Introduction à Numpy et Matplotlib

Un *tableau* est une structure de données ayant les particularités suivantes :

- les données sont du même type ;
- cela occupe des emplacements contigus en mémoire ;
- il en résulte un accès rapide aux données ;
- une contrainte : on doit définir à l'avance la taille du tableau pour réserver le bloc mémoire nécessaire.

La structure de tableau est pertinente lorsque l'on souhaite manipuler des données de nature homogène et que les opérations que l'on souhaite effectuer sont des opérations de consultation des cases du tableau ou d'affectation d'une case du tableau. Par exemple, le fait de chercher le minimum d'une liste d'entiers non triée. En terme de complexité, l'unité de mesure est la «longueur» du tableau.

Certaines opérations élémentaires ont un coût au moins linéaire car elle demandent une recopie d'une partie du tableau. Par exemple :

- supprimer une case du tableau ou en insérer une ;
- sélectionner un sous-tableau ;
- trier des éléments.

On s'intéresse dans la suite au module `numpy` qui permet de manipuler des objets numériques organisés en tableaux. On importe les fonctions du module `numpy` avec :

```
import numpy
```

Cela permet alors d'utiliser l'intégralité des fonctions du module (en les faisant précéder par le préfixe `numpy`) :

```
a = numpy.cos(numpy.pi/4)
```

Pour alléger l'écriture du préfixe, on peut créer un alias lors de l'importation :

```
import numpy as np
a = np.cos(np.pi/4)
```

Il se peut que l'on ait besoin de n'utiliser qu'un nombre restreint de fonctions du module :

```
from numpy import cos, sin, pi, ...
```

Elles sont alors accessibles sans les faire précéder du nom du module (mais s'il existait au préalable des fonctions du même nom alors ces dernières se retrouvent écrasées par les nouvelles fonctions importées). Il existe aussi la commande :

```
from numpy import *
```

qui importe toutes les fonctions en permettant ainsi de se passer du préfixe.

```
from numpy import *
a = cos(pi/4)
```

1 - Notion de tableau Numpy

■ Le module `numpy` apporte un type d'objet appelé *tableau* (*array* en anglais) dont la syntaxe de base est la suivante :

```
np.array(liste,dtype = typ)
```

où `liste` est une liste de valeurs et `typ` le type de données (par exemple *float*). Le paramètre dtype est optionnel : par défaut, en cas d'hétérogénéité de type, toutes les données de la liste seront automatiquement converties vers le type le plus fort (des flottants par exemple si la liste contient des flottants et des entiers).

```
>>> a = np.array([1,2,3])
>>> a
array([1, 2, 3])
>>> b = np.array([1,2.5,'a'])
>>> b
array(['1','2.5','a'])
```

L'implémentation en mémoire des tableaux est optimisée par rapport à celle des listes. Cela permet d'accéder ou de modifier plus rapidement les valeurs d'un élément, ce qui se révèle essentiel dans les méthodes numériques de calcul où les tableaux contiennent souvent des dizaines de milliers de valeurs.
La désignation des éléments d'un tableau est identique à celle des listes. Comme ces dernières, les tableaux sont des objets mutables : on peut modifier un élément dans un tableau.

```
>>> a = np.array([1,2,3,4,5])
>>> a[0]
1
>>> a[1] = 7
>>> a
array([1,7,3,4,5])
```

Si le type du nouvel élément n'est pas identique à ceux des éléments déjà présents dans le tableau, soit l'élément est converti, soit cela engendre une erreur.

```
>>> a = np.array([1,2,3,4,5])
>>> a[2] = 8.5
>>> a
np.array([1,2,8,4,5])
>>> a[3] = 'pangolin'
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-28-5c6c611e4369> in <module>()
----> 1 a[3] = 'pangolin'
ValueError: invalid literal for int() with base 10: 'pangolin'
```

■ Pour créer un tableau, on peut utiliser une syntaxe analogue à celle des listes.

```
>>> a = np.array([k for k in range(10)])
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

On peut transformer une liste en tableau et inversement.

```
>>> a = np.array([k for k in range(10)])
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> b = list(a)
>>> b
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> c = np.array(b)
>>> c
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

On peut créer des tableaux particuliers :

— `np.zeros(n,dtype = typ)` → tableau de n zéros dont le type est `typ` (le type par défaut est `float64`) ;

— `np.arange(start,stop,step)` → analogue à l'opérateur comme `range` mais tolérant des arguments réels (la valeur `stop` n'est pas comprise) ;

— `np.linspace(start,stop,num,endpoint = bool)` → tableau de `num` valeurs espacées régulièrement allant de `start` à `stop` (par défaut `num` = 50 et la valeur `stop` est comprise si `endpoint` est mis à `True` ce qui est le cas par défaut).

```
>>> np.zeros(10,dtype=int)
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
>>> np.arange(1,2,0.1)
array([1. , 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9])
>>> np.linspace(1,2,10,endpoint=False)
array([1. , 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9])
>>> np.linspace(1 , 2 , 10)
array([ 1. , 1.11111111, 1.22222222, 1.33333333, 1.44444444, 1.55555556, 1.66666667,
    1.77777778, 1.88888889, 2.  ] )
>>> np.linspace(1,2,11)
array ([1. , 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 2. ])
```

■ Les opérations algébriques usuelles sont effectuées «composante par composante» :

```
>>> a = np.array([0, 1, 2, 2, 3, 5])
>>> b = np.array([1, 2, 0, 2, 5, 3])
>>> a+b
array([1, 3, 2, 4, 8, 8])
>>> a*b
array([0,  2,  0,  4, 15, 15])
```

Par ailleurs, l'application d'une fonction `f` à un tableau `T` produit un tableau `f(T)` constitué des images de chacun des éléments du tableau `T` par la fonction `f` :

```
>>> a = np.array([0, 1, 2, 2, 3, 5])
>>> a**2
array([0, 1, 4, 4, 9, 25])
>>> 1/(a+1)
array([1. , 0.5, 0.33333333, 0.33333333, 0.25, 0.16666667])
>>> np.floor(np.exp(a))
array([1. ,2. , 7. , 7. , 20. , 148. ])
```

2 - Tableaux multidimensionnels

■ Les tableaux bidimensionnels permettent de considérer des *matrices* dont les intérêts opérationnels sont multiples : résolution des systèmes linéaires, changement de bases, etc. Pour créer un tableau bidimensionnel de taille $m \times n$ (c'est-à-dire comportant m lignes et n colonnes), on peut utiliser la syntaxe suivante :

```
tab = np.array([[liste_1],...,[liste_m]])
```

où chacune des listes `liste_k` comporte n éléments. Par exemple :

```
>>> tab1 = np.array([[1,2,3,4],[5,6,7,8],[9,10,11,12]])
>>> tab1
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
```

On peut également créer des tableaux de tailles supérieures ; par exemple pour un tableau tridimensionnel :

```
>>> tab2 = np.array([[[1,2],[3,4]],[[5,6],[7,8]],[[9,10],[11,12]]])
>>> tab2
array([[[ 1,  2],
        [ 3,  4]],
       [[ 5,  6],
        [ 7,  8]],
       [[ 9, 10],
        [11, 12]]])
```

Pour considérer un élément d'un tableau multidimensionnel, on utilise un indice multiple :

```
>>> tab1[1][2]
7
>>> tab1[1,2]  # Syntaxe alternative
7
>>> tab2[1][0][1]
6
>>> tab2[1,0,1]   # Syntaxe alternative
6
```

■ La lecture ou l'écriture d'éléments d'un tableau peut se faire par coupes (*slices* en anglais), suivant une ou éventuellement plusieurs des dimensions du tableau. Le format d'une coupe dans un tableau `T` est le suivant `T[debut:fin:pas]` où `debut` désigne le premier indice de position (compris), `fin` le dernier indice de position (non compris) et `pas` la période de coupe.

```
>>> tab3 = np.array([k for k in range(10)])
>>> tab3
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
>>> tab3[0:11:3]
array([0, 3, 6, 9])
>>> tab3[-1:0:-3]
 array([10,  7,  4,  1])
>>> tab1
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
>>> tab1[1:3,0:2]
array([[ 5,  6],
       [ 9, 10]])
>>> tab1[1:3,:]
array([[ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
>>> tab1[:,1:3]
array([[ 2,  3],
       [ 6,  7],
       [10, 11]])
```

■ Comme pour les tableaux monodimensionnels, les opérations classiques sont appliquées «élément par élément».

```
>>> tab1*2
array([[ 2,  4,  6,  8],
       [10, 12, 14, 16],
       [18, 20, 22, 24]])
>>> tab2*tab2
array([[[  1,   4],
        [  9,  16]],
       [[ 25,  36],
        [ 49,  64]],
       [[ 81, 100],
        [121, 144]]])
```

```
>>> np.exp(tab2)
array([[[  2.71828183e+00,   7.38905610e+00],
        [  2.00855369e+01,   5.45981500e+01]],
       [[  1.48413159e+02,   4.03428793e+02],
        [  1.09663316e+03,   2.98095799e+03]],
       [[  8.10308393e+03,   2.20264658e+04],
        [  5.98741417e+04,   1.62754791e+05]]])
```

■ Listons maintenant quelques opérations à la syntaxe moins naturelle.

◦ *Transposition*

```
>>> tab1
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
>>> transpose(tab1)  # ou tab1.transpose(), ou tab1.T
array([[ 1,  5,  9],
       [ 2,  6, 10],
       [ 3,  7, 11],
       [ 4,  8, 12]])
```

◦ *Concaténation*

Cela consiste en une agrégation de tableaux de *tailles compatibles* : deux tableaux accolés verticalement doivent avoir le même nombre de colonnes ; deux tableaux accolés horizontalement doivent avoir le même nombre de lignes.

```
>>> tab3 = np.array([[13, 14, 15, 16]])   # pourquoi ces doubles crochets ?
>>> tab3
array([[13, 14, 15, 16]])
>>> tab4 = np.concatenate((tab1,tab3),axis=0)  # pourquoi ces parenthèses ?
>>> tab4                                  # axis=0 pour concatenation verticale
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12],
       [13, 14, 15, 16]])
>>> tab5 = np.concatenate((tab1,tab3),axis=1) # axis=1 pour concatenation
    horizontale
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-95-66e734811591> in <module>()
----> 1 tab5 = np.concatenate((tab1,tab3),axis=1)
ValueError: all the input array dimensions except for the concatenation axis
must match exactly
```

L'erreur est due à une incompatibilité entre le nombre de lignes de tab1 (3 lignes) et tab3 (1 ligne).

```
>>> tab5 = np.concatenate((tab1.T,tab3.T),axis=1)
>>> tab5
array([[ 1,  5,  9, 13],
       [ 2,  6, 10, 14],
       [ 3,  7, 11, 15],
       [ 4,  8, 12, 16]])
```

◦ *Produit matriciel*

Le produit matriciel entre tableaux numpy aux dimensions compatibles peut être réalisé par l'intermédiaire de la fonction dot.

```
>>> A = np.array([[1,2,3],[1,2,3]])
>>> A
array([[1, 2, 3],
       [1, 2, 3]])
>>> B = np.array([[1,3],[2,4],[1,3],[2,4]])
>>> B
array([[1, 3],
       [2, 4],
       [1, 3],
       [2, 4]])
>>> np.dot(B, A)
array([[ 4,  8, 12],
       [ 6, 12, 18],
       [ 4,  8, 12],
       [ 6, 12, 18]])
>>> np.dot(A, B)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
ValueError: shapes (2,3) and (4,2) not aligned: 3 (dim 1) != 4 (dim 0)
```

■ Les méthodes et attributs les plus usuels des tableaux numpy sont :

— `T.dtype` → type des données du tableau,

— `T.size` → taille du tableau, c'est-à-dire son nombre de cases (y compris dans le cas de tableaux multidimensionnels),

— `T.shape` → dimensions du tableau, c'est-à-dire tuple correspondant au nombre de lignes et au nombre de colonnes,

— `T.sum()`, `T.mean()`, `T.max()`, `T.mmin()` → somme, moyenne, maximum, minimum de toutes les valeurs de `T`,

— `T.argmax()` → plus petit indice `i` tel que `T[i] = T.max()` (idem avec `argmin()`),

— `T.round()` → arrondi de chaque valeur de `T` à la valeur entière la plus proche,

— `T.sort()` → tri des valeurs de `T` dans l'ordre croissant,

— `T.copy()` → copie totalement indépendante de `T`.

Exercice 1

Soit la fonction f qui à $x > 0$ associe $\lfloor \ln(x) \rfloor$.
Définir une fonction python `images(n)` donnant le tableau des valeurs des $f(k)$ pour $k \in [\![1, n]\!]$.
Définir une fonction python `imagesbis(n)` donnant le tableau des valeurs des $\frac{f(k)}{k}$ pour $k \in [\![1, n]\!]$.

Exercice 2

Soit la matrice carrée :

$$\mathrm{M}(n) = \begin{bmatrix} 0 & 1 & 2 & \cdots & n \\ 1 & 1 & 2 & & \\ 2 & 2 & 2 & \cdots & n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ n & \cdots & n & \cdots & n \end{bmatrix}$$

Écrire la fonction `M(n)` renvoyant le tableau semblable à la matrice ci-dessus.

3 - Représentation graphique

Il existe en Python des modules contenant des fonctions permettant de générer des figures en deux ou trois dimensions puis de les afficher ou les enregistrer sous différents formats. Nous allons nous intéresser pour le moment au tracé en deux dimensions. Le module à importer est `matplotlib.pyplot` que l'on abrège couramment en `plt` *via* la commande !

```
import matplotlib.pyplot as plt
```

Par défaut, au moment de l'importation, Python crée une «feuille de figures» numérotée 1 dans laquelle on peut insérer une ou plusieurs figures. On peut créer d'autres pages par la commande `plt.figure(n)` où `n` est entier et on navigue entre les pages par la même commande. Par défaut, on est dans la feuille 1 et les tracés sont toujours faits dans la figure courante. On détruit la feuille courante par la commande `plt.close()`.
Si l'on ne veut pas travailler avec le principe de «figure courante», on peut récupérer l'objet dans une variable par `fig=plt.figure(n)`. Dans ce cas, on peut invoquer les méthodes associées à l'objet `fig`. Par exemple, `fig.title("la fonction cos")` ou la commande `plt.title("la fonction cos")` auront le même effet si `fig` est la figure courante. La commande pour sauvegarder la feuille de figures courante sous forme d'un fichier pdf exploitable est `plt.savefig(nom_du_fichier)`. On affiche les différentes pages de figures dans des fenêtres interactives extérieures par `plt.show()`.

Exercice 3

Expérimenter ces instruction avec le code suivant :

```
plt.title("la fonction cos")
X = np.linspace(0,2*np.pi)
plt.plot(X,np.cos(X),'r:s')
plt.show()
```

La commande standard pour insérer une courbe dans la feuille courante est :

```
plt.plot ( listex, listey, ch_tracé , label = "nom_de_la_courbe" )
```

où `listex, listey` constituent les deux listes donnant les coordonnées des points à relier, le paramètre optionnel `label` permet de donner un nom à la courbe pour la légende et `ch_tracé` est une chaîne de caractères précisant le type du tracé *via* trois informations :

- le premier caractère de `ch` indique la couleur via la première lettre du nom de la couleur en anglais b(lue),r(ed),g(reen),c(yan),m(agenta),y(ellow),k(black),w(hite).
- les caractères suivant indiquent la façon de relier les points ('-' pour une ligne, '- -' pour des pointillés tirets, ' :' pour des pointillés points).
- le dernier caractère indique comment sont signalés sur le graphique les points de la courbe donnés explicitement *via* les listes `listex` et `listey` ('o' pour des petits cercles, 's' pour des petits carrés, 'v' pour des triangles, des pixels par défaut).

Par défaut, le tracé se fait dans la couleur courante (qui change à chaque plot) et en trait plein. Les noms des courbes apparaissent dans la légende générée par la commande `plt.legend()`. On peut forcer le tracé d'une grille en filigrane dans la figure par `plt.grid(True)`.

Exercice 4

On suppose donnée une liste de points (représentés par des couples de réels (x_i, y_i) en Python) représentant par exemple des mesures expérimentales. On suppose que, idéalement, les quantités x et y doivent être liées par une relation affine $y = ax + b$ (où a et b sont des constantes). On veut déterminer les valeurs optimales de a et b c'est-à-dire celles qui minimisent la quantité $\sum_{i=1}^{n} (y_i - ax_i - b)^2$.

On admet que les formules mathématiques donnant a et b sont :

$$\mathrm{M}_x = \frac{1}{n}\sum_{i=1}^{n} x_i \quad \mathrm{M}_y = \frac{1}{n}\sum_{i=1}^{n} y_i \quad \mathrm{M}_{xy} = \frac{1}{n}\sum_{i=1}^{n} x_i y_i \quad \mathrm{M}_{x^2} = \frac{1}{n}\sum_{i=1}^{n} x_i^2$$

$$a = \frac{\mathrm{M}_{xy} - \mathrm{M}_x \mathrm{M}_y}{\mathrm{M}_{x^2} - \mathrm{M}_x^2} \quad b = \frac{\mathrm{M}_y \mathrm{M}_{x^2} - \mathrm{M}_x \mathrm{M}_{xy}}{\mathrm{M}_{x^2} - \mathrm{M}_x^2}.$$

Écrire une fonction représentant sur un même graphique la droite d'équation $y = ax + b$ et les points.

Exercice 5

On considère la suite (u_n) dépendant d'un paramètre $a \in [0,4]$ et définie par la relation de récurrence suivante :

$$u_0 = 0,4 \quad \text{et} \quad u_{n+1} = f(u_n, a) \quad \text{où} \quad f(x,a) = a\,x\,(1-x).$$

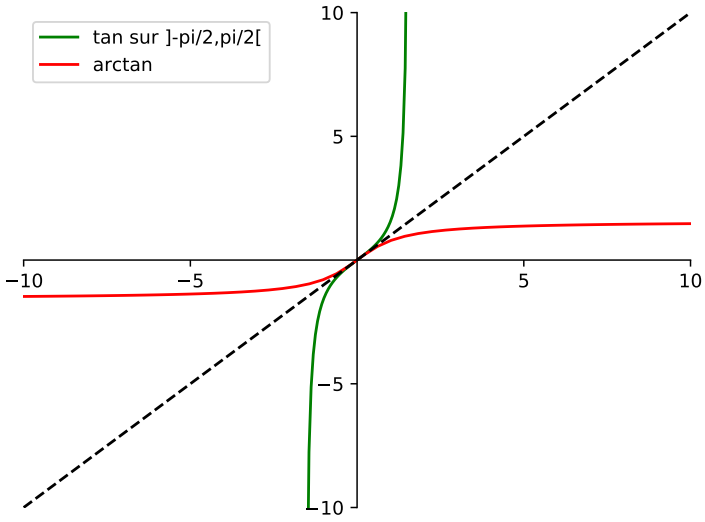
1. Écrire une fonction `f` de deux arguments x et a renvoyant le réel $f(x,a)$.
2. Écrire une fonction `U` de deux arguments, un entier naturel N et un réel a, et renvoyant la liste des N premiers termes de la suite u_n.
3. Tracer sur un même graphique les 30 premiers termes de la suite (u_n) en fonction de n pour $a = 2,7$, $a = 3,2$, et $a = 3,7$.
4. Tracer sur un même graphique (u_n) en fonction de a, pour a variant de 0 à 4 avec un pas de 0,001, et ceci pour les valeurs de n comprises entre 200 et 250.

Par défaut, python adapte les axes aux valeurs des abscisses fournies pour la première courbe. On peut modifier les limites du tracé par `plt.xlim(val_min,val_max)` et `plt.ylim(val_min,val_max)` ou par `plt.axis([valx_min,valx_max,valy_min,valy_max])`.
On donne un nom aux axes par `plt.xlabel('nom_abscisse')` et `plt.xlabel('nom_ordonnées')`. On peut aussi modifier les graduations placées automatiquement sur l'axe par les commandes `plt.xticks(liste)` et `plt.yticks(liste)` où `liste` est la liste des valeurs que l'on veut voir placées sur l'axe.
Par défaut, le tracé s'effectue dans un cadre formé de quatre bords ('right','top','bottom','left') ; les bords bas et gauche indiquent respectivement les abscisses et les ordonnées. Cette configuration de base n'est pas adaptée pour le tracé traditionnel des courbes en mathématiques où l'on trace des courbes dans un repère orthonormé avec des axes passant par le point (0,0). Pour cela if faut modifier l'objet "cadre", d'abord en le récupérant dans une variable puis en effaçant les bords inutiles (haut et droite par exemple), en limitant les graduations sur les deux bords qu'on conserve (gauche et bas) et enfin en déplaçant ces bords sur l'abscisse ou l'ordonnée 0. Voici les commandes adaptées :

```
ax = plt.gca()
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')
ax.xaxis.set_ticks_position('bottom')
ax.yaxis.set_ticks_position('left')
ax.spines['left'].set_position(('data',0))
ax.spines['bottom'].set_position(('data',0))
```

Exercice 6

Reproduire la figure suivante :



On peut tracer sur une même feuille plusieurs figures différentes (on parle de sous-figures). Il faut pour cela les organiser en lignes et colonnes (par exemple, trois lignes de deux figures). Les différentes figures sont alors numérotées dans l'ordre en commençant par la première ligne. Par exemple, pour six figures organisées en trois lignes et deux colonnes,

$$\begin{array}{cc} F_1 & F_2 \\ F_3 & F_4 \\ F_5 & F_6 \end{array}$$

Pour passer une sous-figure en figure courante, on utilise la commande

```
plt.subplot( nbre_lignes , nbre_colonnes , numéro )
```

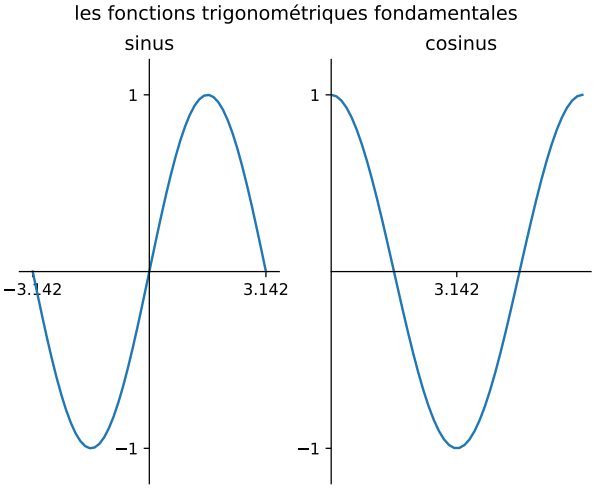
Le tracé se fait alors comme d'habitude avec `plt.plot`. La commande `plt.suptitle(nom)\verb`, où `nom` est une chaîne de caractères, permet de donner un nom global à la page de figure courante tandis que la commande `plt.title(nom)` donne un nom à la figure (ou sous-figure) en cours. Par exemple, le code suivant génère la page de figures qui suit :

```
X = np.linspace(-np.pi,np.pi)
XX = np.linspace(0,2*np.pi)
plt.suptitle("les fonctions trigonométriques fondamentales")

plt.subplot(1,2,1)
ax = plt.gca()
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')
ax.xaxis.set_ticks_position('bottom')
ax.yaxis.set_ticks_position('left')
ax.spines['left'].set_position(('data',0))
ax.spines['bottom'].set_position(('data',0))
plt.xticks([-np.pi,np.pi])
plt.yticks([-1,1])
plt.axis([-3.5,3.5,-1.2,1.2])
plt.plot(X,np.sin(X))
plt.title("sinus")
```

```
plt.subplot(1,2,2)
plt.plot(XX,np.cos(XX))
ax = plt.gca()
ax.spines['top'].set_color('none')
ax.spines['right'].set_color('none')
ax.xaxis.set_ticks_position('bottom')
ax.yaxis.set_ticks_position('left')
ax.spines['bottom'].set_position(('data',0))
plt.xticks([-np.pi,np.pi])
# plt.xticks([0,np.pi,2*np.pi],[r'$0$',r'$\pi$',r'$2\pi$'])
plt.yticks([-1,1])
plt.axis([0,6.5,-1.2,1.2])
plt.title("cosinus")

plt.show()
```



Exercice 7

Reproduire la figure suivante :