```
"""
Python script to generate 3D models of some surfaces using the methods of [1]

  [1]: Hanson, Andrew J.
       "A Construction for Computer Visualization of Certain Complex Curves."
       (1994).
       https://homes.luddy.indiana.edu/hansona/papers/CP2-94.pdf
"""

import numpy as np
import numpy.linalg as la
from typing import List, Tuple, Iterator, Callable

Vector3     = np.ndarray[Tuple[3], np.dtype[np.float64]]
Patch       = np.ndarray[Tuple[3, int, int], np.dtype[np.float64]]
ComplexGrid = np.ndarray[Tuple[int, int], np.dtype[np.complex128]]
Projection  = Callable[[ComplexGrid, ComplexGrid], Patch]

class Model:
    def __init__(self, p: Projection):
        self.patches: List[Patch] = []
        self.projection = p

    def append_patch(self, z1: ComplexGrid, z2: ComplexGrid):
        """
        Appends a 2D array of points in C^2 to the model
        and updates the bounding box
        """
        self.patches.append(self.projection(z1, z2))

    def vertices(self) -> Iterator[Vector3]:
        """
        Iterate through the vertices of the mesh
        """

        return (x[:, j, i] for x in self.patches
                for i in range(x.shape[2]) for j in range(x.shape[1]))

    def write_obj(self, path: str):
        """
        Exports the model to the an OBJ file at `path`
        """
        with open(path, "w") as f:
            vs = set()       # vertices
            eff_vis = list() # effective indixes

            f.write("# vertices\n")

            vi = 0 # OBJ indices start at 1
            for x1, x2, x3 in self.vertices():
                v = f"{x1:.3f} {x2:.3f} {x3:.3f}"
                eff_vis.append(vi)

                if v not in vs:
                    vs.add(v)
                    vi += 1

                    f.write(f"v {v}\n")

            f.write("\n# faces\n")
```

```
            v_i = 0
            fs = set() # faces
            for patch in self.patches:
                _, cols, rows = patch.shape

                for theta_i in range(rows - 1):
                    for xi_i in range(cols - 1):
                        v1 = eff_vis[v_i + xi_i + theta_i * cols]
                        v2 = eff_vis[v_i + xi_i + (theta_i + 1) * cols]
                        v3 = eff_vis[v_i + (xi_i + 1) + theta_i * cols]
                        v4 = eff_vis[v_i + (xi_i + 1) + (theta_i + 1) * cols]

                        if (v1, v2, v3) not in fs:
                            fs.add((v1, v2, v3))
                            f.write(f"f {v1} {v2} {v3}\n")
                        if (v2, v4, v3) not in fs:
                            fs.add((v1, v2, v3))
                            f.write(f"f {v2} {v4} {v3}\n")

                v_i += rows * cols

def normalized_p(z1: ComplexGrid, z2: ComplexGrid) -> Patch:
    "Projection R^4 -> S^3 + inclusion R^3 -> S^3"
    assert z1.shape == z2.shape

    z = np.stack((z1, z2), axis=0)
    w = z/la.norm(z, axis=0)
    v = np.vstack([w.real, w.imag])
    x = v[:3, :, :]/(1 - v[3, :, :])

    return x

def riemman_sphere_p(z1: ComplexGrid, z2: ComplexGrid) -> Patch:
    "Riemman-sphere transform + projection on the first 3 coordinates"
    assert z1.shape == z2.shape

    norm_sqrd = z1.real ** 2 + z1.imag ** 2 + z2.real ** 2 + z2.imag ** 2

    u0 = 2 * (norm_sqrd/4)
    u1 = z1.real
    u2 = z1.imag

    return np.vstack([u0, u1, u2])/(1 + norm_sqrd/4)

def s(k: int, n: int) -> np.complex128:
    return np.exp(2 * np.pi * 1j * k / n)

EPS = 0.1
XI_MAX = 15

def fermat_surface(n: int, eps: float = EPS, xi_max: float = XI_MAX) -> Model:
    """
    Returns a model of the Fermat surface z1^n + z2^n = 1
    """
    if n <= 0:
        raise ValueError(f"Parameter \"n\" should be positive: n = {n}")

    m = Model(normalized_p)

    theta, xi = np.meshgrid(np.arange(0, np.pi / 2 + eps, eps),
                            np.arange(-xi_max, xi_max + eps, eps))
```

```
    for k1 in range(n):
        for k2 in range(n):
            u1 = (np.exp(xi + 1j * theta)+np.exp(-xi - 1j * theta))/2
            z1 = s(k1, n) * (u1**(2/n))
            u2 = (np.exp(xi + 1j * theta)-np.exp(-xi - 1j * theta))/2j
            z2 = s(k2, n) * (u2**(2/n))
            z = np.stack((z1, z2), axis=0)

            m.append_patch(z)

    return m

def brieskorn_surface(z3: complex, n: int,
                      patches: Iterator[Tuple[int, int]] = None,
                      eps: float = EPS, xi_max: float = XI_MAX) -> Model:
    """
    Returns the model of the Brieskorn surface z1^(6 n - 1) + z2^3 + z3^2 = 0
    """
    if not (1 <= n <= 11):
        raise ValueError(f"\"n\" should be in between 1 and 11: n = {n}")

    if abs(z3) > 1:
        raise ValueError(f"\"z3\" should have norm smaller than 1: z3 = {z3}")

    k0 = 6*n - 1

    if patches is None:
        patches = ((k1, k2) for k1 in range(3) for k2 in range(k0))

    m = Model(normalized_p)

    theta, xi = np.meshgrid(np.arange(0, np.pi / 2 + eps, eps),
                            np.arange(-xi_max, xi_max + eps, eps))

    for k1, k2 in patches:
        if not (0 <= k1 < 3) or not (0 <= k2 < k0): continue

        u1 = (np.exp(xi + 1j * theta)-np.exp(-xi - 1j * theta))/2j
        z1 = -(z3**(2/k0)) * s(k2, k0) * (u2**(2/k0))
        u2 = (np.exp(xi + 1j * theta)+np.exp(-xi - 1j * theta))/2
        z2 = -(z3**(2/3)) * s(k1, 3) * (u1**(2/3))

        m.append_patch(z2, z1)

    return m
```