Surfaces

November 26, 2020

1 Tracé de surfaces

Marc Lorenzi

20 novembre 2020

[1]:
```
import matplotlib.pyplot as plt
from matplotlib.patches import Polygon
import math
from vect_utils import *
```

[3]:
```
plt.rcParams['figure.figsize'] = (8, 6)
```

1.1 1. Premiers pas

1.1.1 1.1 Surfaces paramétrées

Soit $F : \mathcal{D} \subset \mathbb{R}^2 \to \mathbb{R}^3$ où $\mathcal{D}$ est une partie du plan. L'ensemble $S = F(\mathcal{D})$ est une partie de l'espace, chaque point $M \in S$ étant *paramétré* par un (ou plusieurs) couple(s) $(u, v) \in \mathcal{D}$ tel(s) que $F(u, v) = M$. Lorsque la fonction F est suffisamment régulière, l'ensemble S ressemble au voisinage de chacun de ses points à une *surface*.

Nous nous proposons dans ce notebook de dessiner la surface S, paramétrée par F. Pour simplifier, nous supposerons que le domaine de définition de F, $\mathcal{D}$, est un rectangle $[a, b] \times [c, d]$, que nous représenterons en Python par le quadruplet (a, b, c, d).

Par exemple, un cylindre de base circulaire de rayon $R > 0$ et de hauteur $h > 0$ est l'ensemble des points $(x, y, z) \in \mathbb{R}^3$ tels que

$$(C) \quad \begin{cases} x^2 + y^2 = R^2 \\ 0 \leq z \leq h \end{cases}$$

Ce cylindre peut être paramétré par

$$(C) \quad \begin{cases} x &= R\cos u \\ y &= R\sin u \\ z &= v \end{cases}, \quad (u, v) \in \mathcal{D} = [0, 2\pi] \times [0, h]$$

De même, la sphère de centre O et de rayon $R > 0$ peut être paramétrée par les coordonnées sphériques :

$$(S) \quad \left\{ \begin{array}{rcl} x & = & R\cos u \cos v \\ y & = & R\cos u \sin v \\ z & = & R\sin u \end{array} \right. , \quad (u,v) \in \mathcal{D} = [-\frac{\pi}{2}, \frac{\pi}{2}] \times [0, 2\pi]$$

Pour définir le paramétrage de la sphère en Python, il faut nous donner à la fois la fonction de (u, v) qui paramètre notre surface, et l'ensemble de définition $\mathcal{D}$. Ce qui donne :

[4]:
```
def F_sphere(u, v, R=1):
    x = R * math.cos(u) * math.cos(v)
    y = R * math.cos(u) * math.sin(v)
    z = R * math.sin(u)
    return (x, y, z)

bornes_sphere = (- math.pi / 2, math.pi / 2, 0, 2 * math.pi)
```

1.1.2 1.2 Autres exemples

Nous utiliserons dans le notebook les exemples qui suivent. Pour chacun d'eux, nous définissons la fonction F qui paramètre la surface, ainsi que les bornes de son ensemble de définition.

Voici un paramétrage d'un *tore*.

[5]:
```
def F_tore(u, v):
    x = (3 + math.cos(u)) * math.cos(v)
    y = (3 + math.cos(u)) * math.sin(v)
    z = math.sin(u)
    return (x, y, z)

bornes_tore = (0, 2 * math.pi, 0, 2 * math.pi)
```

Voici un paramétrage de *l'hyperboloïde de révolution à une nappe*.

[6]:
```
def F_hyper(u, v):
    r = math.sqrt(1 + u ** 2)
    x = r * math.cos(v)
    y = r * math.sin(v)
    z = u
    return (x, y, z)

bornes_hyper = (-3, 3, 0, 2 * math.pi)
```

Nous utiliserons aussi les deux paramétrages `F_exp` et `F_sin`.

[7]:
```
F_exp = lambda u, v: (u, v, 2 * math.exp(-u ** 2 - v ** 2))
bornes_exp = (-2, 2, -2, 2)
```

[8]:
```
def F_sin(u, v):
    r = math.sqrt(u ** 2 + v ** 2)
    if r == 0: return (u, v, 3)
    else:
        return (u, v, 3 * math.sin(r) / r)

bornes_sin = [-10, 10, -10, 10]
```

1.1.3 1.3 Points réguliers, plan tangent

Lorsque la fonction F est de classe $\mathcal{C}^1$ sur $\mathcal{D}$ (ce que nous supposerons toujours), on peut calculer en tout point de $\mathcal{D}$ les vecteurs $\frac{\partial F}{\partial u}$ et $\frac{\partial F}{\partial v}$. Si ces deux vecteurs sont libres, le point $F(u,v)$ est un *point régulier* de la surface. Sinon, ce point est *singulier*.Le plan passant par $F(u,v)$ et dirigé par les dérivées partielles de F au point (u,v) est le *plan tangent* à la surface au point $F(u,v)$. Pour les fonctions F que nous considérerons, quasiment tous les points seront réguliers, mais pas tous. Par exemple, pour la sphère on a

$$\frac{\partial F}{\partial u} = \begin{pmatrix} -R\sin u\cos v \\ -R\sin u\sin v \\ R\cos u \end{pmatrix}$$

et

$$\frac{\partial F}{\partial v} = \begin{pmatrix} -R\cos u\sin v \\ R\cos u\cos v \\ 0 \end{pmatrix}$$

On en tire

$$\frac{\partial F}{\partial u} \wedge \frac{\partial F}{\partial v} = -R\cos u F(u,v)$$

La norme de ce vecteur est

$$||\frac{\partial F}{\partial u} \wedge \frac{\partial F}{\partial v}|| = R^2\cos u$$

et elle est nulle si et seulement si $u = \pm\frac{\pi}{2}$, c'est à dire lorsque le point $F(u,v)$ est le pôle nord ou le pôle sud. Remarquons que dans ce cas, $\frac{\partial F}{\partial v} = 0$.

Notre paramétrage de la sphère possède donc deux points singulers.

Nous ne nous en servirons pas dans la suite du notebook, mais la dérivée partielle de F par rapport à u peut être approchée par

$$\frac{\partial F}{\partial u}(u,v) \simeq \frac{F(u+h,v) - F(u,v)}{h}$$

où h est un réel non nul « petit ». Il faut juste s'assurer que le point $(u+h, v)$ est dans le domaine de définition de F. Si ce n'est pas le cas, mais si $(u-h, v) \in \mathcal{D}$, on peut utiliser

$$\frac{\partial F}{\partial u}(u, v) \simeq \frac{F(u, v) - F(u-h, v)}{h}$$

Si $\mathcal{D}$ est un rectangle, on est toujours dans l'un de ces deux cas, pourvu que h soit petit devant les dimensions du rectangle.

Les fonctions `diffu` et `diffv` calculent une valeur approchée des dérivées partielles de F au point (u, v).

[9]:
```
def diffu(F, u, v, bornes, h=1e-5):
    if u + h <= bornes[1]:
        return mul(1 / h, sub(F(u + h, v), F(u, v)))
    else:
        return mul(1 / h, sub(F(u, v), F(u - h, v)))
```

[10]:
```
def diffv(F, u, v, bornes, h=1e-5):
    if v + h <= bornes[3]:
        return mul(1 / h, sub(F(u, v + h), F(u, v)))
    else:
        return mul(1 / h, sub(F(u, v), F(u, v - h)))
```

Voici des vecteurs tangents à la sphère au pôle nord, obtenu (entre autres) pour le paramètre $(\frac{\pi}{2}, 0)$.

[11]:
```
print(F_sphere(math.pi / 2, 0))
print(diffu(F_sphere, math.pi / 2, 0, bornes_sphere))
print(diffv(F_sphere, math.pi / 2, 0, bornes_sphere))
```

```
(6.123233995736766e-17, 0.0, 1.0)
(-0.9999999999898843, 0.0, 5.000000413701854e-06)
(-3.0616184769693227e-22, 6.123233995634712e-17, 0.0)
```

On retrouve bien que la seconde dérivée partielle est nulle. La première dérivée partielle est quant à elle un vecteur horizontal.

1.1.4 1.4 Discrétisation

Il est évidemment impossible de dessiner *tous* les points d'une surface. Avant toute chose, il faut opérer une *discrétisation*. La fonction `discretiser` prend en paramètres

- Une fonction `F` de deux variables.
- Une liste $[a, b, c, d]$ de flottants qui représente l'ensemble de définition de F, $\mathcal{D} = [a, b] \times [c, d]$.
- Deux entiers n_u et n_v.

Pour $0 \leq i \leq n_u$ et $0 \leq j \leq n_v$, soient

$$u_i = a + i\frac{b-a}{n_u}$$

et

$$v_j = c + j\frac{d-c}{n_v}$$

La fonction `discretiser` renvoie une matrice m de taille $(n_u+1)\times(n_v+1)$ telle que $m_{ij} = F(u_i, v_j)$.

[12]:
```
def subdi(a, b, n):
    d = (b - a) / n
    return [a + k * d for k in range(n + 1)]
```

[13]:
```
subdi(0, 1, 5)
```

[13]:
```
[0.0, 0.2, 0.4, 0.6000000000000001, 0.8, 1.0]
```

[14]:
```
def discretiser(F, bornes, nu, nv):
    umin, umax, vmin, vmax = bornes
    us = subdi(umin, umax, nu)
    vs = subdi(vmin, vmax, nv)
    m = [[F(u,v) for v in vs] for u in us]
    return m
```

[15]:
```
discretiser(F_sphere, bornes_sphere, 4, 4)
```

[15]:
```
[[(6.123233995736766e-17, 0.0, -1.0),
  (3.749399456654644e-33, 6.123233995736766e-17, -1.0),
  (-6.123233995736766e-17, 7.498798913309288e-33, -1.0),
  (-1.1248198369963932e-32, -6.123233995736766e-17, -1.0),
  (6.123233995736766e-17, -1.4997597826618576e-32, -1.0)],
 [(0.7071067811865476, 0.0, -0.7071067811865475),
  (4.329780281177467e-17, 0.7071067811865476, -0.7071067811865475),
  (-0.7071067811865476, 8.659560562354934e-17, -0.7071067811865475),
  (-1.29893408435324e-16, -0.7071067811865476, -0.7071067811865475),
  (0.7071067811865476, -1.7319121124709868e-16, -0.7071067811865475)],
 [(1.0, 0.0, 0.0),
  (6.123233995736766e-17, 1.0, 0.0),
  (-1.0, 1.2246467991473532e-16, 0.0),
  (-1.8369701987210297e-16, -1.0, 0.0),
  (1.0, -2.4492935982947064e-16, 0.0)],
 [(0.7071067811865476, 0.0, 0.7071067811865475),
  (4.329780281177467e-17, 0.7071067811865476, 0.7071067811865475),
  (-0.7071067811865476, 8.659560562354934e-17, 0.7071067811865475),
  (-1.29893408435324e-16, -0.7071067811865476, 0.7071067811865475),
  (0.7071067811865476, -1.7319121124709868e-16, 0.7071067811865475)],
 [(6.123233995736766e-17, 0.0, 1.0),
  (3.749399456654644e-33, 6.123233995736766e-17, 1.0),
  (-6.123233995736766e-17, 7.498798913309288e-33, 1.0),
  (-1.1248198369963932e-32, -6.123233995736766e-17, 1.0),
```

```
(6.123233995736766e-17, -1.4997597826618576e-32, 1.0)]]
```

1.1.5 1.5 Premiers tests

Histoire de contrôler que tout se passe bien, faisons un test rapide. La fonction `F_sin` que nous avons définie plus haut est de la forme

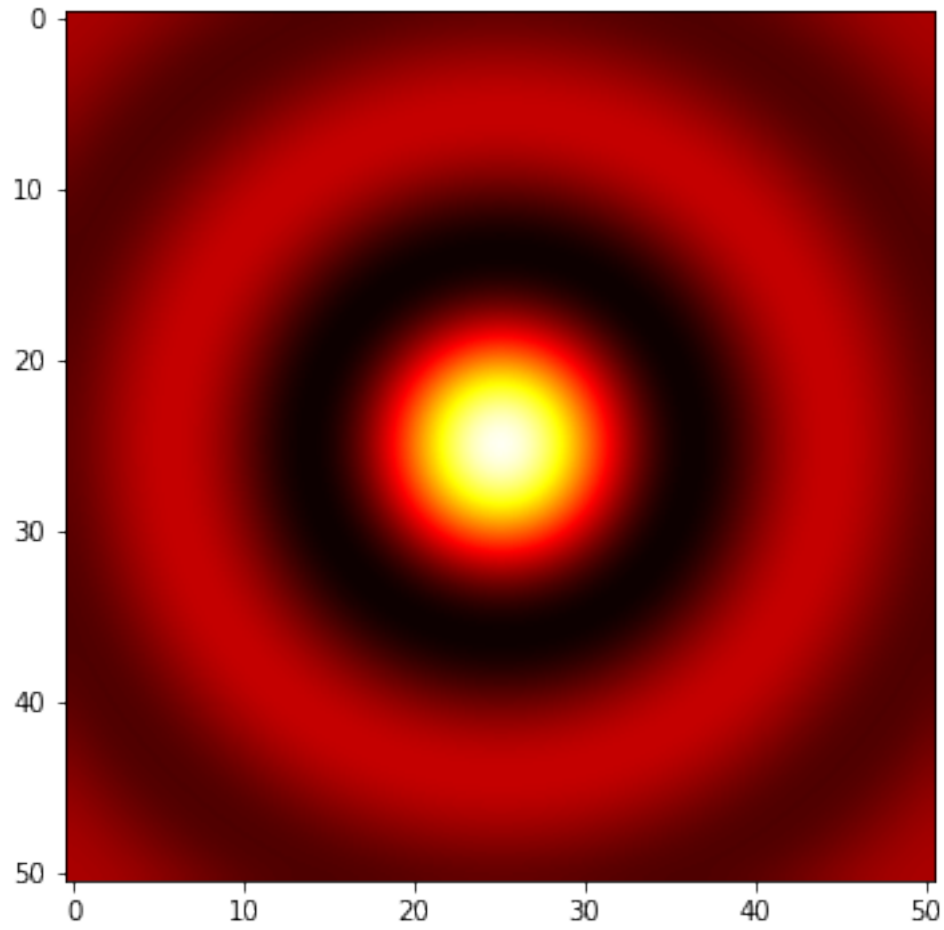
$$F(u,v) = (u, v, \frac{\sin r}{r}) \text{ où } r = \sqrt{u^2 + v^2}$$

u et v sont donc l'abscisse et l'ordonnée des points de la surface. Un tracé rapide de la troisème coordonnée des éléments de la matrice obtenue avec `discretiser` avec `imshow` nous donnera une idée de la surface. Nous devrions obtenir un dessin avec des cercles concentriques de différentes couleurs.

[16]:
```
def matrice(p, q):
    m = p * [None]
    for i in range(p): m[i] = q * [0]
    return m
```

[17]:
```
def plot_dens(F, bornes, nu, nv):
    m = discretiser(F, bornes, nu, nv)
    m1 = matrice(nu + 1, nv + 1)
    for i in range(nu + 1):
        for j in range(nv + 1):
            m1[i][j] = m[i][j][2]
    plt.imshow(m1, cmap='hot', interpolation='bicubic')
```

[18]:
```
plot_dens(F_sin, bornes_sin, 50, 50)
```



Tout va bien, nous pouvons continuer.

1.2 2. Tracé en fil de fer

1.2.1 2.1 Un peu d'algèbre

Le module `vect_utils` contient quelques fonctions permettant de manipuler facilement des quantités vectorielles. Ces fonctions ne présentent pas de difficultés. Signalons simplement que nous représentons les points et les vecteurs de l'espace par des triplets de flottants.

La fonction `vecteur` prend deux points A et B en paramètres. El renvoie le vecteur $\overrightarrow{AB}$.

[19]:
```
vecteur((1, 2, 3), (2, 1, 4))
```

[19]:
```
(1, -1, 1)
```

Les fonctions `add` et `sub` prennent deux vecteurs en paramètres et renvoient leur somme. Nous avons d'ailleurs déjà utilisé `sub` dans le code des fonctions `diffu` et `diffv`.

[20]:
```
add((1, 2, 1), (3, -2, -2))
```

[20]:
```
(4, 0, -1)
```

[21]:
```
sub((1, 2, 1), (3, -2, -2))
```

[21]:
```
(-2, 4, 3)
```

La fonction `mul` prend un flottant t et un vecteur u en paramètres et renvoie le vecteur tu.

[22]:
```
mul(4, (3, -2, -2))
```

[22]:
```
(12, -8, -8)
```

La fonction `prod_scal` renvoie le produit scalaire des vecteurs u et v.

[23]:
```
prod_scal((1, 2, 3), (2, 1, 4))
```

[23]:
```
16
```

La fonction `norme2` prend un vecteur u en paramètre et renvoie $||u||^2$. La fonction `norme` renvoie quant à elle la norme de u.

[24]:
```
norme((1, 2, 3))
```

[24]:
```
3.7416573867739413
```

La fonction `normaliser` prend un vecteur u non nul en paramètre et renvoie le vecteur unitaire $u\,/\,||u||$.

[25]:
```
normaliser((1, 2, 3))
```

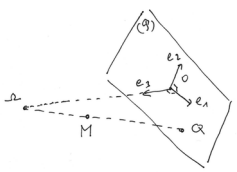
[25]:
```
(0.2672612419124244, 0.5345224838248488, 0.8017837257372732)
```

La fonction `prod_vect` prend deux vecteurs u et v en paramètres et renvoie leur produit vectoriel $u \wedge v$.

[26]:
```
prod_vect((1, 2, 3), (0, 0, 1))
```

[26]:
```
(2, -1, 0)
```

1.2.2 2.2 Bases adaptées



L'écran de mon ordinateur étant de dimension 2, je ne peux pas y afficher de façon immédiate des objets de l'espace. Pour arriver à mes fins je dois les *projeter*. Il existe de nombreux types de projections. Nous allons en choisir un qui est relativement naturel et qui donne d'excellents résultats.

Soit Ω un point de l'espace, qui représente un *observateur*. Cet observateur regarde une scène composée de points. Pour projeter un point M de la scène, l'observateur trace la droite (ΩM). Cette droite coupe un *certain* plan $\mathcal{P}$ (censé être l'écran de l'ordinateur) en un point Q, qui est le projeté de M.

La question qui se pose immédiatement est : quel plan $\mathcal{P}$ choisir ? Nous allons prendre le plan passant par O et orthogonal au vecteur $\overrightarrow{O\Omega}$. Notons tout de suite e_3 le vecteur unitaire colinéaire à $\overrightarrow{O\Omega}$ et de même sens. Nous allons maintenant munir le plan $\mathcal{P}$ d'une base orthonormée (e_1, e_2) bien adaptée à l'observateur.

- On choisit pour e_1 un vecteur horizontal unitaire orthogonal à e_3, l'idée étant que l'observateur ne regarde pas la scène avec la tête de travers. Le vecteur e_1 peut être obtenu en normalisant le vecteur $k \wedge e_3$, où $k = (0, 0, 1)$. Ceci ne fonctionne évidemment que si les vecteurs e_3 et k ne sont pas colinéaires. Si e_3 est colinéaire à k, on choisit arbitrairement $e_1 = (0, 1, 0)$.
- On pose ensuite $e_2 = e_3 \wedge e_1$.

La famille $\mathcal{B} = (e_1, e_2, e_3)$ est une base orthonormée directe de l'espace. Nous l'appellerons la *base adaptée* à l'observateur.

C'est ici que l'on dit merci au module `vect_utils`. La fonction `base_adaptee` prend en paramètre un point représentant l'observateur. Elle renvoie la base $\mathcal{B}$ adaptée à l'observateur. Bien évidemment, l'observateur ne doit pas être placé en $O = (0, 0, 0)$.

[27]:
```
def base_adaptee(Obs):
    e3 = normaliser(vecteur((0, 0, 0), Obs))
    u = prod_vect((0, 0, 1), e3)
    if u == (0, 0, 0): e1 = (0, 1, 0)
    else: e1 = normaliser(u)
    e2 = prod_vect(e3, e1)
    return [e1, e2, e3]
```

[28]:
```
base_adaptee((1, 2, 1))
```

[28]:
```
[(-0.8944271909999159, 0.4472135954999579, 0.0),
 (-0.18257418583505539, -0.36514837167011077, 0.9128709291752769),
 (0.4082482904638631, 0.8164965809277261, 0.4082482904638631)]
```

[29]:
```
base_adaptee((0, 0, 5))
```

[29]:
```
[(0, 1, 0), (-1.0, 0.0, 0.0), (0.0, 0.0, 1.0)]
```

1.2.3 2.3 Projections

Ô lecteur, je sens grandir ton impatience à tracer des surfaces … nous y sommes presque. Il reste à calculer les coordonnées des projetés dans une base adaptée. Soient donc Ω un observateur, M un point de l'espace, et Q son projeté sur le plan passant par O et dirigé par (e_1, e_2). Il existe un réel t tel que

$$\overrightarrow{\Omega Q} = t\overrightarrow{\Omega M}$$

Par ailleurs, comme $\overrightarrow{OQ}$ est dans le plan de projection,

$$< \overrightarrow{OQ}, e_3 >= 0$$

Écrivons

$$\overrightarrow{OQ} = \overrightarrow{O\Omega} + \overrightarrow{\Omega Q} = \overrightarrow{O\Omega} + t\overrightarrow{\Omega M}$$

Effectuons le produit scalaire par e_3. Il vient alors

$$0 =< \overrightarrow{OQ}, e_3 >=< \overrightarrow{O\Omega}, e_3 > +t < \overrightarrow{\Omega M}, e_3 >$$

On en déduit la valeur de t, puis le vecteur $\overrightarrow{OQ}$. Les coordonnées de ce point dans la base adaptée s'obtiennent ensuite par des produits scalaires de ce vecteur avec e_1 et e_2.

La fonction `projeter_point` fait ce travail. Elle prend en paramètres une point M, un observateur `Obs` et une base adaptée. Elle renvoie le couple (x, y) des coordonnée du projeté du point M dans le plan de projection.

[30]:
```
def projeter_point(M, Obs, base):
    OObs = vecteur((0, 0, 0), Obs)
    ObsM = vecteur(Obs, M)
    t = - prod_scal(OObs, base[2]) / prod_scal(ObsM, base[2])
    OQ = add(OObs, mul(t, ObsM))
    x = prod_scal(OQ, base[0])
    y = prod_scal(OQ, base[1])
    return (x, y)
```

La fonction `projeter_surface` prend en paramètres

- Une matrice m (obtenue par exemple par la fonction `discretiser`) contenant les points de la surface.
- Un observateur `Obs`.

Elle renvoie la matrice des projections des points contenus dans m.

[31]:
```
def projeter_surface(m, Obs):
    nu = len(m) - 1
    nv = len(m[0]) - 1
```

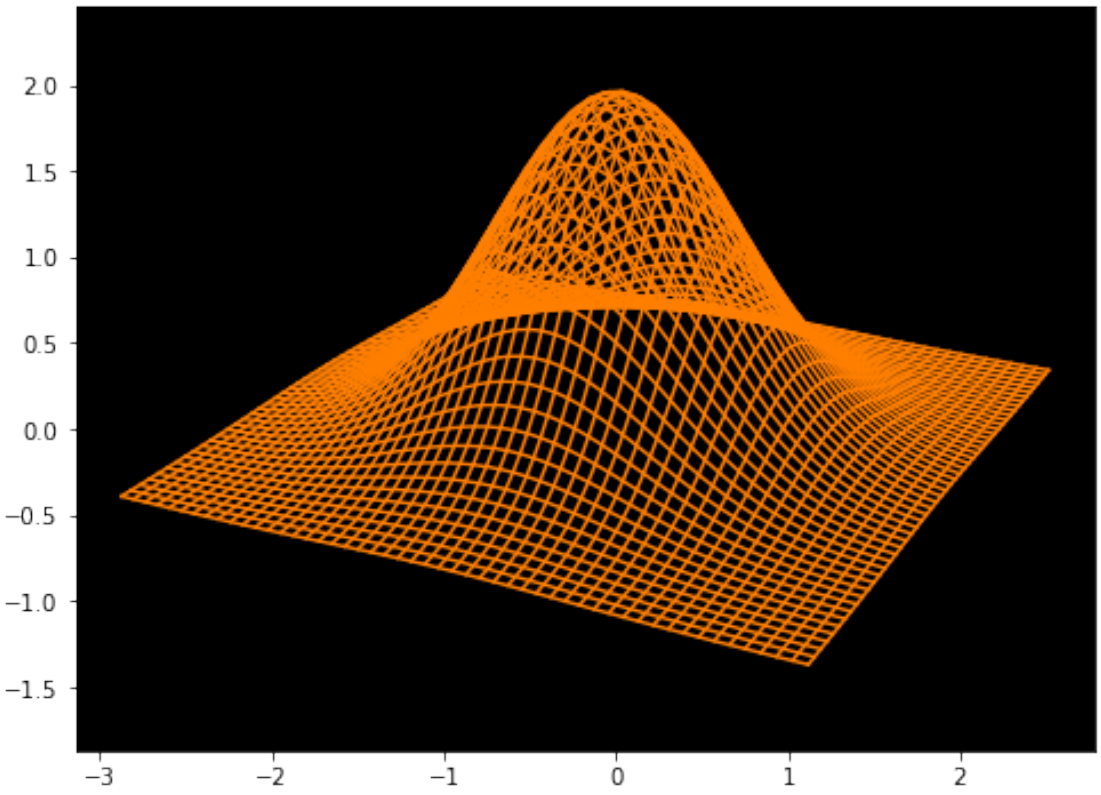
```
    m1 = matrice(nu + 1, nv + 1)
    base = base_adaptee(Obs)
    for i in range(nu + 1):
        for j in range(nv + 1):
            m1[i][j] = projeter_point(m[i][j], Obs, base)
    return m1
```

1.2.4 2.4 Tracé « fil de fer »

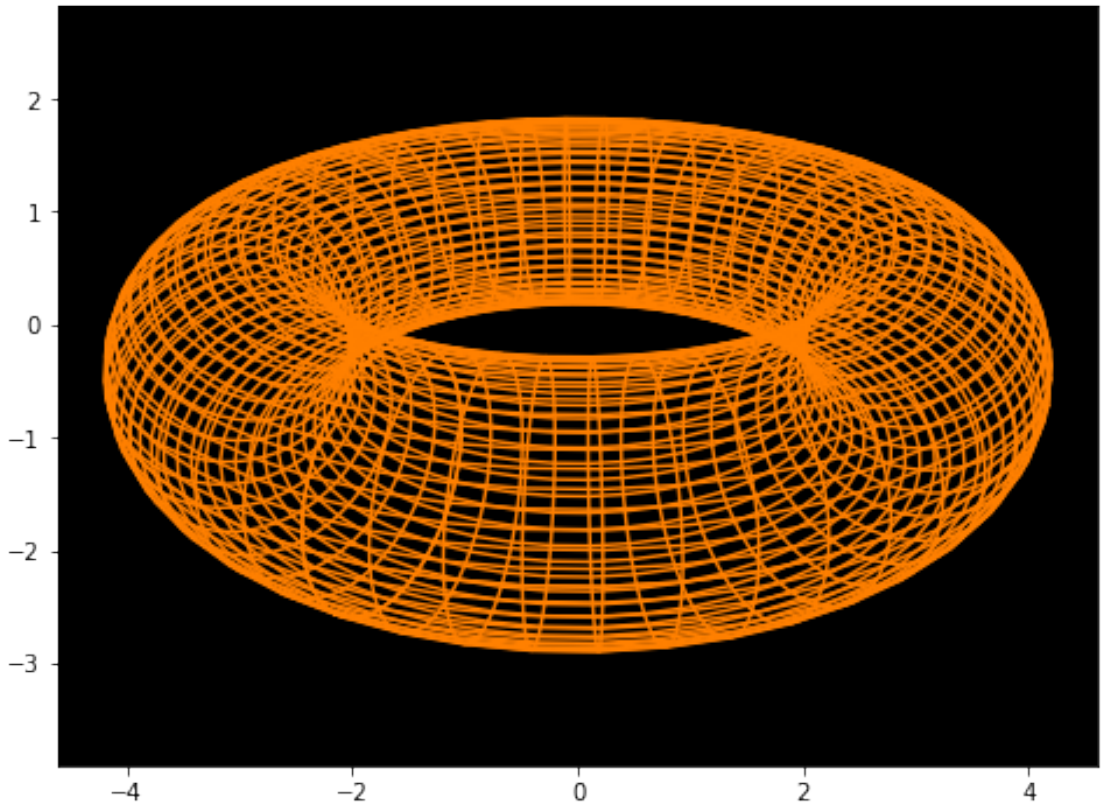
Enfin, nous y voilà. La fonction de tracé de surface effectue un tracé « en fil de fer ». En appelant u et v les paramètres, cette fonction trace les lignes à u constant et les lignes à v constant.

[32]:
```
def tracer1(F, bornes, nu, nv, Obs):
    fig, ax = plt.subplots()
    ax.set_facecolor('black')
    c = (1., 0.5, 0.)
    m = discretiser(F, bornes, nu, nv)
    m1 = projeter_surface(m, Obs)
    for i in range(nu + 1):
        xs = [m1[i][j][0] for j in range(nv + 1)]
        ys = [m1[i][j][1] for j in range(nv + 1)]
        plt.plot(xs, ys, color=c)
    for j in range(nv + 1):
        xs = [m1[i][j][0] for i in range(nu + 1)]
        ys = [m1[i][j][1] for i in range(nu + 1)]
        plt.plot(xs, ys, color=c)
        plt.axis('equal')
```

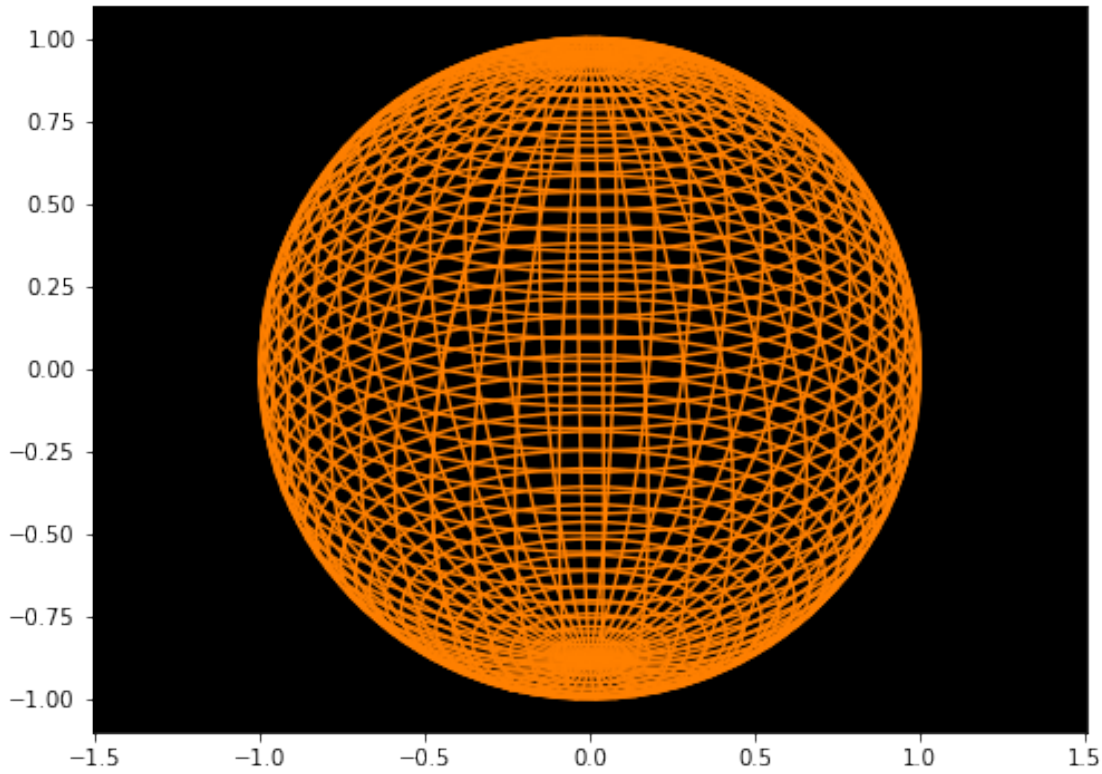
[33]:
```
tracer1(F_exp, bornes_exp, 50, 50, (10, 5, 5))
```



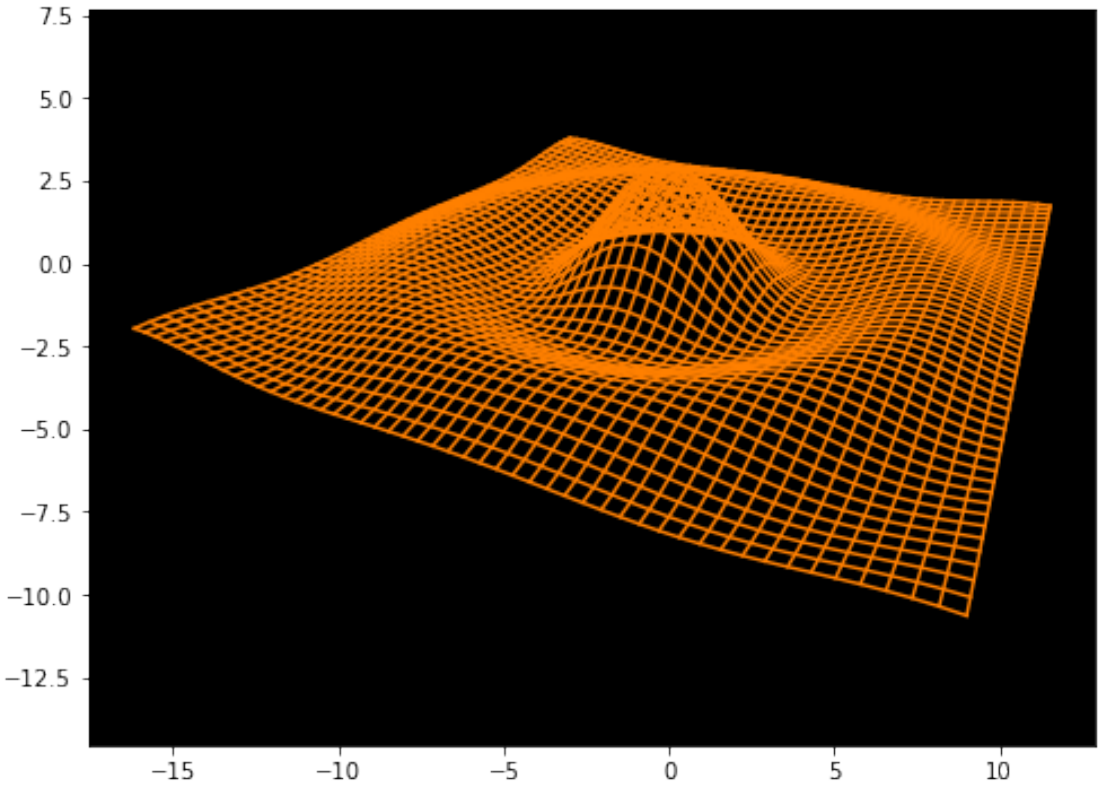
[34]:
```
tracer1(F_tore, bornes_tore, 50, 50, (10, 5, 5))
```



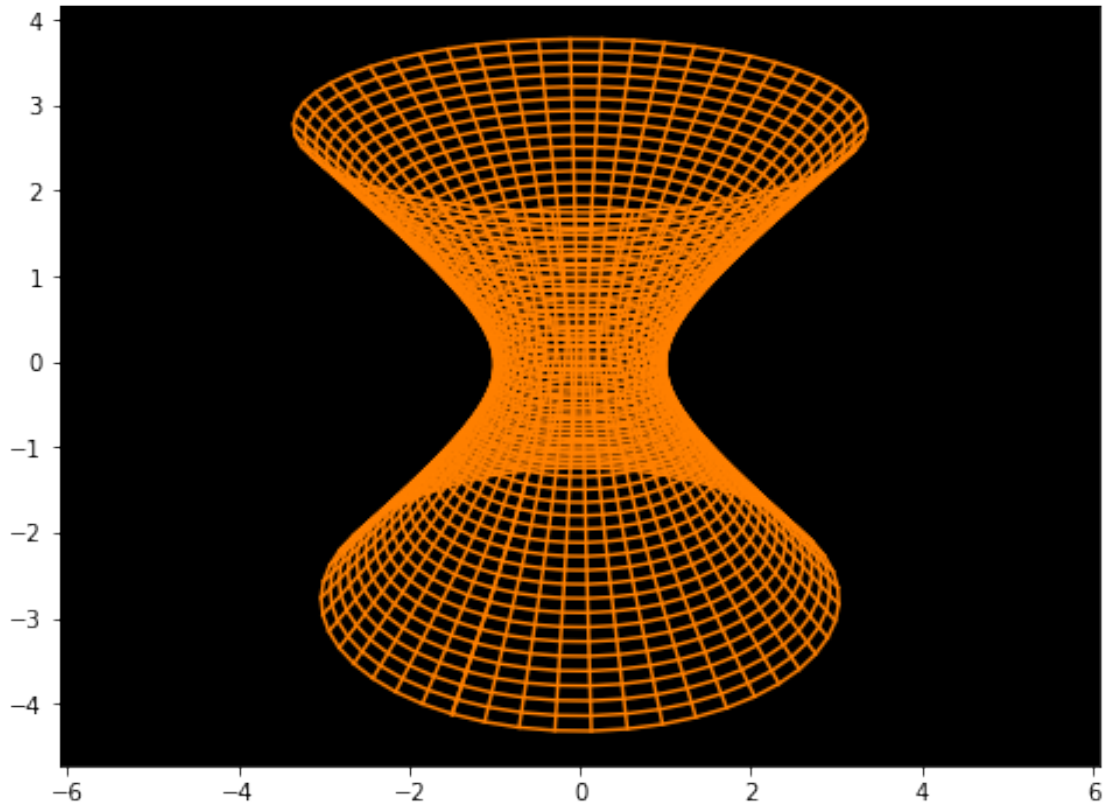
[35]:
```
tracer1(F_sphere, bornes_sphere, 50, 50, (10, 5, 5))
```



[36]:
```
tracer1(F_sin, bornes_sin, 50, 50, (20, 10, 10))
```



[37]:
```
tracer1(F_hyper, bornes_hyper, 50, 50, (20, 10, 10))
```



Et voilà. Évidemment, les résultats sont un peu décevants. On obtient une vue de la surface qui nous donne une bonne idée de sa « forme », mais on sent que l'on doit pouvoir faire mieux. C'est effectivement ce que nous allons faire, en nous attaquant au problème de *l'élimination des parties cachées*.

1.3 3. Élimination des parties cachées

1.3.1 3.1 Le principe

Une fois la surface discrétisée, il est facile d'obtenir à partir de la discrétisation un découpage de la surface en triangles. Nous appellerons ces triangles les *faces* de la surface. Pourquoi des triangles ? On pourrait aussi bien la découper en quadrilatères, mais l'avantage des triangles est que trois points de l'espace sont toujours dans un même plan, alors que ce n'est pas le cas pour quatre points. Ainsi, en choisissant des triangles, notre surface possède des faces planes.

Une fois les faces calculées, il suffit de les projeter et de les afficher. Toute l'astuce consiste à afficher d'abord les faces les plus éloignées de l'observateur. Ainsi, l'affichage de faces « proches » efface les faces qui sont derrière, et qui sont donc automatiquement « cachées ». Cette technique s'appelle *l'algorithme du peintre*.

1.3.2 3.2 Le calcul des faces

La fonction `faces` renvoie la liste des faces de la surface définie par la fonction F. Son code suffit à expliquer son fonctionnement. Signalons simplement que nous représentons une face en Python par une liste de trois points. La fonction `faces` renvoie donc une liste de listes de trois triplets de flottants.

[38]:
```
def faces(F, bornes, nu, nv):
    m = discretiser(F, bornes, nu, nv)
    s1 = [[m[i][j], m[i+1][j], m[i+1][j+1]] for i in range(nu) for j in range(nv)]
    s2 = [[m[i][j], m[i+1][j+1], m[i][j+1]] for i in range(nu) for j in range(nv)]
    return s1 + s2
```

Exemple ...

[39]:
```
faces(F_tore, bornes_tore, 2, 2)
```

[39]:
```
[[(4.0, 0.0, 0.0),
  (2.0, 0.0, 1.2246467991473532e-16),
  (-2.0, 2.4492935982947064e-16, 1.2246467991473532e-16)],
 [(-4.0, 4.898587196589413e-16, 0.0),
  (-2.0, 2.4492935982947064e-16, 1.2246467991473532e-16),
  (2.0, -4.898587196589413e-16, 1.2246467991473532e-16)],
 [(2.0, 0.0, 1.2246467991473532e-16),
  (4.0, 0.0, -2.4492935982947064e-16),
  (-4.0, 4.898587196589413e-16, -2.4492935982947064e-16)],
 [(-2.0, 2.4492935982947064e-16, 1.2246467991473532e-16),
  (-4.0, 4.898587196589413e-16, -2.4492935982947064e-16),
  (4.0, -9.797174393178826e-16, -2.4492935982947064e-16)],
 [(4.0, 0.0, 0.0),
  (-2.0, 2.4492935982947064e-16, 1.2246467991473532e-16),
  (-4.0, 4.898587196589413e-16, 0.0)],
 [(-4.0, 4.898587196589413e-16, 0.0),
  (2.0, -4.898587196589413e-16, 1.2246467991473532e-16),
  (4.0, -9.797174393178826e-16, 0.0)],
 [(2.0, 0.0, 1.2246467991473532e-16),
  (-4.0, 4.898587196589413e-16, -2.4492935982947064e-16),
  (-2.0, 2.4492935982947064e-16, 1.2246467991473532e-16)],
 [(-2.0, 2.4492935982947064e-16, 1.2246467991473532e-16),
  (4.0, -9.797174393178826e-16, -2.4492935982947064e-16),
  (2.0, -4.898587196589413e-16, 1.2246467991473532e-16)]]
```

1.3.3 3.3 Quelques fonctions utiles

La fonction `projeter_face` renvoie la liste des projections des points de la face. Elle nécessite en paramètres un observateur et une base adaptée.

[40]:
```
def projeter_face(f, Obs, base):
    return [projeter_point(P, Obs, base) for P in face]
```

La fonction `distance2` prend en paramètres deux points A et B et renvoie $d(A,B)^2$.

[41]:
```
def distance2(P, Q):
    return norme2(vecteur(P, Q))
```

La fonction `centre_face` renvoie le barycentre des sommets de la face.

[42]:
```
def centre_face(f):
    x = (f[0][0] + f[1][0] + f[2][0]) / 3
    y = (f[0][1] + f[1][1] + f[2][1]) / 3
    z = (f[0][2] + f[1][2] + f[2][2]) / 3
    return (x, y, z)
```

La fonction `distance_face` prend en paramètres une face et un observateur. Elle renvoie la distance (au carré) de l'observateur au centre de la face.

[43]:
```
def distance_face(f, Obs):
    P = centre_face(f)
    return distance2(P, Obs)
```

1.3.4 3.4 De quelle couleur tracer les faces ?

Voilà une bonne question. Quitte à faire bien les choses, autant choisir des couleurs réalistes. Imaginons notre scène éclairée par une source de lumière. Cette source de lumière est supposée ponctuelle et infiniment lointaine. Elle est donc caractérisée par une direction dans l'espace, bref par un vecteur non nul. Nous appelons ce vecteur `light` et nous en faisons une variable globale.

[44]:
```
light = normaliser((1, 1, 1))
```

La fonction `normale` prend une face en paramètre et renvoie un vecteur unitaire orthogonal à cette face. Si la face est « dégénérée » (3 sommets alignés), la fonction renvoie le vecteur nul.

Remarquons que la face a pour sommets $F(u,v)$, $F(u+h,v)$ et $F(u,v+h)$ où (u,v) est un point de l'ensemble de définition de F et h est un réel non nul et « petit ». La fonction `normale` calcule les vecteurs

$$w_1 = F(u+h,v) - F(u,v) \simeq h\frac{\partial F}{\partial u}(u,v)$$

et

$$w_2 = F(u,v+h) - F(u,v) \simeq h\frac{\partial F}{\partial v}(u,v)$$

Ces deux vecteurs sont ainsi approximativement colinéaires aux dérivées partielles de F au point (u, v) et donc approximativement tangents à la surface. Ainsi, leur produit vectoriel est une approximation d'un vecteur normal à la surface.

[45]:
```
def normale(f):
    w1 = vecteur(f[0], f[1])
    w2 = vecteur(f[0], f[2])
    w = prod_vect(w1, w2)
    r = math.sqrt(norme2(w))
    if r == 0: return (0, 0, 0)
    else: return mul(1 / r, w)
```

Selon l'inclinaison de ce vecteur normal par rapport à la direction de la lumière, la face est plus ou moins éclairée. Un angle nul produit un éclairement maximal, un angle de $\frac{\pi}{2}$ produit un éclairement nul. Ce modèle simpliste d'éclairement nous suffira en première approximation. Nous décidons de choisir comme intensité de l'éclairement de la face le réel

$$I = 0.3 + 0.7|\cos\theta|$$

où θ est l'angle entre la direction de la lumière et le vecteur normal à la face.

Remarquons que $I \geq 0.3$. Ainsi, une face ne sera jamais complètement « noire ». C'est ce que l'on appelle la *réflexion ambiante*. La partie de l'intensité faisant intervenir θ est la *réflexion diffuse*.

Remarque technique. La fonction `tracer_face` utilise des objets de type `Polygon`. Ces objets sont définis dans le module `matplotlib.patches`.

[51]:
```
def tracer_face(f, Obs, base, ax, edge):
    f1 = [projeter_point(P, Obs, base) for P in f]
    n = normale(f)
    I = 0.3 + 0.7 * abs(prod_scal(n, light))
    if I >= 1: I = 1
    fc = (I, 0.7 * I, 0)
    if edge: ec = (0, 0, 0)
    else: ec = fc
    p = Polygon(f1, facecolor=fc, edgecolor=ec)
    ax.add_patch(p)
```

1.3.5 3.5 La fonction de tracé

Voici enfin la fonction de tracé. La fonction

- Calcule les faces de la surface.
- Trie ces faces par distances décroissantes par rapport à l'observateur.
- Trace chacune des faces.

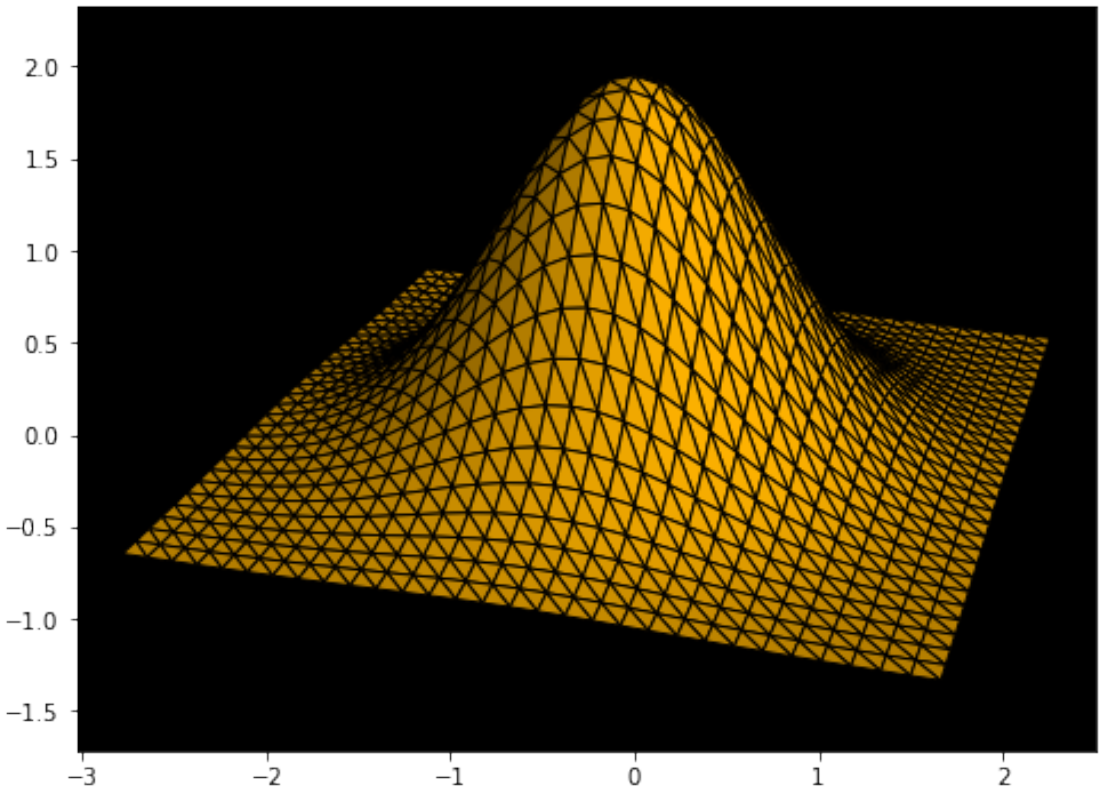
[47]:
```
def tracer2(F, bornes, nu, nv, Obs, edge=True):
    fig, ax = plt.subplots()
```

```
    ax.set_facecolor('black')
    fs = faces(F, bornes, nu, nv)
    fs.sort(key=lambda f: -distance2(centre_face(f), Obs))
    base = base_adaptee(Obs)
    for f in fs:
        tracer_face(f, Obs, base, ax, edge)
    plt.axis('equal')
```

Nous pouvons maintenant tracer nos surfaces préférées. Pour que le code soit plus lisible, j'ai écrit quelque chose de peu efficace. Par exemple, les faces d'une surface ont des sommets en commun. Le fait de projeter les faces une par une entraîne une répétition des calculs qui pourrait être évitée. En plus, nos fonctions de tracé reposent sur `matplotlib`, qui n'est pas une bibliothèque écrite pour tracer vite.

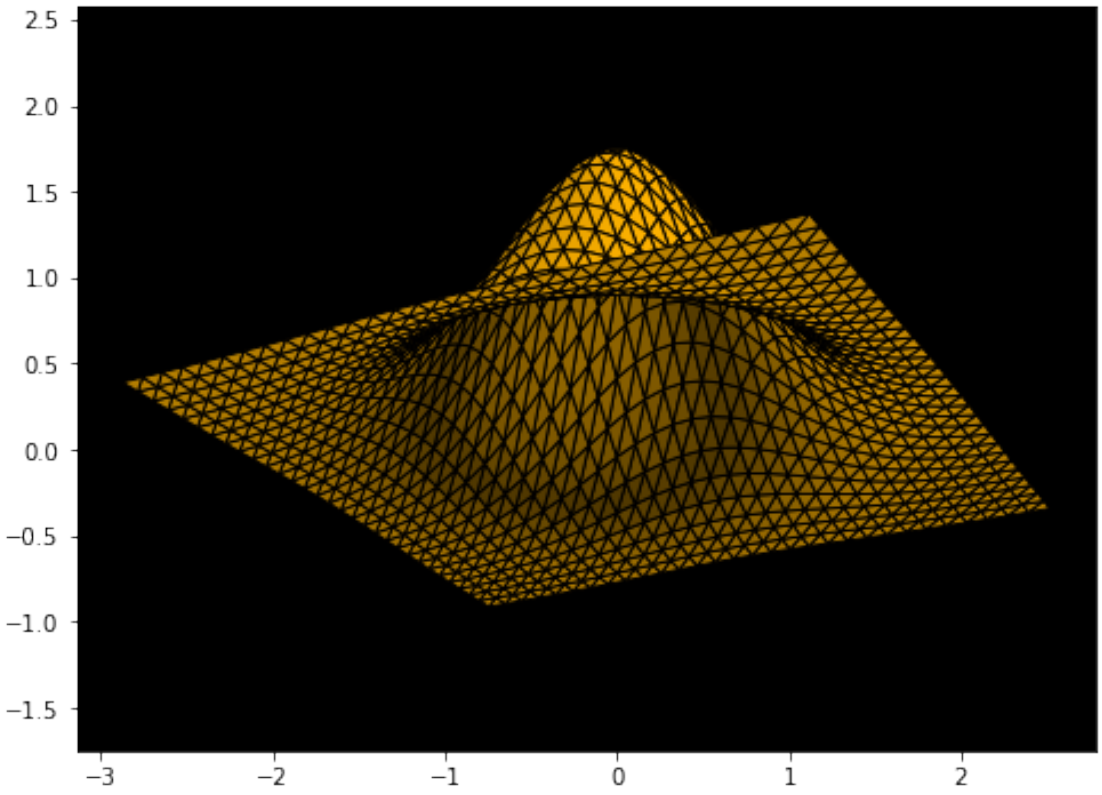
Soyez donc (un tout petit peu) patients en évaluant les cellules ci-dessous.

[52]:
```
tracer2(F_exp, bornes_exp, 30, 30, (10, 3, 5), edge=True)
```



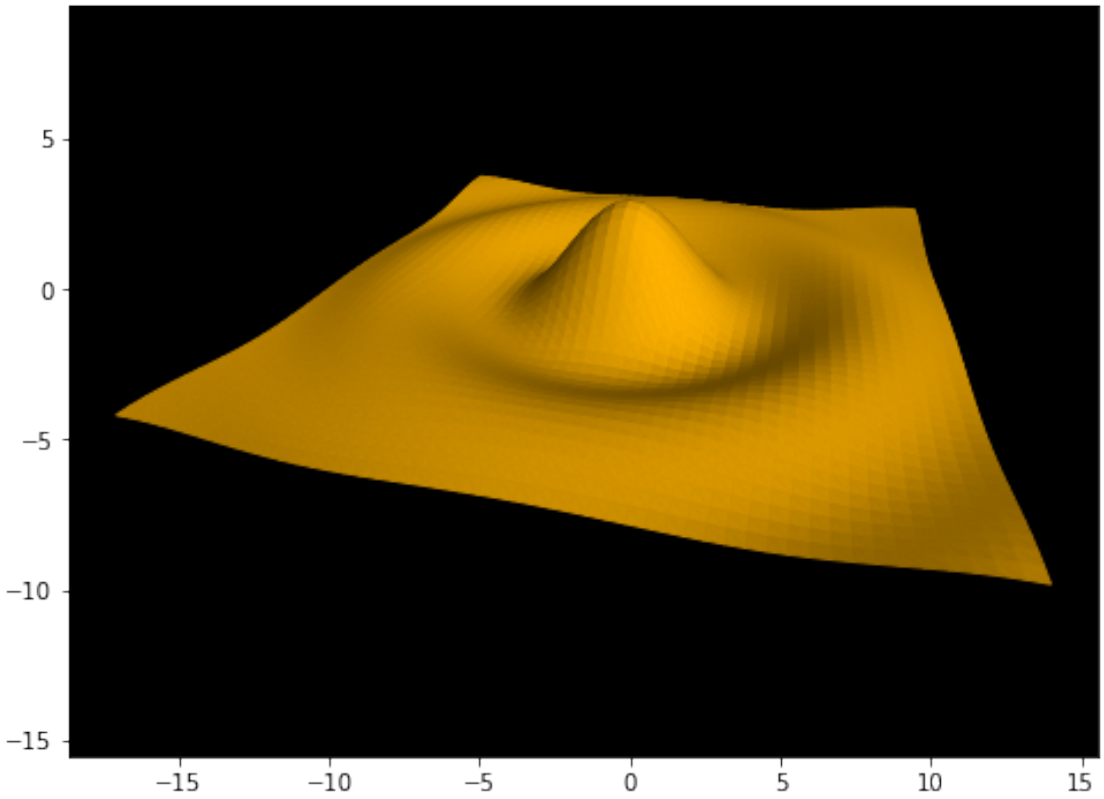
La même surface, vue par-dessous.

[53]:
```
tracer2(F_exp, bornes_exp, 30, 30, (10, 5, -5), edge=True)
```

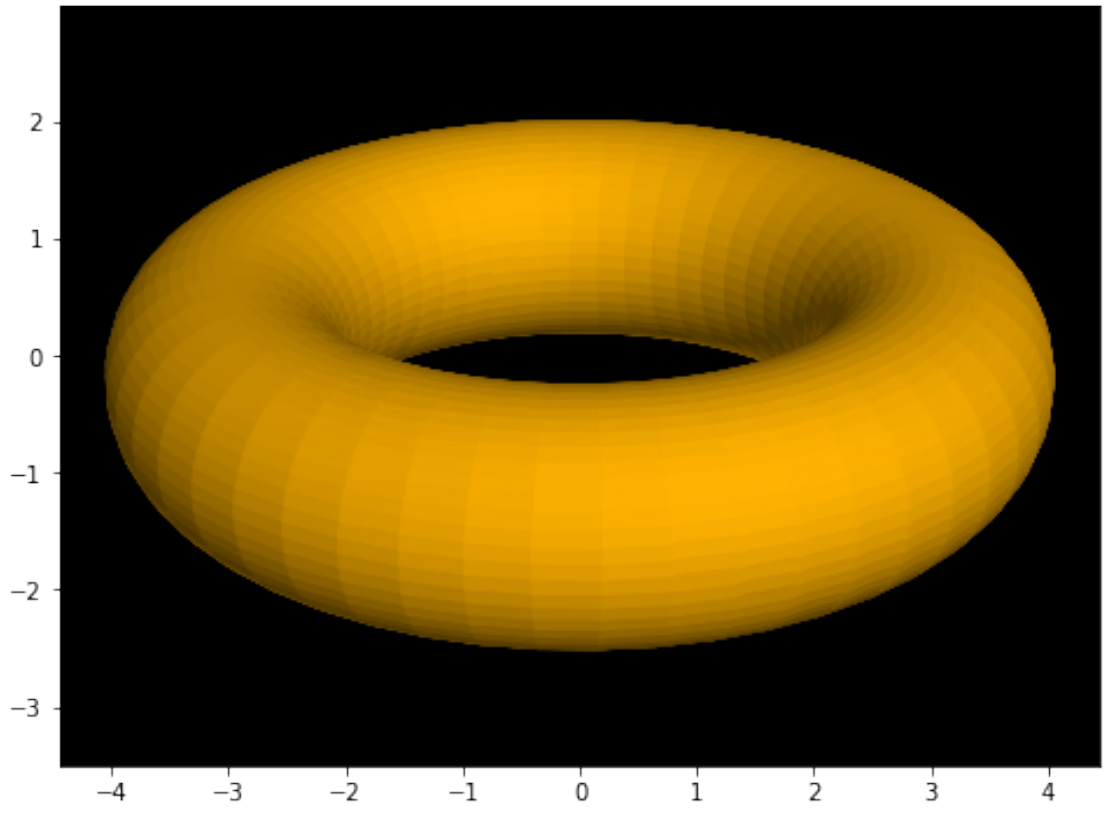


Voici le tracé de nos autres exemples, sans les bordures des faces.

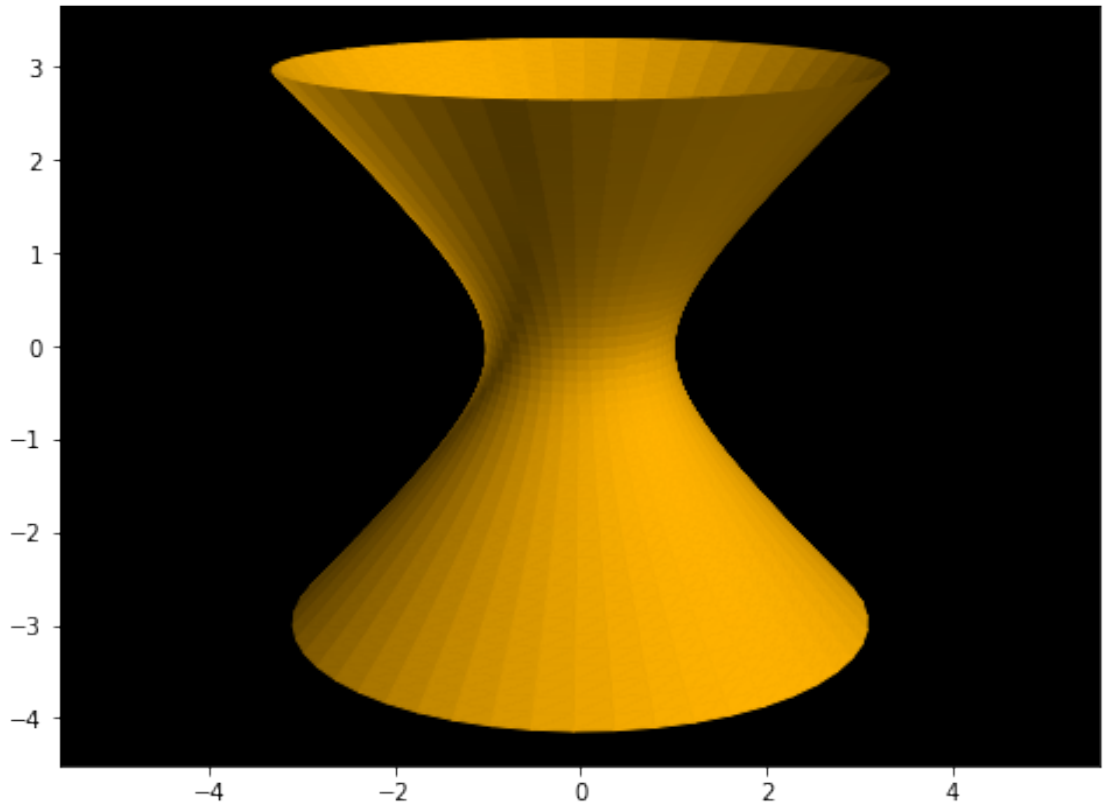
[55]:
```
tracer2(F_sin, bornes_sin, 50, 50, (20, 5, 10), edge=False)
```



[56]:
```
tracer2(F_tore, bornes_tore, 50, 50, (20, 10, 10), edge=False)
```



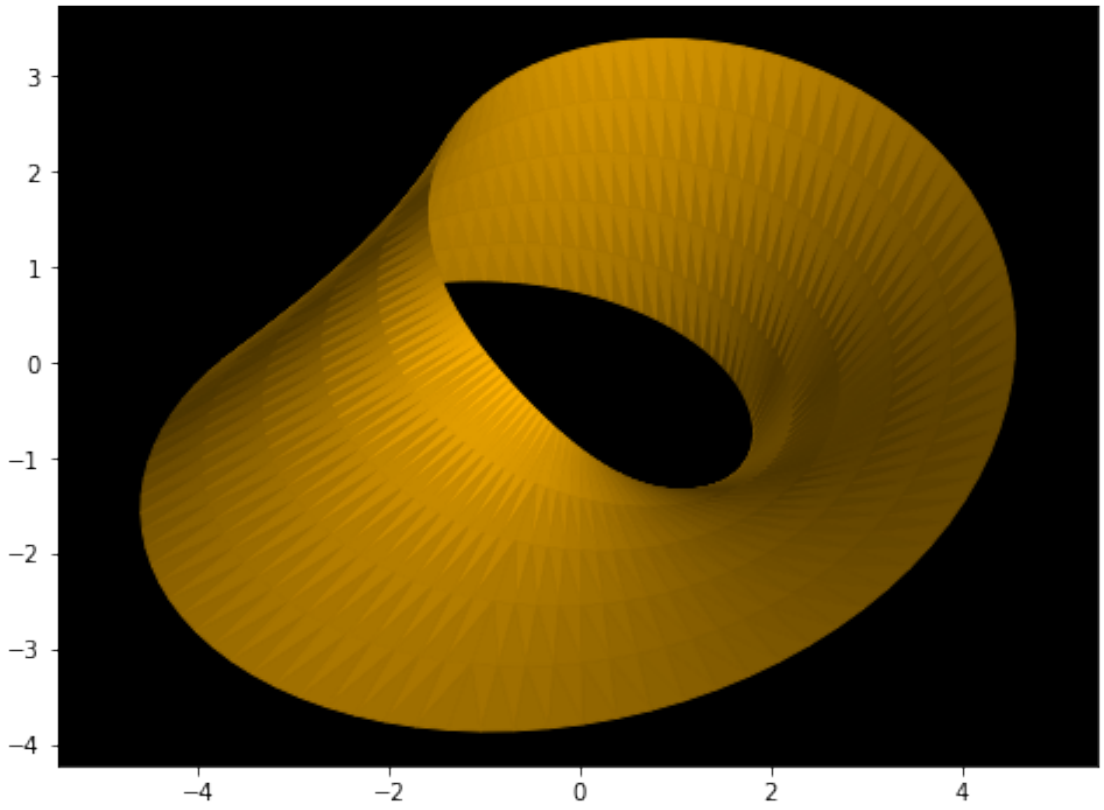
[57]:
```
tracer2(F_hyper, bornes_hyper, 50, 50, (20, 3, 5), edge=False)
```



Vous pouvez maintenant tracer tout ce qui vous plaira. Par exemple, un ruban de Möbius.

[58]:
```
def F_mobius(s, t, R=3):
    x = (R + s * math.cos(t / 2)) * math.cos(t)
    y = (R + s * math.cos(t / 2)) * math.sin(t)
    z = s * math.sin(t / 2)
    return (x, y, z)

bornes_mobius = [-2, 2, 0, 2 * math.pi]
```

[59]:
```
tracer2(F_mobius, bornes_mobius, 5, 100, (20, 5, 20), edge=False)
```



1.4 4. Conclusion

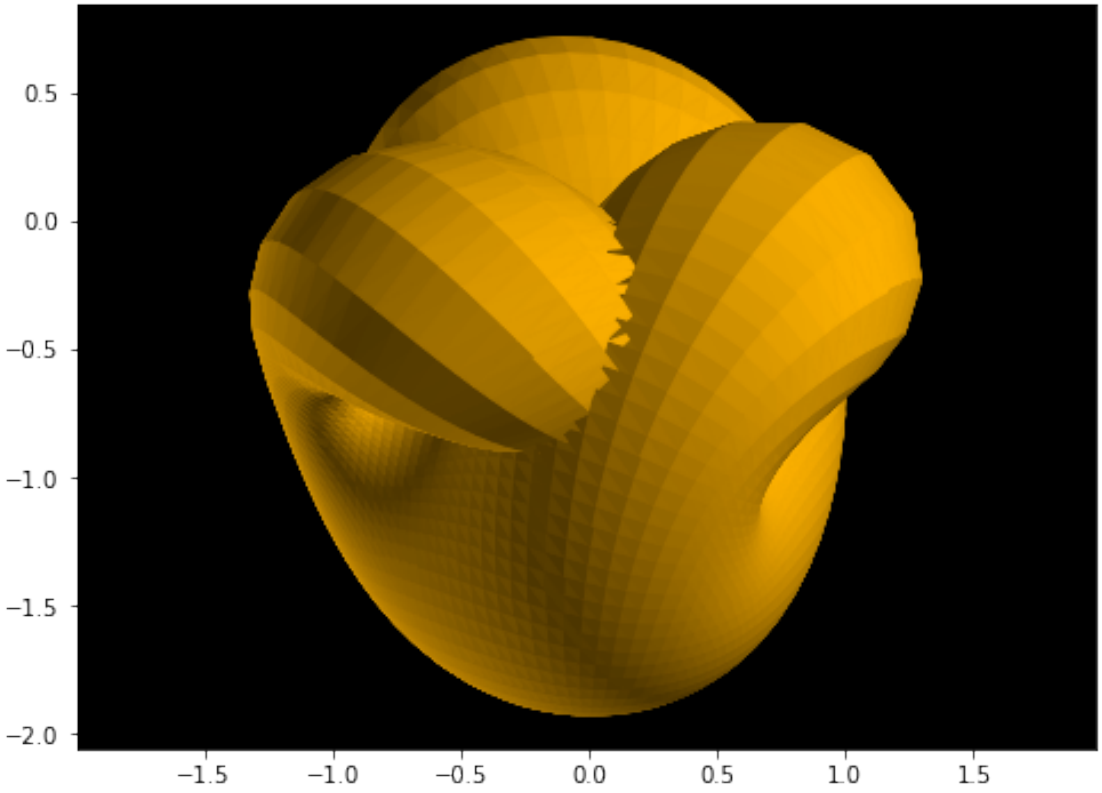
1.4.1 4.1 Défauts

L'algorithme du peintre donne des résultats assez satisfaisants pour des surfaces « simples ». Il possède cependant un certain nombre d'inconvénients. L'un d'entre eux est que la distance d'une face à l'observateur est une valeur assez vague. De quel point de la face parlons nous ? Deux faces peuvent avoir des points à la fois plus proches et plus éloignés, laquelle des deux doit-on tracer en premier ? En particulier, deux faces peuvent s'intersecter, et dans ce cas l'algorithme donne des résultats pas convaincants du tout. Prenons pour exemple la *surface de Boy*. Il y aurait beaucoup à dire sur cette surface. Le lecteur intéressé ira faire un tour sur Internet.

[60]:
```
def F_boy(u, v):
    r5 = math.sqrt(5)
    w = u * (math.cos(v) + 1j * math.sin(v))
    w1 = w ** 6 + r5 * w ** 3 - 1
    g1 = -3/2 * (w * (1 - w ** 4) / w1).imag
    g2 = -3/2 * (w * (1 + w ** 4) / w1).real
    g3 = ((1 + w ** 6) / w1).imag - 1/2
    r = g1 ** 2 + g2 ** 2 + g3 ** 2
```

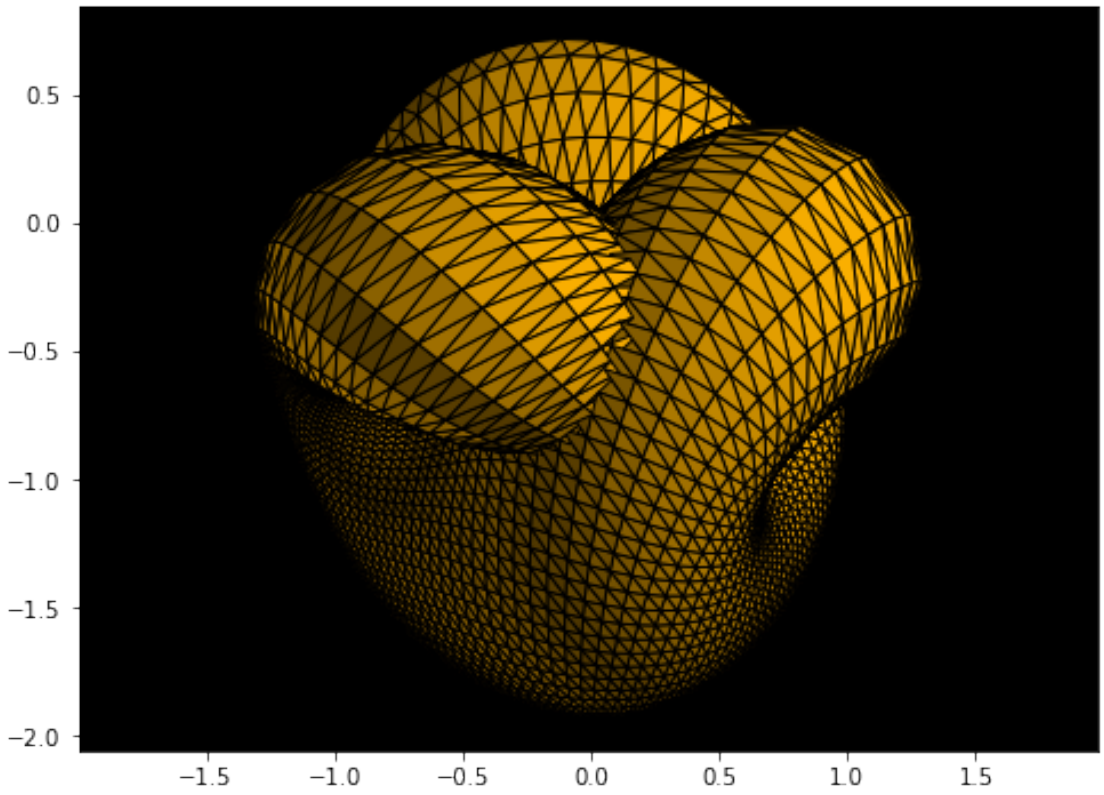
```
    return (g1 / r, g2 / r, g3 / r)

bornes_boy = [0, 1, 0, 2 * math.pi]
```

[61]:
```
tracer2(F_boy, bornes_boy, 50, 100, (100, -20, 40), edge=False)
```



Cette surface possède des points multiples. En ces points, les faces s'intersectent et l'apparence finale est minable. Pour remédier à cela on peut, au prix d'une augmentation de la complexité de l'algorithme, détecter les intersections de faces et par exemple, en cas d'intersection, découper les faces en « sous-faces ». Nous ne tenterons pas l'expérience ici, il faudrait tout réécrire ! Pour limiter les dégâts, on peut afficher les bordures des faces. Ça cache la misère …

[62]:
```
tracer2(F_boy, bornes_boy, 50, 100, (100, -20, 40), edge=True)
```



1.4.2 4.2 Lissage

Une autre amélioration consiste à « lisser » les couleurs. Un certain nombre de techniques existent pour cela, dont les deux plus connues sont le lissage de Gouraud et le lissage de Phong.

- L'algorithme de Gouraud (Gouraud shading) calcule les couleurs à attribuer aux sommets de la face, puis interpole à partir de ces couleurs les couleurs à attribuer aux autres points de la face. Cet algorithme donne de bons résultats mais présente quelques inconvénients. Par exemple, les effets lumineux localisés à l'intérieur d'une face (point brillant, etc.) sont totalement gommés.
- L'algorithme de Phong (Phong shading) calcule les normales aux sommets de la face, puis interpole à partir de ces normales les normales aux autres points de la face. On possède alors en chaque point de la face un vecteur normal qui permet de lui attribuer une couleur. Cet algorithme donne de bien meilleurs résultats que l'algorithme de Gouraud, mais les temps de calcul sont plus importants.

1.4.3 4.3 Modèle d'éclairement

Ici aussi, des progrès sont faciles à réaliser. Nous avons uniquement tenu compte dans notre modèle d'éclairement de

- La réflexion ambiante, qui est une « couleur constante » attachée à la surface.
- La réflexion diffuse, qui est une lumière renvoyée par la surface dans toutes les directions, et qui ne dépend que de l'angle avec lequel les rayons issus de la source lumineuse rencontrent la surface.

On peut sans difficulté y ajouter une composante de *réflexion spéculaire*. Ce type de réflexion dépend aussi de l'angle que fait la face avec la droite qui va de l'observateur à la face. Cela permet en particulier de simuler des objets plus ou moins brillants.

Libre à vous de tenter l'expérience, ce n'est pas si difficile que cela. Il suffit de modifier la fonction `tracer_face`, au paragraphe 3.4.

1.4.4 4.4 Le mot de la fin

Si j'osais, je dirais que nous n'avons fait qu'effleurer la surface …