me570_geometry.py

```python
"""
 Please merge the functions and classes from this file with the same file from the previous
 homework assignment
"""

import numbers
import math
import numpy as np
from matplotlib import cm, pyplot as plt


def numel(var):
    """
    Counts the number of entries in a numpy array, or returns 1 for fundamental numerical
    types

    [This function is the same as the one from HW2]
    """
    if isinstance(var, numbers.Number):
        size = int(1)
    elif isinstance(var, np.ndarray):
        size = var.size
    else:
        # breakpoint()
        raise NotImplementedError(f'number of elements for type {type(var)}')
    return size


def rot2d(theta):
    """
    Create a 2-D rotation matrix from the angle  theta according to (1).
    """
    rot_theta = np.array([[math.cos(theta), -math.sin(theta)],
                          [math.sin(theta), math.cos(theta)]])
    return rot_theta


def line_linspace(a_line, b_line, t_min, t_max, nb_points):
    """
    Generates a discrete number of  nb_points points along the curve
    (t)=( a(1)t + b(1), a(2)t + b(2))  R^2 for t ranging from  tMin to  tMax.
    """
    t_sequence = np.linspace(t_min, t_max, nb_points)
    theta_points = a_line * t_sequence + b_line
    return theta_points


class Grid:
    """
    A function to store the coordinates of points on a 2-D grid and evaluate arbitrary
    functions on those points.

    [This class is the same as the one from HW2]
    """
    def __init__(self, xx_grid, yy_grid):
        """
        Stores the input arguments in attributes.
        """
        self.xx_grid = xx_grid
        self.yy_grid = yy_grid

    def eval(self, fun):
        """
        This function evaluates the function  fun (which should be a function)
        on each point defined by the grid.
        """

        dim_domain = [numel(self.xx_grid), numel(self.yy_grid)]
        dim_range = [numel(fun(np.array([[0], [0]])))]
        fun_eval = np.nan * np.ones(dim_domain + dim_range)
        for idx_x in range(0, dim_domain[0]):
            for idx_y in range(0, dim_domain[1]):
                x_eval = np.array([[self.xx_grid[idx_x]],
                                   [self.yy_grid[idx_y]]])
                fun_eval[idx_x, idx_y, :] = np.reshape(fun(x_eval),
                                                       [1, 1, dim_range[0]])

        # If the last dimension is a singleton, remove it
        if dim_range == [1]:
            fun_eval = np.reshape(fun_eval, dim_domain)

        return fun_eval

    def mesh(self):
        """
        Shorhand for calling meshgrid on the points of the grid
        """

        return np.meshgrid(self.xx_grid, self.yy_grid)
```

```python
def clip(val, threshold):
    """
    If val is a scalar, threshold its value; if it is a vector, normalized it
    """
    if isinstance(val, np.ndarray):
        val_norm = np.linalg.norm(val)
        if val_norm > threshold:
            val /= val_norm
    elif isinstance(val, numbers.Number):
        val = min(val, threshold)
    else:
        raise ValueError('Numeric format not recognized')

    return val


def field_plot_threshold(f_handle, threshold=10, nb_grid=61):
    """
    The function evaluates the function  f_handle on points placed on a regular grid.
    """

    xx_grid = np.linspace(-11, 11, nb_grid)
    yy_grid = np.linspace(-11, 11, nb_grid)
    grid = Grid(xx_grid, yy_grid)

    f_handle_clip = lambda val: clip(f_handle(val), threshold)
    f_eval = grid.eval(f_handle_clip)

    [xx_mesh, yy_mesh] = grid.mesh()
    f_dim = numel(f_handle_clip(np.zeros((2, 1))))
    if f_dim == 1:
        # scalar field
        fig = plt.gcf()
        axis = fig.add_subplot(111, projection='3d')

        axis.plot_surface(xx_mesh,
                          yy_mesh,
                          f_eval.transpose(),
                          cmap=cm.gnuplot2)
        axis.view_init(90, -90)
    elif f_dim == 2:
        # vector field
        # grid.eval gives the result transposed with respect to what meshgrid expects
        f_eval = f_eval.transpose((1, 0, 2))
        # vector field
        plt.quiver(xx_mesh,
                   yy_mesh,
                   f_eval[:, :, 0],
                   f_eval[:, :, 1],
                   angles='xy',
                   scale_units='xy')
    else:
        raise NotImplementedError(
            'Field plotting for dimension greater than two not implemented')

    plt.xlabel('x')
    plt.ylabel('y')


class Sphere:
    """ Class for plotting and computing distances to spheres (circles, in 2-D). """
    def __init__(self, center, radius, distance_influence):
        """
        Save the parameters describing the sphere as internal attributes.
        """
        self.center = center
        self.radius = radius
        self.distance_influence = distance_influence

    def plot(self, color):
        """
    This function draws the sphere (i.e., a circle) of the given radius, and the specified color,
    and then draws another circle in gray with radius equal to the distance of influence.
        """
        # Get current axes
        axes = plt.gca()

        # Added these in to make circles look like circles and actually get circles to display
        axes.axis('equal')
        axes.plot()

        # Add circle as a patch
        if self.radius > 0:
            # Circle is filled in
            kwargs = {'facecolor': (0.3, 0.3, 0.3)}
            radius_influence = self.radius + self.distance_influence
        else:
            # Circle is hollow
            kwargs = {'fill': False}
            radius_influence = -self.radius - self.distance_influence

        center = (self.center[0, 0], self.center[1, 0])
        axes.add_patch(
            plt.Circle(center,
                       radius=abs(self.radius),
                       edgecolor=color,
                       **kwargs))

        axes.add_patch(
            plt.Circle(center,
                       radius=radius_influence,
                       edgecolor=(0.7, 0.7, 0.7),
                       fill=False))

    def distance(self, points):
        """
    Computes the signed distance between points and the sphere, while taking into account whether
    the sphere is hollow or filled in.
        """
        d_points_sphere = np.linalg.norm(points - self.center, axis=0) - abs(
            self.radius)

        if self.radius < 0:
            return -d_points_sphere
        return d_points_sphere

    def distance_grad(self, points):
        """
    Computes the gradient of the signed distance between points and the sphere, consistently with
    the definition of Sphere.distance.
        """
        denom = np.linalg.norm(points - self.center, axis=0)
        if not all(denom):
            return np.zeros((2, 1))

        grad_d_points_sphere = (points - self.center) / np.linalg.norm(
            points - self.center, axis=0)

        if self.radius < 0:
            return -grad_d_points_sphere
        return grad_d_points_sphere

    def is_collision(self, points):
        """
        Determines whether a point collides with the sphere using its distance
        as the metric for collision

        distance <= 0 means collision in both solid and hollow
        """
        flag_points = [point <= 0 for point in self.distance(points)]
        return flag_points

    def flip(self):
        """
        Turns the Sphere from Solid -> Hollow and vice versa
        """
        self.radius = -self.radius


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

    @property
    def nb_vertices(self):
        """ Number of vertices """
        return self.vertices.shape[1]

    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.
        """
        if len(style) == 0:
            style = 'k'

        directions = np.diff(self.vertices_loop)
        plt.quiver(self.vertices[0, :],
                   self.vertices[1, :],
                   directions[0, :],
                   directions[1, :],
                   color=style,
                   angles='xy',
                   scale_units='xy',
                   scale=1.)
```

```
@property
def vertices_loop(self):
    """
    Returns self.vertices with the first vertex repeated at the end
    """
    return np.hstack((self.vertices, self.vertices[:, [0]]))

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 running_sum < 0

def is_self_occluded(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.
    """
    vertex = self.vertices[:, [idx_vertex]]
    vertex_next = self.vertices[:, [(idx_vertex + 1) % self.nb_vertices]]
    vertex_prev = self.vertices[:, [(idx_vertex - 1) % self.nb_vertices]]

    # The point is occluded if, measuring angles using p-vertex as the "zero angle",
    # the angle for vertex_prev is smaller than the one for vertex_next
    # Using the 'unsigned' angles means that we do not have to worry separately
    # about negative angles
    angle_p_prev = angle(vertex, point, vertex_prev, 'unsigned')
    angle_p_next = angle(vertex, point, vertex_next, 'unsigned')

    return angle_p_prev < angle_p_next

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
    with any of the edges of the polygon (see Edge.is_collision).
    """
    nb_test_points = test_points.shape[1]
    nb_vertices = self.vertices.shape[1]

    # Initial default: all flags are True
    flag_points = [True] * nb_test_points
    vertex = self.vertices[:, [idx_vertex]]
    for idx_point in range(0, nb_test_points):
        point = test_points[:, [idx_point]]

        # If it is self occluded, bail out
        if self.is_self_occluded(idx_vertex, point):
            flag_points[idx_point] = False
        else:
            # Build the vertex-point edge (it is the same for all other edges)
            edge_vertex_point = Edge(np.hstack([point, vertex]))
            # Then iterate over all edges in the polygon
            for idx_vertex_collision in range(0, self.nb_vertices):
                edge_vertex_vertex = Edge(self.vertices[:, [
                    idx_vertex_collision,
                    (idx_vertex_collision + 1) % nb_vertices
                ]])
                # The final result is the and of all the checks with individual edges
                flag_points[
                    idx_point] &= not edge_vertex_point.is_collision(
                        edge_vertex_vertex)

                # Early bail out after one collision
                if not flag_points[idx_point]:
                    break

    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.
        """

        flag_points = [False] * test_points.shape[1]
        # We iterate over the polygon vertices, and process all the test points in parallel
        for idx_vertex in range(0, self.nb_vertices):
            flag_points_vertex = self.is_visible(idx_vertex, test_points)
            # Accumulate the new flags with the previous ones
            flag_points = [
                flag_prev or flag_new
                for flag_prev, flag_new in zip(flag_points, flag_points_vertex)
            ]

        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

    @property
    def direction(self):
        """ Difference between tip and base """
        return self.vertices[:, [1]] - self.vertices[:, [0]]

    @property
    def base(self):
        """ Coordinates of the first vertex"""
        return self.vertices[:, [0]]

    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.
        """

        # Write the lines from the two edges as x_i(t_i)=edge_base+edge.direction*t_i
        # Then finds the parameters for the intersection by solving the linear system obtained from
        # x_1(t_1)=x_2(t_2)

        # Tolerance for cases involving parallel lines and endpoints
        tol = 1e-6

        # The matrix of the linear system
        a_directions = np.hstack([self.direction, -edge.direction])
        if abs(np.linalg.det(a_directions)) < tol:
            # Lines are practically parallel
            return False
        # The vector of the linear system
        b_bases = np.hstack([edge.base - self.base])

        # Solve the linear system
        t_param = np.linalg.solve(a_directions, b_bases)
        t_self = t_param[0, 0]
        t_other = t_param[1, 0]

        # Check that collision point is strictly between endpoints of each edge
        flag_collision = tol < t_self < 1.0 - tol and tol < t_other < 1.0 - tol

        return flag_collision


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
    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 = (edge_angle + 2 * math.pi) % (2 * math.pi)
    else:
        raise ValueError('Invalid argument angle_type')

    return edge_angle
```

me570_potential.py

```
"""
Classes to define potential and potential planner for the sphere world
"""

import math
import numpy as np
from matplotlib import pyplot as plt
from scipy import io as scio
import me570_geometry
from me570_qp import qp_supervisor


class SphereWorld:
    """ Class for loading and plotting a 2-D sphereworld. """
    def __init__(self):
        """
        Load the sphere world from the provided file sphereworld.mat, and sets the
    following attributes:
     -  world: a  nb_spheres list of  Sphere objects defining all the spherical obstacles in the
    sphere world.
     -  x_start, a [2 x nb_start] array of initial starting locations (one for each column).
     -  x_goal, a [2 x nb_goal] vector containing the coordinates of different goal locations (one
    for each column).
        """
        data = scio.loadmat('sphereWorld.mat')

        self.world = []
        for sphere_args in np.reshape(data['world'], (-1, )):
            sphere_args[1] = np.asscalar(sphere_args[1])
            sphere_args[2] = np.asscalar(sphere_args[2])
            self.world.append(me570_geometry.Sphere(*sphere_args))

        self.x_goal = data['xGoal']
        self.x_start = data['xStart']
        self.theta_start = data['thetaStart']

    def plot(self):
        """
        Uses Sphere.plot to draw the spherical obstacles together with a
        * marker at the goal location.
        """

        for sphere in self.world:
            sphere.plot('k')

        plt.scatter(self.x_goal[0, :], self.x_goal[1, :], c='g', marker='*')

        plt.xlim([-11, 11])
        plt.ylim([-11, 11])


class RepulsiveSphere:
    """ Repulsive potential for a sphere """
    def __init__(self, sphere):
        """
        Save the arguments to internal attributes
        """
        self.sphere = sphere

    def eval(self, x_eval):
        """
    Evaluate the repulsive potential from  sphere at the location x= x_eval. The function returns
    the repulsive potential as given by
    (  eq:repulsive  ).
        """
        d_subi_x = self.sphere.distance(x_eval)

        if d_subi_x > self.sphere.distance_influence:
            u_rep = 0
        elif 0 < d_subi_x < self.sphere.distance_influence:
            u_rep = 0.5 * (1 / d_subi_x -
                           1 / self.sphere.distance_influence)**2
        else:
            u_rep = math.nan

        return u_rep

    def grad(self, x_eval):
        """
        Compute the gradient of U_ rep for a single sphere, as given by
        (eq:repulsive-gradient).
        """
        d_subi_x = self.sphere.distance(x_eval)
        grad_d_subi_x = self.sphere.distance_grad(x_eval)

        if d_subi_x > self.sphere.distance_influence:
            grad_u_rep = 0
        elif 0 < d_subi_x < self.sphere.distance_influence:
            grad_u_rep = -(1 / d_subi_x - 1 / self.sphere.distance_influence
                           ) * (1 / d_subi_x**2) * grad_d_subi_x
        else:
            grad_u_rep = math.nan

        return grad_u_rep


class Attractive:
    """ Repulsive potential for a sphere """
    def __init__(self, potential):
        """
        Save the arguments to internal attributes
        """
        self.potential = potential

    def eval(self, x_eval):
        """
        Evaluate the attractive potential  U_ attr at a point  xEval with respect to a goal location
    potential.xGoal given by the formula: If  potential.shape is equal to  'conic', use p=1. If
    potential.shape is equal to  'quadratic', use p=2.
        """
        # Test if the necessary keys are a subset of self.potential
        if {'shape', 'x_goal'} <= self.potential.keys():
            attr_shape = self.potential['shape'].lower()
            if attr_shape == "conic":
                u_attr = np.linalg.norm(x_eval - self.potential['x_goal'])
            elif attr_shape == "quadratic":
                u_attr = np.linalg.norm(x_eval - self.potential['x_goal'])**2
            else:
                raise NotImplementedError(
                    f"Attractive Potential is undefined for shape, \"{attr_shape}\""
                )
        else:
            raise ValueError(
                'Definition of potential does not indicate a shape')

        return u_attr

    def grad(self, x_eval):
        """
    Evaluate the gradient of the attractive potential  U_ attr at a point  xEval. The gradient is
    given by the formula If  potential['shape'] is equal to  'conic', use p=1; if it is equal to
    'quadratic', use p=2.
        """
        # Test if the necessary keys are a subset of self.potential
        if {'shape', 'x_goal'} <= self.potential.keys():
            attr_shape = self.potential['shape'].lower()
            if attr_shape == "conic":
                grad_u_attr = (x_eval - self.potential['x_goal']
                               ) / np.linalg.norm(x_eval -
                                                  self.potential['x_goal'])
            elif attr_shape == "quadratic":
                grad_u_attr = x_eval - self.potential['x_goal']
            else:
                raise NotImplementedError(
                    f"Attractive Potential is undefined for shape, \"{attr_shape}\""
                )
        else:
            raise ValueError(
                'Definition of potential does not indicate a shape')
        return grad_u_attr


class Total:
    """ Combines attractive and repulsive potentials """
    def __init__(self, world, potential):
        """
        Save the arguments to internal attributes
        """
        self.world: SphereWorld = world
        self.potential: dict = potential

    def eval(self, x_eval):
        """
        Compute the function U=U_attr+a*iU_rep,i, where a is given by the variable
    potential.repulsiveWeight
        """
        # Ensure the proper fields are present
        # obstacles: list[RepulsiveSphere] = []
        obstacle_potential = 0
        if {'shape', 'x_goal', 'repulsive_weight'} <= self.potential.keys():
            attr = Attractive(self.potential)
            for sphere in self.world.world:
                obstacle_potential = obstacle_potential + RepulsiveSphere(
                    sphere).eval(x_eval)
        else:
            raise ValueError(
                "Must have all necessary fields: 'shape', 'x_goal', and 'repulsive_weight'"
            )

        u_eval = attr.eval(
            x_eval) + self.potential['repulsive_weight'] * obstacle_potential

        return u_eval

    def grad(self, x_eval):
        """
    Compute the gradient of the total potential,
    U= U_ attr+    _i U_ rep,i, where   is given by the variable
    potential.repulsiveWeight
        """
        # Ensure the proper fields are present
        obstacle_gradient = np.zeros((2, 1))
        if {'shape', 'x_goal', 'repulsive_weight'} <= self.potential.keys():
            attr = Attractive(self.potential)
            for sphere in self.world.world:
                obstacle_gradient = obstacle_gradient + RepulsiveSphere(
                    sphere).grad(x_eval)
        else:
            raise ValueError(
                "Must have all necessary fields: 'shape', 'x_goal', and 'repulsive_weight'"
            )

        grad_u_eval = attr.grad(
            x_eval) + self.potential['repulsive_weight'] * obstacle_gradient
        return grad_u_eval


class Planner:
    """
    Planner for creating the path from start -> goal
    """
    def run(self, x_start, planner_parameters):
        """
        This function uses a given function ( planner_parameters['control']) to implement a generic
    potential-based planner with step size  planner_parameters['epsilon'], and evaluates the cost
    along the returned path. The planner must stop when either the number of steps given by
    planner_parameters['nb_steps'] is reached, or when the norm of the vector given by
    planner_parameters['control'] is less than 5 10^-3 (equivalently,  5e-3).
        """

        # Create the trajectory path of the robot, pre-filling the list with 0s
        # in case the planner stops early
        x_path = np.zeros((2, planner_parameters['nb_steps']))

        # The first value is always going to be where the robot starts
        x_path[:, 0] = x_start.T

        # Create the list of potential values, pre-filling the list with 0s in case
        # the planner stops early
        u_path = np.zeros(planner_parameters['nb_steps'])

        # The first potential is always going to be evaluated at the start location
        u_path[0] = planner_parameters['U'](x_start)

        for k in range(planner_parameters['nb_steps'] - 1):
            x_val = np.vstack(x_path[:, k])
            control_val = planner_parameters['control'](x_val)

            # Determines if the planner should stop since the gradient is essentially 0
            # (meaning either stuck or found goal)
            if np.linalg.norm(control_val) < 5e-3:
                x_path[:, k:] = math.nan
                u_path[k:] = math.nan
                break

            x_path[:, k + 1] = (x_val +
                                planner_parameters['epsilon'] * control_val).T

            u_path[k + 1] = planner_parameters['U'](x_val)
        return x_path, u_path

    def run_plot(self):
        """
        This function performs the following steps:
     - Loads the problem data from the file !70!DarkSeaGreen2 sphereworld.mat.
     - For each goal location in  world.xGoal:
     - Uses the function Sphereworld.plot to plot the world in a first figure.
```

```
 - Sets  planner_parameters['U'] to the negative of  Total.grad.
 - it:grad-handle Calls the function Potential.planner with the problem data and the input
arguments. The function needs to be called five times, using each one of the initial locations
given in  x_start (also provided in !70!DarkSeaGreen2 sphereworld.mat).
 - it:plot-plan After each call, plot the resulting trajectory superimposed to the world in the
first subplot; in a second subplot, show  u_path (using the same color and using the  semilogy
command).
    """
    world = SphereWorld()
    # self.plot_quadratics(
    #     {
    #         'repulsive_weight': 35,
    #         'epsilon': 2e-2,
    #         'nb_steps': 600
    #     }, world)
    # self.plot_conics(
    #     {
    #         'repulsive_weight': 1,
    #         'epsilon': 3.55e-1,
    #         'nb_steps': 1000
    #     }, world)
    self.plot_quad_clfcbf(
        {
            'repulsive_weight': 5,
            'epsilon': 6.0e-2,
            'nb_steps': 100
        }, world)

    plt.show()

def plot_conics(self, config, world):
    """
    Method used for plotting conic potential functions
    """
    plt.rcParams["figure.figsize"] = (8, 8)
    _, axs = plt.subplots(world.x_goal.shape[1], 2)
    colors = plt.cm.get_cmap('hsv', world.x_start.shape[1] + 1)

    total_potential = None
    planner_parameters = None
    plt.subplots_adjust(hspace=0.305)

    for i, loc in enumerate(world.x_goal.T):

        # Setting up the constants and parameters we need
        total_potential = Total(
            world, {
                'x_goal': np.vstack(loc),
                'shape': 'conic',
                'repulsive_weight': config['repulsive_weight']
            })
        planner_parameters = {
            'U': lambda point: total_potential.eval(point),
            'control': lambda point: -total_potential.grad(point),
            'epsilon': config['epsilon'],
            'nb_steps': config['nb_steps']
        }

        for color_num, start_loc in enumerate(world.x_start.T):
            curr_start = np.vstack(start_loc)
            x_path, u_path = self.run(curr_start, planner_parameters)
            # Make sure we are plotting below the world we are concerned with
            plt.sca(axs[i, 0])
            plt.plot(x_path[0, :], x_path[1, :], color=colors(color_num))

            # Plotting the potential on the right-hand subplot
            plt.sca(axs[i, 1])
            plt.title(f"Goal {i}, Potential Conic")
            plt.xlabel('# steps')
            plt.ylabel('U')
            plt.semilogy(np.arange(0, planner_parameters['nb_steps']),
                         u_path,
                         color=colors(color_num))

        # Set current axis to left column to draw the world
        plt.sca(axs[i, 0])
        plt.title(f"Goal {i}, Trajectories Conic")
        world.plot()

def plot_quadratics(self, config, world):
    """
    Method for plotting quadratic potential functions
    """
    plt.rcParams["figure.figsize"] = (8, 8)
    _, axs = plt.subplots(world.x_goal.shape[1], 2)
    colors = plt.cm.get_cmap('hsv', world.x_start.shape[1] + 1)

    plt.subplots_adjust(hspace=0.305)

    total_potential = None
    planner_parameters = None
    for i, loc in enumerate(world.x_goal.T):

        # Setting up the constants and parameters we need
```

```python
            total_potential = Total(
                world, {
                    'x_goal': np.vstack(loc),
                    'shape': 'quadratic',
                    'repulsive_weight': config['repulsive_weight']
                })
            planner_parameters = {
                'U': lambda point: total_potential.eval(point),
                'control': lambda point: -total_potential.grad(point),
                'epsilon': config['epsilon'],
                'nb_steps': config['nb_steps']
            }

            for color_num, start_loc in enumerate(world.x_start.T):
                curr_start = np.vstack(start_loc)
                x_path, u_path = self.run(curr_start, planner_parameters)
                # Make sure we are plotting below the world we are concerned with
                plt.sca(axs[i, 0])
                plt.plot(x_path[0, :], x_path[1, :], color=colors(color_num))

                # Plotting the potential on the right-hand subplot
                plt.sca(axs[i, 1])
                plt.title(f"Goal {i}, Potential Quadratic")
                plt.xlabel('# steps')
                plt.ylabel('U')
                plt.semilogy(np.arange(0, planner_parameters['nb_steps']),
                             u_path,
                             color=colors(color_num))

            # Set current axis to left column to draw the world
            plt.sca(axs[i, 0])
            plt.title(f"Goal {i}, Trajectories Quadratic")
            world.plot()

    def plot_quad_clfcbf(self, config, world):
        """
        Method for plotting quadratic potential functions using CLF-CBF Formulation
        """
        plt.rcParams["figure.figsize"] = (8, 8)
        _, axs = plt.subplots(world.x_goal.shape[1], 2)
        colors = plt.cm.get_cmap('hsv', world.x_start.shape[1] + 1)

        plt.subplots_adjust(hspace=0.305)

        # Get rid of pylint errors of defining a variable in a loop
        potential = None
        planner_parameters = None
        for i, loc in enumerate(world.x_goal.T):

            # Setting up the constants and parameters we need
            potential = {
                'x_goal': np.vstack(loc),
                'shape': 'quadratic',
                'repulsive_weight': config['repulsive_weight']
            }
            planner_parameters = {
                'U': lambda point: Total(world, potential).eval(point),
                'control':
                lambda point: clfcbf_control(point, world, potential),
                'epsilon': config['epsilon'],
                'nb_steps': config['nb_steps']
            }

            for color_num, start_loc in enumerate(world.x_start.T):
                curr_start = np.vstack(start_loc)
                x_path, u_path = self.run(curr_start, planner_parameters)
                # Make sure we are plotting below the world we are concerned with
                plt.sca(axs[i, 0])
                plt.plot(x_path[0, :], x_path[1, :], color=colors(color_num))

                # Plotting the potential on the right-hand subplot
                plt.sca(axs[i, 1])
                plt.title(f"Goal {i}, Potential Quadratic")
                plt.xlabel('# steps')
                plt.ylabel('U')
                plt.semilogy(np.arange(0, planner_parameters['nb_steps']),
                             u_path,
                             color=colors(color_num))

            # Set current axis to left column to draw the world
            plt.sca(axs[i, 0])
            plt.title(f"Goal {i}, Trajectories Quadratic")
            world.plot()


def clfcbf_control(x_eval, world, potential):
    """
    Compute u^* according to
    (  eq:clfcbf-qp  ).
    """
    nb_obstacles = len(world.world)
    a_barrier = np.zeros((nb_obstacles, 2))
    b_barrier = np.zeros((nb_obstacles, 1))
    u_ref = -(Attractive(potential).grad(x_eval))
    for obst_num, sphere in enumerate(world.world):
        a_barrier[obst_num, :] = -(sphere.distance_grad(x_eval)).T
        b_barrier[obst_num, 0] = -(potential['repulsive_weight'] *
                                   sphere.distance(x_eval))

    u_opt = qp_supervisor(a_barrier, b_barrier, u_ref)
    return u_opt
```

me570_robot.py

```
"""
Please merge the functions and classes from this file with the same file from the previous
homework assignment
"""
import math
import numpy as np
import matplotlib.pyplot as plt
import me570_geometry as gm
import me570_potential as pot


class TwoLink:
    """
    Class for creating our Two_Link Manipulator
    """
    def __init__(self) -> None:
        """
        Creates the two polygons necessary for the robot
        """
        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))

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

    def kinematic_map(self, theta):
        """
        The function returns the coordinate of the end effector, plus the vertices of the links, all
    transformed according to  _1, _2.
        """

        # Rotation matrices
        w_r_beta_1 = gm.rot2d(theta[0, 0])
        beta_1_r_beta_2 = gm.rot2d(theta[1, 0])
        w_r_beta_2 = w_r_beta_1 @ beta_1_r_beta_2

        # Translation matrix
        beta_1_t_beta_2 = np.vstack((5, 0))
        w_t_beta_2 = gm.rot2d(theta[0, 0]) @ beta_1_t_beta_2

        # Transform End effector from β₂ to the world
        vertex_effector_transf = (w_r_beta_2 @ np.vstack((5, 0))) + w_t_beta_2

        # Polygon1's coordinates are in the β₁ coordinate space, so we need to calculate ʷp_β₁
        polygon_1_vert_transf = np.array([[], []])
        for i in range(self._polygons[0].nb_vertices):
            vertex = np.vstack(self._polygons[0].vertices[:, i])
            vertex_transf = (w_r_beta_1 @ vertex)
            polygon_1_vert_transf = np.hstack(
                (polygon_1_vert_transf, vertex_transf))

        polygon1_transf = gm.Polygon(polygon_1_vert_transf)

        # Polygon2's coordinates are in the β₂ coordinate space, so we need to calculate ʷp_β₂
        polygon_2_vert_transf = np.array([[], []])
        for i in range(self._polygons[1].nb_vertices):
            vertex = np.vstack(self._polygons[1].vertices[:, i])
            vertex_transf = (w_r_beta_2 @ vertex) + w_t_beta_2
            polygon_2_vert_transf = np.hstack(
                (polygon_2_vert_transf, vertex_transf))

        polygon2_transf = gm.Polygon(polygon_2_vert_transf)

        return vertex_effector_transf, polygon1_transf, polygon2_transf

    def plot(self, theta, color):
        """
        This function should use TwoLink.kinematic_map from the previous question together with
        the method Polygon.plot from Homework 1 to plot the manipulator.
        """
        [_, polygon1_transf, polygon2_transf] = self.kinematic_map(theta)
        polygon1_transf.plot(color)
        polygon2_transf.plot(color)

    def is_collision(self, theta, points):
        """
        For each specified configuration, returns  True if  any of the links of the manipulator
        collides with  any of the points, and  False otherwise. Use the function
        Polygon.is_collision to check if each link of the manipulator is in collision.
        """

        flag_theta = [False] * theta.shape[1]

        for i in range(theta.shape[1]):
            config = np.vstack(theta[:, i])
            [_, polygon1_transf, polygon2_transf] = self.kinematic_map(config)

            flag_points = polygon1_transf.is_collision(points)
            # Must logically reverse the array because Polygon.is_collision is
            # returning the incorrect (opposite) answer
            if True in np.logical_not(flag_points):
                flag_theta[i] = True

            flag_points = polygon2_transf.is_collision(points)
            # Must logically reverse the array because Polygon.is_collision is
            # returning the incorrect (opposite) answer
            if True in np.logical_not(flag_points):
                flag_theta[i] = True

        return flag_theta

    def plot_collision(self, theta, points):
        """
        This function should:
     - Use TwoLink.is_collision for determining if each configuration is a collision or not.
     - Use TwoLink.plot to plot the manipulator for all configurations, using a red color when the
    manipulator is in collision, and green otherwise.
     - Plot the points specified by  points as black asterisks.
        """
        collisions = self.is_collision(theta, points)
        for i, is_collision in enumerate(collisions):
            if is_collision:
                color = 'r'
            else:
                color = 'g'

            self.plot(np.vstack(theta[:, i]), color)
            plt.scatter(points[0, :], points[1, :], c='k', marker='*')

    def jacobian(self, theta, theta_dot):
        """
        Implement the map for the Jacobian of the position of the end effector with respect to the
        joint angles as derived in Question~ q:jacobian-effector.
        """
        vertex_effector_dot = np.zeros((2, theta.shape[1]))

        for i in range(theta.shape[1]):
            curr_theta = theta[:, i]
            curr_theta_dot = np.vstack(theta_dot[:, i])

            sin_theta_1 = math.sin(curr_theta[0])
            cos_theta_1 = math.cos(curr_theta[0])

            sin_theta_2 = math.sin(curr_theta[1])
            cos_theta_2 = math.cos(curr_theta[1])

            derivative_at_point = (5 * np.array(
                [[(-sin_theta_1 * cos_theta_2 - cos_theta_1 * sin_theta_2) -
                  sin_theta_1,
                  (-cos_theta_1 * sin_theta_2 - sin_theta_1 * cos_theta_2) +
                  cos_theta_1],
                 [(cos_theta_1 * cos_theta_2 - sin_theta_1 * sin_theta_2) +
                  cos_theta_1,
                  (-sin_theta_1 * sin_theta_2 + cos_theta_1 * cos_theta_2) +
                  sin_theta_1]])) @ curr_theta_dot

            vertex_effector_dot[:, i] = (derivative_at_point).T

        return vertex_effector_dot

    def jacobian_matrix(self, theta):
        """
        Compute the matrix representation of the Jacobian of the position of the end effector with
    respect to the joint angles as derived in Question~ q:jacobian-matrix.
        """
        sin_1 = math.sin(theta[0, 0])
        cos_1 = math.cos(theta[0, 0])

        sin_2 = math.sin(theta[1, 0])
        cos_2 = math.cos(theta[1, 0])

        jtheta = np.zeros((2, 2))

        jtheta[0, 0] = 5 * (-sin_1 * cos_2 - cos_1 * sin_2) - (5 * sin_1)
        jtheta[0, 1] = 5 * (-cos_1 * sin_2 - sin_1 * cos_2) + (5 * cos_1)
        jtheta[1, 0] = 5 * (cos_1 * cos_2 - sin_1 * sin_2) + (5 * cos_1)
        jtheta[1, 1] = 5 * (-sin_1 * sin_2 + cos_1 * cos_2) + (5 * sin_1)

        return jtheta


class TwoLinkPotential:
    """ Combines attractive and repulsive potentials """
    def __init__(self, world, potential):
        """
        Save the arguments to internal attributes
        """
        self.world = world
        self.potential = potential
        self.robot = TwoLink()
        self.total_pot = pot.Total(world, potential)

    def eval(self, theta_eval):
        """
    Compute the potential U pulled back through the kinematic map of the two-link manipulator, i.e.,
    U(  Wp_ eff(  )), where U is defined as in Question~ q:total-potential, and   Wp_ eff( ) is the
    position of the end effector in the world frame as a function of the joint angles   = _1\\ _2.
        """

        # Transform the coordinates of the end effector into the world
        transf_end_effector, _, _ = self.robot.kinematic_map(theta_eval)

        # evaluate the potential at the coordinate in the world
        u_eval_theta = self.total_pot.eval(transf_end_effector)

        return u_eval_theta

    def grad(self, theta_eval):
        """
    Compute the gradient of the potential U pulled back through the kinematic map of the two-link
    manipulator, i.e.,  _   U(  Wp_ eff(  )).
        """
        # Transform the coordinates of the end effector into the world
        trans_end_effector, _, _ = self.robot.kinematic_map(theta_eval)

        # evaluate the potential gradient at the coordinate in the world
        grad_u_eval_theta = (
            self.total_pot.grad(trans_end_effector).T
            @ self.robot.jacobian_matrix(trans_end_effector)).T
        return grad_u_eval_theta

    def run_plot(self, planner_parameters):
        """
    This function performs the same steps as Planner.run_plot in Question~ q:potentialPlannerTest,
    except for the following:
     - In step  it:grad-handle:  planner_parameters['U'] should be set to  @twolink_total, and
    planner_parameters['control'] to the negative of  @twolink_totalGrad.
     - In step  it:grad-handle: Use the contents of the variable  thetaStart instead of  xStart to
    initialize the planner, and use only the second goal  x_goal[:,1].
     - In step  it:plot-plan: Use Twolink.plotAnimate to plot a decimated version of the results of
    the planner. Note that the output  xPath from Potential.planner will really contain a sequence
    of join angles, rather than a sequence of 2-D points. Plot only every 5th or 10th column of
    xPath (e.g., use  xPath(:,1:5:end)). To avoid clutter, plot a different figure for each start.
        """
```

me570_hw3.py

```
"""
Defines a module to test general methods in other files
"""
import matplotlib.pyplot as plt
import numpy as np
import me570_geometry as gm


def sphere_testCollision():
    """
Generates one figure with a sphere (with arbitrary parameters) and  nb_points=100 random points that
are colored according to the sign of their distance from the sphere (red for negative, green for
positive). Generates a second figure in the same way (and the same set of points) but flipping the
sign of the radius  r of the sphere. For each sampled point, plot also the result of the output
pointsSphere.
    """
    plt.figure()

    center = np.vstack((0, 0))
    radius = 3
    influence_dist = 1
    sphere = gm.Sphere(center, radius, influence_dist)

    test_points = np.random.uniform(-radius * 2, radius * 2, size=(2, 100))

    collision_x = []
    collision_y = []
    non_collision_x = []
    non_collision_y = []
```

```python
    flag_points = sphere.is_collision(test_points)
    for i, point in enumerate(test_points.T):
        x_point = point[0]
        y_point = point[1]
        if flag_points[i]:
            collision_x.append(x_point)
            collision_y.append(y_point)
        else:
            non_collision_x.append(x_point)
            non_collision_y.append(y_point)

    sphere.plot('k')
    plt.scatter(collision_x, collision_y, color='r', zorder=10)
    plt.scatter(non_collision_x, non_collision_y, color='g', zorder=10)

    plt.show()

    plt.figure()
    collision_x.clear()
    collision_y.clear()
    non_collision_x.clear()
    non_collision_y.clear()
    flag_points.clear()

    sphere.flip()

    flag_points = sphere.is_collision(test_points)
    for i, point in enumerate(test_points.T):
        x_point = point[0]
        y_point = point[1]
        if flag_points[i]:
            collision_x.append(x_point)
            collision_y.append(y_point)
        else:
            non_collision_x.append(x_point)
            non_collision_y.append(y_point)

    sphere.plot('k')
    plt.scatter(collision_x, collision_y, color='r', zorder=10)
    plt.scatter(non_collision_x, non_collision_y, color='g', zorder=10)

    plt.show()


sphere_testCollision()

# attr_pot = pot.Attractive({'shape': 'quadratic', 'x_goal': np.vstack((0, 0))})

# f_handle_attr = lambda point: attr_pot.eval(point)
# gm.field_plot_threshold(f_handle_attr)

# my_sphere = gm.Sphere(np.vstack((0, 0)), -6, 5)
# repulsive_pot = pot.RepulsiveSphere(my_sphere)
# plt.figure()
# my_sphere.plot('g')
# plt.show()

# f_handle_repulsive = lambda point: repulsive_pot.eval(point)
# gm.field_plot_threshold(f_handle_repulsive, 6, 200)
# plt.show()

# world = pot.SphereWorld()
# f_handle_total = lambda point: world_potential.eval(point)

# potential_1 = {
#     'shape': 'quadratic',
#     'x_goal': np.vstack(world.x_goal[:, 0]),
#     'repulsive_weight': 0.01
# }
# world_potential = pot.Total(world, potential_1)

# plt.figure()
# gm.field_plot_threshold(f_handle_total, 10, 200)
# plt.show()

# potential_2 = {
#     'shape': 'quadratic',
#     'x_goal': np.vstack(world.x_goal[:, 1]),
#     'repulsive_weight': 0.01
# }

# world_potential = pot.Total(world, potential_2)

# plt.figure()
# gm.field_plot_threshold(f_handle_total, 10, 200)
# plt.show()

# potential_3 = {
#     'shape': 'conic',
#     'x_goal': np.vstack(world.x_goal[:, 0]),
#     'repulsive_weight': 0.01
# }

# world_potential = pot.Total(world, potential_3)
# plt.figure()
# gm.field_plot_threshold(f_handle_total, 10, 200)
# plt.show()

# potential_4 = {
#     'shape': 'conic',
#     'x_goal': np.vstack(world.x_goal[:, 1]),
#     'repulsive_weight': 0.01
# }

# world_potential = pot.Total(world, potential_4)

# plt.figure()
# gm.field_plot_threshold(f_handle_total, 10, 200)
# plt.show()

# my_potential_planner = pot.Planner()
# my_potential_planner.run_plot()

# world = pot.SphereWorld()
# goal_loc = world.x_goal
# potential = {
#     'x_goal': np.vstack(world.x_goal[:, 1]),
#     'shape': 'quadratic',
#     'repulsive_weight': 35
# }
# f_handle = lambda point: pot.clfcbf_control(point, world, potential)
# gm.field_plot_threshold(f_handle, 10, 20)
# world.plot()
# plt.show()

# total_potential = pot.Total(world, potential)
# total_pot_handle = lambda point: total_potential.eval(point)
# total_pot_grad_handle = lambda point: -total_potential.grad(point)

# plt.figure()
# gm.field_plot_threshold(total_pot_handle, 10, 200)

# plt.figure()
# gm.field_plot_threshold(total_pot_grad_handle, 10, 30)

# plt.show()
```