Thème info 2 – Exemples de résolution numérique d'une équation différentielle

Il s'agit ici de (re)voir quelques méthodes numériques fournissant une approximation de la solution d'un *problème de Cauchy* (équation différentielle assortie d'une condition initiale, de sorte qu'il y ait une solution unique, cf. le chapitre 10 du cours de maths...).

I - Méthodes à un pas pour une équation scalaire d'ordre 1

1) Généralités

On cherche à résoudre numériquement (et approximativement !), sur un intervalle $[t_0, t_0 + T]$, une équation (E) de la forme $y' = F(t, y)$ avec la condition initiale $y(t_0) = y_0$.
Soit un *pas* $h > 0$, on pose $N_h = \lfloor T/h \rfloor$ et, pour $k \in [\![0, N_h]\!]$, $t_k = t_0 + kh$.
Une *méthode à un pas* consiste à approcher les valeurs $y(t_k)$ par les y_k définis par

$$y_0 = y_0 \ (!) \quad \text{et} \quad \forall k \in [\![0, N_h[\![\quad y_{k+1} = y_k + h.\Phi(t_k, y_k, h) \quad (R) \ ;$$

L'initialisation pour y_0 n'est pas si triviale qu'elle en a l'air, car on ne dispose en général que d'une valeur numérique approchée (cf. les définitions de la stabilité et de la convergence ci-dessous).

Par exemple, pour la méthode d'Euler, $\Phi(t_k, y_k, h) = F(t_k, y_k)$ ne dépend pas de h.
Noter que la solution étudiée vérifie

$$y(t_{k+1}) = y(t_k) + \int_{t_k}^{t_{k+1}} F\big(t, y(t)\big)\,\mathrm{d}t.$$

La méthode d'Euler consiste donc à approcher la fonction intégrée ci-dessus par la constante égale à la valeur qu'elle prend en t_k, puisque $t_{k+1} - t_k = h$.

On parle de *méthode explicite* car les valeurs successives se calculent directement sans résoudre d'équation.
La méthode est dite *consistante* si et seulement si

$$\varepsilon(h) \underset{h\to 0}{\longrightarrow} 0 \quad \text{où} \quad \varepsilon(h) = \max_{0 \le k < N_h} \left| \frac{y(t_{k+1}) - y(t_k)}{h} - \Phi(t_k, y(t_k), h) \right|.$$

On dit que la méthode est *d'ordre* p si et seulement si $\varepsilon(h) \underset{h\to 0}{=} O(h^p)$.
La méthode est dite *stable* si et seulement s'il existe une constante S telle que, pour toutes suites (y_k), $(\widetilde{y}_k)$ définies par

$$\forall k \in [\![0, N_h[\![\quad \begin{cases} y_{k+1} = y_k + h.\Phi(t_k, y_k, h) \\ \widetilde{y}_{k+1} = \widetilde{y}_k + h.\Phi(t_k, \widetilde{y}_k, h) + \varepsilon_k \end{cases}$$

on ait

$$\max_{0 \le k \le N_h} |\widetilde{y}_k - y_k| \le S\left(|\widetilde{y}_0 - y_0| + \sum_{k=0}^{N_h - 1} \varepsilon_k\right).$$

La méthode est dite *convergente* si et seulement si $\max\limits_{0 \le k \le N_h} |y(t_k) - y_k|$ tend vers 0 lorsque y_0 tend vers $y(t_0)$ et h tend vers 0.

En supposant F "suffisamment régulière", on démontre (voir un cours d'analyse numérique...) que :

- si la méthode est consistante et stable, alors elle converge ;
- la méthode est consistante si et seulement si $\Phi(t, y, 0) = F(t, y)$;
- si Φ est lipschitzienne en y, alors la méthode est stable.

Noter que cette présentation des méthodes *à pas constant* peut être généralisée à l'utilisation d'un *pas variable* (h étant remplacé par h_k...). La valeur de h peut ainsi être adaptée pour concentrer les points de la subdivision aux endroits où la solution varie le plus rapidement. On parle de *méthodes adaptatives*. Le choix d'une telle méthode permet d'améliorer la précision aux endroits "sensibles" et d'éviter l'accumulation d'erreurs dues aux calculs approchés à des endroits où l'on n'a pas besoin de beaucoup de points...

2) Programmation en Python

Écrire une fonction d'en-tête `X(t0,T,h)` renvoyant le vecteur `numpy` $[t_0, \ldots, t_{N_h}]$ (cf. `np.arange`).

Écrire une fonction d'en-tête `Y(t0,T,h,y0,Phi)` renvoyant le vecteur `numpy` $[y_0, \ldots, y_{N_h}]$, construit selon la relation (R) du § **1**. Cette fonction `Y` appelle la fonction `Phi`, qui devra avoir été définie au préalable et correspond à la méthode choisie (cf. le § **6**).

Noter que `Phi` elle-même appellera la fonction `F` définissant l'équation différentielle (E) (cf. les exemples des § **3**, **4** et **5**) !

Ainsi `plt.plot(X(t0,T,h),Y(t0,T,h,y0,Phi))` tracera le graphe de la solution approchée.

Pour plus d'options pour les graphiques, voir l'annexe `pyplot` à la section **IV**.

Noter qu'**il faut** utiliser des tableaux `numpy` et non des "listes" Python pour pouvoir calculer des combinaisons linéaires !

3) Rappel : méthode d'Euler

C'est la méthode à un pas définie par $\Phi(t, y, h) = F(t, y)$, ce qui consiste à approcher $y'(t_k)$ par $\dfrac{y_{k+1} - y_k}{h}$.
On peut montrer que, pour F suffisamment régulière, la méthode converge et elle est d'ordre 1.

4) Méthode de Heun

C'est la méthode à un pas définie par $\Phi(t, y, h) = \dfrac{1}{2}\big(F(t, y) + F(t + h, y + hF(t, y))\big)$, ce qui — d'une certaine manière — consiste à approcher $\displaystyle\int_{t_k}^{t_{k+1}} F(u, y(u))\,\mathrm{d}u$ par l'aire d'un trapèze au lieu du rectangle utilisé dans la méthode d'Euler.

On peut montrer que, pour F suffisamment régulière, la méthode converge et elle est d'ordre 2.

5) Méthode de Runge-Kutta d'ordre 4 (RK4 pour les intimes)

Méthode très utilisée, plus sophistiquée mais plus précise !
$\Phi(t, y, h)$ est calculé de la façon suivante en posant :

$$a = F(t, y) \;\; ; \quad b = F\left(t + \frac{h}{2}, y + \frac{h}{2}.a\right) \;\; ; \quad c = F\left(t + \frac{h}{2}, y + \frac{h}{2}.b\right) \;\; ; \quad d = F(t + h, y + h.c)$$

et enfin

$$\Phi(t, y, h) = \frac{1}{6}\big(a + 2b + 2c + d\big)$$

On montre que, pour F suffisamment régulière, la méthode converge et elle est d'ordre... 4 !

6) Comparaison des trois méthodes : premier exemple

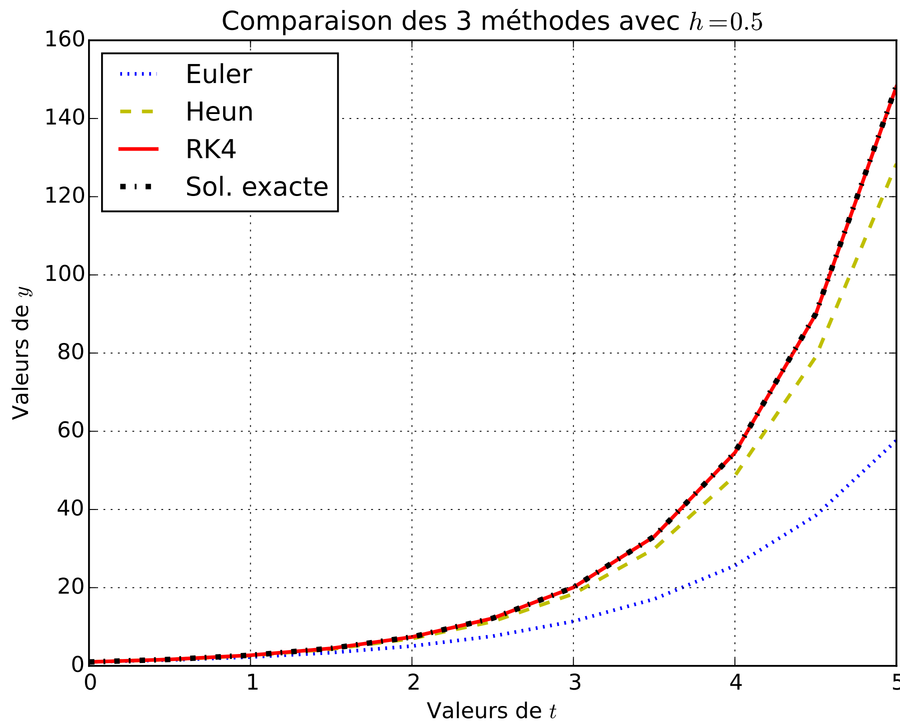
Programmer les trois fonctions Φ ci-dessus, en les baptisant différemment (par exemple `Euler`, `Heun`, `RK4` !), ce qui permet de tracer sur un même graphe les trois courbes (plus éventuellement celle de la solution exacte si on la connaît !).

Tester avec $F(t, y) = y$, par exemple sur $[0, 5]$ avec $y_0 = 1$, la solution exacte est alors $t \mapsto \exp t$!

Tracer les courbes avec diverses valeurs de h. Pour $h = 0.5$, on vérifie la "hiérarchie" attendue, la méthode RK4 donnant une courbe qui se superpose quasiment avec celle de la solution exacte (en pointillés) !

Noter que, pour tracer le graphe d'une fonction f avec `plt.plot(X,Y)`, on doit construire le tableau `X` des abscisses des points souhaités, ainsi que le tableau `Y` des ordonnées correspondantes. Or `numpy` permet d'appliquer certaines opérations directement sur le tableau `X` : par exemple, les fonctions polynomiales sont interprétées correctement : `Y=X^2-3*X+2` donnera le tableau rempli avec les $f(x)$ pour x dans `X`, où $f : x \mapsto x^2 - 3x + 2$.

`numpy` fournit également des versions *vectorisées* de la plupart des fonctions usuelles : par exemple, `np.exp(X)` donne le tableau rempli avec les $\exp(x)$ pour x dans `X`... Plus de détails au § **IV-1**.



Conseils pratiques : pour plus de souplesse, il est recommandé d'insérer les commandes à exécuter pour les tests **à la fin du script** et non dans le "shell" où la reprise de plusieurs commandes précédentes est fastidieuse (on peut aussi définir une unique fonction qui exécute les tests et que l'on appelle depuis le "shell", mais la liste des paramètres peut s'allonger très vite...). Dans ce contexte, on peut se permettre d'utiliser des variables globales, typiquement ici $t0$, T, h, $y0$, qui seront réutilisées pour les différents tracés. On pourra alors par exemple modifier la valeur de h (à un seul endroit, là où elle est définie) et taper F5 pour refaire les tracés avec la nouvelle valeur.

II - Cas des équations d'ordre 2

Une équation différentielle d'ordre 2, de la forme

$$(E_2) \quad u'' = G(t, u, u'),$$

peut être remplacée par l'équation *vectorielle*

$$(E) \quad y' = F(t, y) \quad \text{où} \quad y = \begin{pmatrix} u \\ v \end{pmatrix} \quad \text{et} \quad F(t, y) = \begin{pmatrix} v \\ G(t, u, v) \end{pmatrix},$$

dont les solutions sont les fonctions *vectorielles* $t \mapsto \begin{pmatrix} u(t) \\ u'(t) \end{pmatrix}$ où u est solution de (E_2).

En remarquant que les méthodes de la section **I** s'appliquent aussi bien à des fonctions vectorielles, adapter les programmes pour afficher le graphe de la solution approchée de (E_2).

On pourra initialiser le tableau des y_k par `y=np.zeros((2,Nh))` (tableau de 2 lignes et N_h colonnes) ; ainsi le vecteur colonne y_k sera obtenu par `y[:,k]` et le tableau ligne des u_k par `y[0,:]`.

Exemple : équation du pendule (non amorti) $u'' = -\sin u$.

Tester différentes conditions initiales.

On remarquera qu'une condition initiale consiste en un vecteur $\begin{pmatrix} u_0 \\ v_0 \end{pmatrix}$, où u_0 est la position initiale et v_0 la vitesse initiale.

On essaiera notamment les conditions initiales $\begin{pmatrix} 0 \\ 1.8 \end{pmatrix}$, $\begin{pmatrix} 0 \\ 2 \end{pmatrix}$, $\begin{pmatrix} 0 \\ 2.2 \end{pmatrix}$, $\begin{pmatrix} \pi \\ 0 \end{pmatrix}$.

III - Filtrage numérique

1) Retour sur le TP 4 de SII

Revoir le texte dudit TP pour les détails… Étant donné un signal d'entrée *bruité* e, sur un intervalle de temps $[0, T]$, la sortie filtrée "brute" avec une *bande passante* b est la solution s de l'équation différentielle

$$\frac{1}{b}y' + y = e$$

telle que $s(0) = 0$ (Heaviside !). On a donc ici $F(t, y) = b(e(t) - y)$.

a) Sinus bruité

Tester le filtrage du signal défini par $e(t) = \sin(\omega t) + a\sin(r\omega t)$. Là encore, on pourra définir ω, a et r comme variables globales de façon à pouvoir les modifier facilement sans allonger les listes de paramètres des fonctions.
Afficher le signal d'entrée sur le même graphique que les sorties filtrées brutes.
Penser à zoomer pour mieux comparer les différentes courbes.

Post-traitement : dans ce cas particulier, on sait traiter le signal de sortie (décalage temporel et amplification). On pose

$$\phi = -\arctan\left(\frac{\omega}{b}\right), \quad \delta = \frac{\phi}{\omega}, \quad A = \sqrt{1 + \left(\frac{\omega}{b}\right)^2}.$$

Le graphe de la sortie traitée est alors donné par

```
plt.plot(X+delta,A*Y)
```

où X, Y sont les tableaux des abscisses et des ordonnées correspondant à la sortie brute.

Modifier le programme pour afficher la sortie traitée et le signal d'entrée sur le même graphique.

b) Rampe bruitée

Tester le filtrage avec $e(t) = t + a\sin(\omega t)$, corriger manuellement le *traînage*…

2) Filtrage de données acquises

Il s'agit ici d'adapter les principes précédents au cas où le signal d'entrée est donné, non plus par une fonction "mathématique", mais par des valeurs discrètes, stockées dans un fichier texte provenant typiquement du module d'acquisition de données d'une machine. À titre d'exemple, une machine du labo de SII a fourni le fichier `Filtrage_fort.csv`, à copier du dossier `PSIetoile` vers votre dossier de travail !

a) Récupération des données

Ce fichier contient 1 000 lignes, chacune contenant deux valeurs séparées par des `';'`.
Attention ! La version française d'Excel crée des fichiers `.csv` où les données sont séparées par des `';'`, afin de pouvoir utiliser la virgule comme séparateur décimal. **Mais** `numpy` ne comprend que le **point** décimal. Dans le fichier fourni, les virgules ont déjà été remplacées par des points, il faudra penser à le faire dans la "vraie vie" !
La commande `donnees=np.loadtxt('Filtrage_fort.csv',delimiter=';')` crée un tableau `numpy` rempli par les valeurs trouvées dans le fichier texte fourni, à condition que le texte soit "interprétable" (cf. la mise en garde ci-dessus).
S'il n'y a pas eu d'erreur, `np.shape(donnees)` donne les dimensions du tableau créé.
Ici, la première colonne contient les valeurs (discrètes) acquises en tant que signal d'entrée. La seconde colonne contient le résultat du filtrage effectué par le logiciel intégré dans la machine (avec l'option "filtrage fort" dont on ne sait pas grand chose…).
On peut facilement récupérer une colonne entière, par exemple `e=donnees[:,0]` extrait la première colonne.
Dans ce cas particulier, les instants auquels les données ont été acquises ne sont pas dans le fichier, il fallait les noter au moment de l'acquisition, en l'occurrence toutes les millisecondes durant 1s !
Par exemple, `plt.plot(np.arange(0,1,0.001),e)` tracera le graphe de l'entrée brute.

b) Filtrage

Adapter les programmes précédents, pour le cas présent où l'on n'a pas la **fonction** e, mais le **tableau** `e` tel que `e[k]` contient la valeur $e(t_k)$.

IV - Annexe numpy et pyplot

1) Quelques remarques sur numpy

On suppose le module `numpy` chargé avec l'alias habituel : `import numpy as np`

On crée un tableau t plein de n zéros (de type `float` par défaut) par `t=np.zeros(n)`. Comme les listes Python, le tableau est indexé de 0 à $n-1$.

On peut ensuite modifier les valeurs stockées par des affectations telles que `t[k]=expr`.

Il existe deux commandes créant un tableau rempli avec des valeurs *équiréparties* :

- `np.arange(start,stop,step)` fonctionne de façon similaire à `range` mais renvoie un tableau `numpy` (`a` comme `array`...) ;
- `np.linspace(start,stop,num)` est similaire, mais **calcule le pas** en fonction du nombre `num` de points souhaité. L'option `retstep=True` fait que `linspace` renvoie le couple (t, h) où h est le pas calculé.

Voir l'aide de `numpy` pour tous les détails.
Pour un contrôle total, il est toujours possible de remplir un tableau "à la main"...

Application d'une fonction à tous les éléments d'un tableau

Objectif essentiel pour tracer des graphes avec `plot`, puisqu'il faut fournir le tableau des abscisses et le tableau des ordonnées...
Deux solutions classiques (la première est conseillée dans le contexte de ce thème) :

- `numpy` fournit des versions *vectorisées* des fonctions usuelles : `np.exp(t)` renvoie le tableau rempli par les $\exp(x)$ pour x dans t, où le tableau `numpy` t contient les abscisses t_k (cela ne fonctionne pas avec les listes Python...) ;
- une **fonction** `f` définie par l'utilisateur peut être *vectorisée* : `np.vectorize(f)` renvoie une fonction qui reçoit comme paramètre un tableau `numpy` et renvoie le tableau obtenu en appliquant `f` à tous ses éléments ;
- `map` permet d'appliquer une fonction à tous les éléments d'un "itérable" Python (y compris les listes !), le résultat étant lui-même un "itérable", ce qui nécessite parfois une conversion : on est déçu par `print(map(exp,t))` tandis que `print(list(map(exp,t)))` donne le résultat attendu...

On peut toujours aussi remplir un tableau à l'aide d'une boucle !

2) Quelques options pour les graphiques avec plot

On suppose le module `pyplot` chargé avec l'alias habituel : `import matplotlib.pyplot as plt`

- La commande de base est `plt.plot(X,Y)` qui trace la "ligne brisée" reliant les points de coordonnées $(X[k], Y[k])$, X et Y étant deux tableaux de flottants de même taille. Plusieurs commandes de cette forme permettent de mémoriser plusieurs "courbes" (qui sont en fait des lignes brisées !).
- La commande `plt.show()` affiche le graphique à l'écran. Par défaut, les intervalles pour les valeurs des abscisses et des ordonnées sont déterminés automatiquement, de même que les unités et graduations sur les axes.

  Un zoom interactif est possible dans la fenêtre graphique, ainsi que des translations et changements d'échelles.

  Attention ! `plt.show()` bloque l'exécution du script, les instructions qui la suivent ne seront exécutées qu'après fermeture de la fenêtre graphique.
- La commande `plt.savefig(nom_de_fichier)` permet d'enregistrer la figure dans un fichier, dont on donne le nom en paramètre, dans une chaîne de caractères. Divers formats sont disponibles et automatiquement utilisés si le nom du fichier se termine par l'extension correspondante : .pdf, .eps, .png, .svg,...
- La commande `plt.grid()` ajoute des lignes en pointillés pour chaque graduation des axes.
- La commande `plt.title('Titre')` définit le titre de la figure.
- La commande `plt.xlabel('Valeurs de $t$')` définit l'étiquette de l'axe des abscisses. De même avec `plt.ylabel` pour l'axe des ordonnées. Les `'$'` permettent de formater des formules mathématiques comme en LaTeX.

Options usuelles pour `plot`

La syntaxe générale `plt.plot(X,Y,options)` permet de personnaliser le tracé.

Voici les options les plus courantes :

- *couleur de la courbe* : `color='black'` donne un tracé en noir ; les autres couleurs disponibles sont : `'blue'`, `'green'`, `'red'`, `'cyan'`, `'magenta'`, `'yellow'`, `'white'` ;
- *style de trait* : `linestyle='-'` donne un trait continu, `'--'` des tirets, `':'` des pointillés, `'-.'` des tirets et points alternés.

  Pour ces deux premières options, on peut utiliser des abréviations : `'k:'` donnera des pointillés noirs, etc.
- *épaisseur du tracé* : `linewidth=2` donne un trait plus épais (valeur 1 par défaut) ; comme on s'y attend, plus la valeur est élevée, plus le trait est épais !
- *marques pour les points* : `marker='x'` matérialise la position de chaque point par un `'x'` ; il y a une grande variété de marqueurs disponibles, `o`, `+`, `*`, `.`, `v`, `^`, $<$, $>$, etc. (voir l'aide de `matplotlib` si besoin) ; la taille des marqueurs peut être réglée grâce à l'option `markersize` ;
- *légende* : `label='Euler'` définit le nom associé à un tracé ; alors la commande `plt.legend()` affiche un cartouche avec les différents styles de tracés suivis du nom associé par l'option `label`. On peut choisir la position du cartouche en précisant par exemple `plt.legend(loc=2)` (0 pour laisser Python choisir, 1 pour en haut à droite, 2 pour en haut à gauche, etc.).