TPC² — T.C.D'I.

COURBES, LISTES, NOMBRES ALÉATOIRES

§1. Courbes représentatives

Déjà, deux librairies.

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

Pour tracer une courbe, on place un certain nombre de points qu'on relie par des segments. S'il y a suffisamment de points, on ne voit pas les segments, mais une courbe lisse. Rangeons ce nombre dans une constante.

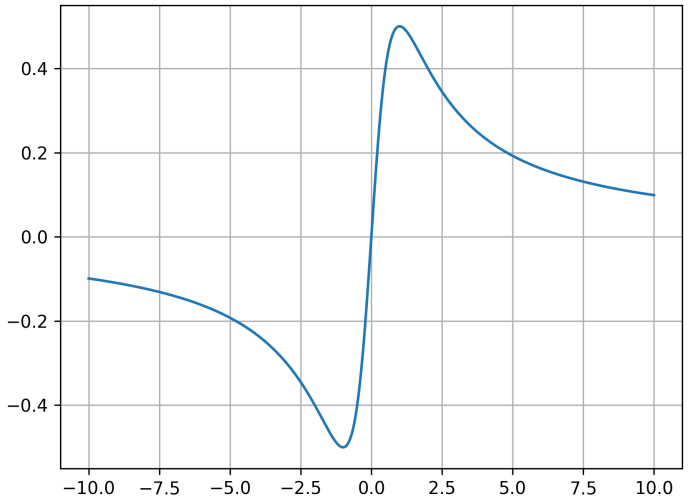
```
NB_POINTS = 1000
```

Une fonction pour tester...

```
def f(x) :
    return x / (1 + x ** 2)
```

...et les commandes pour obtenir la courbe, ici sur l'intervalle $[-10\,;\,10]$. Comme on aura besoin de tracer des courbes très souvent, ces quatre lignes sont à connaître par cœur.

```
absc = linspace(-10, 10, NB_POINTS)
ords = [f(x) for x in absc]
plot(absc, ords)
grid(True) ; show()
```



§2. Dérivées (1er acte)

Voyons maintenant comment tracer plusieurs courbes sur le même graphique. Par exemple, celle d'une fonction, et celle de sa dérivée. Pour calculer le nombre dérivé en un point, on utilise la formule du taux d'accroissement, avec un petit h (on y reviendra en détail plus tard, en fait, il y a une meilleure formule que ça).

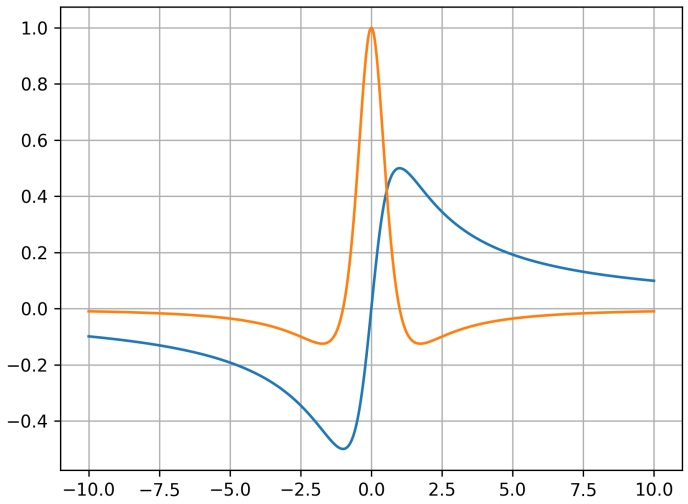
```
def NombreDérivé(f, x) :
    h = 1e-5
    return (f(x + h) - f(x)) / h
```

Mais nous on veut la *fonction* dérivée, pas juste un nombre. C'est plus abstrait : on écrit un programme qui renvoie une fonction.

```
def Dérivée(f) :
    h = 1e-5
    def df(x) :
        return (f(x + h) - f(x)) / h
    return df
```

Maintenant voyons comment superposer plusieurs courbes. On range dans une liste les fonctions à représenter, et on fait une boucle. Dans cette boucle, on écrit les instructions qui doivent être exécutées *pour chaque* fonction : le calcul des valeurs, et la construction de la courbe. En dehors de la boucle, on écrit les instructions qui ne sont exécutés qu'une fois : la construction de la liste d'abscisses, et la finalisation du dessin.

```
absc = linspace(-10, 10, NB_POINTS)
for fct in [f, Dérivée(f)] :
    ords = [ fct(x) for x in absc ]
    plot(absc, ords)
grid(True) ; show()
```

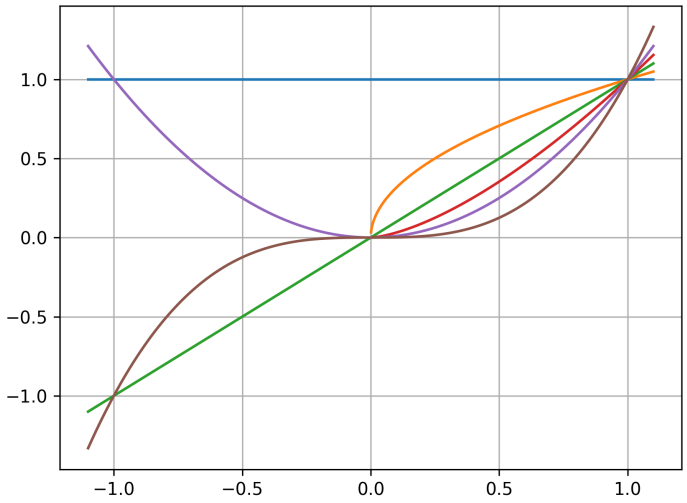


Voyons avec une famille de fonctions paramétrées par un n.

```
def f(n, x) :
    return x ** n
```

Même chose, mais cette fois-ci ce sont les différentes valeurs du paramètre qu'on range dans une liste.

```
absc = linspace(-1.1, 1.1, NB_POINTS)
for n in [0, 0.5, 1, 1.5, 2, 3] :
    ords = [ f(n, x) for x in absc ]
    plot(absc, ords)
grid(True) ; show()
```



Remarques.

a) Lorsque n n'est pas entier, les fonctions f_n ci-dessus ne sont définies que pour $x \geqslant 0$. Cela produit un message d'avertissement (en rouge) dans la console, et les courbes correspondantes paraissent donc « amputées ».

b) Dans les exemples de ce paragraphe, toutes les fonctions sont tracées sur le même intervalle. Si on veut un intervalle différent pour chaque fonction, il faut aussi ranger les intervalles dans des listes (voir les exercices).

30°

Exercice 1 — Soit $f : [a\,;b] \to \mathbf{R}$ une fonction dérivable.

1) Rappeler l'équation de la tangente à la courbe représentative de f en x_0.

2) Écrire un programme `CourbeEtTangente(`f`, `a`, `b`, `x_0`)` qui, sur un même graphique, trace la courbe représentative de f et sa tangente en x_0.

40°

Exercice 2

1) Proposer une fonction f dont la courbe représentative est (dans un repère orthonormé) le demi-cercle (supérieur) de centre $(1\,;1)$ et de rayon 4.

2) Idem pour le demi-cercle inférieur.

3) En déduire un programme `Cercle(`x_Ω`, `y_Ω`, R)` qui trace le cercle de centre $(x_\Omega\,;y_\Omega)$ et de rayon R.

50°

Exercice 3

1) Rappeler le domaine de définition de la fonction tangente.

2) Écrire un programme `CourbeTangente(`$n_{\min}$`, `$n_{\max}$`)` qui trace la courbe représentative de la fonction tangente de $n_{\min}\pi - \pi/2$ jusqu'à $n_{\max}\pi + \pi/2$. Les paramètres $n_{\min}$ et $n_{\max}$ sont deux entiers relatifs, avec $n_{\min} \leqslant n_{\max}$.

§3. Dégradés

On étudiera les couleurs plus tard. En voici trois pour aujourd'hui.

```
1 ROUGE = (1.0, 0.0, 0.0)
2 JAUNE = (0.9, 0.9, 0.3)
3 BLEU  = (0.2, 0.4, 1.0)
```

Pour obtenir un dégradé d'une couleur à une autre, il faut faire varier chacune des trois composantes. Le plus simple est d'utiliser des fonctions affines. Supposons qu'on souhaite aller d'une couleur $c_1 = (r_1\,;g_1\,;b_1)$ à une couleur $(r_2\,;g_2\,;b_2)$. Le raisonnement est le même pour chaque composante, donc on va juste expliquer pour la première : on cherche une fonction affine $\varphi(t)$, qui varie de r_1 à r_2 lorsque t varie de 0 à 1.

Cette fonction s'écrit $\varphi(t) = p + t \times m$, le coefficient directeur est donné (par exemple) par

$$m = \frac{\varphi(1) - \varphi(0)}{1 - 0} = \frac{r_2 - r_1}{1} = r_2 - r_1,$$

et l'ordonnée à l'origine est directement $p = \varphi(0) = r_1$. Ce qui donne finalement $\varphi(t) = r_1 + t \times (r_2 - r_1)$.

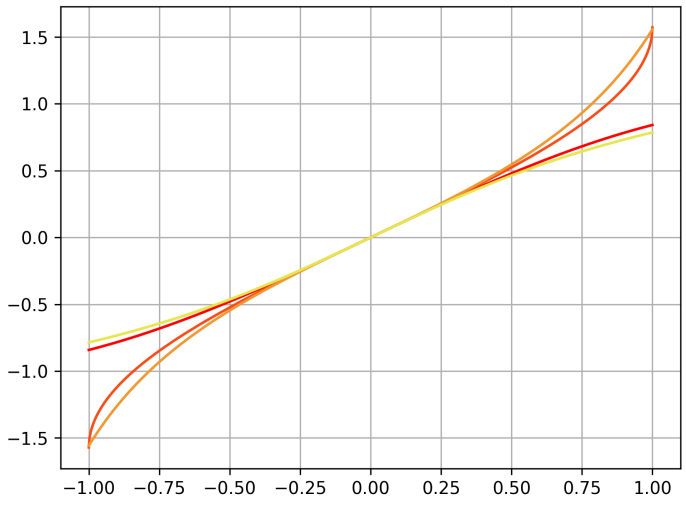
```
def Dégradé(c1, c2, t) :
    (r1, g1, b1) = c1 ; (r2, g2, b2) = c2
    r = r1 + t * (r2 - r1)
    g = g1 + t * (g2 - g1)
    b = b1 + t * (b2 - b1)
    return (r, g, b)
```

Testons ça avec une liste de fonctions, qu'on représente sur un intervalle commun $[a\,;\,b]$. S'il y a n fonctions, il faut choisir n couleurs, ici (par exemple) du `ROUGE` au `JAUNE`. Le paramètre t doit donc prendre n valeurs de 0 à 1 (inclus), on le prend de la forme $t = i/(n-1)$, avec $i = 0\,;\,1\,;\,\ldots\,;\,n-1$ (ce qui fait bien n valeurs).

```
def RepresenterPlusieurs(Fonctions, a, b) :
    absc = linspace(a, b, NB_POINTS)
    for i in range(len(Fonctions)) :
        f = Fonctions[i]
        ords = [f(x) for x in absc]
        t = i / (len(Fonctions) - 1)
        clr  = Dégradé(ROUGE, JAUNE, t)
        plot(absc, ords, color = clr)
    grid(True) ; show()
```

Testons avec quatre fonctions classiques (qui ont toutes le même comportement au voisinage de zéro, donc des courbes qui se ressemblent).

```
>>> from numpy import sin, arcsin, tan, arctan
>>> RepresenterPlusieurs([sin, arcsin, tan, arctan], -1, 1)
```



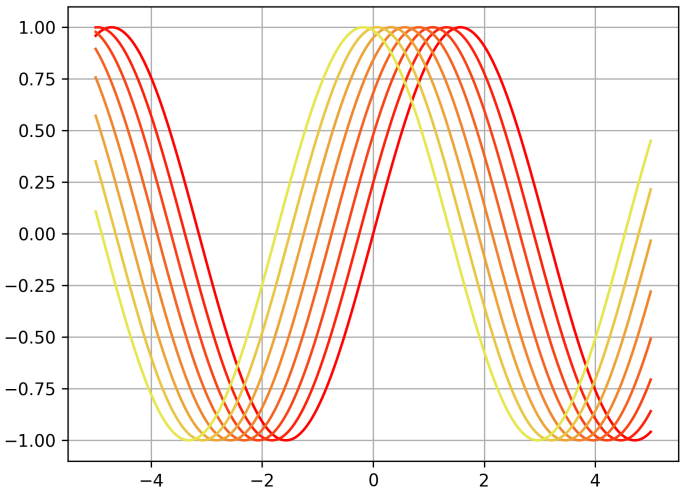
Même chose pour une famille de fonctions paramétrées.

```
def RepresenterFamille(f, a, b, Paramètres) :
    absc = linspace(a, b, NB_POINTS)
    for i in range(len(Paramètres)) :
        ords = [f(Paramètres[i], x) for x in absc]
        t = i / (len(Paramètres) - 1)
        clr  = Dégradé(ROUGE, JAUNE, t)
        plot(absc, ords, color = clr)
    grid(True) ; show()
```

Testons avec des sinusoïdes de mêmes amplitude et pulsation, mais de déphasages différents.

```
from numpy import sin
def g(n, x) :
    return sin(x + n / 4)
```

```
>>> RepresenterFamille(g, −5, 5, [0, 1, 2, 3, 4, 5, 6, 7])
```



30°

Exercice 4 — On considère, pour $b \in \mathbf{R}$, la fonction définie par la formule $f_b(x) = x^2 + bx + 3$.

1) Proposer un triplet `VERT` convaincant.

2) Tracer, sur un même graphique et dans un dégradé allant du bleu au vert, les courbes représentatives des fonctions f_b, pour b allant de -10 à 10.

3) En recommençant avec un intervalle de plus en plus petit pour b, trouver graphiquement la valeur de b pour laquelle la fonction f_b s'annule exactement une fois.

4) Calculer le discriminant de f_b pour vérifier le résultat obtenu précédemment.

§4. Nombres aléatoires

Pour choisir un réel au hasard dans un intervalle $[a\,;\,b[$, on utilise le programme ci-dessous. Si $b < a$, le nombre renvoyé sera dans $]b\,;\,a]$.

```
from random import random
def Aléa(a = 0.0, b = 1.0) :
    return a + (b - a) * random()
```

Les arguments a et b, dans le programme ci-dessus, ont des valeurs *par défaut* : on peut les omettre lorsqu'on utilise le programme, et dans ce cas tout se passera comme si $a = 0$ et $b = 1$ (les valeurs « par défaut »).

```
>>> Aléa()
0.5746143397125896
>>> Aléa()
0.4072545746747832
>>> Aléa(−10, 10)
-6.663954314744826
>>> Aléa(−10, 10)
-4.287557421175445
```

Petite application : écrivons un programme qui renvoie une couleur aléatoire.

```
def CouleurAléa() :
    r = Aléa() ; g = Aléa() ; b = Aléa()
    return (r, g, b)
```

Exercice 5

1) Compléter le programme ci-dessous, qui renvoie une fonction affine aléatoire.

```
def AffineAléa() :
    m = ... ; p = ...
    def f(x) :
        return ...
    return f
```

2) Écrire un programme qui, sur un même graphique, trace plein de droites de toutes les couleurs.

Exercice 6 — Un avant-goût des séries de Fourier.

1) Écrire un programme `ListeAléa(`n`)` qui étant donné un entier naturel n renvoie une liste `[`a_0`, `a_1`, `a_2`, ..., `a_n`]` constituée de nombres aléatoires entre -1 et 1.

2) Compléter le programme ci-dessous, qui après avoir choisi de tels nombres, construit la fonction

$$f(x) = \sum_{k=0}^{n} \frac{a_k}{k^2+1} \times \cos(kx).$$

```
from numpy import cos
def FonctionAléa(n) :
    a = ListeAléa(n)
    def f(x) :
        s = 0
        for k in ... :
            s += ...
    return f
```

3) Tracer la courbe d'une fonction construite par le programme précédent, sur l'intervalle $[-\pi\,;\,5\pi]$. Recommencer plusieurs fois (puisque c'est aléatoire, la fonction, et donc la courbe, vont changer à chaque fois).

§5. Subdivisions d'un intervalle

On veut construire une liste de n nombres, régulièrement répartis de a jusqu'à b. Appelons ces nombres $x_0 = a < x_1 < x_2 < \cdots < x_{n-1} = b$. Puisque la numérotation commence à zéro, il faut bien s'arrêter à l'indice $n-1$ pour avoir en tout n valeurs.

Maintenant on cherche la formule qui donne x_i en fonction de l'indice i. Puisqu'il y a n nombres, ils délimitent $n-1$ sous-intervalles (c'est le principe « des poteaux et des intervalles »), et chacun de ces sous-intervalles doit avoir la même longueur. Puisque la longueur totale de $[a\,;\,b]$ est $b-a$, chaque sous-intervalle a pour longueur

$$h = \frac{b-a}{n-1}$$

(on divise par le nombre de sous-intervalles) et on obtient la formule $x_i = a + i \times h$. Le programme ci-dessous est exactement équivalent à `linspace(`a`, `b`, `n`)` de la librairie `numpy`, mais il faut savoir l'écrire si c'est demandé.

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

Pour ceux qui ne sont pas à l'aise avec les listes, on peut aussi l'écrire en « explicitant » la boucle.

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

```
>>> Subdivision(0, 10, 5)
[0.0, 2.5, 5.0, 7.5, 10.0]
```

Maintenant une variante : on ne donne plus le nombre de points, mais un `Pas` (constant), c'est-à-dire la valeur du h ci-dessus. Dans ce cas, le dernier sous-intervalle (qui se termine à b) peut être strictement plus court que les autres (si la division de $b-a$ par h ne « tombe pas juste »). Une remarque essentielle : ici on utilise une boucle `while`, puisqu'on ne sait pas à l'avance combien d'éléments il y aura dans la liste.

```
def SubdivisionAvecPas(a, b, Pas) :
    L = [] ; x = a
    while x <= b :
        L.append(x) ; x += Pas
    return L
```

```
>>> SubdivisionAvecPas(0, 10, 3)
[0, 3, 6, 9]
```

Remarque.

Dans le cas où la division de $b-a$ par le pas ne donne pas un nombre entier, on peut choisir de faire figurer b à la fin de la liste (dans ce cas, le dernier sous-intervalle est plus court que les autres), ou de ne pas le faire figurer (et alors, la liste ne termine pas à b, mais avant). Ci-dessus, on a pris la seconde option, et le programme est alors exactement équivalent à `arange(a, b, Pas)` du module `numpy`.

Encore une autre variante : une subdivision aléatoire (donc avec des pas non constants, en particulier). On veut toujours n valeurs en tout, allant de a jusqu'à b (inclus). On met donc $n-2$ nombres au hasard dans une liste, on ajoute a et b, puis on range la liste dans l'ordre croissant.

```
def SubdivisionAléa(a, b, n) :
    L = [Aléa(a, b) for i in range(n - 2)]
    L.append(a) ; L.append(b) ; L.sort()
    return L
```

```
>>> SubdivisionAléa(0, 10, 5)
[0, 4.841238163742164, 6.975002488435814, 9.989789130977131, 10]
```

Variante de la variante : on veut en plus qu'il y ait un écart minimal entre chaque point, disons $h_{\min}$. L'idée est la suivante : on note $b' = b - (n-1) \times h_{\min}$ et on construit une subdivision aléatoire $[x'_0,\ x'_1,\ \ldots,\ x'_{n-1}]$ de $[a\,;\,b']$. Ensuite on pose $x_i = x'_i + i \times h_{\min}$. De cette manière, on aura

$$x_{i+1} - x_i = (x'_{i+1} + (i+1)h_{\min}) - (x'_i + ih_{\min}) = (x'_{i+1} - x'_i) + h_{\min} \geqslant h_{\min}.$$

Remarquons que la construction n'est possible que si l'écart minimal, multiplié par le nombre de sous-intervalles, n'excède pas la longueur totale de l'intervalle à subdiviser (si on essaie de couper $[0\,;\,10]$ en quatre intervalles de longueur au moins 3, « ça ne rentre pas »).

```
def SubdivisionAléaMin(a, b, n, Pas_min) :
    if (n - 1) * Pas_min > abs(b - a) :
        raise Exception("Intervalle trop court.")
    Faux_b = b - (n - 1) * Pas_min
    L = SubdivisionAléa(a, Faux_b, n)
    return [L[i] + i * Pas_min for i in range(n)]
```

```
>>> SubdivisionAléaMin(0, 10, 5, 2)
[0, 2.778624980324009, 4.835124954636804, 7.751552652621818, 10]
```

40°

Exercice 7 — Soit n un entier strictement positif. Écrire un programme `Intermédiaires(n)` qui construit et renvoie une liste croissante constituée des entiers de 0 à n, et, entre chacun d'eux, un réel aléatoire.

```
>>> Intermédiaires(5)
[0, 0.644393015498002, 1, 1.761713892572682, 2, 2.066726643158812, 3,
 3.931244536819314, 4, 4.199760524383134, 5]
```

40°

Exercice 8 — Écrire un programme `SubdivisionDeuxPas(a, b, p1, p2)` qui, construit une liste de nombres démarrant à a, et finissant au plus à b, en avançant alternativement du pas p_1 et du pas p_2.

```
>>> SubdivisionDeuxPas(0, 10, 2, -1)
[0, 2, 1, 3, 2, 4, 3, 5, 4, 6, 5, 7, 6, 8, 7, 9, 8, 10, 9]
```

80°

Exercice 9 — Écrire un programme `PartitionAléaRéels(n)` qui construit et renvoie liste de n réels positifs aléatoires, dont la somme vaut 1.

```
>>> PartitionAléaRéels(3)
[0.3406514546217752, 0.22504024169859374, 0.43430830367963097]
```

Puis écrire un programme `PartitionAléaEntiers(n)` qui construit et renvoie une liste de n entiers naturels aléatoires dont la somme vaut 100.

```
>>> PartitionAléaEntiers(4)
[13, 38, 3, 46]
>>> PartitionAléaEntiers(10)
[15, 5, 8, 0, 1, 26, 2, 2, 10, 31]
```

§6. Méthode d'Euler (1er acte)

Commençons avec la fonction exponentielle : elle est définie comme l'unique fonction dérivable $y : \mathbf{R} \to \mathbf{R}$ telle que $y'(t) = y(t)$ pour tout réel t (ça, c'est une *équation différentielle*) et $y(0) = 1$ (et ça, c'est une *condition initiale*).

On souhaite, à partir de cette définition, en calculer des valeurs approchées sur l'intervalle $[0\,;\,t_{\max}]$. Ça fait une infinité d'images à calculer, c'est trop : on va calculer seulement n valeurs, pour un n qu'on choisira. Posons $h = t_{\max}/n$ et $x_i = i \times h$. Pour chaque indice i, on cherche une valeur approchée y_i de la solution théorique $y(t_i)$.

On procède de proche en proche, en utilisant une *approximation affine* (c'est l'autre petit nom des « développements limités à l'ordre un ») : si h est petit, alors

$$y(t_{i+1}) = y(t_i + h) \simeq y(t_i) + h \times y'(t_i) = y(t_i) + h \times y(t_i)$$

(puisqu'ici $y'(t) = y(t)$ pour tout t). On part de $y(t_0)$, qu'on connaît exactement : c'est $y(0) = 1$. Donc on pose $y_0 = 1$. Ensuite, on avance d'un pas :

$$y(t_1) \simeq y(t_0) + h \times y(t_0) \simeq y_0 + h \times y_0.$$

On pose donc $y_1 = y_0 + h \times y_0$. Et on avance encore d'un pas :

$$y(t_2) \simeq y(t_1) + h \times y(t_1) \simeq y_1 + h \times y_1,$$

donc on pose $y_2 = y_1 + h \times y_1$. Et ainsi de suite. Remarquons que les erreurs se cumulent : en effet, la formule est en elle-même une approximation, et en plus dans cette formule, on remplace les quantités qui apparaissent par des valeurs approchées. Rédigeons le programme correspondant : la variable y représente à chaque étape y_i, et la variable t représente à chaque étape t_i.

```
def Exponentielle(t_max, n) :
    absc = [0.0] ; t = 0.0
    ords = [1.0] ; y = 1.0
    h = t_max / n
    for i in range(n) :
        y += h * y
        ords.append(y)
        t += h
        absc.append(t)
    plot(absc, ords, ".", color = ROUGE)
```

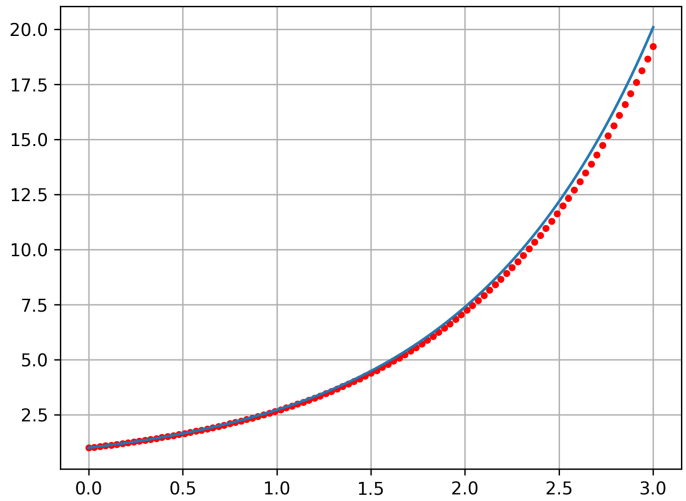
Remarque.
Ici on a divisé la longueur $t_{\max}$ de l'intervalle par n, ce qui veut dire qu'on a n sous-intervalles et donc $n+1$ points. On en a seulement calculé n, car le premier est donné : c'est la condition initiale $y_0 = y(0) = 1$.

Traçons en bleu la courbe théorique, et en rouge les points $(t_i\,;\,y_i)$. On voit qu'on s'écarte progressivement de la solution théorique. Cependant, si on augmente n, cette erreur diminue (et tend vers zéro).

```
# Résolution numérique
t_max = 3.0
n     = 100
Exponentielle(t_max, n)

# Valeurs théoriques
from numpy import exp
absc = linspace(0, t_max, NB_POINTS)
ords = [exp(x) for x in absc]
plot(absc, ords)

# Finalisation
grid(True) ; show()
```



Passons au cas général. Pour utiliser l'algorithme (qui s'appelle la *méthode d'Euler*), on a besoin d'un *problème de Cauchy*, c'est-à-dire une équation différentielle et une condition initiale. La forme générale est

$$\begin{cases} y'(t) = \mathrm{F}(y(t), t) \quad \text{pour } t \in \mathrm{I}, \\ y(t_0) = y_0, \end{cases}$$

avec I un intervalle ouvert contenant t_0. La fonction F représente l'équation différentielle (on va voir ça dans un instant), et t_0 et y_0 sont donnés. Pour la résolution, on prend évidemment $t_{\max} \in \mathrm{I}$. Avec les mêmes notations que précédemment (y_i la valeur approchée de la solution théorique $y(t_i)$) la formule d'itération s'écrit

$$y_{i+1} = y_i + h \times \mathrm{F}(y_i, t_i),$$

formule qu'on obtient à partir de l'approximation $y(t_i + h) \simeq y(t_i) + h \times y'(t_i)$. Voici déjà le programme qui trace la courbe de la solution approchée (cette fois-ci on relie les points).

```
def Euler(F, y0, t0, tmax, n) :
    absc = [t0] ; t = t0
    ords = [y0] ; y = y0
    h = (tmax - t0) / n
    for i in range(n) :
        y += h * F(y, t)
        ords.append(y)
        t += h
        absc.append(t)
    plot(absc, ords, color = BLEU)
```

Venons-en à cette fonction F. C'est une fonction de deux variables, x et t, telle que $y'(t) = \mathrm{F}(y(t), t)$ (autrement dit quand on remplace x par $y(t)$, on obtient l'équation différentielle). Par exemple, avec $\mathrm{F}(x, t) = x$, on obtient

$$y'(t) = \mathrm{F}(y(t), t) = y(t),$$

c'est l'équation différentielle de la fonction exponentielle.

Voyons un autre exemple pour tester le programme `Euler` :

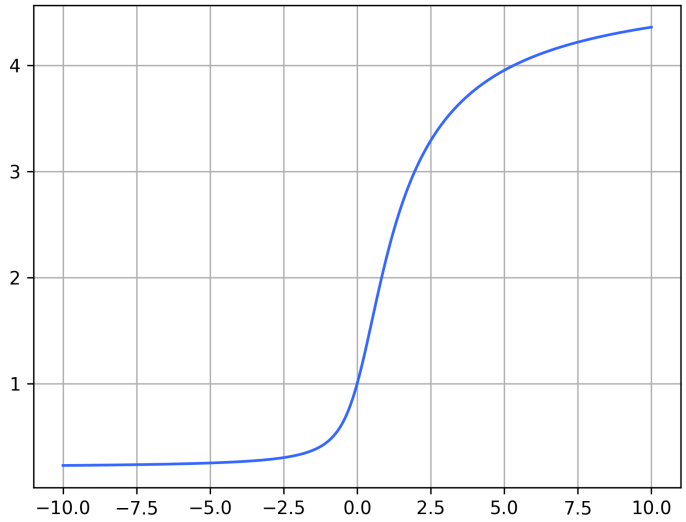
$$\begin{cases} y'(t) = \dfrac{y(t)}{t^2 + 1} \quad \text{pour } t \in \mathbf{R}, \\ y(0) = 1. \end{cases}$$

On commence par définir les paramètres et la fonction F qui représente l'équation différentielle.

```
y0 = 1.0
t0 = 0.0
n = 1000
def F(x, t) :
    return x / (t ** 2 + 1)
```

On ne peut pas tracer la courbe sur $\mathbf{R}$ tout entier (il faut bien s'arrêter à un moment !) donc on va prendre $[-10\,;\,10]$. Comme la résolution numérique se fait toujours à partir de t_0, et que celui-ci est au milieu de l'intervalle, on fait en deux fois : d'abord sur $[t_0\,;\,10]$ et ensuite sur $[-10\,;\,t_0]$. Pour le deuxième cas le pas sera négatif, mais ça n'a aucune importance. On obtient deux morceaux de courbe, mais si on les peint de la même couleur, on n'en voit qu'un.

```
tmax = 10.0
Euler(F, y0, t0, tmax, n)
tmax = -10.0
Euler(F, y0, t0, tmax, n)
grid(True) ; show()
```



Remarque.

On utilise toujours `grid(True)` et `show()` *après avoir construit toutes les courbes*. C'est la raison pour laquelle on ne les a pas fait figurer *dans* le programme `Euler` : on peut ainsi l'appeler plusieurs fois pour construire plusieurs morceaux de courbes, et les utiliser pour afficher le résultat une fois toutes les constructions faites.

40°

Exercice 10 — Pour chaque problème de Cauchy ci-dessous, tracer avec Python la courbe représentative de sa solution.

1) $\begin{cases} y'(t) - y(t) = t & \text{pour } t \in \mathbf{R}, \\ y(0) = 1. \end{cases}$

2) $\begin{cases} y'(t) = y(t) - \dfrac{1}{t} & \text{pour } t \in]0\,;\,+\infty[, \\ y(1) = 0. \end{cases}$

3) $\begin{cases} y'(t) = \sqrt{y(t)^2 + 1} & \text{pour } t \in \mathbf{R}, \\ y(0) = 1. \end{cases}$

4) $\begin{cases} y'(t) = \cos(y(t)) & \text{pour } t \in \mathbf{R}, \\ y(0) = 0. \end{cases}$

60°

Exercice 11 — On considère le problème de Cauchy

$$\begin{cases} y'(t) = \dfrac{1}{t} & \text{pour } t \in \mathbf{R}, \\ y(1) = 0. \end{cases}$$

1) Quelle est la fonction solution de ce problème ?

2) Tracer sa courbe représentative sur l'intervalle [0,01 ; 10].

3) Sans utiliser la fonction `ln`, calculer une valeur approchée de ln(3) à 0,001 près.

70°

Exercice 12

1) Sur un même graphique, tracer les solutions des problèmes de Cauchy

$$\begin{cases} y'(t) = y(t) & \text{pour } t \in \mathbf{R}, \\ y(0) = \alpha. \end{cases}$$

pour différentes valeurs du paramètre α.

2) Les courbes se coupent-elles ?

3) Recommencer avec une autre équation différentielle. Y a-t-il des cas où les courbes se coupent ?