1 Rotation

In [1]:
```python
import numpy as np
from scipy.linalg import expm
import open3d
import matplotlib.pyplot as plt
from tqdm import tqdm
vis = open3d.visualization.Visualizer()
vis.create_window(visible = False)
```

```
Jupyter environment detected. Enabling Open3D WebVisualizer.
[Open3D INFO] WebRTC GUI backend enabled.
[Open3D INFO] WebRTCWindowSystem: HTTP handshake server disabled.
```

Out[1]: True

In [2]:
```python
def hat(x):
    '''
    '''
    assert len(x) == 3, 'x must be a 3X1 vector'
    answer = np.zeros((3,3), dtype = np.float64)
    answer[0,1] = -x[2]
    answer[0,2] = x[1]
    answer[1,2] = -x[0]
    answer[1,0] = x[2]
    answer[2,0] = -x[1]
    answer[2,1] = x[0]
    return answer
```

In [3]:
```python
def quat2Rot(q):
    '''
    '''
    assert len(q) == 4, 'Quaternions are collection of 4 numbers'
    assert np.linalg.norm(q) - 1< 1e-3, 'Quaternions representing rotations should be unit norm'
    qs = q[0]
    qv = q[1:].reshape(-1,1)
    Eq = np.hstack([-qv, qs * np.eye(3) + hat(qv)])
    Gq = np.hstack([-qv, qs * np.eye(3) - hat(qv)])
    return Eq @ Gq.T
```

In [4]:
```python
def quat2AxisAngle(q):
    '''
    '''
    assert len(q) == 4, 'Quaternions are collection of 4 numbers'
    assert np.linalg.norm(q) - 1 < 1e-3, 'Quaternions representing rotations should be unit norm'
    qs,qv = q[0], q[1:]
    theta = 2 * np.arccos(qs)
    axis = qv / np.sin(theta/2) if theta != 0 else np.zeros((3,1))
    return axis,theta
```

In [5]:
```python
def AxisAngle2Rotation(axis, theta):
    '''
    '''
    assert len(axis) == 3, 'Axis must be 3d vector'
    axis_hat = hat(axis)
    return np.eye(3) + np.sin(theta) * axis_hat + (1 - np.cos(theta)) * (axis_hat @ axis_hat)
```

1.1

In [6]:
```python
p = np.array([1/np.sqrt(2), 1/np.sqrt(2), 0, 0])
q = np.array([1/np.sqrt(2),  0, 1/np.sqrt(2),0])
r = (p + q) / 2
print(f"norm of quaternion r : {np.linalg.norm(r)}")
```

```
norm of quaternion r : 0.8660254037844386
```

In [7]:
```python
r = r / np.linalg.norm(r)
print(f"new norm of r : {np.linalg.norm(r)}")
print(f"quaternion that is scalar multplication of and has unit norm : {r}")
```

```
new norm of r : 0.9999999999999999
quaternion that is scalar multplication of and has unit norm : [0.81649658 0.40824829 0.40824829 0.
]
```

In [8]:
```python
M_r = quat2Rot(r)
print(f"Rotation matrix corresponding to quaternion r : {M_r}")
```

```
Rotation matrix corresponding to quaternion r : [[ 0.66666667  0.33333333  0.66666667]
 [ 0.33333333  0.66666667 -0.66666667]
 [-0.66666667  0.66666667  0.33333333]]
```

In [9]:
```
axis_r, theta_r = quat2AxisAngle(r)
print(f'axis of rotation corresponding to r : {axis_r}')
print(f'angle of rotation corresponding to r : {theta_r} radians | {theta_r * 180 / np.pi} radians')
```

```
axis of rotation corresponding to r : [0.70710678 0.70710678 0.        ]
angle of rotation corresponding to r : 1.2309594173407747 radians | 70.52877936550931 radians
```

1.2

In [10]:
```
#Exponential coordinate = (axis) * angle corresponding to the rotation
axis_p, theta_p = quat2AxisAngle(p)
axis_q, theta_q = quat2AxisAngle(q)
print(f"Exponential coordinate of p : {axis_p * theta_p}")
print(f"Exponential coordinate of q : {axis_q * theta_q}")
```

```
Exponential coordinate of p : [1.57079633 0.         0.        ]
Exponential coordinate of q : [0.         1.57079633 0.        ]
```

1.3.a

In [11]:
```
axis_p
```

Out[11]:
```
array([1., 0., 0.])
```

In [12]:
```
theta_p
```

Out[12]:
```
1.5707963267948968
```

In [13]:
```
expm(hat(axis_p* theta_p))
```

Out[13]:
```
array([[ 1.,  0.,  0.],
       [ 0.,  0., -1.],
       [ 0.,  1.,  0.]])
```

In [14]:
```
expm(hat(axis_q * theta_q))
```

Out[14]:
```
array([[ 0.,  0.,  1.],
       [ 0.,  1.,  0.],
       [-1.,  0.,  0.]])
```

In [15]:
```
omega_p = axis_p * theta_p
omega_q = axis_q * theta_q
hat_omega_p = hat(omega_p)
hat_omega_q = hat(omega_q)
R_p = AxisAngle2Rotation(axis_p, theta_p)
R_q = AxisAngle2Rotation(axis_q, theta_q)

print(f"hat of omega_p of p : \n {hat_omega_p} \n ")
print(f"hat of omega_q of q : \n {hat_omega_q} \n")
print(f"Rotation matrix from omega_p : \n {R_p} \n")
print(f"Rotation matrix from omega_q : \n {R_q} \n")
```

```
hat of omega_p of p :
 [[ 0.         -0.          0.        ]
 [ 0.          0.         -1.57079633]
 [-0.          1.57079633  0.        ]]

hat of omega_q of q :
 [[ 0.         -0.          1.57079633]
 [ 0.          0.         -0.        ]
 [-1.57079633  0.          0.        ]]

Rotation matrix from omega_p :
 [[ 1.  0.  0.]
 [ 0.  0. -1.]
 [ 0.  1.  0.]]

Rotation matrix from omega_q :
 [[ 0.  0.  1.]
 [ 0.  1.  0.]
 [-1.  0.  0.]]
```

1.3.b

In [16]:
```
x1 = expm(hat_omega_p + hat_omega_q)
```

In [17]:
```
x2 = expm(hat_omega_p) @ expm(hat_omega_q)
```

In [18]:
```
print(f"exp([w1] + [w2]) == exp([w1]) exp([w2]) : {x1.all() == x2.all()}")
```

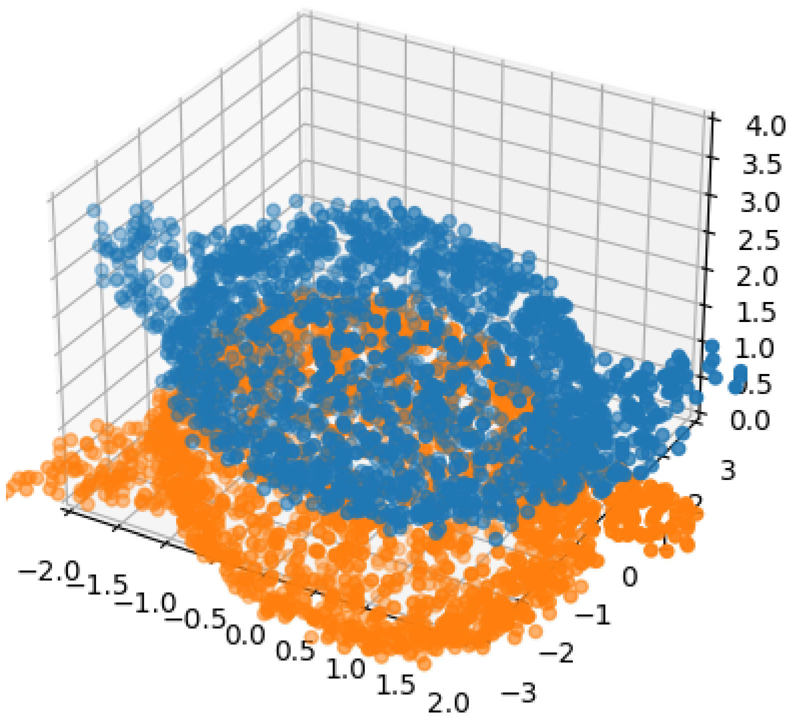
```
exp([w1] + [w2]) == exp([w1]) exp([w2]) : False
```

1.3.c

In [19]:
```
from mpl_toolkits.mplot3d import Axes3D
from mpl_toolkits import mplot3d
```

In [20]:
```
# Note Matplotlib is only suitable for simple 3D visualization.
# For later problems, you should not use Matplotlib to do the plotting

import numpy as np
import matplotlib.pyplot as plt
def show_points(points):
    fig = plt.figure()
    ax = fig.add_subplot(1,1,1,projection = '3d')
    ax.set_xlim3d([-5, 5])
    ax.set_ylim3d([-5, 5])
    ax.set_zlim3d([0, 4])
    ax.scatter(points[0], points[2], points[1])

def compare_points(points1, points2):
    fig = plt.figure()
    ax = fig.add_subplot(1,1,1,projection = '3d')
    ax.set_xlim3d([-2, 2])
    ax.set_ylim3d([-3, 3])
    ax.set_zlim3d([0, 4])
    ax.scatter(points1[0], points1[2], points1[1])
    ax.scatter(points2[0], points2[2], points2[1])
```

In [21]:
```
npz = np.load('HW1_P1.npz')
X = npz['X']
Y = npz['Y']
compare_points(X, Y)  # noisy teapotsand
```



In [22]:
```
print(np.linalg.norm(X - Y, ord= 'fro'))
```

```
165.61814811090522
```

In [23]:
```
from scipy.misc import derivative
```

In [24]:
```
def newton(f, x0, eps = 1e-4, iterations = 500):
    '''
    : Newtons iterative method to find root of a function
    : Newtons update : x -> x - f(x) / f'(x)
    : input --> function, previous estimate, max iterations, threshold t
    : output --> next estimate of x
    '''
    x = x0
    for iter in range(iterations):
        df = derivative(f, x, 1e-10)
        x_next = x - f(x)/df
        if np.linalg.norm(x_next - x) < 1e-9:
            print(f'optimal value at {x_next} found in {iter + 1} iterations')
            break
        x = x_next
    return x
```

In [25]:
```python
# copy-paste your hw0 solve module here
def hw0_solve(A, b, eps = 1e-6):
    x = np.zeros(A.shape[1])
    h = lambda l : np.linalg.inv(A.T @ A + 2 * l * np.eye(A.shape[1])) @ A.T @ b
    f = lambda l : h(l).T @ h(l) - eps #this is the function we want to
    l0 = 0 #starting point value of lambda
    l = newton(f, l0)
    x = h(l) #once we find desired lambda value, x = h(lambda)
    return x
```

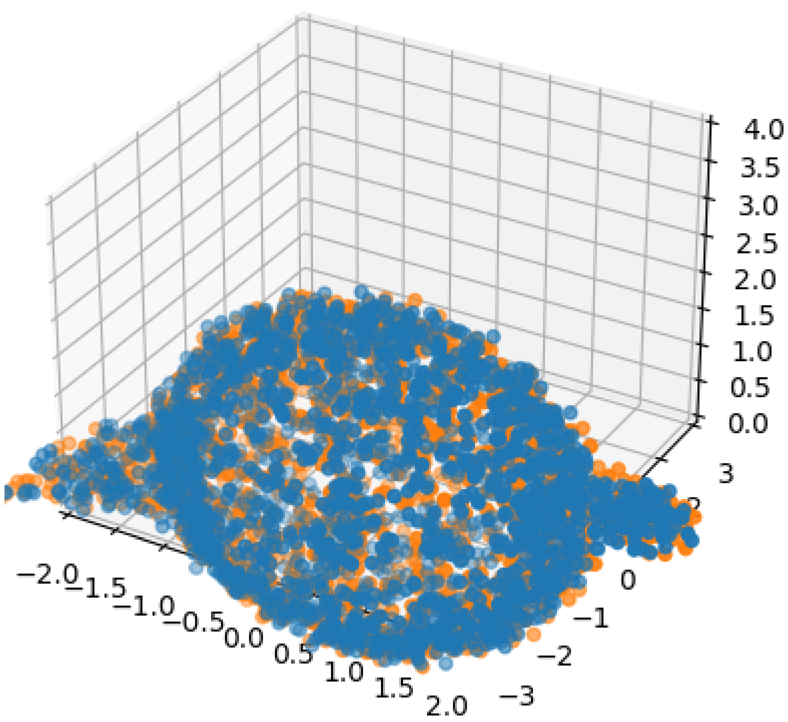
In [26]:
```python
def find_A(R,X = X):
    '''
    '''
    A = []
    for i in range(X.shape[1]):
        a = -R @ hat(X[:,i])
        A.append(a)
    return np.array(A).reshape(-1,3)

def find_B(R,X = X,Y = Y):
    '''
    '''
    B = []
    for i in range(X.shape[1]):
        b = Y[:,i] - R @ X[:,i]
        B.append(b)
    return np.array(B).reshape(-1,1)
```

In [27]:
```python
R1 = np.eye(3)
# solve this problem here, and store your final results in R1

for it in tqdm(range(100)):
    R2 = R1.copy()
    A = find_A(R1)
    B = find_B(R1)
    delta_omega = hw0_solve(A,B)
    R2 = R2 @ expm(hat(delta_omega))
    if np.allclose(R1, R2):
        print(f"Converged in {it + 1} iterations")
        break
    R1 = R2
```

```
 17%|█████████████████
| 17/100 [00:07<00:38,  2.18it/s]

Converged in 18 iterations
```

In [28]:
```python
# Testing code, you should see the points of the 2 teapots roughly overlap
compare_points(R1@X, Y)
R1.T@R1
print(np.linalg.norm(R1 @ X - Y, ord= 'fro'))
```

```
10.965105821981895
```



1.4.a

In [29]:
```
p1 = -p
q1 = -q
axis_p1, theta_p1 = quat2AxisAngle(p1)
axis_q1, theta_q1 = quat2AxisAngle(q1)
print(f"exponential coordinate of -p : {axis_p1 * theta_p1}")
print(f"exponential coordinate of p : {axis_p * theta_p}")
print(f"rotation matrix corresponding to p and -p equal : {AxisAngle2Rotation(axis_p1, theta_p1).all() == Ax
print("*" * 100)
print(f"exponential coordinate of -q : {axis_q1 * theta_q1}")
print(f"exponential coordinate of q : {axis_q * theta_q}")
print(f"rotation matrix corresponding to q and -q equal : {AxisAngle2Rotation(axis_q1, theta_q1).all() == Ax
```

```
exponential coordinate of -p : [-4.71238898 -0.         -0.        ]
exponential coordinate of p : [1.57079633 0.         0.        ]
rotation matrix corresponding to p and -p equal : True
****************************************************************************************************
exponential coordinate of -q : [-0.         -4.71238898 -0.        ]
exponential coordinate of q : [0.         1.57079633 0.        ]
rotation matrix corresponding to q and -q equal : True
```

We observe that the exponential coordinates of $(p, -p)$ and $(q, -q)$ are different but they correspond to the same rotation matrix. In fact, $R(q) = R(-q)$ holds for any valid rotation representing quaternion

$$E(q) = \begin{bmatrix} -q_v & q_s I + \hat{q}_v \end{bmatrix}$$
$$G(q) = \begin{bmatrix} -q_v & q_s I - \hat{q}_v \end{bmatrix}$$
$$\implies R(q) = E(q)G(q)^T = \begin{bmatrix} -q_v & q_s I + \hat{q}_v \end{bmatrix} \begin{bmatrix} -q_v^T \\ q_s I - \hat{q}_v \end{bmatrix} = q_v q_v^T + (q_s I + \hat{q}_v)(q_s I - \hat{q}_v)$$

Now for -q,
$$E(-q) = \begin{bmatrix} q_v & -q_s I - \hat{q}_v \end{bmatrix}$$
$$G(-q) = \begin{bmatrix} q_v & -q_s I + \hat{q}_v \end{bmatrix}$$
$$\implies R(-q) = E(-q)G(-q)^T = \begin{bmatrix} q_v & -q_s I - \hat{q}_v \end{bmatrix} \begin{bmatrix} q_v^T \\ -q_s I + \hat{q}_v \end{bmatrix} = q_v q_v^T + (q_s I + \hat{q}_v)(q_s I - \hat{q}_v)$$

Hence we see that $R(-q) = R(q)$

1.4.b

When designing a neural network to output quaternions, we cannot use L2 distance between ground truth quaternion and output quaternion because if $q_{out} = -q_{gt}$, then L2 distance would indicate a large error whereas in reality and geometrically, they both represent the same orientation and hence are actually close to each other.

2 Geometry

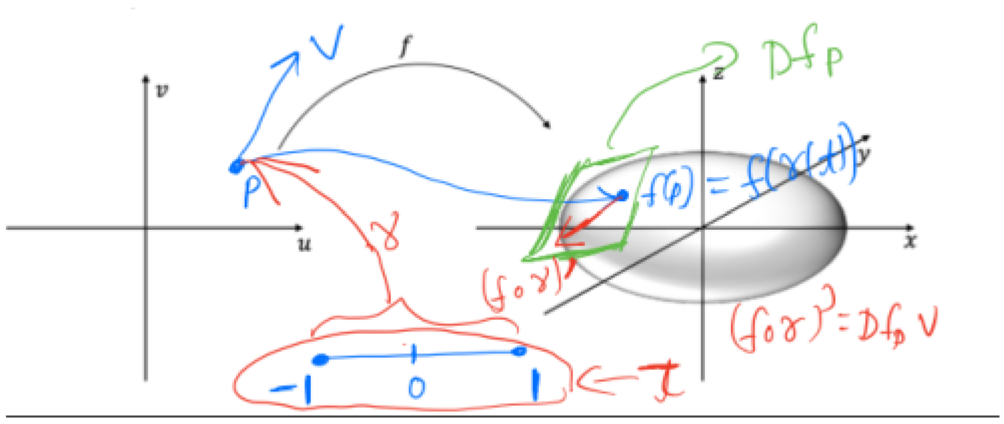
2.1 your solution here

In [30]:
```
import cv2
```

In [31]:
```
img = cv2.imread('./differential answer.png')
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
```

In [32]:
```
plt.imshow(img)
plt.axis('off')
```

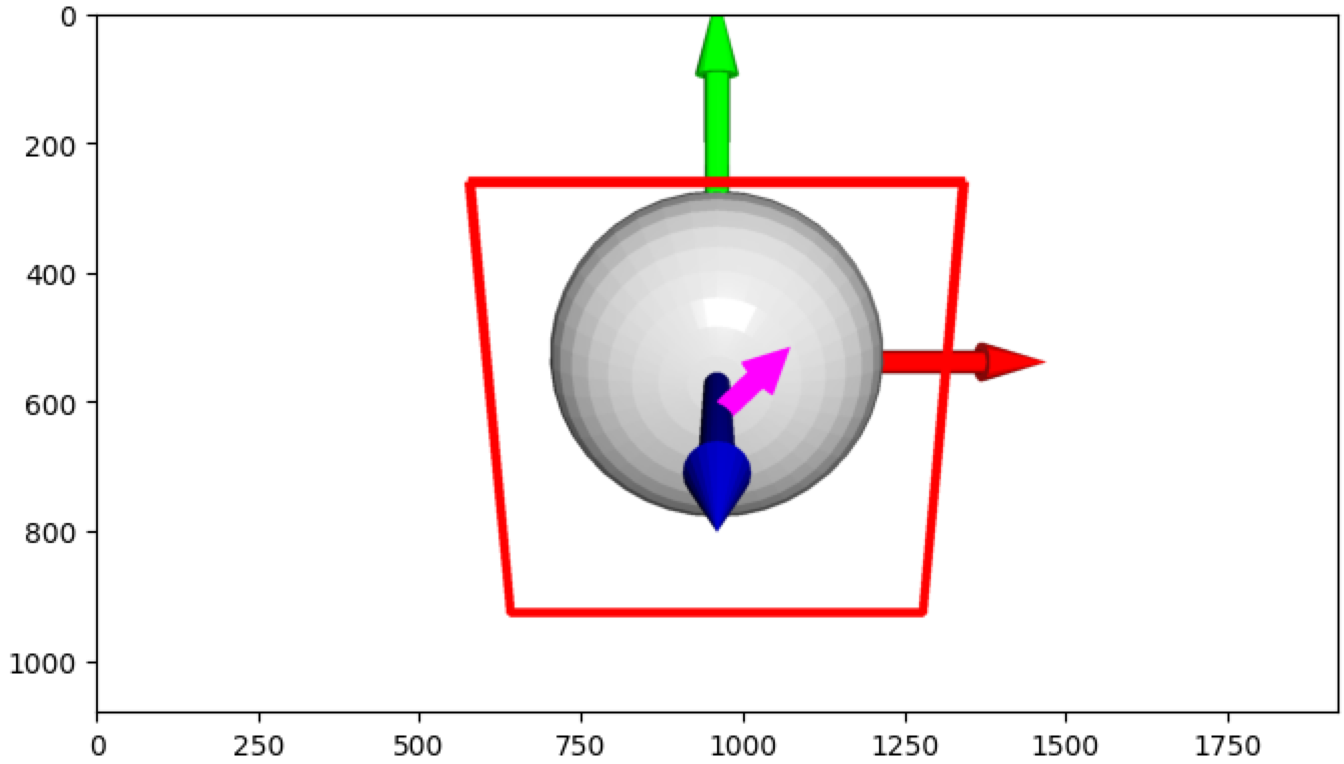
Out[32]:
```
(-0.5, 658.5, 272.5, -0.5)
```



In [33]:
```
a, b, c = 1, 1, 0.5
```

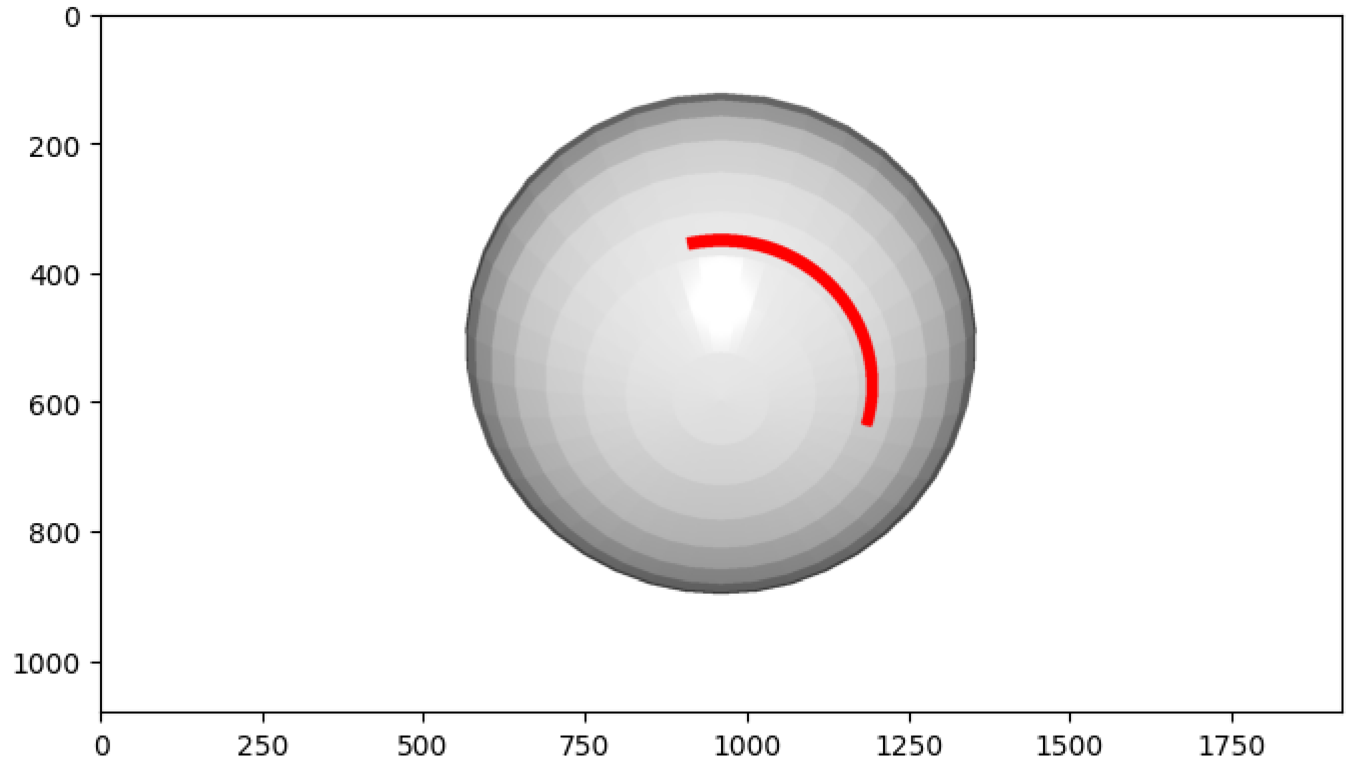
In [34]:
```
# These are some convenient functions to create open3d geometries and plot them
# The viewing direction is fine-tuned for this problem, you should not change them
def draw_geometries(geoms):
    for g in geoms:
        vis.add_geometry(g)
    view_ctl = vis.get_view_control()
    view_ctl.set_up((0, 1e-4, 1))
    view_ctl.set_front((0, 0.5, 2))
    view_ctl.set_lookat((0, 0, 0))
    # do not change this view point
    vis.update_renderer()
    img = vis.capture_screen_float_buffer(True)
    plt.figure(figsize=(8,6))
    plt.imshow(np.asarray(img)[::-1, ::-1])
    for g in geoms:
        vis.remove_geometry(g)

def create_arrow_from_vector(origin, vector):
    '''
    origin: origin of the arrow
    vector: direction of the arrow
    '''
    v = np.array(vector)
    v /= np.linalg.norm(v)
    z = np.array([0,0,1])
    angle = np.arccos(z@v)

    arrow = open3d.geometry.TriangleMesh.create_arrow(0.05, 0.1, 0.25, 0.2)
    arrow.paint_uniform_color([1,0,1])
    T = np.eye(4)
    T[:3, 3] = np.array(origin)
    T[:3,:3] = open3d.geometry.get_rotation_matrix_from_axis_angle(np.cross(z, v) * angle)
    arrow.transform(T)
    return arrow

def create_ellipsoid(a,b,c):
    sphere = open3d.geometry.TriangleMesh.create_sphere()
    sphere.transform(np.diag([a,b,c,1]))
    sphere.compute_vertex_normals()
    return sphere


def create_lines(points):
    lines = []
    for p1, p2 in zip(points[:-1], points[1:]):
        height = np.linalg.norm(p2-p1)
        center = (p1+p2) / 2
        d = p2-p1
        d /= np.linalg.norm(d)
        axis = np.cross(np.array([0,0,1]), d)
        axis /= np.linalg.norm(axis)
        angle = np.arccos(np.array([0,0,1]) @ d)
        R = open3d.geometry.get_rotation_matrix_from_axis_angle(axis * angle)

        T = np.eye(4)
        T[:3,:3]=R
        T[:3,3] = center
        cylinder = open3d.geometry.TriangleMesh.create_cylinder(0.02, height)
        cylinder.transform(T)
        cylinder.paint_uniform_color([1,0,0])
        lines.append(cylinder)
    return lines
```

In [35]:
```
# exapmle code to draw ellipsoid, curve, and arrows
arrow = create_arrow_from_vector([0.,0.,1.], [1.,1.,0.])
ellipsoid = create_ellipsoid(a, b, c)
cf = open3d.geometry.TriangleMesh.create_coordinate_frame()
cf.scale(2, (0,0,0))
curve = create_lines(np.array([[1,1,1], [-1,1,1], [-1,-1,1], [1,-1,1], [1,1,1]], dtype=np.float64))
draw_geometries([ellipsoid, cf, arrow] + curve)
```



2.2

In [36]:
```
points = []
x = lambda t : 0.5 * np.cos(t + np.pi/4)
y = lambda t : 0.5 * np.sin(t + np.pi/4)
z = lambda t : np.sqrt(3)/4

t = np.arange(-1,1,0.005)
x = x(t)
y = y(t)
z = z(t)
lines = []
for i in range(len(x)):
    lines.append([x[i], y[i], z])
lines = np.array(lines)
lines
curve = create_lines(lines)
draw_geometries([ellipsoid] + curve)
```



2.3.a

$Df_p = \begin{bmatrix} \frac{\partial f}{\partial u} & \frac{\partial f}{\partial v} \end{bmatrix}$ where $f(u,v) = \begin{bmatrix} acosusinv \\ bsinusinv \\ ccosv \end{bmatrix}$

$\implies Df_p = \begin{bmatrix} -asinusinv & acosucosv \\ bcosusinv & bsinucosv \\ 0 & -csinv \end{bmatrix}$

2.3.b Df_p maps the movement of a point $P \in R^2$ to the movement of the corresponding image of the point $f(p) \in R^3$ on the surface.

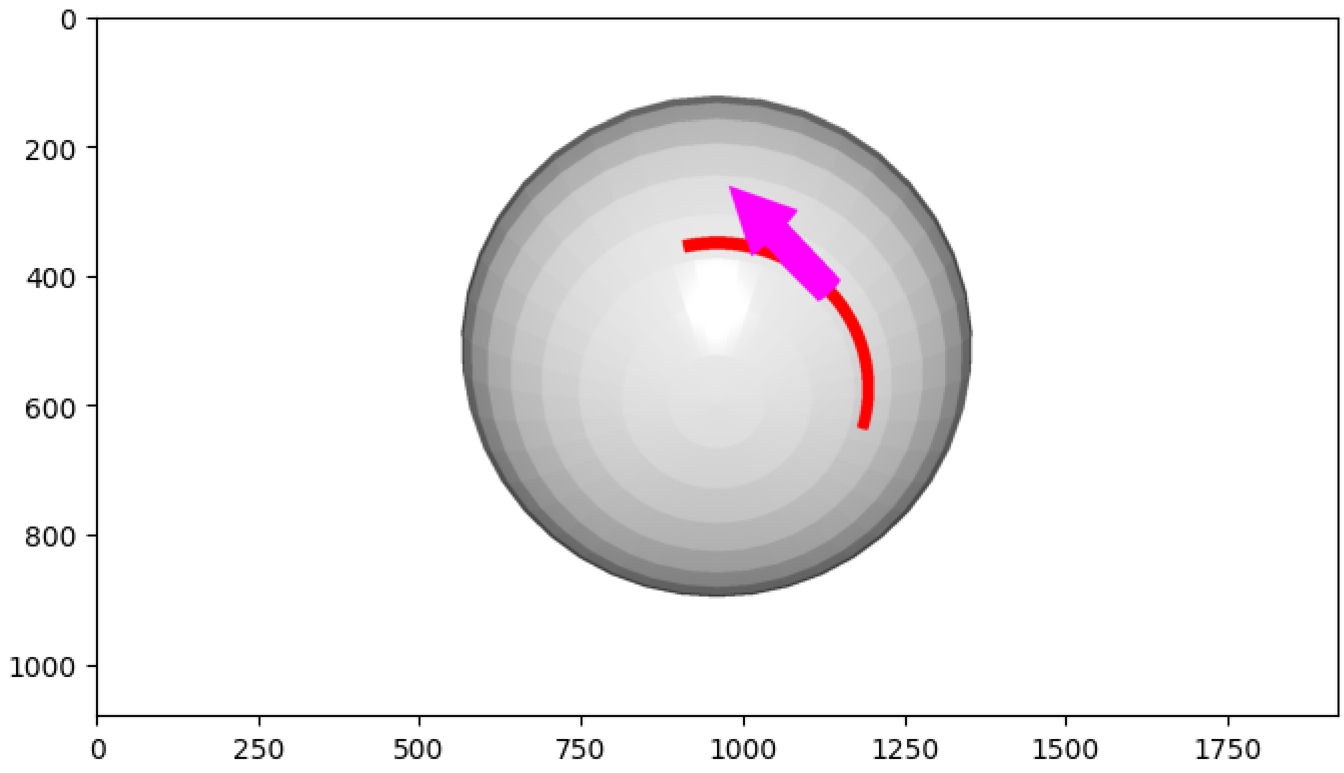
2.3.c

$p = (\frac{\pi}{4}, \frac{\pi}{6}), v = [1,0]^T$

$Df_p(v) = \begin{bmatrix} \frac{-1}{2\sqrt{2}} \\ \frac{1}{2\sqrt{2}} \\ 0 \end{bmatrix}$

This will be a vector originating at $f(p)$ which is $f(\frac{\pi}{4}, \frac{\pi}{6}) = \begin{bmatrix} \frac{1}{2\sqrt{2}} \\ \frac{1}{2\sqrt{2}} \\ \frac{\sqrt{3}}{4} \end{bmatrix}$

In [37]:
```
origin = np.array([1/(2 * np.sqrt(2)),1/(2 * np.sqrt(2)), np.sqrt(3)/4])
vector = np.array([-1/(2 * np.sqrt(2)),1/(2 * np.sqrt(2)), 0])
arrow = create_arrow_from_vector(origin = origin, vector= vector)
draw_geometries([ellipsoid, arrow]+ curve)
```



2.3.d

f_u, f_v is the basis of the tangent plane at p, hence the normal vector to the tangent plane at p can be described as $N_p = \frac{f_u \times f_v}{||f_u \times f_v||}$

In [38]:
```
pi = np.pi
f_u = np.array([-np.sin(pi/4) * np.sin(pi/6), np.cos(pi/4) * np.sin(pi/6), 0])
f_v = np.array([np.cos(pi/4) * np.cos(pi/6), np.sin(pi/4)*np.cos(pi/6), -0.5*np.sin(pi/6)])
N = np.cross(f_u, f_v)
N = N / np.linalg.norm(N)
print(f"Surface normal at p : {N}")
```

```
Surface normal at p : [-0.19611614 -0.19611614 -0.96076892]
```

So the normal vector to the tangent plane at $f(p)$ is given by $N_p = \begin{bmatrix} -0.19611 \\ -0.19611 \\ -0.9607 \end{bmatrix}$

2.3.e Given the tangent space basis f_u, f_v, we can convert this to orthogonal basis by Gram Scmidt Orthogonalisation.

Let $[\alpha_1, \alpha_2]$ be the orthogonal basis of the tangent plane. Let $\alpha_1 = f_u$. Now we need to find α_2 such that it is orthogonal to α_1. By Gram Schmidt Orthoganility theorem, suppose we have the original basis as $[\beta_1, \beta_2, \ldots . \beta_m]$. From this we want to construct the orthogonal basis $[\alpha_1, \alpha_2, \ldots . \alpha_m]$. We set $\alpha_1 = \beta_1$, and then the other vectors can be constructed as $\alpha_{m+1} = \beta_{m+1} - \sum_{k=1}^{m} \frac{<\beta_{m+1}, \alpha_k>}{||\alpha_k||^2} \alpha_k$

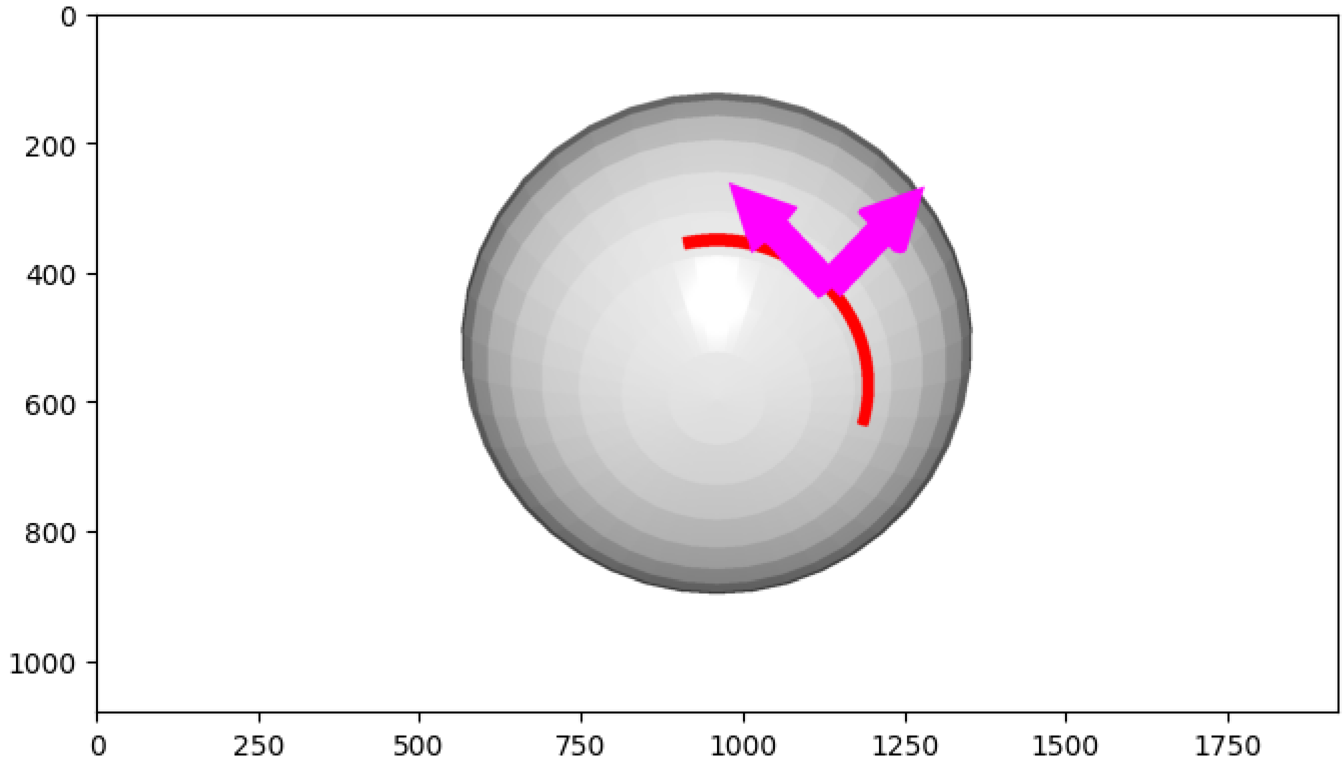
Using this, we set $\alpha_1 = f_u$, then $\alpha_2 = f_v - \frac{<f_v, \alpha_1>}{||\alpha_1||^2} \alpha_1$

In [39]:
```
alpha_2 = f_v - ((f_u.T @ f_v) / (np.linalg.norm(f_u))**2 ) * f_u
alpha_2
print(f"orthogonal basis : {[f_u , alpha_2]}")
```

```
orthogonal basis : [array([-0.35355339,  0.35355339,  0.        ]), array([ 0.61237244,  0.61237244, -0.25
])]
```

Realised later that $f_u^T f_v = 0$ which implies that we already had orthogonal basis. But we also get the same using gram schmidt transformation so atleat I did that correct

In [40]:
```
basis1 = create_arrow_from_vector(origin, f_u)
basis2 = create_arrow_from_vector(origin, f_v)
draw_geometries([ellipsoid, basis1, basis2] + curve)
```



2.4.a

We can find the arc length of the curve using the first fundamental form. $I_{p(t)} = Df^T_{p(t)} Df_{p(t)}$ Putting the general form of Df_p, and setting $v = [1, 0]^T$ we get $I_{p(t)}(X(t), X(t)) = sin^2(v)$

Now we have the parametrisation that $\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} t + \frac{\pi}{4} \\ \frac{\pi}{6} \end{bmatrix}$

$\implies v = \frac{\pi}{6}$ $\implies I_{p(t)}(v, v) = sin^2(\pi/6) = \frac{1}{4}$

Hence $s(t) = \int_0^t \sqrt{I_{p(t)}(v, v)} dt$

$\implies s(t) = \int_0^t \frac{1}{2} dt \implies s(t) = \frac{t}{2}$

Hence for the arc length parametrization, we substitute $t = 2s$ into $f(p(t))$

2.4.b

$$h_v(s) = f(s) = \begin{bmatrix} 0.5cos(2s + \frac{\pi}{4}) \\ 0.5sin(2s + \frac{\pi}{4}) \\ \frac{\sqrt{3}}{4} \end{bmatrix}$$

2.4.c

$$T(s) = \frac{df}{ds} = \begin{bmatrix} -sin(2s + \frac{\pi}{4}) \\ cos(2s + \frac{\pi}{4}) \\ 0 \end{bmatrix}$$

$$\frac{dN}{ds} = -kT(s) \implies N = -\kappa \begin{bmatrix} 0.5cos(2s + \frac{\pi}{4}) + c_1 \\ 0.5sin(2s + \frac{\pi}{4}) + c_2 \\ c_3 \end{bmatrix}$$

This above equations is satisfied if $c_1, c_2, c_3 = 0$ and $\kappa^2 = 4 \implies \kappa = 2$ Therefore

$$N(s) = \begin{bmatrix} -cos(2s + \frac{\pi}{4}) \\ -sin(2s + \frac{\pi}{4}) \\ 0 \end{bmatrix}$$

$$N(0) = \begin{bmatrix} -\frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} \\ 0 \end{bmatrix}$$

and normal to surface at point p is $N_p = \begin{bmatrix} -0.19611 \\ -0.19611 \\ -0.9607 \end{bmatrix}$ which is clearly different than the normal to the curve passing through point p

2.5.a

I used matlab to get the symblic expression of N and compute DN_p We obtain :

$$N_p = \begin{bmatrix} -0.5cosusin^2 v \\ -0.5sinusin^2 v \\ -sinvcosv \end{bmatrix} \frac{1}{\sqrt{0.3125 - 0.1875cos^2(2v) - 0.125cos(2v)}}$$

I am not writing the symbolic expression of DN because it is very huge. I use MATLAB to get the symbolic expression and substitute in the values.

2.5.b

Shape operator of a surface depends on the point p and is defined as the matrix $S \in R^{2\times 2}$ s.t. $DN_p = Df_p S$. Using MATLAB, DN_p at $(\frac{\pi}{4}, \frac{\pi}{6})$ is

$$DN_p = \begin{bmatrix} 0.196 & -0.42 \\ -0.196 & -0.42 \\ 0 & 0.1707 \end{bmatrix}$$

$$Df_p = \begin{bmatrix} -\frac{\sqrt{2}}{4} & \frac{\sqrt{6}}{4} \\ \frac{\sqrt{2}}{4} & \frac{\sqrt{6}}{4} \\ 0 & -\frac{1}{4} \end{bmatrix}$$

In [41]:
```
DNp = np.array([[0.196, -0.42 ], [-0.196, -0.42], [0, 0.1707]])
Dfp = np.array([[-np.sqrt(2)/4, np.sqrt(6)/4], [np.sqrt(2)/4, np.sqrt(6)/ 4], [0, -0.25]])

S = np.linalg.lstsq(Dfp, DNp, rcond = -1)[0]

print(f"Shape operator at p is : {S}")
```

```
Shape operator at p is : [[-0.55437172  0.        ]
 [ 0.         -0.68562196]]
```

Shape operator at p is hence $S = \begin{bmatrix} -0.554 & 0 \\ 0 & -0.686 \end{bmatrix}$

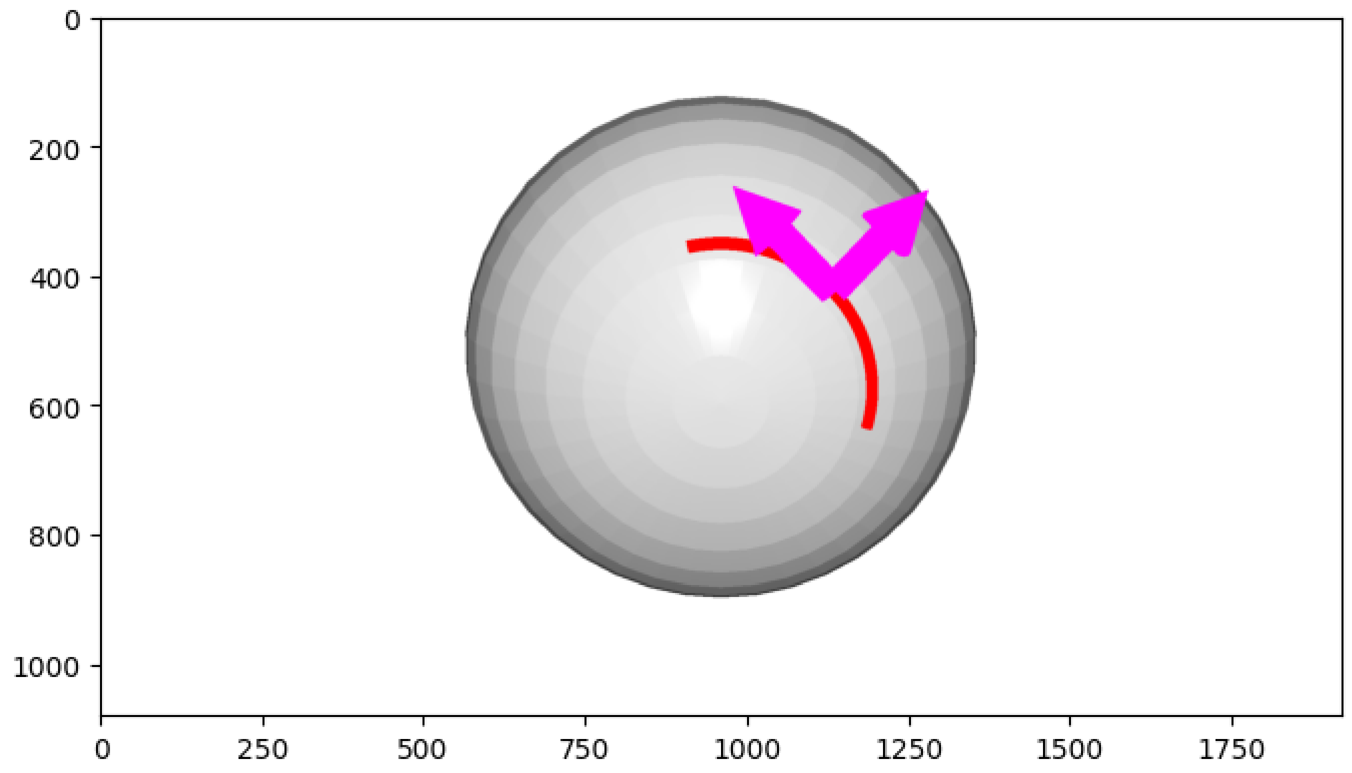
Since the matrix is diagonal, the eigen values are just the diagonal elements. Hence the eigen values of S are -0.554, -0.685 and the eigen vectors of S will be $\begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix}$

In [42]:
```
K, directions = np.linalg.eig(-S)
print(f"Principal curvature directions at p : {directions}")
print(f"Principal curvatures : {K[0], K[1]}")
```

```
Principal curvature directions at p : [[1. 0.]
 [0. 1.]]
Principal curvatures : (0.5543717164502532, 0.6856219642885754)
```

2.5.c

In [43]:
```
principal_direction1 = create_arrow_from_vector(origin,Dfp @ directions[:,0])
principal_direction2 = create_arrow_from_vector(origin ,Dfp @ directions[:,1])
draw_geometries([ellipsoid, principal_direction1, principal_direction2] + curve)
```



2.5.d

Principal directions are orthogonal in the tangent plane.

3 Mesh

3.1

Using Wolfram Alpha to integrate terms, we get $M_p = (\frac{3}{8}k_p^1 + \frac{1}{8}k_p^2)T_1T_1^T + (\frac{3}{8}k_p^2 + \frac{1}{8}k_p^1)T_2T_2^T$

Now since T_1, T_2 are the principal directions, they can be constructed as orthonormal vectors in the tangent plane at p. Assume a local frame of reference attached to the tangent plane at p, we can then assume $T_1 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$ and $T_2 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$

$$\implies M_p = \begin{bmatrix} \frac{3}{8}k_p^1 + \frac{1}{8}k_p^2 & 0 & 0 \\ 0 & \frac{3}{8}k_p^2 + \frac{1}{8}k_p^1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

We observe that M_p becomes a diagonal matrix. Therefore the eigen values of M_p will be equal to the diagonal entries. Hence we have $\lambda_1 = 0, \lambda_2 = \frac{3}{8}k_p^1 + \frac{1}{8}k_p^2, \lambda_3 = \frac{3}{8}k_p^2 + \frac{1}{8}k_p^1$

Corresponding to eigen value $\lambda_1 = 0$, we need to find the eigen vector, say $v = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}$ this has to satisfy

$M_p v = \lambda_1 v$
so v is the nullspace of M_p. Since we only have two pivot variables in M_p, the free variable becomes v_3 andv_1, v_2 are the pivot variables. To satisfy the equation, we have $v_1 = 0, v_2 = 0$ anbd v_3 can take on any value since it is a free variable.

$$\implies v = \begin{bmatrix} 0 \\ 0 \\ v_3 \end{bmatrix}$$

We can clearly observe that v_3 is perpendicular to both T_1, T_2 which implies it is orthogonal to the tangent plane at p. Hence we prove that one of the eigen vectors of M_p is the surface normal at point p

3.2

We saw above that besides $\lambda_1 = 0$, we have $\lambda_2 = \frac{3}{8}k_p^1 + \frac{1}{8}k_p^2, \lambda_3 = \frac{3}{8}k_p^2 + \frac{1}{8}k_p^1$. Corresponding eigen vectors are $v_1 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$ and $v_2 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$, which are the principal directions. Hence we have already proved the required statement in this problem.

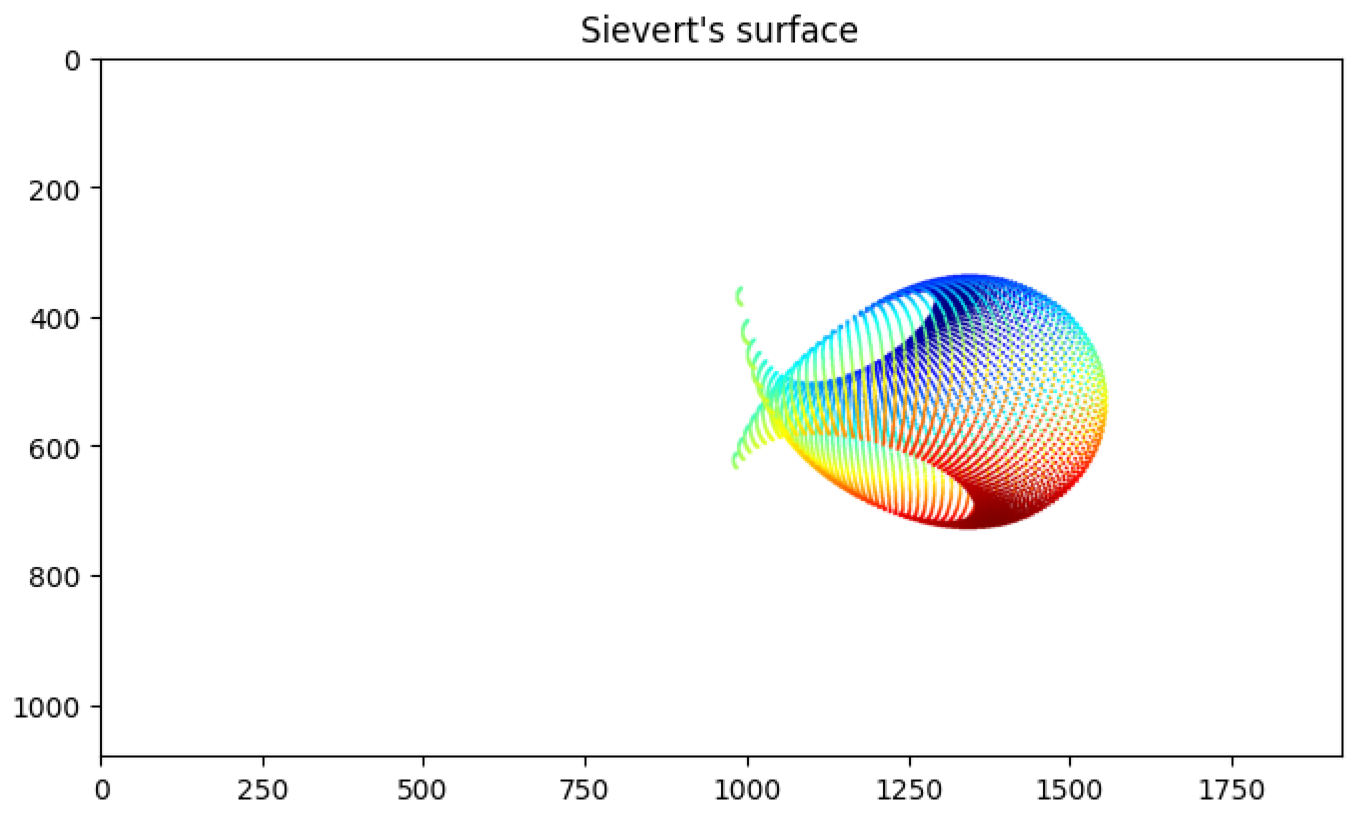
3.3

In [44]:
```
# You may want to restart your notebook here, to reinitialize Open3D

import open3d
vis = open3d.visualization.Visualizer()
vis.create_window(visible = False)

# Make sure you call this function to draw the points for proper viewing direction
def draw_geometries(geoms):
    for g in geoms:
        vis.add_geometry(g)
    view_ctl = vis.get_view_control()
    view_ctl.set_up((0, 1, 0))
    view_ctl.set_front((0, 2, 1))
    view_ctl.set_lookat((0, 0, 0))
    view_ctl.set_zoom(1)
    # do not change this view point
    vis.update_renderer()
    img = vis.capture_screen_float_buffer(True)
    plt.figure(figsize=(8,6))
    plt.imshow(np.asarray(img))
    for g in geoms:
        vis.remove_geometry(g)
```

In [45]:
```
import trimesh
sievert_mesh = trimesh.load('sievert.obj')
pcd = open3d.geometry.PointCloud()
pcd.points = open3d.utility.Vector3dVector(sievert_mesh.vertices)
draw_geometries([pcd])
plt.title("Sievert's surface")
```

```
WARNING - 2022-10-23 23:04:56,173 - graph - graph-tool unavailable, some operations will be much slower
WARNING - 2022-10-23 23:04:56,243 - assimp - pyassimp unavailable, using only native loaders
WARNING - 2022-10-23 23:04:56,246 - creation - shapely.geometry.Polygon not installed, some functions will not work!
```

Out[45]: Text(0.5, 1.0, "Sievert's surface")

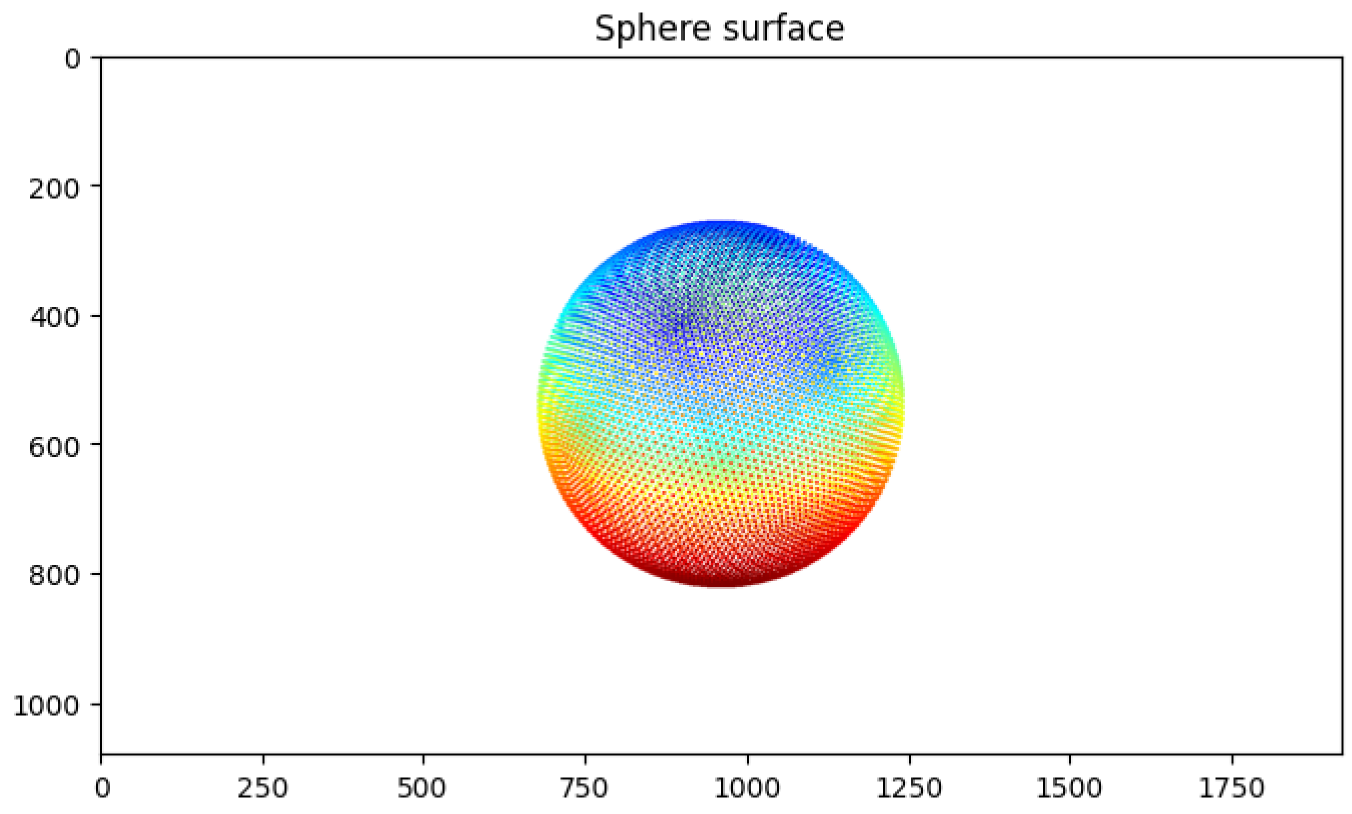


In [46]:
```
sievert_mesh.vertex_normals.shape
```

Out[46]: (10201, 3)

In [47]:
```
sphere_mesh = trimesh.load('icosphere.obj')
pcd = open3d.geometry.PointCloud()
pcd.points = open3d.utility.Vector3dVector(sphere_mesh.vertices)
draw_geometries([pcd])
plt.title('Sphere surface')
```

Out[47]: Text(0.5, 1.0, 'Sphere surface')



In [48]:
```
sphere_mesh
```

Out[48]: <trimesh.base.Trimesh at 0x7f8a15805a90>

In [49]:
```
sphere_mesh.face_normals
```

Out[49]:
```
array([[-0.01856648, -0.01121014, -0.99976478],
       [-0.01856643,  0.01147461, -0.99976178],
       [-0.00172356, -0.02161937, -0.99976479],
       ...,
       [-0.69204029, -0.3557269 , -0.62812309],
       [-0.90006126, -0.250008  , -0.35691137],
       [ 0.27529129, -0.90214694,  0.33218308]])
```

In [50]:
```
sphere_mesh.vertex_normals.shape
```

Out[50]: (10242, 3)

In [51]:
```
def normalize_edge_vector(vector):
    '''
    given a matrix containing the first edge of the triangle meshes
    normalize this edge because we want the first unit vector for Dfp to be this edge
    '''

    return vector / np.linalg.norm(vector, axis = 1, keepdims= True)
```

In [52]:
```
sphere_vertex_normals = trimesh.geometry.mean_vertex_normals(faces = sphere_mesh.faces, face_normals = spher
```

In [53]:
```python
def Rusinkiewicz(mesh, vertex_normals, triangle_normal):
    '''
    Compute the principal curvatures of each face of the mesh sing Rusinkiewicz method
    Construct three edges from vertex of each face
    e0 = mesh.vertices[mesh.faces[:,2]] - mesh.vertices[mesh.faces[:,1]]
    e1 = mesh.vertices[mesh.faces[:,2]] - mesh.vertices[mesh.faces[:,0]]
    e2 = mesh.vertices[mesh.faces[:,0]] - mesh.vertices[mesh.faces[:,1]]

    Normalize each vector in e0 as this will be our first vector for Dfp
    The second vector for our Dfp can be constructed using cross product of e0 and face normal

    Next we need the vertex normals
    Passed in vertex_normals

    Then solve a least square problem to find S using S[Df^T]e_i = Df^T (n_k - n_j)
    e_i is edge, n_k and n_i are the vertex normals at the other two vertices connected by e_i
    '''

    e0 = np.asarray(mesh.vertices[mesh.faces[:,2], :] - mesh.vertices[mesh.faces[:,1], :])
    e1 = np.asarray(mesh.vertices[mesh.faces[:,0], :] - mesh.vertices[mesh.faces[:,2], :])
    e2 = np.asarray(mesh.vertices[mesh.faces[:,1], :] - mesh.vertices[mesh.faces[:,0], :])

    e0_normalized  = normalize_edge_vector(e0)
#     e1 = normalize_edge_vector(e1)
    principal_curvatures = np.zeros((mesh.faces.shape[0], 2))
    T1 = np.zeros((mesh.faces.shape[0], 3))
    T2 = np.zeros_like(T1)

    for it in tqdm(range(mesh.faces.shape[0])):
        zeta_u = e0_normalized[it]
        normal = triangle_normal[it]
        zeta_v = np.cross(zeta_u, normal)
        zeta_v /= np.linalg.norm(zeta_v)
# #
# #         zeta_v = e1[it]
#         print(f"norm zeta_v : {np.linalg.norm(zeta_v)}")
#         zeta_v = e1[it] - e1[it].T @ e0_normalized[it] / (np.linalg.norm(e0_normalized[it])**2) * e0_norma
#         zeta_v = zeta_v / np.linalg.norm(zeta_v)
#         print(f"zeta_v norm : {np.linalg.norm(zeta_v)}")
#         print(f"zeta_v orthogonal to zeta_u : {zeta_v.T @ zeta_u}")
        Df_transpose = np.vstack([zeta_u, zeta_v])
        n0 = vertex_normals[mesh.faces[it, 0], :].reshape(-1,1)
        n1 = vertex_normals[mesh.faces[it, 1], :].reshape(-1,1)
        n2 = vertex_normals[mesh.faces[it, 2], :].reshape(-1,1)

        #S[Dfp^T]e_i = Dfp^T(n_j - n_k) ---> write as A[S11 ; S12; S21; S22] = b, A = 6 X 4, b = 6 X 1
        A = np.array([[zeta_u @ e0[it].T, zeta_v @ e0[it].T, 0,0],
                      [0,0,zeta_u @ e0[it].T, zeta_v @ e0[it].T],
                       [zeta_u @ e1[it].T, zeta_v @ e1[it].T, 0,0],
                      [0,0,zeta_u @ e1[it].T, zeta_v @ e1[it].T],
                       [zeta_u @ e2[it].T, zeta_v @ e2[it].T, 0,0],
                      [0,0,zeta_u @ e2[it].T, zeta_v @ e2[it].T]]
                      )
        b = np.vstack([Df_transpose @ (n2 - n1),Df_transpose @ (n0 - n2), Df_transpose @ (n1 - n0)])
        S = np.linalg.lstsq(A,b, rcond = None)[0]
        S = S.reshape(2,2)
        eig_values, eig_vectors = np.linalg.eig(S)
#         print(f"eigen values : {eig_values}")
        max_eig = np.argmax(eig_values)
        min_eig = np.argmin(eig_values)
        principal_curvatures[it,0] = np.abs(eig_values[max_eig])
        principal_curvatures[it,1] = np.abs(eig_values[min_eig])
        T1[it,:2] = eig_vectors[:,min_eig]
        T2[it,:2] = eig_vectors[:,max_eig]

    return principal_curvatures, T1, T2
```

In [54]:
```python
sphere_curvatures, sphere_T1, sphere_T2 =  Rusinkiewicz(sphere_mesh, sphere_vertex_normals, sphere_mesh.face
```

```
  0%|
| 0/20480 [00:00<?, ?it/s]/tmp/ipykernel_7399/2152285262.py:63: ComplexWarning: Casting complex values to r
eal discards the imaginary part
  T1[it,:2] = eig_vectors[:,min_eig]
/tmp/ipykernel_7399/2152285262.py:64: ComplexWarning: Casting complex values to real discards the imaginary
part
  T2[it,:2] = eig_vectors[:,max_eig]
100%|
| 20480/20480 [00:04<00:00, 4525.53it/s]
```

In [55]:
```python
sievert_vertex_normals = trimesh.geometry.mean_vertex_normals(faces = sievert_mesh.faces, face_normals = sie
```

In [56]:
```
sievert_curvatures, sievert_T1, sievert_T2 = Rusinkiewicz(sievert_mesh,sievert_vertex_normals, sievert_mesh.
```

```
100%|██████████████████████████████| 20000/20000 [00:04<00:00, 4605.55it/s]
```

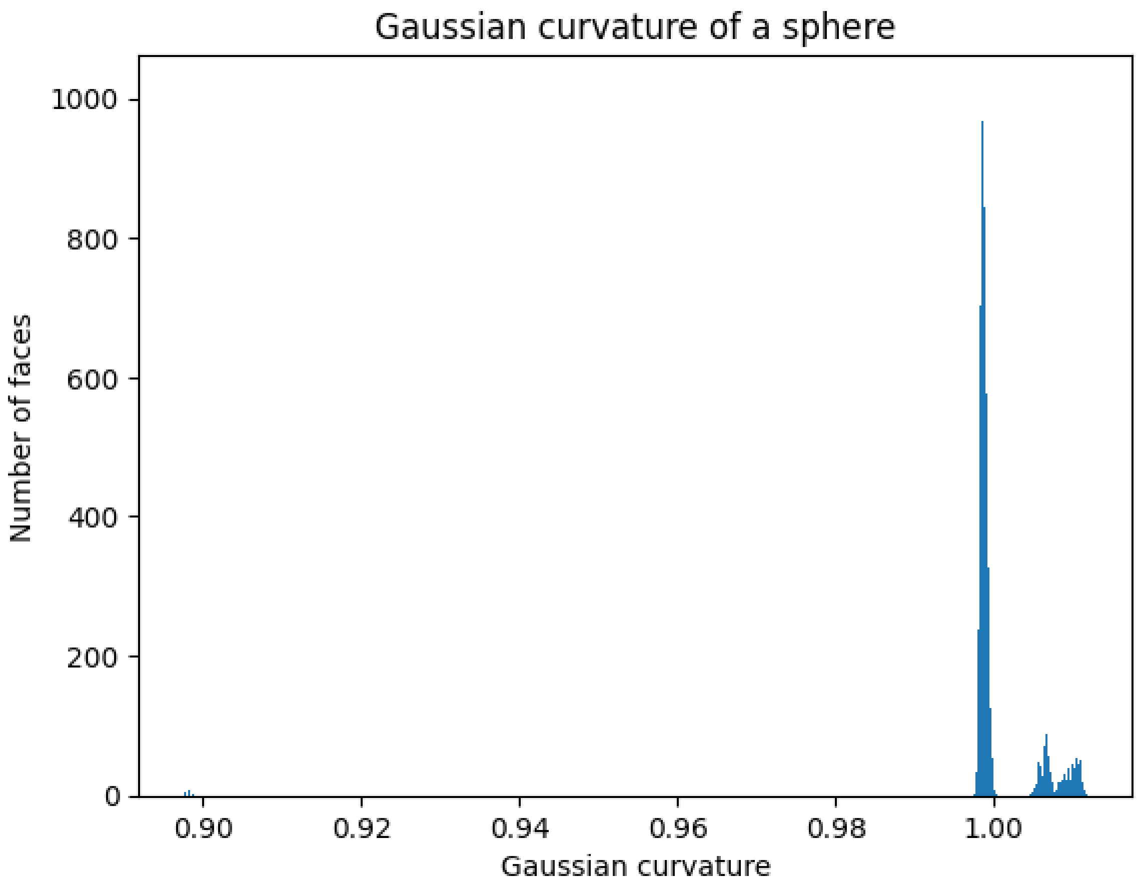
3.4

In [57]:
```
def gaussian_curvature(principal_curvatures):
    '''
    given the principal curvature value of each face
    find gaussian curvature = k1 * k2
    '''
    return principal_curvatures[:,+0] * principal_curvatures[:,1]

def mean_curvature(principal_curvatures):
    '''
    given the principal curvature value of each face
    find mean curvature = 1/2*(k1 + k2)
    '''
    return 0.5 * (principal_curvatures[:,0] + principal_curvatures[:,1])
```

In [58]:
```
sphere_gaussian_curvature = gaussian_curvature(sphere_curvatures)
sievert_gaussian_curvature = gaussian_curvature(sievert_curvatures)

sphere_mean_curvature = mean_curvature(sphere_curvatures)
sievert_mean_curvature = mean_curvature(sievert_curvatures)
```

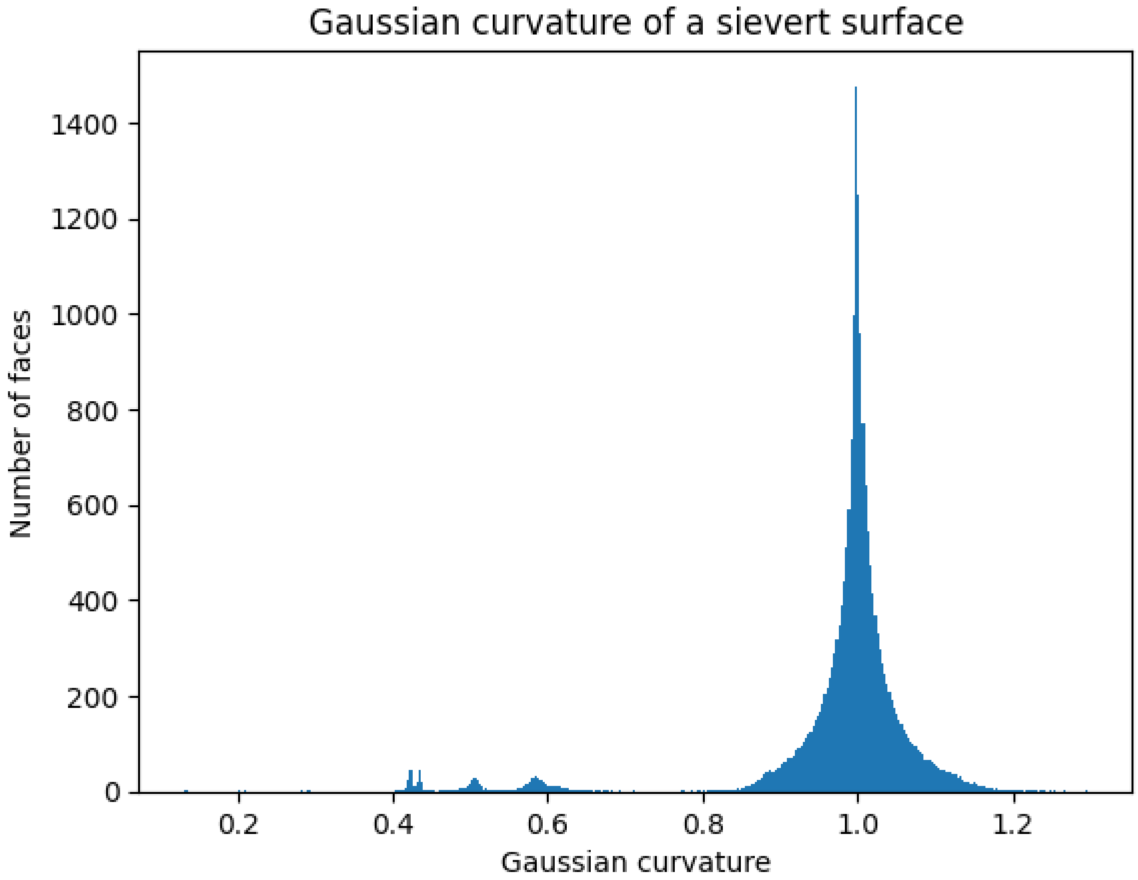
In [59]:
```
plt.hist(sphere_gaussian_curvature, bins = 'auto')
plt.xlabel('Gaussian curvature')
plt.ylabel('Number of faces')
plt.title('Gaussian curvature of a sphere')
```

Out[59]:
```
Text(0.5, 1.0, 'Gaussian curvature of a sphere')
```



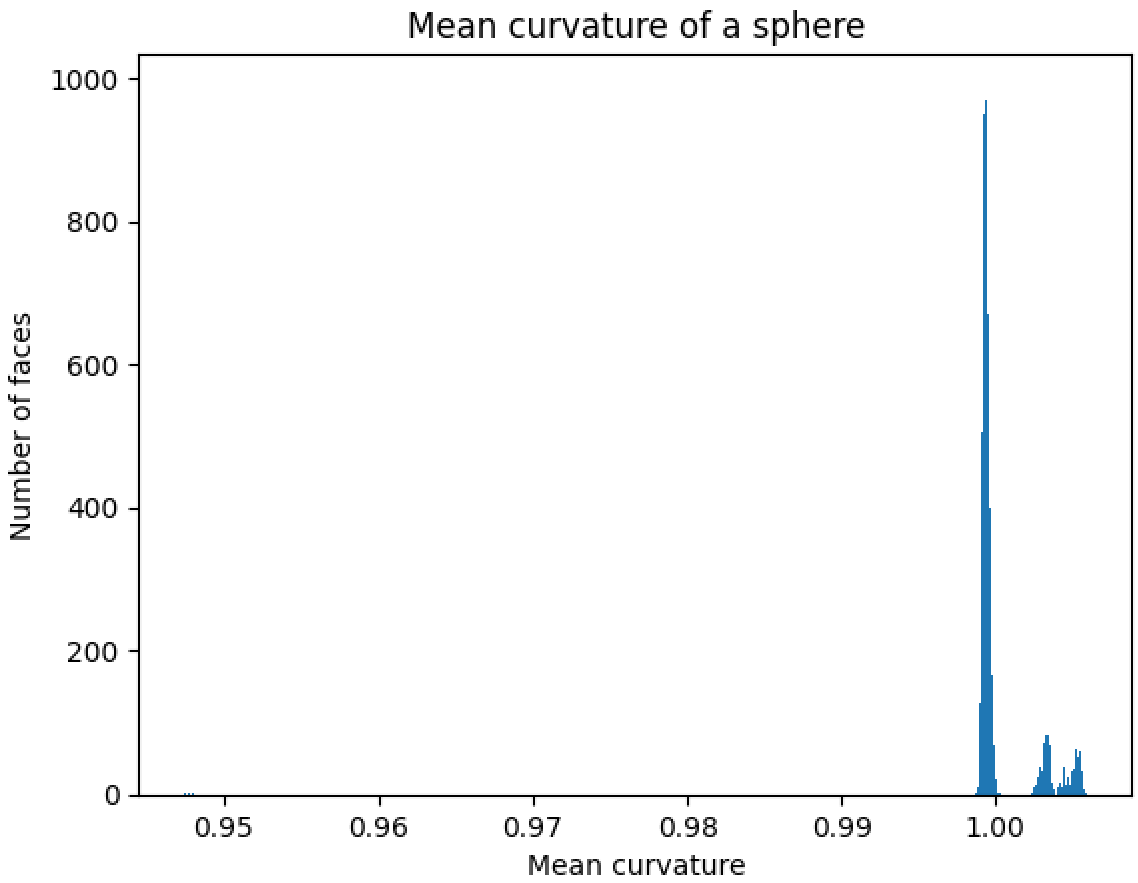
In [60]:
```
plt.hist(sievert_gaussian_curvature, bins = 'auto')
plt.xlabel('Gaussian curvature')
plt.ylabel('Number of faces')
plt.title('Gaussian curvature of a sievert surface')
```

Out[60]: Text(0.5, 1.0, 'Gaussian curvature of a sievert surface')



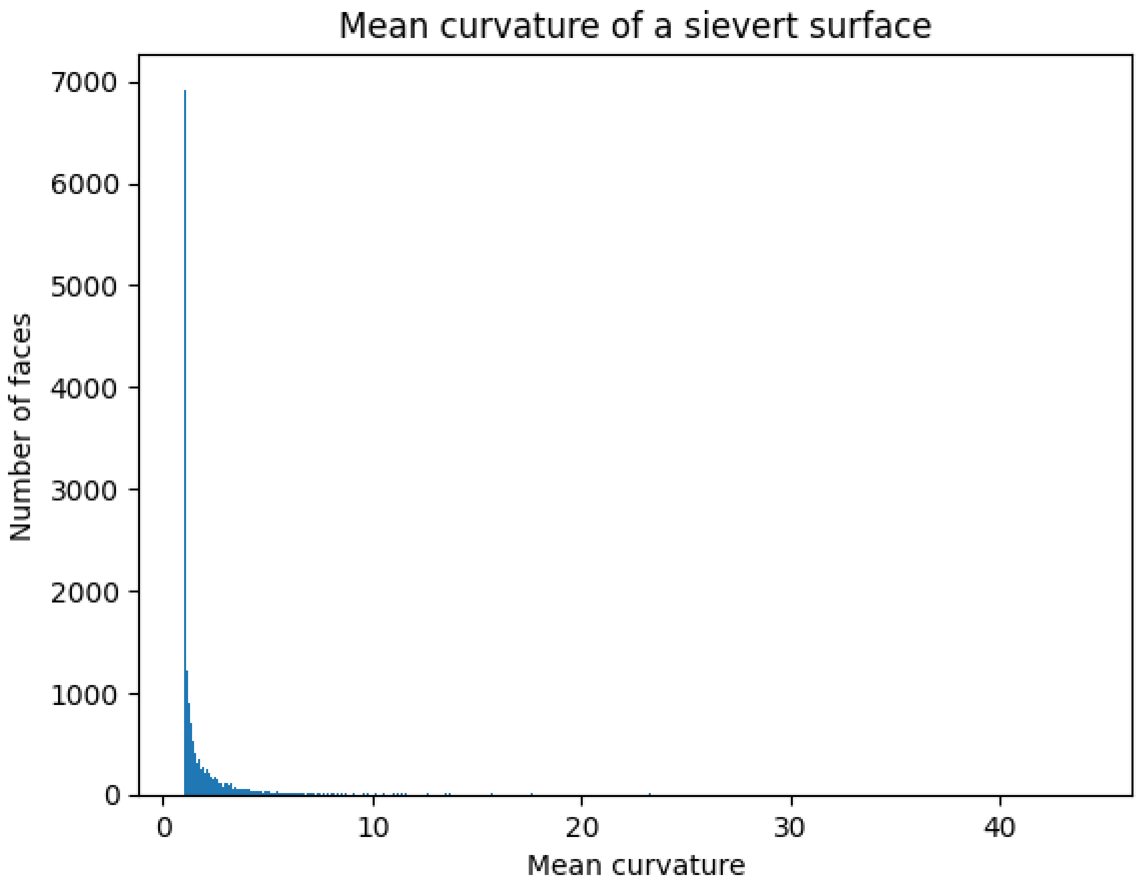
In [61]:
```
plt.hist(sphere_mean_curvature, bins = 'auto')
plt.xlabel('Mean curvature')
plt.ylabel('Number of faces')
plt.title('Mean curvature of a sphere')
```

Out[61]: Text(0.5, 1.0, 'Mean curvature of a sphere')



In [62]:
```
plt.hist(sievert_mean_curvature, bins = 'auto')
plt.xlabel('Mean curvature')
plt.ylabel('Number of faces')
plt.title('Mean curvature of a sievert surface')
```

Out[62]: Text(0.5, 1.0, 'Mean curvature of a sievert surface')



4 Point Cloud

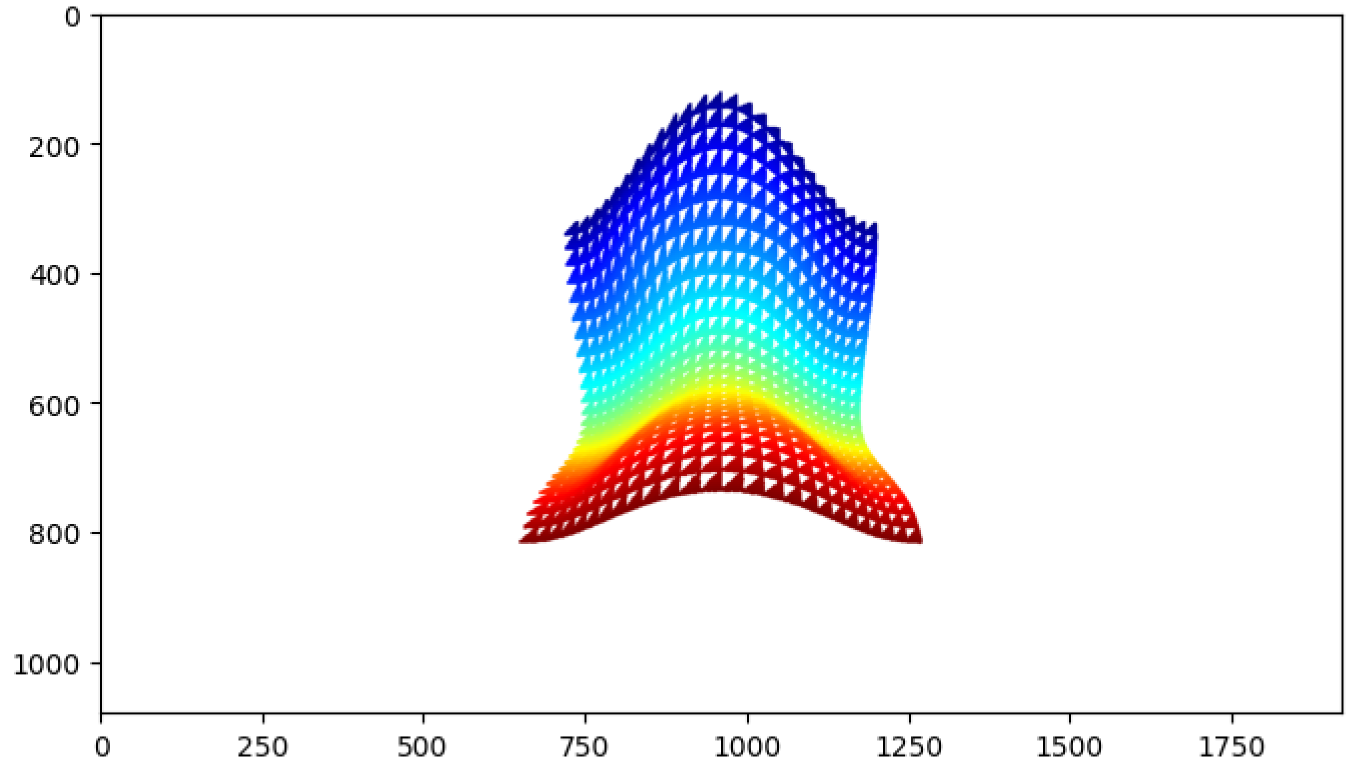
4.1

In [63]:
```
saddle_mesh = trimesh.load('./saddle.obj')
samples_large = trimesh.sample.sample_surface_even(saddle_mesh, count=100_000)
```

In [64]:
```
samples_large.shape
```

Out[64]: (104834, 3)

In [65]:
```
#visualize the generated uniform point cloud
point_cloud = open3d.geometry.PointCloud()
point_cloud.points = open3d.utility.Vector3dVector(np.array(samples_large))
draw_geometries([point_cloud])
```

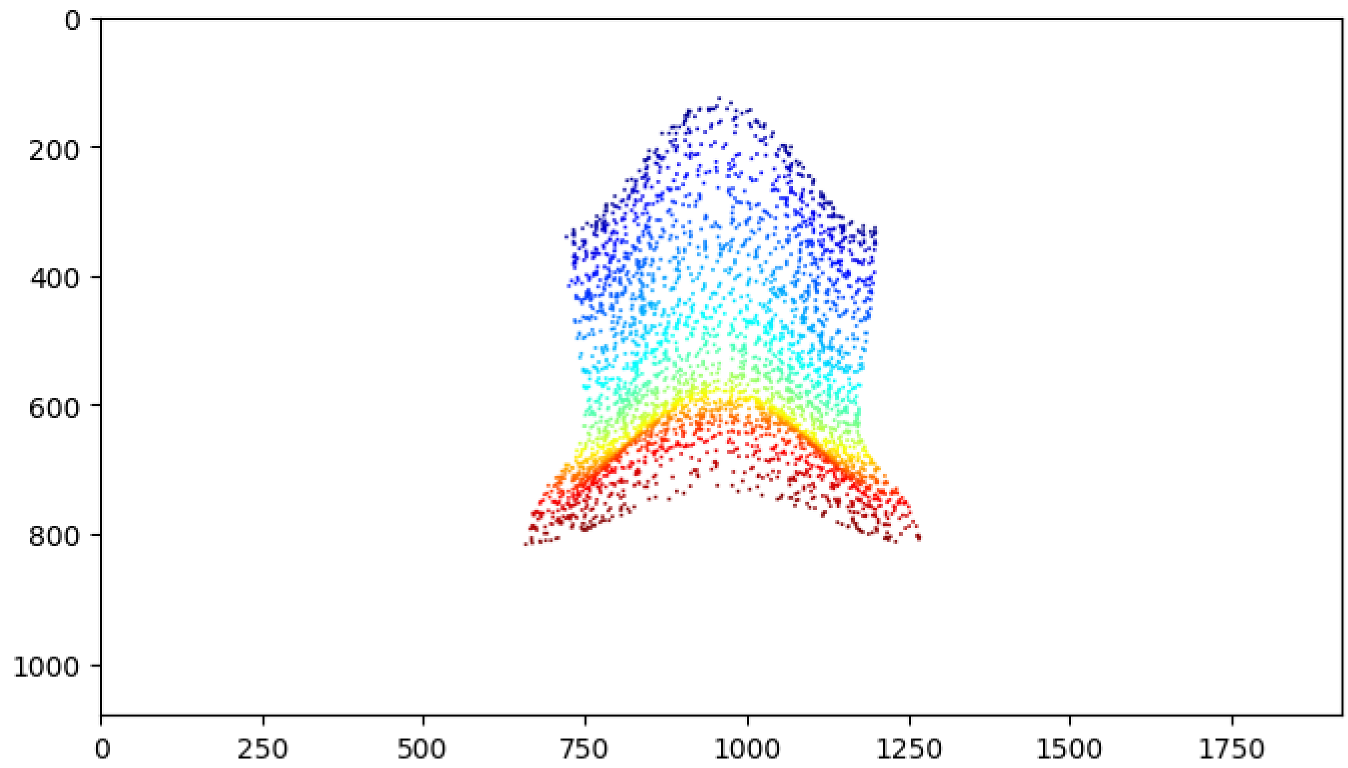


4.2

In [66]:
```
def iterative_furthest_sampling(original_pc, sample_points = 4000):
    '''
    '''
    point_index = np.arange(original_pc.shape[0], dtype = 'int')
    points_sampled = np.zeros(sample_points, dtype = 'int')
    dist_ij = np.full(original_pc.shape[0], float("inf"))

    #First select a random point from larger sample and put to smaller sample set
    point = np.random.choice(point_index, size = 1, replace = False)
    points_sampled[0]= point_index[point]

    #Remove the sampled point from original set
    point_index = np.delete(point_index, point)

    #Start of Iterative Furthest Point Sampling algorithm
    for it in tqdm(range(1,sample_points)):
        latest = points_sampled[it-1]
        temp = np.linalg.norm(original_pc[latest] - original_pc[point_index], axis = 1) #Distance of last po
        dist_ij[point_index] = np.minimum(temp, dist_ij[point_index])

        #Now we have distance from small set to larger set, select next sample point as one with largest dis
        new_sample = np.argmax(dist_ij[point_index])
        points_sampled[it] = new_sample
        point_index = np.delete(point_index, new_sample)

    return original_pc[points_sampled]
```

In [67]:
```
small_sample = iterative_furthest_sampling(np.array(samples_large))
point_cloud_small = open3d.geometry.PointCloud()
point_cloud_small.points = open3d.utility.Vector3dVector(small_sample)
draw_geometries([point_cloud_small])
```

```
100%|| 3999/3999 [00:18<00:00, 219.80it/s]
```



4.3

In [68]:
```
def nearest_neighbours(point, original_set = np.array(samples_large), num_neighbours = 50):
    '''
    '''
    dist = np.linalg.norm(point - original_set, axis = 1)
    indices = np.argsort(dist)[:num_neighbours + 1]
    return original_set[indices]
```

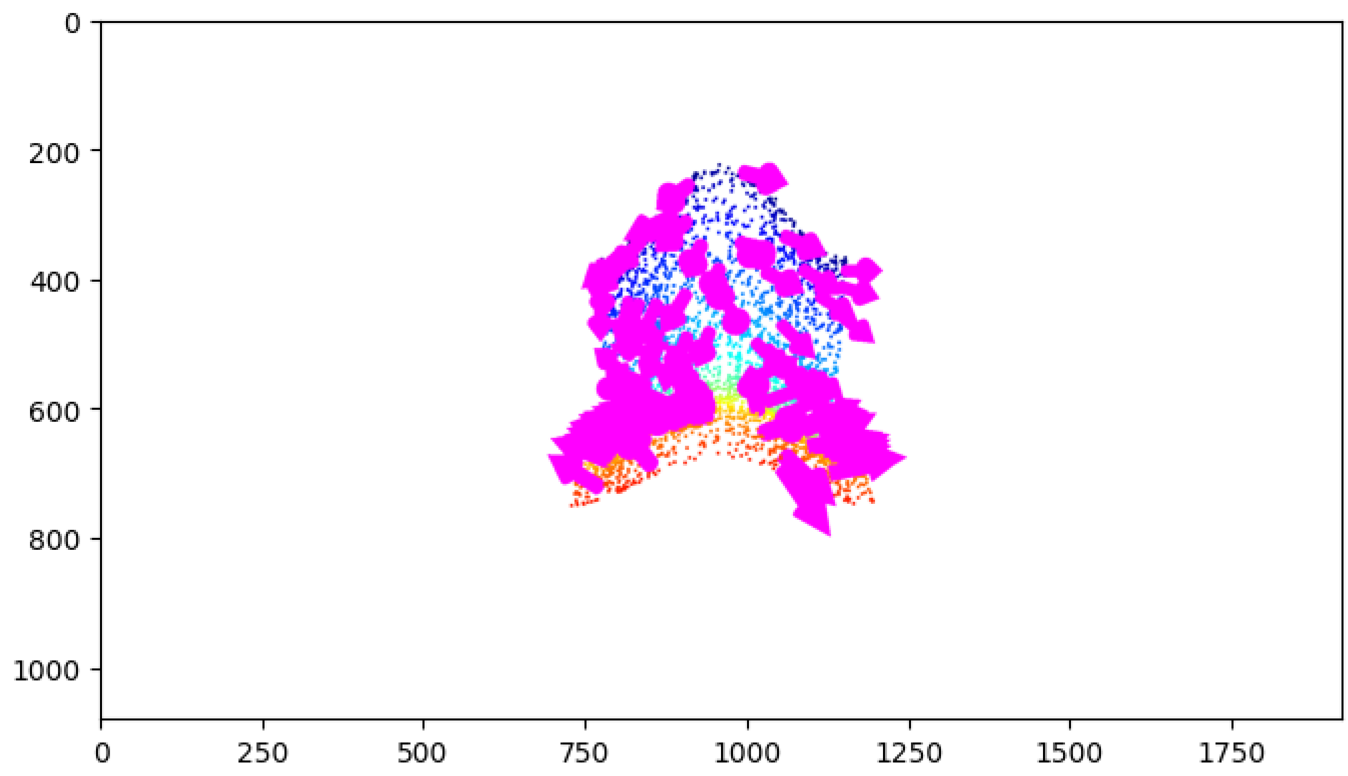
In [69]:
```
point_cloud_normals = np.zeros((4000, 3))
origin = np.zeros_like(point_cloud_normals)
for i in tqdm(range(4000)):
    point = small_sample[i]
    origin[i] = point
    neighbours = nearest_neighbours(point).T
    mu = np.mean(neighbours)
    M = (neighbours - mu) @ (neighbours - mu).T
    eig_values, eig_vectors = np.linalg.eig(M)
    min_eig_index = np.argmin(eig_values)
    normal = eig_vectors[:,min_eig_index]
    if normal[1] < 0:
        normal = -normal
    point_cloud_normals[i,:] = normal
```

```
100%|| 4000/4000 [00:44<00:00, 88.96it/s]
```

In [70]:
```
arrows = [create_arrow_from_vector(origin[i], point_cloud_normals[i]) for i in range(4000)]
```

In [71]:
```python
geometries = [point_cloud_small]

for i in tqdm(range(0,4000,40)):
    geometries.append(arrows[i])

draw_geometries(geometries)
```

```
100%|██████████████████████████████████████████████████| 100/100 [00:00<00:00, 657414.42it/s]
```



4.4 your solution here

5

5.1

Hours spent : 30-35 hrs

5.2

Hours spent on course : 10-15 hrs (reviewing lecture videos, some revision of older slides, writing notes)

5.3

It would be great if for future assignments, we could be provided with a conda yaml file for creating an environemnt with all necessary packages. Otherwise sometimes it becomes a pain to manually install all the packages and resolve dependencies.

Thank you

In [ ]: