

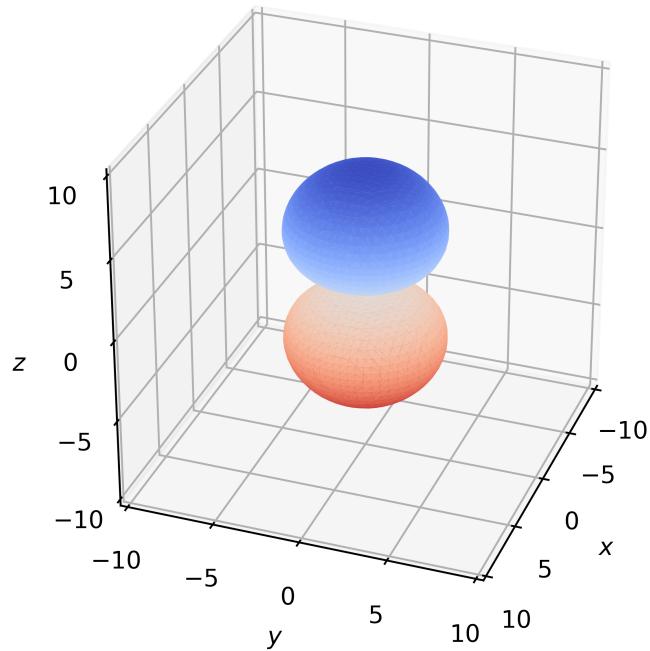
# 氢原子轨道与杂化轨道绘制

Solara570

---

## O、目标

使用Python绘制原子轨道与杂化轨道（的等概率密度面），效果如下：



此处演示使用的版本为 `Python 3.9.7`, 建议版本号 `>= 3.7`.

其他还会使用的包及版本号如下：（版本号有一点差别也不要紧，通常不会有大问题）

- `numpy 1.21.4`
  - `scipy 1.10.1`
  - `scikit-image 0.21.0`
  - `matplotlib 3.5.0`
- 

## 一、要点回顾

1. 氢原子轨道波函数 $\Psi$ 是如下定态薛定谔方程的解

$$\left( -\frac{\hbar^2}{2\mu} \nabla^2 - \frac{e^2}{4\pi\epsilon_0 r} \right) \Psi = E\Psi \quad (1)$$

其中 $\hbar$ 为约化普朗克常数,  $\mu = \frac{m_p m_e}{m_p + m_e}$ 为原子核与电子的约化质量,  $\nabla^2$ 为拉普拉斯算符 (梯度的散度),  $e$ 为电子电量,  $\epsilon_0$ 为真空介电常数,  $E$ 为原子轨道能量.

## 2. 量子数( $n, \ell, m$ )的引入

为了让 $\Psi$ 有物理意义 (单值、有限), 求解过程还会自然地引入三个量子数: 主量子数 $n$ , 角量子数 $\ell$ 和磁量子数 $m$ . 它们需要满足的关系有:  $n \in \mathbb{N}^+$ ,  $\ell \in \mathbb{N}$ ,  $m \in \mathbb{Z}$ ,  $n > \ell$ ,  $\ell \geq |m|$ .

不同的( $n, \ell, m$ )组合产生不同的原子轨道, 如 $(n, \ell, m) = (1, 0, 0)$ 表示1s轨道,  $(n, \ell, m) = (3, 2, 1)$ 表示3d<sub>1</sub>轨道.

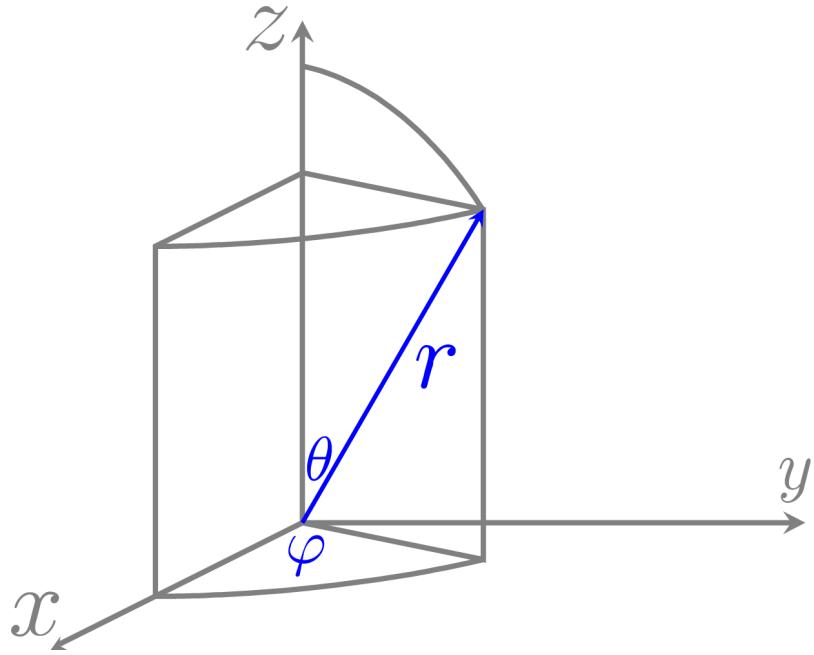
## 3. 氢原子轨道波函数的完整形式

$$\Psi_{n\ell m}(r, \theta, \varphi) = \sqrt{\left(\frac{2}{na_0^*}\right)^3 \frac{(n-\ell-1)!}{2n(n+\ell)!}} \rho^\ell \exp\left(-\frac{\rho}{2}\right) L_{n-\ell-1}^{2\ell+1}(\rho) \cdot Y_\ell^m(\theta, \varphi) \quad (2)$$

其中 $\rho = \frac{2r}{na_0^*}$ , 是一个简化书写形式的无量纲中间量.  $a_0^*$ 是约化玻尔半径, 大约是52.9 pm.  $L_{n-\ell-1}^{2\ell+1}(\cdot)$ 是连带拉盖尔多项式.  $Y_\ell^m(\theta, \varphi)$ 是球谐函数.

为了方便画图, 这里采用类似原子单位的处理方式, 令 $a_0^* = 1$ . 此时 $r$ 代表着真实的径向距离是 $a_0^*$ (= 52.9 pm)的多少倍.

注: 此处使用的球坐标系的形式——径向距离 $r$ , 极角 $\theta$ 和方位角 $\varphi$ .



## 4. 其他

- 波函数的模平方 $|\Psi(r, \theta, \varphi)|^2$ 代表了粒子出现在 $(r, \theta, \varphi)$ 处的概率密度
- 波函数可以叠加 (线性组合):  $\Psi_{\text{new}} = c_1 \Psi_1 + c_2 \Psi_2$

## 二、氢原子轨道的绘制

# 1. 导入包

需要提前安装 `numpy`, `scipy`, `scikit-image`, `matplotlib`.

导入的主要内容有:

- $\Psi$ 中出现的特殊函数, 例如阶乘, 连带拉盖尔多项式 $L_{n-\ell-1}^{2\ell+1}(\rho)$ 和球谐函数 $Y_\ell^m(\theta, \varphi)$
- 在三维空间中寻找等值面的 `marching cubes` 算法
- 绘图所需的组件

In [1]:

```
import numpy as np

# 特殊函数: 阶乘 (factorial)、连带拉盖尔多项式 (genlaguerre)、球谐函数 (sph_harm)
from scipy.special import factorial, genlaguerre, sph_harm

# 等值面搜索: marching_cubes
from skimage.measure import marching_cubes

# 绘图组件
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d.art3d import Poly3DCollection

# Jupyter Notebook内嵌绘图用, 本地运行则不写这一行
%matplotlib notebook
```

## 2. 定义氢原子轨道波函数

把定义抄过来:

$$\Psi_{nlm}(r, \theta, \varphi) = \sqrt{\left(\frac{2}{na_0^*}\right)^3 \frac{(n-\ell-1)!}{2n(n+\ell)!}} \rho^\ell \exp\left(-\frac{\rho}{2}\right) L_{n-\ell-1}^{2\ell+1}(\rho) \cdot Y_\ell^m(\theta, \varphi) \quad (3)$$

其中 $\rho = \frac{2r}{na_0^*}$ , 是一个简化书写形式的无量纲中间量.  $a_0^*$ 是约化玻尔半径, 大约是52.9 pm.  
 $L_{n-\ell-1}^{2\ell+1}(\cdot)$ 是连带拉盖尔多项式.  $Y_\ell^m(\theta, \varphi)$ 是球谐函数.

为了方便画图, 这里采用类似原子单位的处理方式, 令 $a_0^* = 1$ . 此时 $r$ 代表着真实的径向距离是 $a_0^*(= 52.9 \text{ pm})$ 的多少倍.

注: 此处使用的球坐标系的形式——径向距离 $r$ , 极角 $\theta$ 和方位角 $\varphi$ .

直接翻译成代码就行.

In [2]:

```
# 定义约化玻尔半径a0, 原子单位下直接设置为1
a0 = 1

def hydrogen_wave_function(n, l, m):
    # 径向部分R(r), 注意genlaguerre的参数顺序
    def R(r):
        factor = np.sqrt((2. / (n * a0))**3 * \
                          factorial(n - 1 - 1) / (2 * n * factorial(n + 1))) # 系数
        rho = 2 * r / (n * a0) # 中间量rho
        return factor * (rho**l) * np.exp(-rho / 2) * \
               genlaguerre(n - 1 - 1, 2 * l + 1)(rho)

    # 角向部分Y(theta, phi)就是球谐函数sph_harm
    # 注意量子数l与m 和 自变量theta与phi的顺序都是颠倒的!
    def Y(theta, phi):
        return sph_harm(m, l, phi, theta)
```

```
# 径向部分R(r) 和角向部分Y(theta, phi) 相乘, 得到一个关于(r, theta, phi) 的函数, 即原子轨道
return lambda r, theta, phi: R(r) * Y(theta, phi)
```

再代入 $n$ ,  $\ell$ 和 $m$ 三个量子数就得到了对应的轨道波函数 `hydrogen_wave_function(n, l, m)`.

以 $2p_0$ 轨道 (又名 $2p_z$ 轨道) 为例, 对应的量子数为 $(n, \ell, m) = (2, 1, 0)$ :

In [3]:

```
# 原子轨道参数, 当前为2p0轨道
n, l, m = 2, 1, 0

# 2p0轨道波函数
psi = hydrogen_wave_function(n, l, m)

# `psi`本身是一个函数, 后面还可以继续带参数(r, theta, phi)来计算实际的波函数值
print(f"psi = {psi}")
print(f"psi(1, np.pi/4, np.pi/3) = {psi(1, np.pi/4, np.pi/3)}")      # 注意结果是复数

psi = <function hydrogen_wave_function.<locals>.lambda at 0x0000020F8F9500D0>
psi(1, np.pi/4, np.pi/3) = (0.04277478503902706+0j)
```

### 3. 定义坐标范围

把空间网格化, 确定边界 `limit` 和三个正方向上取的点数 `n_points`.

网格化的区域是一个边长为 `2 * limit` 且中心在原点的正方体. 每个方向上均匀取 `n_points` 个点, 网格点总数为  $(n\_points)^3$ .

In [4]:

```
# limit: 确定网格点的区间范围
limit = 10

# n_points: 每个方向上均匀取点的数目
n_points = 50

# vec: 临时变量, 从 -limit 到 limit 均匀取 n_points 个点得到的向量
vec = np.linspace(-limit, limit, n_points)

print(f"vec = {vec}")
print(f"len(vec) = {len(vec)}")

vec = array([-10.          , -9.59183673, -9.18367347, -8.7755102 ,
   -8.36734694, -7.95918367, -7.55102041, -7.14285714,
   -6.73469388, -6.32653061, -5.91836735, -5.51020408,
   -5.10204082, -4.69387755, -4.28571429, -3.87755102,
   -3.46938776, -3.06122449, -2.65306122, -2.24489796,
   -1.83673469, -1.42857143, -1.02040816, -0.6122449 ,
   -0.20408163,  0.20408163,  0.6122449 ,  1.02040816,
    1.42857143,  1.83673469,  2.24489796,  2.65306122,
    3.06122449,  3.46938776,  3.87755102,  4.28571429,
    4.69387755,  5.10204082,  5.51020408,  5.91836735,
    6.32653061,  6.73469388,  7.14285714,  7.55102041,
    7.95918367,  8.36734694,  8.7755102 ,  9.18367347,
    9.59183673,  10.        ])

len(vec) = 50
```

得到单个方向上的均匀取点后, 用 `np.meshgrid` 生成三维网格的格点坐标.

In [5]:

```
# 生成三维网格
# X记录每个点的x坐标, Y记录y坐标, Z记录z坐标
X, Y, Z = np.meshgrid(vec, vec, vec)

print(f"X.shape = {X.shape}, Y.shape = {Y.shape}, Z.shape = {Z.shape}")
```

```
X.shape = (50, 50, 50) Y.shape = (50, 50, 50) Z.shape = (50, 50, 50)
```

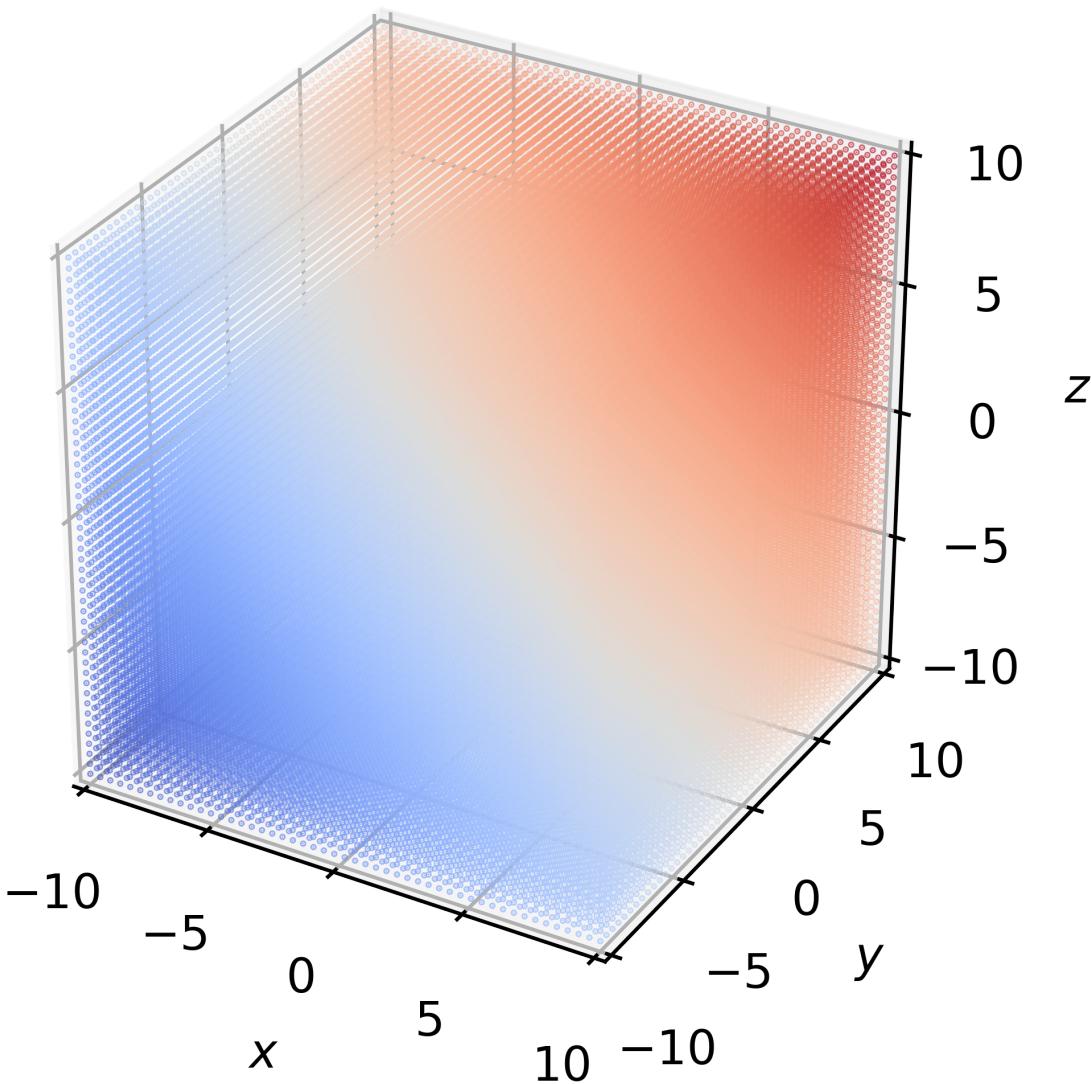
In [6]:

```
# 查看每个点的直角坐标(x, y, z)
# 后续并不使用`coord_xyz`, 只用来确认数据
coords_xyz = np.vstack(list(map(np.ravel, (X, Y, Z)))) .T

print(f"len(coords_xyz) = {len(coords_xyz)}")
print(coords_xyz)

len(coords_xyz) = 125000
[[-10.          -10.          -10.          ]
 [-10.          -10.         -9.59183673]
 [-10.          -10.         -9.18367347]
 ...
 [ 10.           10.          9.18367347]
 [ 10.           10.          9.59183673]
 [ 10.           10.          10.          ]]
```

或者直接把它们标出来:



格点是直角坐标( $x, y, z$ ), 然而 $\Psi$ 是关于( $r, \theta, \varphi$ )的函数, 需要把格点按下式变换为球坐标.

$$\begin{cases} x = r \sin \theta \cos \varphi \\ y = r \sin \theta \sin \varphi \\ z = r \cos \theta \end{cases} \Leftrightarrow \begin{cases} r = \sqrt{x^2 + y^2 + z^2} \\ \theta = \arccos \frac{z}{r} = \arccos \frac{z}{\sqrt{x^2 + y^2 + z^2}} \\ \varphi = \arctan \frac{y}{x} \end{cases} \quad (4)$$

```
In [7]: # 由坐标变换, 每个格点从直角坐标系下的(x, y, z)变成球坐标系下的(R, THETA, PHI)
R = np.sqrt(X**2 + Y**2 + Z**2)
THETA = np.arccos(Z / R)
PHI = np.arctan2(Y, X)

print(f"R.shape = {R.shape}, THETA.shape = {THETA.shape}, PHI.shape = {PHI.shape}")

R.shape = (50, 50, 50) THETA.shape = (50, 50, 50) PHI.shape = (50, 50, 50)
```

```
In [8]: # 查看每个点的球坐标(r, theta, phi)
# 后续并不使用`coord_rtf`, 只用来确认数据
coords_rtf = np.vstack(list(map(np.ravel, (R, THETA, PHI)))).T

print(len(coords_rtf))
print(coords_rtf)

len(coords_rtf) = 125000
[[17.32050808 2.18627604 -2.35619449]
 [17.08810498 2.16677211 -2.35619449]
 [16.86237997 2.14673823 -2.35619449]
 ...
 [16.86237997 0.99485442 0.78539816]
 [17.08810498 0.97482054 0.78539816]
 [17.32050808 0.95531662 0.78539816]]
```

```
In [9]: # 代入波函数`psi`中, 得到每个格点的波函数值
psi_values = psi(R, THETA, PHI)

print(psi_values.shape)
print(psi_values[0][0][0]) # 直角坐标系下(-10, -10, -10)处的波函数值

psi_values.shape = (50, 50, 50)
psi_values[0][0][0] = (-0.0001728819028298787+0j)
```

## 4. 等概率密度面绘制

使用 `marching_cubes` 寻找概率密度 $|\Psi|^2$ 的等值面 (i.e. 所有 $|\Psi|^2 = \text{常数}C$ 的点的集合)

```
In [10]: # prob_dens: 概率密度|Psi|^2
prob_dens = np.abs(psi_values)**2
```

```
In [11]: # iso_value: 目标值, 也就是上面的常数C
iso_value = 4e-4

# 初次尝试
verts, faces, _, _ = marching_cubes(
    prob_dens,
    level=iso_value,
)
print(verts)
print(faces)
```

```
verts = array([[13.979592, 21.        , 16.        ],
   [14.        , 21.        , 15.840323],
   [14.        , 20.949831, 16.        ],
   ...,
```

```

[35.204258, 27.      , 34.      ],
[35.02041 , 28.      , 16.      ],
[35.02041 , 28.      , 33.      ]], dtype=float32)
faces = array([[ 2,     1,     0],
               [ 0,     3,     2],
               [ 6,     5,     4],
               ...,
               [3830, 3902, 3813],
               [3903, 3836, 3822],
               [3836, 3903, 3825]])

```

结果为一个三角化的等值面. `verts` 本应该代表顶点坐标, 但是数值不正确 (默认从0开始计数, 且步长默认为1). `faces` 代表每个三角面的顶点编号.

为了获得正确的顶点坐标, 需要再做三件事:

- 给 `marching_cubes` 加上 `spacing` 参数表示每个方向的步长. 步长 `step` 可以根据 `limit` 和 `n_points` 参数简单计算得到.
- 平移结果至坐标原点. 我们选择的区间 `[-limit, limit]` 是对称的, 所以直接给每个坐标减掉向量 `(limit, limit, limit)` 即可.
- 可能是受到 `marching_cubes` 坐标输出顺序的影响, 所有顶点的  $x$  坐标和  $y$  坐标需要互换 (相关说明见附录A, 这里只提结论, 不再展开)

```
In [12]: # 计算步长step: [-limit, limit]区间被分成(n_points-1)份, 每段的长度就是step
step = 2 * limit / (n_points - 1)
```

```

# 把step传入marching_cubes的spacing参数中
verts, faces, _, _ = marching_cubes(
    prob_dens,
    level=iso_value,
    spacing=(step, step, step),
)

# 将结果平移至原点
verts -= limit

# 最后互换x坐标和y坐标
verts[:, [0, 1]] = verts[:, [1, 0]]

# 结果确认
print(f"verts = {verts}")

```

```

verts = array([[-1.42857143, -4.29404395, -3.46938776],
              [-1.42857143, -4.28571429, -3.53456186],
              [-1.44904857, -4.28571429, -3.46938776],
              ...,
              [ 1.02040816,  4.36908488,  3.87755102],
              [ 1.42857143,  4.29404434, -3.46938776],
              [ 1.42857143,  4.29404434,  3.46938776]])

```

此时的 `verts` 坐标全部正常.

获得正确的等值面后, 接下来就是把它画出来. 这里介绍3种可视化方法:

- `ax.plot_trisurf`
- `Poly3DCollection`
- 将写入一个 `obj` 文件中, 借助其他软件 (如Blender, 3D Viewer等) 展示结果

为了前两种方法的方便, 先写个辅助函数, 调用时返回一个调整好的画布和内框.

```
In [13]: # 辅助函数, 每次返回一个调整好的画布`fig`和内框`ax`
```

```

def new_fig_and_ax(plot_range=10):
    fig = plt.figure()
    ax = fig.add_subplot(1, 1, 1, projection="3d")

    # 设置坐标比例关系, 让图像变得方正
    ax.set_box_aspect([1, 1, 1])

    # 设置合适的视图俯仰角和方位角, 让坐标轴方向符合直观
    # x轴对着屏幕外方向, y轴朝右, z轴朝上
    ax.view_init(elev=30, azim=20)

    # 设置坐标轴标签
    ax.set_xlabel("$x$")
    ax.set_ylabel("$y$")
    ax.set_zlabel("$z$")

    # 设置坐标轴范围为[-plot_range, plot_range]
    ax.set_xlim(-plot_range, plot_range)
    ax.set_ylim(-plot_range, plot_range)
    ax.set_zlim(-plot_range, plot_range)

    # 设置坐标轴刻度
    ticks = np.linspace(-plot_range, plot_range, 5)
    ax.set_xticks(ticks)
    ax.set_yticks(ticks)
    ax.set_zticks(ticks)

return fig, ax

```

### (1) 方法一: 借助 `ax.plot_trisurf`

此处 `plot_trisurf` 的传参顺序: 所有点的 `X` 坐标, 所有点的 `Y` 坐标, 所有三角面的顶点编号, 所有点的 `Z` 坐标.

上述参数3来自 `faces`; 其他参数来自 `verts`, 但需要把同类坐标整合在一起.

```

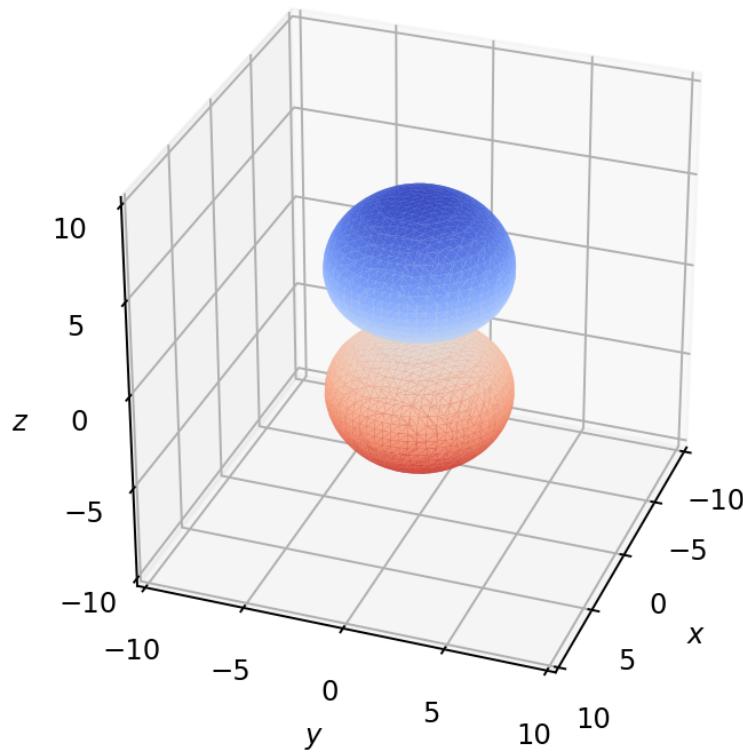
In [14]: # 方法一: ax.plot_trisurf
fig, ax = new_fig_and_ax()

iso_surface = ax.plot_trisurf(
    verts[:, 0], verts[:, 1], faces, verts[:, 2],
    lw=0, # 线宽设置为0
    cmap="coolwarm_r", # 可以带上颜色让图像更好看, 但是注意这个颜色与波函数的物理意义无关
)

# 展示结果
plt.show()

# 或者使用plt.savefig(...) 把图像存储到文件中
# plt.savefig(
#     "orbital_plot.png",
#     transparent=True,
#     bbox_inches='tight',
#     dpi=600,
# )

```



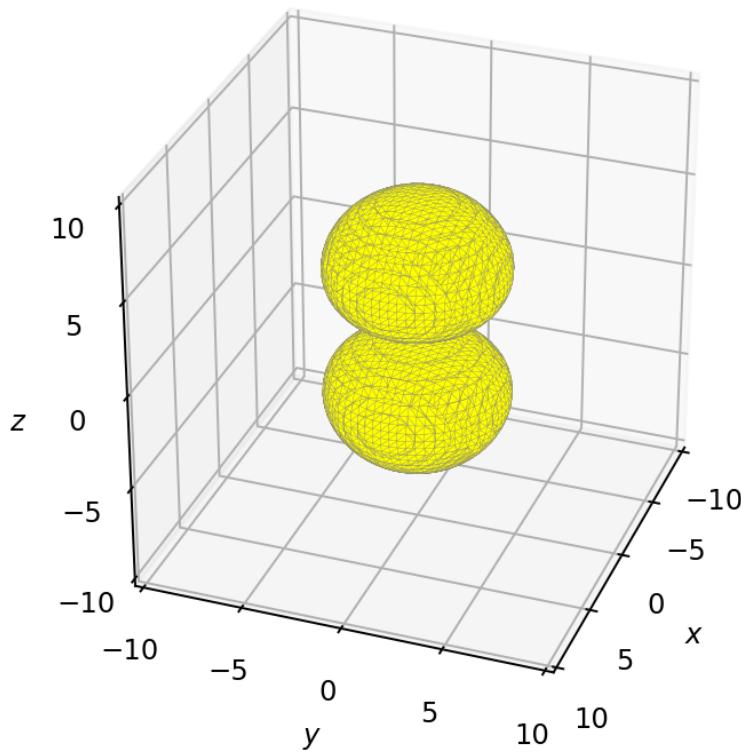
## (2) 方法二: 借助 Poly3DCollection

结果和 `plot_trisurf` 类似, 但是 `Poly3DCollection` 的参数更简单: 每个三角面三个顶点坐标的集合, 只需用 `verts[faces]` 即可.

```
In [15]: # 方法二: Poly3DCollection
fig, ax = new_fig_and_ax()

mesh = Poly3DCollection(verts[faces], lw=0.1) # 线宽0.1
mesh.set_facecolor("yellow") # 黄色的表面
mesh.set_edgecolor("grey") # 灰色的边线
ax.add_collection3d(mesh) # 直接加入ax中

plt.show()
```



### (3) 方法三: 写入 obj 文件, 借助其他软件展示结果

处理起来最自由的一种方式, 而且 obj 文件的结构非常简单, 这里只需要知道:

- o 开头的行 —— 对象信息: o 【对象名称】
- v 开头的行 —— 顶点信息: v 【顶点x坐标】 【顶点y坐标】 【顶点z坐标】
- f 开头的行 —— 面信息: f 【三角面顶点1编号】 【三角面顶点2编号】 【三角面顶点3编号】

In [16]: # 方法三: 写入obj文件

```
filename = "isosurface.obj"
with open(filename, "w") as f:
    # (1) o开头的行: 写入对象名称, 可以自行指定
    f.write(f"o Isosurface-2pz {iso_value}\n")
    # (2) v开头的行: 写入顶点坐标(x, y, z)
    for vert_coords in verts:
        x, y, z = vert_coords
        f.write("v %s %s %s \n" % (x, y, z))
    # (3) f开头的行: 写入三角面的顶点编号(id_1, id_2, id_3), 注意编号从1开始计
    for vert_ids in faces:
        id_1, id_2, id_3 = vert_ids + 1
        f.write("f %s %s %s \n" % (id_1, id_2, id_3))
```

In [17]: # obj文件内容

```
with open(filename, "r") as f:
    lines = f.readlines()

print(f"{len(lines)}\n")
print("".join(lines[:5]), "...")
```

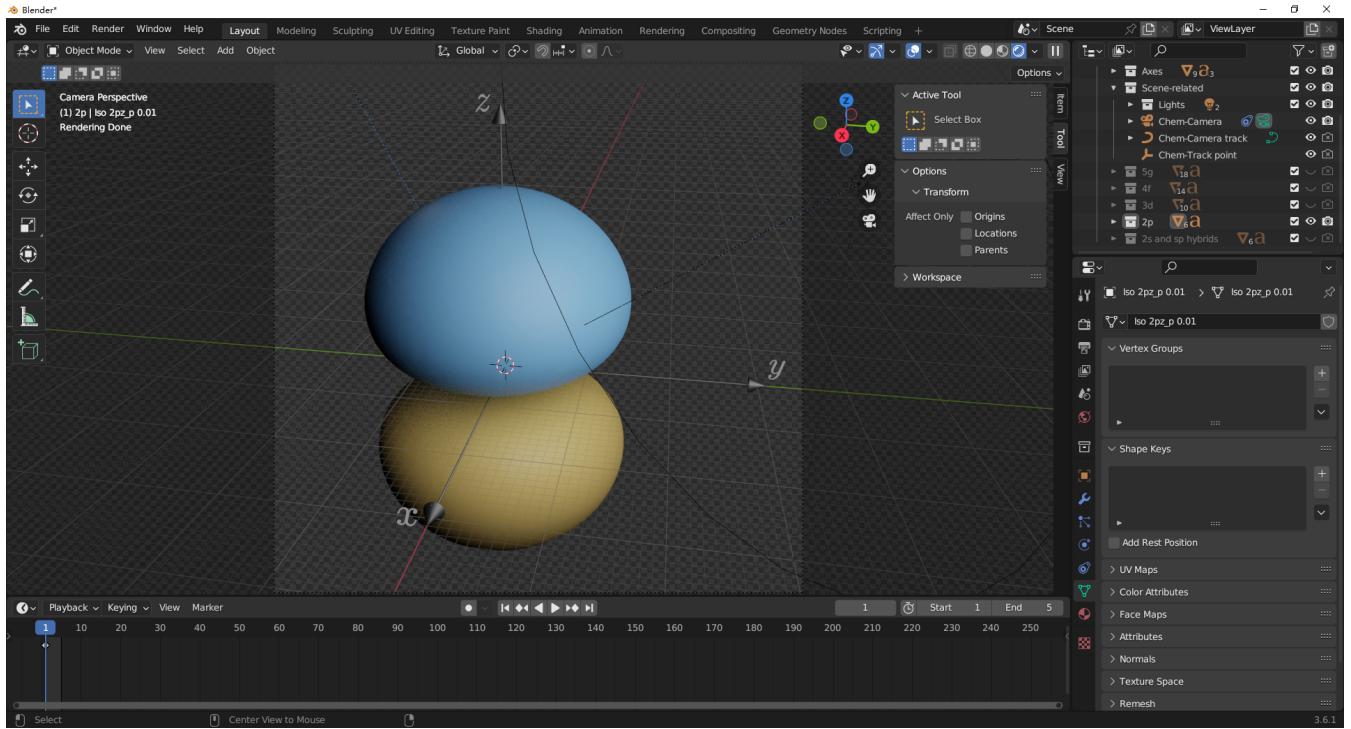
len(lines) = 11705

```

o Isosurface-2pz 0.0004
v -1.4285714285714288 -4.294043949672154 -3.4693877551020407
v -1.4285714285714288 -4.285714285714286 -3.534561857885244
v -1.4490485677913743 -4.285714285714286 -3.4693877551020407
v -1.4285714285714288 -4.285714285714286 -3.2705976525131533
...
f 3825 3902 3805
f 3903 3831 3811
f 3831 3903 3814
f 3904 3837 3823
f 3837 3904 3826

```

然后将 `obj` 文件导入其他软件中处理, 例如下面是用Blender绘制的结果.



## 5. “化学视角”原子轨道的绘制

因为波函数的角向部分（也就是球谐函数）含有 $e^{im\varphi}$ , 其中 $i = \sqrt{-1}$ 为虚数单位, 所以刚才的波函数大多是复函数. 不过, 我们可以选择 $\Psi_{n\ell m}$ 和 $\Psi_{n\ell \bar{m}}$ （角标 $\bar{m}$ 代表 $-m$ ）的线性组合来消去复数, 从而得到实函数. 原理就是著名的欧拉公式:

$$\begin{aligned} \frac{1}{2} (e^{im\varphi} + e^{-im\varphi}) &= \cos m\varphi \\ \frac{1}{2i} (e^{im\varphi} - e^{-im\varphi}) &= \sin m\varphi \end{aligned} \tag{5}$$

如果 $\Psi_{n\ell m}$ 和 $\Psi_{n\ell \bar{m}}$ 的角向部分恰好互为共轭, 也就是满足 $Y_\ell^{-m}(\theta, \varphi) = [Y_\ell^m(\theta, \varphi)]^*$ , 那么就可以直接组合:

$$\begin{bmatrix} \Psi_{\text{new},1} \\ \Psi_{\text{new},2} \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ -i & i \end{bmatrix} \begin{bmatrix} \Psi_{n\ell m} \\ \Psi_{n\ell \bar{m}} \end{bmatrix} \tag{6}$$

但是 `scipy` 使用的球谐函数带有Condon-Shortley相位因子 $(-1)^m$ , 此时满足的关系则是 $Y_\ell^{-m}(\theta, \varphi) = (-1)^m [Y_\ell^m(\theta, \varphi)]^*$ , 并不能按上式直接加和. 为了消除CS相因子的影响, 当磁量子数 $m$ 为正数时, 我们给它再乘上一个 $(-1)^m$ . 所以真正的组合其实是这样的:

$$\begin{bmatrix} \Psi_{\text{new},1} \\ \Psi_{\text{new},2} \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} (-1)^m & 1 \\ (-1)^{m+1} \cdot i & i \end{bmatrix} \begin{bmatrix} \Psi_{n\ell m} \\ \Psi_{n\ell \bar{m}} \end{bmatrix} \quad (m > 0) \quad (7)$$

例如 $2p_x$ 和 $2p_y$ 轨道, 它们是 $\Psi_{211}$ 和 $\Psi_{21\bar{1}}$ 的线性组合. 将 $(n, \ell, m) = (2, 1, 1)$ 代入上式, 得到:

$$\begin{bmatrix} \Psi_{2p_x} \\ \Psi_{2p_y} \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} -1 & 1 \\ i & i \end{bmatrix} \begin{bmatrix} \Psi_{211} \\ \Psi_{21\bar{1}} \end{bmatrix} \quad (8)$$

In [18]:

```
# 2px轨道波函数
psi_2px = lambda r, theta, phi: \
    (-1) * 1 / np.sqrt(2) * hydrogen_wave_function(2, 1, 1)(r, theta, phi) + \
    1 / np.sqrt(2) * hydrogen_wave_function(2, 1, -1)(r, theta, phi)

# 2py轨道波函数
psi_2py = lambda r, theta, phi: \
    1j / np.sqrt(2) * hydrogen_wave_function(2, 1, 1)(r, theta, phi) + \
    1j / np.sqrt(2) * hydrogen_wave_function(2, 1, -1)(r, theta, phi)

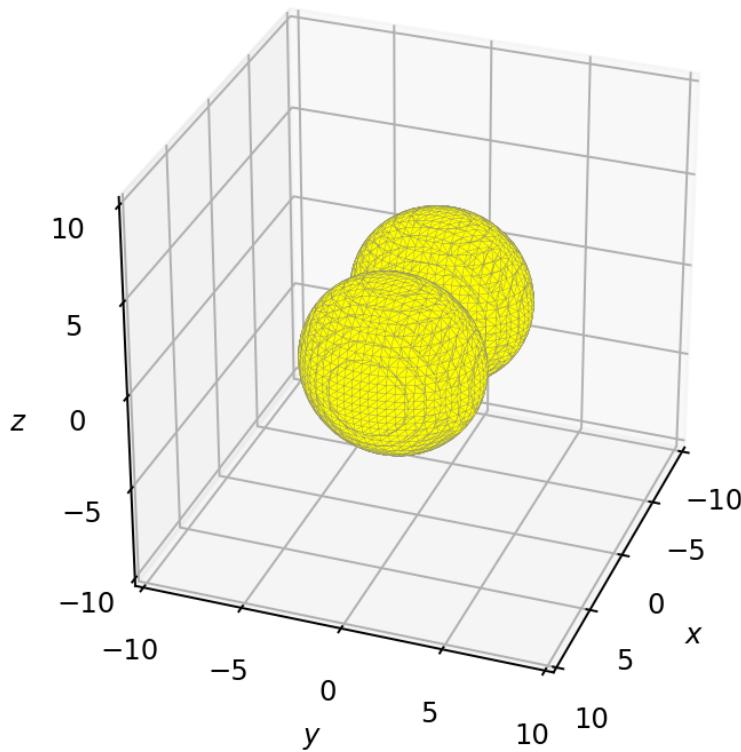
# (思考: 这里同时用到了两个波函数, 并且计算的结果仍旧是复数, 是否有更简洁&更适当的写法? )
# (提示: 使用z.real或者z.imag获取复数z的实部或虚部)
```

按之前的做法画 $2p_x$ 轨道的图像, 网格和等值面的常数 $C$ 用之前的值, 其他变量名加上 2px 后缀做区分.

In [19]:

```
# 找到等值面
iso_value = 4e-4
psi_2px_values = psi_2px(R, THETA, PHI)
prob_dens_2px = np.abs(psi_2px_values)**2 # 2px轨道概率密度
verts_2px, faces_2px, _, _ = marching_cubes(
    prob_dens_2px, level=iso_value,
    spacing = (step, step, step),
)
verts_2px == limit
verts_2px[:, [0, 1]] = verts_2px[:, [1, 0]]

# 绘制等值面
fig, ax = new_fig_and_ax()
mesh_2px = Poly3DCollection(verts_2px[faces_2px], lw=0.1) # 线宽0.1
mesh_2px.set_facecolor("yellow") # 黄色的表面
mesh_2px.set_edgecolor("grey") # 灰色的边线
ax.add_collection3d(mesh_2px) # 直接加入ax中
plt.show()
```



形状与 $2p_z$ 轨道一致, 只是方向沿着 $x$ 轴, 确实是期望的结果.

$2p_y$ 轨道的画法类似, 不再赘述.

### 三、杂化轨道的绘制

杂化轨道本质上也是原子轨道的线性组合, 所以绘制方法同上面的第二章第5节, 但前提是需要知道组合形式.

以 $2s$ ,  $2p_x$ ,  $2p_y$ 和 $2p_z$ 轨道构成的 $sp^3$ 杂化轨道为例, 对应的线性组合为:

$$\begin{bmatrix} \Psi_{sp^3,a} \\ \Psi_{sp^3,b} \\ \Psi_{sp^3,c} \\ \Psi_{sp^3,d} \end{bmatrix} = \frac{1}{2} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & 1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix} \begin{bmatrix} \Psi_{2s} \\ \Psi_{2p_x} \\ \Psi_{2p_y} \\ \Psi_{2p_z} \end{bmatrix} \quad (9)$$

In [20]:

```
# 2s轨道
psi_2s = hydrogen_wave_function(2, 0, 0)

# 2px, 2py轨道定义同上, 只是这里换一种简洁的写法
psi_2px = lambda r, theta, phi: \
    (-1) * np.sqrt(2) * hydrogen_wave_function(2, 1, 1)(r, theta, phi).real
psi_2py = lambda r, theta, phi: \
    (-1) * np.sqrt(2) * hydrogen_wave_function(2, 1, 1)(r, theta, phi).imag

# 2pz轨道是最开始介绍的2p0轨道
psi_2pz = hydrogen_wave_function(2, 1, 0)
```

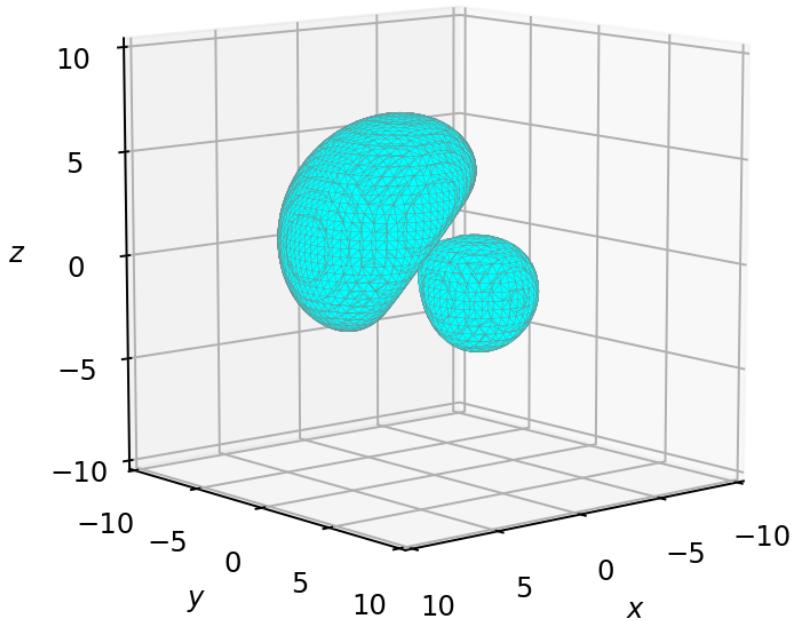
```
# 上述杂化轨道sp3_c的定义
```

```
psi_sp3_c = lambda r, theta, phi: \
    1 / 2 * psi_2s(r, theta, phi) - \
    1 / 2 * psi_2px(r, theta, phi) + \
    1 / 2 * psi_2py(r, theta, phi) - \
    1 / 2 * psi_2pz(r, theta, phi)
```

In [21]:

```
# 找到等值面
iso_value = 4e-4
psi_sp3_c_values = psi_sp3_c(R, THETA, PHI)
prob_dens_sp3_c = np.abs(psi_sp3_c_values)**2
verts_sp3_c, faces_sp3_c, _, _ = marching_cubes(
    prob_dens_sp3_c, level=iso_value,
    spacing = (step, step, step),
)
verts_sp3_c -= limit
verts_sp3_c[:, [0, 1]] = verts_sp3_c[:, [1, 0]]

# 绘制等值面
fig, ax = new_fig_and_ax()
mesh_sp3_c = Poly3DCollection(verts_sp3_c[faces_sp3_c], lw=0.1)
mesh_sp3_c.set_facecolor("cyan")
mesh_sp3_c.set_edgecolor("grey")
ax.add_collection3d(mesh_sp3_c)
ax.view_init(elev=10, azim=50) # 适当调整视角
plt.show()
```



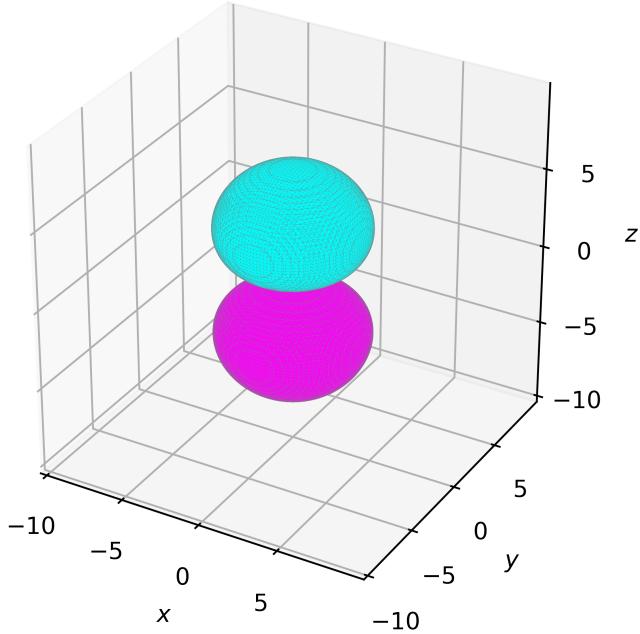
其余杂化轨道的绘制方法类似, 同样不再赘述.

## 四、练习题

### 1. 波函数的正负

利用 `matplotlib` (即前两种可视化方法) 重新绘制  $2p_z$  轨道, 但是要使用不同的颜色体现出波函数的正负情况, 效果如下:

(提示: 分别获取波函数为正和波函数为负的部分)



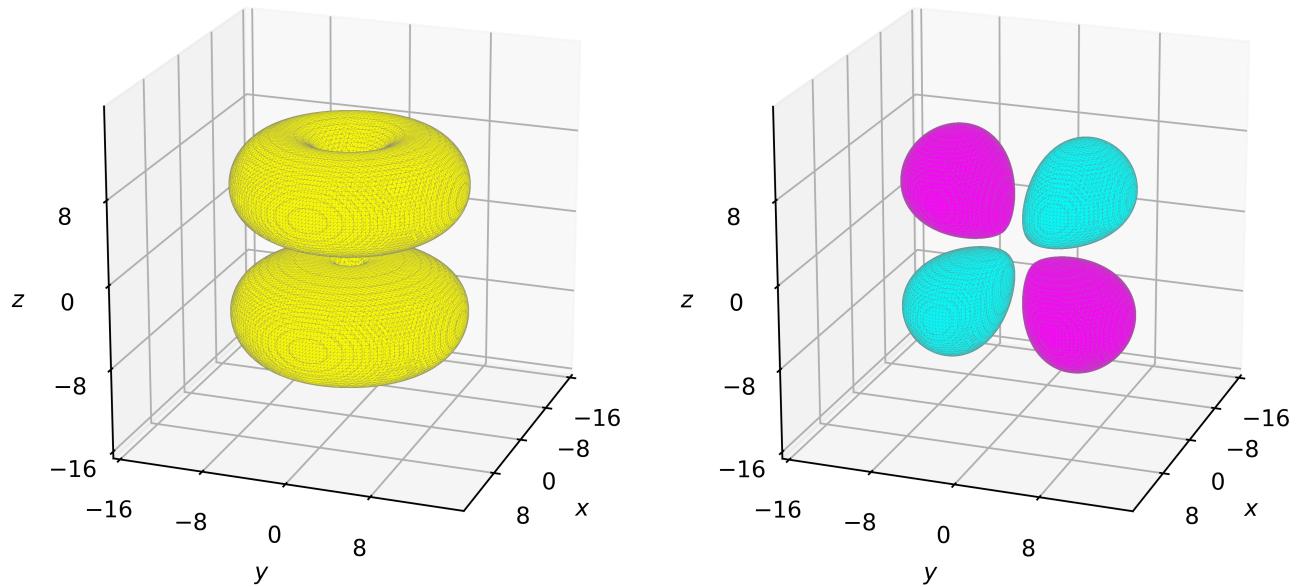
### 2. 绘制3d轨道

3d轨道有两种常见的表现形式, 一种是  $(3d_{-2}, 3d_{-1}, 3d_0, 3d_1, 3d_2)$ , 另一种是  $(3d_{xy}, 3d_{yz}, 3d_{z^2}, 3d_{xz}, 3d_{x^2-y^2})$ .

- 画出第一种表现形式的3d轨道, 也就是直接将量子数  $(n, \ell, m) = (3, 2, m)$  代入  $\Psi_{n\ell m}$  后得到的轨道, 其中磁量子数  $m \in \mathbb{Z}$  且  $-2 \leq m \leq 2$ .
- 根据以下变换关系, 画出第二种表现形式的3d轨道.

$$\begin{bmatrix} \Psi_{3d_{xy}} \\ \Psi_{3d_{yz}} \\ \Psi_{3d_{z^2}} \\ \Psi_{3d_{xz}} \\ \Psi_{3d_{x^2-y^2}} \end{bmatrix} = \begin{bmatrix} \frac{i}{\sqrt{2}} & 0 & 0 & 0 & -\frac{i}{\sqrt{2}} \\ 0 & \frac{i}{\sqrt{2}} & 0 & \frac{i}{\sqrt{2}} & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & \frac{1}{\sqrt{2}} & 0 & -\frac{1}{\sqrt{2}} & 0 \\ \frac{1}{\sqrt{2}} & 0 & 0 & 0 & \frac{1}{\sqrt{2}} \end{bmatrix} \begin{bmatrix} \Psi_{3d_{-2}} \\ \Psi_{3d_{-1}} \\ \Psi_{3d_0} \\ \Psi_{3d_1} \\ \Psi_{3d_2} \end{bmatrix} \quad (10)$$

作为参考, 如下两张图分别是  $3d_{-1}$  轨道和  $3d_{yz}$  轨道:



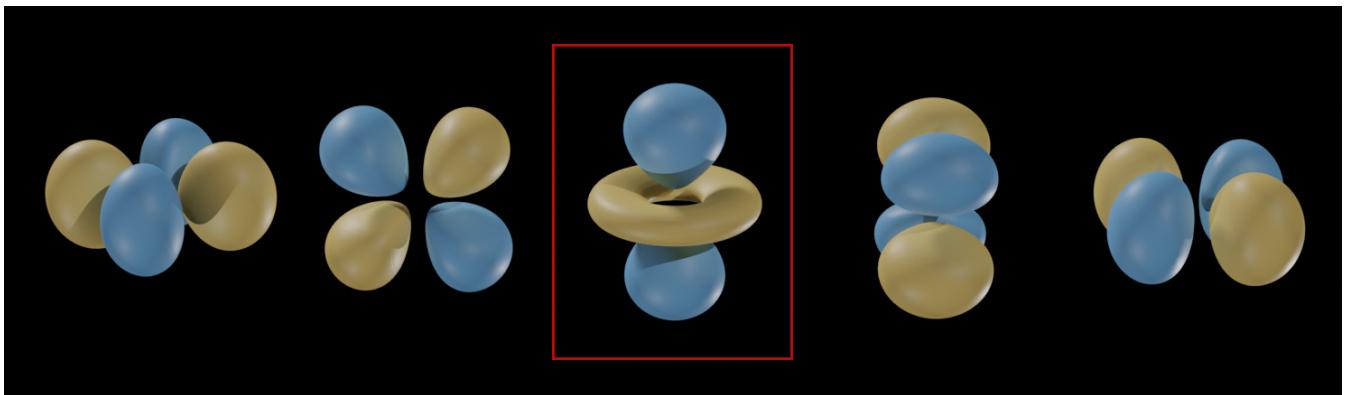
### 3. 绘制剩余的sp<sup>3</sup>杂化轨道

画出第三章中其余三个sp<sup>3</sup>杂化轨道的图像. 它们的形状应该与已有的轨道形状完全一致, 只是方向上有所区别.

### 4. “等价”的原子轨道 \*

2p轨道 ( $2p_x, 2p_y, 2p_z$ ) 是三个\*\*等价\*\*的原子轨道: 它们的形状完全一致, 只是朝向不同.

3d轨道 ( $3d_{xy}, 3d_{yz}, 3d_{z^2}, 3d_{xz}, 3d_{x^2-y^2}$ ) 大多是花瓣形的, 有四个分立的花瓣. 然而 $3d_{z^2}$ 轨道并非如此, 它在 $xOy$ 平面上有一个连通的环面, 让它显得格格不入.



于是就出现了3d轨道的另一种比较罕见的表现形式, 变换关系如下 (注意轨道顺序) :

$$\begin{bmatrix} \Psi_{3d, new1} \\ \Psi_{3d, new2} \\ \Psi_{3d, new3} \\ \Psi_{3d, new4} \\ \Psi_{3d, new5} \end{bmatrix} = M \begin{bmatrix} \Psi_{3d_{z^2}} \\ \Psi_{3d_{xz}} \\ \Psi_{3d_{yz}} \\ \Psi_{3d_{x^2-y^2}} \\ \Psi_{3d_{xy}} \end{bmatrix} \quad (11)$$

其中

$$M = \begin{bmatrix} \sqrt{\frac{1}{5}} & \sqrt{\frac{2}{5}} & 0 & \sqrt{\frac{2}{5}} & 0 \\ \sqrt{\frac{1}{5}} & \sqrt{\frac{2}{5}} \cos \frac{2\pi}{5} & \sqrt{\frac{2}{5}} \sin \frac{2\pi}{5} & \sqrt{\frac{2}{5}} \cos \frac{4\pi}{5} & \sqrt{\frac{2}{5}} \sin \frac{4\pi}{5} \\ \sqrt{\frac{1}{5}} & \sqrt{\frac{2}{5}} \cos \frac{4\pi}{5} & \sqrt{\frac{2}{5}} \sin \frac{4\pi}{5} & \sqrt{\frac{2}{5}} \cos \frac{8\pi}{5} & \sqrt{\frac{2}{5}} \sin \frac{8\pi}{5} \\ \sqrt{\frac{1}{5}} & \sqrt{\frac{2}{5}} \cos \frac{6\pi}{5} & \sqrt{\frac{2}{5}} \sin \frac{6\pi}{5} & \sqrt{\frac{2}{5}} \cos \frac{12\pi}{5} & \sqrt{\frac{2}{5}} \sin \frac{12\pi}{5} \\ \sqrt{\frac{1}{5}} & \sqrt{\frac{2}{5}} \cos \frac{8\pi}{5} & \sqrt{\frac{2}{5}} \sin \frac{8\pi}{5} & \sqrt{\frac{2}{5}} \cos \frac{16\pi}{5} & \sqrt{\frac{2}{5}} \sin \frac{16\pi}{5} \end{bmatrix} \quad (12)$$

- 尝试绘制出这5个新的3d轨道, 你发现了什么?
- 仔细观察变换矩阵中各个元素的规律, 你能否把这种变换方式推广到其他的轨道上, 例如4f轨道或5g轨道?

(提示: 下面是2p轨道的另一种表现形式)

$$\begin{bmatrix} \Psi_{2p, new1} \\ \Psi_{2p, new2} \\ \Psi_{2p, new3} \end{bmatrix} = \begin{bmatrix} \sqrt{\frac{1}{3}} & \sqrt{\frac{2}{3}} & 0 \\ \sqrt{\frac{1}{3}} & \sqrt{\frac{2}{3}} \cos \frac{2\pi}{3} & \sqrt{\frac{2}{3}} \sin \frac{2\pi}{3} \\ \sqrt{\frac{1}{3}} & \sqrt{\frac{2}{3}} \cos \frac{4\pi}{3} & \sqrt{\frac{2}{3}} \sin \frac{4\pi}{3} \end{bmatrix} \begin{bmatrix} \Psi_{2p_z} \\ \Psi_{2p_x} \\ \Psi_{2p_y} \end{bmatrix} \quad (13)$$

## 附录A 关于 `marching_cubes` 坐标互换的说明

之前提到了 `marching_cubes` 的结果要做一次坐标互换, 似乎是 `scikit-image` 默认的坐标输出顺序有关.

验证是否需要互换/如何互换也比较简单, 用一个半主轴都不相等的椭球面即可. 例如下面这个椭球的半主轴长度分别为8, 2和5.

$$\frac{x^2}{8^2} + \frac{y^2}{2^2} + \frac{z^2}{5^2} = 1 \quad (14)$$

```
In [22]: def ellipsoid(x, y, z):
    return (x/8)**2 + (y/2)**2 + (z/5)**2
```

定义完之后, 就可以寻找等值面 `ellipsoid(x, y, z)=1`

```
In [23]: # 测试网格
test_limit = 10
test_n_points = 50
test_vec = np.linspace(-test_limit, test_limit, test_n_points)
Xt, Yt, Zt = np.meshgrid(test_vec, test_vec, test_vec)
test_fig, test_ax = new_fig_and_ax(plot_range=test_limit)

# 计算并处理椭球等值面 (调整spacing & 平移至中心)
test_step = 2 * test_limit / (test_n_points - 1)
test_verts, test_faces, _, _ = marching_cubes(
    ellipsoid(Xt, Yt, Zt),
    level = 1,
    spacing = (test_step, test_step, test_step),
)
test_verts -= test_limit

# 添加椭球等值面, 带一定透明度以便于观察
```

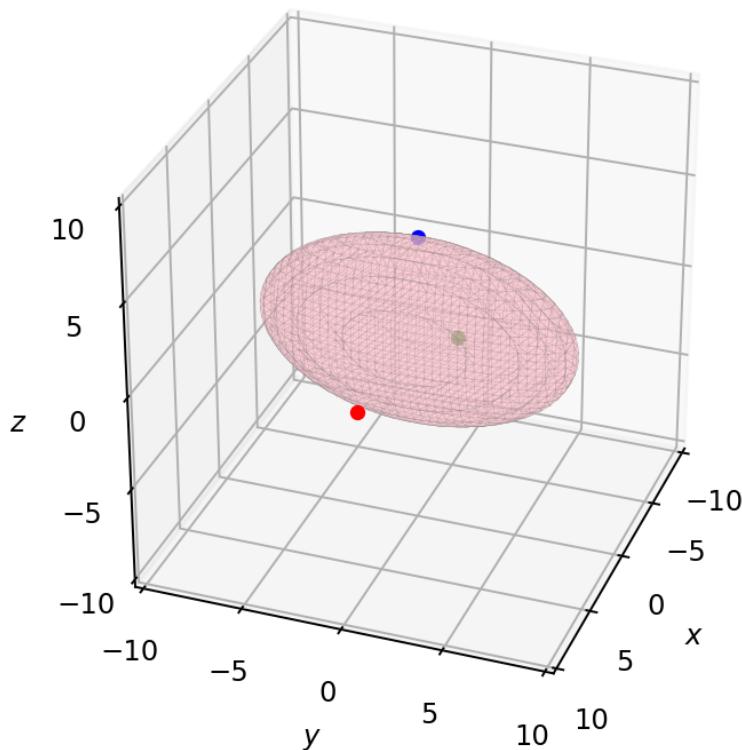
```

test_mesh = Poly3DCollection(test_verts[test_faces], lw=0.1, alpha=0.5)
test_mesh.set_facecolor("pink")
test_mesh.set_edgecolor("grey")
test_ax.add_collection3d(test_mesh)

# 添加(8, 0, 0)、(0, 2, 0)、(0, 0, 5)作为参考
# 这些点都应落在等值面(x/8)^2 + (y/2)^2 + (z/5)^2 = 1上
test_ax.scatter(8, 0, 0, color='red', s=20)
test_ax.scatter(0, 2, 0, color='green', s=20)
test_ax.scatter(0, 0, 5, color='blue', s=20)

plt.show()

```



三个点理应都落在上面, 然而只有 $z$ 轴的 $(0, 0, 5)$ 是正确的, 这是因为 $xOy$ 平面上的两个轴方向颠倒了.

解决这个问题很简单, 只需要互换 `test_verts` 的 $x$ 坐标和 $y$ 坐标即可.

In [24]:

```

# 互换所有顶点的x坐标和y坐标
test_verts[:, [0, 1]] = test_verts[:, [1, 0]]

# 重新绘图
new_test_fig, new_test_ax = new_fig_and_ax(plot_range=test_limit)
test_mesh = Poly3DCollection(test_verts[test_faces], lw=0.1, alpha=0.5)
test_mesh.set_facecolor("yellow")
test_mesh.set_edgecolor("grey")
new_test_ax.add_collection3d(test_mesh)
new_test_ax.scatter(8, 0, 0, color='red', s=20)
new_test_ax.scatter(0, 2, 0, color='green', s=20)
new_test_ax.scatter(0, 0, 5, color='blue', s=20)

# 可以发现这次三个点都落在等值面上, 问题解决
plt.show()

```

