Chapitre 11 D : approximation numérique des solutions d'équations différentielles (approfondissement)

Valentin Melot — Terminale spé maths A

27 avril 2021

Certaines équations différentielles ne possèdent pas d'expression sous forme close, c'est-à-dire exprimable à partir de fonctions simples. On souhaite donc désormais donner une approximation.

1 Formalisation

Le traitement numérique d'une équation différentielle ordinaire requiert toujours, d'une façon ou d'une autre, trois éléments.

1.1 Mise sous forme résolue

Une équation différentielle ordinaire peut être mise sous une forme dite *forme résolue*, c'est-à-dire :

$$y'(t) = F(t, y(t))$$

où F est une fonction à *deux* paramètres, que l'on peut définir dans Python en respectant impérativement l'ordre des paramètres.

Exemple 1 Pour l'équation $y'(t) + ty(t) = 0$, on a $F : (t, y) \longmapsto -ty$.

La fonction peut être définie de la façon suivante avec Python :

```
def F(t,y):
    return -t*y
```

Exemple 2 Pour l'équation $y'(t) + y^2(t) = 0$, on a $F : (t, y) \longmapsto -y^2$. On peut écrire en Python (sur une seule ligne) :

```
def F(t,y): return -y**2
```

On remarque que l'un des paramètres de la fonction n'intervient pas.

Remarque 3 Attention, dans certaines bibliothèques de traitement automatique d'équations différentielles, l'ordre des paramètres est inversé. L'ordre retenu ici est celui recommandé pour l'utilisation de la bibliothèque `scipy`.

Remarque 4 Une telle formalisation d'une équation différentielle est nécessairement restrictive. Certaines équations ne peuvent pas être mises sous cette forme, et requièrent donc un travail préliminaire. Par exemple :

— Les équations faisant intervenir des composées dans y, telles que $y'(t) = y(t^2)$;

— Les équations dans lesquelles y' est multiplié par une quantité susceptible de s'annuler, telles que $\sin(t)y'(t) = y(t)$;

— Les équations dans lesquelles l'expression de y' n'est pas univoque, par exemple $y'^2 = y$.

1.2 Intervalles du temps

L'approximation numérique d'une équation différentielle ne permet que d'obtenir les valeurs des solutions en un nombre fini de points.

Par la suite, on *discrétise les valeurs du temps*, c'est-à-dire que l'on se donne un ensemble T, fini, et l'on approximera les valeurs de $y(t)$ pour $t \in T$ uniquement. T est donc de la forme :

$$\{t_0 \,; t_1 \,; t_2 \,; \dots \,; t_n\}.$$

Le *pas* de la suite est la quantité $t_{i+1} - t_i$. Lorsque la suite $(t_i)_{0 \leqslant i \leqslant n}$ est une suite arithmétique, on parle de *pas constant*. Plus le pas sera faible, plus le calcul sera précis, mais plus les calculs seront lents. Avec Python, on peut utiliser le module `numpy`, dont la fonction `linspace` crée un tableau à pas constant.

```
>>> import numpy as np
>>> T = np.linspace(0,10,1001) #Tableau à pas constant 0.01 seconde sur 10 secondes
>>> T
array([ 0.  ,  0.01,  0.02, ...,  9.98,  9.99, 10.  ])
```

On calculera donc une suite de valeurs $(y_i)_{0 \leqslant i \leqslant n} = (y(t_i))_{0 \leqslant i \leqslant n}$.

Remarque 5 Certaines bibliothèques parviennent à définir automatiquement un pas de temps le plus adapté pour la résolution du problème. Elles peuvent aussi recevoir une liste de temps auxquels on veut *obligatoirement* évaluer la solution, et compléter automatiquement cette liste avec des temps intermédiaires adaptés à la résolution du problème.

Dans tous les cas, lorsque l'on utilise des fonctions de bibliothèques, il est essentiel de se référer à la documentation.

Remarque 6 Comme on l'a vu au chapitre précédent, certaines équations peuvent ne pas avoir de solution définie sur $\mathbf{R}$ tout entier. Si la solution de l'équation différentielle considérée n'est pas définie en chacun des points de T, deux comportement sont possibles :

— Soit l'algorithme n'identifiera pas le problème, et renverra une solution incorrecte.

— Soit l'algorithme dispose de mécanismes de contrôle, et sera capable d'identifier le problème. Il pourra alors par exemple renvoyer des valeurs indéfinies (`NaN`), ou émettre un message d'erreur (une *exception*) ou un avertissement.

1.3 Condition initiale

Ainsi qu'il a été vu dans le chapitre précédent, il est fréquent que la solution d'une équation différentielle soit unique à condition de disposer d'une condition initiale (de la forme $y(t_i) = y_i$). Un théorème puissant, appelé *théorème de Cauchy-Lipschitz* garantit dans un certain nombre de cas l'unicité de la solution lorsqu'une condition initiale est donnée.

Le plus souvent, on donne une condition initiale correspondant à la valeur du début de l'intervalle de temps, ce qui est le plus sensé lorsque l'équation modélise une évolution au cours du temps. Autrement dit, on fixe un paramètre $y_0 = y(t_0)$.

Remarque 7 Il est fréquent que l'on soit amené à étudier l'effet de la variation de la condition initiale sur le comportement d'un système. Les bibliothèques acceptent souvent de traiter en une seule fois plusieurs solutions initiales. Plutôt qu'une valeur y_0 unique, on leur passe alors un tableau de valeurs correspondant à chacune des solutions initiales à traiter, et elles renvoient un tableau de tableaux correspondant aux y_i pour chaque solution initiale.

2 Méthode d'Euler

2.1 Principe (essentiel)

La méthode d'Euler est la plus simple des méthodes de résolution numérique d'équation différentielle. Rappelons que si y est dérivable en t, alors on peut écrire pour $h \in \mathbf{R}$:

$$y(t+h) = y(t) + hy'(t) + hR(h)$$

avec $\lim\limits_{h\to 0} R(h) = 0$. Or, en l'espèce, on connaît $y'(t)$, qui grâce à la *forme résolue* de l'équation différentielle :

$$y(t+h) = y(t) + h \cdot F(t, y(t)) + hR(h).$$

Si h est pris suffisamment petit, il est légitime [1] de négliger le dernier terme de la somme. Autrement dit, cela revient à approximer y par sa tangente prise au point (t_i, y_i), dont on connaît le coefficient directeur.

En choisissant alors $h = t_{i+1} - t_i$, la suite $(y_i)_{0\leqslant i\leqslant n}$ vérifie pour tout $0 \leqslant i \leqslant n-1$:

$$y_{i+1} = y_i + (t_{i+1} - t_i) \cdot F(t_i, y_i).$$

Finalement, on a obtenu une estimation de la fonction y en chacun des points $t_1, t_2, \ldots, t_n$.

1. Sous certaines conditions relatives à la fonction F. Cela peut bien sûr être démontré, mais la démonstration est hors de portée en terminale. Plus exactement, on démontre que lorsque l'on néglige ce dernier terme, lorsque le pas p tend vers 0, les valeurs des estimées $y_p(t)$ tendent vers la vraie valeur $y(t)$. Voir l'exercice 8 pour un exemple concret.

2.2 Exemple théorique : approximation de l'exponentielle par une suite géométrique

L'unique solution de l'équation $y' = y$ avec pour condition initiale $y(0) = 1$ est la fonction exponentielle.

Pour appliquer la méthode d'Euler, on pose donc $F : (t, y) \longmapsto y$. On se place sur l'intervalle $[0, 1]$, et l'on choisit une discrétisation à pas constant $p_n = \frac{1}{n}$ (on fera varier n par la suite).

La méthode d'Euler nous donnera une suite $(y_{n,i})_{0 \leqslant i \leqslant n}$ telle que pour tout $0 \leqslant i \leqslant n$, $y_{n,i}$ approxime la solution de l'équation différentielle au temps $t_{n,i} = \frac{i}{n}$.

La suite obtenue (qui dépend du paramètre n) est alors :

$$\begin{cases} y_{n,0} = 1 \\ y_{n,i+1} = y_{n,i} + \frac{1}{n} F(t_i, y_{n,i}) = \left(1 + \frac{1}{n}\right) y_{n,i}. \end{cases}$$

Donc $(y_{n,i})_{0 \leqslant i \leqslant n}$ est une suite géométrique de raison $y_1 = 1 + \frac{1}{n}$, et de premier terme 1. Autrement dit, pour tout $0 \leqslant i \leqslant n$,

$$y_{n,i} = \left(1 + \frac{1}{n}\right)^i.$$

L'exercice suivant vise à démontrer que l'approximation est correcte, ou autrement dit que l'erreur d'approximation converge bien vers 0 lorsque n devient grand.

Exercice 8 On rappelle que $y_{n,i}$ est une approximation de $\exp(t_{n,i})$, avec $t_{n,i} = \frac{i}{n}$. On a donc $\exp(t_{n,i}) = \alpha_n^i$ avec $\alpha_n = \exp(t_{n,1})$.

On appelle $\varepsilon_{n,i} = |\exp(t_{n,i}) - y_{n,i}|$ l'*erreur d'approximation* en $t_{n,i}$ pour la méthode d'Euler à pas $\frac{1}{n}$.

1. On fixe provisoirement n.
   (a) En utilisant la convexité de l'exponentielle, démontrer que $\alpha_n \geqslant y_{n,1}$.
   (b) En déduire que pour tout $0 \leqslant i \leqslant n$, $\exp(t_{n,i}) \geqslant y_{n,i}$, puis que la suite des erreurs $(\varepsilon_{n,i})_{0\leqslant i\leqslant n}$ est croissante. Autrement dit, l'erreur la plus élevée est celle commise à l'extrémité droite de l'intervalle.
2. Il suffit donc de démontrer que cette erreur $\varepsilon_{n,n}$ converge vers 0 lorsque n tend vers $+\infty$, ou autrement dit que $y_{n,n}$ tend vers $\exp(t_{n,n}) = e$ pour $n \to +\infty$.
   (a) En utilisant la valeur de la dérivée de $x \longmapsto \ln(1+x)$ en 1, justifier que $\lim\limits_{n\to+\infty} n \ln\left(1+\frac{1}{n}\right) = 1$.
   (b) Conclure.

2.3 Implémentation numérique.

On propose l'implémentation suivante :

```
def euler(F, T, y0):
    """
    Entrées :
        * F : fonction à deux paramètres t et y
        * T : tableau des temps, triés dans l'ordre croissant
        * y0 : réel définissant une condition initiale
    Sortie : en chaque point de l'intervalle de temps T, une estimée de
    l'équation différentielle y'(t) = F(t, y(t)) ; y(t0) = y0 par la méthode
    d'Euler.
    """
    y = [y0]
    for i in range(len(T) - 1):
        y_iplusun = y[i] + (T[i+1] - T[i])*F(T[i], y[i])
        y.append(y_iplusun)
    return y
```

On peut par exemple résoudre $y' = y$ sur l'intervalle $[0, 4]$, et observer l'effet de la discrétisation du temps. On utilise la bibliothèque `matplotlib.pyplot` pour afficher les courbes obtenues, et `numpy` pour faciliter le calcul sur des tableaux de valeurs.

```
import matplotlib.pyplot as plt
import numpy as np

def F(t,y):
    return y

# Crée trois tableaux de discrétisation de l'intervalle [0,4]. Le troisième paramètre
    donne le nombre total de valeurs (par exemple, T3 = {0, 2, 4}).
T3 = np.linspace(0,4,3)
T10 = np.linspace(0,4,10)
T100 = np.linspace(0,4,100)

appr_exp3 = euler(F, T3, 1)
plt.plot(T3, appr_exp3) #Affiche les points de coordonnées (T3[i], appr_exp3[i])

appr_exp10 = euler(F, T10, 1)
plt.plot(T10, appr_exp10)

appr_exp100 = euler(F,T100, 1)
plt.plot(T100, appr_exp100)

vraie_exp = np.exp(T100) #Façon rapide de calculer les exp(T100[i])
plt.plot(T100, vraie_exp)

plt.show() #Affiche les quatre courbes
```

Le résultat obtenu est présenté en figure 1.

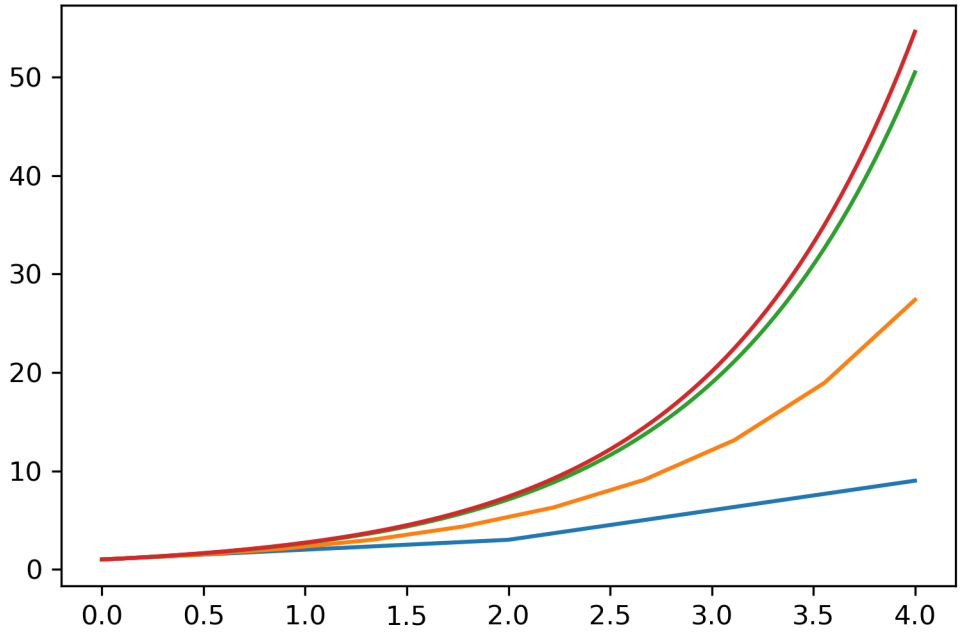

FIGURE 1 – Effet de la discrétisation du temps pour le calcul de l'exponentielle par la méthode d'Euler. La courbe rouge est la courbe véritable de la fonction exponentielle sur $[0, 4]$. Les courbes bleue, orange et verte sont données par la méthode d'Euler à pas $p = 2$, $p = 4/9$ et $p = 4/99$.

On peut de cette façon approximer des solutions d'équations bien plus difficiles. Reprenons par exemple le circuit RC à résistance variable en régime sinusoïdal vu au chapitre précédent, soit (les constantes sont choisies égales à 1 pour simplifier) :

$$\frac{\mathrm{d}U}{\mathrm{d}t} + (1 - \cos t)U = \cos(t).$$

En fixant $U(t = 0) = 0$, on peut visualiser une solution sur l'intervalle $[0, 20]$ en écrivant le code suivant, dont la sortie est présentée en figure 2.

```
import numpy as np
import matplotlib.pyplot as plt
def F(t,y):
    from math import cos
    return cos(t) - (1-cos(t))*y

T = np.linspace(0,20,10000)
U = euler(F, T, 0)
plt.xlabel("t (s)")
plt.ylabel("U (V)")
plt.plot(T,U)
plt.show()
```

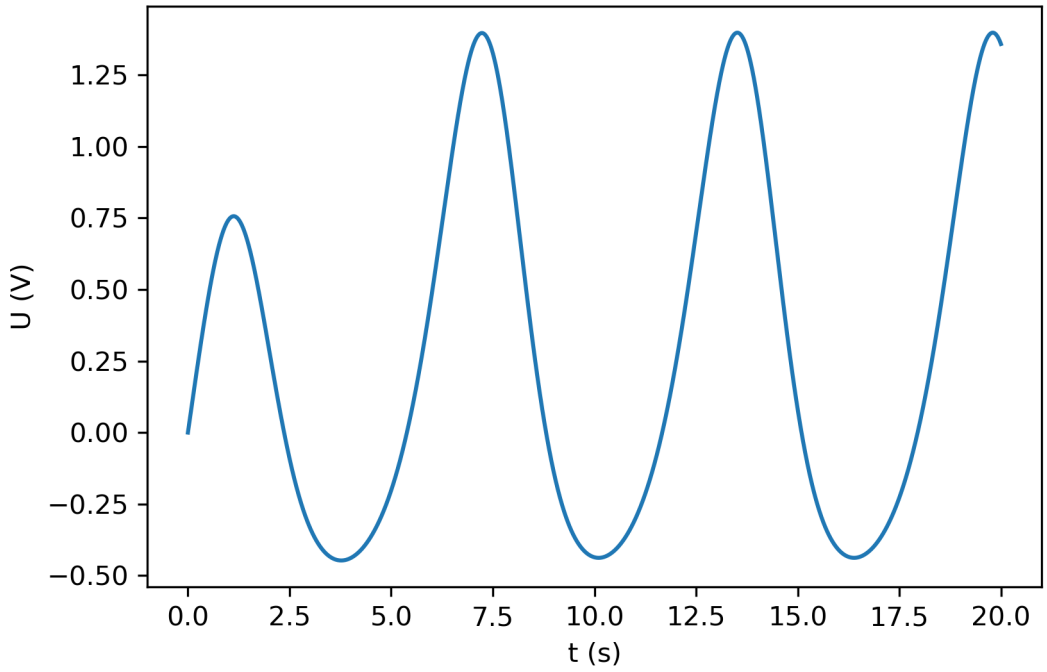

FIGURE 2 – Courbe représentative de la fonction U dans le circuit RC à résistance variable en régime sinusoïdal, avec $U(0) = 0$.

Le *portrait de phase* désigne la trajectoire suivie par le point $(U(t), U'(t))$ au cours du temps. On peut l'afficher en utilisant le code suivant, dont la sortie est donnée en figure 3.

```
Uprime = [F(T[i],U[i]) for i in range(len(T))]
plt.plot(Uprime, U)
plt.show()
```

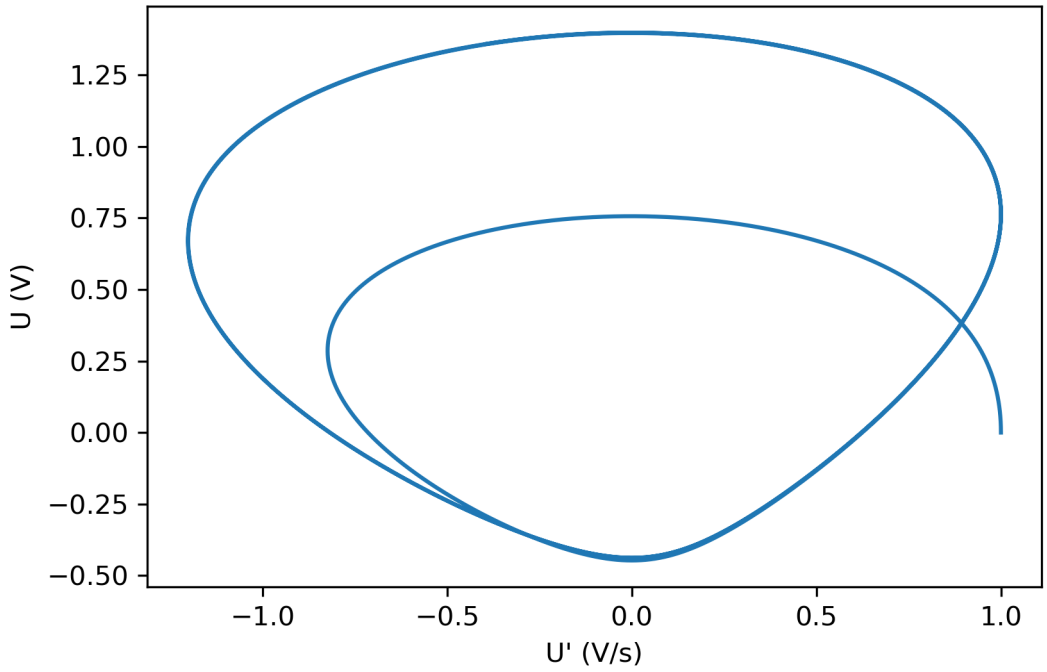

FIGURE 3 – Portrait de phase du circuit RC à résistance variable en régime sinusoïdal avec $U(0) = 0$.

Exercice 9 Ajuster l'exemple précédent pour étudier l'effet d'un déphasage φ dans le signal d'entrée.

Exercice 10 Ajuster l'exemple pour étudier l'effet d'une modification de la pulsation du signal d'entrée.

Exercice 11 Coder une fonction utilisant la méthode d'Euler pour afficher la courbe représentative d'une primitive de la fonction $x \longmapsto e^{-x^2}$, puis de la fonction $x \longmapsto \frac{\sin x}{x}$.

Exercice 12 Modifier la fonction `euler` pour qu'elle prenne prenne non plus une condition initiale y_0 unique, mais un tableau de conditions initiales, et renvoie un tableau de tableaux correspondant à chacune des conditions initiales.

Exercice 13 Modifier la fonction `euler` pour qu'elle ne prenne non plus une condition *initiale*, mais une condition *finale*, c'est-à-dire une valeur $y_n = y(t_n)$.

3 Utilisation de `scipy`

La méthode d'Euler repose sur le fait d'approximer une courbe par sa tangente en un point. Cette approximation, bien qu'elle donne des solutions acceptables, reste cependant assez violente. Bien que l'erreur tende vers 0 pour un nombre de tour de boucles n grand, cette convergence peut être jugée trop lente.

En outre, la méthode d'Euler est peu « robuste », dans le sens où des petites erreurs de calcul survenues en un point s'accumulent lorsque l'on s'éloigne de la condition initiale, sans qu'il soit possible de les rattraper.

Enfin, la méthode d'Euler ne permet pas de gérer correctement les « problèmes de bords », c'est-à-dire les situations où l'équation différentielle n'est pas définie sur $\mathbf{R}$ tout entier. Dans cette situation, les solutions obtenues seront erronées, sans qu'aucun mécanisme ne permette de le signaler.

Il existe de nombreuses méthodes alternatives à celle d'Euler, plus ou moins précises et robustes, et plus ou moins coûteuses en calculs. L'une des plus utilisées en pratique est appelée *méthode de Runge-Kutta* ; elle raffine la méthode d'Euler en évaluant la fonction F en des points supplémentaires pour affiner l'estimée. Le choix de l'*ordre* permet de décider si les approximations sont faites par des droites (ordre 1), des paraboles (ordre 2), etc.

La bibliothèque Python `scipy.integrate` contient diverses fonctions « clef en main » capables d'approximer des solutions d'équations différentielles par des sophistiquées. La fonction `scipy.integrate.solve_ivp`, sert d'interface à une résolution automatisée de problème. Il suffit de lui passer comme paramètre la fonction F, les extrémités de l'intervalle de temps, et la condition initiale ; elle construit elle-même l'ensemble des temps de façon optimisée. La valeur de retour est un objet qui contient les valeurs de t et de y. Par exemple, pour l'équation $y' = y$ sur $[0, 4]$ avec comme condition initiale $y(0) = 1$:

```
from scipy.integrate import solve_ivp
import matplotlib.pyplot as plt
import numpy as np

def F(t,y):
    return y

sol = solve_ivp(F, [0,4], [1]) #Attention aux crochets autour de la condition initiale
# sol.t est un tableau contenant les temps
# sol.y[0] est un tableau contenant les valeurs de y

plt.plot(sol.t, sol.y[0])

# On compare avec la véritable courbe de l'exponentielle
t = np.linspace(0,4,100)
vraie_exp = np.exp(t)
plt.plot(t, y)
plt.show()
```

Le résultat est affiché en figure 4.

Remarque 14 Attention : il n'est pas souhaitable de chercher à recopier ce morceau de code dans vos futurs projets informatiques. Les fonctions contenues dans les bibliothèques évoluent rapidement, et il est possible que `solve_ivp` soit, dans trois, cinq ou dix ans, dépréciée. Par ailleurs, cette fonction, utilisée de cette façon, est adaptée pour résoudre un problème précis, mais pourrait être peu pertinente pour résoudre d'autres problèmes. En copiant-collant ce code et en changeant seulement la fonction `F`, vous pourriez obtenir un code peu performant ou erroné.

Lorsque vous serez amenés à devoir résoudre numériquement des équations différentielles pour vos projets ultérieurs, cherchez dans la documentation du module `scipy.integrate` les fonctions à utiliser, et ne vous fiez pas à des documents trop anciens.

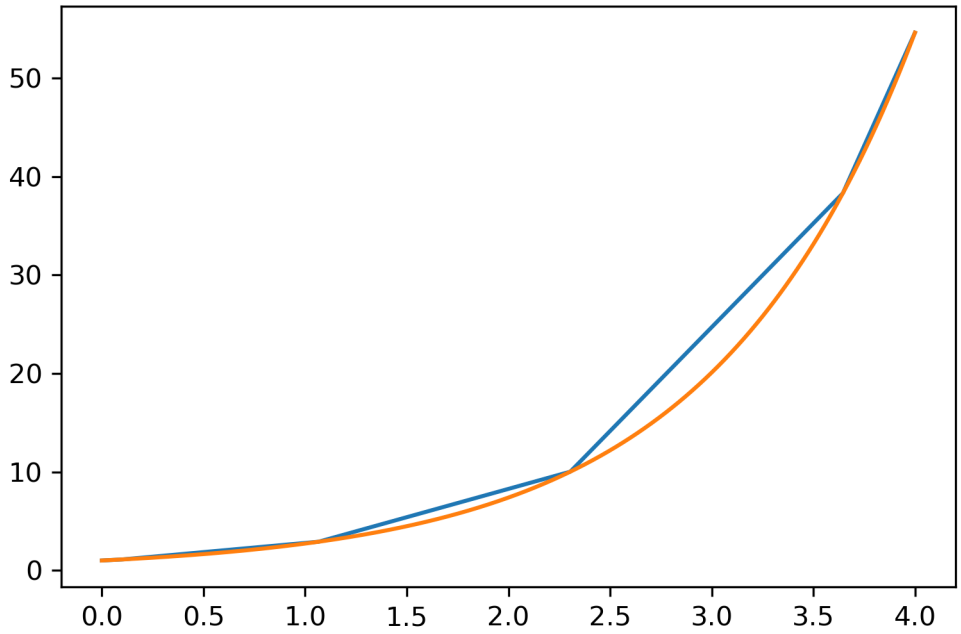

FIGURE 4 – Approximation de la fonction exponentielle sur l'intervalle $[0, 4]$ donnée par la méthode de Runge-Kutta à l'ordre 5(4).

4 Généralisation

En réalité, il est facile de généraliser ces méthodes de résolution d'équation différentielles à des problèmes plus complexes que les équations à une inconnue d'ordre 1.

Systèmes à plusieurs inconnues. Pour un système à plusieurs inconnues, on peut écrire les équations de façon à exprimer les dérivées des inconnues comme une fonction de celles-ci. Par exemple, s'agissant du système de Lotka-Volterra, dont on rappelle l'équation :

$$\begin{cases} \ell'(t) = (\alpha - \beta r(t))\ell(t) \\ r'(t) = (\delta\ell(t) - \gamma)r(t), \end{cases}$$

on peut écrire que $(\ell'(t), r'(t)) = F(t, (\ell(t), r(t)))$ avec

$$F(t, (\ell, r)) = ((\alpha - \beta r)\ell, (\delta\ell - \gamma)r).$$

La méthode d'Euler et les méthodes plus puissantes évoquées ci-dessus s'appliquent donc de la même façon. Les fonctions de la bibliothèque `scipy.integrate` acceptent de travailler avec une telle fonction F multiparamétrique.

Équations d'ordres supérieurs. Une fois que l'on sait résoudre un système d'équations différentielles d'ordre 1 à plusieurs inconnues, tous les systèmes d'ordre supérieur s'y ramènent par une astuce assez simple.

Considérons par exemple l'équation $y''(t) + y'(t) + y(t) = f(t)$. En posant $z = y'$, on a $z'(t) = y''(t) = f(t) - y'(t) - y(t) = f(t) - z(t) - y(t)$. Autrement dit, (y, z) est solution du système d'équations :

$$\begin{cases} y'(t) = z(t) \\ z'(t) = f(t) - z(t) - y(t). \end{cases}$$

On s'est ramenés à un système d'ordre 1, dont il est possible d'approximer un résultat avec la méthode d'Euler ou avec `scipy.integrate.solve_ivp`.

Les acquis de ce chapitre

Démonstration exigible : néant.

Savoirs et savoirs-faire indispensables.

1. Savoir transformer une équation différentielle pour la mettre sous forme résolue.
2. Avoir compris le principe sur lequel repose la méthode d'Euler.

Démonstrations qu'il faut avoir comprises :

— Convergence des estimées de l'exponentielle par la méthode d'Euler à pas constant lorsque le pas tend vers 0.

Principaux approfondissements.

1. Implémentation de la méthode d'Euler.
2. Utilisation de `scipy.integrate.solve_ivp` pour résoudre des équations différentielles d'ordre 1 à une inconnue.
3. Résolution numérique d'équations différentielles à plusieurs inconnues ou d'ordres supérieurs, par exemple l'équation de Lotka-Volterra (pour les plus motivés).