me570_geometry.py

```python
"""
Classes and functions for Polygons and Edges
"""

import math
import numpy as np
import matplotlib.pyplot as plt


def no_edge_collisions(polygon, edge_to_check):
    """
    This function ensures that an inputted edge does not intersect with any of the
    edges of the polygon.
    Returns True if no intersection found, False otherwise
    """
    edge_shape = []
    num_cols = polygon.vertices.shape[1]

    for i in range(num_cols):
        new_edge = np.vstack(
            (polygon.vertices[:, i], polygon.vertices[:,
                                                      (i + 1) % num_cols])).T
        edge_shape.append(Edge(new_edge))

    for edge in edge_shape:
        if edge.is_collision(edge_to_check):
            return False

    return True


def between_segment(first, middle, last):
    """
    Function determines whether point p_2 lies along the segment designated by p_1
    and p_2
    """
    first_x = first[0, 0]
    first_y = first[1, 0]

    middle_x = middle[0, 0]
    middle_y = middle[1, 0]

    last_x = last[0, 0]
    last_y = last[1, 0]

    return bool((min(first_x, last_x) <= middle_x <= max(first_x, last_x))
                and (min(first_y, last_y) <= middle_y <= max(first_y, last_y)))


def angle(vertex0, vertex1, vertex2, angle_type='signed'):
    """
    Compute the angle between two edges  vertex0-vertex1 and  vertex0-vertex2 having an endpoint in
    common. The angle is computed by starting from the edge  vertex0-- vertex1, and then
    ``walking'' in a counterclockwise manner until the edge  vertex0-vertex2 is found.
    The angle is computed by starting from the vertex0-vertex1 edge, and then “walking” in a
    counterclockwise manner until the is found.
    """
    # tolerance to check for coincident points
    tol = 2.22e-16

    # compute vectors corresponding to the two edges, and normalize
```

```python
    vec1 = vertex1 - vertex0
    vec2 = vertex2 - vertex0

    norm_vec1 = np.linalg.norm(vec1)
    norm_vec2 = np.linalg.norm(vec2)
    if norm_vec1 < tol or norm_vec2 < tol:
        # vertex1 or vertex2 coincides with vertex0, abort
        edge_angle = math.nan
        return edge_angle

    vec1 = vec1 / norm_vec1
    vec2 = vec2 / norm_vec2

    # Transform vec1 and vec2 into flat 3-D vectors,
    # so that they can be used with np.inner and np.cross
    vec1flat = np.vstack([vec1, 0]).flatten()
    vec2flat = np.vstack([vec2, 0]).flatten()

    c_angle = np.inner(vec1flat, vec2flat)
    s_angle = np.inner(np.array([0, 0, 1]), np.cross(vec1flat, vec2flat))

    edge_angle = math.atan2(s_angle, c_angle)

    angle_type = angle_type.lower()
    if angle_type == 'signed':
        # nothing to do
        pass
    elif angle_type == 'unsigned':
        edge_angle = math.modf(edge_angle + 2 * math.pi, 2 * math.pi)
    else:
        raise ValueError('Invalid argument angle_type')

    return edge_angle


class Polygon:
    """ Class for plotting, drawing, checking visibility and collision with polygons. """
    def __init__(self, vertices):
        """
        Save the input coordinates to the internal attribute  vertices.
        """
        self.vertices = vertices

    def flip(self):
        """
        Reverse the order of the vertices (i.e., transform the polygon from filled in
        to hollow and viceversa).
        """
        self.vertices = np.fliplr(self.vertices)

    def plot(self, style):
        """
        Plot the polygon using Matplotlib.
        """

        # To obtain the directions of the arrows needed, we can take the displacement of each vertex
        # to itself. This requires an np.diff() with itself, calculating out[i] = a[i+1] - a[i], and
        # then concatenating the last vertex - the first.

        displacement = np.hstack(
            (np.diff(self.vertices),
             (self.vertices[:, 0] - self.vertices[:, -1]).reshape(2, 1)))

        x_values = self.vertices[0, :]
        y_values = self.vertices[1, :]
```

```python
    x_displacement = displacement[0, :]
    y_displacement = displacement[1, :]

    plt.quiver(x_values,
               y_values,
               x_displacement,
               y_displacement,
               scale=1,
               scale_units='xy',
               angles='xy',
               color='black')
    bool(style)

def is_filled(self):
    """
    Checks the ordering of the vertices, and returns whether the polygon is filled in or not.
    """

    # Iteratres over the columns of the 2D Matrix to perform the calculation
    # sum((x_2 - x_1) * (y_2 + y_1))
    # If the sum is negative, then the polygon is oriented counter-clockwise,
    # clockwise otherwise.
    num_cols = self.vertices.shape[1]
    running_sum = 0

    for i in range(num_cols - 1):
        x_vals = self.vertices[0, :]
        y_vals = self.vertices[1, :]

        # modulus is for the last element to be compared with the first to close the shape
        running_sum += (x_vals[(i+1) % num_cols] - x_vals[i]) * \
            (y_vals[i] + y_vals[(i+1) % num_cols])

    return bool(running_sum < 0)

def is_self_occcluded(self, idx_vertex, point):
    """
    Given the corner of a polygon, checks whether a given point is self-occluded or not by
    that polygon (i.e., if it is ``inside'' the corner's cone or not). Points on boundary
    (i.e., on one of the sides of the corner) are not considered self-occluded. Note that
    to check self-occlusion, we just need a vertex index  idx_vertex. From this, one can
    obtain the corresponding  vertex, and the  vertex_prev and  vertex_next that precede
    and follow that vertex in the polygon.
    """

    solid = self.is_filled()
    # if idx_vertex == 0, -1 in python refers to the last so it works
    prev_vertex = np.vstack(self.vertices[:, idx_vertex - 1])
    vertex = np.vstack(self.vertices[:, idx_vertex])
    # if idx is the end, we need to loop around to the beginning
    next_vertex = np.vstack(self.vertices[:, (idx_vertex + 1) %
                                          self.vertices.shape[1]])

    # Ensure the vertices are all different
    if (np.array_equal(prev_vertex, vertex)
            or np.array_equal(next_vertex, vertex)):
        return False

    # Solid Case
    # GOAL: If orientation hits prev_vertex first, we are self-occluded
    # Compute signed angle between prev and next vertex:
    #     case 1: signed angle is negative
    #         compute signed angle from prev -> point
    #         if the sign negative:
    #             if angle is >= first_angle then it's occluded
    #             else good
```

```python
    #         if the sign is positive:
    #             good

    #     case 2: signed angle is positive
    #         compute signed angle from prev -> point
    #         if the sign is positive:
    #             angle is <= first_angle, then good
    #             else occluded
    #         if the sign is negative:
    #             occluded

    prev_next_angle = angle(vertex, prev_vertex, next_vertex)
    prev_point_angle = angle(vertex, prev_vertex, point)

    flag_point = False

    if solid:
        if prev_next_angle < 0:
            if prev_point_angle < 0:
                flag_point = prev_point_angle >= prev_next_angle
            else:
                flag_point = False
        else:
            if prev_point_angle > 0:
                flag_point = prev_point_angle > prev_next_angle
            else:
                flag_point = True
    else:
        # Hollow Case

        # GOAL: If orientation hits prev_vertex first, we are self-occluded
        # Compute the signed angle between prev and the next vertex:
        #     case 1: signed angle is positive:
        #         compute signed angle from prev -> point
        #         if the sign is positive:
        #             angle <= first_angle, good
        #             else, self-occluded
        #         if the sign is negative:
        #             self-occluded

        #     case 2: signed angle is negative:
        #         compute the signed angle from prev -> point
        #         if the sign is negative:
        #             angle > first_angle, self-occluded
        #             else, good
        #         if the sign is positive:
        #             good

        if prev_next_angle > 0:
            if prev_point_angle > 0:
                flag_point = prev_point_angle > prev_next_angle
            else:
                flag_point = True
        else:
            if prev_point_angle < 0:
                flag_point = prev_point_angle > prev_next_angle
            else:
                flag_point = False

    return flag_point

def is_visible(self, idx_vertex, test_points):
    """
    Checks whether a point p is visible from a vertex v of a polygon. In order to be visible,
    two conditions need to be satisfied: enumerate  point p should not be self-occluded with
    respect to the vertex v (see Polygon.is_self_occluded). The segment p--v should not collide
```

```python
        with any of the edges of the polygon (see Edge.is_collision).
        """

        vertex = np.vstack(self.vertices[:, idx_vertex])

        flag_points = []
        for point in test_points.T:
            point = np.vstack(point)
            edge_to_check = Edge(np.hstack((vertex, point)))
            if (not self.is_self_occcluded(idx_vertex, point)
                    and no_edge_collisions(self, edge_to_check)):
                flag_points.append(True)
            else:
                flag_points.append(False)

        return flag_points

    def is_collision(self, test_points):
        """
        Checks whether the a point is in collsion with a polygon (that is, inside for a filled in
        polygon, and outside for a hollow polygon). In the context of this homework, this function
        is best implemented using Polygon.is_visible.
        """
        visible_points = np.zeros(test_points.shape[1])
        flag_points = []

        for i in range(self.vertices.shape[1]):
            visible_list = self.is_visible(i, test_points)
            for j in range(test_points.shape[1]):
                if visible_list[j] and visible_points[j] != 1:
                    visible_points[j] = 1

        for val in visible_points:
            if val == 1:
                flag_points.append(False)
            else:
                flag_points.append(True)

        return flag_points


class Edge:
    """ Class for storing edges and checking collisions among them. """
    def __init__(self, vertices):
        """
        Save the input coordinates to the internal attribute vertices.
        """
        self.vertices = vertices

    def is_collision(self, edge):
        """
        Returns  True if the two edges intersect.  Note: if the two edges overlap but are colinear,
        or they overlap only at a single endpoint, they are not considered as intersecting (i.e.,
        in these cases the function returns  False). If one of the two edges has zero length, the
        function should always return the result that edges are non-intersecting.
        """

        # Check to make sure the edge isn't length 0
        if np.linalg.norm(np.diff(edge.vertices) == 0):
            return False

        # swap vertices to use diff
        edge.vertices[:, [1, 0]] = edge.vertices[:, [0, 1]]

        matrix = np.hstack((np.diff(self.vertices), np.diff(edge.vertices)))
```

```
        p_1 = np.vstack(self.vertices[:, 0])
        p_2 = np.vstack(self.vertices[:, 1])

        p_3 = np.vstack(edge.vertices[:, 0])
        p_4 = np.vstack(edge.vertices[:, 1])

        # Lines are collinear and can be tested for overlap vs parallelism
        if np.linalg.det(matrix) == 0:
            if between_segment(p_1, p_3, p_2) or between_segment(
                    p_3, p_1, p_4) or between_segment(
                        p_1, p_4, p_2) or between_segment(p_3, p_2, p_4):
                return False
            return True

        segment_timings = np.linalg.solve(matrix, p_3 - p_1)
        segment_1_t = abs(segment_timings[0, 0])
        segment_2_u = abs(segment_timings[1, 0])

        tol = 2.22e-16
        # Cases for times:
        t_is_endpoint = segment_1_t in (0, -1 * tol,
                                        tol) or segment_1_t in (1 - tol, 1,
                                                                1 + tol)
        t_on_segment = (-1 * tol) < segment_1_t < 1 + tol

        u_is_endpoint = segment_2_u in (0, -1 * tol,
                                        tol) or segment_2_u in (1 - tol, 1,
                                                                1 + tol)
        u_on_segment = (-1 * tol) <= segment_2_u <= 1 + tol

        # Corner cases
        # 1 Endpoint touching Endpoint
        # 2 Endpoint touching line (T-like shape)
        # if (t_on_segment and u_on_segment):
        #     return True
        if (t_is_endpoint and u_is_endpoint):
            return False
        if (t_on_segment and u_is_endpoint) or (t_is_endpoint
                                                and u_on_segment):
            return False
        return bool(t_on_segment and u_on_segment)
```

me570_robot.py

```
"""
Representation of a simple robot used in the assignments
"""

import numpy as np
import me570_geometry as gm


class TwoLink:
    """ A class containing methods for a two-link manipulator. """
    def __init__(self):
        add_y_reflection = lambda vertices: np.hstack(
            [vertices, np.fliplr(np.diag([1, -1]).dot(vertices))])

        vertices1 = np.array([[0, 5], [-1.11, -0.511]])
        vertices1 = add_y_reflection(vertices1)
        vertices2 = np.array([[0, 3.97, 4.17, 5.38, 5.61, 4.5],
                              [-0.47, -0.5, -0.75, -0.97, -0.5, -0.313]])
        vertices2 = add_y_reflection(vertices2)
        self.Polygons = (gm.Polygon(vertices1), gm.Polygon(vertices2))
```

```python
    def polygons(self):
        """
        Returns two polygons that represent the links in a simple 2-D two-link manipulator.
        """
        return self.Polygons
```

me570_queue.py

```python
"""
A pedagogical implementation of a priority queue
"""


class PriorityElement:
    """ Store a key and a value about an element of the queue. """
    def __init__(self, key, value):
        """
        Stores the arguments as internal attributes.
        """
        self.element = (key, value)

    def __str__(self) -> str:
        return f"{self.element}"


class Priority:
    """ Implements a priority queue """
    def __init__(self):
        """
        Initializes the internal attribute  queue to be an empty list.
        """
        self.queue = []

    def insert(self, key, cost):
        """
        Add an element to the queue.
        """
        self.queue.append(PriorityElement(key, cost))

    def min_extract(self):
        """
        Extract the element with minimum cost from the queue.
        """
        if len(self.queue) == 0:
            return None, None

        min_location = 0
        for i, p_element in enumerate(self.queue):
            if p_element.element[1] < self.queue[min_location].element[1]:
                min_location = i

        key = self.queue[min_location].element[0]
        cost = self.queue[min_location].element[1]

        del self.queue[min_location]

        return key, cost

    def is_member(self, key):
        """
        Check whether an element with a given key is in the queue or not.
        """
        flag = False
```

```python
        for _, p_element in enumerate(self.queue):
            if p_element.element[0] == key:
                flag = True
                break

        return flag
```

me570_hw1.py

```python
"""
Test functions for HW1
"""

import numpy as np
import matplotlib.pyplot as plt
import me570_robot as robot
import me570_queue as PriorityQueue


def polygon_is_visible_test():
    """
This function should perform the following operations:
- Create an array  test_points with dimensions [2 x 5] containing points generated uniformly at
random using np.random.rand and scaled to approximately occupy the rectangle [0,5] [-2,2] (i.e., the
x coordinates of the points should fall between 0 and 5, while the y coordinates between -2 and 2).
- Obtain the polygons  polygon1 and  polygon2 from Two_Link.Polygons.
- item:test-polygon For each polygon  polygon1,  polygon2, display a separate figure using the
following:
- Create the array  test_points_with_polygon by concatenating  test_points with the coordinates of
the polygon (i.e., the coordinates of the polygon become also test points).
- Plot the polygon (use Polygon.plot).
- item:test-visibility For each vertex v in the polygon:
- Compute the visibility of each point in  test_points_with_polygon with respect to that polygon
(using Polygon.is_visible).
- Plot lines from the vertex v to each point in  test_points_with_polygon in green if the
corresponding point is visible, and in red otherwise.
- Reverse the order of the vertices in the two polygons using Polygon.flip.
- Repeat item item:test-polygon above with the reversed polygons.
    """
    test_points = np.random.rand(2, 5)

    # Scale x coordinates to uniformly cover [0, 5)
    test_points[0, :] *= 5

    # Scale y coordinates to uniformly cover [-2, 2)
    #   formula used: low + ((high - low) * random_value)
    test_points[1, :] *= 4  # high - low
    test_points[1, :] -= 2  # low

    # Obtain polygon1 and polygon2
    two_link = robot.TwoLink()
    robot_polygons = two_link.polygons()

    for polygon in robot_polygons:
        test_points_with_polygon = np.hstack((polygon.vertices, test_points))
        plt.figure()
        polygon.plot([])
        for i in range(polygon.vertices.shape[1]):
            vertex = np.vstack(polygon.vertices[:, i])
            for j in range(test_points_with_polygon.shape[1]):
                point_to_test = np.vstack(test_points_with_polygon[:, j])
                visible = polygon.is_visible(i, point_to_test)
```

```python
            values = np.hstack((vertex, point_to_test))
            x_vals = values[0, :]
            y_vals = values[1, :]

            if visible[0]:
                plt.plot(x_vals,
                         y_vals,
                         'g',
                         marker='o',
                         markeredgecolor='k',
                         markerfacecolor='k',
                         linewidth=1,
                         alpha=0.8)
            else:
                plt.plot(x_vals,
                         y_vals,
                         'r',
                         marker='o',
                         markeredgecolor='k',
                         markerfacecolor='k',
                         linewidth=0.5,
                         alpha=0.8)
    plt.show()

    polygon.flip()
    plt.figure()
    polygon.plot([])

    for i in range(polygon.vertices.shape[1]):
        vertex = np.vstack(polygon.vertices[:, i])
        for j in range(test_points_with_polygon.shape[1]):
            point_to_test = np.vstack(test_points_with_polygon[:, j])
            visible = polygon.is_visible(i, point_to_test)

            values = np.hstack((vertex, point_to_test))
            x_vals = values[0, :]
            y_vals = values[1, :]

            if visible[0]:
                plt.plot(x_vals,
                         y_vals,
                         'g',
                         marker='o',
                         markeredgecolor='k',
                         markerfacecolor='k',
                         linewidth=1,
                         alpha=0.8)
            else:
                plt.plot(x_vals,
                         y_vals,
                         'r',
                         marker='o',
                         markeredgecolor='k',
                         markerfacecolor='k',
                         linewidth=0.5,
                         alpha=0.8)
    plt.show()


def polygon_is_collision_test():
    """
    This function is the same as polygon_is_visible_test, but use
the following:
 - Compute whether each point in  test_points_with_polygon is in collision with the polygon or not
using Polygon.is_collision.
 - Plot each point in  test_points_with_polygon in green if it is not in collision, and red
```

```
otherwise.  Moreover, increase the number of test points from 5 to 100 (i.e.,  testPoints should
have dimension [2 x 100]).
    """

    test_points = np.random.rand(2, 100)

    # Scale x coordinates to uniformly cover [0, 5)
    test_points[0, :] *= 5

    # Scale y coordinates to uniformly cover [-2, 2)
    #   formula used: low + ((high - low) * random_value)
    test_points[1, :] *= 4  # high - low
    test_points[1, :] -= 2  # low

    # Obtain polygon1 and polygon2
    two_link = robot.TwoLink()
    robot_polygons = two_link.polygons()

    for polygon in robot_polygons:
        test_points_with_polygon = np.hstack((polygon.vertices, test_points))
        plt.figure()
        polygon.plot([])

        green_x = []
        green_y = []
        red_x = []
        red_y = []

        flagged_points = polygon.is_collision(test_points_with_polygon)
        for i, point in enumerate(test_points_with_polygon.T):
            x_point = point[0]
            y_point = point[1]
            if flagged_points[i] is True:
                red_x.append(x_point)
                red_y.append(y_point)
            else:
                green_x.append(x_point)
                green_y.append(y_point)

        plt.scatter(green_x, green_y, color='green')
        plt.scatter(red_x, red_y, color='red')

        plt.show()

        green_x.clear()
        green_y.clear()
        red_x.clear()
        red_y.clear()

        polygon.flip()
        plt.figure()
        polygon.plot([])

        flagged_points = polygon.is_collision(test_points_with_polygon)
        for i, point in enumerate(test_points_with_polygon.T):
            x_point = point[0]
            y_point = point[1]
            if flagged_points[i] is True:
                red_x.append(x_point)
                red_y.append(y_point)
            else:
                green_x.append(x_point)
                green_y.append(y_point)

        plt.scatter(green_x, green_y, color='green')
        plt.scatter(red_x, red_y, color='red')
```

```python
        plt.show()


def priority_test():
    """
    The function should perform the following steps:  enumerate
 - Initialize an empty queue.
 - Add three elements (as shown in Table~tab:priority-test-inputs and in that order) to that queue.
 - Extract a minimum element.
 - Add another element (as shown in Table~tab:priority-test-inputs).
 - Check if an element is present.
 - Remove all elements by repeated extractions.  enumerate After each step, display the content of
pQueue.
    """
    my_queue = PriorityQueue.Priority()

    my_queue.insert("Oranges", 4.5)
    my_queue.insert("Apples", 1)
    my_queue.insert("Bananas", 2.7)
    print(*my_queue.queue, sep=", ")

    key, cost = my_queue.min_extract()
    print(f"({key}, {cost})")

    my_queue.insert("Cantaloupe", 3)
    print(*my_queue.queue, sep=", ")

    print(f"Oranges is in my queue? --> {my_queue.is_member('Oranges')}")
    print(f"Milk is in my queue? --> {my_queue.is_member('Milk')}")

    while True:
        (key, value) = my_queue.min_extract()
        if (key is None and value is None):
            break

        print(f"Removed: ({key}, {value}) --> remaining: ", end=" ")
        print(*my_queue.queue, sep=", ")


priority_test()
```