Curvetopia - Documentation

Aaditya Vij *Paras Bisht*

Code -Walkthrough(Curvetopia_final)

The code begins by loading polyline data from a CSV file into two variables, `output_data1` and `output_data2`, using the `read_csv_` function. These polylines represent the paths to be processed. The first dataset, `output_data1`, is kept as the original data for later comparison. The second dataset, `output_data2`, undergoes a series of transformations.

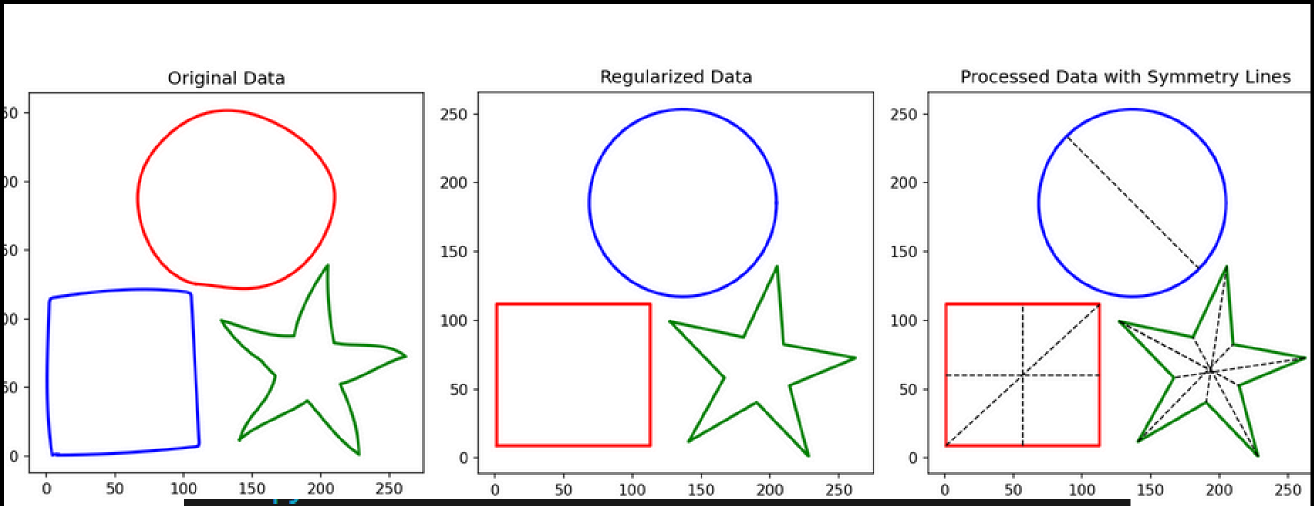
Initially, the code splits `output_data2` at sharp turns, making the paths easier to process. It then simplifies the polylines by reducing them to straight-line segments, which are easier to work with computationally. Afterward, it attempts to combine adjacent polylines into longer, coherent paths, followed by another simplification step to refine these merged polylines into straight lines. Collinear straight segments are further merged into longer lines to create a more streamlined dataset.

Next, the processed paths are classified into geometric shapes, such as circles, ellipses, rectangles, or other forms. The number of each detected shape is stored in the `shape_count` variable and printed out for analysis.

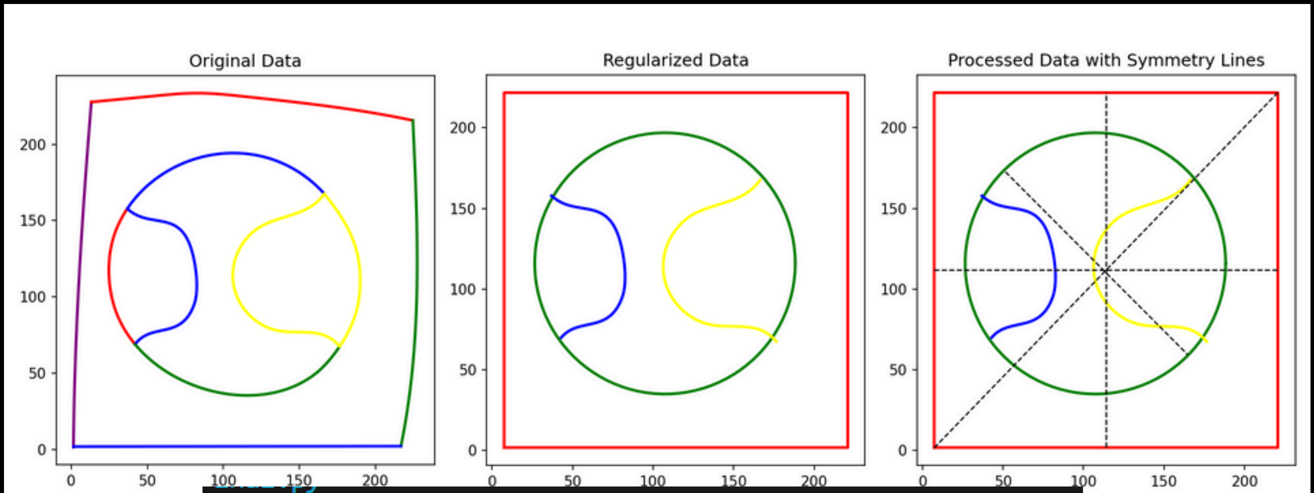
Finally, the code prepares a figure with two side-by-side plots. The first plot displays the original polylines (`output_data1`), while the second plot shows the processed polylines (`output_data2`) along with added symmetry lines for shapes like circles and rectangles. This visual comparison helps in understanding how the polylines were transformed through the processing steps. The figure is then displayed to the user, highlighting the differences between the original and processed data.

Additionally, the visual representation in the second plot not only highlights the changes made during processing but also emphasizes the importance of symmetry in geometric analysis. By adding symmetry lines to shapes such as circles and rectangles, the code provides a clear visual cue about the inherent symmetry of these forms. This enhances the understanding of how well the processed data conforms to ideal geometric shapes, and it also allows for easy identification of any deviations from perfect symmetry. Such insights can be particularly valuable in applications where precision and accuracy in shape recognition are critical, enabling further refinement or adjustments based on the observed results.

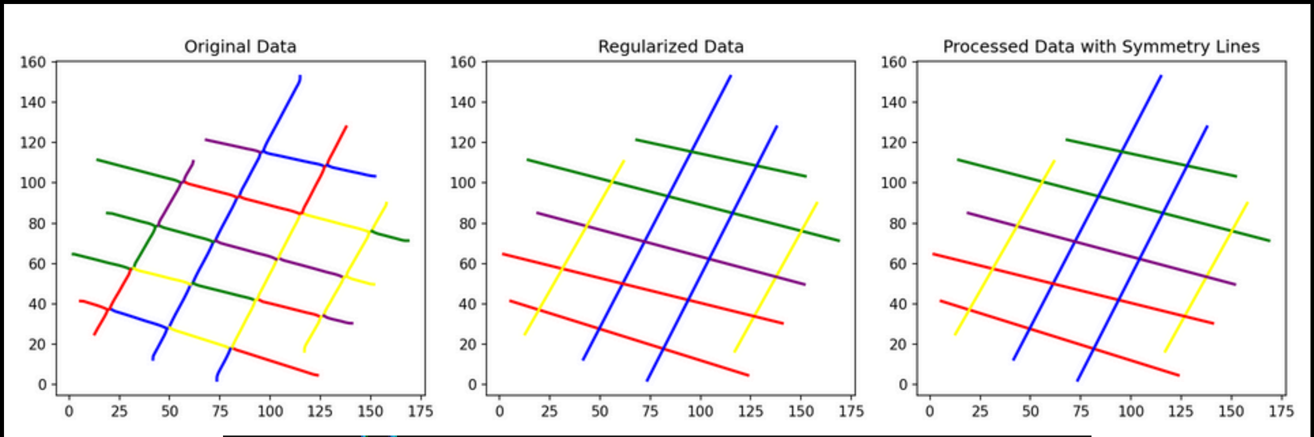
Outputs



```
Shape Counts: {'rectangle': 1, 'star': 1, 'circle': 1}
```



```
Shape Counts: {'rectangle': 1, 'circle': 1, 'other': 2}
```



```
Shape Counts: {'straight line': 9}
```

Curvetopia - Documentation

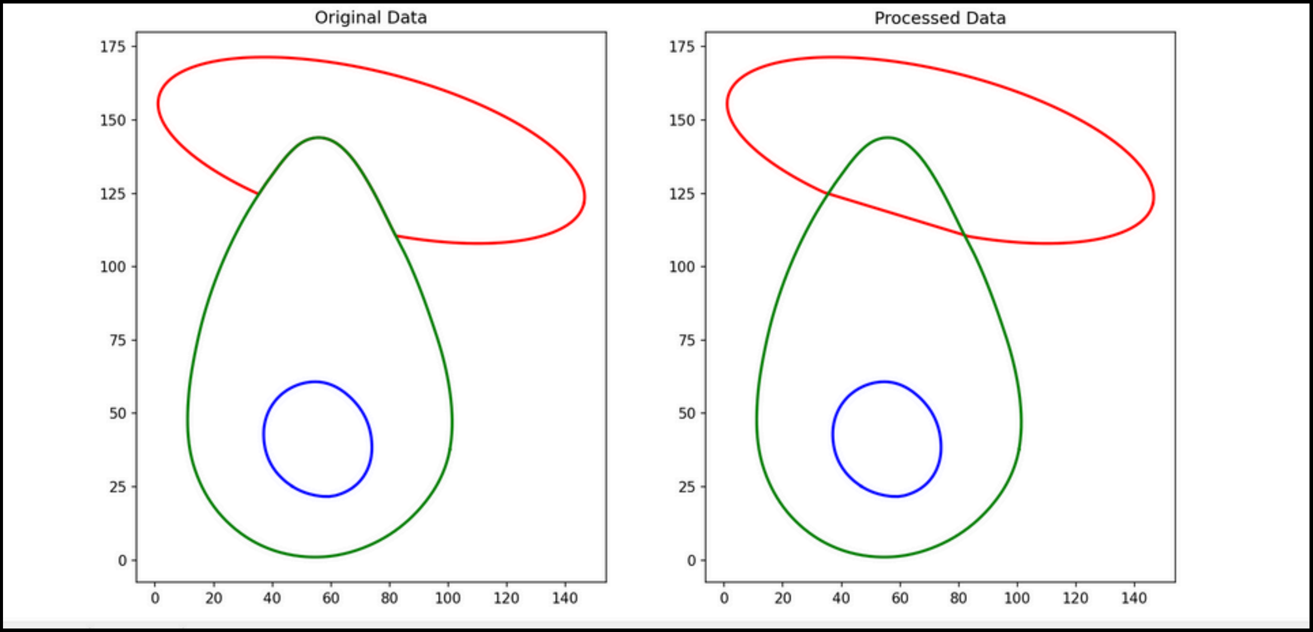
Aaditya Vij *Paras Bisht*

Code -Walkthrough(occlusion_partial)

This code is designed to process and visualize path data, with a focus on detecting and handling overlaps between paths. The `plot` function is used to visualize the paths on a graph, where different colors are assigned to distinguish between them.

The core functionality is provided by the `find_overlap` function, which identifies overlapping segments between two circular paths. It does this by comparing the proximity of points on these paths to determine if they overlap. To further refine the analysis, the `are_almost_parallel` function checks if vectors formed by consecutive points are nearly parallel, helping to identify sharp angles at specific points on the paths.

The main processing occurs in the `process_paths` function. Here, the code identifies overlapping segments with sharp angles and removes them, replacing the overlap with a straight line to simplify the path. After processing, the original and modified paths are plotted side by side for comparison, allowing for a clear visualization of the changes made.



Curvetopia - Documentation

Aaditya Vij *Paras Bisht*

The code is a geometric processing pipeline that reads polyline data from a CSV file, identifies sharp turns, splits polylines at these points, combines nearly straight segments, and visualizes the results. It leverages NumPy for numerical operations and Matplotlib for visualization. The code is modular, with functions handling specific tasks like angle calculation, distance measurement, and polyline manipulation. While it's well-structured and readable, it could benefit from better error handling, performance optimizations, and scalability improvements.

1. read_csv - This function reads a CSV file containing coordinates for different paths or polylines. The file is expected to have a specific structure, where each row includes a path identifier, a segment identifier, and the coordinates of the points in that segment.
The goal is to parse this data into a list of NumPy arrays, with each array representing the coordinates of a single path.

2. calculate_angle

- Purpose:
  - This function calculates the angle between two line segments that meet at a common point. Specifically, it calculates the angle formed at a vertex (point p2) by the segments p1p2 and p2p3.
- Step-by-Step Breakdown:
  a. Vector Formation:
    - Two vectors are created:
      - a = (p1[0] - p2[0], p1[1] - p2[1]): This is the vector from p2 to p1.
      - b = (p3[0] - p2[0], p3[1] - p2[1]): This is the vector from p2 to p3.

1. Dot Product Calculation:
    - The dot product of vectors a and b is calculated using the formula dot_product = a[0] * b[0] + a[1] * b[1]. The dot product gives a measure of how aligned the two vectors are.
2. Magnitude Calculation:
    - The magnitudes (lengths) of the vectors are calculated:
        - mag_a = math.sqrt(a[0] ** 2 + a[1] ** 2) for vector a.
        - mag_b = math.sqrt(b[0] ** 2 + b[1] ** 2) for vector b.
3. Cosine of the Angle:
    - The cosine of the angle between the two vectors is calculated using the dot product and magnitudes: cosine_angle = dot_product / (mag_a * mag_b).
    - The value of cosine_angle is clamped between -1 and 1 using max(min(cosine_angle, 1.0), -1.0) to avoid any numerical inaccuracies that could result in an invalid input for the acos function.
4. Angle Calculation:
    - The angle in radians is computed using angle = math.acos(cosine_angle), which is then converted to degrees using math.degrees(angle).
5. Edge Case Handling:
    - If either of the vectors has zero magnitude (indicating overlapping points), the function returns 0 degrees, implying no angle.

- Output: The function returns the angle in degrees formed between the two segments at the point p2.

3. plot Function

- Purpose:
    - This function visualizes the given paths (polylines) on a 2D plot. Each path is plotted in a different color to distinguish them.
- Step-by-Step Breakdown:
    - Color Selection:
        - A list of colors (colours) is defined, including options like red, green, blue, etc.

- .Plotting Each Path:
    - The function iterates over each path in paths_XYs. For each path:
        - A color is selected based on the path's index: c = colours[path_index % len(colours)]. This ensures that colors repeat if there are more paths than colors.
        - The path is plotted using ax.plot(XYs[:, 0], XYs[:, 1], c=c, linewidth=2), where XYs[:, 0] are the x-coordinates and XYs[:, 1] are the y-coordinates.
  - Plot Formatting:
    - The aspect ratio of the plot is set to equal using ax.set_aspect('equal') to ensure that distances are represented uniformly on both axes.
    - The plot is given a title using ax.set_title(title).
- Output: A 2D plot showing all the paths, each in a different color.

4. **split_polyline_at_sharp_turns**

- Purpose:
  - This function splits a polyline into multiple segments wherever there is a sharp turn. A sharp turn is defined as an angle below a certain threshold.
- Step-by-Step Breakdown:
  - Initialization:
    - The function initializes an empty list new_polylines to store the resulting segments after splitting.
  - Iterating Through Polylines:
    - The function iterates over each polyline in polylines.
    - For each polyline, it initializes a current_polyline list to accumulate points until a sharp turn is detected.
  - Detecting Sharp Turns:
    - The function iterates through the points in the polyline:
        - If enough points are available on both sides of the current point (i.e., the indices allow checking at i-interval and i+interval), it calculates the angle using calculate_angle.
        - If the calculated angle is less than the threshold_angle, it signifies a sharp turn.

- .Upon detecting a sharp turn:
    - The current polyline segment (excluding the last point) is added to new_polylines.
    - The current_polyline is reinitialized to start from the last two points (to ensure continuity), and the loop index is advanced by interval to skip over the sharp turn.

  a. Adding Remaining Points:
    - After exiting the loop, the last segment of the polyline (or the entire polyline if no sharp turns were detected) is added to new_polylines.

  b. Returning the Result:
    - The function returns new_polylines, which contains the original polyline split into segments at sharp turns.
- Output: A list of polylines, each representing a segment of the original polyline split at sharp turns.

5. **point_line_distance**

- Purpose:
    - This function calculates the perpendicular distance from a point to a line segment. It is used to determine how far a point is from the straight line connecting two other points.
- Step-by-Step Breakdown:

  a. Edge Case Handling:
    - If the two points forming the line segment (p1 and p2) are the same, the function returns the Euclidean distance between the point p and p1 (or p2), as there's no line segment to measure against.

  b. Distance Calculation:
    - The function calculates the perpendicular distance using the formula: distance= ∣ (p2−p1)×(p1−p) ∣ length of p2−p1\text{distance} = \frac{|(p2-p1) \times (p1-p)|}{\text{length of } p2-p1}distance=length of p2−p1 ∣ (p2−p1)×(p1−p) ∣ where p2-p1 is the vector along the line segment and p1-p is the vector from p1 to the point p. The cross product gives the area of the parallelogram formed by these vectors, and dividing by the length of p2-p1 gives the height (perpendicular distance).

6. **simplify_to_straight_lines**

- Purpose:
    - This function simplifies polylines by reducing segments that are almost straight lines to their endpoints. If a polyline is close to a straight line (within a given distance threshold), it is replaced by a line connecting its endpoints.
- Step-by-Step Breakdown:
    a. Initialization:
        - The function initializes an empty list simplified_polylines to store the simplified polylines.

    b. Iterating Through Polylines:
        - For each polyline in polylines, it initializes a current_polyline with the first point.
        - The function then iterates through the points in the polyline.

    c. Distance Check:
        - For each point in the polyline:
            - The distance from the current point to the line segment formed by the current_polyline's first and last points is calculated using point_line_distance.
            - If the distance exceeds the distance_threshold, the polyline segment from the first point to the previous point is added to simplified_polylines, and current_polyline is reinitialized to start a new segment.

    d. Finalization:
        - After exiting the loop, the last segment of the polyline (or the entire polyline if no segments exceeded the distance threshold) is added to simplified_polylines.

    e. Returning the Result:
        - The function returns simplified_polylines, which contains the simplified polylines.
- Output: A list of simplified polylines, where almost straight segments have been reduced to their endpoints.

7. euclidean_distance Function

- Purpose:
  - This function calculates the Euclidean distance between two points in a 2D space. It is used to measure the straight-line distance between two points.
- Step-by-Step Breakdown:
  a. Distance Calculation:
     - The function calculates the Euclidean distance using the formula: distance=(p1[0]−p2[0])2+(p1[1]−p2[1])2\text{distance} = \sqrt{(p1[0] - p2[0])^2 + (p1[1] - p2[1])^2}distance= (p1[0]−p2[0])2+(p1[1]−p2[1])2 This formula is derived from the Pythagorean theorem.
- Output: The function returns the Euclidean distance between points p1 and p2

8. **are_almost_parallel**

- Purpose:
  - This function checks whether two vectors are nearly parallel by comparing the angle between them with a specified threshold. It helps determine if two line segments are aligned.
- Step-by-Step Breakdown:
  a. Magnitude and Dot Product Calculation:
     - The function calculates the magnitudes of the vectors v1 and v2 and their dot product.
  b. Cosine of the Angle:
     - The cosine of the angle between the two vectors is computed using the dot product and magnitudes.
  c. Parallel Check:
     - The function checks if the absolute value of the cosine is close to 1 (within the cosine_threshold), which would indicate that the vectors are nearly parallel.
- Output: The function returns a boolean indicating whether the vectors are nearly parallel.

7. euclidean_distance Function
- Purpose:
  - This function calculates the Euclidean distance between two points in a 2D space. It is used to measure the straight-line distance between two points.
- Step-by-Step Breakdown:
  a. Distance Calculation:
    - The function calculates the Euclidean distance using the formula: distance=(p1[0]−p2[0])2+(p1[1]−p2[1])2\text{distance} = \sqrt{(p1[0] - p2[0])^2 + (p1[1] - p2[1])^2}distance= (p1[0]−p2[0])2+(p1[1]−p2[1])2 This formula is derived from the Pythagorean theorem.
- Output: The function returns the Euclidean distance between points p1 and p2

8. **are_almost_parallel**
- Purpose:
  - This function checks whether two vectors are nearly parallel by comparing the angle between them with a specified threshold. It helps determine if two line segments are aligned.
- Step-by-Step Breakdown:
  a. Magnitude and Dot Product Calculation:
    - The function calculates the magnitudes of the vectors v1 and v2 and their dot product.

  b. Cosine of the Angle:
    - The cosine of the angle between the two vectors is computed using the dot product and magnitudes.

  c. Parallel Check:
    - The function checks if the absolute value of the cosine is close to 1 (within the cosine_threshold), which would indicate that the vectors are nearly parallel.
- Output: The function returns a boolean indicating whether the vectors are nearly parallel.

9. **combine_straight_polylines**

- Purpose:
  - This function merges nearly straight and parallel polylines into longer segments. If two polylines are close and nearly parallel, they are combined into a single polyline.
- Step-by-Step Breakdown:
  a. Initialization:
     - The function initializes two lists: combined_polylines for storing the merged polylines and non_straight_polylines for those that cannot be merged.

  b. Merging Process:
     - The function iterates over the polylines, attempting to merge each with others in polylines_to_check:
       - It checks the proximity and parallelism of the polylines using euclidean_distance and are_almost_parallel.
       - If two polylines can be merged, they are combined into a single polyline.
       - If no merge is possible, the polyline is added to combined_polylines.

  c. Finalization:
     - The function adds any remaining unmerged polylines to combined_polylines.

  d. Returning the Result:
     - The function returns combined_polylines, which contains the merged polylines.
- Output: A list of merged polylines.

10. classify_shape(XYs)

Classifies a set of 2D points into specific geometric shapes based on their structure and arrangement.

- Parameters:
  - XYs: A list of 2D points representing the shape. The points should form a closed loop or polygon.
- Returns:
  - A string representing the classified shape. Possible values include:
    - "straight line": If the points form a straight line.
    - "circle": If the points approximately form a circle.
    - "ellipse": If the points approximately form an ellipse.
    - "rectangle": If the points approximately form a rectangle.
    - "regular_polygon": If the points approximately form a regular polygon.
    - "star": If the points approximately form a star shape.
    - "other": If the shape does not fit into any of the above categories.
    -

11. process_paths(paths_XYs, circle_tolerance=0.1, rect_angle_tolerance=10, min_points=50)

Processes a list of polylines by removing duplicate points, classifying each polyline into a shape, and generating standard shape approximations where applicable.

- Parameters:
  - paths_XYs: A list of polylines, each represented as a list of 2D points. Each polyline may represent a potential geometric shape.
  - circle_tolerance: A float representing the tolerance for approximating a circle (default is 0.1). It defines how close the shape needs to be to a perfect circle.
  - rect_angle_tolerance: A float representing the angle tolerance for approximating a rectangle (default is 10 degrees). It defines the allowable deviation from right angles.
  - min_points: An integer representing the minimum number of points required to perform shape classification (default is 50).

Returns:

- A tuple containing:
  - processed_paths: A list of processed polylines. Each polyline is transformed into a standard shape representation (e.g., circle, ellipse, rectangle) where applicable.
  - shape_count: A dictionary where keys are shape types and values are the count of each shape type found in the input.

12. **is_approx_rectangle(XYs, angle_tolerance=10)**

Determines if a set of 2D points approximates a rectangle by checking the angles between the sides and the number of vertices.

- Parameters:
  - XYs: A list of 2D points. The points should ideally form a closed shape with four sides.
  - angle_tolerance: A float representing the angle tolerance to consider the shape a rectangle (default is 10 degrees). This is the allowable deviation from 90-degree angles.
- Returns:
  - True if the points form an approximate rectangle based on angle checks; otherwise, False.
  -

13. make_rectangle(XYs)

Generates a list of 2D points representing a rectangle approximation based on the given points.

- Parameters:
  - XYs: A list of 2D points representing a shape that is approximately a rectangle.
- Returns:
  - A list of 2D points representing the corners of the approximated rectangle. The rectangle is centered at the centroid of the original points.

14. **add_symmetry_lines(ax, shape, XYs)**
Adds symmetry lines to a plot for various shapes including circles, ellipses, rectangles, and regular polygons. The symmetry lines are visual aids that highlight symmetrical properties of the shape.

- Parameters:
    - ax: A Matplotlib axis object to plot the lines on. This is used to draw the symmetry lines.
    - shape: A string representing the shape type. It determines which type of symmetry lines to add (e.g., "circle", "ellipse", "rectangle", "regular_polygon").
    - XYs: A list of 2D points representing the shape to which symmetry lines will be added.
- Returns:
    - None. This function directly modifies the provided plot (ax).
    -

15. **find_star_symmetry_lines(XYs)**
Finds and generates symmetry lines for a star shape by calculating lines from the centroid to each vertex of the star.

- Parameters:
    - XYs: A list of 2D points representing a star shape. The points should ideally be in a pattern that resembles a star.
- Returns:
    - A list of lines as pairs of 2D points. Each line represents a symmetry line extending from the centroid of the star to its vertices.
    -

16. plot_with_symmetry_lines(paths_XYs, title, ax)
Plots polylines and their corresponding symmetry lines based on their classified shapes. This function visualizes both the original and processed shapes along with symmetry lines on the same plot.

Parameters:

  - paths_XYs: A list of polylines, each represented as a list of 2D points. These are the shapes to be plotted.
  - title: A string representing the title of the plot.
  - ax: A Matplotlib axis object where the shapes and symmetry lines will be plotted.
- Returns:
  - None. This function modifies the provided plot (ax) and displays the processed shapes with symmetry lines.

17. **find_overlap(path1, path2, atol=0.1)**

Description:
Finds overlapping segments between two circular paths. Checks both forward and reverse overlaps, returning indices where the paths overlap.
Parameters:

- path1 (np.ndarray): An array of XY coordinates representing the first path.
- path2 (np.ndarray): An array of XY coordinates representing the second path.
- atol (float): The absolute tolerance for comparing points to determine if they are close enough to be considered overlapping (default is 0.1).

Returns:

- overlap_indices (list of tuples): A list where each tuple contains two tuples representing the start and end indices of overlapping segments in path1 and path2.

18. **is_sharp_angle(path, index)**

Description:
Determines if the angle at a specific point in a path is sharp. Sharp angles are detected by checking if the vectors on either side of the point are not nearly parallel.
Parameters:

- path (np.ndarray): An array of XY coordinates representing the path.
- index (int): The index of the point in the path where the angle is being checked.

Returns:

- bool: True if the angle is sharp, otherwise False.

19. **are_almost_parallel(vector1, vector2, tolerance=0.1)**
Description: The vectors are normalized, and their dot product is used to determine if they are nearly parallel within a specified tolerance.
Parameters:

- vector1 (np.ndarray): The first vector to compare.
- vector2 (np.ndarray): The second vector to compare.
- tolerance (float): The tolerance for determining if the vectors are almost parallel (default is 0.1).

Returns:

- bool: True if the vectors are almost parallel, otherwise False.

20. **remove_overlap_with_straight_line(path, overlap_start, overlap_end)**
Description: Removes overlapping segments from a path and replaces them with a straight line. Handles both normal and circular cases where the path wraps around.
Parameters:

- path (np.ndarray): An array of XY coordinates representing the path.
- overlap_start (int): The starting index of the overlap.
- overlap_end (int): The ending index of the overlap.

Returns:

- new_path (np.ndarray): The modified path with the overlap removed and replaced by a straight line.

21. process_paths(paths_XYs, angle_threshold=130)
Description: Processes a list of paths by removing overlapping portions where sharp angles are detected. The paths are adjusted to replace these overlaps with straight lines.
Parameters:

- paths_XYs (list of np.ndarray): A list of arrays where each array contains XY coordinates of a path.
- angle_threshold (int): The angle threshold to determine if an angle is sharp (default is 130 degrees).

Returns:

- paths_XYs (list of np.ndarray): The list of processed paths with overlaps removed.