction Arctan est `atan`.

Démontrer que l'expression proposée est correcte : on ne cherchera pas à remplacer y dans l'équation différentielle (calcul délicat **et** pas convaincant…), mais on appliquera la méthodologie de résolution des ED linéaires non homogènes.
(On peut néanmoins vérifier facilement que $y(0) = y_0$).

Exigibles numériques

- Passage d'une fonction comme argument à une autre fonction
- Utilisation de l'aide Python : ici, sur la fonction `odeint`, où l'import correspondant a dû être fait auparavant (cf début du fichier : erreur sinon).

```
>>> help(odeint)
Help on function odeint in module scipy.integrate.odepack:

odeint(func, y0, t, args=(), Dfun=None, col_deriv=0, full_output=0, ml=None, mu=None, rtol=Non
e, atol=None, tcrit=None, h0=0.0, hmax=0.0, hmin=0.0, ixpr=0, mxstep=0, mxhnil=0, mxordn=12, m
xords=5, printmessg=0, tfirst=False)
    Integrate a system of ordinary differential equations.

    .. note:: For new code, use `scipy.integrate.solve_ivp` to solve a
              differential equation.

    Solve a system of ordinary differential equations using lsoda from the
    FORTRAN library odepack.

    Solves the initial value problem for stiff or non-stiff systems
    of first order ode-s::

        dy/dt = func(y, t, ...)  [or func(t, y, ...)]

    where y can be a vector.

    .. note:: By default, the required order of the first two arguments of
              `func` are in the opposite order of the arguments in the system
              definition function used by the `scipy.integrate.ode` class and
              the function `scipy.integrate.solve_ivp`.  To use a function with
              the signature ``func(t, y, ...)``, the argument `tfirst` must be
              set to ``True``.
```

On regarde d'abord quels sont les arguments à passer impérativement à la fonction : ce sont ceux qui ne sont pas accompagnés de '=' dans la déclaration de fonction ; ils sont décrits plus précisément ensuite.

- `func, y0, t`

Les autres types d'arguments sont également décrits ensuite :

- `args=()` indique qu'on peut passer des arguments en plus, dans un tuple ; ce qu'en fait la fonction est détaillé dans l'aide.
- Les arguments du type `tfirst=False` : on peut les ignorer complètement, ce qu'on fait en général.

  Pour comprendre, techniquement : même si l'on ne passe rien concernant cet argument, `tfirst` est alors connu par `odeint`, et vaut `False`.

  Si par hasard on souhaite le modifier à `True`, il faudra alors appeler `odeint` (simple exemple ci-dessous pour les arguments obligatoires) ainsi :

  `odeint(F,1,tList,tfirst=True)`

  l'aide explique ensuite que c'est ainsi qu'il faut procéder si la fonction F (c'est-à-dire `func`) a été implémentée par

  `def F(t,y)` au lieu de `def F(y,t)`

  Nous coderons dans les TD les fonctions des ED correctement : nous n'aurons donc jamais besoin de modifier cet argument.

```
Parameters
----------
func : callable(y, t, ...) or callable(t, y, ...)
    Computes the derivative of y at t.
    If the signature is ``callable(t, y, ...)``, then the argument
    `tfirst` must be set ``True``.
y0 : array
    Initial condition on y (can be a vector).
t : array
    A sequence of time points for which to solve for y.  The initial
    value point should be the first element of this sequence.
    This sequence must be monotonically increasing or monotonically
    decreasing; repeated values are allowed.
args : tuple, optional
    Extra arguments to pass to function.
Dfun : callable(y, t, ...) or callable(t, y, ...)
    Gradient (Jacobian) of `func`.
    If the signature is ``callable(t, y, ...)``, then the argument
    `tfirst` must be set ``True``.
col_deriv : bool, optional
    True if `Dfun` defines derivatives down columns (faster),
    otherwise `Dfun` should define derivatives across rows.
full_output : bool, optional
    True if to return a dictionary of optional outputs as the second output
printmessg : bool, optional
    Whether to print the convergence message
tfirst: bool, optional
    If True, the first two arguments of `func` (and `Dfun`, if given)
    must ``t, y`` instead of the default ``y, t``.

    .. versionadded:: 1.1.0
```

On regarde ensuite ce que retourne la fonction :

```
Returns
-------
y : array, shape (len(t), len(y0))
    Array containing the value of y for each desired time in t,
    with the initial value `y0` in the first row.
infodict : dict, only returned if full_output == True
    Dictionary containing additional output information

    =======  ============================================================
    key      meaning
    =======  ============================================================
    'hu'     vector of step sizes successfully used for each time step.
    'tcur'   vector with the value of t reached for each time step.
             (will always be at least as large as the input times).
    'tolsf'  vector of tolerance scale factors, greater than 1.0,
             computed when a request for too much accuracy was detected.
    'tsw'    value of t at the time of the last method switch
             (given for each time step)
    'nst'    cumulative number of time steps
    'nfe'    cumulative number of function evaluations for each time step
    'nje'    cumulative number of jacobian evaluations for each time step
    'nqu'    a vector of method orders for each successful step.
    'imxer'  index of the component of largest magnitude in the
             weighted local error vector (e / ewt) on an error return, -1
             otherwise.
    'lenrw'  the length of the double work array required.
    'leniw'  the length of integer work array required.
    'mused'  a vector of method indicators for each successful time step:
             1: adams (nonstiff), 2: bdf (stiff)
    =======  ============================================================
```

Pour utiliser la fonction, c'est en général suffisant, avec des exemples souvent utiles proposés dans l'aide (ici, ils sont intéressants pour le TD suivant, n°2).

```
Other Parameters
----------------
ml, mu : int, optional
    If either of these are not None or non-negative, then the
    Jacobian is assumed to be banded.  These give the number of
    lower and upper non-zero diagonals in this banded matrix.
    For the banded case, `Dfun` should return a matrix whose
    rows contain the non-zero bands (starting with the lowest diagonal).
    Thus, the return matrix `jac` from `Dfun` should have shape
    ``(ml + mu + 1, len(y0))`` when ``ml >=0`` or ``mu >=0``.
    The data in `jac` must be stored such that ``jac[i - j + mu, j]``
    holds the derivative of the `i`th equation with respect to the `j`th
    state variable.  If `col_deriv` is True, the transpose of this
    `jac` must be returned.
rtol, atol : float, optional
    The input parameters `rtol` and `atol` determine the error
    control performed by the solver.  The solver will control the
    vector, e, of estimated local errors in y, according to an
    inequality of the form ``max-norm of (e / ewt) <= 1``,
    where ewt is a vector of positive error weights computed as
    ``ewt = rtol * abs(y) + atol``.
    rtol and atol can be either vectors the same length as y or scalars.
    Defaults to 1.49012e-8.
tcrit : ndarray, optional
    Vector of critical points (e.g. singularities) where integration
    care should be taken.
h0 : float, (0: solver-determined), optional
    The step size to be attempted on the first step.
hmax : float, (0: solver-determined), optional
    The maximum absolute step size allowed.
hmin : float, (0: solver-determined), optional
    The minimum absolute step size allowed.
ixpr : bool, optional
    Whether to generate extra printing at method switches.


mxstep : int, (0: solver-determined), optional
    Maximum number of (internally defined) steps allowed for each
    integration point in t.
mxhnil : int, (0: solver-determined), optional
    Maximum number of messages printed.
mxordn : int, (0: solver-determined), optional
    Maximum order to be allowed for the non-stiff (Adams) method.
mxords : int, (0: solver-determined), optional
    Maximum order to be allowed for the stiff (BDF) method.

See Also
--------
solve_ivp : Solve an initial value problem for a system of ODEs.
ode : a more object-oriented integrator based on VODE.
quad : for finding the area under a curve.

Examples
--------
The second order differential equation for the angle `theta` of a
pendulum acted on by gravity with friction can be written::

    theta''(t) + b*theta'(t) + c*sin(theta(t)) = 0

where `b` and `c` are positive constants, and a prime (') denotes a
derivative.  To solve this equation with `odeint`, we must first convert
it to a system of first order equations.  By defining the angular
velocity ``omega(t) = theta'(t)``, we obtain the system::

    theta'(t) = omega(t)
    omega'(t) = -b*omega(t) - c*sin(theta(t))
```

```
Let `y` be the vector [`theta`, `omega`].  We implement this system
in python as:

>>> def pend(y, t, b, c):
...     theta, omega = y
...     dydt = [omega, -b*omega - c*np.sin(theta)]
...     return dydt
...

We assume the constants are `b` = 0.25 and `c` = 5.0:

>>> b = 0.25
>>> c = 5.0

For initial conditions, we assume the pendulum is nearly vertical
with `theta(0)` = `pi` - 0.1, and is initially at rest, so
`omega(0)` = 0.  Then the vector of initial conditions is

>>> y0 = [np.pi - 0.1, 0.0]

We will generate a solution at 101 evenly spaced samples in the interval
0 <= `t` <= 10.  So our array of times is:

>>> t = np.linspace(0, 10, 101)

Call `odeint` to generate the solution.  To pass the parameters
`b` and `c` to `pend`, we give them to `odeint` using the `args`
argument.

>>> from scipy.integrate import odeint
>>> sol = odeint(pend, y0, t, args=(b, c))


The solution is an array with shape (101, 2).  The first column
is `theta(t)`, and the second is `omega(t)`.  The following code
plots both components.

>>> import matplotlib.pyplot as plt
>>> plt.plot(t, sol[:, 0], 'b', label='theta(t)')
>>> plt.plot(t, sol[:, 1], 'g', label='omega(t)')
>>> plt.legend(loc='best')
>>> plt.xlabel('t')
>>> plt.grid()
>>> plt.show()
```