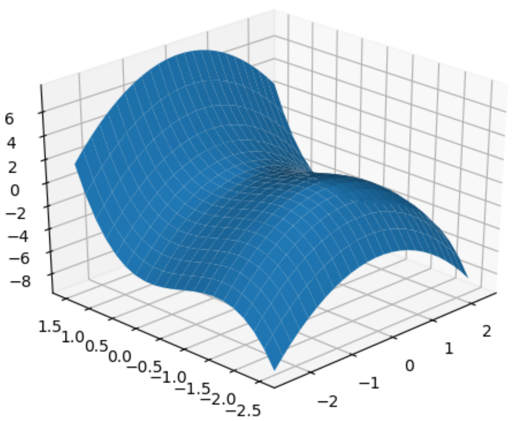
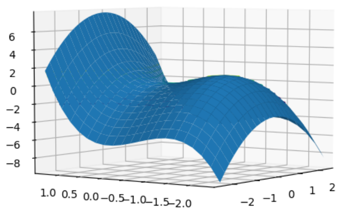
Python : numpy et matplotlib avec deux variables

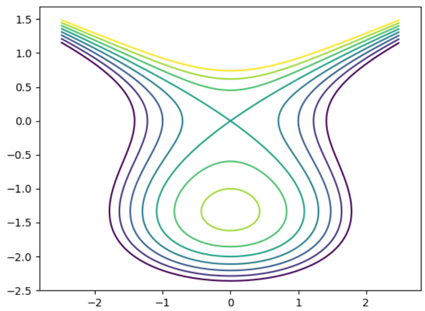
Vidéo ■ partie 4.1. Numpy
Vidéo ■ partie 4.2. Matplotlib



Le but de ce chapitre est d'approfondir notre connaissance de numpy *et* matplotlib *en passant à la dimension 2. Nous allons introduire les tableaux à double entrée qui sont comme des matrices et visualiser les fonctions de deux variables.*

Voici le graphe de la fonction $f(x,y) = y^3 + 2y^2 - x^2$ (ci-dessus et ci-dessous à gauche, dessinés sous deux points de vue différents) ainsi que ses lignes de niveau dans le plan (ci-dessous à droite).





1. Numpy (deux dimensions)

Nous avons vu comment définir un vecteur (un tableau à une dimension), comme par exemple `[1 2 3 4]`. Nous allons maintenant étudier les tableaux à deux dimensions.

1.1. Tableau

Un tableau à deux dimensions est comme une matrice (ou un tableau à double entrée).

- **Définition.** Un tableau se définit comme une suite de lignes

```
A = np.array([[1, 2, 3], [4, 5, 6]])
```

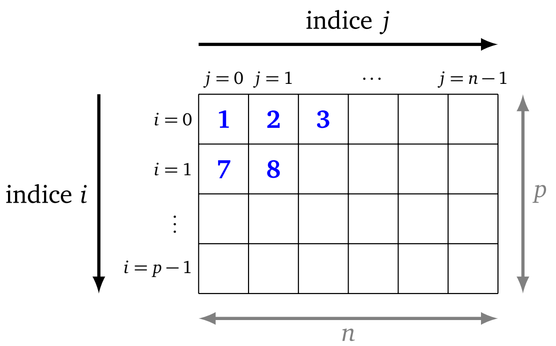
  (en ayant au préalable importé et renommé le module *numpy* en `np`) et s'affiche ainsi :

```
[[1 2 3]
 [4 5 6]]
```

  C'est un tableau à deux lignes et trois colonnes qui correspond à la matrice :

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}.$$

- **Taille.** On récupère la taille du tableau par la fonction `shape()`. Par exemple `np.shape(A)` renvoie ici $(2,3)$, pour 2 lignes et 3 colonnes.
- **Parcourir les éléments.** On accède aux éléments par des instructions du type `A[i,j]` où i est le numéro de ligne et j celui de la colonne. Voici le code pour afficher les éléments un par un.




```
p, n = np.shape(A)
for i in range(p):
    for j in range(n):
        print(A[i,j])
```

Tableau

- **Fonctions.** Les fonctions s'appliquent élément par élément. Par exemple `np.sqrt(A)` renvoie un tableau ayant la même forme, chaque élément étant la racine carrée de l'élément initial.

```
[[1.         1.41421356 1.73205081]
 [2.         2.23606798 2.44948974]]
```

- **Définition (suite).** `np.zeros((p,n))` renvoie un tableau de p lignes et n colonnes rempli de 0. La fonction `np.ones()` fonctionne sur le même principe.

1.2. Conversion tableau-vecteur

Il est facile et souvent utile de passer d'un tableau à un vecteur et réciproquement.

- **Tableau vers vecteur.** On obtient tous les éléments d'un tableau regroupés dans un vecteur par la commande d'aplatissement `flatten()`. Par exemple si

```
A = np.array([[1, 2, 3], [4, 5, 6]])
```

  alors la commande `X = A.flatten()` renvoie le vecteur X :

```
[1 2 3 4 5 6]
```

- **Vecteur vers tableau.** L'opération inverse se fait avec la fonction `reshape()` qui prend en entrée la nouvelle taille désirée. Par exemple : `X.reshape((2,3))` redonne exactement `A`. Par contre `X.reshape((3,2))` renvoie un tableau à 3 lignes et 2 colonnes :

```
[[1 2]
 [3 4]
 [5 6]]
```

1.3. Fonctions de deux variables

- **Évaluation sur des vecteurs.** Supposons que nous ayons défini une fonction *Python* de deux variables `f(x,y)`. Si `VX` et `VY` sont deux vecteurs de même taille, alors `f(VX,VY)` renvoie un vecteur composé des $f(x_i, y_i)$ pour x_i dans `VX` et y_i dans `VY`. Par exemple, si `VX` vaut `[1 2 3 4]` et `VY` vaut `[5 6 7 8]` alors `f(VX,VY)` est le vecteur de longueur 4 composé de $f(1,5)$, $f(2,6)$, $f(3,7)$, $f(4,8)$. Mais ceci n'est pas suffisant pour tracer des fonctions de deux variables.
- **Grille.** Pour dessiner le graphe d'une fonction $f(x, y)$, il faut calculer des valeurs $z = f(x, y)$ pour des (x, y) parcourant une grille. Voici comment définir simplement une grille à l'aide `meshgrid()`.

```
n = 5
VX = np.linspace(0, 2, n)
VY = np.linspace(0, 2, n)
X,Y = np.meshgrid(VX, VY)

def f(x,y):
    return x**2 + y**2

Z = f(X,Y)
```

- **Explications.**
  — `VX` est un découpage de l'axe des x en n valeurs.
  — `VY` est la même chose pour l'axe des y.
  — `X` et `Y` renvoyés par `meshgrid()` forment la grille. Ce sont des tableaux $n \times n$. Le premier représente les abscisses des points de la grille, le second les ordonnées. (Ce n'est pas très naturel mais c'est comme ça !)

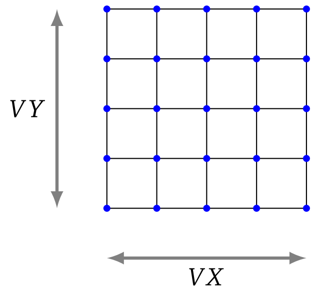


Tableau X

```
[[0.  0.5 1.  1.5 2. ]
 [0.  0.5 1.  1.5 2. ]
 [0.  0.5 1.  1.5 2. ]
 [0.  0.5 1.  1.5 2. ]
 [0.  0.5 1.  1.5 2. ]]
```

Tableau Y

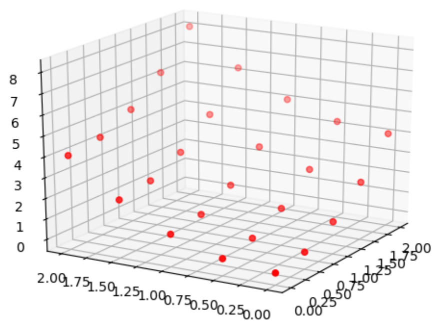
```
[[0.  0.  0.  0.  0. ]
 [0.5 0.5 0.5 0.5 0.5]
 [1.  1.  1.  1.  1. ]
 [1.5 1.5 1.5 1.5 1.5]
 [2.  2.  2.  2.  2. ]]
```

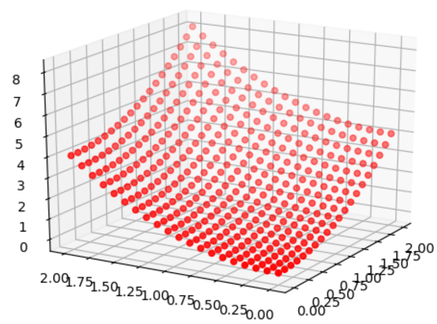
  — Enfin `Z = f(X,Y)` calcule les valeurs $z = f(x, y)$ pour tous les éléments de (x, y) de la grille et renvoie un tableau des valeurs.

Tableau Z

```
[[0.   0.25 1.   2.25 4.  ]
 [0.25 0.5  1.25 2.5  4.25]
 [1.   1.25 2.   3.25 5.  ]
 [2.25 2.5  3.25 4.5  6.25]
 [4.   4.25 5.   6.25 8.  ]]
```

- **Points.** Ainsi X, Y, Z forment une liste des n^2 points de l'espace dont on donne d'abord l'abscisse (dans X), puis l'ordonnée (dans Y), puis la hauteur (dans Z). Si on augmente la valeur de n, on commence à voir apparaître la surface d'équation $z = f(x, y)$. Ci-dessous le cas $n = 5$ à gauche et $n = 20$ à droite.





2. Un peu plus sur numpy

2.1. Ses propres fonctions

On peut définir ses propres fonctions. Dans le cas le plus simple, il n'y a rien de spécial à faire.

```
def ma_formule(x):
    return np.cos(x)**2 - np.sin(x)**2
```

Alors `ma_formule(X)` renvoie le résultat, que X soit un nombre, un vecteur ou un tableau.
On peut aussi utiliser les fonctions « lambda » :

```
f = lambda n:n*(n+1)/2
```

à utiliser sous la forme usuelle `Y = f(X)`.

2.2. Ses propres fonctions (suite)

Par contre, la fonction suivante ne sait traiter directement ni un vecteur ni un tableau, à cause du test de positivité.

```
def valeur_absolue(x):
    if x >= 0:
        return x
    else:
        return -x
```

Un appel `valeur_absolue(x)` fonctionne lorsque x est un nombre, mais si X est un vecteur ou un tableau alors un appel `valeur_absolue(X)` renvoie une erreur. Il faut « vectoriser » la fonction par la commande :

```
vec_valeur_absolue = np.vectorize(valeur_absolue)
```

Maintenant `vec_valeur_absolue(X)` fonctionne pour les nombres, les vecteurs et les tableaux.

Remarque.
Il peut être utile de préciser que chaque composante de sortie doit être un nombre flottant :

```
vec_fonction = np.vectorize(fonction, otypes=[np.float64])
```

2.3. Le zéro et l'infini

Le module *numpy* gère bien les problèmes rencontrés lors des calculs. Prenons l'exemple du vecteur X suivant :

```
[-1 0 1 2 3]
```

- La commande `1/X` renvoie le vecteur :
  ```
  [-1. inf 1. 0.5 0.33333333]
  ```
  La commande émet un avertissement (mais n'interrompt pas le programme). Comme valeur de 1/0 elle renvoie `inf` pour l'infini ∞.
- La commande `Y = np.log(X)` donne un vecteur Y :
  ```
  [ nan -inf 0. 0.69314718 1.09861229]
  ```
  où « `nan` » est une abréviation de *Not A Number* car le logarithme n'est pas défini pour des valeurs négatives et `-inf` représente $-\infty$ (qui est la limite en 0 du logarithme).
- La commande `Z = np.exp(Y)` donne `[nan 0. 1. 2. 3.]`. Ce qui est presque le vecteur X de départ et cohérent avec la formule $\exp(\ln(x)) = x$ pour $x > 0$.

2.4. Utilisation comme une liste

Soit le vecteur X défini par la commande :

```
X = np.linspace(0,10,num=100)
```

Pour récupérer une partie du vecteur, la syntaxe est la même que pour les listes *Python*.

- Élément de rang 50 : `X[50]`.
- Dernier élément : `X[-1]`.
- Éléments de rang 10 à 19 : `X[10:20]`.
- Éléments du début au rang 9 : `X[:10]`.
- Éléments du rang 90 à la fin : `X[90:]`.

Voici des fonctionnalités moins utiles.

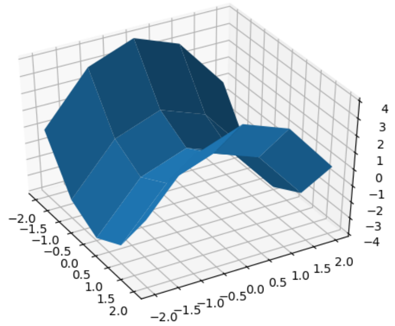
- Ajouter un élément à un vecteur avec `append()`. Par exemple si `X = np.arange(0,5,0.5)` alors la commande `Y = np.append(X,8.5)` construit un nouveau vecteur qui se termine par l'élément 8.5.
- Revenir à une liste. Utiliser la conversion `list(X)` pour obtenir une liste *Python* à partir d'un vecteur *numpy*.

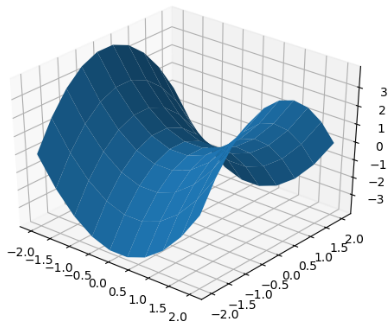
3. Matplotlib : deux variables

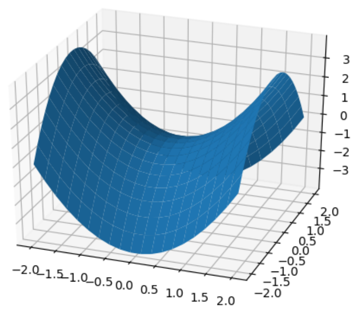
3.1. Graphes

On calcule d'abord une grille (X, Y) de points et les valeurs Z de la fonction sur cette grille. Ensuite le tracé du graphe se fait grâce à l'instruction `plot_surface(X, Y, Z)`.

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

n = 5
VX = np.linspace(-2.0, 2.0, n)
VY = np.linspace(-2.0, 2.0, n)
X,Y = np.meshgrid(VX, VY)

def f(x,y):
    return x**2-y**2

Z = f(X,Y)

fig = plt.figure()
ax = plt.axes(projection='3d')
ax.view_init(40, -30)

ax.plot_surface(X, Y, Z)

plt.show()
```

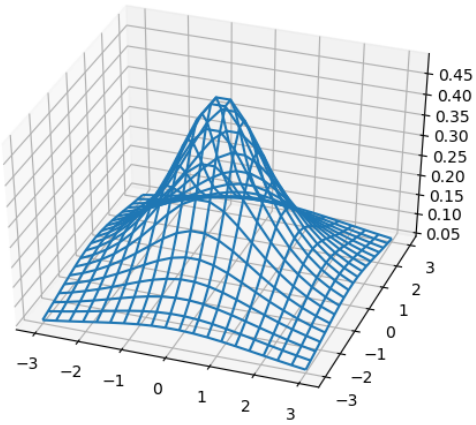






Si on augmente n, la grille est plus dense et la surface dessinée paraît plus lisse car il y a davantage de polygones. Les tracés de la « selle de cheval » ci-dessus sont effectués pour $n = 5$, $n = 10$, puis $n = 20$. L'affichage 3D permet de tourner la surface pour mieux l'appréhender. On peut aussi fixer le point de vue avec `view_init()`.

Il existe de multiples variantes et coloriages possibles.

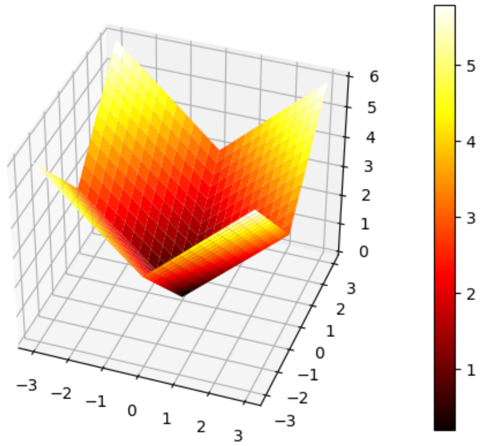


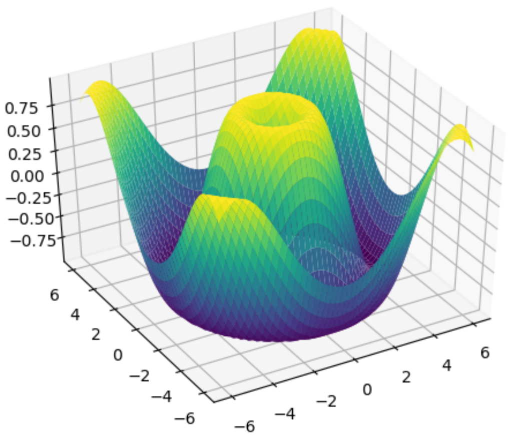
$$f(x,y)=\frac{1}{2+x^2+y^2}$$

`plot_wireframe(X, Y, Z)`

$f(x,y)=|x|+|y|$

Barre des couleurs



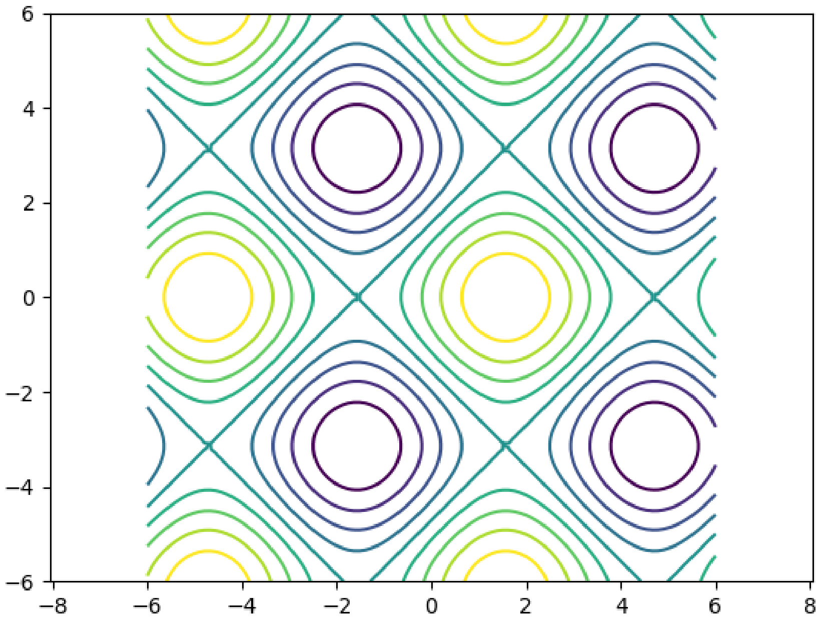


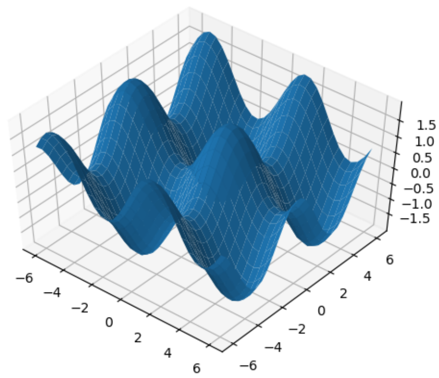
$$f(x,y)=\sin\left(\sqrt{x^2+y^2}\right)$$

3.2. Lignes de niveau

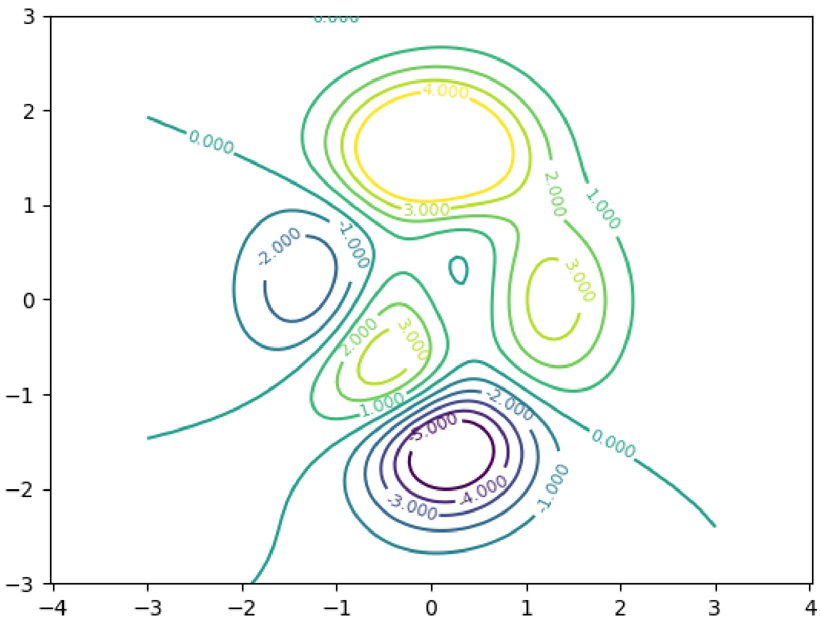
Il nous sera particulièrement utile de trouver les lignes de niveau d'une fonction f, en particulier pour trouver tous les (x,y) tels que $f(x,y)\geqslant 0$ par exemple. Le principe est similaire au tracé de la surface et s'effectue par la commande :

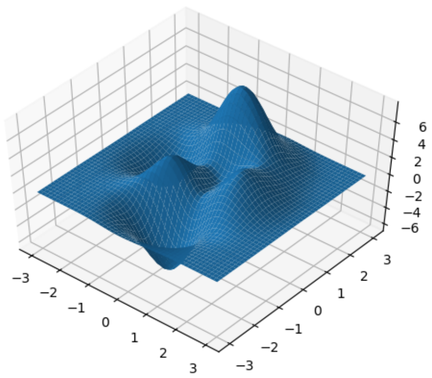
```
plt.contour(X, Y, Z)
```



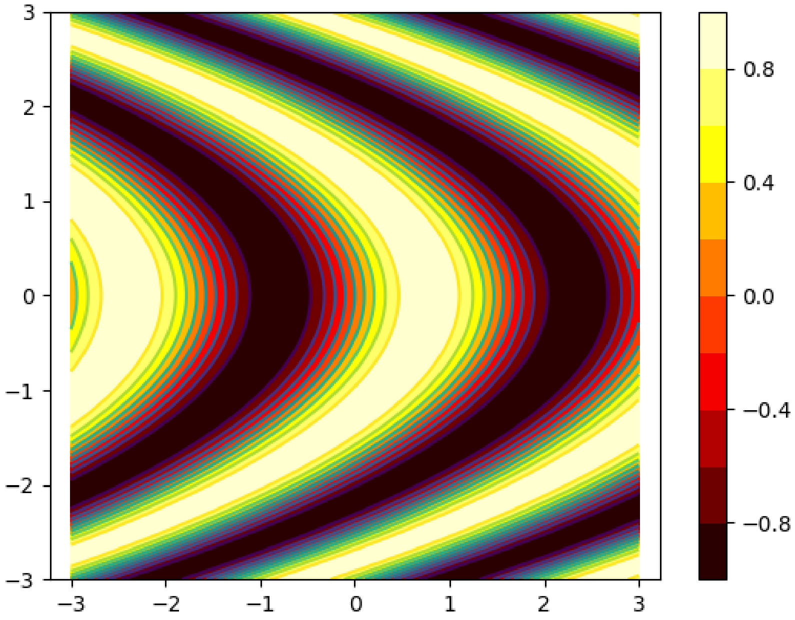


On peut préciser le nombre de niveaux tracés `plt.contour(X, Y, Z, nb_niveaux)` (ci-dessus). Le graphe en dimension 3 ci-contre.

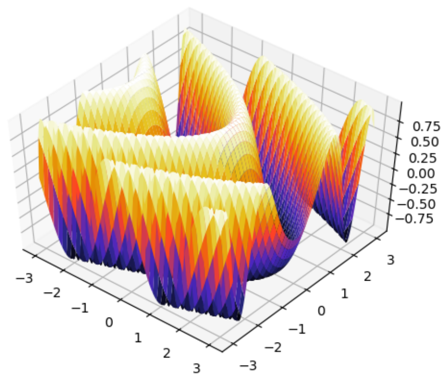




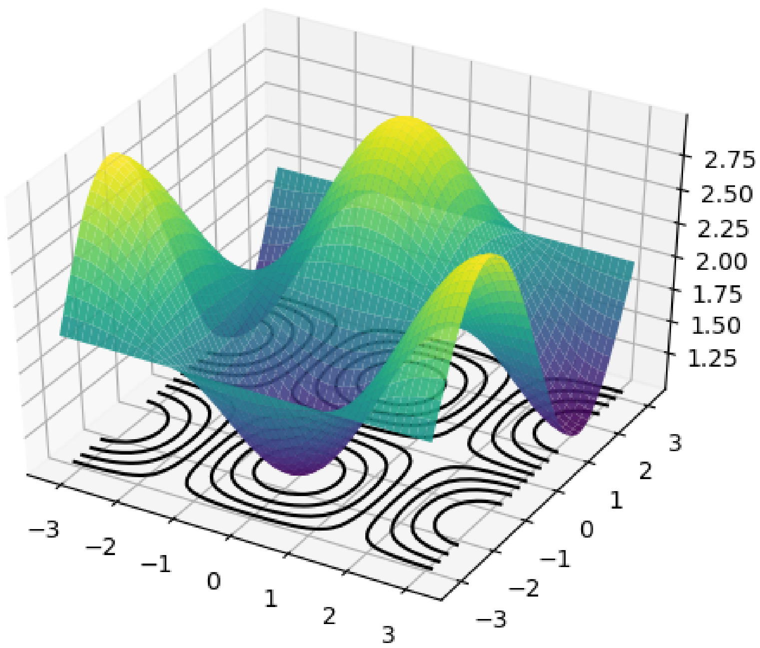
On peut spécifier les niveaux à afficher par `plt.contour(X, Y, Z, mes_niveaux)` où `mes_niveaux = np.arange(-5,5,1)` par exemple. On peut en profiter pour afficher la valeur du niveau, comme l'altitude sur une carte topographique (ci-dessus les lignes de niveau, ci-contre le graphe 3D).



`plt.contourf(X, Y, Z)` colorie le plan au lieu de tracer les lignes de niveau (ci-dessus les lignes de niveau, ci-contre le graphe 3D).



On peut même tracer les lignes de niveau sous la surface comme ci-dessous.


2.75
2.50
2.25
2.00
1.75
1.50
1.25
−3
−2
−1
0
1
2
3
3
2
1
0
−1
−2
−3