TP5
Systèmes différentiels linéaires

1 L'outil `odeint`

Nous allons voir ici comment résoudre (numériquement) un problème de Cauchy à l'aide de Python ; et observer sur des exemples des caractéristiques des trajectoires.
Nous travaillerons notamment sur des systèmes 2×2 : dans ce cas, les « trajectoires » des solutions se représentent pas des courbes du plan, et nous pourrons illustrer les notions d'état d'équilibre et de convergence (ou divergence) de solutions.

1.1 Résolution d'un système différentiel : la commande `odeint`

Le package `scipy` de Python comporte la fonction `odeint` (*ordinary differential equation integration*) qui permet de trouver l'unique solution d'un problème de Cauchy donné.
La résolution ne se fait pas de manière symbolique (*ie* en manipulant des polynômes, des exponentielles, etc.) mais de manière numérique (ce qui donne des valeurs approchées).

Les imports nécessaires sont :

```
import numpy as np # numpy
import matplotlib.pyplot as plt # pour dessiner
from scipy.integrate import odeint # solveur d'équa diff
```

Par ailleurs, on a vu en cours que la résolution d'un système différentiel est fortement liée à la réduction de la matrice associée : nous importerons donc le package d'algèbre linéaire `numpy.linalg`.

```
import numpy.linalg as al # outils d'algèbre linéaire, notamment de réduction
```

1.2 Syntaxe de `odeint`

La fonction `odeint` résout une équation de type $X' = f(X,t)$ avec la condition initiale $X(t_0) = X_0$. Elle reçoit trois arguments :

1. la fonction f, qui reçoit un `np.array` et un réel t et renvoie un `np.array`.
   NB : dans notre programme les équations sont en fait de la forme $X' = f(X)$ (on parle d'équations différentielles *autonomes*). Il faut néanmoins faire apparaître t dans les arguments de la fonction f, quitte à renvoyer un résultat indépendant de t.

2. la condition initiale X_0 (objet de mêmes dimensions que X).

3. l'intervalle de résolution $[a,b]$ sous forme d'un `np.array` de réels `[t0,t1,t2,...,tn]` tels que $t_0 = a$ et $t_n = b$. Le premier élément `t0` doit être le point définissant la condition initiale.

Par exemple, pour résoudre le système $\begin{cases} x' = -y \\ y' = -2x + y \end{cases}$ sur l'intervalle $[0,1]$, avec la condition initiale $x(0) = 3$, $y(0) = 5$, on écrira :

```
# fonction f(X,t)
def fonc(X,t):
    A = ............
    # la matrice associée au système
    return ............
    # on renvoie le produit matriciel AX

# intervalle de résolution sous forme de linspace
T = ....

# calcul de la solution
sol = odeint( ............ , [....... , .......] , ...........)
```

On rappelle que la commande numpy `np.dot` effectue un produit matriciel.

L'objet `sol` renvoyé par la commande `odeint` est une matrice (type `np.array`) à 2 colonnes, et autant de lignes que de points dans l'argument `T=[t0,t1,t2,...,tn]` : la 1ère colonne contient les valeurs approchées de $x(t_0), x(t_1), ..., x(t_n)$; et la seconde colonne la même chose pour la fonction y.

Exercice 1. `sol` étant un `np.array` à deux colonnes, donner les syntaxes permettant de créer un `np.array` `x` égal à la première colonne de `sol`, et un `np.array` `y` égal à la seconde colonne de `sol`.

```
x = ............
y = ............
```

1.3 Représentation(s) graphique(s)

Étant données des fonctions x_1, x_2, ..., x_n définies sur un intervalle I, on peut envisager deux représentations :

- Les n courbes des x_i sur l'intervalle I.
- L'ensemble des points $(x_1(t), x_2(t), \ldots, x_n(t))$ pour $t \in$ I (qu'on a appelé trajectoire du système).
  Pour cette dernière représentation, on se limitera au cas $n = 2$: ainsi les points $(x_1(t), x_2(t))$ sont des points du plan et on obtiendra une figure en 2 dimensions.

1.4 Courbes des x_i

La commande `odeint` renvoie, pour A $\in \mathcal{M}_n(\mathbb{R})$ et `T` un `linspace` égal à $[t_0, t_1, ..., t_p]$, la matrice

$$\begin{pmatrix} x_1(t_0) & x_2(t_0) & \ldots & x_n(t_0) \\ x_1(t_1) & x_2(t_1) & \ldots & x_n(t_1) \\ \vdots & \vdots & & \vdots \\ x_1(t_p) & x_2(t_p) & \ldots & x_n(t_p) \end{pmatrix}$$

Munis des `array` T $= [t_0, t_1, ..., t_p]$ et X$_1 = [x_1(t_0), \ldots, x_1(t_p)]$, on peut tracer la courbe représentative de x_1 sur l'intervalle $[t_0, t_p]$ par la commande

```
plt.plot( ............ , ............ )
```

Exercice 2. Compléter le code suivant qui permet de tracer les courbes de x, y et z sur l'intervalle $[0,5]$, où x et y sont les solutions de $\begin{cases} x' = -2x+2y-2z \\ y' = -2x-4z \\ z' = 2x-2y+2z \end{cases}$ vérifiant $x(0)=0, y(0)=1, z(0)=3$.

```
A = np.array(.....................................................)
# la matrice du système X'=AX

def fonc(X,t):
    return .............

T = np.linspace( ..................................... )
# dans ce qui suivra t0 est le premier élément de T

sol = odeint(............. , ............. , .............)
# le second argument contient la condition initiale
# de la forme [x(t0),y(t0),z(t0)]

x = .............
y = .............
z = .............
# récupération des x(ti),y(ti),z(ti), cf exo 1

plt.plot(............. , ............. color='red', label='x(t)')
plt.plot(............. , ............. , color='blue', label='y(t)')
plt.plot(............. , ............. , color='green', label='z(t)')
plt.legend()
# tracé des trois courbes x(t),y(t),z(t) avec une légende

plt.show()
# affichage
```

1.5 Outils de réduction

On a vu dans le cours que la structure et les propriétés des solutions d'un système différentiel sont fortement reliées aux éléments propres de la matrice associée à ce système.
Python contient un package d'algèbre linéaire (`numpy.linalg`) qui contient la commande `eig` permettant de déterminer valeurs propres et vecteurs propres d'une matrice diagonalisable (dans le cas contraire, les algos numériques ne fonctionnent pas bien).
Cherchons par exemple à diagonaliser la matrice $A = \begin{pmatrix} -2 & 2 & -2 \\ -2 & 0 & -4 \\ 2 & -2 & 2 \end{pmatrix}$ vue dans le système précédent. Après l'import :

```
import numpy.linalg as al
```

on peut taper dans la console :

```
D,P=al.eig([[-2,2,-2],[-2,0,-4],[2,-2,2]])
print(D)
print(P)
```

ce qui renvoie

```
[ 8.8817842e-16  2.0000000e+00 -2.0000000e+00]
[[-0.81649658  0.57735027 -0.57735027]
 [-0.40824829  0.57735027  0.57735027]
 [ 0.40824829 -0.57735027  0.57735027]]
```

Le premier `array` est la diagonale de la matrice D telle que $M = PDP^{-1}$; le second `array` est la matrice P.
La lecture des colonnes de P donne donc les sous-espaces propres ; la première colonne de P est par exemple

$\begin{pmatrix}-0.81649658\\-0.40824829\\0.40824829\end{pmatrix}$, ce qui montre que $E_0(A) = \text{Vect}\left(\begin{pmatrix}0.81649658\\-0.40824829\\0.40824829\end{pmatrix}\right) = \text{Vect}\left(\begin{pmatrix}2\\-1\\1\end{pmatrix}\right)$.

NB : ces éléments propres sont obtenus par des méthodes numériques (assez souvent comme limites de suites). On voit alors qu'on obtient des valeurs approchées des valeurs propres : pour ±2 ça ne pose pas trop de problème mais la valeur propre nulle apparaît sous la forme 8.88×10^{-16}. On ne peut pas faire mieux avec ces outils.
Par ailleurs les générateurs obtenus pour les SEP peuvent apparaître assez désagréables... mais là encore c'est un effet des méthodes numériques. Dans des cas favorables on saura revenir à des colonnes plus « fréquentables » en divisant les générateurs par des coefficients bien choisis...

Exercice 3. On a vu que $E_0(A) = \text{Vect}\left(\begin{pmatrix}2\\1\\-1\end{pmatrix}\right)$.
À l'aide de la sortie Python ci-dessus, déterminer deux colonnes C_2 et C_3 à coefficients entiers telles que $E_2(A) = \text{Vect}(C_2)$ et $E_{-2}(A) = \text{Vect}(C_3)$.
Observer le comportement en $+\infty$ des solutions du système de l'exercice précédent pour les conditions initiales :

- $X(0)$ « quelconque » (faites plusieurs essais)
- $X(0) = C_i$ $(i = 1, 2, 3)$
- $X(0) = C_1 + C_2$
- $X(0) = C_1 + C_3$
- $X(0) = 2C_1 - C_3$
- $X(0) = 2C_1 + 0.0001C_2 - C_3$

Prévoir la limite $t \to +\infty$ de la solution ayant pour conditions initiales $\alpha C_1 + \beta C_2 + \gamma C_3$, où α, β, γ sont des réels quelconques.

Exercice 4 (pour les enthousiastes)**.** Démontrer la conjecture de l'exercice précédent.

2 Pour aller plus loin : étude de trajectoires en deux dimensions

On se place maintenant en deux dimensions.
On adapte ici le code précédent pour tracer la trajectoire du système $\begin{cases} x' = y \\ y' = -2x + y \end{cases}$ sur $[0, 1]$, avec $x(0) = 1$ et $y(0) = 2$.

```
A = np.array( .... )
# la matrice du système X'=AX

def fonc(X,t):
    return np.dot(A,X)

T = np.linspace( ... )
# dans ce qui suit t0 est le premier élément de T

sol = odeint(fonc, ... ,T)
# les ... contiennent la condition initiale de la forme [x(t0),y(t0)]

x = ...
y = ...
# récupération des x(ti) et y(ti), cf exo 1

plt.plot(... , ...)
# tracé de la courbe formée par les points (x(t),y(t))

plt.show()
# affichage
```

2.1 Trajectoires et champ de vecteurs

Dans la suite, on considérera les 5 matrices suivantes.

Matrice	Valeur propre 1	Sous-espace propre 1	Valeur propre 2	Sous-espace propre 2
$A_1 = \begin{pmatrix} 0 & -1 \\ -2 & 1 \end{pmatrix}$	…	Vect $\left(\begin{pmatrix} 1 \\ 1 \end{pmatrix}\right)$	…	Vect $\left(\begin{pmatrix} -1 \\ 2 \end{pmatrix}\right)$
$A_2 = \begin{pmatrix} -3/2 & -1/2 \\ -1 & -1 \end{pmatrix}$	…	Vect $\left(\begin{pmatrix} 1 \\ 1 \end{pmatrix}\right)$	…	Vect $\left(\begin{pmatrix} -1 \\ 2 \end{pmatrix}\right)$
$A_3 = \begin{pmatrix} 7/3 & 2/3 \\ 4/3 & 5/3 \end{pmatrix}$	…	Vect $\left(\begin{pmatrix} 1 \\ 1 \end{pmatrix}\right)$	…	Vect $\left(\begin{pmatrix} -1 \\ 2 \end{pmatrix}\right)$
$A_4 = \begin{pmatrix} 1/3 & 1/6 \\ 1/3 & 1/6 \end{pmatrix}$	…	Vect $\left(\begin{pmatrix} 1 \\ 1 \end{pmatrix}\right)$	…	Vect $\left(\begin{pmatrix} -1 \\ 2 \end{pmatrix}\right)$
$A_5 = \begin{pmatrix} -1 & 1 \\ 2 & -2 \end{pmatrix}$	…	Vect $\left(\begin{pmatrix} 1 \\ 1 \end{pmatrix}\right)$	…	Vect $\left(\begin{pmatrix} -1 \\ 2 \end{pmatrix}\right)$

(pour plus de simplicité et de lisibilité dans les dessins qui suivront, les directions propres sont les mêmes pour les 5 matrices)

Exercice 5.

1. Utiliser la commande `al.eig` pour compléter le tableau précédent.
2. Pour les systèmes $X' = A_i X$:
   - Déterminer les états d'équilibre.
   - Discuter l'existence de solutions divergentes.

Exercice 6. Compléter le code suivant, qui affiche les courbes représentatives des solutions du système $X' = AX$ pour une matrice A et un intervalle de résolution I définis dans le code.

```
A = np.array( .... )
# la matrice du système X'=AX

def fonc(X,t):
    return np.dot(A,X)

T = np.linspace( ... )

sol = odeint(fonc, ... ,T)
# les ... contiennent une condition initiale de la forme [x(t0),y(t0)]

x = ...
y = ...
# récupération des x(ti) et y(ti), cf exo 1

plt.plot(... , ... , color='red', label='x(t)')
plt.plot(... , ... , color='blue', label='y(t)')
plt.legend()
# tracé des deux courbes x(t),y(t) avec une légende

plt.show()
# affichage
```

Tester ce code sur les 5 matrices ci-dessus. On pourra résoudre sur $[0,1]$ et faire varier la condition initiale.

2.2 Champ de vecteurs et trajectoires

On va dans cette section illustrer la dépendance existant entre la condition initiale d'une solution et son comportement asymptotique, que nous avons esquissé dans ce qui précède.

On définit la notion de champ de vecteurs. Considérons le système différentiel $\begin{cases} x' = y \\ y' = -2x + y \end{cases}$. Si, pour une valeur t_0, une solution vérifie $x(t_0) = a$, $y(t_0) = b$, alors le système montre que $x'(t_0) = b, y'(t_0) = -2a + b$. On

peut alors associer au point (a,b) du plan le vecteur $(b,-2a+b)$: si la trajectoire d'une solution passe par (a,b), le vecteur tangent à la trajectoire en ce point sera $(b,-2a+b)$.

Dans cet exemple, on appelle champ de vecteurs le tracé de tous les vecteurs $(b,-2a+b)$ pour $(a,b) \in \mathbb{R}^2$ (évidemment, il n'est pas possible de tracer TOUS ces vecteurs ; on en tracera un certain nombre bien répartis dans le plan). Si on superpose le tracé d'une trajectoire du système différentiel à ce dessin, on verra qu'une trajectoire « suit les flèches » du champ de vecteurs.
Pour tracer un champ de vecteurs on utilise la commande `plt.streamplot` de Matplotlib. Il faut générer un « quadrillage » du plan qui définira les points où on trace un vecteur.

HP : la commande `np.meshgrid`
Soient `absc=[a,b,c,d]` une liste d'abscisses, et `ordo=[e,f,g,h]` une liste d'ordonnées.
La commande `np.meshgrid(absc,ordo)` désigne le produit cartésien de ces deux listes, qui s'identifie à la matrice

$$\begin{pmatrix} (a,e) & (b,e) & (c,e) & (d,e) \\ (a,f) & (b,f) & (c,f) & (d,f) \\ (a,g) & (b,g) & (c,g) & (d,g) \\ (a,h) & (b,h) & (c,h) & (d,h) \end{pmatrix}$$

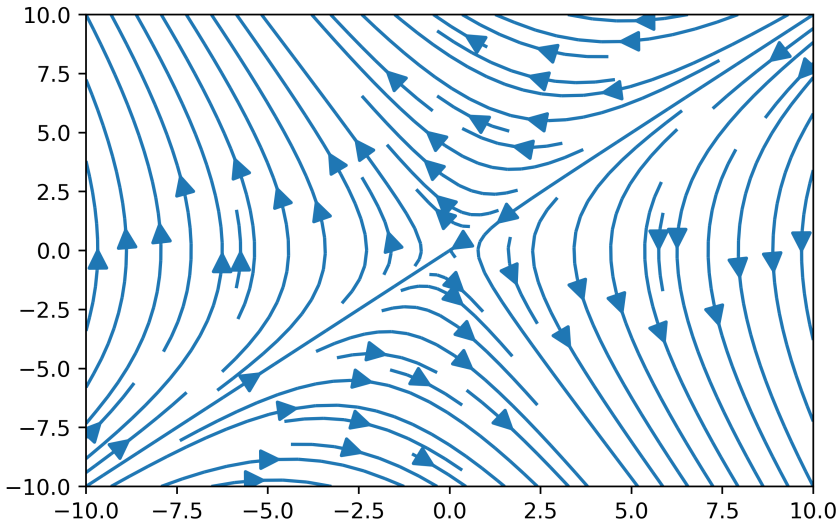
Elle permet ensuite de séparer les abscisses et les ordonnées de ce nouvel objet : si on tape `X,Y=np.meshgrid(absc,ordo)`, les X et Y renvoyés sont :

$$\mathrm{X} = \begin{pmatrix} a & b & c & d \\ a & b & c & d \\ a & b & c & d \\ a & b & c & d \end{pmatrix} \quad \text{et} \quad \mathrm{Y} = \begin{pmatrix} e & e & e & e \\ f & f & f & f \\ g & g & g & g \\ h & h & h & h \end{pmatrix}$$

Le code suivant utilise `np.meshgrid` et permet le tracé du champ de vecteurs associé à la matrice $\mathrm{A} = \begin{pmatrix} 0 & -1 \\ -2 & 1 \end{pmatrix}$ sur le rectangle $[-10,10] \times [-10,10]$.

```
absc=np.linspace(-10,10,100)
ordo=np.linspace(-10,10,100)
X,Y=np.meshgrid(absc,ordo)
plt.streamplot(X, Y,-Y,-2*X+Y,arrowsize=2)

plt.show()
```

On obtient le schéma suivant :



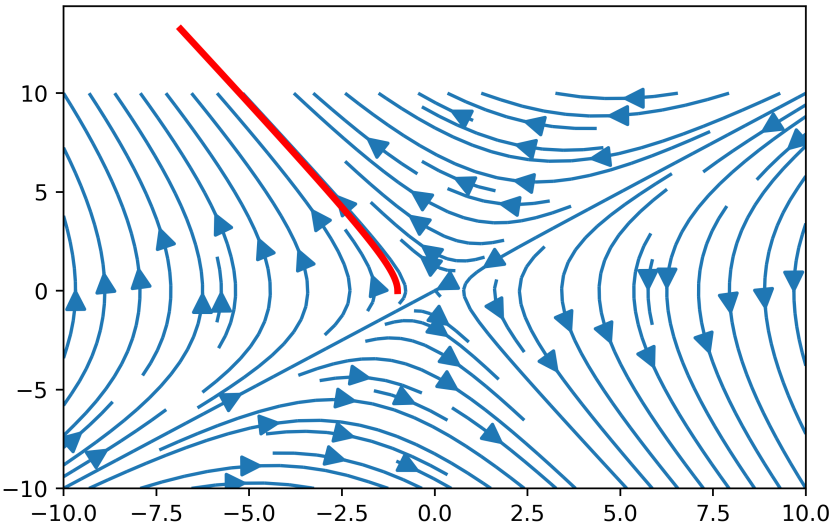
sur lequel on note des « directions de divergence » (coins haut-gauche et bas-droit) et des « directions de convergence » (coins haut-droite et bas-gauche). Nous préciserons cela plus tard.

Superposons maintenant une trajectoire : on demande par exemple $x(0) = -1$, $y(0) = 0$. On trace alors sur le même dessin la trajectoire correspondante, pour $t \in [0,3/2]$[1] :

[1] Valeur choisie pour ne pas trop déborder : avec des exponentielles divergentes, ça part vite !

```
A = np.array( .... )
# la matrice du système X'=AX

def fonc(X,t):
    return np.dot(A,X)

T = np.linspace( ... )

sol = odeint(fonc, ... ,T)
# les ... contiennent une condition initiale de la forme [x(t0),y(t0)]

x = ...
y = ...
# récupération des x(ti) et y(ti), cf exo 1

plt.plot(... , ... ,linewidth=3,color='red')
# tracé en trait épais pour la visibilité

absc=np.linspace(-10,10,100)
ordo=np.linspace(-10,10,100)
X,Y=np.meshgrid(absc,ordo)
plt.streamplot(X, Y,A[0,0]*X+A[0,1]*Y,A[1,0]*X+A[1,1]*Y,arrowsize=2)
# superposition du champ de vecteurs

plt.show()
# affichage
```

et on obtient (ça déborde un peu)



Exercice 7. Reproduire cette expérience avec d'autres conditions initiales (et éventuellement en allongeant ou en rétrécissant l'intervalle défini par le linspace T pour avoir une trajectoire bien proportionnée).
Tester notamment le cas d'une condition initiale appartenant à un sous-espace propre. Qu'observe-t-on ?

Exercice 8. Reprendre l'exercice pour les matrices $A_2, ..., A_5$. On visualisera les faits suivants :

- Avec A_2 toute trajectoire obtenue tend vers $(0,0)$: on visualise en effet que toutes les flèches ramènent à l'origine.
- Avec A_3, toute trajectoire non nulle diverge (les flèches montrent que toute trajectoire qui n'est pas $(0,0)$ explose).
- Avec A_4, la trajectoire converge ssi la condition initiale est dans $\mathrm{Ker}(A_3)$ (et dans ce cas la trajectoire est en fait constante !). Sinon, elle diverge.

Les points d'équilibre sont Vect $\left(\begin{pmatrix}-1\\2\end{pmatrix}\right)$: ils se représentent donc par la droite d'équation $y=-2x$ (à superposer sur le champ de vecteurs). On remarque alors que les flèches s'éloignent de l'équilibre (on parle d'équilibre instable).

- Avec A_5, toute trajectoire converge vers un point d'équilibre.
  Les points d'équilibre sont cette fois Vect $\left(\begin{pmatrix}1\\1\end{pmatrix}\right)$; donc la droite $y=x$. Cette fois les flèches se dirigent vers l'équilibre (on parle d'équilibre stable).

Et dans le cas non diagonalisable ?? Pas de chance c'est celui qui donne les plus jolies figures. Tester par exemple les systèmes $\begin{cases}x'=2x+3y\\y'=-4x-4y\end{cases}$, $\begin{cases}x'=x-5y\\y'=2x+3y\end{cases}$, $\begin{cases}x'=-y\\y'=x\end{cases}$ (pas de valeur propre réelle). Mais pour les résoudre il faudrait faire de la trigonométrie et des nombres complexes...