AUTOUR DES ÉQUATIONS DIFFÉRENTIELLES ORDINAIRES

Toutes les parties sont totalement indépendantes, et constituent des petits problèmes. À l'intérieur d'une partie, les exercices dépendent les uns des autres et doivent être faits dans l'ordre.

Et, parce qu'il faut bien s'amuser un peu en ces temps de *confinage*, certains exercices utiliseront des fonctions inhabituelles de la librairie `matplotlib`. Elles ne sont évidemment pas à connaître.

§1. Vitesse de convergence : Euler vs Heun

On rappelle ci-dessous deux algorithmes pour résoudre de manière approchée le problème de Cauchy

$$\begin{cases} y'(t) = \mathrm{F}(y(t), t) & \text{pour } t \text{ entre } t_0 \text{ et } t_{\max}, \\ y(t_0) = y_0. \end{cases}$$

Le premier des deux correspond à la formule d'itération d'Euler $y_{n+1} = y_n + h_n \times \mathrm{F}(y_n, t_n)$ qui est à connaître.

```
def Euler(F, y0, t0, tmax, N) :
    h = (tmax - t0) / N
    y = y0 ; Y = [y0]
    t = t0 ; T = [t0]
    for n in range(N) :
        y += h * F(y, t) ; Y.append(y)
        t += h ; T.append(t)
    return (T, Y)
```

```
def Heun(F, y0, t0, tmax, N) :
    h = (tmax - t0) / N
    y = y0 ; Y = [y0]
    t = t0 ; T = [t0]
    for n in range(N) :
        y_ = y + h * F(y, t)
        y += h * (F(y, t) + F(y_, t + h)) / 2
        Y.append(y)
        t += h ; T.append(t)
    return (T, Y)
```

On rappelle que ces deux programmes renvoient deux listes : `T = [`t_0`, `t_1`, ..., `t_N`]` où $t_n = t_0 + nh$, avec $h = (t_{\max} - t_0)/\mathrm{N}$, de sorte que $t_\mathrm{N} = t_{\max}$, et `Y = [`y_0`, ..., `y_N`]`, où y_n est censé être une valeur approchée de $y(t_n)$ pour y la solution de l'équation différentielle.

Exercice 1

1) Vérifier que $f(t) = \exp(-t^2/2)$ est solution du problème de Cauchy

$$\begin{cases} y'(t) = -t \times y(t), \\ y(0) = 1. \end{cases}$$

2) Qui est la fonction F dans ce cas ? Qui sont t_0 et y_0 ?

3) Que faut-il écrire (en Python) pour appliquer l'un des deux programmes ci-dessous à la résolution de cette équation différentielle particulière ?

Exercice 2 — Définissons la fonction de l'exercice précédent, à des fins expérimentales.

```
from numpy import exp
def f(t) :
    return exp(-t**2 / 2)
```

On donne ensuite le programme ci-dessous.

```
# Calcul de l'erreur en tmax
def Erreur(tmax, N, Méthode) :
    (T, Y) = Méthode(F, y0, t0, tmax, N)
    return f(tmax) - Y[-1]
```

1) Quelles « valeurs » peuvent prendre l'argument `Méthode` de ce programme ?

2) Expliquer pourquoi `Y[-1]`.

Exercice 3 — On donne le programme suivant.

```
from matplotlib.pyplot import figure
from numpy import log as ln

def NuageErreur(tmax, Méthode) :

    fig = figure("Erreur avec le programme " + Méthode.__name__)

    # Erreur en fonction de N
    abscisses = [ 100, 200, 500, 1000, 2000, 5000 ]
    ordonnées = [ Erreur(tmax, N, Méthode) for N in abscisses]
    Normal = fig.add_subplot(1, 2, 1)
    Normal.grid(True)
    Normal.scatter(abscisses, ordonnées)

    # ln(abs(Erreur)) en fonction de ln(N)
    abscisses = [ ln(N) for N in abscisses ]
    ordonnées = [ ln(abs(err)) for err in ordonnées ]
    Logarithmique = fig.add_subplot(1, 2, 2)
    Logarithmique.grid(True)
    Logarithmique.scatter(abscisses, ordonnées)

    fig.show()
```

On est curieux : on teste...

```
>>> NuageErreur(3, Euler)
>>> NuageErreur(3, Heun)
```

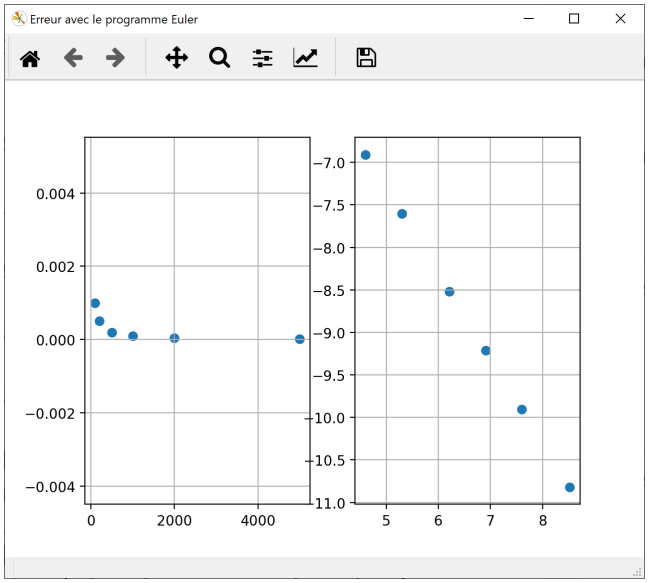
...et on obtient les graphiques à la suite de l'exercice.

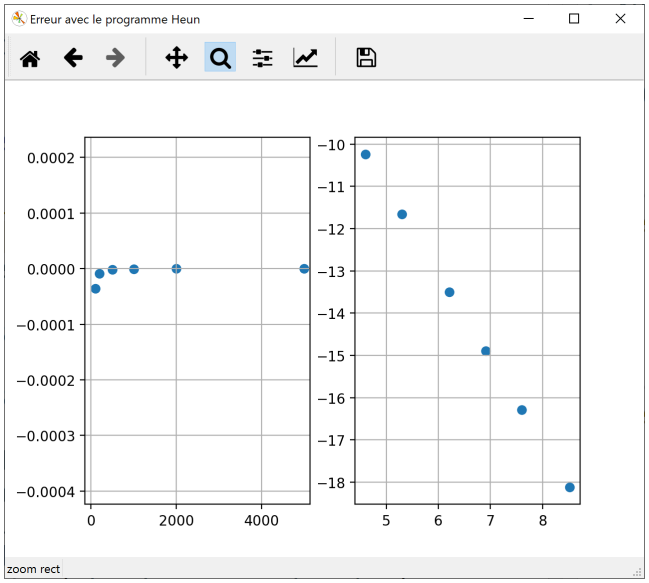
1) On note E(N) l'erreur commise en $t_{\max}$, c'est-à-dire $y(t_N) - y_N$. Le graphique en échelle logarithmique semble être à chaque fois une droite, c'est-à-dire qu'on a $\ln(|E(N)|) \simeq \beta - \alpha \ln(N)$ pour certaines constantes α et β. Qu'est-ce que cela nous apprend sur la forme de E(N) ?

2) Lire sur les graphiques les valeurs de α.

3) On dit qu'une méthode est d'ordre α si l'erreur commise en un t fixé est divisée par μ^α lorsque N est multiplié par μ. Quelle est l'ordre de la méthode d'Euler ? Et celui de la méthode de Heun ?

4) Si plusieurs méthodes ont des ordres α différents, quelle est celle qui converge le plus rapidement (c'est-à-dire qui nécessite la valeur la moins élevée de N pour avoir des résultats convenables) ? De celles d'Euler et Heun, laquelle est la meilleure ?





Exercice 4 — On donne pour finir ce petit programme.

```
from numpy import polyfit
def RégressionAffine(abscisses, ordonnées) :
    éq = polyfit(abscisses, ordonnées, 1)
    print("y = mx + p")
    print("m =", éq[0])
    print("p =", éq[1])
```

1) À quoi sert-il ? On décrira ce qu'il prend en argument, et le résultat qu'il produit/ce qu'il renvoie.

2) Écrire un programme `ModélisationErreur(tmax, Méthode)` qui permet d'obtenir des constantes α et K tels que $|\mathrm{E(N)}| \simeq \mathrm{K/N^{\alpha}}$, pour une `Méthode` donnée de résolution approchée.

Une remarque : on peut vérifier en changeant d'équation différentielle que l'ordre d'une méthode ne dépend *que* de N. On peut constater expérimentalement une dépendance en $t_{\max}$. Elle est trompeuse : en fait on a un équivalent $|\mathrm{E(N)}| \sim \mathrm{K/N^{\alpha}}$ pour $\mathrm{N} \to +\infty$, et cet équivant est moins rapidement précis lorsque $t_{\max}$ s'éloigne de t_0. Moins rapidement précis, mais toujours valable.

§2. Résolution d'une édo avec pas variable et abscisses obligées

On considère le même problème de Cauchy

$$\begin{cases} y'(t) = \mathrm{F}(y(t), t) & \text{pour } t \text{ entre } t_0 \text{ et } t_{\max}, \\ y(t_0) = y_0, \end{cases}$$

mais cette fois-ci les abscisses t_0, t_1, …, $t_{\mathrm{N}} = t_{\max}$ sont données, et le pas $h_n = t_{n+1} - t_n$ n'est pas forcément constant. On cherche toujours des valeurs approchées y_n de $y(t_n)$ où y est la solution de l'équation différentielle.

Exercice 5 — Voici ce que devient le programme qui met en œuvre la méthode d'Euler.

```
def Euler(F, y0, T) :
    Y = [y0] ; y = y0
    for n in range(1, len(T)) :
```

```
        h = T[n] - T[n - 1]
        y += h * F(y, T[n - 1])
        Y.append(y)
    return Y
```

1) Qu'est-ce que le `T` ci-dessus ?

2) Si on connaît t_0, $t_{\max}$ et N, comment construire (en Python) le `T` qui correspond à un pas constant ? Par exemple avec un programme `Subdivision(a, b, N)` qu'on explicitera.

Exercice 6

1) Combien y a-t-il d'éléments dans la liste produite par `Subdivision(a, b, N)` ci-dessus ?

2) Écrire une variante `SubdivisionSansBouts(a, b, N)` qui renvoie la même liste, sans le point a et le point b. On suppose que N est au moins égal à deux.

Dans la suite, voici notre grande préoccupation : on veut construire une liste `T` contenant $\mathrm{N}+1$ valeurs, la première étant a, la dernière étant b. Pour fixer les idées, on va supposer que $a < b$. On donne, c'est la nouveauté, une liste `V`, rangée dans l'ordre croissant strict, de nombres dans $]a\,;b[$. Et on veut que la subdivision `T` (qui sera donc croissante) contiennent obligatoirement les éléments de `V`. On notera r leur nombre : $r =$ `len(V)`.

Exercice 7 — On donne le programme suivant.

```
from random import randint as EntAléa
def RépartirParDéfaut(a, b, V, N) :
    r = len(V) ; Nombres = [0] * (r + 1)
    # Premier morceau
    Nombres[0] = int((V[0] - a) / (b - a) * (N - r - 1))
    # Morceaux intermédiaires
    for i in range(1, r) :
        Nombres[i] = int((V[i] - V[i - 1]) / (b - a) * (N - r - 1))
    # Dernier morceau
    Nombres[r] = int((b - V[r - 1]) / (b - a) * (N - r - 1))
    return Nombres
```

1) Rappeler ce qu'est `int(x)` pour un réel positif x.

2) Hormis a, b et les éléments de `V`, il y aura donc $\mathrm{N}-r-1$ valeurs dans la subdivision `T`. Si `Nombres[0]` et `Nombres[r]` représentent respectivement le nombre de points (de cette subdivision) à choisir dans $]a\,;v_0[$ et $]v_{r-1}\,;b[$, et si `Nombres[i]` est le nombre qu'il y en a dans $]v_{i-1}\,;v_i[$ pour $1 \leqslant i \leqslant r-1$, justifier que l'algorithme ci-dessus produit une répartition à peu près équitable (regarder le nombre de points dans chaque sous-intervalle par rapport à la longueur de celui-ci).

3) Justifier aussi qu'il n'y a pas *trop* de points, c'est-à-dire que

$$\sum_{i=0}^{r} \texttt{Nombres[}i\texttt{]} \leqslant \mathrm{N}-r-1.$$

4) Écrire un programme `AjouterCeuxQuiManquent(V, N, Nombres)` qui ne renvoie rien, mais modifie la liste `Nombres` ci-dessus pour que la somme de ses éléments soit exactement $\mathrm{N}-r-1$. On répartira aléatoirement les éléments manquant dans les différentes « boîtes », en utilisant par exemple le programme `EntAléa` importé ci-dessus.

5) En déduire un programme `SubdivisionAvecObligés(a, b, N, V)` qui résout le problème posé.

Exercice 8 — Déduire de tout ce qui précède un (très court) programme `RésolutionAvecObligés(Méthode, F, y0, t0, tmax, N, V)`, où `Méthode` est par exemple le programme `Euler(F, y0, T)`, qui résout l'équation différentielle de manière approchée, avec N pas, pas forcément réguliers, mais avec les valeurs de la liste `V` figurant obligatoirement dans la subdivision. Il renverra le couple `(T, Y)` correspondant.

§3. Interpolation et courbes de Bézier

Exercice 9 — Coefficients binomiaux

1) Écrire un programme `LigneSuivante(L)` qui prend en argument une liste correspondant à une ligne du triangle de Pascal, et qui construit et renvoie la liste `S` correspondant à la ligne suivante.

2) En déduire un programme `Binomiaux(n)` qui renvoie la liste d'ordre n du triangle de Pascal. Celle qui contient les « k parmi n », pour k allant de 0 à n.

Exercice 10 — Courbes de Bézier. Alors Bézier, c'est pas la ville (qui ne s'écrit pas comme ça), c'est Pierre ; et Pierre, à part avoir été à Supélec, il a imaginé ceci :

$$\overrightarrow{\mathrm{OP}(t)} = \sum_{k=0}^{n} \binom{n}{k} t^k (1-t)^{n-k} \times \overrightarrow{\mathrm{OP}_k},$$

où O est l'origine du repère et les P_k sont $n+1$ points donnés. Cette équation définit un unique point $\mathrm{P}(t) = (x(t), y(t))$, et ceux qui préfèrent les coordonnées aux vecteurs pourront écrire

$$x(t) = \sum_{k=0}^{n} \binom{n}{k} t^k (1-t)^{n-k} \times x_k,$$

et la même chose avec $y(t)$ et les y_k, où $\mathrm{P}_k = (x_k, y_k)$.

1) Écrire le programme `Bézier(Points)` qui fabrique et renvoie le programme `P(t)` calculant le point $(x(t), y(t))$ défini ci-dessus.

On donne ensuite ce programme, pour *voir* ce qu'est une courbe de Bézier.

```
from numpy import linspace
from matplotlib.pyplot import axis, grid, plot, show

ROUGE = (1, 0, 0) ; BLEU  = (0, 0, 1)
def AfficherBézier(Points) :

    # Courbe
    P = Bézier(Points) ; T = linspace(0, 1, 1000)
    abscisses = [] ; ordonnées = []
    for t in T :
        (x, y) = P(t)
        abscisses.append(x) ; ordonnées.append(y)
    plot(abscisses, ordonnées, color = BLEU)

    # Ligne brisée des points de contrôle
    abscisses = [] ; ordonnées = []
    for (x, y) in Points :
        abscisses.append(x) ; ordonnées.append(y)
    plot(abscisses, ordonnées, ":", color = ROUGE)
    axis("equal") ; grid(True) ; show()
```

2) À quoi sert l'argument `":"` dans la fonction `plot` ?

3) Tester, par exemple avec ceci.

```
>>> AfficherBézier([(0, 0), (1, -1), (2, 0)])
>>> AfficherBézier([(0, 0), (2, -1), (2, 1), (4, 0)])
>>> AfficherBézier([(0, 0), (1, -1), (2, -1), (4, 1), (4, 0)])
```

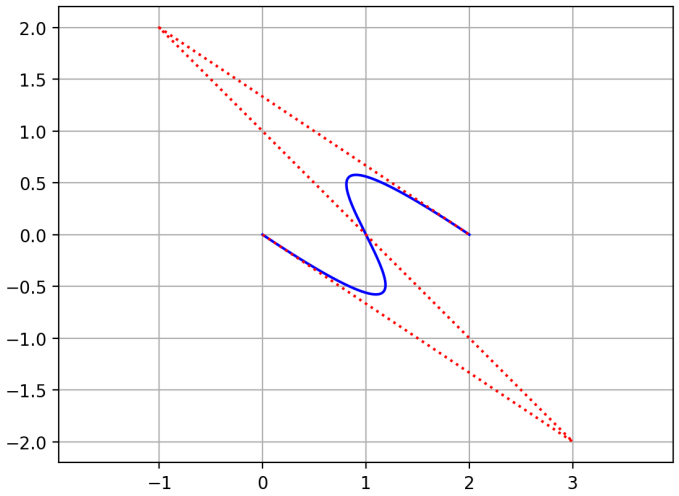
4) Les P_k sont appelés les *points de contrôle*. Est-ce que la courbe $\{\mathrm{P}(t)\ 0 \leqslant t \leqslant 1\}$ passe par ces points de contrôle ?

5) Que peut-on dire du premier et du dernier tronçon de la ligne brisée (c'est-à-dire des droites (P_0P_1) et $(P_{n-1}P_n)$) ?

Exercice 11 — On donne deux points $A(x_A\,;y_A)$ et $B(x_B\,;y_B)$ et on considère une courbe de Bézier démarrant à A et terminant à B. On suppose pour fixer les idées que $x_A < x_B$, et on prend $x \in [x_A\,;x_B]$.

1) Pourquoi est-on sûr qu'il existe $t \in [0\,;1]$ tel que $P(t) = (x, y)$ avec le x précédent ? Remarque : en revanche il se peut que ce t ne soit pas unique, comme le montre l'exemple ci-après.

```
>>> AfficherBézier([(0, 0), (3, -2), (-1, 2), (2, 0)])
```



2) Écrire un programme `Trouver_y(xA, xB, x, P)` qui étant données les abscisses en jeu et la fonction $P(t)$ renvoie l'un des y possibles (pour ceux qui ne voient pas comment faire la réponse commence par *dicho* et finit par *tomie*).

Exercice 12 — Interpolation. La méthode d'Euler (ou de Heun) c'est bien, mais ça vous fabrique un *nuage de points*, pas une *fonction*. En particulier, si on veut lire l'image d'un point qui n'est pas dans la subdivision choisie on est embêté. Alors voici un programme

```
def Interpoler(T, Y, Yp) :

    # Calcul des morceaux
    Morceaux = []
    for i in range(len(T) - 1) :
        A = (T[i], Y[i])
        B = (T[i + 1], Y[i + 1])
        if abs(Yp[i + 1] - Yp[i]) < 1e-5 :
            h = (T[i + 1] - T[i]) / 2
            I = (T[i] + h, Y[i] + h * Yp[i])
            J = (T[i + 1] - h, Y[i + 1] - h * Yp[i + 1])
            Morceaux.append( Bézier([A, I, J, B]) )
        else :
            I = Intersection(A[0], A[1], Yp[i], B[0], B[1], Yp[i + 1])
            Morceaux.append( Bézier([A, I, B]) )

    # Fabrication de la fonction interpolatrice
    def f(t) :
        i = Recherche(T, t)
        return Trouver_y(T[i], T[i+1], t, Morceaux[i])

    return f
```

qui part du nuage et en fait une fonction.

1) On part de la liste `T = [`t_0`, `t_1`, ..., `t_N`]` contenant les points de la subdivision utilisée. On la suppose rangée dans l'ordre croissant. Écrire le programme `Recherche(T, t)` qui étant donné $t \in [t_0\,;t_N[$ détermine l'indice i tel que $t \in [t_i\,;t_{i+1}[$.

2) Que devient une courbe de Bézier si elle n'a que deux points de contrôle ?

3) On rappelle que si deux droites on des coeffients directeurs distincts, alors elle se coupent en un point et un seul. Écrire le programme `Intersection(xA, yA, mA, xB, yB, mB)` qui étant données deux $(\mathrm{A}, m_\mathrm{A})$ et $(\mathrm{B}, m_\mathrm{B})$ représentées par un point et leur coefficient directeur, calcule et renvoie les coordonnées $(x_\mathrm{I}, y_\mathrm{I})$ de leur intersection. On suppose que $m_\mathrm{A} \neq m_\mathrm{B}$.

4) Que représente la liste `Yp` passée en argument de ce programme ?

Exercice 13 — On donne les deux programmes suivants.

```
def EulerAvecDérivée(F, y0, t0, tmax, N) :
    h = (tmax - t0) / N
    y = y0 ; Y = [y0] ; Yp = []
    t = t0 ; T = [t0]
    for n in range(N) :
        dy = F(y, t)
        y += h * dy ; Y.append(y) ; Yp.append(dy)
        t += h ; T.append(t)
    Yp.append(F(y, t))
    return (T, Y, Yp)
```

```
def HeunAvecDérivée(F, y0, t0, tmax, N) :
    h = (tmax - t0) / N
    y = y0 ; Y = [y0] ; Yp = []
    t = t0 ; T = [t0]
    for n in range(N) :
        dy = F(y, t)
        y_ = y + h * dy
        y += h * (dy + F(y_, t + h)) / 2 ;
        Y.append(y) ; Yp.append(dy)
        t += h ; T.append(t)
    Yp.append(F(y, t))
    return (T, Y, Yp)
```

On pourrait, pour chacun d'eux, se contenter de la version habituelle et écrire à la fin `Yp = [F(Y[n], T[n]) for n in range(0, N + 1)]`. Expliquer pourquoi ce serait moins efficace.

Exercice 14 — Tout ce qui précède pour en arriver à ce programme.

```
from matplotlib.pyplot import figure
def RésultatInterpolation(F, y0, t0, tmax, N) :

    fig = figure("Interpolation")

    # Nuage de points avec interpolation affine par morceaux
    (T, Y, Yp) = HeunAvecDérivée(F, y0, t0, tmax, N)
    IAffinePM = fig.add_subplot(1, 2, 1)
    IAffinePM.grid(True) ; IAffinePM.plot(T, Y)

    # Interpolation façon Bézier
    f = Interpoler(T, Y, Yp)
    abscisses = linspace(t0, tmax, 1000)
```

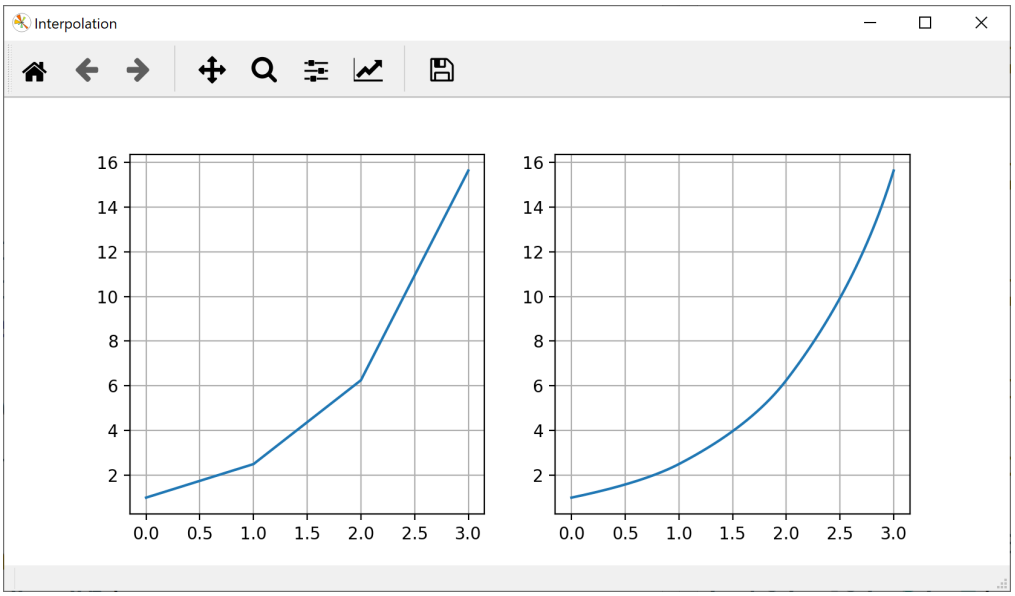
```
    ordonnées = [ f(t) for t in abscisses ]
    IBézier = fig.add_subplot(1, 2, 2)
    IBézier.grid(True) ; IBézier.plot(abscisses, ordonnées)

    fig.show()
```

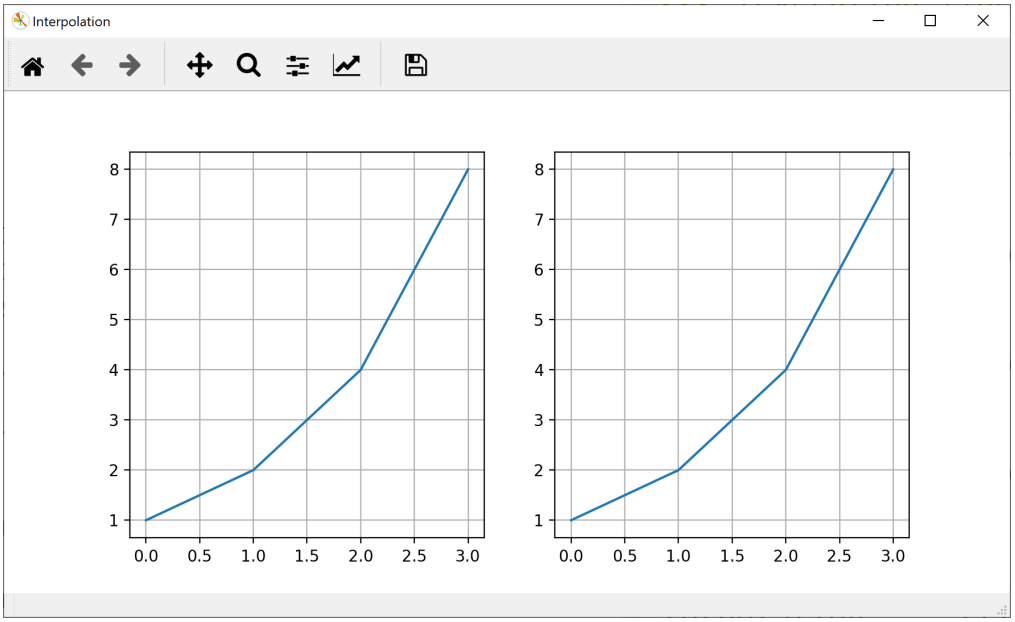
Testons-le, juste pour bien voir l'effet, avec une toute petite valeur de N, par exemple N = 2 ou N = 3. On utilisera l'équation différe tielle de la fonction exponentielle, dont on connaît bien la solution.

```
t0 = 0 ; y0 = 1
def F(x, t) :
    return x
```

```
>>> RésultatInterpolation(F, y0, t0, 3, 3)
```



1) Comment est obtenue la courbe de l'interpolation affine par morceaux (alors qu'on n'a écrit aucun programme qui la calcule) ?
2) Question subtile mais essentielle : si on a bien compris ce qu'est la méthode d'Euler, la réponse est évidente. On donne ci-dessous le résultat obtenu par le même calcul, mais en remplaçant dans le programme ci-dessous `HeunAvecDérivée` par `EulerAvecDérivée`. On voit que l'interpolation par courbe de Bézier donne la même chose que l'interpolation affine par morceaux. Expliquer pourquoi.



§4. Méthode d'Euler implicite, cas scalaire : Picard vs Newton

Jusque là on a parlé de méthode d'Euler, sans préciser « explicite » (on aurait pu, c'est son nom complet), qui correspond à la formule d'itération $y_{n+1} = y_n + h_n \times \mathrm{F}(y_n, t_n)$. La *méthode d'Euler implicite* utilise la valeur de la dérivée non pas au point où l'on est, mais au point qu'on calcule : elle correspond à la formule d'itération

$$y_{n+1} = y_n + h_n \times \mathrm{F}(y_{n+1}, t_{n+1}).$$

Elle s'appelle *implicite* parce qu'elle ne donne pas de formule pour y_{n+1} ; pour le trouver il faut résoudre l'équation

$$y_n + h_n \times \mathrm{F}(x, t_{n+1}) - x = 0,$$

d'inconnue x. Avec une information importante : si le pas est petit, y_{n+1} n'est sans doute pas très différent de y_n, donc y_n est une bonne estimation de la solution.

Exercice 15 — La méthode de Picard. Soit $f : \mathbf{R} \to \mathbf{R}$ une fonction *strictement contractante*, c'est-à-dire K-lipschitzienne pour un $\mathrm{K} < 1$. On définit une suite par son premier terme $x_0 \in \mathbf{R}$ et la formule de récurrence $x_{n+1} = f(x_n)$.

1) Montrer que l'équation $f(x) = x$ possède au plus une solution.

2) Soient m et n deux entiers, avec $m \leqslant n$. Prouver que

$$|x_n - x_m| \leqslant |f(x_0) - x_0| \times \frac{\mathrm{K}^m}{1 - \mathrm{K}}.$$

3) En déduire que la suite $(x_n)_{n\geqslant 0}$ est bornée. D'après le théorème de Bolzano et Weierstrass, cela implique l'existence d'une sous-suite convergeant vers un $\ell \in \mathbf{R}$.

4) En découpant $|\ell - x_n| = |\ell - x_{\varphi(n)} + x_{\varphi(n)} - x_n|$, prouver qu'en fait $x_n \longrightarrow \ell$.

5) Justifier que ℓ est une solution de l'équation $f(x) = x$.

6) Écrire un programme `RésoudrePicard(f, x_approché)` qui calcule cette solution. On renverra le x_{n+1} qui correspond à la condition d'arrêt $|x_{n+1} - x_n| \leqslant 10^{-10}$.

7) Qui est, dans le cas particulier de l'équation en préambule, la fonction f ?

Exercice 16 — Une remarque importante : on suppose, à un t_{n+1} fixé, que la fonction $x \mapsto \mathrm{F}(x, t_{n+1})$ est K-lipschitzienne. Soit $f(x) = y_n + h_n \times \mathrm{F}(x, t_{n+1})$. Est-ce que les hypothèses de l'exercice précédent s'appliquent ? Si oui, cela veut dire que la formule d'itération d'Euler implicite définit correctement y_{n+1}, donc il y a intérêt à ce que la réponse soit « oui ».

Exercice 17 — La méthode de Newton. On considère une équation écrite sous la forme $g(x) = 0$. On rappelle que partant d'une estimation $x_0 = x_{\text{approché}}$ de la solution, l'itération de Newton

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

converge (sous des hypothèses raisonnables) vers une solution.

1) Écrire un programme `Dérivée(g, x, h = 1e-7)` qui renvoie une valeur approchée de $g'(x)$ en utilisant un taux d'accroissement symétrisé de pas $\pm h$.

2) En déduire le programme `RésoudreNewton(g, x_approché)`. On renverra le x_{n+1} qui correspond à la condition d'arrêt $|x_{n+1} - x_n| \leqslant 10^{-10}$.

3) Qui est, dans le cas particulier de l'équation en préambule, la fonction g ?

Exercice 18 — Il est temps de la programmer, cette méthode d'Euler implicite.

```
def EulerImplicite(F, y0, t0, tmax, N) :
    h = (tmax - t0) / N
    y = y0 ; Y = [y0]
```

```
    t = t0 ; T = [t0]
    for n in range(N) :
        def f(x) :
            return ...
        y = ... ; Y.append(y)
        t += h ; T.append(t)
    return (T, Y)
```

1) Compléter le programme ci-dessus, en utilisant `RésoudrePicard`.

2) Reprendre la question, en changeant `f(x)` en `g(x)` dans le programme ci-dessus, et `RésoudrePicard` par `RésoudreNewton`.

Exercice 19 — Un exemple. On considère l'équation différentielle $ty'(t) - \cos(y(t)) = 0$, avec une condition initiale de la forme $y(0) = p$ pour $p \in \mathbf{R}$.

1) Expliquer pourquoi la méthode d'Euler explicite ne s'applique pas.

2) Écrire un programme `Solution(p)` qui dessine, sur un même graphique, la courbe de la solution sur $[0\,;\,1]$ et sur $[-1\,;\,0]$.

```
from numpy import cos
from matplotlib.pyplot import grid, plot, show
VERT = (0.1, 0.7, 0.4)
```

3) Est-ce que les deux « morceaux » se recollent de manière $\mathscr{C}^1$, autrement l'équation a-t-elle des solutions sur $\mathbf{R}$?

§5. Méthode d'Euler implicite, cas vectoriel : fonctions de Bessel

Les fonctions de Bessel J_n sont des solutions de l'équation différentielle

$$t^2y''(t) + ty'(t) + (t^2 - n^2)y(t) = 0,$$

avec certaines conditions initiales dont on parlera plus tard. Cette équation différentielle est d'ordre deux, on a donc besoin d'une méthode *vectorielle* pour la résoudre.

Exercice 20 — Où l'on se ramène à une équation vectorielle d'ordre un. Soit y une solution du problème précédent. On pose

$$\mathrm{Y}(t) = \begin{bmatrix} y(t) \\ y'(t) \end{bmatrix}$$

qui définit une fonction $\mathbf{R} \to \mathbf{R}^2$, de classe $\mathscr{C}^1$ si y est de classe $\mathscr{C}^2$.

1) Définir une fonction $\mathrm{F}_n : \mathbf{R}^2 \times \mathbf{R} \to \mathbf{R}^2$ telle que pour tout $t \in \mathbf{R}$ on a $\mathrm{Y}'(t) = \mathrm{F}_n(\mathrm{Y}(t), t)$.

2) Compléter le code ci-dessous qui définit cette fonction en Python. On suppose que X est une matrice-colonne et on rappelle qu'on accède à sa composante i en écrivant `X[i, 0]`. Les indices commencent à zéro.

```
from numpy import matrix, zeros
def Fn(X, t) :
    return matrix([
     [...],
     [...]])
```

Apportons quelques précisions sur la bibliothèque `numpy`, utilisée dans l'exercice précédent et tout au long de cette partie. On récupère la taille d'une matrice A en écrivant `(m, n) = A.shape`. Pour créer une matrice de zéros, on écrit `A = matrix(zeros((m, n)))`, en faisant attention à la double-paire de parenthèses. Les opérations matricielles s'écrivent comme on l'imagine, à supposer que les tailles sont compatibles : `A + B`, `A - B`, `A * B`, `t * A`, `A ** (-1)`, etc.

Exercice 21 — Dérivées des fonctions de plusieurs variables. Si $f : \mathbf{R}^d \to \mathbf{R}^d$ est une fonction de classe $\mathscr{C}^1$, on peut calculer sa dérivée en $\mathrm{X} \in \mathbf{R}^d$ dans la direction $\mathrm{V} \in \mathbf{R}^d$ avec l'approximation

$$(\partial_{\mathrm{V}} f)(\mathrm{X}) \simeq \frac{f(\mathrm{X}+h\mathrm{V}) - f(\mathrm{X}-h\mathrm{V})}{2h} \in \mathbf{R}^d.$$

Si V est le j-ème vecteur U_j de la base usuelle de $\mathbf{R}^d$, on note simplement $\partial_j f$ cette dérivée. Enfin on pose

$$f'(\mathrm{X}) = \begin{bmatrix}(\partial_0 f)(\mathrm{X}) & (\partial_1 f)(\mathrm{X}) & \dots & (\partial_{d-1} f)(\mathrm{X})\end{bmatrix} \in \mathscr{M}_d(\mathbf{R}),$$

obtenue en plaçant côte à côte les matrices-colonnes des dérivées dans les directions principales, et on l'appellera (sans doute abusivement) la dérivée de f en X.

1) Écrire un programme `Unitaire(d, j)` qui construit et renvoie la matrice-colonne U_j (des zéros partout, sauf sur la ligne j).

2) Écrire le programme `DérivéeDirectionnelle(f, X, V, h = 1e-7)`.

3) En déduire le programme `Dérivée(f, X, h = 1e-7)` qui construit et renvoie la matrice $f'(\mathrm{X})$ (en tout cas son approximation).

Exercice 22 — Méthode de Newton pour les systèmes d'équations. On donne une fonction $g : \mathbf{R}^d \to \mathbf{R}^d$, et on cherche un vecteur $\mathrm{X} \in \mathbf{R}^d$ tel que $g(\mathrm{X}) = \overrightarrow{0}$. On utilise la méthode de Newton, qui part d'une estimation $\mathrm{X}_0 = \mathrm{X}_{\text{approché}}$ d'une solution, et qui utilise la formule d'itération

$$\mathrm{X}_{n+1} = \mathrm{X}_n - g'(\mathrm{X}_n)^{-1} g(\mathrm{X}_n).$$

On remarquera que c'est la *même* formule que dans le cas scalaire.

1) Écrire un programme `Norme(A)` qui calcule $\|\mathrm{A}\|_\infty = \max_{i,j} |a_{i,j}|$.

2) En déduire le programme `RésoudreNewton(g, X_approché)` qui applique l'itération jusqu'à avoir $\|\mathrm{X}_{n+1} - \mathrm{X}_n\|_\infty \leqslant 10^{-10}$, et on renverra le X_{n+1} correspondant. Pour définir X_0, on fera une *copie* plutôt qu'une affectation simple, en écrivant par exemple `X = matrix(X_approché)`.

Exercice 23 — La méthode d'Euler implicite consiste à utilise la formule d'itération

$$\mathrm{Y}_{n+1} = \mathrm{Y}_n + h_n \times \mathrm{F}(\mathrm{Y}_{n+1}, t_{n+1}),$$

ce qui demande de résoudre l'équation $\mathrm{Y}_n + h_n \times \mathrm{F}(\mathrm{X}, t_{n+1}) = \mathrm{X}$.

1) Mettre cette équation sous la forme $g(\mathrm{X}) = \overrightarrow{0}$, pour une fonction $g : \mathbf{R}^d \to \mathbf{R}^d$ qu'on précisera.

2) Compléter le programme ci-dessous.

```
def EulerImpliciteVectoriel(F, Y0, t0, tmax, N) :
    h = (tmax - t0) / N
    Y = Y0 ; Les_Y = [matrix(Y)]
    t = t0 ; T = [t0]
    for n in range(N) :
        def g(X) :
            return ...
        Y = ... ; Les_Y.append(matrix(Y))
        t += h ; T.append(t)
    return (T, Les_Y)
```

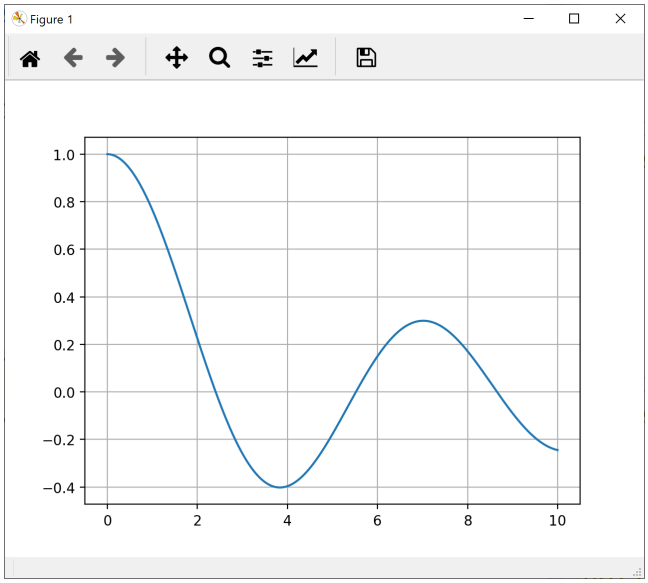
Exercice 24

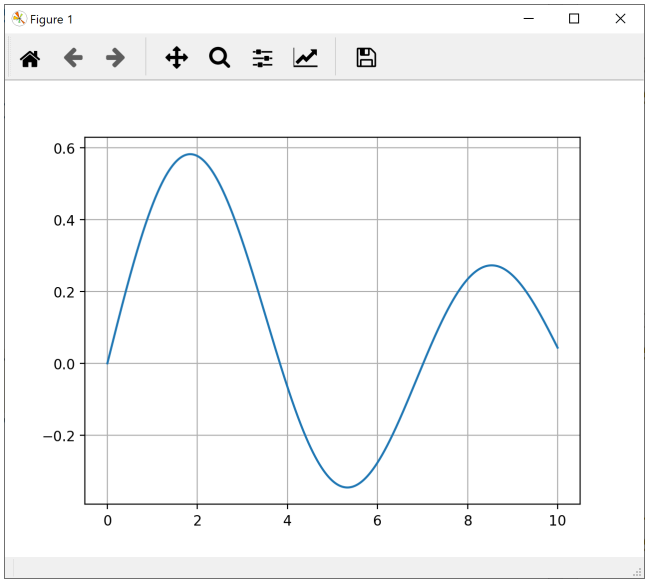
1) Peut-on utiliser la méthode d'Euler explicite pour résoudre l'équation de Bessel ?

2) Écrire un programme `RésoudreBessel(n, y0, yp0, tmax, N)` qui renvoie le couple de listes `(T, J)` avec `T = [`t_0`, ...]` comme d'habitude et `J` la liste des approximations de $\mathrm{J}(t_0)$, $\mathrm{J}(t_1)$, etc.. Les paramètres sont les conditions initiales : $\mathrm{J}(0) = y_0$ et $\mathrm{J}'(0) = y'_0$.

3) On donne $\mathrm{J}_0(0) = 1$ et $\mathrm{J}'_0(0) = 0$. Écrire le code Python qui permet d'obtenir la courbe ci-dessous.

4) Même question avec $\mathrm{J}_1(0) = 0$ et $\mathrm{J}'_1(0) = 1/2$.





Exercice 25 — Zéros des fonctions de Bessel

1) Soit y telle que $y(a)y(b) < 0$. On suppose que a et b sont proches, que y est régulière, et qu'on n'a aucun moyen de connaître les valeurs de y dans $]a\,;\,b[$. Donner la meilleure approximation possible pour un $t \in]a\,;\,b[$ tel que $y(t) = 0$.

2) Écrire un programme `Zéros(T, Y)` qui étant donnés une liste d'abscisses et une liste de valeurs approchées des images correspondantes par une fonction y, renvoie la liste des zéros de y dans l'intervalle ouvert $]t_0\,;\,t_\mathrm{N}[$.

3) En déduire un programme `ZérosBessel(tmax)` qui donne la liste de tous les zéros de J_0 dans l'intervalle $]0\,;\,t_{\max}[$. On en trouve facilement les valeurs en ligne, s'il y a besoin de vérifier.

§6. Un exemple de graphique animé : mouvements de particules

Un *point*. C'est une liste de longueur neuf (en tout cas dans cette partie). Définissons les quelques constantes

```
POS_x    = 0 ; POS_y    = 1
VIT_x    = 2 ; VIT_y    = 3
ACC_x    = 4 ; ACC_y    = 5
COULEUR = 6 ; RÉGLE    = 7 ; T_préc   = 8
```

de sorte que si `P` est un point, `P[POS_x]` est son abscisse, `P[VIT_x]` est la composante en abscisse de sa vitesse, etc. et pour les deux derniers éléments, on verra plus loin.

Les points se meuvent dans l'univers $[0\,;\,1]^2$. Voici une première façon de fabriquer un point.

```
from random import random as Aléa
from time import time as Maintenant
from numpy import sqrt, cos, sin, pi
```

```
def PointFixeAléa(couleur) :
    x = Aléa() ; y = Aléa()
    vx = 0 ; vy = 0 ; ax = 0 ; ay = 0
    def Règle(P) :
        pass
    return [x, y, vx, vy, ax, ay, couleur, Règle, Maintenant()]
```

On peut désormais le dire : la composante `P[T_préc]` représente le dernier instant où la position du point a été mesurée/calculée.

Exercice 26 — Le point fabriqué par la fonction ci-dessus ne bouge pas : sa vitesse et son accélération sont nulles.

1) Écrire un programme `PointMRUAléa(v, couleur)`, calqué sur celui ci-dessus, qui attribue en plus au point une vitesse de norme v dans une direction choisie aléatoirement. Les imports de `cos`, `sin` et `pi` doivent servir ici.

2) Écrire un programme `PointMotoriséAléa(a, couleur)`, toujours sur le même modèle, avec de nouveau une vitesse nulle mais cette fois-ci une accélération de norme a et de direction aléatoire.

Exercice 27 — Où l'on dessine tout. Voici.

```
from matplotlib.pyplot import figure
def Animer(Points) :

    # Petite interface
    fig = figure("Silence, ça bouge") ; Continuer = True
    def Interrompre(evt) :
        nonlocal Continuer
        Continuer = False
    fig.canvas.mpl_connect('close_event', Interrompre)

    # Initialisation de l'affichage
    ax  = fig.add_subplot(1, 1, 1)
    ax.axis([0, 1, 0, 1])
    X = [P[POS_x] for P in Points]
    Y = [P[POS_y] for P in Points]
    C = [P[COULEUR] for P in Points]
    Nuage = ax.scatter(X, Y, color = C)
    fig.canvas.draw() ; fig.show()

    # Boucle principale
    while Continuer :
        Déplacer(Points)
        Coords = [[P[POS_x], P[POS_y]] for P in Points]
        Nuage.set_offsets(Coords)
        fig.canvas.draw() ; fig.canvas.flush_events()
```

Compléter le programme

```
def Déplacer(Points) :
    for P in Points :
        # Déplacement cinématique
        t = Maintenant() ; dt = t - P[T_préc]
        P[VIT_x] += ... ; P[POS_x] += ...
        P[VIT_y] += ... ; P[POS_y] += ...
        P[T_préc] = t
        # Application de la règle particulière au point
```

```
        P[RÉGLE](P)
```

qui met à jour les positions les points. Pour chacun d'eux, on commencera par appliquer les lois de la cinématique pour recalculer leurs vitesses et positions, avant d'appliquer la règle (dont on ne sait toujours pas à quoi elle sert, mais ça va venir). Le code

```
>>> VERT  = (0, 0.7, 0.4)
>>> Points = [PointMotoriséAléa(0.01, VERT) for _ in range(10)]
>>> Animer(Points)
```

permet d'observer un petit groupe de points dans leur environnement naturel, positionnés aléatoirement, qui démarrent, prennent de la vitesse et s'en vont. Ce n'est pas encore très intéressant.

Exercice 28 — Les trois idéologies des points. Les points créés jusque là sont *salmantins*[1]. Pour eux, l'univers est plat, limité, et quand par malheur on en franchit un bord, on tombe dans un précipice sans fond et on disparaît.

1) D'autres points sont *maniens*[2]. Ils se déplacent en rebondissant contre les parois de l'univers. Supposons par exemple un rebond contre la paroi à droite. On le détecte en remarquant que $x > 1$, où (x, y) sont les coordonnées après le déplacement. On change alors a_x en $-a_x$, v_x en $-v_x$, et x en $2 - x$. Justifier ces deux formules, traiter le cas des trois autres parois, et compléter le programme ci-dessous.

```
def RebondSansInteraction(P) :
    if P[POS_x] > 1 :
        P[POS_x] = 2 - P[POS_x]
        P[VIT_x] = -P[VIT_x] ; P[ACC_x] = -P[ACC_x]
    elif P[POS_x] < 0 :
        ...
    if P[POS_y] > ... :
        ...
    elif ...
```

On peut alors tester.

```
>>> Points = [PointMotoriséAléa(0.01, VERT) for _ in range(10)]
>>> for P in Points : P[RÉGLE] = RebondSansInteraction
>>> Animer(Points)
```

2) Les points qui restent sont *pas que maniens*[3]. Pour eux l'univers est un tore, et ils peuvent en faire le tour dans les deux directions : lorsqu'ils franchissent un bord, ils réapparaissent, sans changement de vitesse ou d'accélération, par le bord opposé. Sur le modèle de la question précédente, écrire un programme `ToreSansInteraction(P)` qui modélise ce comportement.

Exercice 29 — Attracteurs et répulseurs. On fixe une constante $\mathrm{K} = 10^{-3}$ et on définit deux familles de points : les *attracteurs* et les *répulseurs*. Un point $\mathrm{P}(x, y)$ (qui n'est pas dans ces deux familles) sera dit *sous influance* si à chaque étape du mouvement son accélération est recalculée selon les règles

$$a_x = \left(\sum_{\mathrm{A} \in \texttt{Attracteurs}} \frac{\mathrm{K}}{\mathrm{AP}^3} \times (x_\mathrm{A} - x) \right) - \left(\sum_{\mathrm{R} \in \texttt{Répulseurs}} \frac{\mathrm{K}}{\mathrm{RP}^3} \times (x_\mathrm{R} - x) \right),$$

et la formule analogue en ordonnée.

1) Écrire le programme `Distance(A, B)` qui calcule la distance entre deux points (les deux points étant des listes de longueur neuf, toujours).

1. Voir par exemple `https://fr.wikipedia.org/wiki/Christophe_Colomb_devant_le_conseil_de_Salamanque`.
2. Qui comme chacun sait sont les habitants de Souzy, dans le Rhône.
3. Le lecteur appréciera mes efforts pour amener en douceur ce *subtil* calembour.

2) Compléter le programme ci-dessous, qui fabrique la règle de déplacement des points sous influence. Avant de calculer l'accélération, on applique le programme `Mouvement`, qui sera par exemple l'un des deux qu'on a vu `RebondSansInteraction` ou `ToreSansInteraction`.

```
def MSI(Attracteurs, Répulseurs, Mouvement) :
    def Règle(P) :
        Mouvement(P)
        P[ACC_x] = 0 ; P[ACC_y] = 0
        for A in Attracteurs :
            ...
        for R in Répulseurs :
            ...
    return Règle
```

3) Compléter le programme ci-dessous, qui crée une animation avec a attracteurs fixes rouges, r répulseurs fixes bleus, i points sous influence maniens verts, et lance l'animation.

```
ROUGE = (0.8, 0.1, 0) ; BLEU = (0.1, 0.5, 0.9)
def Animation_1(a, r, i) :
    Attracteurs = ...
    Répulseurs = ...
    SousInfluence = ...
    for P in SousInfluence :
        P[RÉGLE] = MSI(Attracteurs, Répulseurs, RebondSansInteraction)
    Animer(...)
```

4) Écrire un programme `Animation_2(n)` qui crée n points initialement en M.R.U. (par exemple avec une vitesse $v = 0{,}01\,u.g./s$), maniens, noirs, se repoussant tous mutuellement, et lance l'animation.

Exercice 30 — On veut maintenant créer une animation avec n attracteurs fixes, toujours rouges, et un unique point sous influence. Lorsque celui-ci se rapproche trop d'un point attracteur, l'attracteur se téléporte loin de lui.

1) Écrire un programme `ChoisirLoinDe(M, d)` qui prend en argument un point M et un réel $d > 0$ pas trop grand, et qui renvoie un couple $(x, y) \in [0\,;1]^2$ aléatoire à une distance au moins d de M.

2) Compléter le programme ci-dessous, qui gère la règle de téléportation des attracteurs, M étant le point sous influence.

```
def Téléporteur(M) :
    # La distance de déclenchement
    d = 0.05
    def Règle(P) :
        ...
    return Règle
```

3) Compléter le programme `Animation_3(n)` qui répond au problème posé.

```
def Animation_3(n) :
    Attracteurs = ...
    M = ...
    M[RÉGLE] = ...
    for P in Attracteurs :
        ...
    Animer(...)
```

Premier problème — Vitesse de convergence : Euler vs Heun

Exercice 1

1) On a $f'(t) = -t \times \exp(-t^2/2) = -t \times f(t)$ et $f(0) = \exp(-0^2/2) = 1$ donc f est bien une (la) solution du problème de Cauchy.

2) On pose $\mathrm{F}(x,t) = -t \times x$, de sorte que $f(y(t),t) = -t \times y(t) = y'(t)$. D'autre part ici $t_0 = 0$ et $y_0 = 1$.

3) En Python on écrit

```
# Paramètres du problème
t0 = 0 ; y0 = 1
def F(x, t) :
    return -t * x

# Paramètres de la résolution
tmax = 5 ; N = 1000

# Résolution (calcul des valeurs approchées en certains points)
(T, Y) = Heun(F, y0, t0, tmax, N)
```

Exercice 2

1) L'argument `Méthode` désigne un programme comme `Euler` ou `Heun`.

2) L'expression `Y[-1]` désigne le dernier élément de la liste `Y`, c'est-à-dire, puisque celle-ci est de longueur $\mathrm{N}+1$, la valeur de `Y[N]`. C'est-à-dire $y_{\mathrm{N}} \simeq y(t_{\mathrm{N}}) = y(t_{\max})$.

Exercice 3

1) Si $\ln(|\mathrm{E}(\mathrm{N})|) \simeq \beta - \alpha \ln(\mathrm{N})$, on obtient, en passant à l'exponentielle : $|\mathrm{E}(\mathrm{N})| \simeq e^{\beta} \times \mathrm{N}^{-\alpha}$. En fait, dans ce cas, on peut être plus rigoureux et passer aux équivalents : si $\ln(|\mathrm{E}(\mathrm{N})|) = \beta - \alpha \ln(\mathrm{N}) + o(1)$ alors

$$|\mathrm{E}(\mathrm{N})| = \frac{e^{\beta}}{\mathrm{N}^{\alpha}} \times (1 + o(1)) \sim \frac{e^{\beta}}{\mathrm{N}^{\alpha}}.$$

2) Pour la méthode d'Euler, on utilise les points $(4{,}7\,;\,-6{,}9)$ et $(8{,}5\,;\,-10{,}8)$, qu'on a évidemment pris aussi éloignés que possible pour réduire l'erreur de lecture. Ce qui donne

$$\alpha \simeq -\frac{(-10{,}8) - (-6{,}9)}{8{,}5 - 4{,}7} \simeq 1{,}03.$$

Pour la méthode de Heun on utilise les points $(4{,}7\,;\,-10{,}2)$ et $(8{,}5\,;\,-18{,}1)$ et on trouve

$$\alpha \simeq -\frac{(-18{,}1) - (-10{,}2)}{8{,}5 - 4{,}7} \simeq 2{,}08.$$

3) En général : de $|\mathrm{E}(\mathrm{N})| \simeq \mathrm{K}/\mathrm{N}^{\alpha}$ on déduit $|\mathrm{E}(\mu\mathrm{N})| \simeq \mathrm{K}/(\mu^{\alpha}\mathrm{N}^{\alpha}) = |\mathrm{E}(\mathrm{N})|/\mu^{\alpha}$ donc l'ordre de la méthode correspond au coefficient directeur des questions précédentes.

Pour la méthode d'Euler, on trouve $\alpha \simeq 1$ et pour la méthode de Heun on trouve $\alpha \simeq 2$.

4) Puisque $|\mathrm{E}(\mathrm{N})| \sim \mathrm{K}/\mathrm{N}^{\alpha}$, plus α est grand et plus l'erreur tend rapidement vers zéro, même si la constante K est dans le même temps plus grande. Donc c'est la méthode de Heun la meilleure.

Exercice 4

1) Le programme `RégressionAffine` prend en argument deux listes de même longueurs, la première des nombres x_i et la seconde des nombres y_i. Il ne renvoie rien, mais affiche dans la console l'équation d'une

droite $y = mx + p$, en précisant évidemment les valeurs de m et p, dont la courbe passe aussi près que possible des points (x_i, y_i).

2) Reprenons les calculs effectués dans le programme `NuageErreur`, mais pour obtenir cette fois-ci l'équation de la droite plutôt que le dessin du nuage.

```
def ModélisationErreur(tmax, Méthode) :
    # Valeurs
    abscisses = [ 100, 200, 500, 1000, 2000, 5000 ]
    ordonnées = [ Erreur(tmax, N, Méthode) for N in abscisses]
    # Passage aux logarithmes
    abscisses = [ ln(N) for N in abscisses ]
    ordonnées = [ ln(abs(err)) for err in ordonnées ]
    # Affichage
    RégressionAffine(abscisses, ordonnées)
```

L'ordre de la méthode est l'opposé de la valeur du coefficient directeur m qui s'affiche dans la console.

Deuxième problème — Résolution d'une édo avec pas variable et abscisses obligées

Exercice 5

1) La variable `T` représente la liste `[`t_0`, `t_1`, ..., `t_{N}`]`.

2) Écrivons un programme qui construit une subdivision régulière en N segments de l'intervalle $[a\,;\,b]$.

```
def Subdivision(a, b, N) :
    h = (b - a) / N
    return [a + n * h for n in range(N + 1)]
```

On peut alors construire la liste en écrivant `T = Subdivision(t0, tmax, N)`.

Exercice 6

1) Il y a N segments donc N + 1 abscisses.

2) Remarquons que l'hypothèse $\mathrm{N} \geqslant 2$ permet d'assurer que la liste produite par le programme ci-dessous n'est pas vide.

```
def SubdivisionSansBouts(a, b, N) :
    h = (b - a) / N
    return [a + n * h for n in range(1, N)]
```

Exercice 7

1) La fonction `int(x)` convertit le flottant x en un entier, en tronquant sa partie décimale (ce qui donne $\lfloor x \rfloor$ pour les $x \geqslant 0$, mais $\lceil x \rceil$ pour les $x < 0$).

2) Le but de la question est de définir ce qu'on appelle une répartition *équitable* : on voudrait que le nombre de points dans chaque segment soit proportionnel à la longueur de celui-ci. Notons pour simplifier $a = v_{-1}$, $b = v_r$ et n_i le nombre de points placés dans $]v_{i-1}\,;\,v_i[$. La proportionnalité signifierait

$$\frac{n_i}{v_i - v_{i-1}} = \text{cste}$$

or ici on a par construction

$$n_i \leqslant \frac{v_i - v_{i-1}}{b-a} \times (\mathrm{N} - r - 1) < n_i + 1 \quad \Leftrightarrow \frac{\mathrm{N}-r-1}{b-a} - \frac{1}{v_i - v_{i-1}} < \frac{n_i}{v_i - v_{i-1}} \leqslant \frac{\mathrm{N}-r-1}{b-a}.$$

Si $1/(v_i - v_{i-1})$ est petit devant $(\mathrm{N}-r-1)/(b-a)$, on aura bien un rapport *à peu près* constant. Si l'un des intervalle est très petit, c'est-à-dire si $v_i - v_{i-1}$ est de l'ordre de $(b-a)/\mathrm{N}$, alors il y aura zéro ou un point

entre v_{i-1} et v_i dans la subdivision finale. C'est un cas pathologique : on ne peut évidemment pas espérer une proportionnalité sur un échantillon si petit, et de plus ceci ne peut se produire que pour très peu d'indices i (à moins que r soit de l'ordre de N, ce qui est idiot : si c'est pour imposer *tous* les points de la subdivision, autant utiliser le programme du début de la partie). On va donc laisser ces situations de côté, et supposer que r est petit devant N (donc devant $\mathrm{N}-r-1$), et que $v_i - v_{i-1}$ est de l'ordre de $(b-a)/r$. Ainsi

$$\frac{1}{v_i - v_{i-1}} \simeq \frac{r}{b-a} \ll \frac{\mathrm{N}-r-1}{b-a},$$

et c'est bien ce qu'on voulait.

3) Réutilisons la majoration de la question précédente :

$$\sum_{i=0}^{r} n_i \leqslant \sum_{i=0}^{r} \frac{v_i - v_{i-1}}{b-a} \times (\mathrm{N}-r-1) = \mathrm{N}-r-1,$$

parce qu'évidemment la somme des $v_i - v_{i-1}$ est télescopique et donne $v_r - v_{-1} = b-a$. C'est exactement la majoration demandée.

4)

```
def AjouterCeuxQuiManquent(V, N, Nombres) :
    # Calcul du nombre de manquant
    nb = 0
    for n in Nombres :
        nb += n
    # Ajout des manquants
    while nb < N - len(V) - 1 :
        i = EntAléa(0, len(V))
        Nombres[i] += 1
        nb += 1
```

5) Le programme principal est un peu pénible à écrire, car il faut gérer les cas particuliers du premier et du dernier segment, et aussi le fait qu'il y « un poteau de plus » qu'il n'y a d'intervalles.

```
def SubdivisionAvecObligés(a, b, N, V) :
    Nombres = RépartirParDéfaut(a, b, V, N)
    AjouterCeuxQuiManquent(V, N, Nombres)
    X = [a] + SubdivisionSansBouts(a, V[0], Nombres[0] + 1)
    for i in range(1, len(V)) :
        X.append(V[i - 1])
        X += SubdivisionSansBouts(V[i - 1], V[i], Nombres[i] + 1)
    X.append(V[len(V) - 1])
    X += SubdivisionSansBouts(V[len(V) - 1], b, Nombres[len(V)] + 1)
    X.append(b)
    return X
```

Testons, pour observer les « cas pathologiques » décrit dans la deuxième question. Plaçons volontairement deux points trop près, par rapport au pas que donnerait la subdivision régulière (ici 3 et 3,5) : on voit qu'alors aucun point n'est ajouté entre les deux.

```
>>> SubdivisionAvecObligés(0, 10, 15, [2, 3, 3.5])
[0, 0.6666666666666666, 1.3333333333333333, 2, 2.5, 3, 3.5,
 4.222222222222222, 4.944444444444445, 5.666666666666666,
 6.388888888888889, 7.111111111111111, 7.833333333333333,
 8.555555555555555, 9.277777777777779, 10]
```

Exercice 8

Exercice purement décoratif.

```
def RésolutionAvecObligés(Méthode, F, y0, t0, tmax, N, V) :
    T = SubdivisionAvecObligés(t0, tmax, N, V)
    return (T, Méthode(F, y0, T))
```

Troisième problème — Interpolation et courbes de Bézier

Exercice 9

1) Chaque ligne commence et termine par un, et les cases « intermédiaires » se déduisent de la ligne précédente par l'application de la formule de Pascal.

```
def LigneSuivante(L) :
    S = [1]
    for i in range(1, len(L)) :
        S.append(L[i - 1] + L[i])
    S.append(1)
    return S
```

2)

```
def Binomiaux(n) :
    L = [1]
    for _ in range(n) :
        L = LigneSuivante(L)
    return L
```

Exercice 10

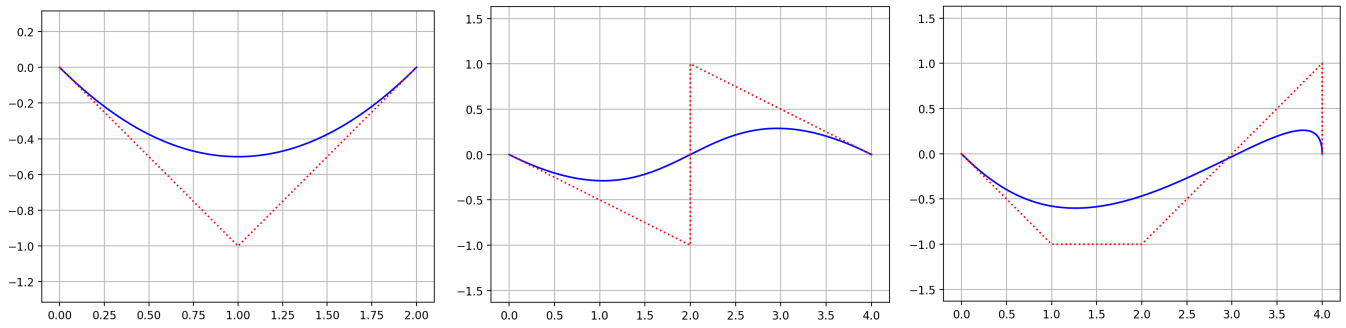
1) Attention à la valeur de n : c'est le nombre de points diminué de 1, vu que leurs indices vont de 0 à n (ce qui fait $n+1$ points). Dans l'algorithme ci-dessous, on s'efforce de *ne pas calculer plusieurs fois* les coefficients binomiaux et la quantité c ci-dessous.

En particulier, on calcule les coefficients binomiaux *en dehors* de la fonction `P` (sinon ils sont recalculés à chaque nouvelle valeur de t). C'est ce qu'on appelle un *pré-calcul* et c'est une méthode très importante.

```
def Bézier(Points) :
    n = len(Points) - 1
    B = Binomiaux(n)
    def P(t) :
        x = 0 ; y = 0
        for k in range(0, n + 1) :
            c = B[k] * t ** k * (1 - t) ** (n - k)
            x += c * Points[k][0]
            y += c * Points[k][1]
        return (x, y)
    return P
```

2) Le paramètre `":"` permet d'obtenir une ligne en pointillés. Ce n'est évidement pas à connaître : on teste et on regarde, ou bien on va consulter la documentation. Par défaut, le trait est plein. Autre variante possible : le paramètre `"-"` permet d'obtenir une ligne interrompue (c'est-à-dire en petits tirets).

3) Voici, dans l'ordre, les trois dessins obtenus.



4) La courbe « démarre » au premier point de contrôle et « termine » au dernier, mais elle ne passe pas (en général) par les points de contrôles intermédiaires.

5) Le premier tronçon (P_0P_1) est tangeant à la courbe en P_0, et le dernier $(P_{n-1}P_n)$ est tangeant à la courbe en P_n.

C'est d'ailleurs assez facile à prouver : t^2 étant en facteur dans les termes qui définissent $x(t)$ pour $k \geqslant 2$, seuls les deux premiers restent lorsqu'on calcule la dérivée en zéro :

$$x'(0) = n(-1)(1-0)^{n-1}x_0 + n((1-0)^{n-1} - 0(n-1)(1-0)^{n-2})x_1 = n(x_1 - x_0).$$

Le calcul est le même pour $y'(0)$ et on obtient donc un vecteur vitesse $\overrightarrow{OP}'(0) = n\overrightarrow{P_0P_1}$.

On peut faire aussi le calcul en $t = 1$, ou bien effectuer le reparamétrage $Q(t) = P(1-t)$, ce qui ne change évidemment pas la courbe, mais seulement son sens de parcours. L'application à $Q(t)$ de ce qui précède donne $\overrightarrow{OQ}'(0) = n\overrightarrow{P_nP_{n-1}}(0)$ soit $-\overrightarrow{OP}'(1) = -n\overrightarrow{P_nP_{n-1}} = n\overrightarrow{P_{n-1}P_n}$.

Exercice 11

1) Il s'agit de prouver que l'équation $x(t) = x$ admet une solution (attention au notation : on a une fonction $x(t)$ et un nombre fixé x). Or la fonction $x(t)$ est continue sur $[0\,;\,1]$ (c'est une fonction polynomiale) et on a $x(0) = x_A$ et $x(1) = x_B$. C'est donc tout simplement le théorème des valeurs intermédiaires qui nous assure, pour les $x \in [x_A\,;\,x_B]$, l'existence d'au moins une solution.

2) On va donc appliquer l'algorithme de dichotomie à la fonction $t \mapsto x(t) - x$. Une fois t trouvé, il n'y a plus qu'à regarder qui est $y(t)$.

```
def Trouver_y(xA, xB, x, P) :
    # On procède par dichotomie
    def f(t) :
        return P(t)[0] - x
    a = 0 ; b = 1
    while b - a > 1e-5 :
        c = (a + b) / 2
        if f(a) * f(c) <= 0 :
            b = c
        else :
            a = c
    return P((a + b) / 2)[1]
```

Exercice 12

1) On applique l'algorithme de recherche dichotomique, qui est à connaître. Dans la version ci-dessous, on a un invariant de boucle : à chaque itération, on a `L[i]` $\leqslant t <$ `L[j]`.

```
def Recherche(T, t) :
    i = 0 ; j = len(T) - 1
    while j - i > 1 :
        c = (i + j) // 2
```

```
        if T[c] > t :
            j = c
        else :
            i = c
    return i
```

2) Lorsqu'il n'y a que deux points de contrôle, on retrouve l'expression $x(t) = (1-t)x_0 + tx_1$ (et idem en ordonnée) qui est la paramétrisation habituelle des segments. La courbe de Bézier est donc réduite dans ce cas-là au segment AB.

Cette question était censée étudier le premier cas du `if`, sauf que j'ai ensuite changé l'algorithme en réalisant que j'avais oublié un cas. Regardons donc maintenant ce qui se passe lorsque les dérivées en t_i et t_{i+1} sont (à peu près) égales. Les tangentes sont donc parallèles, et il y a deux situations : soit elles sont confondues, et le mieux qu'on puisse faire pour approcher la courbe, c'est de relier les deux points par un segment de droite (d'où la question) ; soit elles sont parallèles non confondues (c'est le cas que j'avais oublié). Dans ce cas, on utilise une courbe de Bézier avec quatre points de contrôle, pour avoir un point d'inflexion exactement au milieu de l'intervalle.

Le cas où les deux tangentes sont confondues est un cas particulier du second : les quatres points de contrôle sont alors alignés, et on peut vérifier que la courbe de Bézier est encore le segment de droite qui relie A et B.

3) On fait un petit calcul, et on trouve que $y = y_A + m_A(x - x_A)$ et $y = y_B + m_B(x - x_B)$ se coupe en (x, y) avec

$$x = \frac{y_B - y_A + m_A x_A - m_B x_B}{m_A - m_B}.$$

D'où le programme suivant.

```
def Intersection(xA, yA, mA, xB, yB, mB) :
    x = (yB - yA + mA*xA - mB*xB) / (mA - mB)
    y = yA + mA * (x - xA)
    return (x, y)
```

4) On l'a déjà dit dans une réponse précente : la liste `Yp` représente la liste des (valeurs approchées des) dérivées en les t_n.

Exercice 13

Si on calcule les dérivées seulement à la fin, on recalcule tous les $\mathrm{F}(y_n, t_n)$ (sauf le dernier qu'on avait pas encore calculé). Les versions proposées divisent donc par deux les calculs (étant vu que le calcul de la fonction F est en général le plus compliqué, on peut imaginer par exemple

$$\mathrm{F}(x, t) = x + \sqrt{\cos(tx) + 1},$$

(voire pire si on est dans $\mathbf{R}^d$ et pas dans $\mathbf{R}$) alors que tous les autres calculs sont des addtions et multiplications.

Exercice 14

1) Rappelons que la fonction `plot(X, Y)` trace la ligne brisée dont les sommets sont les (x_n, y_n), avec `X = [..., `x_n`, ...]` et `Y = [..., `y_n`, ...]`. C'est donc précisément la courbe de l'interpolation affine par morceaux du nuage de points.

2) Voilà une question plus subtile ! La méthode d'Euler calcule y_{n+1} en faisant l'approximation dite affine, c'est-à-dire en supposant que la solution est confondue avec la tangente en t_n jusqu'à l'abscisse t_{n+1}. Le point suivant est donc sur cette tangente, ce qui veut dire que l'intersection calculée par le programme de l'exercice **12** *est* le point suivant (t_{n+1}, y_{n+1}). On l'a déjà mentionné : lorsque tous les points de contrôle sont alignés, la courbe de Bézier est un segment de droite. Et donc on retrouve l'interpolation affine par morceaux.

Quatrième problème — Méthode d'Euler implicite, cas scalaire : Picard vs Newton

Exercice 15

1) Supposons qu'il existe deux solutions distinctes x et y : on a alors un problème, puisque

$$|x-y| = |f(x)-f(y)| \leqslant \mathrm{K} \times |x-y| < |x-y|.$$

Donc il y a au plus une solution.

2) Commençons par remarquer, pour un indice k quelconque, qu'on a

$$|x_{k+1}-x_k| = |f(x_k)-x_k| \leqslant \mathrm{K} \times |f(x_{k-1})-x_{k-1}| \leqslant \cdots \leqslant \mathrm{K}^k \times |f(x_0)-x_0|.$$

Ceci se prouve facilement par récurrence. Maintenant ajoutons des termes (ça paraît assez grossier comme majoration, mais nous allons voir que la vitesse de convergence est très rapide, et donc que les termes qu'on ajoute ne sont pas bien grands :

$$|x_n-x_m| \leqslant |x_n-x_{n-1}| + |x_{n-1}-x_{n-2}| + \cdots + |x_{m+1}-x_m| \leqslant \mathrm{K}^m \times |\mathrm{K}^{n-m-1} + \cdots + \mathrm{K} + 1| \times |f(x_0)-x_0|.$$

On applique la formule de la somme géométrique, ce qui est possible vu que K n'est pas égal à 1, et on trouve la relation demandée.

3) Par exemple avec $m = 0$, on obtient que pour tout indice n on a

$$x_n \in \left[x_0 - |f(x_0)-x_0| \times \frac{1}{1-\mathrm{K}}\,;\, x_0 + |f(x_0)-x_0| \times \frac{1}{1-\mathrm{K}}\right]$$

ce qui prouve bien que la suite est bornée.

4) Soit $(x_{\varphi(n)})_{n\geqslant 0}$ une sous-suite convergente. Fixons $\varepsilon > 0$ et un rang n_0 à partir duquel on a $|\ell - x_{\varphi(n)}| \leqslant \varepsilon/2$. On peut alors écrire

$$|\ell - x_n| \leqslant |\ell - x_{\varphi(n)}| + |x_{\varphi(n)} - x_n| \leqslant \frac{\varepsilon}{2} + \frac{\mathrm{K}^{\varphi(n)-n}}{1-\mathrm{K}} \times |f(x_0)-x_0|.$$

Si $\varphi(n) - n$ tend vers l'infini, alors à partir d'un certain rang n_1 le deuxième terme ci-dessus est lui aussi inférieur à $\varepsilon/2$, et pour $n \geqslant \max\{n_0\,;\, n_1\}$ on a $|\ell - x_n| \leqslant \varepsilon$ ce qui prouve que $x_n \longrightarrow \ell$. Sinon, $\varphi(n) - n$, qui est une suite croissante d'entiers, est bornée : elle converge donc, et il est bien connu que lorsqu'une suite d'entiers converge alors elle est en fait stationnaire à partir d'un certain rang. Mais $\varphi(n) - n = p$ à partir d'un certain rang signifie que la sous-suite $(x_{\varphi(n)})_{n\geqslant 0}$ est simplement un *décalage* de la suite de départ. Donc si elle converge, la suite de départ aussi (et vers la même limite, évidemment).

5) Puisque f est lipschitzienne, elle est en particulier continue, on peut donc faire tendre n vers l'infini dans la relation $x_{n+1} = f(x_n)$ pour obtenir $\ell = f(\ell)$.

6) On l'a laissé entendre, mais pas encore écrit précisément : la majoration établie à la deuxième question montre, en faisant tendre n vers l'infini, que

$$|\ell - x_m| \leqslant |f(x_0)-x_0| \times \frac{\mathrm{K}^m}{1-\mathrm{K}}.$$

On a donc une vitesse de convergence au moins géométrique : c'est très rapide. Cela étant comme on ne connaît pas K, on ne va pas utiliser cette majoration comme critère d'arrêt, et suivre la consigne de l'énoncé.

```
def RésoudrePicard(f, x_approché) :
    x = x_approché
    y = f(x)
    while abs(y - x) > 1e-10 :
        x = y ; y = f(x)
    return y
```

7) Dans le cas de la méthode d'Euler implicite, la fonction f est $f(x) = y_n + h_n \times \mathrm{F}(x, t_{n+1})$.

Exercice 16

Puisque F est K-lipschitzienne par rapport à se première variable, on a

$$|f(y) - f(x)| = |y_n + h_n\mathrm{F}(y, t_{n+1}) - y_n - h_n\mathrm{F}(x, t_{n+1}| = |h_n| \times |\mathrm{F}(y, t_{n+1}) - \mathrm{F}(x, t_{n+1})| \leqslant |h_n| \times \mathrm{K}.$$

Soit h un majorant de h_n (même si le pas de la méthode est irrégulier, il doit tendre vers zéro quand $\mathrm{N} \to \infty$ pour que la méthode converge). Lorsque N est assez grand on doit donc avoir $|h_n| \times \mathrm{K} \leqslant |h| \times \mathrm{K} < 1$. Et l'hypothèse de l'exercice précédent est donc vérifiée.

Exercice 17

1)

```
def Dérivée(g, x, h = 1e-7) :
    return (g(x + h) - g(x - h)) / (2 * h)
```

2)

```
def RésoudreNewton(g, x_approché) :
    x = x_approché
    h = g(x) / Dérivée(g, x)
    while abs(h) > 1e-10 :
        x -= h
        h = g(x) / Dérivée(g, x)
    return x - h
```

3) Dans le cas de la méthode d'Euler implicite, la fonction g est $g(x) = y_n + h_n\mathrm{F}(x, t_{n+1}) - x$. Ou son opposée, ça ne change rien.

Exercice 18

1)

```
def EulerImplicite(F, y0, t0, tmax, N) :
    h = (tmax - t0) / N
    y = y0 ; Y = [y0]
    t = t0 ; T = [t0]
    for n in range(N) :
        def f(x) :
            return y + h * F(x, t + h)
        y = RésoudrePicard(f, y) ; Y.append(y)
        t += h ; T.append(t)
    return (T, Y)
```

2)

```
def EulerImplicite(F, y0, t0, tmax, N) :
    h = (tmax - t0) / N
    y = y0 ; Y = [y0]
    t = t0 ; T = [t0]
    for n in range(N) :
        def g(x) :
            return y + h * F(x, t + h) - x
        y = RésoudreNewton(g, y) ; Y.append(y)
        t += h ; T.append(t)
    return (T, Y)
```

Exercice 19

1) Si l'on résout l'équation en $y'(t)$, pour obtenir la formule d'itération, on trouve

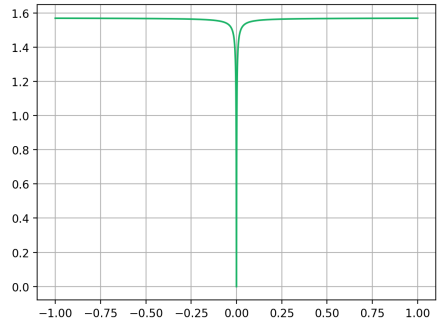
$$y'(t) = \frac{\cos(y(t))}{t}.$$

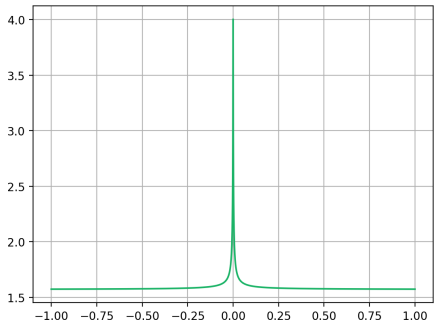
Et c'est un problème, parce qu'on ne peut pas remplacer t par zéro dans cette expression. La méthode d'Euler explicite ne peut donc pas s'appliquer.

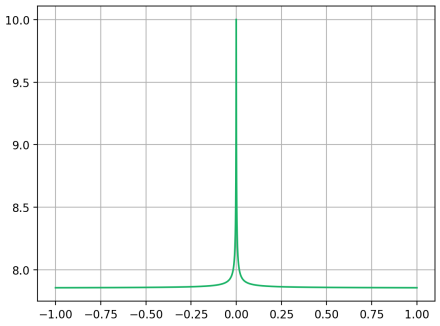
2) L'avantage de la méthode implicite, c'est que la formule d'itération fait intervenir t_{n+1} et pas t_n : pour $n = 0$ on n'aura donc pas besoin de calculer F en t_0.

```
def Solution(p) :
    t0 = 0 ; y0 = p
    def F(x, t) :
        return cos(x) / t
    # Partie "positive"
    (T, Y) = EulerImplicite(F, y0, t0, 1, 1000)
    plot(T, Y, color = VERT)
    # Partie "négative"
    (T, Y) = EulerImplicite(F, y0, t0, -1, 1000)
    plot(T, Y, color = VERT)
    grid(True) ; show()
```

3) On fait quelques essais, et la conjecture arrive très vite : la solution est $\mathscr{C}^1$ sur $\mathbf{R}$ si et seulement si $y_0 \equiv \pi/2[\pi]$ (et dans ce cas la solution est constante).







Cinquième problème — Méthode d'Euler implicite, cas vectoriel : fonctions de Bessel

Exercice 20

1) D'abord on « résout » l'équation en la dérivée la plus grande, c'est-à-dire on exprime $y''(t)$ en fonction du reste. On obtient

$$y''(t) = \left(\left(\frac{n}{t}\right)^2 - 1\right) y(t) - \frac{1}{t} y'(t).$$

Ensuite on pose (rappelons l'idée : on remplace $y(t)$ et $y'(t)$ par deux variables muettes x_0 et x_1)

$$\mathrm{F}_n\left(\begin{bmatrix} x_0 \\ x_1 \end{bmatrix}, t\right) = \begin{bmatrix} x_1 \\ ((n/t)^2 - 1)x_0 - x_1/t \end{bmatrix}$$

et on constate qu'on a alors $\mathrm{F}_n(\mathrm{Y}(t), t) = \mathrm{Y}'(t)$ pour tout t.

2)

```
def Fn(X, t) :
    return matrix([
     [X[1, 0]],
     [((n / t)**2 - 1) * X[0, 0]] - X[1, 0] / t])
```

Exercice 21

1)

```
def DérivéeDirectionelle(f, X, V, h = 1e-7) :
    return (f(X + h*V) - f(X - h*V)) / (2 * h)
```

2)

```
def Unitaire(d, i) :
    V = matrix(zeros((d, 1)))
    V[i, 0] = 1
    return V
```

3) Soit J la matrice $f'(\mathrm{X})$. Puisqu'on la calcule colonne par colonne, il faut à chaque étape recopier au bon endroit dans J la colonne $(\partial_j f)(\mathrm{X})$ qu'on vient de calculer.

```
def Dérivée(f, X) :
    (d, _) = X.shape
    J = matrix(zeros((d, d)))
    for j in range(d) :
        C = DérivéeDirectionelle(f, X, Unitaire(d, j))
        for i in range(d) :
            J[i, j] = C[i, 0]
    return J
```

Exercice 22

1)

```
def Norme(M) :
    (m, n) = M.shape
    nrm = 0
    for i in range(m) :
        for j in range(n) :
            if abs(M[i][j]) > nrm :
                nrm = abs(M[i][j])
    return nrm
```

2)

```
def RésoudreNewton(g, X_approché) :
    X = matrix(X_approché)
    H = Dérivée(g, X) ** (-1) * g(X)
    while Norme(H) > 1e-10 :
        X -= H
        H = Dérivée(g, X) ** (-1) * g(X)
    return X - H
```

Exercice 23

1) La fonction g est simplement $g(\mathrm{X}) = \mathrm{Y}_n + h_n \times \mathrm{F}(\mathrm{X}, t_{n+1}) - \mathrm{X}$.

2) Si le pas est petit, la fonction inconnue Y varie peu d'un pas à l'autre, et donc Y_n est sans doute une bonne estimation de départ pour résoudre l'équation qui donne Y_{n+1}.

```
def EulerImpliciteVectoriel(F, Y0, t0, tmax, N) :
    h = (tmax - t0) / N
    Y = Y0 ; Les_Y = [matrix(Y)]
    t = t0 ; T = [t0]
```

```
    for n in range(N) :
        def g(X) :
            return Y + h * F(X, t + h) - X
        Y = Newton(g, Y) ; Les_Y.append(matrix(Y))
        t += h ; T.append(t)
    return (T, Les_Y)
```

Exercice 24

1) Dans la formule d'itération on ne peut pas remplacer t par $t_0 = 0$ (c'est une valeur interdite). C'est le seul problème, donc on peut appliquer la méthode d'Euler explicite si et seulement si t_0 est non nul. Et précisément les conditions initiales pour le problème de Bessel sont données en $t_0 = 0$, donc, on va être dans ce cas défavorable.

2) Plusieurs subtilités importantes ici. D'abord, la fonction F_n de la première question doit être incorporée *à l'intérieur* de `RésoudreBessel`, sinon on ne connaît pas la valeur de n ! D'autre part, la méthode d'Euler renvoie une liste d'éléments de $\mathbf{R}^2$: pour chaque t_n on a $\mathrm{J}(t_n)$ et $\mathrm{J}'(t_n)$. Il faut extraire les premières composantes pour avoir seulement $\mathrm{J}(t_n)$.

```
def RésoudreBessel(n, y0, yp0, tmax, N) :
    t0 = 0 ; Y0 = matrix([[y0], [yp0]])
    def F(X, t) :
        return matrix([[X[1, 0]],
                       [-1/t * X[1, 0] + ((n/t)**2 - 1) * X[0, 0]]])
    (T, Les_Y) = EulerImpliciteVectoriel(F, Y0, t0, tmax, N)
    J = [Y[0, 0] for Y in Les_Y]
    return (T, J)
```

3) Attention ici, les matrices de `numpy` sont typées, il faut donc bien écrire `0.0` ou `1.0` et pas `0` ou `1` quand on donne les conditions initiales, sinon on a des soucis. D'autre part, on a besoin de trois fonctions de la librairie `matplotlib.pyplot`, qu'il faut donc importer.

```
>>> (T, J) = RésoudreBessel(0, 1.0, 0.0, 10, 10000)
>>> plot(T, J) ; grid(True) ; show()
```

4)

```
>>> (T, J) = RésoudreBessel(1, 0.0, 0.5, 10, 10000)
>>> plot(T, J) ; grid(True) ; show()
```

Exercice 25

1) Le mieux qu'on puisse faire, c'est d'approcher la fonction y sur $[a\,;b]$ par une fonction affine. Donc

$$y(t) \simeq y(a) + \frac{y(b) - y(a)}{b - a} \times (t - a).$$

Dès lors en résolvant $y(t) = 0$ avec la formule approchée ci-dessus, on trouve l'approximation (qui est homogène !)

$$t \simeq a - y(a) \times \frac{b - a}{y(b) - y(a)},$$

qu'on s'empresse de traduire en Python évidemment.

```
def ApprocherZéro(a, ya, b, yb) :
    return a - ya * (b - a) / (yb - ya)
```

2) On cherche les intervalles $]t_i\,;t_{i+1}[$ tels que $y_i \times y_{i+1} < 0$, et sur chacun d'eux on applique la formule précédente. Il faut aussi traiter à part le cas (qui ne se produira sans doute jamais) où un zéro est pile sur une abscisse de la subdivision. Et l'énoncé demande d'exclure les extrémités donc on ne teste $y_i = 0$ que pour $0 < i < \texttt{len(T)} - 1$.

```
def Zéros(T, Y) :
    Z = []
    for i in range(0, len(T) - 1) :
        if i > 0 and Y[i] == 0 :
            Z.append(T[i])
        elif Y[i] * Y[i + 1] < 0 :
            Z.append(ApprocherZéro(T[i], Y[i], T[i+1], Y[i+1]))
    return Z
```

3)

```
def ZérosBessel(tmax) :
    (T, J) = RésoudreBessel(0, 1.0, 0.0, tmax, 10000)
    return Zéros(T, J)
```

Sixième problème — Un exemple de graphique animé : mouvements de particules

Exercice 26

1) On choisit une direction aléatoire, c'est-à-dire un nombre θ entre 0 et 2π qui représente un angle, et on prend $v_x = v\cos(\theta)$ et $v_y = v\sin(\theta)$.

```
def PointMRUAléa(v, couleur) :
    x = Aléa() ; y = Aléa() ; thêta = 2 * pi * Aléa()
    vx = v * cos(thêta) ; vy = v * sin(thêta)
    ax = 0 ; ay = 0
    def Règle(P) :
        pass
    return [x, y, vx, vy, ax, ay, couleur, Règle, Maintenant()]
```

2) Même idée que dans le programme précédent, mais pour l'accélération plutôt que pour la vitesse.

```
def PointMotoriséAléa(a, couleur) :
    x = Aléa() ; y = Aléa()
    vx = 0 ; vy = 0 ; thêta = 2 * pi * Aléa()
    ax = a * cos(thêta) ; ay = a * sin(thêta)
    def Règle(P) :
        pass
    return [x, y, vx, vy, ax, ay, couleur, Règle, Maintenant()]
```

Exercice 27

On sait que pendant un très court instant $\mathrm{d}t$, la vitesse v_x varie de $a_x\mathrm{d}t$ (idem en ordonnée), et la position x de $v_x\mathrm{d}t$ (idem en ordonnée).

```
def Déplacer(Points) :
    for P in Points :
        # Déplacement cinématique
        t = Maintenant() ; dt = t - P[T_préc]
        P[VIT_x] += dt * P[ACC_x] ; P[POS_x] += dt * P[VIT_x]
        P[VIT_y] += dt * P[ACC_y] ; P[POS_y] += dt * P[VIT_y]
        P[T_préc] = t
        # Application de la règle particulière au point
        P[RÉGLE](P)
```

Exercice 28

1) Si $x > 1$, alors le point a parcouru (en abscisses) $x - 1$ unités graphiques en dehors du décor. S'il avait rebondi, il aurait parcouru la même longueur (toujours en abscisses) après le rebond, depuis le bord. Ce qui l'amène à l'abscisse $1 - (x - 1) = 2 - x$. Puisque il est reparti à l'envers, la vitesse et l'accélération sont changées en leurs opposées.

Pour le rebond à gauche, qu'on détecte en remarquant que $x < 0$, l'accélération et la vitesse sont également changées en leur opposées, et l'abscisses aussi (on a parcouru $x - 0$ en dehors du décor donc on doit revenir d'autant à l'intérieur, mais dans le sens opposé).

Les formules sont les mêmes en ordonnées. Remarquons que les rebonds horizontaux et verticaux se traient de manière indépendante, ce qui en particulier gère correctement les éventuels rebonds dans les coins.

```
def RebondSansInteraction(P) :
    if P[POS_x] > 1 :
        P[POS_x] = 2 - P[POS_x]
        P[VIT_x] = -P[VIT_x] ; P[ACC_x] = -P[ACC_x]
    elif P[POS_x] < 0 :
        P[POS_x] = -P[POS_x]
        P[VIT_x] = -P[VIT_x] ; P[ACC_x] = -P[ACC_x]
    if P[POS_y] > 1 :
        P[POS_y] = 2 - P[POS_y]
        P[VIT_y] = -P[VIT_y] ; P[ACC_y] = -P[ACC_y]
    elif P[POS_y] < 0 :
        P[POS_y] = -P[POS_y]
        P[VIT_y] = -P[VIT_y] ; P[ACC_y] = -P[ACC_y]
```

2) La situation du tore est beaucoup plus simple.

```
def ToreSansInteraction(P) :
    P[POS_x] %= 1 ; P[POS_y] %= 1
```

Exercice 29

1)

```
def Distance(A, B) :
    return sqrt((B[POS_x] - A[POS_x]) ** 2 + (B[POS_y] - A[POS_y]) ** 2)
```

2) On n'oublie pas de définir la constante quelque part, sinon évidemment ça ne marchera pas.

```
K = 1e-3
def MSI(Attracteurs, Répulseurs, Mouvement) :
    def Règle(P) :
        Mouvement(P)
        P[ACC_x] = 0 ; P[ACC_y] = 0
        for A in Attracteurs :
            r = Distance(P, A)
            P[ACC_x] += K / r ** 3 * (A[POS_x] - P[POS_x])
            P[ACC_y] += K / r ** 3 * (A[POS_y] - P[POS_y])
        for R in Répulseurs :
            r = Distance(P, R)
            P[ACC_x] -= K / r ** 3 * (R[POS_x] - P[POS_x])
            P[ACC_y] -= K / r ** 3 * (R[POS_y] - P[POS_y])
    return Règle
```

3)

```
def Animation_1(a, r, i) :
```

```
    Attracteurs = [PointFixeAléa(ROUGE) for _ in range(a)]
    Répulseurs = [PointFixeAléa(BLEU) for _ in range(r)]
    SousInfluence = [PointFixeAléa(VERT) for _ in range(i)]
    for P in SousInfluence :
        P[RÉGLE] = MSI(Attracteurs, Répulseurs, RebondSansInteraction)
    Animer(Attracteurs + Répulseurs + SousInfluence)
```

4) Ce deuxième cas est plus ardu : il faut, pour chaque point P, donner comme liste de répulseurs l'ensemble des points privé de P. On doit donc calculer une nouvelle liste `R` de répulseurs pour chacun des points. On rappelle que `L[:i]` désigne la sous-liste allant jusqu'à la case i exclue, et `L[i+1:]` la sous-liste démarrant à la case $i+1$. Syntaxe très pratique pour construire une liste privée d'une case précise.

```
NOIR = (0, 0, 0)
def Animation_2(n) :
    SousInfluence = [PointMRUAléa(0.01, NOIR) for _ in range(n)]
    for i in range(len(SousInfluence)) :
        R = SousInfluence[:i] + SousInfluence[i+1:]
        SousInfluence[i][RÉGLE] = MSI([], R, RebondSansInteraction)
    Animer(SousInfluence)
```

Exercice 30

1) Ici on ne peut pas appliqué le programme `Distance`, car il s'applique à deux points (qui rappelons-le sont des listes de longueur neuf), alors qu'on a juste un point et un couple (x, y). Tant pis, on recalcule manuellement la distance (et autant la laisser au carré pour s'éviter un calcul de racine carré qui est toujours plus lent que les autres).

```
def ChoisirLoinDe(M, d) :
    x = Aléa() ; y = Aléa()
    while (M[POS_x] - x)**2 + (M[POS_y] - y)**2 < d**2 :
        x = Aléa() ; y = Aléa()
    return (x, y)
```

2)

```
def Téléporteur(M) :
    # La distance de déclenchement
    d = 0.05
    def Règle(P) :
        if Distance(M, P) < d :
            (x, y) = ChoisirLoinDe(M, d)
            P[POS_x] = x ; P[POS_y] = y
    return Règle
```

3)

```
def Animation_3(n) :
    Attracteurs = [PointFixeAléa(ROUGE) for _ in range(n)]
    M = PointFixeAléa(VERT)
    M[RÉGLE] = MSI(Attracteurs, [], RebondSansInteraction)
    for P in Attracteurs :
        P[RÉGLE] = Téléporteur(M)
    Animer(Attracteurs + [M])
```