```python
import random
import numpy as np
import matplotlib.pyplot as plt


def norm(vector):
    return np.sqrt(np.sum(np.square(vector)))


def custom_argsort(arr):
    return sorted(range(len(arr)), key=lambda x: arr[x])


def generate_points(n, r_min, r_max, theta_min, theta_max):
    points = []
    for _ in range(n):
        r = random.uniform(r_min, r_max)
        theta = random.uniform(theta_min, theta_max)
        x = r * np.cos(np.radians(theta))
        y = r * np.sin(np.radians(theta))
        distance = np.sqrt(x**2 + y**2)
        points.append((x, y, distance))
    return points


def find_closest_point(points, P_0, closest_points):
    points_array = np.array([point[:2] for point in points])
    P_0_array = np.array(P_0)

    distances = np.array([norm(point - P_0_array) for point in points_array])
    for point in closest_points:
        if point in points:
            index = points.index(point)
            distances[index] = float("inf")

    closest_point_index = np.argmin(distances)
    return closest_point_index, points[closest_point_index]


def find_unit_vector(P_from, P_to):
    vector = np.array(P_to[:2]) - np.array(P_from[:2])
    vector_norm = norm(vector)
    if vector_norm == 0:
        return np.array([0, 0])
    return vector / vector_norm


def filter_points_by_dot_product(points, base_point, reference_vector):
    remaining_points = []
    for point in points:
        if np.array_equal(point[:2], base_point[:2]):
            continue
        unit_vector = find_unit_vector(base_point, point[:2])
        dot_product = np.dot(reference_vector, unit_vector)
        if dot_product >= 0:
            remaining_points.append(point)
    return np.array(remaining_points)


def find_intersection(midpoint1, normal1, midpoint2, normal2):
    A = np.array([normal1, -normal2]).T
    b = np.array(midpoint2) - np.array(midpoint1)
    if np.linalg.det(A) == 0:
        return None
    intersection = linalg_solve(A, b)
```

```python
    return midpoint1 + intersection[0] * normal1


def linalg_solve(A, b):
    n = len(A)
    M = [list(row) for row in A]
    for i in range(n):
        M[i].append(b[i])
    for k in range(n):
        max_row = max(range(k, n), key=lambda i: abs(M[i][k]))
        M[k], M[max_row] = M[max_row], M[k]

        for i in range(k + 1, n):
            factor = M[i][k] / M[k][k]
            for j in range(k, n + 1):
                M[i][j] -= factor * M[k][j]

    x = [0] * n
    for i in range(n - 1, -1, -1):
        x[i] = M[i][n] / M[i][i]
        for j in range(i - 1, -1, -1):
            M[j][n] -= M[j][i] * x[i]

    return x


def sort_points_cyclic_order(points):
    center = np.mean(points, axis=0)
    angles = np.arctan2(points[:, 1] - center[1], points[:, 0] - center[0])
    sorted_indices = custom_argsort(angles)
    return points[sorted_indices]


def main():
    P_0 = (0, 0)
    points_first = (
        generate_points(n=5, r_min=2.5, r_max=15, theta_min=5, theta_max=85)
        + generate_points(n=5, r_min=2.5, r_max=15, theta_min=95, theta_max=175)
        + generate_points(n=5, r_min=2.5, r_max=15, theta_min=185, theta_max=265)
        + generate_points(n=5, r_min=2.5, r_max=15, theta_min=275, theta_max=355)
    )
    print(f"All generated points: {points_first}")
    points = points_first.copy()
    closest_points = []

    while len(points) > 0:
        closest_point_index, closest_point = find_closest_point(
            points, P_0, closest_points
        )
        closest_points.append(closest_point)

        reference_vector = find_unit_vector(P_from=closest_point, P_to=P_0)
        points = filter_points_by_dot_product(points, closest_point, reference_vector)

        print(f"Selected closest point: {closest_point}")

        if points.size == 0:
            break

    midpoints = [(np.array(P[:2]) + np.array(P_0)) / 2 for P in closest_points]
    normals = []
    for P in closest_points:
        vector = np.array(P[:2]) - np.array(P_0)
        normal = np.array([-vector[1], vector[0]])
        unit_normal = normal / norm(normal)
```

```
            normals.append(unit_normal)

    midpoints = np.array(midpoints)
    angles = np.arctan2(midpoints[:, 1], midpoints[:, 0])
    sorted_indices = custom_argsort(angles)
    midpoints = midpoints[sorted_indices]
    normals = np.array(normals)[sorted_indices]

    intersection_points = []
    for i in range(len(midpoints)):
        next_index = (i + 1) % len(midpoints)
        intersection = find_intersection(
            midpoints[i], normals[i], midpoints[next_index], normals[next_index]
        )
        if intersection is not None:
            intersection_points.append(intersection)

    plt.figure(figsize=(10, 8))
    plt.scatter(
        [point[0] for point in points_first],
        [point[1] for point in points_first],
        color="gray",
        label="First Generated",
    )
    all_points = closest_points + [P_0]
    x_coords = [point[0] for point in all_points]
    y_coords = [point[1] for point in all_points]
    plt.scatter(x_coords, y_coords, color="green", label="Selected Points")

    for point in closest_points:
        plt.plot([P_0[0], point[0]], [P_0[1], point[1]], "gray", linestyle="dotted")
    plt.scatter(
        [point[0] for point in midpoints],
        [point[1] for point in midpoints],
        color="blue",
        label="Midpoints",
    )

    intersection_points = np.array(intersection_points)
    if intersection_points.size > 0:
        plt.scatter(
            intersection_points[:, 0],
            intersection_points[:, 1],
            color="purple",
            label="Intersections",
        )

        for i in range(len(intersection_points)):
            next_index = (i + 1) % len(intersection_points)
            plt.plot(
                [intersection_points[i][0], intersection_points[next_index][0]],
                [intersection_points[i][1], intersection_points[next_index][1]],
                color="purple",
            )

    plt.xlabel("X Coordinates")
    plt.ylabel("Y Coordinates")
    plt.title("Voronoi Cell and Points")
    plt.grid(True)
    plt.legend()
    plt.show()


if __name__ == "__main__":
    main()
```