Classe de PCSI

Cours de physique

Année 2023-2024

Capacités numériques

CAMILLE VERNET
Collège • Lycée • Cpge

Première partie

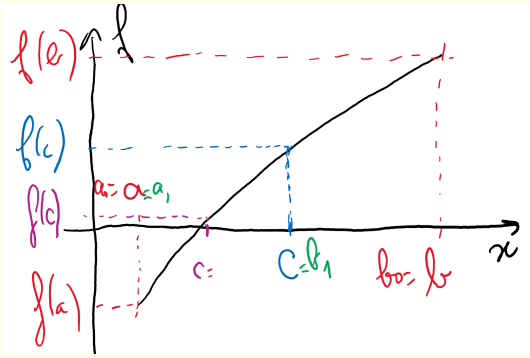
Résolution numérique de $f(x) = 0$ par dichotomie

I Principe de la méthode

Méthode : Résolution de $f(x) = 0$ par dichotomie

On considère une fonction $f : I = [a, b] \longrightarrow \mathbb{R}$, continue et strictement monotone sur l'intervalle $[a, b]$, qui change de signe sur l'intervalle.

On cherche la solution approchée à l'équation $f(x) = 0$ sur l'intervalle $[a, b]$.



- On se place au milieu de l'intervalle $[a, b]$ ($c = (a + b)/2$) et on compare le signe de $f(a)$ avec le signe de $f(c)$ en regardant le signe de $f(a) \times f(c)$.
  - Si $f(a) \times f(c) < 0$, alors f s'annule sur l'intervalle $[a, c]$. Et on procède de la même façon sur ce nouvel intervalle.
  - Sinon, f s'annule sur l'intervalle $[c, b]$. Et on procède de la même façon sur ce nouvel intervalle.
- On procède ainsi jusqu'à un certain critère imposé, qui peut porter soit sur la largeur de l'intervalle de recherche (on cherche jusqu'à ce que $|b - a|$ devienne inférieur à un certain ε), soit sur la proximité de f à 0 (on cherche jusqu'à ce que $|f(c)|$ devienne inférieur à un certain ε).

Comment choisir l'intervalle $[a, b]$ de recherche ?

- Représenter la fonction f dont on cherche l'annulation ;
- Déterminer un intervalle pertinent, sur lequel f est strictement monotone et s'annule.

II Traduction en python

La précision ε peut porter sur :

— la largeur de l'intervalle qui encadre la valeur approchée que l'on renvoie :

```
def dichotomie(f,a,b,eps):
    while  abs(b-a)>eps : # tant que |b-a| est supérieur à eps
        c = (a+b)/2 # centre de l'intervalle
        if f(a)*f(c)<0 # f(a) et f(c) sont de signes opposés
            b=c # f s'annule sur l'intervalle [a,c]
       else:
            a=c # sinon, f s'annule sur [c,b]
    return c
```

— la proximité de la fonction $f(c)$ à 0, où c est la valeur approchée de la racine déterminée :

```
def dichotomie(f,a,b,eps):
    c = (a+b)/2 # centre de l'intervalle
    while  abs(f(c))>eps : # tant que |f(c)| est supérieur à eps
        if f(a)*f(c)<0 # f(a) et f(c) sont de signes opposés
            b=c # f s'annule sur l'intervalle [a,c]
       else:
            a=c # sinon, f s'annule sur [c,b]
       c = (a+b)/2 # centre de l'intervalle
    return c
```

III Utilisation des fonctions de python

L'algorithme de dichotomie est déjà programmé dans python : `bisect` dans la bibliothèque `scipy.integrate`. L'utilisation de cette fonction se fait avec :

```
bisect(f, a, b)
```

où

— `f` attend la fonction (à définir au préalable) dont on cherche l'annulation,

— `a` et `b` est l'intervalle de recherche, $f(a)$ et $f(b)$ ne doivent pas être de même signe, et f doit être strictement monotone sur $[a, b]$.

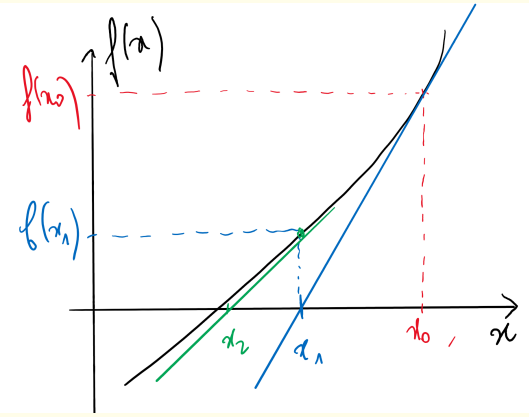
Deuxième partie

Résolution numérique de $f(x) = 0$ par la méthode de Newton

I Principe de la méthode

Méthode : Résolution de $f(x) = 0$ par la méthode de Newton

La méthode de Newton est une autre méthode de résolution numérique de $f(x) = 0$.



- On part du point d'abscisse x_0.
  - On considère la tangente en ce point, d'équation $y = f(x_0) + f'(x_0)(x - x_0)$.
  - On cherche l'intersection de la tangente avec l'axe des abscisses.

    La tangente coupe l'axe des abscisses en $x_1 = x_0 - \dfrac{f(x_0)}{f'(x_0)}$.
- On part maintenant du point d'abscisse x_1.
  - On considère la tangente en ce point, d'équation $y = f(x_1) + f'(x_1)(x - x_1)$
  - On cherche son intersection avec l'axe des abscisses.

    La tangente coupe l'axe des abscisses en $x_2 = x_1 - \dfrac{f(x_1)}{f'(x_1)}$.
- Et ainsi de suite...

On détermine les termes de la suite (x_n) des intersections des tangentes successives définie par récurrence :

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

On cherche ces intersections successives jusqu'à la vérification d'un certain critère, qui peut porter :

- sur les deux intersections successives x_n et x_{n+1}, qui doivent être suffisamment proches $|x_{n+1} - x_n| < \varepsilon$;
- ou sur la proximité de f à 0 à ε près.

Comment choisir l'abscisse x_0 de début de recherche ?

- Représenter la fonction f dont on cherche l'annulation ;
- Choisir x_0 proche de l'annulation de f, à distance d'un extremum local de f.

Dichotomie ou Newton ? La méthode de Newton a l'avantage de converger plus rapidement que la méthode de dichotomie, mais nécessite la connaissance de la dérivée de la fonction f, ou alors son calcul. De plus, la dérivée ne doit pas s'annuler sur l'intervalle de recherche.

II Traduction en python

On écrit la fonction `newton` qui nécessite 4 arguments en entrée :

— la fonction `f` dont on cherche l'annulation ;

— la dérivée `df` de la fonction `f` ;

— l'abscisse de début de la recherche, `x0` ;

— la précision souhaitée, `eps`, telle que $|x_{n+1} - x_n| < \varepsilon$ ou $|f(x_n)| < \varepsilon$ (`newton2`).

```
def newton(f,df,x0,eps):
    """
    f : fonction dont on cherche le zéro
    df : dérivée de la fonction f
    x0 : début de la recherche
    eps : précision souhaitée sur xn, tel que |x_{n+1}-x_n|<eps
    renvoie la valeur de xn
    """
    xn = x0 # initialisation de la suite des xn avec x0
    xn1 = xn - f(xn) / df(xn)
    while abs(xn1-xn)>eps :
        xn1 , xn = xn1 - f(xn1) / df(xn1) , xn1 # calcul de x_{n+1}
    return xn
```

```
def newton2(f,df,x0,eps):
    """
    f : fonction dont on cherche le zéro
    df : dérivée de la fonction f
    x0 : début de la recherche
    eps : précision souhaitée sur xn, tel que |f(xn)|<eps
    renvoie la valeur de xn
    """
    xn = x0 # initialisation de la suite des xn avec x0
    while abs(f(xn))>eps :
        xn = xn - f(xn) / df(xn) # calcul de x_{n+1} à partir de xn, f(xn)
   et df(xn)
    return xn
```

III Utilisation des fonctions de python

L'algorithme de Newton est déjà programmé dans python : `newton` dans la bibliothèque `scipy.integrate`. L'utilisation de cette fonction se fait avec :

```
newton(func, x0, fprime)
```

où

— `func` attend la fonction (à définir au préalable) dont on cherche l'annulation,

— `x0` le point de départ,

— `func` attend la dérivée de la fonction (à définir au préalable) dont on cherche l'annulation.

Troisième partie

Calcul numérique d'une dérivée

I Dérivée d'une fonction

I.1 Principe

Méthode : Calcul numérique d'une dérivée

Pour calculer le nombre dérivé d'une fonction f en un point x, on peut utiliser plusieurs méthodes :

- Schéma avant : $f'(x) \approx \dfrac{f(x+h)-f(x)}{h}$
- Schéma arrière : $f'(x) \approx \dfrac{f(x)-f(x-h)}{h}$
- Schéma milieu : $f'(x) \approx \dfrac{f(x+h)-f(x-h)}{2h}$

avec h un pas de calcul choisi suffisamment petit pour rendre compte des variations locales de la fonction f, mais qui devra rester supérieur au plus petit écart entre deux réels codés par deux flottants différents (2×10^{-16} en python sur 64 bits).

I.2 Implémentation en python

```
def deriv_avant(f,x,h):
    return (f(x+h)-f(x))/h
```

```
def deriv_arriere(f,x,h):
    return (f(x)-f(x-h))/h
```

```
def deriv_milieu(f,x,h):
    return (f(x+h)-f(x-h))/(2*h)
```

II Dérivée d'un tableau de valeurs

Soient t et y deux listes qui contiennent les temps et les positions d'un objet à ces instants.

La vitesse à l'instant t_i est approximée par (dérivée à droite) : $v(t_i) = \dfrac{y(t_{i+1})-y(t_i)}{\Delta t}$, avec $\Delta t = t_{i+1} - t_i$ le pas de temps.

```
t = ... # tableau des instants
y = ... # tableau des positions
v = np.zeros(len(y)-1) # initialisation du tableau des vitesses, qui a un élément de moins que y, car la dernière vitesse ne peut pas être calculée selon la définition précédente
for i in range(0,len(y)-1):
    v[i] = ( y[i+1] - y[i] ) / (t[i+1] - t[i])  # v[i] est la vitesse à l'instant ti
```

.

Quatrième partie

Calcul approché d'une intégrale

I Principe de la méthode

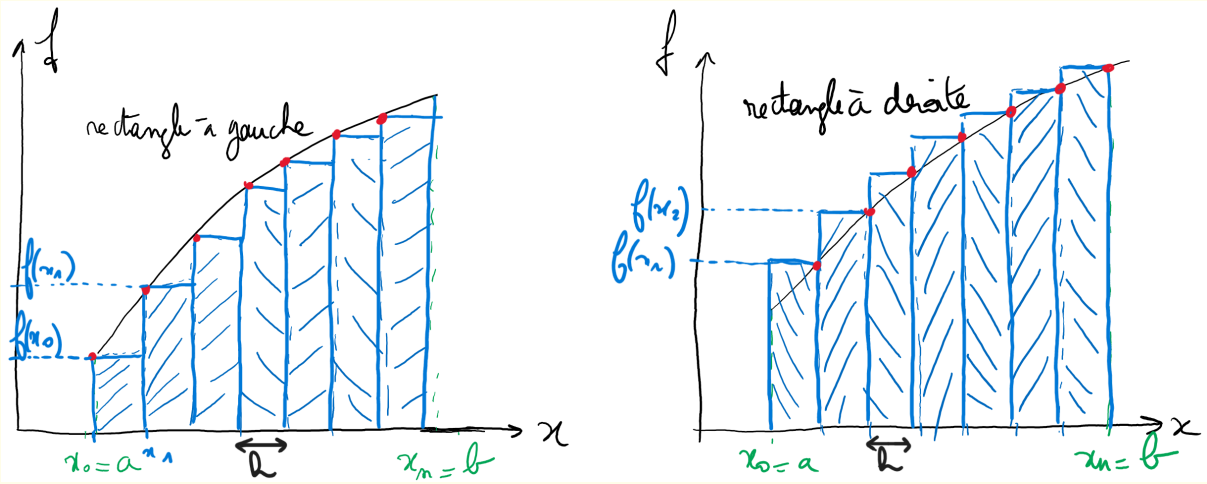
Méthode : Méthode des rectangles pour l'intégration numérique

On souhaite calculer $\int_a^b f(x)\mathrm{d}x$ de façon approchée.

On découpe l'intervalle $[a,b]$ en n intervalles de largeur constante $\dfrac{b-a}{n}$. Pour $i \in [\![0, n-1]\!]$, on note ces intervalles $[x_i, x_{i+1}]$, où $x_i = a + i \times h$.

Sur chaque intervalle $[x_i, x_{i+1}]$, la fonction f est approchée par une fonction g dont le calcul de l'intégrale est plus simple.À votre programme, vous avez la méthode des rectangles, pour laquelle on approxime la fonction f sur chaque intervalle $[x_i, x_{i+1}]$ par une fonction constante :

- méthode des rectangles à gauche : f est approchée par sa valeur prise à la borne inférieure de l'intervalle.
- méthode des rectangles à droite : f est approchée par sa valeur prise à la borne supérieure de l'intervalle.
- méthode des rectangles au milieu : f est approchée par sa valeur prise au milieu de l'intervalle.



L'intégrale est alors approchée par la somme des aires de chaque rectangle, de largeur $h = \dfrac{b-a}{n}$.

- Pour la méthode des rectangles à gauche : $I \approx \sum_{i=0}^{n-1} \Big(f(x_i) \times h\Big)$, soit $\boxed{I \approx h \times \sum_{i=0}^{n-1} f(x_i)}$, avec $x_i = a + i \times h$.
- Pour la méthode des rectangles à droite : $I \approx \sum_{i=0}^{n-1} \Big(f(x_{i+1}) \times h\Big)$, soit $\boxed{I \approx h \times \sum_{i=0}^{n-1} f(x_{i+1}) = h \times \sum_{i=1}^{n} f(x_i)}$, avec $x_i = a + i \times h$.

Comment choisir n ?

- Le résultat est d'autant plus précis que n choisi est grand.
- Cependant de par la représentation des réels par des flottants en machine (en python : sur 64 bits), deux réels « trop proches » sont codés de la même façon (sur 64 bits, le plus petit écart entre deux réels codés distinctement est de 2×10^{-16}.
- La méthode des rectangles est de complexité linéaire en n, le temps de calcul reste donc raisonnable.

Attention

Selon les énoncés, n peut représenter :

— le nombre d'intervalles, alors le pas vaut $h = (b-a)/n$, et il y a $n+1$ points. La somme va alors de 0 à $n-1$.

— le nombre de points, alors le pas vaut $h = (b-a)/(n-1)$, et il y a $n-1$ intervalles. La somme alors de 0 à $n-2$.

II Traduction en python

Il s'agit de traduire le calcul de la somme.

- Pour la méthode des rectangles à gauche, il faut calculer la somme $\boxed{I \approx h \times \sum_{i=0}^{n-1} f(x_i)}$, avec $x_i = a + i \times h$.

```
def rectangle_gauche(f,a,b,n):
    """
    renvoie la valeur approchée de l'intégrale de la fonction f entre a et b, selon la méthode des rectangles à gauche
    n est le nombre d'intervalles (= de rectangles)
    """
    h=(b-a)/n # pas
    S=0 # somme
    for i in range(n): # pour i allant de 0 à n-1
        S = S + f(a+i*h)
    return S*h
```

- Pour la méthode des rectangles à droite, il faut calculer la somme $\boxed{I \approx h \times \sum_{i=0}^{n-1} f(x_{i+1}) = h \times \sum_{i=1}^{n} f(x_i)}$, avec $x_i = a + i \times h$.

```
def rectangle_droite(f,a,b,n):
    """
    renvoie la valeur approchée de l'intégrale de la fonction f entre a et b, selon la méthode des rectangles à droite
    n est le nombre d'intervalles (= de rectangles)
    """
    h=(b-a)/n # pas
    S=0 # somme
    for i in range(n): # pour i allant de 0 à n-1
        S = S + f(a+(i+1)*h)
    # ou for i in range(1,n+1): S=S+f(a+i*h)
    return S*h
```

Cinquième partie

Résolution numérique d'une équation différentielle par la méthode d'Euler

I Principe de la méthode

Méthode : Méthode d'Euler

La méthode d'Euler permet la résolution numérique approchée sur l'intervalle de temps $[t_0, t_f]$ d'une équation différentielle écrite sous la forme

$$\frac{\mathrm{d}X}{\mathrm{d}t} = f(X)$$

connaissant la condition initiale $X(t_0) = X_0$.

L'idée fondamentale est d'approximer la dérivée $\dfrac{\mathrm{d}X}{\mathrm{d}t}$ par son taux d'accroissement sur l'intervalle $[t, t+h]$:

$$\frac{\mathrm{d}X}{\mathrm{d}t} \approx \frac{X(t+h) - X(t)}{h}$$

Autrement dit, cela revient à effectuer un développement limité au premier ordre de $X(t+h)$:

$$X(t+h) \approx X(t) + \frac{\mathrm{d}X}{\mathrm{d}t} \times h$$

Soit, d'après l'équation différentielle

$$X(t+h) \approx X(t) + f(X) \times h$$

L'intervalle de résolution $[t_0, t_f]$ est découpé en n intervalles de largeur h.

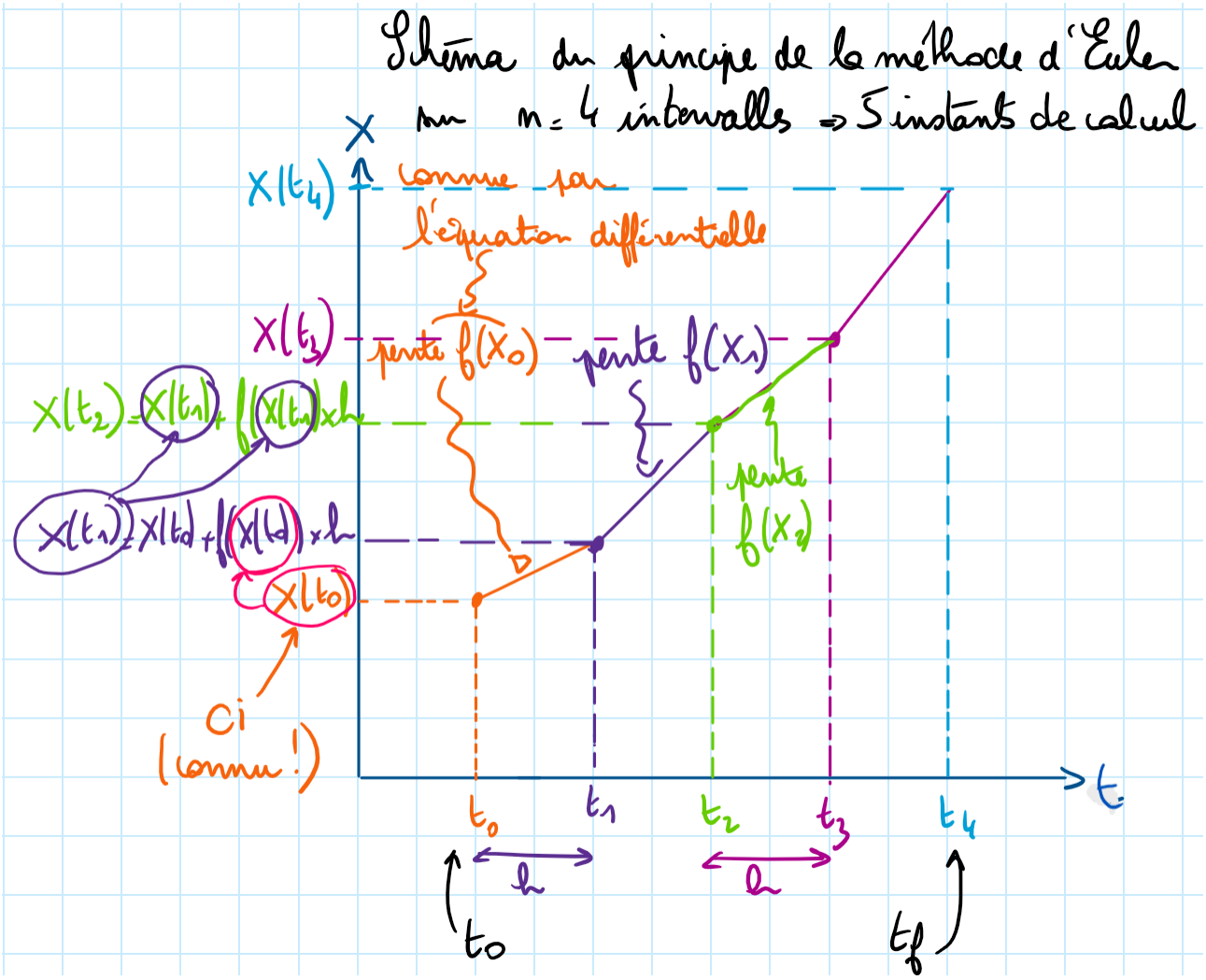
On cherche à déterminer les valeurs de X à chaque instant t_i, pour $i \in [\![1, n]\!]$, connaissant la valeur à l'instant précédent.

On exprime X à l'instant $t_{i+1} = t_i + h$ à l'aide de sa valeur à l'instant t_i :

$$X(t_{i+1}) = X(t_i) + f(X(t_i)) \times h$$

$$X_{i+1} = X_i + f(X_i) \times h$$

Connaissant la condition initiale $X(t_0)$, on peut alors déterminer $X(t_1) = X(t_0) + f(X(t_0)) \times h$, puis $X(t_2)$, ... Ainsi de proche en proche, on détermine les $n+1$ valeurs de X.



Attention

Selon les énoncés, n peut représenter :

- le nombre d'intervalles, alors le pas vaut $h = (b-a)/n$, et il y a $n+1$ points. La somme va alors de 0 à $n-1$.
- le nombre de points, alors le pas vaut $h = (b-a)/(n-1)$, et il y a $n-1$ intervalles. La somme alors de 0 à $n-2$.

II Traduction en python

Soit par exemple l'équation différentielle $\dfrac{\mathrm{d}v}{\mathrm{d}t} = g - \beta v^2$, où g et β sont des constantes définies au préalable, avec $v(0) = 0$.

```
import numpy as np
n= # à définir
t0= # à définir
tf= # à définir
h=(tf-t0)/n
# Liste des n+1 temps
t=[t0+i*h for i in range(0,n+1)]
# liste des vitesses
V=np.zeros(n+1)
V[0]=v0
for i in range(0,n):
    V[i+1]=V[i]+(g-beta*V[i]**2)*h
```

Si on veut écrire une fonction `euler` qui prendra en argument :

— la fonction `f` qui définit l'équation différentielle ;

— les bornes `t0` et `tf` de résolution ;

— le nombre `n` de pas de calcul ;

— la condition initiale `X0`

et renverra la tableau `X` des `n+1` valeurs approchées déterminées.

```
import numpy as np
def Euler(f,X0,t0,tf,n):
    """
    f : fonction définissant l'équation différentielle
    X0 : condition initiale
    t0 : instant initial et tf : instant final
    n : nombre de pas de calcul : h=(tf-t0)/n
    """
    h= (tf-t0)/n    #  pas de calcul
    t=np.linspace(t0,tf,n+1) #  liste des temps (n+1 instants de calculs) répartis régulièrement entre t0 et tf
    X=np.zeros(n+1) # tableau qui contient n+1 zéros au départ
    X[0]=X0      #  le 1er élément du tableau est la condition initiale
    for i in range(0,n): # il reste n instants à calculer,  calcul du point suivant (élément i+1) à partir du pt précédent (élément i) :
        X[i+1]=X[i]+f(t[i],X[i])*h
    return X #la fonction renvoie le tableau X des valeurs de X successives
```

III Utilisation des fonctions de python

Plusieurs fonctions sont déjà définies dans Python pour résoudre numériquement des équations différentielles.

On va utiliser ici la fonction `solve_ivp` disponible dans la bibliothèque `scipy.integrate` qui permet de résoudre les équations différentielles sous la forme

$$\begin{cases} \dfrac{\mathrm{d}y}{\mathrm{d}t} &= f(t,y) \\ y(0) &= y_0 \end{cases}$$

où y est un vecteur de taille N et f une fonction de $\mathbb{R}^N$ dans $\mathbb{R}^N$.

La syntaxe d'utilisation de la fonction `solve_ivp` est la suivante :

```
solve_ivp(fun, t_span, y0, t_eval)
```

— `fun` : la fonction $f(t,y)$;

— `t_span` : l'intervalle de temps de résolution (t_0, t_f), tuple de 2 flottants ;

— `y0` : le vecteur de condition initiale y_0 ;

— `t_eval` : le tableau des instants de résolution.

Par exemple pour résoudre $\dfrac{\mathrm{d}v}{\mathrm{d}t} = g - \beta v^2$, avec $v(0) = 0$.

```
def equa_diff(v):
    return g-beta*v**2
resol=sci.solve_ivp(equa_diff,(0,tf),np.array([0]),t_eval=np.linspace(0,tf,1000)) # tf à définir : choisir une valeur !
temps=resol.t # on récupère les instants de résolutions
# resol.y est le tableau des valeurs de y, chaque colonne correspondant à un instant de résolution.
vitesse=resol.y[0] # récupération de l'unique ligne de resol.y : v
```

IV Adaptation pour une équation différentielle du 2e ordre

Par exemple, l'équation différentielle du pendule pesant

$$\frac{\mathrm{d}^2\theta}{\mathrm{d}t^2} + \omega_0^2 \sin(\theta) = 0$$

avec $\theta(0)$ et $\dot{\theta}(0)$, ne peut pas être résolue analytiquement. Sa résolution nécessite une méthode numérique. L'équation du pendule pesant est une équation du deuxième ordre, qui ne peut pas être résolue directement en utilisant la méthode d'Euler, qui s'applique à une équation différentielle du 1er ordre.

Il est nécessaire de récrire l'équation différentielle du pendule pesant sous la forme d'une équation vectorielle du premier ordre.

Pour cela, on pose

$$X = \begin{pmatrix} \theta \\ \dfrac{\mathrm{d}\theta}{\mathrm{d}t} \end{pmatrix}$$

Alors

$$\begin{aligned} \frac{\mathrm{d}X}{\mathrm{d}t} &= \begin{pmatrix} \dfrac{\mathrm{d}\theta}{\mathrm{d}t} \\ \dfrac{\mathrm{d}^2\theta}{\mathrm{d}t^2} \end{pmatrix} \\ \frac{\mathrm{d}X}{\mathrm{d}t} &= \begin{pmatrix} \dfrac{\mathrm{d}\theta}{\mathrm{d}t} \\ -\omega_0^2\theta \end{pmatrix} \\ \frac{\mathrm{d}X}{\mathrm{d}t} &= f(t, X) \end{aligned}$$

avec $f\left(t, \begin{pmatrix} \theta \\ \dfrac{\mathrm{d}\theta}{\mathrm{d}t} \end{pmatrix}\right) = \begin{pmatrix} \dfrac{\mathrm{d}\theta}{\mathrm{d}t} \\ -\omega_0^2\theta \end{pmatrix}$

On définit alors la fonction

```
def equa_diff_pendule(t,X):
    dtheta=X[1]
    d2theta=-w0**2*np.sin(X[0])
    return np.array([dtheta,d2theta])
```

On peut alors utiliser les fonctions définies précédemment.

Par exemple pour résoudre le pendule pesant, avec $\theta(0) = 3\pi/4$ et $\dot{\theta}(0) = 0$. Après avoir définies les constantes du problème, et la durée t_f de résolution et le nombre n d'intervalles de résolution.

— Avec la fonction de python :

```
solution = sci.solve_ivp(equa_diff_pendule,(0,tf),np.array([3*pi/4,0]),
   t_eval=np.linspace(0,tf,n+1))
theta = solution.y[0] #  récupération de la liste des theta
dthetadt = solution.y[1] #  récupération de la liste des dtheta/dt
```

— Avec la fonction `euler`, qu'il faut au préalable légèrement adapter pour prendre en compte le fait que X_0 peut être un tableau à plusieurs dimensions :

```
def euler2(f,X0,t0,tf,n):
    """
    f : fonction définissant l'équation différentielle
    X0 : condition initiale : tableau numpy de 2 éléments
    t0 : instant initial et tf : instant final
    n : nombre de pas de calcul : h=(tf-t0)/n
    """
    h= (tf-t0)/n # pas de calcul
    t=np.linspace(t0,tf,n+1) # liste des temps (n+1 instants de calculs) répartis régulièrement entre t0 et tf
    X=np.zeros((len(X0),n+1)) # tableau à 2 dimensions, len(X0) lignes et n+1 colonnes
    X[:,0]=X0 # la colonne de rang 0 du tableau est la condition initiale
    for i in range(0,n): # il reste n instants à calculer
    # on calcule le point suivant (colonne i+1) à partir du pt précédent (colonne i) :
        X[:,i+1] = X[:,i] + h * f(t[i],X[i])
    return X #la fonction renvoie le tableau X à deux dimensions

solution = euler2(equa_diff_pendule,np.array([3*pi/4,0]),0,tf,n)
theta = solution[0] # la première ligne contient les valeurs de theta
dthetadt = solution[1] # la deuxième ligne contient les valeurs de dtheta/dt
```