Chapitre 6 : Graphiques et méthode d'Euler

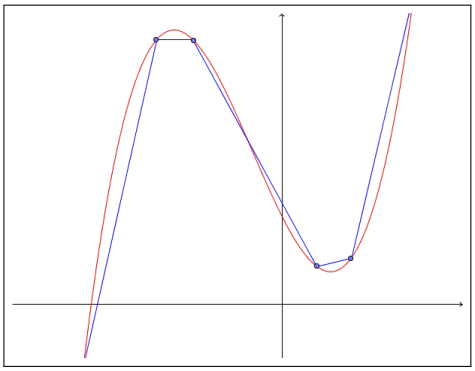
Commandes introduites dans ce cours : `np.linspace`, `matplotlib.pyplot`,`plt.plot`, `plt.show()`,`plt.clf()`,`plt.xlabel`,`plt.ylabel`,`plt.legend`,`plt.grid`,`plt.axis('scaled')`, `plt.subplot`,`scipy.integrate.odeint`.

1 La bibliothèque matplotlib.pyplot

1 A) Principe

Pour tracer la courbe représentative d'une fonction f sur un intervalle $[a,b]$, nous allons fabriquer une liste `[x1,..., xn]` contenant un certain nombre de réels de cet intervalle, et ensuite fabriquer la liste image `[f(x1),...,f(xn)]` associée. La primitive `plot` de la bilbiothèque `matplotlib.pyplot` se chargera alors de placer sur un graphe chacun des points M_i de coordonnées $(x_i, f(x_i))$, puis de relier chaque point M_i à son successeur M_{i+1} par des segments de droite. On obtient ainsi une approximation du graphe de f par une ligne polygonale (communément appelée une ligne brisée).
Ce procédé est simple, mais efficace : si l'on se donne suffisament de points, on obtient une ligne polygonale qui paraît lisse « à l'œil. » Tous les logiciels de calcul scientifique fonctionnent de cette manière.

La figure ci-contre présente un exemple d'approximation d'une courbe par une ligne polygonale.



1 B) Subdivision régulière d'un intervalle

En mathématiques on utilise la définition suivante.

Définition 1.1.
Soit $I=[a,b]$ un intervalle de $\mathbb{R}$. On appelle **subdivision régulière** de $[a,b]$ la suite $t_0,\dots,t_n$ où $n \geqslant 1$ et

$$\forall i \in [\![0,n]\!],\ t_i = a + i\frac{b-a}{n},$$

et donc en particulier on a $a=t_0$ et $b=t_n$. Le réel $h=\dfrac{b-a}{n}$ est alors appelé le **pas** de la subdivision.

Une subdivision de pas $h=\dfrac{b-a}{n}$ d'un intervalle $[a,b]$ est donc le découpage de celui-ci en n intervalles $[t_i,t_{i+1}]$, chacun d'eux étant de longueur égale à h. Pour définir une subdivision en Python, qui sera renvoyée sous forme de liste, il suffit d'utiliser une définition par compréhension. Voilà deux définitions possibles, la première à partir du nombre de points $n+1$ souhaités, la seconde à partir du pas h.

```
>>> a,b=0,1
>>> n=10
>>> [a+k*(b-a)/n for k in range(n+1)]
[0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]

>>> import math as ma
>>> h=1/10
```

```
>>> [a+k*h for k in range(ma.floor((b-a)/h)+1)]
[0.0, 0.1, 0.2, 0.30000000000000004, 0.4, 0.5, 0.6000000000000001, 0.7000000000000001,
0.8, 0.9, 1.0]
```

On utilise souvent ce concept en informatique, à ceci près que l'on n'impose pas à t_0 (resp. t_n) d'être égal à a (resp. b) ; on dira alors que l'on a donné une discrétisation de l'intervalle $[a, b]$. Pour discrétiser un intervalle, on peut alors utiliser les primitives `np.linspace` et `np.arange` : la première prendra en entrée le nombre de points souhaités, et la seconde le pas (en plus des bornes a, b de l'intervalle).

```
>>> np.linspace(0,1,11)
array([ 0. ,  0.1,  0.2,  0.3,  0.4,  0.5,  0.6,  0.7,  0.8,  0.9,  1. ])

>>> np.arange(0,1,1/10)
array([ 0. ,  0.1,  0.2,  0.3,  0.4,  0.5,  0.6,  0.7,  0.8,  0.9])
```

La seconde liste est différente de la première, puisqu'elle ne contient pas le réel 1, mais si l'on souhaite utiliser cette subdivision pour tracer une courbe représentative cela ne posera pas de problème (n sera suffisament grand en pratique pour que l'absence d'un point ne modifie pas le graphe).

1 C) La commande plot du module matplotlib.pyplot

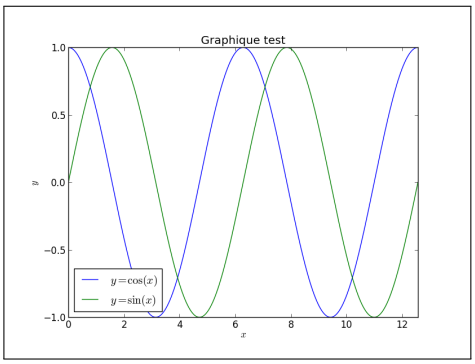
La commande `plot` de la bibliothèque `matplotlib.pyplot` permet de tracer des lignes polygonales. Cette fonction prend en entrée deux listes `Listex` et `Listey` qui contiennent respectivement les abscisses et les ordonnées des points à relier. En argument optionel, on peut donner un nom à la courbe avec la syntaxe `label='nom'`.

Pour tracer un graphique, il faudra alors définir un grand nombre de points à placer. Par exemple, pour tracer les courbes représentatives des fonction cosinus et sinus sur l'intervalle $[0, 4\pi]$, on commence par définir la liste des abscisses en discrétisant l'intervalle $[0, 4\pi]$ en $N + 1$ points, puis les listes des ordonnées des points à placer. Par exemple, pour $N = 1000$:

```
import matplotlib.pyplot as plt

N=1000
Listex=[k*4*ma.pi/N for k in range(N+1)]
Listeycos=[ma.cos(x) for x in Listex]
Listeysin=[ma.sin(x) for x in Listex]

plt.plot(Listex,Listeycos,label='y=cos(x)')
plt.plot(Listex,Listeysin,label='y=sin(x)')
```

On remarque que le graphique ne s'affiche pas encore à l'écran. La commande `plt.show()` permet cet affichage (et on pourra ensuite effacer cette figure avec la commande `plt.clf()`). On peut encore définir les noms et unités éventuelles des axes avec `xlabel` et `ylabel`, un titre avec `title`. Enfin, la commande `legend` permet de définir la position du nom de la courbe sur la figure.

```
plt.xlabel('x')
plt.ylabel('y')
plt.title("Graphique test")
plt.legend(loc=3)

plt.show()
```



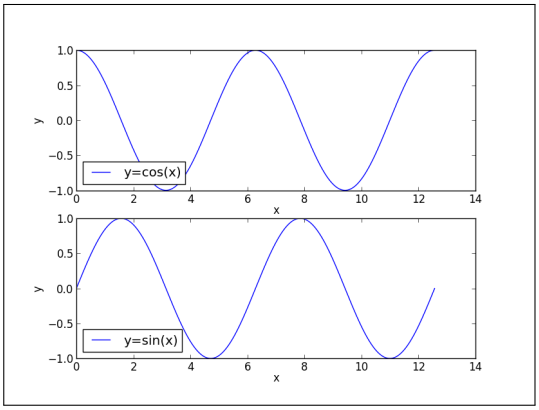
La commande `plt.plot` ouvre une fenêtre graphique délimitée par les plus grandes et plus petites abscisses et ordonnées apparaissant lors de la définition de la courbe. Cela peut être modifié en utilisant *a posteriori* les primitives `xlim` et `ylim` ; par exemple, le code `plt.xlim(-5,5)` tronquera la fenêtre graphique de sorte que la plus grande (resp. plus petite) abscisse apparaissant soit `-5` (resp. `5`).

Remarque 1.2.
Il existe d'autres options de personnalisation des graphiques, dont certaines seront présentées en TP. On peut par exemple utiliser la commande `plt.grid()` pour afficher un quadrillage correspondant à la graduation des axes, et la commande `plt.axis('scaled')` pour avoir la même échelle sur les deux axes.

1 D) La commande subplot

La commande `subplot` permet de tracer plusieurs courbes représentatives dans une même fenêtre graphique. Pour cela, on commence par définir une figure vide avec la commande `plt.figure`, puis on utilise la commande `plt.subplot(n,p,k)` qui signifie que la fenêtre graphique est découpée en $n \times p$ figures (n est le nombre de ligne et p le nombre de colonnes), et le code suivant concerne la k-ème de ces figures (les figures étant numérotées de haut en bas et de gauche à droite). Chacune des figures peut alors recevoir une courbe représentative, ainsi qu'un titre, une légende, etc....

```
plt.figure()

plt.subplot(2,1,1)
plt.plot(Listex,Listeycos,label='y=cos(x)')
plt.xlabel('x')
plt.ylabel('y')
plt.legend(loc=3)

plt.subplot(2,1,2)
plt.plot(Listex,Listeysin,label='y=sin(x)')
plt.xlabel('x')
plt.ylabel('y')
plt.legend(loc=3)

plt.show()
```



2 Résolution d'une équation différentielle

2 A) La méthode d'Euler

Le calcul explicite des solutions d'une équation différentielle n'est pas toujours aisé, et il existe un grand nombre de cas que l'on ne sait pas traiter. On dispose néanmoins de *méthodes numériques* qui permettent d'obtenir une valeur approchée de la solution, ce qui est suffisant dans le cadre des sciences expérimentales. La plus simple d'entre elles est la *méthode d'Euler*.

Cadre. Considérons une fonction $f : \mathbb{R}^2 \to \mathbb{R}$, et I un intervalle véritable de $\mathbb{R}$. On cherche à déterminer les fonctions $y : I \to \mathbb{R}$ qui vérifient l'équation différentielle :

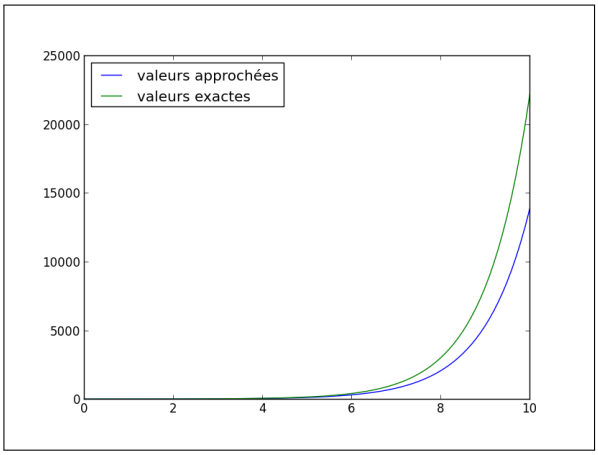
$$\forall t \in I,\ y'(t) = f(y(t), t).$$

On a ici affaire à une équation différentielle du premier ordre (non linéaire et à coefficients non constants a priori). On admet que, si la fonction f est dérivable et f' est continue, alors quels que soient $y_0 \in \mathbb{R}$, $t_0 \in I$, cette équation possède une unique solution y qui vérifie $y(t_0) = y_0$.

Principe de la méthode : approximation d'une dérivée par la pente d'une corde.

Exemple 2.1.
On implémente la méthode d'Euler, puis on l'applique à l'équation différentielle $y' = y$ sur l'intervalle $[0, 10]$ munie de la condition initiale $y(0) = 1$. La solution recherchée est donc la fonction exponentielle, et on représente les valeurs approchées sur un graphe pour observer l'erreur commise.

```
def Euler(f,a,b,y0,pas):
    t=a
    y=y0
    listey=[y]
    listet=[t]
    while t<b:
        y=y+pas*f(y,t)
        t=t+pas
        listey.append(y)
        listet.append(t)
    return(listet,listey)

def f(y,t):
    return(y)

a,b,y0,pas=0,10,1,10**(-1)
listet,listey=Euler(f,a,b,y0,pas)
listeexp=[ma.exp(t) for t in listet]
```



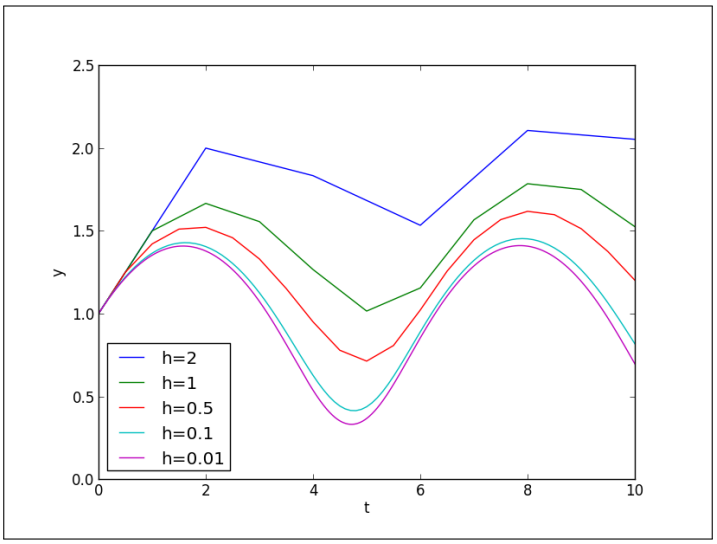
```
a,b,y0,pas=0,10,1,10**(-1)
listet,listey=Euler(f,a,b,y0,pas)
listeexp=[ma.exp(t) for t in listet]
```

```
plt.plot(listet,listey,label="valeurs approchees")
plt.plot(listet,listeexp,label="valeurs exactes")
plt.legend(loc=2)
plt.show()
```

Exercice 2.2.

Soit $y' = \dfrac{\cos t}{1+y^2}$ à résoudre sur l'intervalle $[0, 10]$ muni de la condition initiale $y(0) = 1$.

Le graphique ci-dessous fait apparaître la courbe représentative de la solution, ainsi que les courbes obtenues en appliquant la méthode d'Euler, avec les différents pas suivants : $h = 2$,$h = 1$,$h = \dfrac{1}{2}$, $h = \dfrac{1}{10}$ et $h = \dfrac{1}{100}$. On observe bien la convergence de la ligne polygonale vers la courbe représentative de la solution.



Écrivons un code Python qui utilise les fonctions précédentes et qui permet d'obtenir le graphique précédent

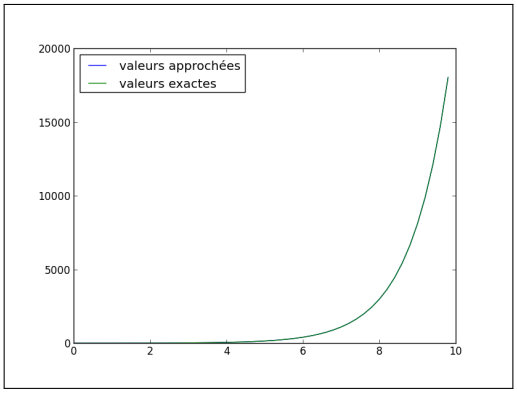
Remarque 2.3.
Intuitivement, il est clair que plus le pas h est petit, plus l'approximation de la solution est précise : on peut démontrer que « la méthode converge » lorsque h tend vers 0.
En pratique, l'implémentation de la méthode pose d'autres difficultés, des erreurs d'arrondis pouvant apparaître à chaque étape : en diminuant la valeur de h on augmente le nombre d'étapes lors de l'exécution de l'algorithme, et ainsi le nombre d'erreurs commises, ce qui peut faire diverger la méthode.

2 B) La commande scipy.integrate.odeint

Pour résoudre numériquement une équation différentielle, on peut utiliser la primitive `odeint` implémentée dans la bibliothèque `scipy.integrate`. La commande `scipy.integrate.odeint(f,y0,listet)` permet de résoudre l'équation $y' = f(y,t)$ munie de la condition initiale $y(a) = y_0$ sur un intervalle $[a,b]$ dont `listet` contient une subdivision : elle renvoie une liste $(y_0, \dots, y_n)$ où y_i est une valeur approchée de $y(t_i)$ (où t_i est le i-ème point de la subdivision et y l'unique solution de l'équation).

L'exemple suivant montre que `odeint` utilise une méthode plus évoluée que la méthode d'Euler : avec un pas $h = \dfrac{1}{10}$, il est déjà impossible à l'œil de faire la différence entre la courbe représentative de l'exponentielle et son approximation.

```
import math as ma
import scipy.integrate as sci

def f(y,t):
    return(y)

a,b,y0,pas=0,10,1,1/5

listet=np.arange(a,b,pas)
listey=sci.odeint(f,y0,listet)
listexp=[ma.exp(x) for x in listet]

plt.plot(listet,listey,label="valeurs approchees")
plt.plot(listet,listexp,label="valeurs exactes")
plt.legend(loc=2)
plt.show()
```



Remarque 2.4.
`odeint` est faite pour résoudre des équations de la forme $y' = f(y,t)$ et non pas $y' = f(t,y)$, attention aux confusions !