Criterion C - Development

1. **Spring forces (Appendix B.1)**

The forces() function in the Masspoint class calculates and applies forces to simulate spring-like behavior. It starts by computing a spring force that pulls the masspoint back toward its original position (origin). This is done by subtracting the current position from the origin, scaling it by the stiffness factor, and capping the magnitude to 1 to avoid instability. This simulates a linear spring with limited strength.

Next, it adds a damping force to reduce oscillation. The damping is applied in the direction opposite to the current velocity, scaled by both the velocity's magnitude and a damping constant. This acts like friction or air resistance, slowing down the motion over time so that the particles are more restricted.

2. **Self-intersection check (Appendix B.2)**

This code checks whether a polygon is self-intersecting by comparing each pair of non-adjacent edges to see if they cross, and if so, gently nudges the shape to try to fix it. It uses a helper function called ccw() that tests whether three points form a counter-clockwise turn, which is a common way to determine geometric orientation. Based on that, the segmentsIntersect() function checks whether two line segments intersect by seeing if their endpoints lie on opposite sides of each other's lines.

The main logic is in fixSelfIntersecting(), which loops through every non-adjacent pair of segments in the polygon and tests for intersections. If it finds one, it calculates the midpoints of the intersecting edges and nudges them apart. The function runs this process for a limited number of iterations (set by maxIterations). It's a lightweight way of maintaining valid, non-overlapping polygons by correcting problems as they arise.

3. **Volume calculator (Appendix B.3)**

This function calculates the 2D area of a polygon, even though it's named getVolume(). It starts by updating the positions of the polygon's points to ensure the calculation is based on the most recent coordinates. Then it iterates through each point in the polygon, pairing it with the previous one in the sequence. For each pair, it adds a specific term to a running total—this term is based on both the x and y values of the two points, structured to capture the orientation and spacing of the polygon's edges.

The method it uses is a version of the shoelace formula, a well-known algorithm for finding the area of a polygon given its vertices in order. Once the loop finishes summing up these contributions, the function takes the absolute value of half the result to get the final area. The output is then rounded to two decimal places. It's a reliable way to measure the size of any simple, closed polygon, regardless of its shape or number of vertices. But it's susceptible to self-intersection, which is why the previous function is so important.

4. **IO Parsing for Regions (Appendix B.4)**

This function, loadRegions(), reads raw string data from two text-based arrays—one containing lists of indices (regionIndices) and the other containing center point coordinates (centers). It loops through each entry in these arrays to construct the necessary inputs for creating Region objects. For each region, it parses all the numbers from the corresponding index string, converts them into integers, and stores them in an array. These integers represent which masspoints belong to that region.

Next, it extracts a pair of floating-point coordinates from the centers array using regex, and uses them to create a centerPos vector. This position represents the central location of the region in the sketch. Finally, it creates a new Region object using a default color, the list of masspoint indices, and the center position, and then adds that region to the global regions array.

The end result is a collection of structured region objects created from plain-text data.