Chapitre 9

Calcul d'intégrales - Approximation de solutions d'équations différentielles par la méthode d'Euler

Simon Dauguet
simon.dauguet@gmail.com

jeudi 9 avril, 2026

Table des matières

1 Calcul d'intégrale

1.1 Présentation

1.1.1 Méthode des rectangles

Dans le chapitre de l'intégration en mathématiques, la méthode des rectangles (*i.e.* les sommes de Riemann) ont été présentées. On rappelle :

Proposition 1.1 (Convergence des sommes de Riemann) :
Soit $f \in \mathcal{C}^0([a,b],\mathbb{R})$. Alors

$$\frac{b-a}{n}\sum_{k=0}^{n-1} f\left(a+k\frac{b-a}{n}\right) \xrightarrow[n\to+\infty]{} \int_a^b f(t)dt.$$

et

$$\frac{b-a}{n}\sum_{k=1}^{n} f\left(a+k\frac{b-a}{n}\right) \xrightarrow[n\to+\infty]{} \int_a^b f(t)dt.$$

Autrement dit, la suite des sommes de Riemann sont des approximations de l'intégrale de la fonction f sur le segment $[a,b]$.

Comme on approxime l'intégrale par des aires de rectangles, on appelle cette méthode d'approximation, la méthode des rectangle.

En fait, sur chacun des petits intervalles de la subdivision du segment $[a,b]$, on approxime f par des constantes, donc par des polynômes de degré 0. Après intégration, ça donne des polynômes de degré 1. Il est possible de généraliser ce résultat.

Remarque :
La démo de cette convergence sera faite en détail dans le chapitre d'intégration en maths.

Définition 1.1 (Méthode des rectangles d'approximation d'une intégrale) :
Soit $f \in \mathcal{C}^0([a,b],\mathbb{R})$.
L'approximation de $\int_a^b f(t)dt$ par la méthode des rectangles consiste à calculer l'un des termes d'une des deux sommes de Riemann associée à f, donc de renvoyer $\frac{b-a}{n}\sum_{k=0}^{n-1} f(a_k)$.

```python
def intRectGch(f:"function", a:float, b:float, n:int) -> float :
    R=0
    for k in range(0,n) :
        R=R+f(a+k*(b-a)/n)
    return((b-a)/n*R)

def intRectDrt(f:"function", a:float, b:float, n:int) -> float :
    R=0
    for k in range(0,n) :
        R=R+f(a+(k+1)*(b-a)/n)
    return((b-a)/n*R)
```

Terminaison

Ces deux algorithmes ne dépendent que d'une boucle `for`. La terminaison est donc assurée (la valeur de `n-k` est le variant de boucle).

Correction

Le variant de boucle est $\forall k \in \{0, \dots, k\}$, $R_k = \sum_{i=0}^{k} f\left(a + i\frac{b-a}{n}\right)$. Et le fait que ça corresponde bien à une approximation de $\int_a^b f(t)dt$ (donc de la convergence vers l'intégrale de la suite fabriquée) est donnée par la preuve du théorème de convergence des sommes de Riemann.

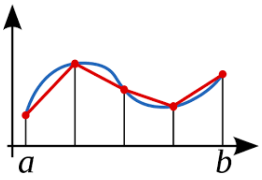
Complexité

La complexité de cet algorithme est évidemment soumis à la complexité de celle de l'évaluation de f. Mais si on note $C(f)$ la complexité de l'évaluation de f, alors la complexité est en $O(nC(f))$.

1.1.2 Méthode des trapèzes

Définition-Propriété 1.2 (Méthode des trapèzes) :
On peut approximer également l'intégrale de f par la sommes des aires des trapèzes sous la courbe de f sur les intervalles d'une subdivision :

$$\frac{b-a}{2n}\sum_{k=0}^{n-1}(f(a_k)+f(a_{k+1})) \xrightarrow[n\to+\infty]{} \int_a^b f(t)dt.$$




```python
def intTrapeze(f:"function", a:float, b:float, n:int) -> float :
    T=0
    for k in range(0,n) :
        T=T+f(a+k*(b-a)/n)+f(a+(k+1)*(b-a)/n)
    return((b-a)/(2*n)*T)
```

Démonstration (Sketch) *:*
La fonction affine correspondante sur chacun des intervalle $[a_k, a_{k+1}]$ est alors

$$g_k : x \mapsto \frac{f(a_{k+1}) - f(a_k)}{a_{k+1} - a_k}(x - a_k) + f(a_k)$$

L'aire du trapèze définie par ces fonctions affines, *i.e.* l'aire sous la courbe de cette droite, est alors

$$\int_{a_k}^{a_{k+1}} g_k(t)dt = \frac{b-a}{n}\frac{f(a_k)+f(a_{k+1})}{2}$$

On approche alors l'intégrale théorique I par la somme des aires des trapèze :

$$T_n = \frac{b-a}{2n}\sum_{k=0}^{n-1}(f(a_k)+f(a_{k+1}))$$

□

Remarque :
Il est alors facile de voir que

$$T_n = \frac{1}{2}(R_n^- + R_n^+)$$

Comme pour la méthode des rectangles, plus le nombre de points est grand, plus le pas est petit, plus la subdivision est fine, et plus l'approximation sera bonne.

Terminaison

De même, l'algorithme ne dépend que d'une boucle `for`, assurant la terminaison.

Correction

De même que pour la méthode des rectangles, c'est la démonstration de la convergence de la proposition précédente qui l'assure. Et cette démo est une généralisation classique de la démo de la convergence des sommes de Riemann (voir maths).

Complexité

La complexité n'est pas plus difficile à calculer que la méthode des rectangles. Si $C(f)$ est la complexité de l'évaluation de f, alors cet algorithme a une complexité en $O(nC(f))$. Donc essentiellement la même que la complexité de la méthode des rectangles. Attention, dans la version proposée, ce n'est pas forcément évident à voir à cause des boucles sous-entendues, notamment dans la fonction `sum`. Le plus sûr reste de tout explicité pour pouvoir faire le calcul correctement.

Comme pour la méthode de Riemann, plus le nombre de points est grand, plus l'approximation est bonne, mais plus le nombre de calculs à faire est grand et donc le temps d'évaluation est long. Il s'agit donc de trouver un compromis entre la lenteur des calculs (qui dépend du nombre de points mais AUSSI de f) et la précision souhaitée pour le calcul de l'intégrale.

1.1.3 Généralisations

On peut généraliser le processus facilement. En effet, dans la méthode des rectangles, on a approximé f par une constante sur chacun des petits intervalles de la subdivision. Dans la méthode des rectangles, on a approximé f par des fonctions affines. Et en fait, on peut approximer f par un polynôme de n'importe quel degré (tant que le degré est le même pour tous les petits intervalles).

Pour pouvoir approximer f facilement par des polynômes, il suffit d'utiliser les polynômes interpolateurs de Lagrange. Mettons que nous voulions approximer f par des polynômes de degré p sur chacun des intervalles de la subdivision. Il suffit alors de redécouper chaque intervalles pour y mettre $p+1$ points régulièrement répartis. Puis on peut établir les polynômes passant par ces points avec les polynômes interpolateur de Lagrange.

En approximant f par des polynômes de degré p, on aurait alors une vitesse de convergence de l'ordre de $1/n^{p+1}$. Autrement dit, si I est l'intégrale théorique de f et I_n le terme obtenu par cette méthode avec une subdivision de l'intervalle en n petits intervalles, alors on aurait

$$I_n - I \underset{n\to+\infty}{=} O(1/n^{p+1}).$$

En particulier, la méthode des rectangles fait des approximations de la fonction par des constantes (donc des polynômes constants) et donc aura une vitesse de convergence en $1/n$. Ce qui n'est pas très rapide. Et la méthode des trapèze (qui fait des approximations de la fonction par des fonctions affines, donc des polynômes de degré 1) a une vitesse de convergence en $1/n^2$. Ce qui est nettement mieux. Mais cette vitesse de convergence a un prix : la complexification de l'expression utilisée.

Bien entendu, plus le degré des polynômes augmente, meilleure est la vitesse de convergence (et donc moins on a besoin d'aller loin dans la suite). Mais plus le degré est grand, plus l'expression des polynômes sera compliqué (il faut utiliser les polynômes interpolateur de Lagrange). Et donc plus le nombres de calculs à faire sera grand pour un seul petit intervalle de la subdivision. Ce qui n'arrange pas tellement la complexité.

1.2 Discussion

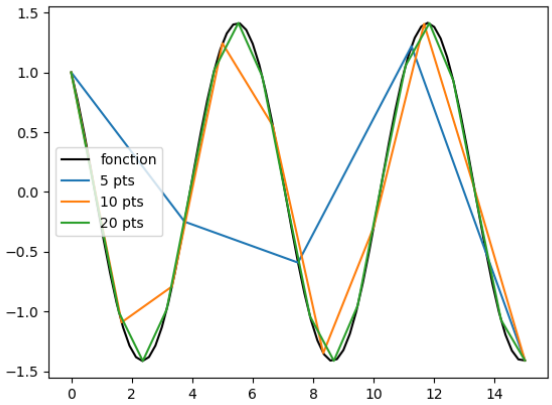
On rappel que python a une méthode un peu particulière de coder les nombres. Il ne fait que des approximations, ce qui peut mener à des incohérence. Notamment, l'addition n'est pas associative, et le produit n'est pas distributif sur l'addition.

Par exemple, sur la fonction précédente **`intRectGch`** appelée sur la fonction $t \mapsto t$ sur $[0, 1]$ renverra `0.494999...`, si le code traduit $\sum_{i=0}^{n-1} hf(t_i)$; ou `0.495`, si le code traduit $h\sum_{i=0}^{n-1} f(t_i)$. Les erreurs ne sont pas les mêmes !

Avec la méthode des trapèzes, avec $n = 100$ et $h\left(\frac{f(a)+f(b)}{2} + \sum_{i=1}^{n-1} f(t_i)\right)$, on obtient la valeur exacte `0.5`.

Bien sûr, le nombre de points joue un rôle très important également dans la qualité de l'approximation. En particulier, le choix de ces points, selon le caractère "accidenté" du graphe de la fonction pourra donner lieu à des incohérences.

Mais plus le nombres de points choisis est grand, meilleure sera l'approximation (théoriquement) et plus lent sera le calcul de l'algorithme.



1.3 Méthode pré-implémentée

Définition 1.3 (Module `scipy.integrate`) :
Le module **`scipy.intergrate`** est le module de Python contenant les manipulations sur les intégrales et même les résolutions d'équations différentielles (voir plus bas).

Définition 1.4 (Fonction `quad` (`scipy.intergrate`)) :
La fonction **`quad`** dans le module **`scipy.integrate`**, permet de calculer une valeur d'une intégrale d'une fonction. Sa syntaxe est :

```
>>> scipy.integrate.quad(f:"function", a:"float", b:float) -> tuple
```

Elle renvoie un couple de deux valeurs (I, ε) : I est une valeur approchée de $\int_a^b f(t)dt$ et ε est une estimation de l'erreur faite lors du calcul par rapport à la valeur théorique de l'intégrale (que python ne peut pas connaître).

Exemple 1.1 :

```
>>> import scipy.integrate as scint      # importation avec un alias du module de calcul intégral
>>> scint.quad(lambda x : x , 0, 1)      # calcul de ∫_0^1 x dx
(0.5, 5.551115123125783e-15)             # valeur approchée de l'intégrale et estimation de l'erreur
>>> import math as mt                    # module math
>>> scint.quad(lambda x : mt.cos(x), 0, mt.pi)   # calcul de ∫_0^π cos(x)dx
(4.9225526349740854e-17, 2.2102239425853306e-14)     # valeur approchée de l'intégrale et estimation de l'erreur commise
```

Remarque :
Théoriquement, le module **`scipy.integrate`** est hors programme et ne doit pas être utilisé. Mais il est pratique d'avoir "un filet de sécurité" et donc de pouvoir comparer les résultats obtenus par rapport à une valeur théorique (que l'on ne peut, en général, pas calculé). Les méthodes de calculs numériques sont des outils d'évaluations de valeurs approchées de solutions d'équations qui ne peuvent pas être résolues avec les seuls outils théoriques (comprendre : par des outils mathématiques). Ce qui est attendu, c'est évidemment la "reconstruction à la main" de ces solutions avec les méthodes présentées dans ce cours.

2 Résolution approchée d'équation différentielle - Méthode d'Euler

2.1 Généralités

On va commencer par étudier les équations différentielles d'ordre 1, puis on généralisera l'étude aux équations différentielles d'ordre plus grand.

Une équation différentielle d'ordre 1 générale, est une équation de la forme $y'(t) = F(t, y(t))$ pour tout t sur un intervalle. La théorie mathématique nous permet d'avoir l'assurance de l'existence de solutions sous certaines conditions (ce sont les conditions de Cauchy-Lipschitz qui demande essentiellement à ce que la fonction F soit Lipschitzienne par rapport à sa deuxième variable). Mais ces théorèmes d'existences sont théoriques ne permettent pas d'avoir des solutions ou d'en construire.

La méthode d'Euler s'inscrit dans le cas où l'existence de solutions est assurée et permet d'avoir une méthode constructiviste pour approcher numériquement une solution de l'équation différentielle. En particulier, la connaissance d'une condition initiale est déterminante dans la méthode d'Euler.

Le principe est le suivant. On considère une équation différentielle $y' = F(t, y)$ sur un intervalle $[a, b]$ où l'on sait qu'il existe des solutions d'après la théorie mathématique. On se donne alors une condition initiale (a, y_0). Et on va partir de cette valeur pour donner une approximation de l'unique solution au problème de Cauchy liée à cette valeur initiale.

On va appeler y l'approximation de la solution du problème de Cauchy. On sait donc que $y(a) = y_0$. Et nous allons déterminer des valeurs de y (donc des valeurs approchées de la solution théorique) en des points successifs. Passons directement au cas général de la construction.

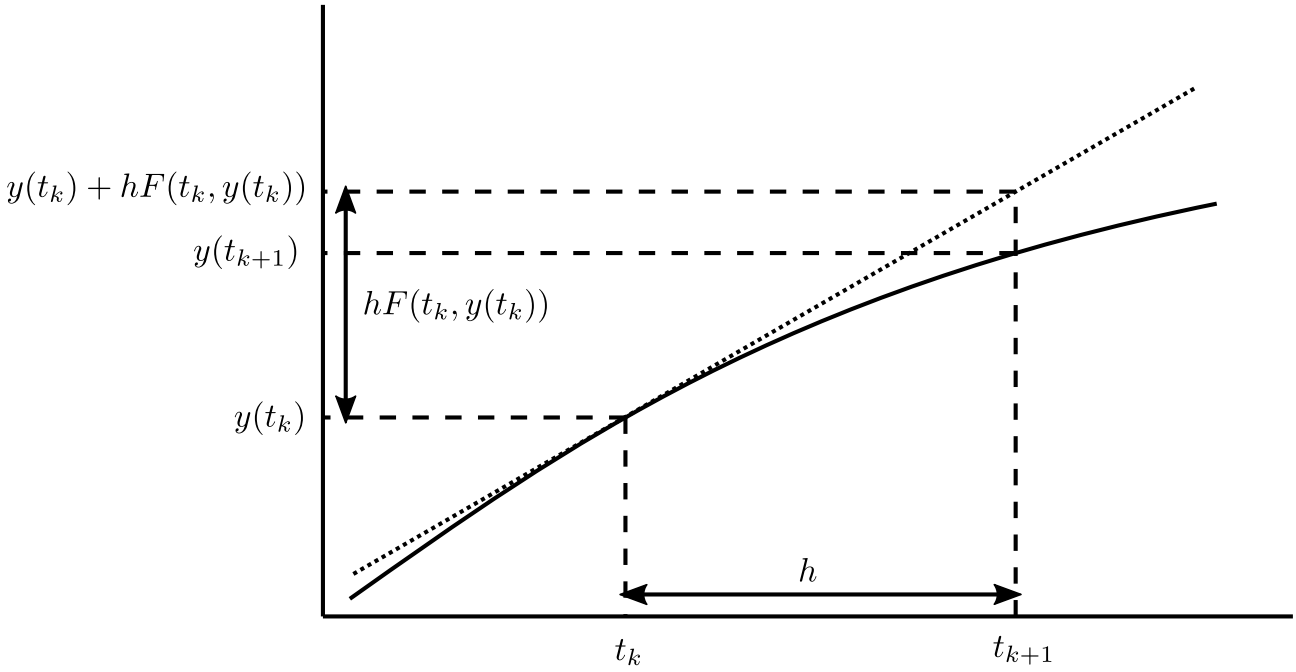
Supposons qu'il existe un point $t_k \in [a, b]$ en lequel on connaisse y (donc une valeur approchée de la solution). Prenons un point un peu plus $t_{k+1} = t_k + h$. Alors, si h est petit, on peut utiliser l'approximation classique d'une fonction par sa tangente (Taylor-Young) ce qui nous donne

$$\begin{aligned} y(t_{k+1}) &= y(t_k + h) \\ &\underset{h\to 0}{=} y(t_k) + hy'(t_k) + o(h) \\ &\underset{h\to 0}{=} y(t_k) + hF(t_k, y(t_k)) + o(h) \end{aligned}$$

En prenant alors h *suffisamment petit*, on pourra alors faire l'approximation

$$y(t_{k+1}) \approx y(t_k) + hF(t_k, y(t_k))$$

et donc, de proche en proche, si les points sont *suffisamment proches* les uns des autres, on peut avoir une approximation des valeurs successives de la fonction.



Puis on recommence le processus en t_{k+1}.

Définition 2.1 (Schéma numérique) :
Un schéma numérique de résolution d'une équation différentielle correspond à la liste des points obtenus par discrétisation de l'équation différentielle.

Dans le cas de la résolution d'une équation différentielle par la méthode d'Euler, le schéma numérique correspondant est donc :

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

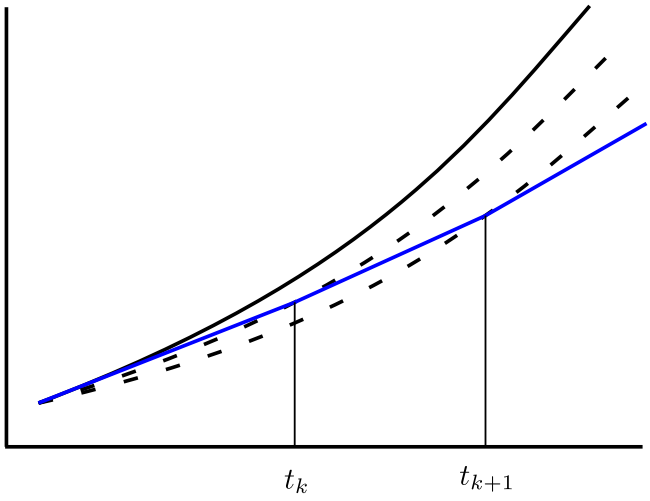
L'expression du schéma numérique de résolution dépend de la méthode de discrétisation utilisée.

Remarque :
Cette méthode de construction ne donne pas, à proprement parlé, de solutions de l'équation différentielle. On a que des approximations en des points successifs de la solution. Mais d'un point de vue numérique, avoir une fonction, c'est en fait se donner suffisamment de points par lesquels passe la courbe.

!!! ATTENTION !!!

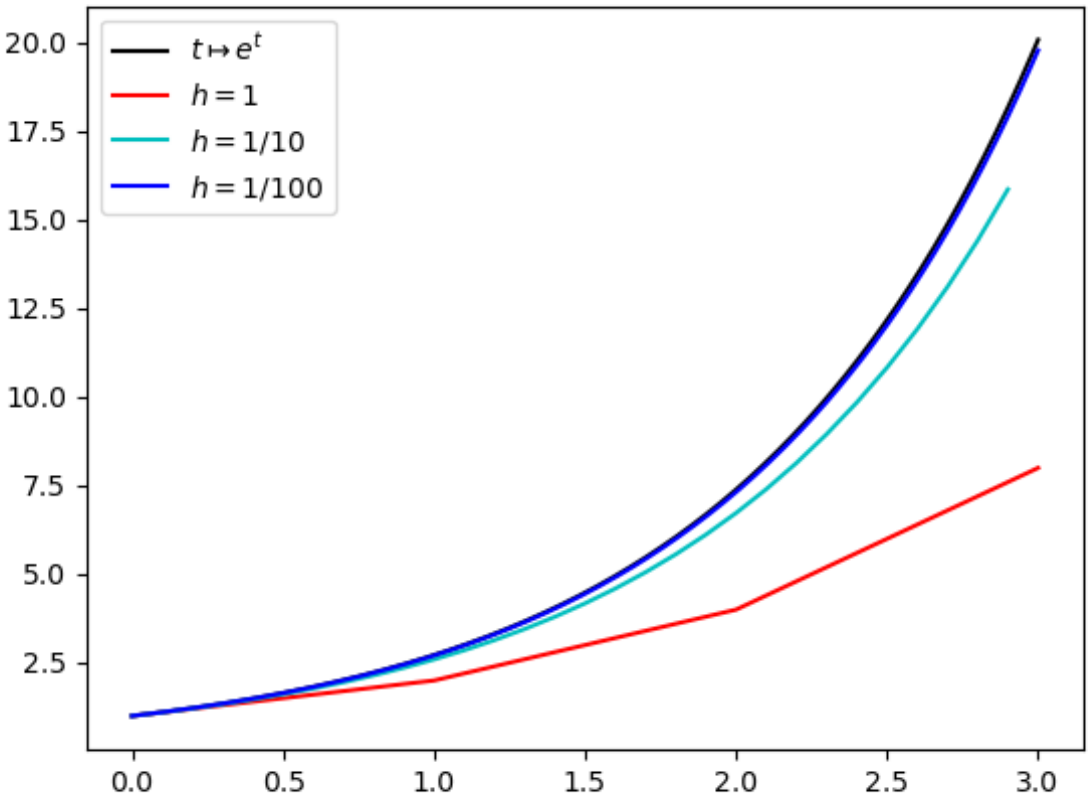
⚠ Il évident que l'étape d'après, en refaisant le processus sur $y(t_{k+1})$ qui est une approximation, on va faire une nouvelle approximation, ce qui va ajouter à l'erreur déjà commise. Donc plus on sera loin du point de départ, plus l'écart avec la solution théorique sera grand.

Ce qui nécessitera de bien choisir le pas h des différents points pour trouver un compromis entre complexité, temps de calculs, et "bonne" approximation.



Remarque :
La méthode d'Euler n'est pas la seule méthode permettant d'approcher des solutions d'une équation différentielle. Mais c'est l'un des plus abordables. Il existe aussi la méthode de Heun ou de Runge-Kutta.

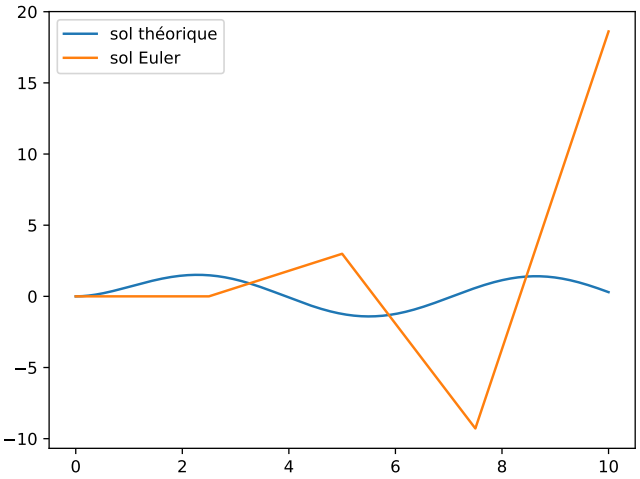
Il est évident que les erreurs faites iront dans le même sens que le pas : plus le pas est grand, moins bonne sera l'approximation.



!!! ATTENTION !!!

⚠ Le choix du pas a une grande importance. Il vaut mieux avoir une petite idée de l'allure de la solution pour pouvoir choisir le pas en conséquence. Par exemple, dans le cas où une solution serait périodique et qu'on choisirait un pas de la longueur de la période, on aboutirait à une solution constante.

Contre-exemple : Importance du pas sur la cohérence du résultat
On considère l'équation différentielle linéaire d'ordre $y' + y = 2\sin(t)$. La solution au problème de Cauchy avec la condition initiale $y(0) = 0$ est alors la fonction $y : t \mapsto e^{-t} - \cos(t) + \sin(t)$. qui va être essentiellement périodique sur $\mathbb{R}_+$. En choisissant mal le nombre de points (donc le pas), on peut obtenir des approximations très loin de la réalité :



Les erreurs commises à chaque étapes de calculs s'additionne. Et donc, "loin" du point de départ, les approximations finissent par être éloignées de la valeur théorique.

2.2 Équation différentielle d'ordre 1

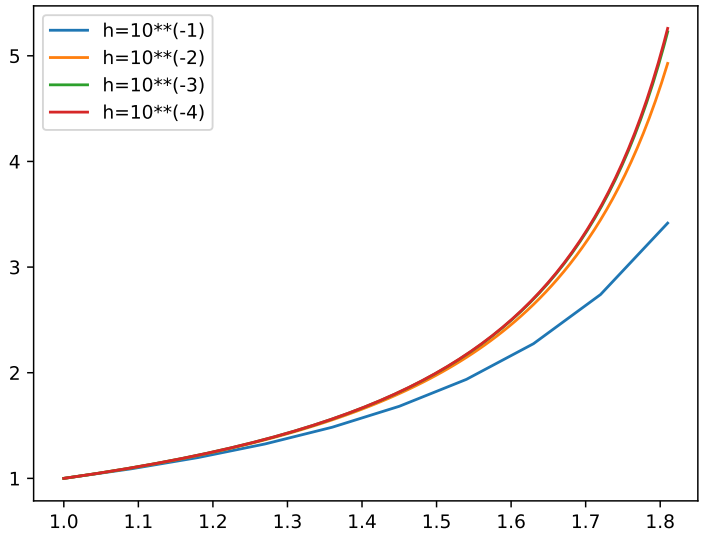
Nous allons présenter ici une version de la méthode d'Euler pour les équations différentielles d'ordre 1 qui va prendre en compte le nombre de points. On rappelle qu'il y a un lien entre le nombre de points et le pas : $h = \frac{b-a}{n}$ où n est le nombre de points de la subdivision $(t_0 = a, \ldots, t_n = b)$.

```
1  def Euler(F:"function", a:float, b:float, y0:float, n:int) -> tuple :
2      """Algorithme de la méthode d'Euler pour l'ordre 1
3      F - function : Fonction de l'équation différentielle y'=F(t,y)
4      a,b - float : intervalle [a,b] de résolution
5      y0 - float : valeur initiale de y(a)
6      n - int : nombre de points de la subdivision de [a,b] """
7      y=y0
8      h=(b-a)/(n-1) # Le pas correspondant
9      X=[a+k*h for k in range(0,n)] # Subdivision de [a,b] avec n points
10     Y=[] # Listes des valeurs de l'approximation
11     for x in X :
12         Y.append(y)
13         y=y+h*F(x,y)     # Valeur suivante
14     return(X,Y)
```

Exemple 2.1 (Effet du pas sur l'écart) :
On va s'intéresser à l'équation différentielle $y' = y^2$ et la condition initiale $y(1) = 2$ et on va faire varier le pas pour observer les différences sur les solutions :

```
>>> for k in range(1,5) :
    L = Euler(lambda t,y : y**2, 1, 1.81, 1, 10**k)
    plt.plot(L[0],L[1],label=f"h=10**(-{k})")


[<matplotlib.lines.Line2D object at 0x75ccb45002e0>]
[<matplotlib.lines.Line2D object at 0x75ccb4500580>]
[<matplotlib.lines.Line2D object at 0x75ccb45016c0>]
[<matplotlib.lines.Line2D object at 0x75ccb4502a10>]
>>> plt.legend()
<matplotlib.legend.Legend object at 0x75ccce853100>
>>> plt.show()
```



2.3 Équation différentielle d'ordre 2

Une équation différentielle d'ordre 2 est une équation de la forme $y'' = F(t, y(t), y'(t))$. On peut alors se ramener à une équation différentielle vectorielle d'ordre 1 et donc, au prix d'une petite adaptation, à la méthode d'Euler précédente. Précisément, on pose la matrice $Y = \begin{pmatrix} y \\ y' \end{pmatrix}$ qui est une matrice dont les

coefficients sont des fonctions. Donc Y est une fonction d'une variable réelle et elle est dérivable. Et alors

$$\begin{aligned} Y' &= \begin{pmatrix} y' \\ y'' \end{pmatrix} \\ &= \begin{pmatrix} y' \\ F(t, y(t), y'(t)) \end{pmatrix} \\ &= G(t, Y(t)) \end{aligned}$$

en posant

$$G\left(t, \begin{pmatrix} a \\ b \end{pmatrix}\right) = \begin{pmatrix} b \\ F(t, a, b) \end{pmatrix}$$

Alors Y vérifie une équation différentielle d'ordre 1. Et on peut alors appliquer la méthode d'Euler.

On notera que l'algorithme de la méthode d'Euler présenté précédemment fonctionne très bien même si F est une fonction vectorielle.

!!! ATTENTION !!!

⚠ Bien prendre garde aux opérations ! La somme vectorielle mathématique ne correspond pas au symbole $+$ sous python. Pour python, `[1,2]+[3,4]` correspond à une concaténation et donne donc `[1,2,3,4]` qui n'est pas vraiment ce que l'on veut faire. Il faudra donc adapter un peu les algorithmes pour s'assurer d'avoir les bonnes opérations.

```
1  def Euler2Var(F:"function", a:float, b:float, Y0:list, n:int) -> tuple :
2      """Algorithme de la méthode d'Euler vectoriel.
3      y0 : list -  vecteur colonne de la forme [[y1],[y2]]
4      F : function - Fonction a deux variables définissant l'équa diff Y'=F(t,Y)
5      a,b : float - Intervalle [a,b] de résolution
6      Y0 : list - Vecteur initiale correspond à Y(a)
7      n : int - Nombres de points de la subdivision
8      """
9      y=Y0
10     h = (b-a)/(n-1)      # Pas de la subdivision en n points
11     X=[a+k*h for k in range(n)] # Subdivision de [a,b] en n points
12     Y=[[0]*n for k in range(len(Y0))] # Liste des valeurs de Y (autant de lignes que
   Y0)
13     for k in range(0,n) :
14         Y[0][k] = y[0][0]
15         Y[1][k] = y[1][0]
16         tab = F(X[k],y)
17         y = [[y[0][0] + h*tab[0][0]], [y[1][0]+h*tab[1][0]]]
18     return(X,Y)
```

Exemple 2.2 (Équation différentielle d'ordre 2 linéaire) :
On va commencer par un cas simple : on va essayer de trouver une solution de l'équation $y'' = -y$ avec la condition initiale $y(0) = 0$ sur l'intervalle $[0, 2\pi]$. La solution est connue, c'est le sinus. Si on pose $Y = \begin{pmatrix} y \\ y' \end{pmatrix}$, alors

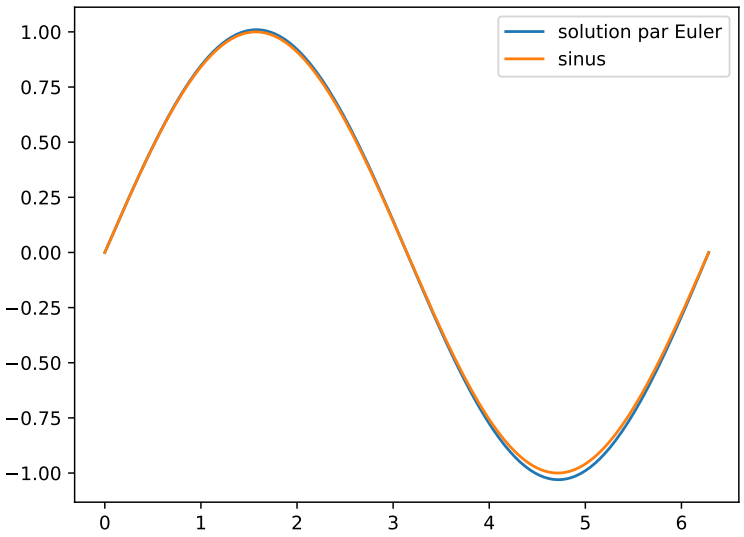
$$Y' = \begin{pmatrix} y' \\ y'' \end{pmatrix} = \begin{pmatrix} y' \\ -y \end{pmatrix} = F(t, Y)$$

avec la fonction

$$F\left(t, \begin{pmatrix} a \\ b \end{pmatrix}\right) = \begin{pmatrix} b \\ -a \end{pmatrix}$$

On a alors le code :

```
>>> L=Euler2Var(lambda t,Y : [Y[1],[-Y[0][0]]], 0, 2*mt.pi, [[0],[1]],501)
>>> plt.plot(L[0],L[1][0],label="solution par Euler")
[<matplotlib.lines.Line2D object at 0x7f5f53eda280>]
>>> plt.plot(L[0],[mt.sin(L[0][k]) for k in range(len(L[0]))], label="sinus")
[<matplotlib.lines.Line2D object at 0x7f5f53eda220>]
>>> plt.legend()
<matplotlib.legend.Legend object at 0x7f5f53eda8e0>
>>> plt.show()
```



Exemple 2.3 (Équation différentielle d'ordre 2 non linéaire) :
L'équation d'un pendule simple est $\ddot{\theta} = -k\sin(\theta)$. Essayons de trouver une solution de cette équation. Pour simplifier, on prendre un coefficient $k = 1$. On pose alors $Y = \begin{pmatrix} y \\ y' \end{pmatrix}$ et on obtient

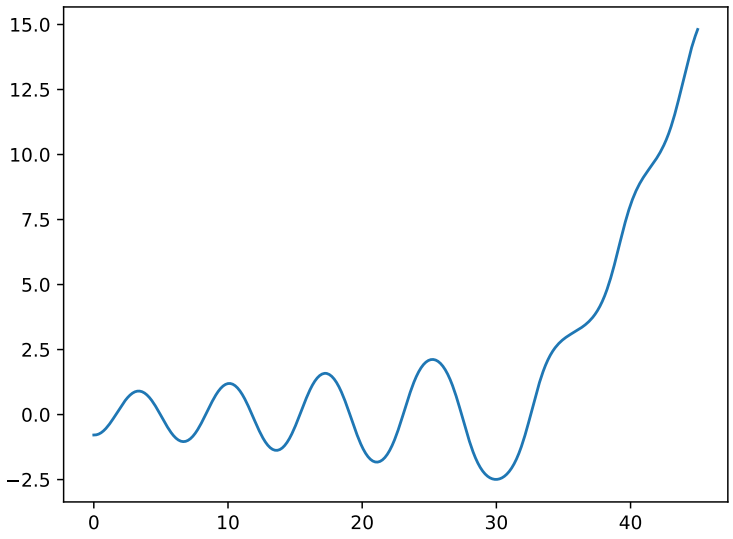
$$Y' = \begin{pmatrix} y' \\ y'' \end{pmatrix} = \begin{pmatrix} y' \\ -\sin(y) \end{pmatrix}$$

donc on considère la fonction

$$F\left(t, \begin{pmatrix} a \\ b \end{pmatrix}\right) = \begin{pmatrix} b \\ -\sin(a) \end{pmatrix}$$

Et d'où le code :

```
>>> T=Euler2Var(lambda t,Y : [Y[1],[-mt.sin(Y[0][0])]], 0, 45, [[-mt.pi/4],[0]], 501)
>>> plt.plot(T[0],T[1][0])
[<matplotlib.lines.Line2D object at 0x7fbca68153d0>]
>>> plt.show()
```



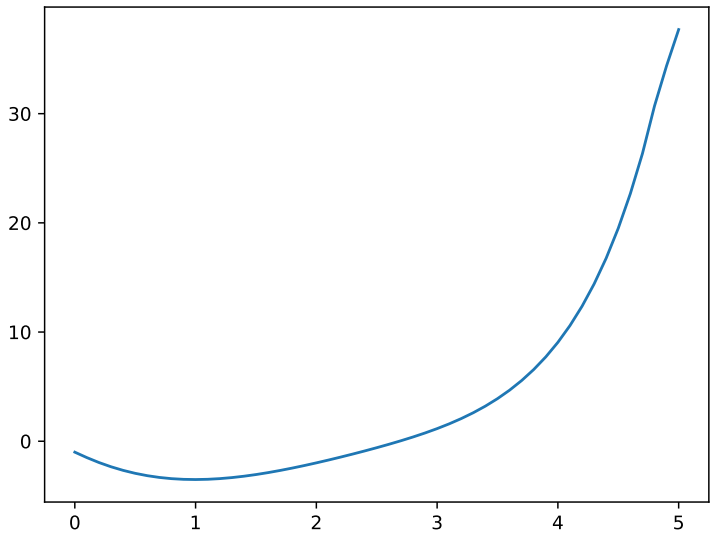
Avec des équations différentielles d'ordre plus grand, la méthode s'adapte très bien. Il suffit de mettre plus de lignes.

Exemple 2.4 (Équation différentielle d'ordre 3) :
On considère l'équation différentielle $y''' = (y + y')^2 - y''^2$. On utilise la fonction

$$F\left(t, \begin{pmatrix} a \\ b \\ c \end{pmatrix}\right) = \begin{pmatrix} b \\ c \\ (a+b)^2 - c^2 \end{pmatrix}$$

et donc

```
def Euler3Var(F:"function", a:float, b:float, Y0:list, n:int) -> tuple :
    y=Y0
    h = (b-a)/(n-1)
    X = [a+k*h for k in range(n)]
    Y=[[0]*n for k in range(len(Y0))]
    for k in range(0,n) :
        Y[0][k] = y[0][0]
        Y[1][k] = y[1][0]
        Y[2][k] = y[2][0]
        tab = F(X[k],y)
        y[0] = [y[0][0] + h*tab[0][0]]
        y[1] = [y[1][0] + h*tab[1][0]]
        y[2] = [y[2][0] + h*tab[2][0]]
    return(X,Y)
>>> E = Euler3Var(lambda t,Y : [Y[1],Y[2],[(Y[0][0]+Y[1][0])**2-Y[2][0]**2]], 0, 5,
                                                                    [[-1],[-5],[5]], 51)
>>> plt.plot(E[0],E[1][0])
>>> plt.show()
```



2.4 Méthode pré-implémenté

Définition 2.2 (Fonction `odeint` (module `scipy.integrate`)) :
La bibliothèque **`scipy.integrate`** contient la fonction **`odeint`** qui permet de résoudre numériquement une équation différentielle de la forme $y' = T(t, y(t))$ sur un intervalle $[a, b]$. La syntaxe est

```
scipy.integrate.odeint(F:"function", y0:list, T:list)
```

où $F : (y, t) \mapsto F(y, t)$ est la fonction définissant l'équation différentielle (éventuellement vectorielle), y_0 est la condition initiale (éventuellement vectorielle) et T est un tableau correspondant à la subdivision de l'intervalle $[a, b]$. Autrement dit $T = (t_0, \ldots, t_n)$ avec $t_0 = a$ et $t_n = b$. Et la condition initiale correspond alors à $y(a) = y_0$.

Exemple 2.5 :

```
>>> scipy.integrate as scint
>>> scint.odeint(lambda y,t : y, [1], [k/10 for k in range(11)])
array([[1.        ],
       [1.10517091],
       [1.22140275],
       [1.34985882],
       [1.49182469],
       [1.64872127],
       [1.8221188 ],
       [2.01375273],
       [2.22554103],
       [2.45960316],
       [2.7182819 ]])
>>> [mt.exp(k/10) for k in range(11)]
[1.0, 1.1051709180756477, 1.2214027581601699, 1.3498588075760032, 1.4918246976412
703, 1.6487212707001282, 1.8221188003905089, 2.0137527074704766, 2.22554092849246
8, 2.45960311115695, 2.718281828459045]
```

Remarque :
Le nom de la fonction **`odeint`** fait référence au nom anglais ODE (Ordinary Differential Equation) (EDO, en français), qui est le générique des équations différentielles.

Il existe aussi des EDP : des Équations aus Dérivées Partielles (en français). Quelques une seront traitées en fin d'année en maths, dans le chapitre sur les fonctions de deux variables.

Remarque :
La fonction **`odeint`** utilise un autre module : **`numpy`**. Le module **`numpy`** est hors programme. Il introduit un nouveau type d'objet : les **`array`**. Ce module est plus pratique pour les calculs numériques (il a été

conçu pour). Il contient toutes les fonctions mathématiques de bases qui ont été recodés pour plus de faciliter. Par exemple, `numpy.linspace(0,1,100)` permet de pouvoir découper l'intervalle $[0,1]$ en 100 valeurs régulièrement répartis. Puis `numpy.exp(numpy.linspace(0,1,100))` correspond alors à appliquer la fonction $\exp$ sur toutes les valeurs de la liste de points. Autrement dit, ça correspond à faire `[mt.exp(k/99) for k in range(100)]`.

Il est à noter toutefois que, bien que ce module soit officiellement hors-programme, et malgré les protestations des profs de CPGE, le concours Centrale continue de l'utiliser.

Remarque :
La fonction `odeint` permet d'auto-corriger les erreurs d'approximations calculatoires. Les résultats obtenus sont donc meilleurs.

Exemple 2.6 :

```
>>> F = lambda t,y : t+mt.sin(y)
>>> L = Euler(F,0,5,0,51)
>>> O = scint.odeint(F,0,L[0])
>>> plt.plot(L[0],L[1],label="Euler")
[<matplotlib.lines.Line2D object at 0x385b960>]
>>> plt.plot(L[0],O,label="odeint")
[<matplotlib.lines.Line2D object at 0x382bb88>]
>>> plt.legend()
<matplotlib.legend.Legend object at 0x34163b0>
>>> plt.show()
```