CoarseMatching Model for UAV Localization in GNSS-denied Environments

Khưu Huỳnh Gia Hưng, Triệu Thiên Ân, Nguyễn Minh Quang,
Phạm Gia Hưng, Võ Nguyễn Bảo Ngọc
Supervised by Assoc. Prof. Ph.D. Phan Thành An
Ho Chi Minh City University of Technology

April 1st, 2025

Abstract: This paper presents a CoarseMatching method for UAV localization in GNSS-denied environments, proposed by Ouyang et al. Based on semantic segmentation and shape vector matching using a reference vector map, the method allows identification of the most similar map regions to UAV-captured images. In this report, we implement and evaluate the CoarseMatching algorithm proposed by Ouyang et al. to demonstrate its effectiveness via geometric example. Beyond reproducing the CoarseMatching baseline, this report conducts a focused investigation of two design factors: (i) the impact of using a single-radius versus three-radius shape vectors, and (ii) the effect of retaining only convex vertices after polygon simplification. These aspects are examined through controlled ablation experiments under a consistent evaluation protocol, enabling quantitative comparisons while preserving the original CFBVM formulation designed by Ouyang et al.
Keywords: UAV localization, shape vector, CoarseMatching, semantic segmentation, vector map.

1 Introduction

Accurate UAV localization in GNSS/GPS-denied environments is a critical challenge in many real-world applications such as search and rescue, military reconnaissance, and autonomous navigation. In the absence of satellite-based positioning, vision-based localization becomes a compelling alternative due to its independence from external signals and reliance only on visual input from onboard cameras.

Vision-based localization methods are generally classified into:

- Relative Visual Localization (RVL): where motion is estimated by comparing consecutive frames (e.g., SLAM, Visual Odometry), but errors accumulate over time.
- Absolute Visual Localization (AVL): which matches the current image against a pre-built map to estimate the UAV's absolute position.

Traditional AVL approaches use high-resolution satellite or aerial imagery for matching. However, this requires substantial memory and is computationally intensive, especially in large-scale scenarios. Lowering the resolution risks loss of feature fidelity and increases false matches. To overcome this, semantic Vector Maps (VMAP) offer a compact, structured, and memory-efficient alternative that encodes labeled geometric objects like buildings or roads.

Among recent advances, the Coarse-to-Fine Building Vector Matching (CFBVM) framework (Fig.1), proposed by Ouyang et al.[1] , combines the geometric strength of vector maps with semantic features to improve localization performance. It consists of two main stages:

- A CoarseMatching stage using geometric shape vectors to identify candidate areas in the map.
- A fine matching stage involving probabilistic refinement using particle filters based on Gaussian Mixture Models (GMM).

In this project, we implement and analyze the CoarseMatching stage as proposed in the CFBVM framework by Ouyang et al. [1], isolating it for experimental evaluation. Although the final refined position of the UAV is typically obtained through subsequent probabilistic filtering (e.g., fine matching with particle filters), the CoarseMatching stage plays a foundational role in narrowing down the search space and enabling real-time processing in large-scale maps. Specifically, we focus on the construction of building shape vectors and their comparison via similarity metrics to propose a set of candidate UAV positions on the reference semantic vector map. This process provides an initial estimation of where the UAV might be located and serves as the essential input to downstream refinement stages. In this work, we examine the CoarseMatching procedure proposed by Ouyang et al.[1], about it mathematical formulation, algorithmic design, and effectiveness—through controlled experiments using simplified building contours in a planar environment.

Beyond the baseline implementation, this report also investigates two focused design questions through controlled experiments. First, we ablate the radial granularity of the shape descriptor by comparing three single-radius settings against the full three-ring configuration, and we summarize results using standard top-k retrieval accuracy and confidence statistics, together with a city-scale runtime stress test over a 1 km radius around the Vietnam Nation University - Ho Chi Minh city, University of Technology library (open polygon dataset to be specified). Second, we examine vertex convexity: retaining only convex vertices after Douglas–Peucker simplification versus keeping all vertices. This study quantifies changes in candidate density, runtime, and matching reliability under the same protocol, and probes whether convex-only filtering mitigates false correspondences associated with concave corners, occlusions, or merged structures.

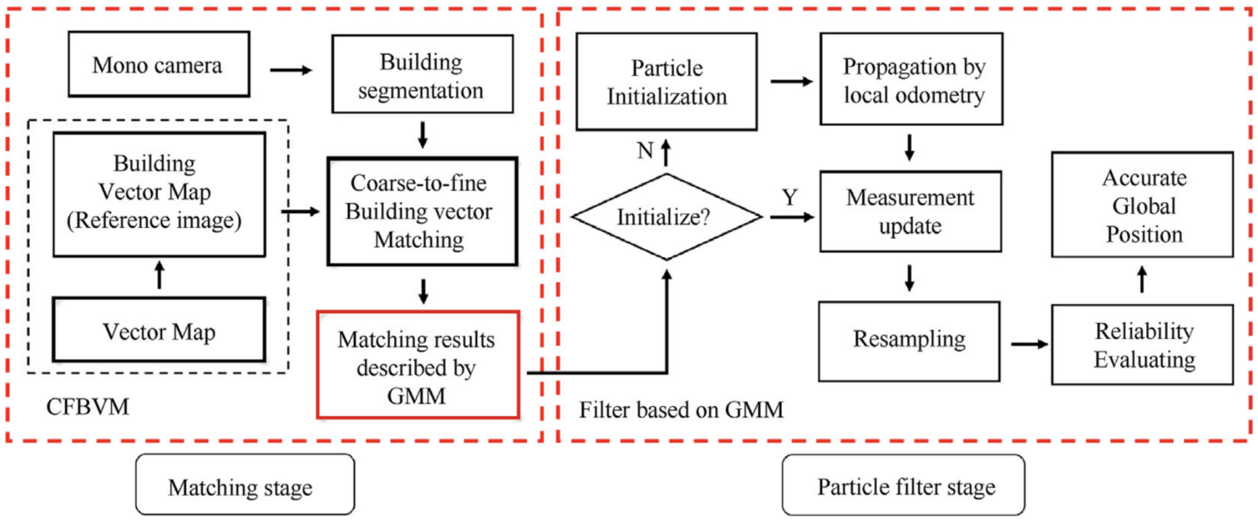

Fig. 1. Flow chart of the proposed localization approach (adapted from Ouyang et al., 2024 [1]).

1.1 Background and Motivation

When a UAV captures an image of its surrounding environment, the observed field-of-view typically covers only a small fraction (often $< 0.1\%$) of the full map. Identifying where this view corresponds on a massive map is computationally difficult, especially under storage and speed constraints.

The CoarseMatching stage in the CFBVM framework mitigates this issue by:

- Images captured by the camera are processed by a convolutional network (Mask R-CNN), trained on a dataset. This process is called semantic segmentation of buildings, where buildings are treated as semantic objects.
- After pixel-level semantic segmentation, the contours of the buildings are identified, Douglas - Peucker algorithm simplifying each building's contour to a small set of vertices.
- Encoding the spatial layout of nearby structures into a fixed-length shape vector, computed from concentric circular sectors around each vertex.
- Comparing shape vectors between the UAV image and the map to find best matches.
- Filtering mismatches early using a distance-based threshold rejection rule.

This step does not yield exact coordinates but significantly narrows down the candidate regions for further refinement. It is thus a critical component in the pipeline of visual-based UAV localization.

1.2 Problem Statement

1.2.1 Objective

Given a reference semantic vector map and a set of building contours extracted from a UAV-captured image, determine the best-matching vertices in the map based on shape vector similarity.This is achieved by computing and comparing shape vectors at building vertices to identify the most likely correspondence between the two sources.

1.2.2 Formal Problem Definition

TABLE 1
SYMBOLS AND DEFINITIONS

Symbol	Definition
Map_{ref}	The reference map.
$\{ABC...\}$	A set of vertex points A, B, C, ...
V_{cam}	A set of vertex points extracted from the segmented UAV image.
V_{ref}	A set of vertex points from the reference vector map, each with a precomputed shape vector.
f^X	Shape vector at vertex X, consisting of 24 sector values and a total area S_0.
$\text{dis}(f^A, f^B)$	Distance between shape vectors f^A and f^B.
δ	Early rejection threshold for CoarseMatching (default value is 0.2).
Match	Mapping from each $X_i \in V_{\text{cam}}$ to $Y_j \in V_{\text{ref}}$ with the smallest shape vector distance.
r1, r2, r3	Radii of concentric circles used in shape vector construction.
S_{ij}^X	Area of buildings within the sector j of ring i centered at vertex X.
S_0	Total area across all 24 sectors around a vertex.

Let:

- $V_{cam} = \{X_1, X_2, ..., X_m\}$ be a set of vertices extracted from a UAV's segmented image.
- $V_{ref} = \{Y_1, Y_2, ..., Y_n\}$ be a set of all vertices in the reference vector map, each pre-associated with a shape vector f_j^Y.

Determine a mapping from each $X_i \in V_{cam}$ to a $Y_j \in V_{ref}$ such that the shape vector distance $\text{dis}(f_{Xi}, f_{Yj})$ is minimized under early rejection constraints.

Input:

- UAV image with segmented building contours $\rightarrow$ simplified into vertex set V_{cam}
- Reference map Map_{ref} containing precomputed shape vectors at each vertex in V_{ref}
- Parameters: 3 radii $r_1 < r_2 < r_3$, and rejection threshold δ

Output:

A mapping Match = $\{X_i \rightarrow Y_j\}$ for each vertex $X_i \in V_{cam}$ to its best-matching one $Y_j \in V_{ref}$

1.2.3 Algorithmic Solution

To solve this problem, we implement a two-procedure solution:

- Procedure 1: BuildShapeVector, which constructs a local shape descriptor at each vertex by computing the distribution of building areas within concentric rings and angular sectors.
- Procedure 2: CoarseMatching, which performs vertex-level matching by comparing shape vectors between the UAV-captured image and the reference map.

These procedures form the core of the proposed CoarseMatching stage by Ouyang et al. [1] and are described in detail in Sections 2.4.1 and 2.4.2, respectively. In this section, we provide a brief summary of the algorithm's structure. The full pseudocode, input/output specification, and implementation details are presented later to ensure technical reproducibility and facilitate in-depth analysis.

Procedure 1: BuildShapeVector(P, r_1, r_2, r_3)

Input: A vertex B, and radii $r_1 < r_2 < r_3$

Output: A 25-dimensional shape vector $f^P = \{S_{11}, ..., S_{38}, S_0\}$

Description: Divide the area around P into 3 concentric rings and 8 directional sectors (total 24 sectors), compute building area in each, and summarize with total area S_0

Procedure 2: CoarseMatching($V_{cam}, V_{ref}, r_1, r_2, r_3, \theta$)

Input: Sets of vertices $\in V_{cam}$, vertices $\in V_{ref}$, shape vector parameters, and threshold δ

Output: Matching $X_i \leftarrow Y_j$ for each vertex

Description: For each vertex $\in V_{cam}$:

- Compute its shape vector
- Compare with each vertex $\in V_{ref}$ (after early rejection)
- Select the best match with smallest distance

1.2.4 Correctness & Time Complexity

The algorithm is guaranteed to terminate, as both procedures involve bounded loops over finite sets.

Each shape vector computation is deterministic based on known geometry.

Time Complexity (for M vertices $\in V_{cam}$ and N vertices $\in V_{ref}$):

- BuildShapeVector: O(M+N) (O(1) per vertex)
- CoarseMatching: O(MN)

2 Methodology

2.1 Polygon Simplification

Ouyang et al.[1] use the Douglas-Peucker algorithm to simplify polygons or polylines in two-dimensional space. The goal of this algorithm is to reduce the number of points in a polygon or polyline while retaining its overall shape as accurately as possible. This algorithm is commonly used for geometric optimization.

In the context of this experiment, the Douglas–Peucker algorithm is applied to extract a reduced set of significant vertices from each polygonal building contour. These simplified vertex sets serve as the input for the subsequent shape vector construction stage in the CoarseMatching pipeline. By minimizing redundancy in vertex representation while preserving the overall geometry, the algorithm ensures computational efficiency and geometric consistency during the matching process.

The details are provided in Appendix A.

2.2 Shape Vector construction

First, Ouyyang et al. calculate the shape vector for each point. The Shape Vector of a point encodes the area of buildings around that point within a specific region, bounded by three circles with radii $r_1 < r_2 < r_3$, where r_3 does not exceed the image's range. Specifically, a point consists of a set of shape vectors; SA_0 is the sum of the values of the remaining shape vector components.

$$f^A = \{S_0^A, S_{11}^A, S_{12}^A, ..., S_{38}^A\},\ S_0^A = \sum_{i=1}^{3} \sum_{j=1}^{8} s_{ij}^A, i = 1,2,3; j = 1,2,...8. \tag{1}$$

2.3 Distance and threshold formula

In this formula, there are three circles, corresponding to i ranging from 1-3, divided into 8 directions (j from 1-8), resulting in 24 sectors. Since Ouyang et al.[1] are in the "matching" stage, they compare images captured by the UAV with the reference map to determine the UAV's position.

Next, the distance between two shape vectors is defined as:

$$\mathrm{dis}(f^A, f^B) = \frac{1}{m}\sum_{i=1}^{3}\sum_{j=1}^{8}|s_{ij}^A - s_{ij}^B|,\ m = \pi(r_3)^2 \tag{2}$$

According to the formula, Ouyang et al.[1] compute each part $|S_{ij}^a - S_{ij}^B|$ for all 24 parts. The smaller the distance between two shape vectors, the more similar the distribution of building areas around the two points, indicating a higher likelihood of the points matching.

The CoarseMatching model is defined by the following formula:

$$dis(\{f^A f^B f^C\}, \{f^{A'} f^{B'} f^{C'}\}) = min(dis(f^X f^{X'})),\ X \in \{A, B, C\} \tag{3}$$

In Formula 3, Ouyang et al.[1] state: "{ABC. . . } and {A'B'C'...} are the vertex points in the aerial image and the reference map", meaning: " {ABC. . . } and {A'B'C'...} are the given sets of points captured by the UAV and the reference map, respectively." Ouyang et al.[1] also note that X belongs to the set A, B, C. From this, it can be inferred that X' belongs to the set containing A', B', C'.

Formula 3 provides the distance between the shape vectors computed by the UAV from the captured images and the shape vectors computed from the reference map. When comparing, a point and its neighboring points in the UAV-captured image are compared with the reference map.

When the similarity between a point in the image and a point in the reference map is identified, that point and its neighboring points are continuously compared with the nearest point in the corresponding position in the reference map. The smallest result (the smallest distance) is recorded in the system. The smallest result here reflects the highest degree of similarity between the location in the image and the location on the map. This is the coarse filtering step.

To reduce computational costs for unnecessary incorrect points, Ouyang et al.[1] include an "early rejection" step:

$$|S^A - S^B|/m < \delta,\ \text{with}\ \delta = 20\%$$

S^A and S^B are S_0^A and S_0^B in Formula (1). Essentially, S^A and S^B are S^A and $S^{A'}$ because this formula is used to compare the two values S_0^A and $S_0^{A'}$ belonging to the sets f^X and $f^{X'}$, where A and A' are vertices from the captured image and the given reference map, respectively.

2.4 Algorithmic solution

2.4.1 BuildShapeVector Algorithm

BuildShapeVector constructs the shape vector at a vertex P, taken from V_{cam}. This shape vector is later used in Procedure 2: CoarseMatching for candidate selection in the reference map.

Input: vertex P, 3 radii $r_1 < r_2 < r_3$

Output: 25-dimensional vector $f^P = \langle S_{11}, \ldots, S_{38}, S_0 \rangle$

Algorithm 1: BuildShapeVector (P , r_1 , r_2 , r_3)

```
1. for i ← 1 to 3
2.     for j ← 1 to 8
3.         $S_{ij} \rightarrow$ area of buildings in sector (i, j) centered at P
4. $S_0 \leftarrow \sum_{i=1}^{3}\sum_{j=1}^{8} S_{ij}$
5. return $\langle S_{11}, \ldots, S_{38}, S_0 \rangle$
```

Lemma 1.0

Given a point P in R^2 and three radii $r_1 < r_2 < r_3$, the BuildShapeVector algorithm always returns a unique 25-dimensional shape vector.

Proof We analyze each step of the BuildShapeVector algorithm

1. Fixed Input:

The function takes as input:

- A point P in 2D space
- Three radii r_1, r_2, r_3 in increasing order

These parameters are fixed and do not change during the function's execution.

2. Space Partitioning is Deterministic:

- For each radius r_i (i $\in \{1,2,3\}$), the space around P is divided into 8 sectors, each with a 45° angle.
- There are 24 sectors in total, uniquely identified by indices (i,j) with i = 1→8, j = 1→8.

3. Area Calculation is Deterministic:

- In each sector (i,j), the algorithm calculates the area of buildings within that region.
- Buildings are clearly defined in the map (GIS or geometric data), and computing the intersection between a building's polygon and a sector is an exact geometric operation.
- Thus, the result S_{ij} is deterministic.

4. Summing Areas S_0 is a Simple Addition: $\sum_{i=1}^{3}\sum_{j=1}^{8} S_{ij}$

This is the sum of 24 known values, entirely deterministic. ■

2.4.2 CoarseMatching Algorithm

As originally formulated by Ouyang et al.[1], the CoarseMatching algorithm identifies the best-matching reference vertex for each UAV-observed vertex based on shape vector similarity. It compares shape vectors between vertices $\in V_{cam}$ and vertices $\in V_{ref}$, using an early rejection threshold to discard dissimilar candidates and select the closest match.

Input: $V_{cam} = \{X_1 \dots X_M\}$, $V_{ref} = \{Y_1 \dots Y_N\}$, $r_1 < r_2 < r_3$, $\delta \in (0,1)$

Output: Match : mapping $\{X_i \leftarrow Y_j\}$

Algorithm 2: CoarseMatching (V_{cam} , V_{ref} , r_1 , r_2 , r_3 , δ)

```
1.  m ← $\pi \cdot r_3^2$
2.  for each $Y_j \in V_{ref}$
3.          $f^{Y_j}$ ← BuildShapeVector($Y_j$ , $r_1$ , $r_2$ , $r_3$)
4.  for each $X_i \in V_{cam}$
5.          $f^{X_i}$ ← BuildShapeVector($X_i$ , $r_1$ , $r_2$ , $r_3$)
6.          bestDist ← +∞ ; bestY ← NIL
7.          for each $Y_j \in V_{ref}$
8.              if | $S_0^{X_i} - S_0^{Y_j}$ | / m ≥ $\delta$
9.                  continue
10.             d ← dist($f^{X_i}$, $f^{Y_j}$)
11.             if d < bestDist
12.                 bestDist ← d
```

```
13.                 bestY ← $Y_j$
14.         Match[$X_i$] ← bestY
15. return Match
```

Theorem 1 (Correctness of CoarseMatching)

The CoarseMatching algorithm maps each point $X_i \in V_{cam}$ to a point $Y_j \in V_{ref}$ such that:

- The difference between the two points does not exceed the threshold θ.
- Y_j is the point with the closest feature vector to X_i in the valid candidate set.

Proof To prove Theorem 1, we use the following loop invariant:

Loop Invariant: For each iteration over $Y_j \in V_{ref}$ for a point $X_i \in V_{cam}$, the pair (bestDist, bestY) always stores the smallest current distance between f^X and all f^Y considered that satisfy the threshold condition:

$$|S_0^{X_i} - S_0^{Y_j}|/m < \delta, \text{ with } \delta = 20\%$$

Initialization: Before starting the loop over each Y_j, we assign:

- bestDist ← $+\infty$
- bestY ← NIL

Maintenance: For each Y_j:

- If the δ threshold is not satisfied, skip.
- If satisfied, compute distance d = dist(f^{X_i}, f^{Y_j}).
- If d < bestDist, update bestDist ← d, bestY ← Y_j.

Thus, at every iteration, the closest candidate up to that point is retained. → The invariant is maintained.

Termination: After the loop ends:

- All $Y_j \in V_{ref}$ have been considered.
- bestY is the point in the threshold-satisfying set with the smallest distance to X_i.
- Assigning Match[X_i] ← bestY is the correct mapping as required by the algorithm. ■

2.4.3 Complexity Analysis

Build Shape Vector Phase (Procedure 1: BuildShapeVector Algorithm)

For each vertex $\in V_{cam}$ (M vertices) and each vertex $\in V_{ref}$ (N vertices), a shape vector with 24 components is computed. Since the computation for each vertex is constant time (fixed number of operations), the complexities are:

$$\text{Time complexity}: \ O(M+N), \ \text{Space complexity}: \ O(M+N).$$

Matching Phase (Procedure 2: CoarseMatching Algorithm)

Each of the M shape vectors from the UAV image is compared to all N shape vectors in the reference map, so in this state we loop through all of N shape vectors for each of the M shape vectors. Since the vector length is constant (24 values), each comparison takes O(1) time. At the same time, we create a new array called "Matching" that contains M values, resulting in:

$$\text{Time complexity}: \ O(MN), \ \text{Space complexity}: \ O(M).$$

3 Simulation of CoarseMatching Algorithm

3.1 Experiment Setup

3.1.1 Scope of the simulation

This experiment simulates the CoarseMatching process using a synthetic map designed entirely in AutoCAD. The environment consists of 10 building polygons, each with a custom number of vertices. Among them, a subset of 5 vertices — defined as those whose surroundings are fully observable within radius r_3 — is selected to form the V_{cam} subset, simulating a partial observation scenario.

The objective is to match these 5 vertices $\in V_{cam}$ to the most similar ones among the 10 $\in V_{ref}$ available in the reference map, based on shape vector similarity. All area values required for shape vector construction are calculated directly using AutoCAD's "Area" tool.

For simplicity, this experiment assumes the UAV camera is oriented perpendicularly to the ground plane, allowing a top-down 2D projection of the building contours.

The size of the UAV image frame is 500x350 (unit). The C++ CoarseMatching implementation (source code) is provided in Appendix D.1.

The figure below illustrates a sample reference map containing building contours along with concentric circular sectors centered at each vertex. These circular regions are used in the shape vector construction process, where geometric properties such as area distribution are extracted for each direction segment. This mechanism is used for both UAV image and reference map shape vector calculation.

This visualization is intended to give readers an overview of the spatial setup and geometric encoding mechanism. In the following sections (Topology and matching results of each map), only the labeled building contours will be shown without the circular overlays, in order to reduce visual clutter and focus on the vertex identifiers.

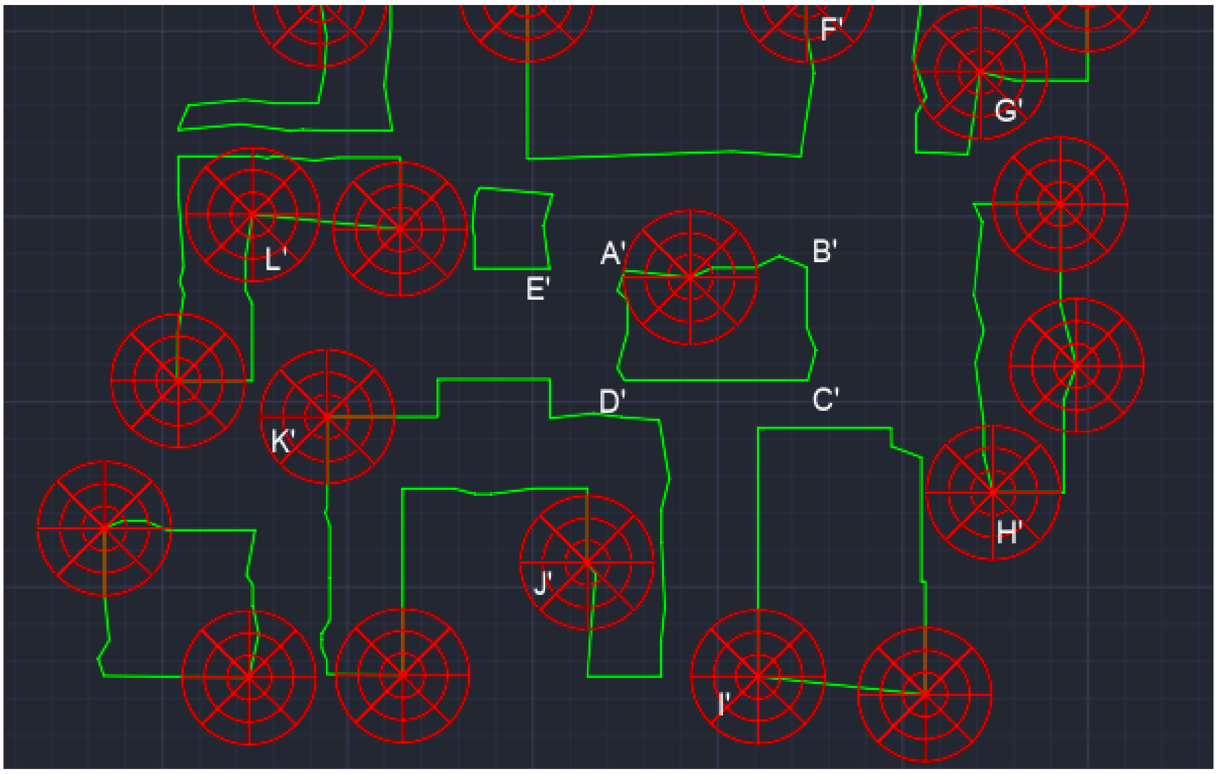

Fig. 2.

3.1.2 Building Color Notation

Image notes (The simulated map was created using design software: AutoCAD):

- The orange frame represents the size of the image captured by the UAV.
- Colored circles are used to compute the Shape Vector at vertices:

○ Red: Vertex extracted from the reference map.

○ Purple: Vertex extracted from the UAV-captured image.

○ Green buildings are from the reference map. Blue buildings are identified in the UAV-captured image.

3.2 Topology and Matching Results of Map 1

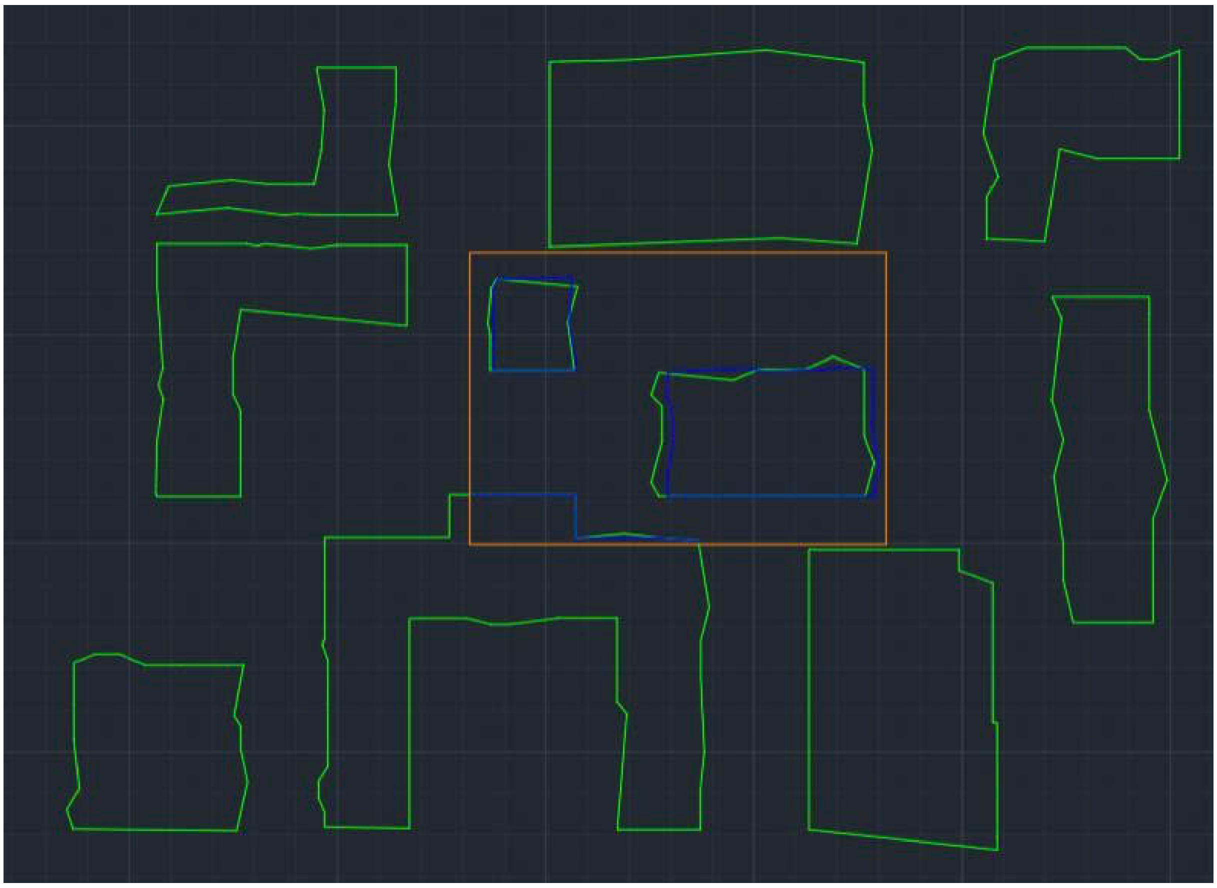

Fig. 3.a

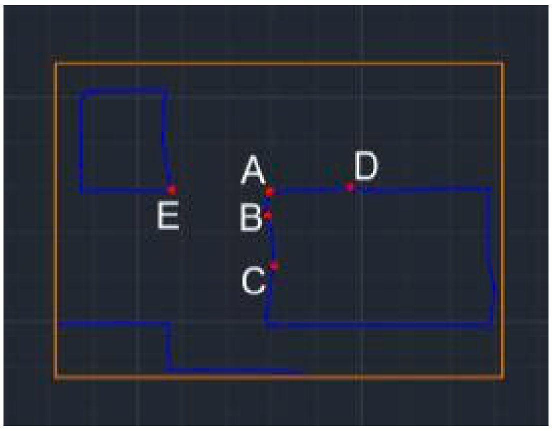

Fig. 3.b

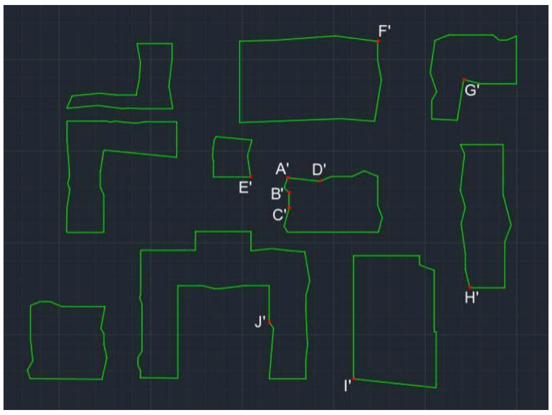

Fig. 3.c

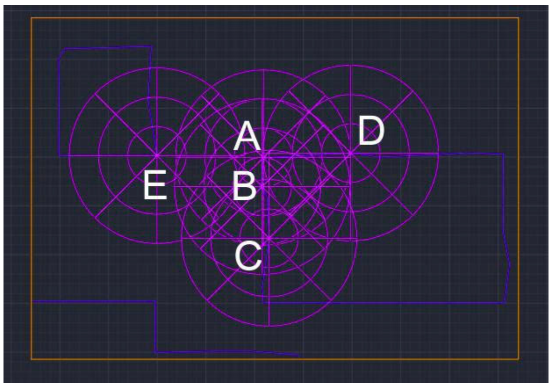

Fig. 3.d

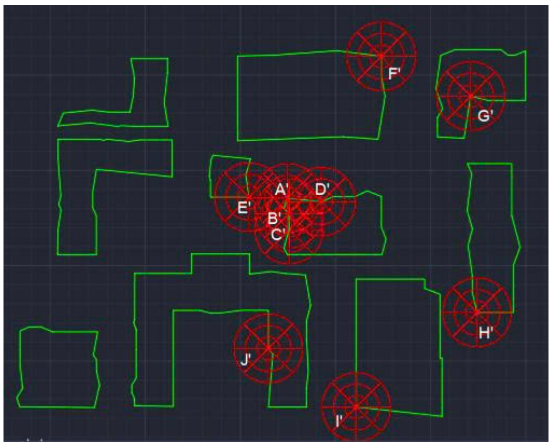

Fig. 3.e

To demonstrate the effectiveness of the CoarseMatching algorithm proposed by Ouyang et al [1], we conducted a test using five input buildings labeled as points A, B, C, D, and E extracted from a simplified UAV image. These were compared against ten candidate vertices in the reference map, labeled A' through J'. For each pair, the Euclidean distance between their shape vectors was computed.

Table 2 and Table 3 display the distance matrices between each input point (A–E) and all candidates (A'–J'), illustrating how the closest match is selected based on the minimum shape vector distance. The shape vector calculation is summarized at appendix B.

Result:

TABLE 2
SHAPE VECTOR DISTANCES BETWEEN VERTICES $\in V_{cam}$ AND VERTICES $\in V_{ref}$

	A′	B′	C′	D′	E′	F′	G′	H′	I′	J′
A	3.3090	51.6207	29.8540	31.8600	48.5482	51.1917	91.0470	51.3743	46.5766	26.2015
B	59.3355	13.1254	22.4157	35.8396	56.0007	58.9065	81.4929	41.5579	36.7602	16.3851
C	32.1298	72.3435	1.6354	52.6515	68.8206	71.9145	62.4885	24.7460	22.3162	8.2420
D	25.1231	26.8570	52.3834	8.0932	71.8324	26.4280	74.6539	76.3472	71.5495	52.9883
E	46.1481	45.8951	68.2292	73.6120	0.6736	46.4204	51.4339	44.8115	49.6092	69.9843

TABLE 3
MATCHING RESULT

Vertex $\in V_{cam}$	Vertex $\in V_{ref}$
A	A'
B	B'
C	C'
D	D'
E	E'

3.3 Topology and Matching Results of Map 2

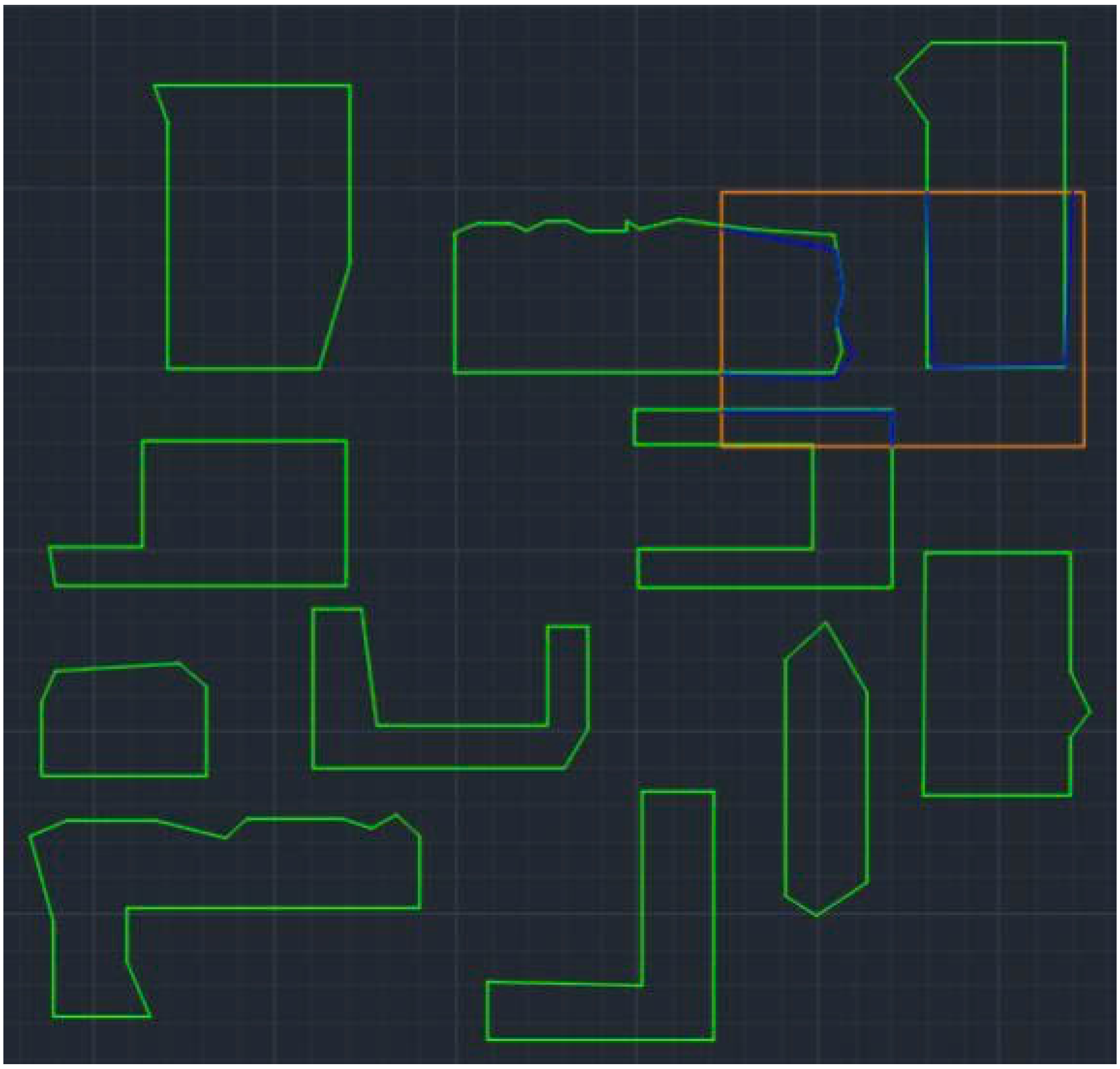

Fig. 4.a

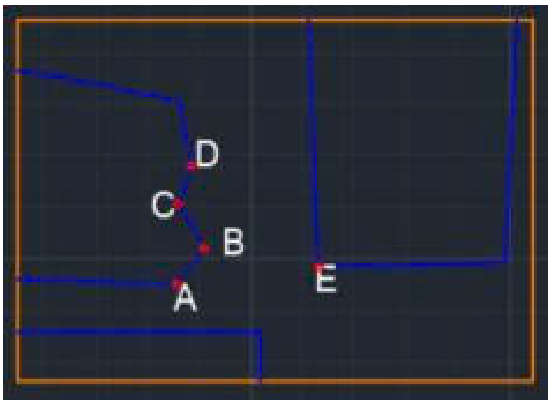

Fig. 4.b

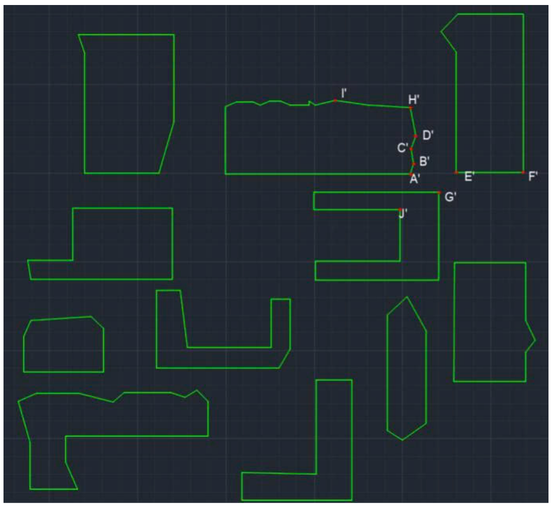

Fig. 4.c

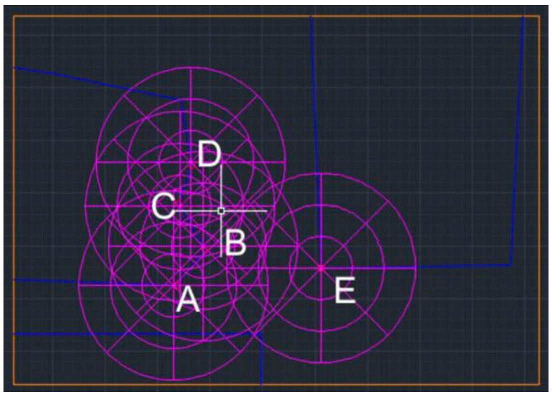

Fig. 4.d

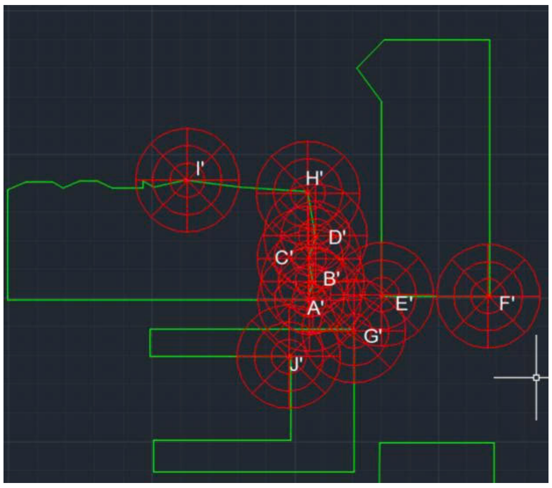

Fig. 4.e

To demonstrate the effectiveness of the proposed CoarseMatching algorithm (adapted from Ouyang et al.[1]), we conducted a test using five input buildings labeled as points A, B, C, D, and E extracted from a simplified UAV image. These were compared against ten candidate vertices in the reference map, labeled A' through J'. For each pair, the Euclidean distance between their shape vectors was computed.

Table 4 and Table 5 display the distance matrices between each input point (A–E) and all candidates (A'–J'), illustrating how the closest match is selected based on the minimum shape vector distance. The shape vector calculation is summarized at appendix B.

Result:

TABLE 4
SHAPE VECTOR DISTANCES BETWEEN VERTICES $\in V_{cam}$ AND VERTICES $\in V_{ref}$

	A'	B'	C'	D'	E'	F'	G'	H'	I'	J'
A	6.4420	27.8883	28.4716	30.9550	62.8757	22.8512	52.1993	49.5935	56.7371	47.8331
B	29.1382	6.7874	21.1620	14.8164	56.5993	17.2312	35.4970	34.4658	55.3184	59.1806
C	28.1516	19.9275	3.6693	12.1283	74.5617	29.5577	32.2968	31.2606	50.5285	68.3058
D	29.8329	18.1234	13.5887	4.1474	69.1587	29.7792	22.7838	21.9178	45.3112	68.0254
E	63.1235	59.6308	70.1284	68.3435	0.7640	49.2359	49.7469	54.3105	71.2846	49.9768

TABLE 5
MATCHING RESULT

Vertex $\in V_{cam}$	Vertex $\in V_{ref}$
A	A'
B	B'
C	C'
D	D'
E	E'

4 Discussion

4.1 Scope and rationale of the extended analyses

This discussion section presents two extended problem analysis focused on the CoarseMatching stage, included two question:

- Question 1 (Q1) : Whether the three-ring shape vector descriptor can be replaced by a single ring ?
- Question 2 (Q2): The effect of filtering vertices by convexity, retaining convex vertices only and discarding concave ones.

These analyses are conducted without modifying the baseline methodology: all definitions of the shape vector, sectorization, distance, and early rejection rule remained exactly as specified earlier in this report and follow the same notation.

To address Q1 and Q2, we conduct controlled experiments within the CoarseMatching pipleline of Ouyang et al.[1] using the same data perparation and parameter settings as the baseline. Unless otherwise noted, Map 1 serves as the primary testbed; depending on the specific analysis, we additionally report results on Map 2 and/or supplementary datasets beyond Map 1/Map 2 to assess robustness and generality. Any deviations from the baseline configuration (e.g, descriptor radial granularity for Q1 or vertex-convexity filtering for Q2) are restricted to the corresponding ablation, while other components remain unchanged.

Specifically, the one-ring variant for Q1 uses the identical pipeline but recues the descriptor's radial granularity to a single ring (eight angular sectors), whereas the convex-only variant for Q2 evaluates the impact of removing concave vertices after polygon simplification (Douglas-Peucker), keeping all other steps unchanged. This design isolate the effect of each change on the CoarseMatching outcome while preserving the comparability to the baseline.

4.2 Protocol and Metrics

This subsection formalizes the evaluation protocol for the extended analyses (Q1: one-ring vs. three-ring; Q2: convex-only vs. all vertices) and specifies the metrics used to quantify performance. All definitions and symbols strictly follow the baseline CoarseMatching formulation already established in this report consistent with Ouyang et al.[1]

We report two complementary indicators widely used in retrieval and matching tasks: an NNDR-inspired confidence score and Top-K accuracy. Both are evaluated per query and then aggregated over the test set. All reported numbers are formatted with two decimal digits for clarity and consistency, as is common practice in IEEE vision papers.

4.2.1 Confidence score

In the context of feature matching for UAV/aircraft localization, the confidence score quantifies the reliability of a candidate correspondence between two features. This measure is derived from the well-known Lowe's ratio test used in the SIFT algorithm[2], which compares the nearest and the second-nearest neighbor distances to assess the distinctiveness of the match.

For a query vertex X, the matching stage produces two key values, let:

- $d_1 = \min_{Y \in V_{\text{ref}}} \text{dis}(f^X, f^Y)$, which represents the smallest shape vector distance value.
- $d_2 = \min_{Y \in V_{\text{ref}} \setminus \{Y^*\}} \text{dis}(f^X, f^Y)$, which represents the second-smallest shape vector distance value , excluding the best match Y^*.

The ratio defined by Lowe [2] is:

$$r = \frac{d_1}{d_2}$$

Lowe's criterion accepts a match if $r < \tau$. Where τ is a threshold (e.g., 0.8) .In practice, Lowe recommends accepting only feature pairs with a ratio of $\frac{d_1}{d_2} < 0.8$,which eliminates up to 90% of incorrect matches while discarding fewer than 5% of correct ones.

In this work, the confidence score is defined as:

$$\text{Conf} = \frac{d_2 - d_1}{d_2} = 1 - \frac{d_1}{d_2}$$

This formulation can be interpreted as the normalized margin between the best and the second-best match:

- If $d_1 << d_2$, then Conf ≈ 1,indicating a highly reliable match
- If $d_1 \approx d_2$, then Conf ≈ 0, indicating ambiguity

Thus, the confidence score provides a direct measure of distinctiveness, where higher values correspond to more discriminative and reliable matches. This concept is similar with the criteria "Nearest Neighbor Distance Ration (NNDR)" suggested by Lowe in SIFT [2]

Confidence Mean

Confidence mean is capable of reflects the average reliability across all candidate matches. Given a set of M correspondences, each with its own confidence value Conf_i, we define the following aggregate statistics:

$$\mu_{\mathrm{Conf}} = \frac{1}{M}\sum_{i=1}^{M} \mathrm{Conf}_i$$

Confidence Median

The confidence median $\widetilde{\mathrm{Conf}}$ provides a robust measure of central tendency, less sensitive outliers than the confidence mean. The confidence median $\widetilde{\mathrm{Conf}}$ is defined as the middle value of the ordered set of confidence scores, $\{\mathrm{Conf}_i\}_{i=1}^{M}$, to compute this statistic:

1. Ordering step: Arrange confidence scores of each vertex in ascending order:

$$\mathrm{Conf}_{(1)} \le \mathrm{Conf}_{(2)} \le \cdots \le \mathrm{Conf}_{(M)}$$

2. Selection rule
   - If M is odd, the median is the value at position $\frac{M+1}{2}$:

$$\widetilde{\mathrm{Conf}} = \mathrm{Conf}_{\left(\frac{M+1}{2}\right)}$$

   - If M is even, the median is the arithmetic mean of the two central values at positions $\frac{M}{2}$ and $\frac{M}{2}+1$:

$$\widetilde{\mathrm{Conf}} = \frac{\mathrm{Conf}_{\left(\frac{M}{2}\right)} + \mathrm{Conf}_{\left(\frac{M}{2}+1\right)}}{2}$$

4.2.2 Top-K accuracy

In addition, $Top-K$ accuracy is a commonly used evaluation metric in recognition and retrieval tasks, measuring whether the algorithm's proposed results contain the correct answer within the first K candidates. In the context of UAV localization, $Top-K$ accuracy indicates whether the ground-truth position is included among the K closest matching candidates. This serves as a standard measure to assess whether the system can find the correct location at an "approximate" level. The metric is equivalent to $Recall@K$ in image retrieval: for example, $Recall@1$ (also known as Top-1 accuracy) is the proportion of UAV images whose corresponding satellite image appears at rank 1; $Recall@5$ (Top-5 accuracy) is the proportion of UAV images whose correct satellite image is within the top 5 results [3]. Many computer vision studies also report Top-1 and Top-5 accuracy to evaluate model performance [4]. Therefore, using $Top-k$ accuracy for evaluating CoarseMatching is appropriate and consistent with common practice.

Formally, in this discussion scope, let N denote the total number of UAV query images, GT_i the ground-truth location of the, $i-th$ query, and $\mathrm{rank}_i(\mathrm{GT})$the rank position of the ground-truth among the retrieved candidates. The indicator function $\mathbf{1}\{\cdot\}$ equals 1 if the condition holds and 0 otherwise. The Top-K accuracy is defined as:

$$\mathrm{Acc}@K = \frac{1}{N}\sum_{i=1}^{N} \mathbf{1}\{\mathrm{rank}_i(\mathrm{GT}) \le K\}$$

Special cases included:

- Acc@1 (Top-1 accuracy): the proportion of UAV images for which the ground-truth location is ranked first.
- Acc@3 (Top-3 accuracy): the proportion of UAV images for which the ground-truth location within the top 3 candidates.

- Acc@5 (Top-5 accuracy): the proportion of UAV images for which the ground-truth location within the top 5 candidates.

This metric is sometimes referred to as *Recall@K* in the retrieval literature and has been widely applied in UAV localization and geolocalization studies. In CoarseMatching pipelines, Top-K accuracy provides a crucial benchmark, as it directly reflects whether the correct correspondence for each vertex survives into the top-K candidate set that will be handed to downstream fine matching and probabilistic refinement.

4.3 Q1 Analysis

4.3.1 Scope of Q1 Experiments

We conduct three controlled experiments in which the shapevector uses a single ring with eight angular sectors (full sectorization but only only one radius active for each experiment), respectively are: Only r_1, Only r_2, Only r_3 on the given Map 1. All settings - polygon simplification, distance in Eq.(2) and the early rejection threshold- follow the baseline formulation in this report and the CoarseMatching description of Ouyang et al. [1]. the baseline for comparison is the three-radius descriptor (r_1, r_2, r_3). We report, without re-defining metrics, the comparative Top-K results and the confidence mean/median for each of the three single-radius runs against the baseline full three radii descriptors.

To assess computational scalability independently of the synthetic maps, we use a separate building-polygon dataset that contained simplified polygon from buildings in Vietnam, to be obtained from a public website. The area of interest is a 1 km radius centered on the Vietnam National University - Ho Chi Minh city, University of Technology library. For all vertices of all building polygons within this area, we run CoarseMatching twice under identical conditions except for descriptor granularity: one ring (single radius, eight sectors) and three-ring (three radii, twenty-four sectors). We report the runtime difference between the one-ring and three-ring settings.

All other components (preprocessing, BuildShapeVector, early rejection, and candidate search) remain unchanged to isolate the impact of the descriptor design.

4.3.2 Impact On Confidence and Acc@K

Conf	r_1	r_2	r_3	r_1, r_2, r_3
A	0.76	0.82	0.78	0.78
B	0.50	0.11	0.49	0.49
C	0.73	0.62	0.59	0.78
D	0.43	0.58	0.68	0.69
E	0.99	0.99	0.99	0.99
$\widetilde{\text{Conf}}$	0.73	0.62	0.68	0.78
μ_{Conf}	0.68	0.63	0.70	0.74

Acc@K	r_1	r_2	r_3	r_1, r_2, r_3
Acc@1	80%	100%	100%	100%
Acc@3	100%	100%	100%	100%
Acc@5	100%	100%	100%	100%

The paragraph below analyzes only the numbers shown in the Q1 table for the four settings (Only r_1, Only r_2, Only r_3, and r_1, r_2, r_3).

Across settings, using all three radii gives the best confidence statistics and non-inferior Acc@K: the combined r_1, r_2, r_3 achieves the highest aggregate confidence ($\widetilde{Conf}$ = 0.78, μConf = 0.74), exceeding any single-radius run (best single-ring μConf = 0.70 at r_3; best single-ring median $\widetilde{Conf}$=0.73 at r_1). For accuracy, r_1, r_2, r_3 attains Acc@1/3/5 = 100% and thus ties the best single-ring settings (r_2 and r_3) while outperforming r_1 at K = 1 (80%).

When a single radius must be used and the priority is the median confidence, choose r_1 ($\widetilde{\text{Conf}}$ =0.73 vs. 0.62 for r_2 and 0.68 for r_3). If the priority is the mean confidence, choose r_3 (μConf = 0.70 vs. 0.68 for r_1 and 0.63 for r_2).

In practice, when constrained to a single radius and considering both confidence and Acc@K, r_3 is the most balanced choice: it delivers Acc@1/3/5 = 100% (matching r_2 and the three-radius baseline) while also providing the strongest μConf among single-ring variants; r_2 shows markedly poor confidence on vertex B (0.11), indicating instability despite perfect Acc@K.

Per-vertex behavior highlights why the three-radius descriptor is preferable by default: E is trivially distinctive across all settings (Conf $\approx$ 0.99), whereas other vertices are radius-sensitive — e.g., D improves from 0.43 (r_1) to 0.68 (r_3), and C's confidence, lower with r_3 (0.59), recovers to 0.78 when all three radii are used - suggesting that aggregating radii mitigates sensitivity to local geometry and yields more uniformly high confidence.

4.3.3 Impact On Runtime

To rigorously evaluate the runtime performance and matching accuracy of the proposed method, the scope of this experiment was expanded. The initial map used in the confidence analysis (MAP 1) contained too few vertices to demonstrate a significant difference in computation time between various configurations. Therefore, to create a more challenging and realistic test case, a larger reference map was employed for this specific analysis.

The reference map was generated from OpenStreetMap data [5], distributed under the Open Database License (ODbL) [6] and obtained from the Geofabrik data extracts service [7]. The extracted area covers a 1-kilometer radius around the Ho Chi Minh City University of Technology library, representing a dense urban environment with approximately 2,200 building vertices. This larger dataset enables a more reliable evaluation of the algorithm's scalability and efficiency.

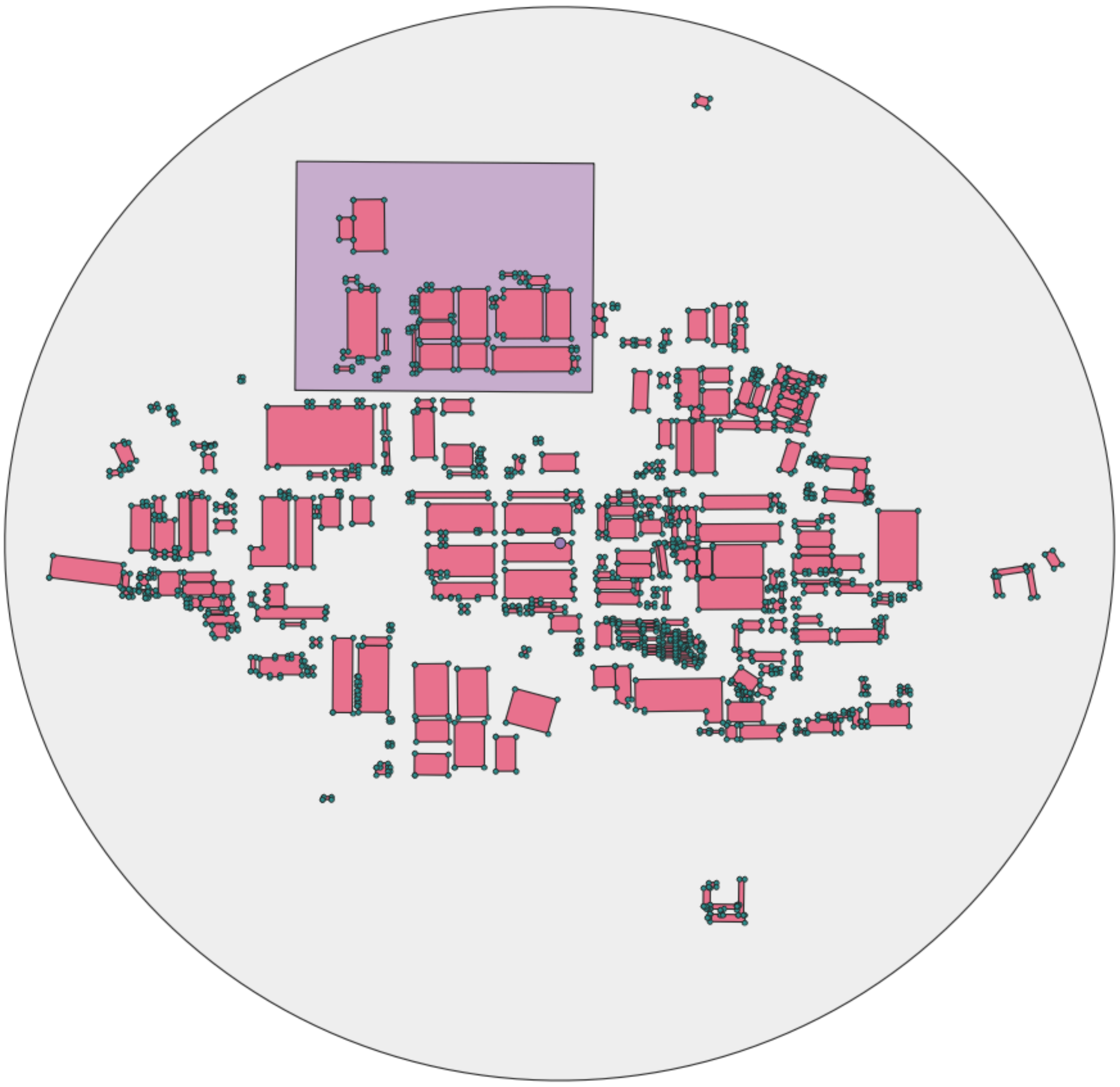

Fig. 5. Simulated reference map using QGIS

To validate the matching algorithm, we computationally simulated aerial imagery from a UAV. This process, which mimics real-world distortions and errors in aerial photography, generates an image based on a 500x400-meter crop of the reference map.

The simulation is configured with the following key parameters:

- Geometric Scaling: Building polygons are uniformly scaled by a factor (scale_factor) of 1.1 to simulate slight variations in altitude or camera lens distortion.
- Positional Noise: Gaussian noise with a standard deviation (noise_std_dev_m) of 0.8 meters is added to each vertex of the building polygons. This represents minor GPS inaccuracies or image processing artifacts.

- Merging Error: This simulates a common segmentation error where closely-located or adjacent buildings are mistakenly identified as a single, larger structure. This is controlled by two parameters:
  - merge_distance_m: A distance threshold (e.g., 1.5 meters) defining how close two buildings must be to be considered for merging.
  - merge_probability: A probability (e.g., 0.3) that a pair of buildings within the distance threshold will actually be merged.

The resulting simulated UAV image, which consists of these altered building shapes, serves as the input for the matching process, providing a challenging and realistic test case for the algorithm's robustness.

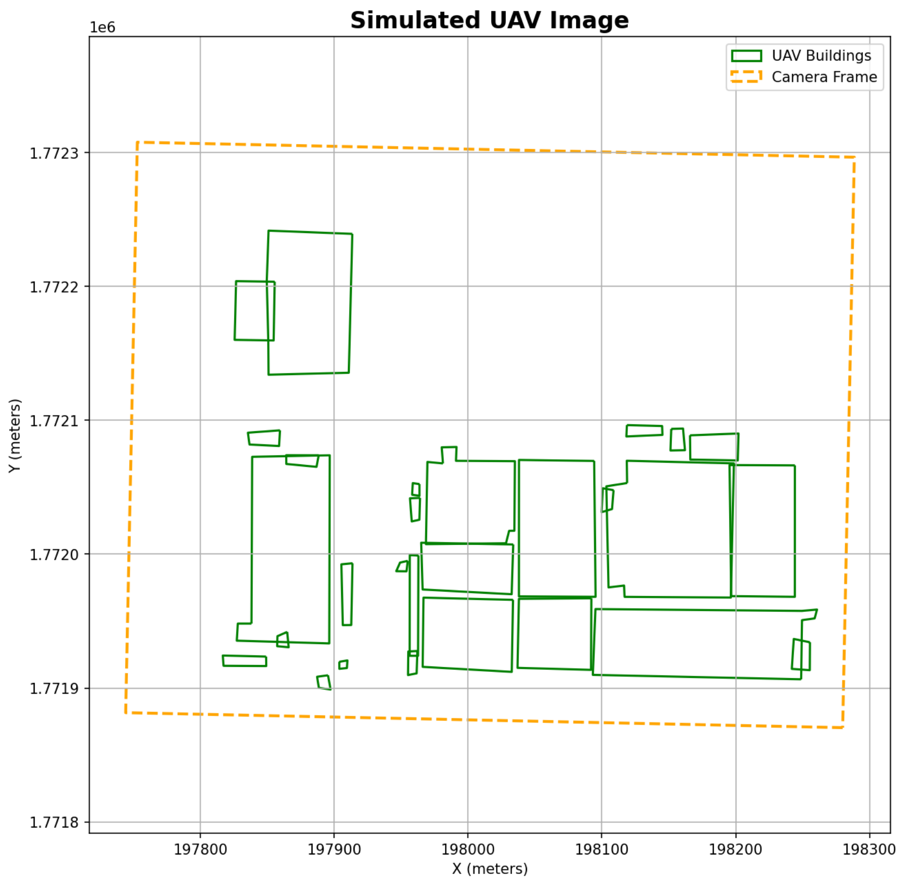

Fig. 6. An example of a randomly-generated UAV image

The matching algorithm's performance is visually verified by overlaying the matched image map points onto the reference map.

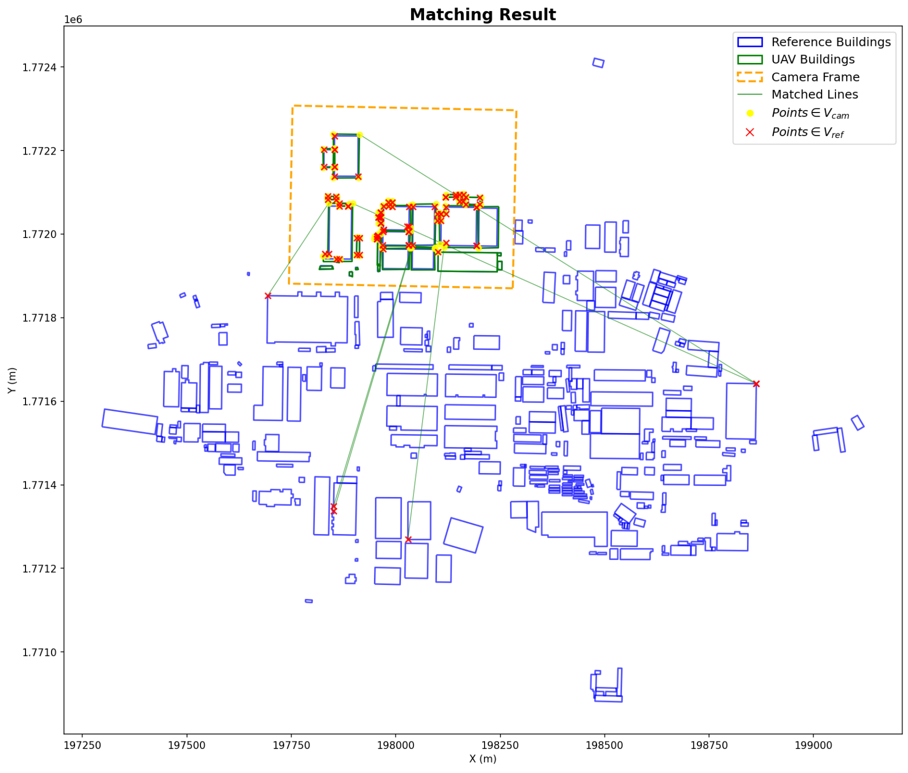

Fig. 7.a. Matching Result (All 3 radii)

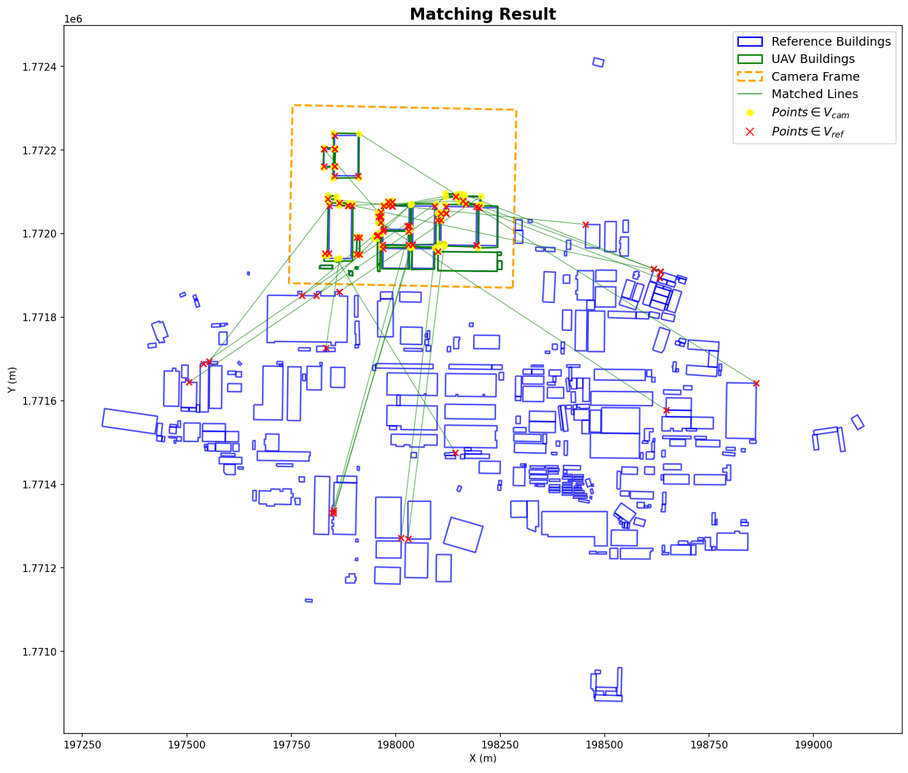

Fig. 7.b. Matching Result (Only r_3)

TABLE 6
RUNTIME STATISTICS (ALL 3 RADII)

Run	Runtime (s)
1	70.40
2	56.60
3	58.96
4	58.69
5	67.56
Average	**62.44**

TABLE 7
RUNTIME STATISTICS (ONLY r_3)

Run	Runtime (s)
1	33.11
2	37.81
3	32.31
4	39.27
5	49.22
Average	**38.34**

In Fig.7, the three-ring descriptor yields visibly tighter correspondences than the single-ring r_3 setting: in Fig.7(a) the matched segments (green) are short and concentrated inside the camera frame, and image-map (V_{cam}, V_{ref}) points nearly coincide in the highlighted block; in Fig.7(b) many long green links radiate across the map, indicating larger deviations between V_{cam} and V_{ref} vertices. This visual evidence aligns with the Q1 statistics: the three-ring setup attains the highest confidence (μConf = 0.78, $\widetilde{Conf}$ = 0.74) while matching the best Acc@K (Acc@1/3/5 = 100%; r_1 drops to Acc@1 = 80%).

We evaluated two configurations: (Fig. 7.a) the full three-ring descriptor (r_1, r_2, r_3) and (Fig. 7.b) a single-ring variant using only r_3 - the radius that achieved the highest Confidence and Acc@K in Sec.4.3.2. As summarized in Table 6 - Table 7, the average runtime drops from 62.44 s (three rings) to 29.38 s (single r_3, corresponding to a 52.95% reduction ($\approx$ 2.13$\times$ speedup)). Given that r_3 also delivers the best accuracy metrics reported earlier, the single-ring configuration offers a superior speed–accuracy trade-off and is preferable for runtime-critical deployments.

4.3.4 Conclusion for Q1

Considering discussed points, the full three-ring (r_1, r_2, r_3) yields the strongest confidence score and non-inferior Acc@K: it attains the μConf = 0.78 and $\widetilde{Conf}$ = 0.74, exceeding any single-radius run (best single-ring $\widetilde{Conf}$ = 0.7 at r_3; best single-ring μConf = 0.73 at r_1), while matching the best accuracy with Acc@1/3/5 = 100% (where r_1 lags at Acc@1 = 80%). Among single-ring variants, r_3 is the most balanced choice: it preserves perfect Acc@1/3/5 and offers the highest $\widetilde{Conf}$ whereas r_2 shows unstable confidence (e.g., 0.11 at vertex B). In terms of efficiency, using only r_3 reduces average runtime from $62.44s$ (three rings) to $29.38s$, a 52.95% reduction, approximately 2.13x speedup. Therefore, our selection policy is: use (r_1, r_2, r_3) by default when robustness and uniformly high confidence are required; use single-ring r_3 for runtime-critical deployments or when a single radius is mandated, as it delivers the best speed-accuracy trade-off.

4.4 Q2 Analysis

4.4.1 Scope of Q2 Experiments

To evaluate the impact of removing concave vertices, two experiments are conducted: one assessing its effect on matching accuracy, and the other on computational efficiency. In the first experiment, which is provided in section 4.4.3, Map 1 is employed to assess the validity of the matching process, supported by mentioned evaluation metrics. In the second experiment, we move beyond Map 1/Map 2 and use a larger city-scale reference map generated from OpenStreetMap data[5], distributed under the Open Database License (ODbL)[6] and obtained from the Geofabrik data extracts service [7]; the extracted area spans a 1-km radius around the Ho Chi Minh City University of Technology library and contains approximately 2,200 building vertices. On this dataset, we apply a convexity filter after Douglas–Peucker simplification that removes concave vertices and retains only convex vertices, while keeping all other CoarseMatching components unchanged to isolate the effect of this modification. We then run the same CoarseMatching pipeline and compare end-to-end runtime against the baseline that matches using all vertices (convex-only vs. all-vertex setting), which will be provided in section 4.4.4 .

4.4.2 Algorithms for Vertex Classification

Without addressing correctness proofs or proposing further methodological extensions, Sec.4.4.2 provides only the minimal mathematical apparatus needed to classify polygon vertices as convex or concave. We employ the classic Shoelace formula [8] to fix polygon orientation (CCW/CW), then label each interior vertex by the sign of the oriented cross product of its adjacent edges under that orientation. The algorithmic steps are included solely to filter out non-convex (concave/flat) vertices, enabling the ablation in §4.4 to compare the effect of “convex-only” retention on Confidence, Acc@K, and runtime within CoarseMatching pipeline from Ouyang et al. [1]; all other components (e.g., Douglas–Peucker simplification and shape-vector construction) remain unchanged.

To determine whether a vertex is convex or concave we evaluate the local turn at that vertex relative to the polygon's global winding. Concretely, for a vertex B with neighbours A and C we compute the scalar cross product

$$\text{cross} = (C_x - B_x)(A_y - B_y) - (C_y - B_y)(A_x - B_x),$$

whose sign indicates the direction of the interior turn. The sign is interpreted with respect to the polygon orientation (obtained from the signed area): if the global orientation is counterclockwise (CCW) then cross > 0 denotes a convex turn, while if the orientation is clockwise CW the inequality flips. A small tolerance ε is applied so that $|\text{cross}| \leq \varepsilon$ is classified as collinear (flat). Terminal vertices of an open polyline are returned separately as endpoints.

Below are the two main algorithms used in this procedure.

PolygonSignedArea Algorithm

PolygonSignedArea computes the signed area of an ordered sequence of planar vertices using Shoelace formula. The sign of the result indicates the polygon winding (positive means CCW), which is used subsequently for vertex classification.

Input: ordered list $coords = \langle (x_1, y_1), \ldots, (x_n, y_n) \rangle$ containing the coordinate of the vertices of all building polygons (closed polygon requires $n \geq 3$).

Output: scalar A equal to the signed area of the polygon.

Algorithm 3: PolygonSignedArea($coords$)

```
1. $A \leftarrow 0.0$
2. $n \leftarrow \text{length}(coords)$
3. **for** $i \leftarrow 0$ **to** $n-1$
4.     $(x1, y1) \leftarrow coords[i]$
5.     $(x2, y2) \leftarrow coords[(i+1) \bmod n]$
6.     $A \leftarrow A + (x1 \cdot y2 - x2 \cdot y1)$
7. $A \leftarrow A/2.0$
8. **return** $A$
```

ClassifyVertices Algorithm

ClassifyVertices labels each interior vertex as convex, concave, or collinear using a three-point interior-turn test and the global winding returned by PolygonSignedArea algorithm. Terminal vertices of an open polyline are returned in endpoints.

Input: labelled sequence $vertices = \langle (\ell_1, (x_1, y_1)), \ldots, (\ell_n, (x_n, y_n)) \rangle$, boolean $incomplete$ (true for open polyline), tolerance $\varepsilon > 0$

Output: record ⟨orientation, area, convex, concave, flat_collinear, endpoints⟩

Algorithm 4: ClassifyVertices(vertices, $incomplete$, $\varepsilon = 10^{-9}$)

```
1. $names \leftarrow [\ell \mid (\ell, (x, y)) \in \text{vertices}]$
2. $coords \leftarrow [(x, y) \mid (\ell, (x, y)) \in \text{vertices}]$
3. $n \leftarrow \text{length}(coords)$
4. **if** not $incomplete$ and $n < 3$
5.     **error**("Closed polygon requires $n \geq 3$")
6. **if** not $incomplete$
7.     $area \leftarrow PolygonSignedArea(coords)$
8.     **if** $area > \varepsilon$
9.         $orientation \leftarrow \text{CCW}$
```

```
10.     **elseif** $area < -\varepsilon$
11.         $orientation \leftarrow$ CW
12.     **else** $orientation \leftarrow$ degenerate
13. **else** // In polyline case, area is not calculated and the orientation is assumed to be CW
14.     $area \leftarrow 0.0$
15.     $orientation \leftarrow$ CW
16. **if** not $incomplete$
17.     append $names[0]$ and $names[n-1]$ onto the end of $endpoints$
18.     $idx\_range \leftarrow 1$ **to** $n-2$
19. **for each** $i \in idx\_range$
20.     $A \leftarrow coords[i-1]$
21.     $B \leftarrow coords[i]$
22.     **if** $incomplete$
23.         $C \leftarrow coords[i+1]$
24.     **else** $C \leftarrow coords[(i+1) \bmod n]$
25.     $cross \leftarrow (C[0]-B[0]) \cdot (A[1]-B[1]) - (C[1]-B[1]) \cdot (A[0]-B[0])$
26.     **if** $|cross| \leq \varepsilon$
27.         append $names[i]$ onto the end of $flat\_collinear$
28.     **else**
29.         **if** $orientation =$ CCW
30.             $t\_convex \leftarrow (cross > 0)$
31.         **elseif** $orientation =$ CW
32.             $t\_convex \leftarrow (cross < 0)$
33.         **if** $t\_convex$
34.             append $names[i]$ onto the end of $convex$
35.         **else** append $names[i]$ onto the end of $concave$
36. **return** $\langle orientation, area, convex, concave, flat\_collinear, endpoints \rangle$
```

Complexity Analysis

For both Algorithm 3 and Algorithm 4, since they are simple and do not contain nested loops, the time complexity is $O(N)$, where N denotes the number of vertices.

4.4.3 Impact On Confidence and Acc@K

There are two stages in this experiment that check the matching accuracy after excluding concave vertices: Convexity filtering (removing concave vertices) and Matching/Evaluating. Map 1 is used in this experiment.

Convexity Filtering

This stage reads building polygons data from a text file, classifies each vertex as Convex, Concave, Flat/-collinear, or Endpoint, and prints the classification per building. The algorithm uses the Shoelace formula to determine polygon orientation and the signed cross product of adjacent edges to classify vertices.

Input: A plain text file (.txt) storing the coordinates of the vertices for all building polygons on Map 1. The structure of the file is organized such that each building is represented as a separate section. A section begins with the building's name, followed by a colon, and then lists its vertices, one per line. Vertex labels may include letters, digits, underscores, hyphens, or apostrophes. It is noted that vertices labeled with numbers do not have an associated shape vector, whereas those labeled with letters do.

Output: For each building, the program prints the building name (optional) followed by lists of vertex names for each class: 'Convex', 'Concave'

Result: The concave vertices labeled with letters are: B', C', D', G' (on the map) and C (on the image). This means at next stage, these vertices are ignored.

Matching/Evaluating

As mentioned, in this stage, vertices B', C', D', G' and C are removed when running the CoarseMatching algorithm.

4.4.4 Impact on Runtime

Based on the algorithm described in Section 4.4.2, an experiment was conducted to evaluate the runtime performance of the approach that filters convex/concave points (keeping only convex points in both V_{ref} and V_{cam}). The results of the classifications are illustrated in Fig. 8 and Fig. 9.

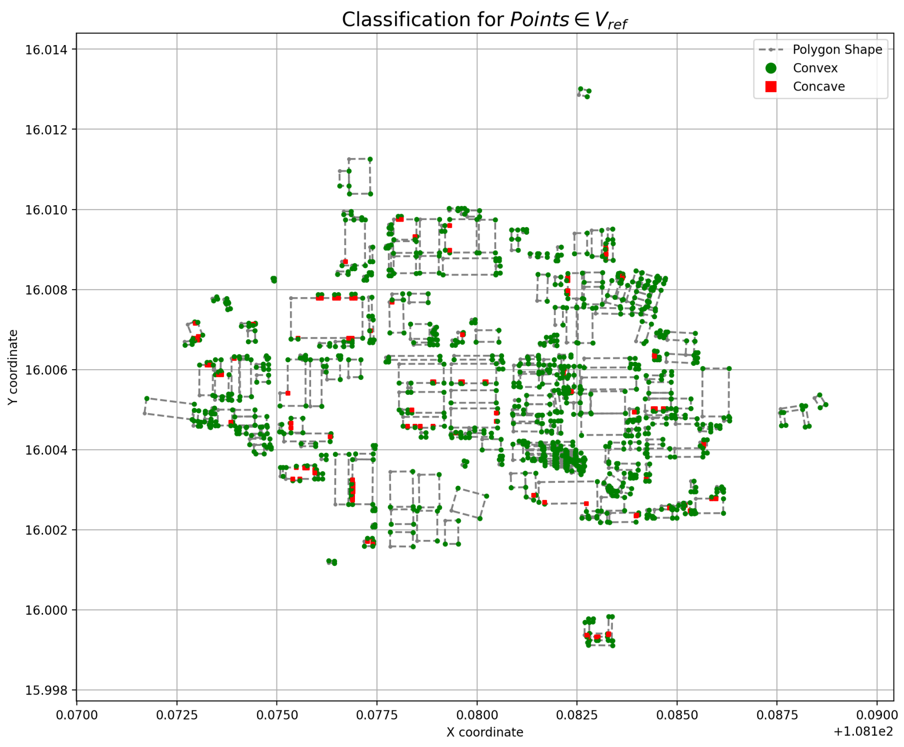

Fig. 8. Classification result for Points $\in V_{ref}$

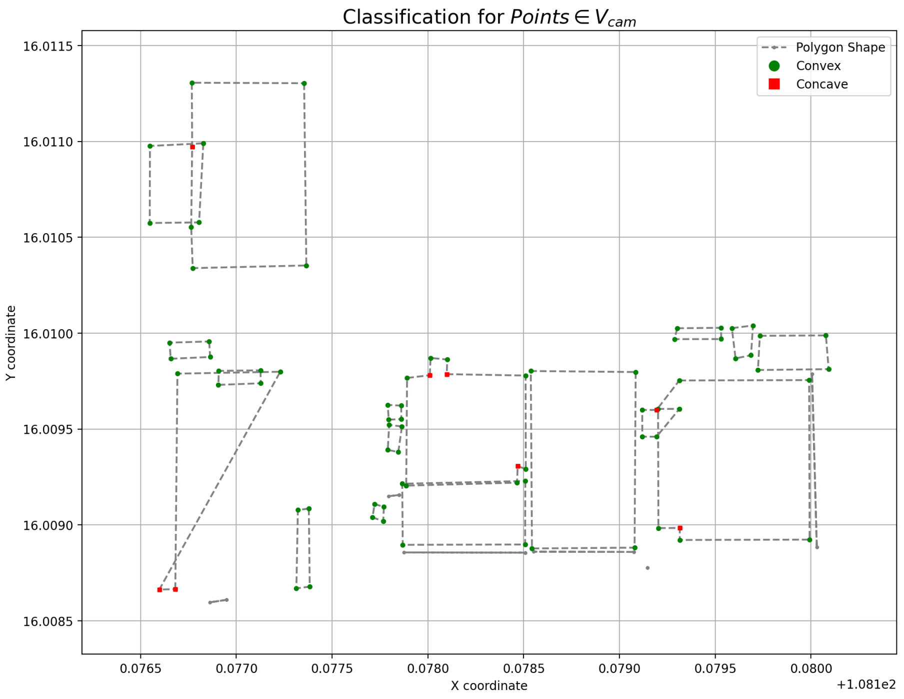

Fig. 9. Classification result for Points $\in V_{cam}$

After this filtering step, the pipeline continues with early rejection, followed by the computation of shape vectors and the point matching procedure, in the same manner as the experiment in Section 4.3.3.

TABLE 8
RUNTIME STATISTICS (ALL 3 RADII)

Run	Runtime (s)
1	55.12
2	53.18
3	53.41
4	52.59
5	54.04
Average	**53.67**

An experiment was conducted to evaluate the performance of a shape vector matching algorithm under two distinct pipelines. The baseline pipeline performs matching on the full set of candidate points. The experimental pipeline introduces a convexity-based filtering step designed to reduce the number of candidate points prior to the matching stage.

Theoretically, the convexity filter is highly effective. It reduced the number of reference points by 44% (from 2,171 to 1,210) and image points by 21% (from 91 to 72). This leads to an estimated 55.9% reduction in the required computational workload (element-wise matching operations).

Despite a theoretical 55.9% reduction in computational workload from convexity filtering, the experimental pipeline (0.148s) was paradoxically slower than the unfiltered main baseline (0.121s). This outcome is attributed to two main factors:

- The baseline matching algorithm is already highly vectorized and efficient, especially on the larger, contiguous dataset used by the baseline pipeline. The performance gains from this vectorized efficiency outweighed the benefit of performing fewer raw calculations.
- Fixed computational overheads, such as function calls and memory management, had a greater relative performance impact on the smaller, filtered dataset, diminishing any potential time savings.

In conclusion, for the tested data scale, the direct matching approach is superior. The benefits of filtering are only hypothesized to emerge on significantly larger datasets where the massive reduction in calculations can overcome the inherent overhead of the process.

5 Conclusion

This study has implemented and evaluated the CoarseMatching component of the CFBVM framework originally proposed by Ouyang et al.[1] for vision-based UAV localization framework for UAVs operating in GNSS/GPS-denied environments. By leveraging semantic segmentation and vector-based geometric representations, the method establishes correspondences between image-derived features and a precomputed reference map.

The implementation focused on constructing and comparing shape vectors at vertex points using the Douglas–Peucker algorithm and multi-scale circular sectors. The CoarseMatching algorithm identifies the most likely correspondences between captured vertices $\in V_{cam}$ and vertices $\in V_{ref}$ based on shape vector similarity. An early-rejection condition based on area comparison was also integrated to reduce computational cost.

Through a worked example using planar geometry and simplified building contours, the method was successfully validated and demonstrated to produce accurate coarse-level matches. Although this step alone does not determine the UAV's final coordinates, it is a critical foundation for subsequent refinement and filtering stages in the full positioning pipeline. Future work may extend this foundation to include fine matching and probabilistic filtering using particle-based methods.

6 References

[1] Ouyang, C., Long, F., Shi, S., Yu, Z., Zhao, K., You, Z., Pi, J., Xing, B., Hu, S. (2023). A semantic vector map-based approach for aircraft positioning in GNSS/GPS denied large-scale environment. *Defence Technology*, 34, 1–10.

[2] Forssén, P.-E., Lowe, D. G. (2007). Shape descriptors for maximally stable extremal regions. In *IEEE International Conference on Computer Vision and Pattern Recognition (CVPR)*, 2007, pp. 1–8.

[3] Yuan, Z., Fang, J., Zhang, S., Zhang, L., Yang, M. (2025). A Faster and More Effective Cross-View Matching Method. *arXiv preprint arXiv:2502.03007*.

[4] Li, C., Luo, Y., Xu, M., Zhou, Y., Chen, J., Chen, D., Wang, Y. (2025). AerialVG: A Challenging Benchmark for Aerial Visual Grounding by Exploring Positional Relations. *arXiv preprint arXiv:2504.07836*.

[5] Wikipedia contributors. OpenStreetMap. *Wikipedia*. Accessed on August 28, 2025.

[6] Open Data Commons. *Open Database License (ODbL) v1.0*. Accessed on August 28, 2025.

[7] Geofabrik GmbH. OpenStreetMap Data Extracts. Accessed on August 28, 2025.

[8] Wikipedia contributors. Shoelace formula. *Wikipedia*. Accessed on August 30, 2025.

[9] Latta, A. (2010). Polyline and Polygon Plotting in Map Objects (Doctoral dissertation, San Diego State University).

[10] Derick Rethans. Algorithms: Ramer-Douglas-Peucker explained, 2024.

Appendix A
Douglas-Peucker

A.1 What are Polygons and Polylines?

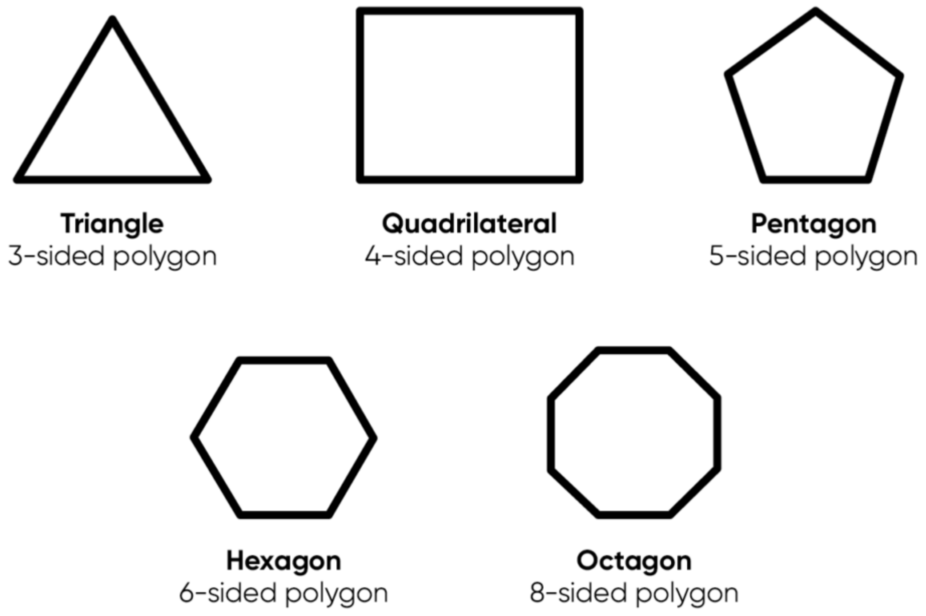

Fig. A.1

Polygon

A polygon is a closed geometric shape with multiple edges and angles. It is defined by a sequence of points (called vertices) connected by line segments to form a closed region.

Polygons can have varying numbers of edges, but they must have at least three edges (e.g., triangles). Examples include squares, rectangles, or more complex polygons with many edges.

Polygons have an interior space and are often used to represent areas in space (e.g., land, cities, or geographic regions).[9]

Polyline

A polyline is an open graphical object defined by a sequence of points connected by line segments but not forming a closed region like a polygon.

Polylines can have any number of points and are often used to represent lines, borders, or connected routes in space.

In GIS, polylines can represent features such as streets, rivers, railway lines, or any sequence of connected line segments.[9]

A.2 Douglas-Peucker Mechanism

This algorithm primarily works by finding a simplified polyline from an original polyline.

By removing unnecessary points, the maximum distance between the original polyline and the simplified polyline does not exceed a predefined threshold, called ε (Epsilon).

Operating principle:

- Connect the two farthest points with a straight line, then calculate the distance (perpendicular projection from a node to the connected line) to derive epsilon.

- Compare the distance d of each node to the line segment:

  • If $d \geq$ epsilon: retain the point.

  • If $d <$ epsilon: remove the point.

A.3 Example

Below is an illustrative example from an academic YouTube channel[10]:

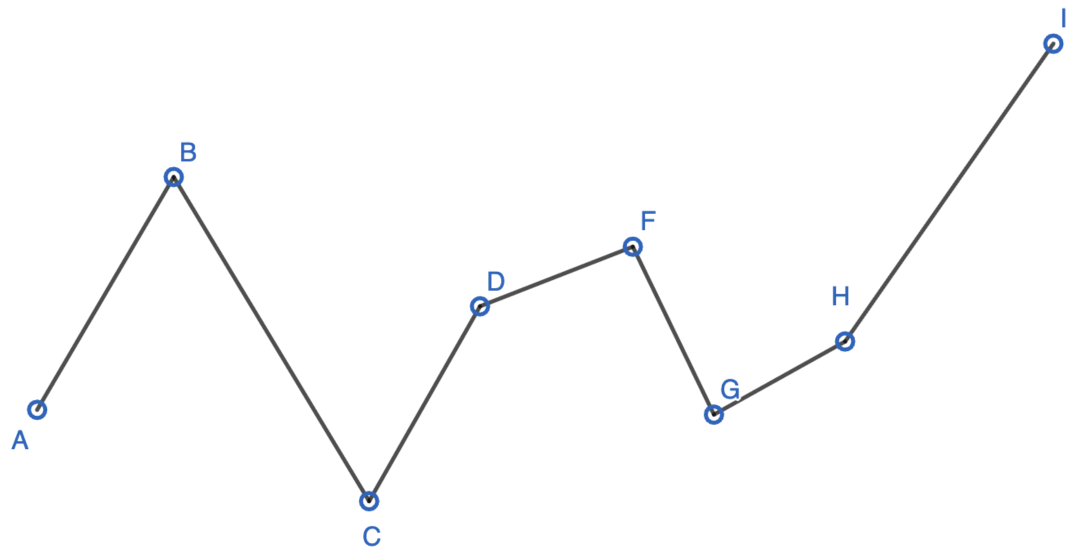

Fig. A.2

Assume epsilon = 15mm (i.e., all distances $\geq$ 15mm will be retained).

In the figure, the distances from the points to the line connecting the start and end points of the graph are labeled.

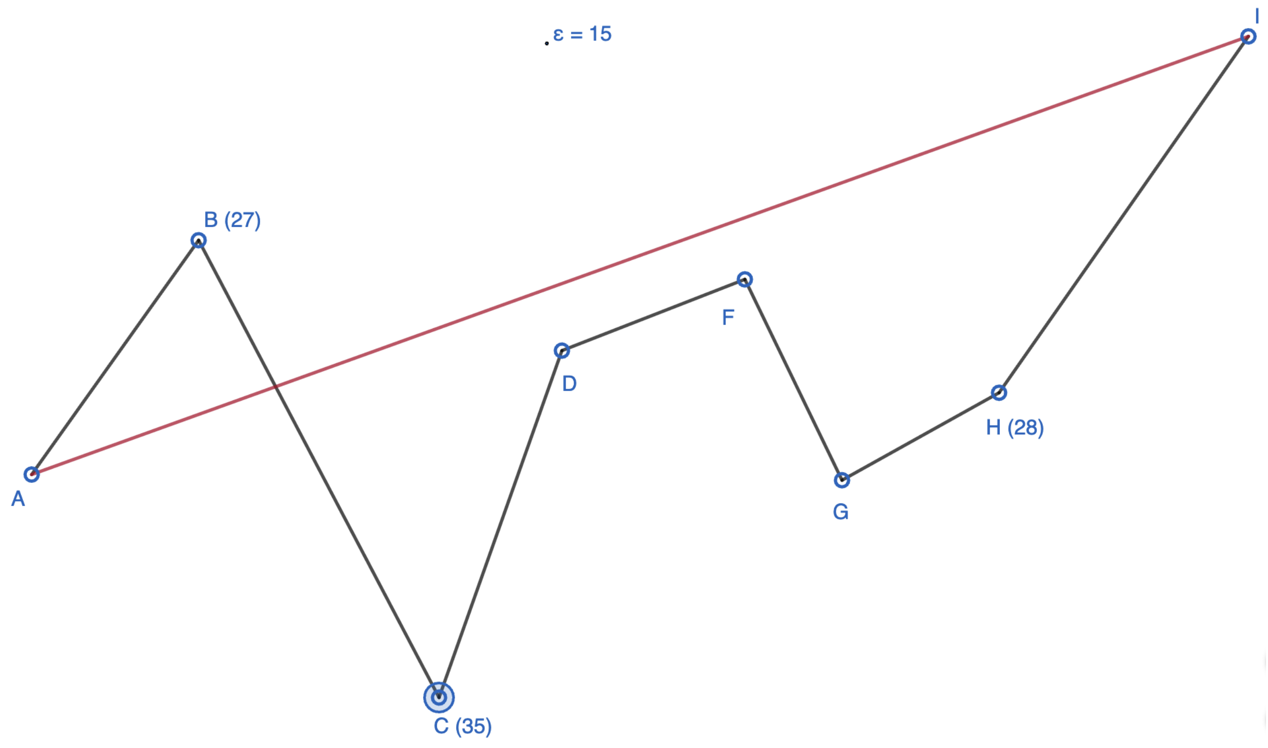

Fig. A.3

The current maximum distance (at vertex C) is 35mm (noted on the graph), and this vertex is marked. Connect points A and C and examine the points whose projections lie on segment AC. Point B to AC has a distance of 46 (since 46 $\geq$ 15), so point B is retained.

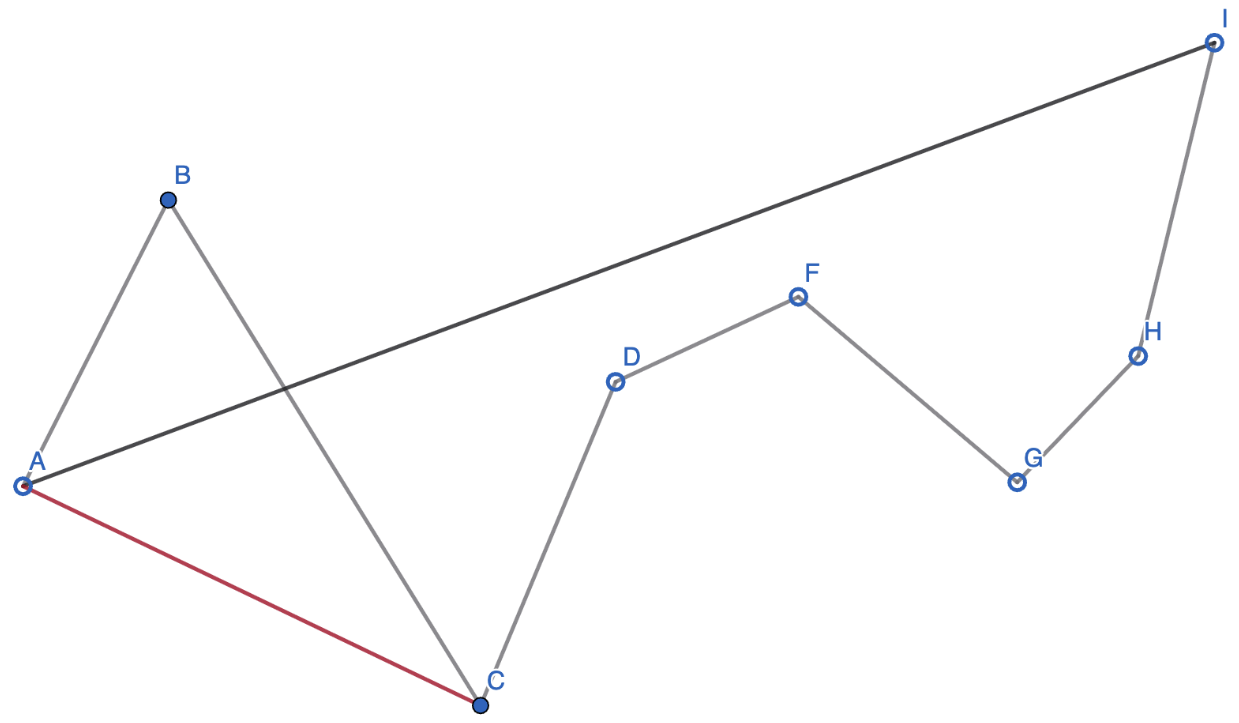

Fig. A.4

Next, connect points C and I and examine the points whose projections lie on segment CI. Among these, measure the distances to CI. Point D has the maximum distance, and $16 \geq 15$, so point D is retained.

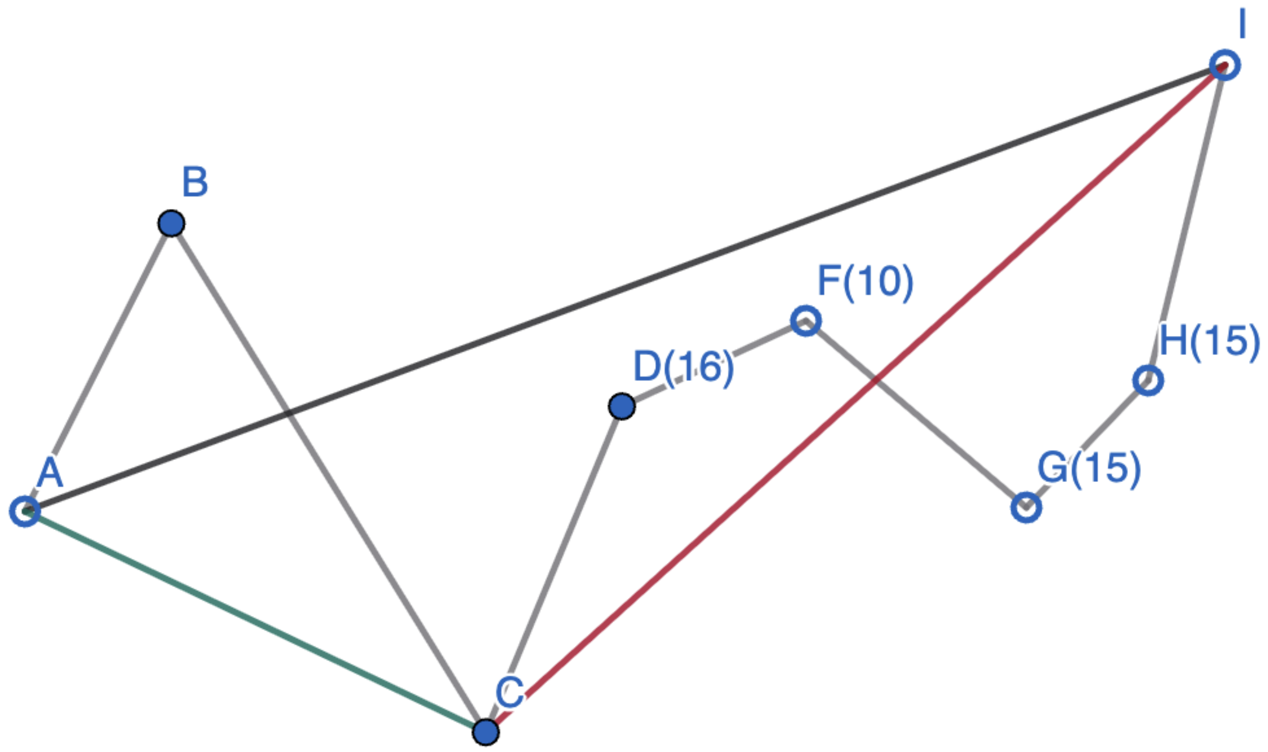

Fig. A.5

Then, connect points D and I and examine the points whose projections lie on segment DI. Within this segment, after measuring the distances from the points to DI, point G is retained (since $25 \geq 15$, and point G has the maximum distance to DI).

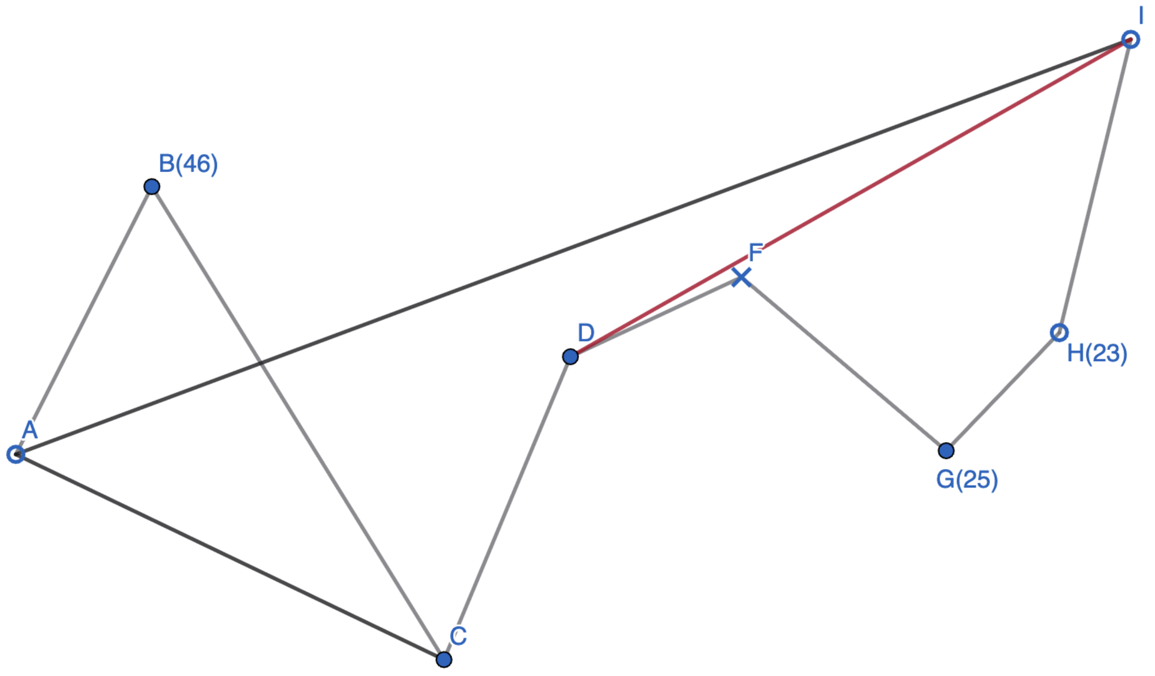

Fig. A.6

Connect point G to I and examine the segment containing GI. The process ends because the distance from H to GI $\leq \varepsilon$, so point H is discarded (marked with an 'X').

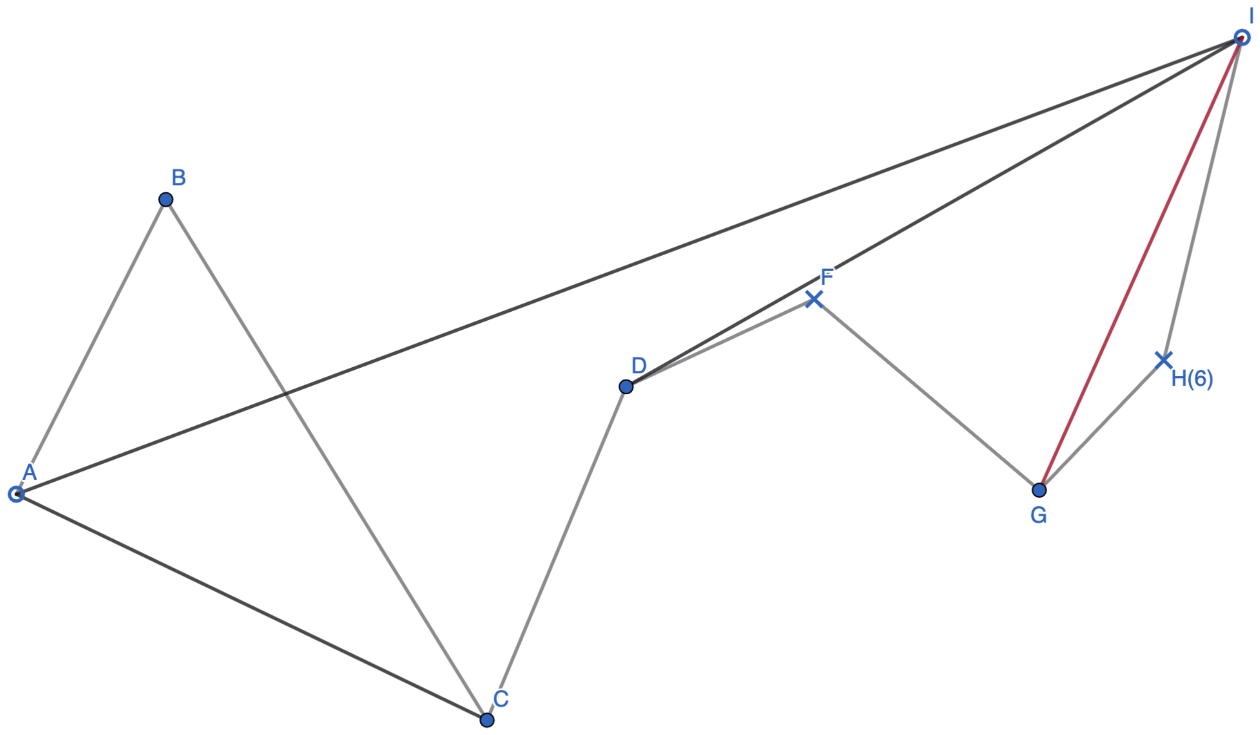

Fig. A.7

Connect the start point, selected points, and end point to obtain a new graph filtered by the Douglas-Peucker algorithm (marked with a green line).

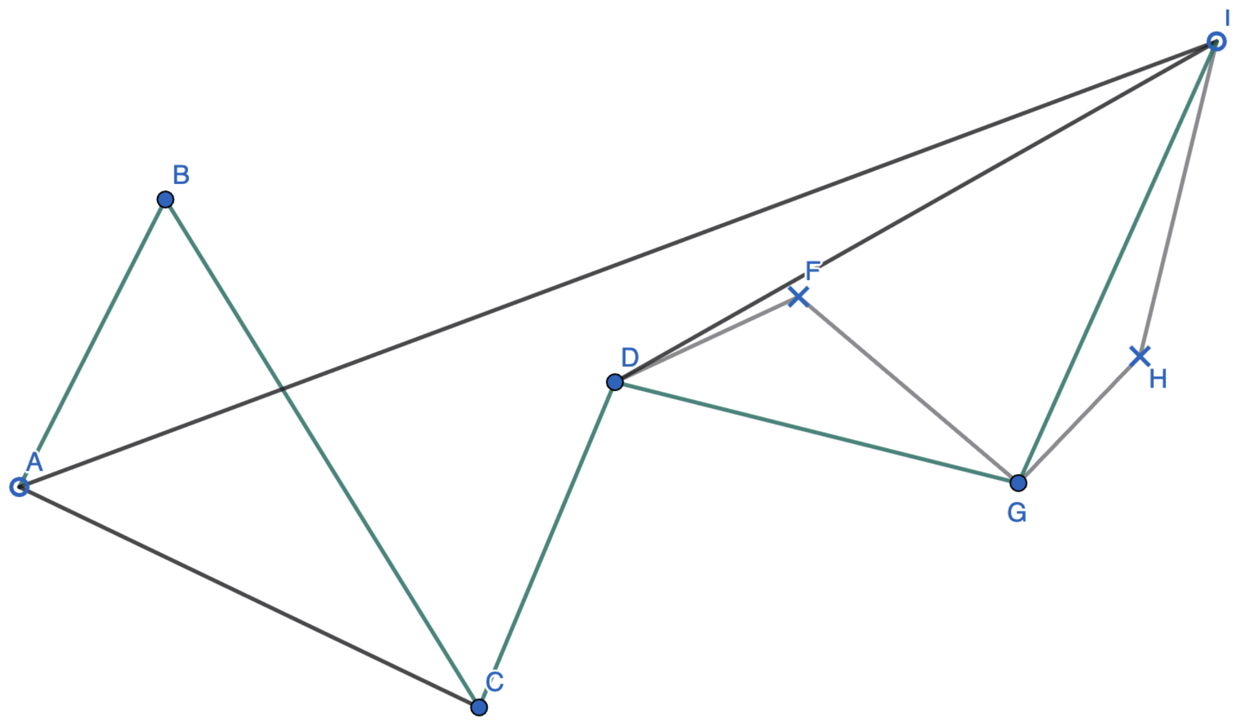

Fig. A.8

Algorithm Complexity:

- **Worst case:** The algorithm complexity is O(n^2), occurring when each recursion removes only one point.
- **Average case:** The algorithm complexity is O(n log n), when each recursion halves the number of points to process.

Appendix B
Shape Vectors Calculation

B.1 MAP 1

Compute A:

A	1	2	3	4	5	6	7	8
1	0	23.8740	353.4292	353.4292	33.3780	0	0	0
2	0	71.6219	1060.2875	962.5809	0	0	0	0
3	0	119.3669	1767.1459	1607.4443	0	0	0	0

$S_0^A = 6352.5578$

Compute B:

B	1	2	3	4	5	6	7	8
1	303.4455	353.4292	353.4292	210.1312	0	0	0	0
2	44.8374	860.1789	1060.2875	836.6863	0	0	0	0
3	0	958.4893	1767.1459	1609.2452	0	0	0	0

$S_0^B = 8357.3056$

Compute C:

C	1	2	3	4	5	6	7	8
1	353.4292	353.4292	353.4292	353.4292	62.3946	0	0	22.2412
2	1060.2875	1060.2875	1060.2875	1060.2875	178.2062	0	0	125.7275
3	1627.0147	1767.1459	1767.1459	758.3552	38.364	0	0	153.3091

$S_0^C = 12154.7711$

Compute D:

D	1	2	3	4	5	6	7	8
1	0	0	292.0189	353.4292	353.4292	329.5552	0	0
2	0	0	957.3803	1060.2875	1060.2875	988.6656	0	0
3	0	0	1700.6773	1767.1459	1767.1459	1647.6654	0	0

$S_0^D = 12277.6879$

Compute E:

E	1	2	3	4	5	6	7	8
1	0	0	0	0	0	0	353.4292	281.7886
2	0	0	0	0	0	0	1060.2875	845.7307
3	0	0	0	0	0	0	1767.1459	1521.6857

$S_0^E = 5830.0676$

Compute A':

A'	1	2	3	4	5	6	7	8
1	0	0	305.9782	353.4292	151.4614	0	0	0
2	0	0	917.9346	974.8939	23.1797	0	0	0
3	0	0	1530.0966	1655.3881	0	0	0	0

$S_0^{A'} = 5912.3617$

Compute B':

B'	1	2	3	4	5	6	7	8
1	353.4292	353.4292	353.4292	353.4292	0	0	0	263.7518
2	372.7313	1015.9978	1060.2875	1060.2875	44.4163	0	0	63.2562
3	0	1027.2213	1767.1459	1767.1459	268.2925	0	0	0

$S_0^{B'} = 10124.2508$

Compute C':

C'	1	2	3	4	5	6	7	8
1	353.4292	353.4292	353.4292	353.4292	120.4818	0	0	0
2	1060.2875	1060.2875	1060.2875	1060.2875	323.711	0	0	144.3669
3	1475.7916	1767.1459	1767.1495	734.8241	33.1693	0	0	182.7585

$S_0^{C'} = 12204.2654$

Compute D':

D'	1	2	3	4	5	6	7	8
1	0	179.5757	353.4292	353.4292	353.4292	353.4292	47.451	0
2	0	389.751	1060.2875	1060.2875	1060.2875	1060.2875	142.353	0
3	0	388.1903	1767.1459	1767.1459	1767.1459	1767.1459	235.7776	0

$S_0^{D'}$ = 14106.5490

Compute E':

E'	1	2	3	4	5	6	7	8
1	0	0	0	0	0	0	353.4292	288.6251
2	0	0	0	0	0	0	1060.2875	866.6082
3	1.6086	0	0	0	0	0	1767.1459	1663.7782

$S_0^{E'}$ = 6001.4827

Compute F':

F'	1	2	3	4	5	6	7	8
1	0	0	0	0	353.4292	353.4292	57.2376	0
2	0	0	0	6.3236	1060.2875	1060.2875	171.7127	0
3	0	0	0	129.6097	1767.1459	1767.1459	286.1878	0

$S_0^{F'}$ = 7012.7966

Compute G':

G'	1	2	3	4	5	6	7	8
1	353.4292	353.4292	111.6616	0	282.5741	353.4292	353.4292	353.4292
2	1060.2875	1060.2875	312.8511	0	847.7223	1060.2875	1060.2875	1060.2875
3	1767.1459	1767.1459	345.8751	0	1412.8704	1435.1897	1734.6194	1767.1459

$S_0^{G'}$ = 18853. 3849

Compute H':

H'	1	2	3	4	5	6	7	8
1	353.4292	353.4292	0	0	0	0	0	107.1231
2	1060.2875	1060.2875	0	0	0	0	0	313.2581
3	1767.1459	1767.1459	0	0	0	0	0	368.2487

$S_0^{H'}$ = 7150.3551

Compute I':

I'	1	2	3	4	5	6	7	8
1	353.4292	353.4292	48.0267	0	0	0	0	0
2	1060.2875	1060.2875	144.0802	0	0	0	0	0
3	1767.1459	1767.1459	240.1337	0	0	0	0	0

$S_0^{I'}$ = 6793.9658

Compute J':

J’	1	2	3	4	5	6	7	8
1	353.4292	353.4292	353.4292	102.3369	0	0	0	0
2	1060.2875	1060.2875	1060.2875	773.8303	0	0	0	0
3	1767.1459	1767.1459	1767.1459	1560.0556	0	0	0	0

$S_0^{J'}$ = 11978.8106

r_1, r_2, r_3	A’	B’	C’	D’	E’	F’	G’	H’	I’	J’
A	0.0331	0.1508	0.2985	0.3047	0.4855	0.1718	0.9105	0.5137	0.4658	0.2620
B	0.1356	0.0694	0.2199	0.3488	0.5643	0.2622	0.8107	0.4113	0.3633	0.1596
C	0.3228	0.1961	0.0178	0.5141	0.6897	0.4388	0.6234	0.2460	0.2217	0.0809
D	0.2512	0.3739	0.5238	0.0719	0.7183	0.2287	0.7465	0.7635	0.7155	0.5299
E	0.4614	0.6013	0.6830	0.7500	0.0067	0.5002	0.5143	0.4481	0.4961	0.6998

B.2 MAP 2

Compute A:

A	1	2	3	4	5	6	7	8
1	293.6330	0	0	0	0	0	340.2050	353.4292
2	655.3045	0	0	372.9559	372.9559	0	1011.8249	1060.2875
3	177.8120	0	0	1754.7708	1754.7708	290.2638	1686.3749	1767.1459

S_0^A = 11891.7341

Compute B:

B	1	2	3	4	5	6	7	8
1	0	0	0	0	59.7962	353.4292	353.4292	116.1699
2	0	0	0	0	83.3930	1000.1532	1060.2875	407.9902
3	0	0	0	244.7156	244.7156	1045.8774	1767.1459	1303.9627

S_0^B = 8041.0656

Compute C:

C	1	2	3	4	5	6	7	8
1	147.9533	0	0	237.2592	353.4292	353.4292	353.4292	353.4292
2	320.0626	0	0	605.5861	1060.2875	1060.2875	1060.2875	1060.2875
3	167.6406	0	0	95.0402	1353.4236	1767.1459	1767.1459	1767.1459

S_0^C = 13883.2701

Compute D:

D	1	2	3	4	5	6	7	8
1	0	0	0	0	205.4759	353.4292	353.4292	273.9407
2	0	0	0	1.4597	849.4542	1060.2875	1060.2875	821.8222
3	0	0	0	222.2248	1767.1459	1767.1459	1767.1459	733.8234

S_0^D = 11237.072

Compute E:

E	1	2	3	4	5	6	7	8
1	353.4292	346.6879	0	0	0	0	0	18.6053
2	1060.2875	1040.0637	0	0	0	0	0	55.8160
3	1767.1459	1767.1459	0	0	0	0	0	93.0266

$S_0^E = 6502.208$

Compute A':

A'	1	2	3	4	5	6	7	8
1	137.6791	0	0	0	0	0	353.4292	353.4292
2	201.7483	0	0	178.6146	178.6146	0	1060.2875	1060.2875
3	121.6068	0	140.2068	1678.6804	1678.6804	140.2068	1767.1459	1767.1459

$S_0^{A'} = 10817.763$

Compute B':

B'	1	2	3	4	5	6	7	8
1	0	0	0	0	215.7501	353.4292	353.4292	287.3110
2	0	0	0	0	64.1779	886.3554	1060.2875	905.9358
3	83.3975	0	0	275.7288	275.7288	881.6188	1767.1459	1763.4087

$S_0^{B'} = 9173.7046$

Compute C':

C'	1	2	3	4	5	6	7	8
1	147.9533	0	0	77.6662	353.4292	353.4292	353.4292	353.4292
2	320.0626	0	0	163.2670	1060.2875	1060.2875	1060.2875	1060.2875
3	167.6406	0	0	6.4149	1110.2356	1767.1459	1767.1459	1767.1459

$S_0^{C'} = 12949.5447$

Compute D':

D'	1	2	3	4	5	6	7	8
1	0	0	0	0	205.4759	353.4292	353.4292	273.9407
2	0	0	0	0	739.1207	1060.2875	1060.2875	821.8222
3	0	0	0	0	1566.9411	1767.1459	1767.1459	1255.0003

$S_0^{D'} = 11224.0261$

Compute E':

E'	1	2	3	4	5	6	7	8
1	353.4292	353.4292	0	0	0	0	0	0
2	1060.2875	1060.2875	0	0	0	0	0	0
3	1767.1459	1767.1459	0	0	0	0	0	0

$S_0^{E'} = 6361.7252$

Compute F':

F’	1	2	3	4	5	6	7	8
1	0	0	0	0	0	0	353.4292	353.4292
2	0	0	0	0	0	0	1060.2875	1060.2875
3	0	0	0	0	0	0	1767.1459	1767.1459

$S_0^{F'} = 6361.7252$

Compute G’:

G’	1	2	3	4	5	6	7	8
1	0	0	0	0	353.4292	353.4292	0	0
2	0	0	0	0	1060.2875	1060.2875	0	0
3	169.5959	35.2596	0	0	1767.1459	1767.1459	0	0

$S_0^{G'} = 6566.5807$

Compute H’:

H’	1	2	3	4	5	6	7	8
1	0	0	0	79.4884	353.4292	353.4292	29.1425	0
2	0	0	0	238.4653	1060.2875	1060.2875	87.4274	0
3	0	0	0	376.1920	1767.1459	1767.1459	145.7124	0

$S_0^{H'} = 7318.1532$

Compute I’:

I’	1	2	3	4	5	6	7	8
1	0	0	294.3815	353.4292	353.4292	303.4215	0	0
2	0	0	883.1446	1060.2875	1060.2875	818.9686	0	0
3	0	0	1471.9077	1767.1459	1767.1459	1504.0077	0	0

$S_0^{I'} = 11637.5568$

Compute J’:

J’	1	2	3	4	5	6	7	8
1	353.4292	353.4292	353.4292	353.4292	0	0	353.4292	353.4292
2	743.8863	1060.2875	1060.2875	1060.2875	0	0	1060.2875	743.8863
3	25.7217	1522.3313	1767.1459	1767.1459	0	0	1522.3313	25.7217

$S_0^{J'} = 14479.8956$

B.3 Per-Ring Shape Vector Distance

Only r_1	A’	B’	C’	D’	E’	F’	G’	H’	I’	J’
A	0.0074	0.0385	0.0303	0.0344	0.0553	0.0574	0.1017	0.0601	0.0540	0.0380
B	0.0393	0.0180	0.0123	0.0540	0.0732	0.0780	0.0725	0.0283	0.0222	0.0062
C	0.0340	0.0119	0.0032	0.0488	0.0824	0.0840	0.0728	0.0336	0.0292	0.0132
D	0.0214	0.0674	0.0523	0.0123	0.0774	0.0286	0.0803	0.0842	0.0781	0.0669
E	0.0568	0.0702	0.0853	0.0857	0.0003	0.0505	0.0600	0.0485	0.0546	0.0707

Only r_2	A’	B’	C’	D’	E’	F’	G’	H’	I’	J’
A	0.0172	0.0957	0.1330	0.1397	0.2133	0.1462	0.4077	0.2325	0.2084	0.1260
B	0.0868	0.0500	0.0873	0.1720	0.2590	0.1957	0.3408	0.1629	0.1389	0.0564
C	0.1303	0.0435	0.0138	0.2037	0.3268	0.2551	0.2902	0.1313	0.1188	0.0364
D	0.1049	0.2073	0.2073	0.0400	0.3130	0.0962	0.3251	0.3396	0.3156	0.2461
E	0.2070	0.2822	0.3395	0.3370	0.0011	0.2157	0.2390	0.1945	0.2185	0.3010

Only r3	A’	B’	C’	D’	E’	F’	G’	H’	I’	J’
A	0,0331	0,1482	0,2909	0,3047	0,4855	0,1508	0,9105	0,5137	0,4658	0,2620
B	0,1356	0,0694	0,1911	0,3488	0,5641	0,2457	0,8107	0,4113	0,3633	0,1596
C	0,3091	0,1638	0,0157	0,5141	0,6897	0,4110	0,6234	0,2460	0,2217	0,0387
D	0,2501	0,3739	0,5238	0,0719	0,7183	0,2216	0,7390	0,7635	0,7155	0,5299
E	0,4614	0,6013	0,6830	0,7500	0,0067	0,5002	0,5143	0,4481	0,4961	0,6998

Appendix C
Confidence & Acc@K Calculation

C.1 MAP 1

Only r_1	1st smallest	2nd smallest
A	A’ (0.0074)	C’ (0.0303)
B	J’ (0.0062)	C’ (0.0123)
C	C’ (0.0032)	B’(0.0119)
D	D’ (0.0123)	A’ (0.0214)
E	E’ (0.0003)	H’ (0.0485)

Only r_2	1st smallest	2nd smallest
A	A’ (0.0172)	B’ (0.0957)
B	B’ (0.0500)	J’ (0.0564)
C	C’ (0.0138)	J’ (0.0364)
D	D’ (0.0400)	F’ (0.0962)
E	E’ (0.0011)	H’ (0.1945)

Only r_3	1st smallest	2nd smallest
A	A’ (0.0331)	B’ (0.1482)
B	B’ (0.0694)	A’ (0.1356)
C	C’ (0.0157)	J’ (0.0387)
D	D’ (0.0719)	F’ (0.2216)
E	E’ (0.0067)	H’ (0.4481)

r_1, r_2, r_3	1st smallest	2nd smallest
A	A’ (0.0331)	B’ (0.1508)
B	B’ (0.0694)	A’ (0.1356)
C	C’ (0.0178)	J’ (0.0809)
D	D’ (0.0719)	F’ (0.2287)
E	E’ (0.0067)	H’ (0.4481)

Appendix D

D.1 CoarseMatching Implement Using C++

```
#include <iostream>
#include <fstream>
#include <string>
#include <sstream>
#include <vector>
#include <cmath>
#include <limits>
#include <iomanip>
#include <algorithm>
#include <numeric>

using namespace std;

struct DistanceInfo {
    string refPos;
    double distance;
};

bool isImage = true;
string getPosition(const string& line) {
    size_t space_pos = line.find(' ');
    string pos = line.substr(1, space_pos - 1);
    isImage = (pos.find("'") == string::npos);
    return pos;
}

vector<double> getData(const string& line) {
    size_t space_pos = line.find(' ');
    string content = line.substr(space_pos + 1);
    vector<double> result;
    stringstream ss(content);
    double value;
    while (ss >> value) {
        result.push_back(value);
    }
    return result;
}

int main() {
    ifstream file("data_map1_full_3_circles_23_08.txt");
    if (!file.is_open()) {
        cerr << "Khong the mo file!" << endl;
        return 1;
    }

    vector<string> imPos;
    vector<string> refPos;
    vector<vector<double>> imData;
    vector<vector<double>> refData;

    string line;
    while (getline(file, line)) {
        if (!line.empty()) {
            string pos = getPosition(line);
            if (isImage) {
                imPos.push_back(pos);
                imData.push_back(getData(line));
            } else {
                refPos.push_back(pos);
                refData.push_back(getData(line));
            }
        }
    }
    file.close();

    double pi = 3.1416;
    double r3 = 90;
    double area = pi * r3 * r3;

    vector<double> all_confidence_scores;

    for (int i = 0; i < imData.size(); ++i) {
        vector<DistanceInfo> distances;

        for (int j = 0; j < refData.size(); ++j) {
            double sum = 0.0;
            for (int k = 0; k < imData[i].size(); ++k) {
                sum += fabs(imData[i][k] - refData[j][k]);
            }
            double dis = sum / area;
            dis = round(dis * 10000.0) / 10000.0;
            distances.push_back({refPos[j], dis});
        }

        sort(distances.begin(), distances.end(), [](const DistanceInfo& a, const
            DistanceInfo& b) {
            return a.distance < b.distance;
        });

        double confidence_score = 0.0;
        if (distances.size() >= 2) {
            double d1 = distances[0].distance;
            double d2 = distances[1].distance;
            if (d2 > 1e-9) {
                confidence_score = (d2 - d1) / d2;
            }
        }

        all_confidence_scores.push_back(confidence_score);


        cout << "==================================================" << endl;
        cout << "Analysis for image point: " << imPos[i] << endl;
        cout << "-> Best Match: " << (distances.empty() ? "N/A" : distances[0].refPos)
             << " (distance: " << fixed << setprecision(4) << (distances.empty() ? 0.0 :
                 distances[0].distance) << ")" << endl;
        cout << "-> Confidence Score: " << fixed << setprecision(2) << confidence_score
```

```
             << " (" << (confidence_score * 100.0) << "%)" << endl;
        cout << "------------------------------------------------------------" << endl;
        cout << left << setw(10) << "Rank" << setw(15) << "Ref Point" << "Distance" <<
            endl;
        cout << "------------------------------------------------------------" << endl;

        for (int k = 0; k < distances.size(); ++k) {
            cout << left << setw(10) << k + 1
                 << setw(15) << distances[k].refPos
                 << fixed << setprecision(4) << distances[k].distance << endl;
        }
        cout << endl << endl;
    }

    double confidence_mean = 0.0;
    double confidence_median = 0.0;

    if (!all_confidence_scores.empty()) {
        double sum_of_scores = std::accumulate(all_confidence_scores.begin(), all_
            confidence_scores.end(), 0.0);
        confidence_mean = sum_of_scores / all_confidence_scores.size();

        sort(all_confidence_scores.begin(), all_confidence_scores.end());
        size_t n = all_confidence_scores.size();
        if (n % 2 == 1) {
            confidence_median = all_confidence_scores[n / 2];
        } else {
            confidence_median = (all_confidence_scores[n / 2 - 1] + all_confidence_scores
                [n / 2]) / 2.0;
        }
    }
    cout << "############################################################" << endl;
    cout << "#                    OVERALL STATISTICS                    #" << endl;
    cout << "############################################################" << endl;
    cout << left << setw(25) << "Confidence Mean:" << fixed << setprecision(2) <<
        confidence_mean << endl;
    cout << left << setw(25) << "Confidence Median:" << fixed << setprecision(2) <<
        confidence_median << endl;
    cout << "############################################################" << endl;
    return 0;
}
```

D.2 ClassifyVertices Implement Using Python

```
import re
import sys

def polygon_signed_area(coords):
    area = 0.0
    n = len(coords)
    for i in range(n):
        x1, y1 = coords[i]
        x2, y2 = coords[(i + 1) % n]
        area += x1 * y2 - x2 * y1
    return area / 2.0

def load_buildings_from_txt_named(filename):
    buildings = {}
    current_name = None
    points_named = []
    line_pattern = re.compile(
        r"^\s*([A-Za-z0-9_\-']+)\s*=\s*\(\s*([+-]?\d+(?:\.\d+)?)\s*,\s*([+-]?\d (?:\.\d+)
            ?)\s*\)\s*,?\s*$"
    )

    with open(filename, "r", encoding="utf-8") as f:
        for line_no, raw in enumerate(f, start=1):
            line = raw.rstrip("\n")
            if not line.strip():
                continue
            stripped = line.strip()
            if stripped.endswith(":"):
```

```python
                if current_name and points_named:
                    buildings[current_name] = points_named
                current_name = stripped[:-1].strip()
                points_named = []
                continue

            m = line_pattern.match(line)
            if not m:
                continue

            if current_name is None:
                raise ValueError(
                    f"Vertex found outside a building section at line {line_no}: "
                    f"{line.strip()!s}. Each vertex must appear under a named building header."
                )

            vname = m.group(1)
            x = float(m.group(2))
            y = float(m.group(3))
            points_named.append((vname, (x, y)))

    if current_name and points_named:
        buildings[current_name] = points_named

    if not buildings:
        raise ValueError(f"No building sections with names found in file: {filename}")

    return buildings

def classify_vertices(vertices, incomplete=False, eps=1e-9):
    if not isinstance(vertices, list) or any(
        not (isinstance(v, tuple) and len(v) == 2 and isinstance(v[0], str)
             and isinstance(v[1], (tuple, list)) and len(v[1]) == 2)
        for v in vertices):
        raise ValueError("Input must be list of (name, (x,y)) tuples")

    names = [n for n, _ in vertices]
    coords = [tuple(p) for _, p in vertices]
    n = len(coords)

    if not incomplete and n < 3:
        raise ValueError("Closed polygon needs at least 3 vertices.")

    if not incomplete:
        area = polygon_signed_area(coords)
        orientation = "CCW" if area > eps else ("CW" if area < -eps else "degenerate")
    else:
        area = 0.0
        orientation = "CW"

    convex = []
    concave = []
    flat_collinear = []
    endpoints = []

    idx_range = range(n)
    if incomplete:
        endpoints.extend([names[0], names[-1]])
        idx_range = range(1, n-1)

    for i in idx_range:
        A = coords[i - 1]
        B = coords[i]
        C = coords[(i + 1) % n] if not incomplete else coords[i + 1]
        cross = (C[0] - B[0]) * (A[1] - B[1]) - (C[1] - B[1]) * (A[0] - B[0])

        if abs(cross) <= eps:
            flat_collinear.append(names[i])
        else:
            if orientation == "CCW":
                t_convex = cross > 0
            elif orientation == "CW":
                t_convex = cross < 0
            else:
                t_convex = cross > 0

            if t_convex:
                convex.append(names[i])
            else:
                concave.append(names[i])

    return {
        "orientation": orientation,
        "area": area,
        "convex": convex,
        "concave": concave,
        "flat_collinear": flat_collinear,
        "endpoints": endpoints
    }

def _format_no_quotes(lst):
    """Return string like [A', B, 3] with no quotes around elements"""
    if not lst:
        return "[]"
    return "[" + ", ".join(str(x) for x in lst) + "]"

def main(filename="buildings(with_name).txt", show_building_name=True):
    buildings = load_buildings_from_txt_named(filename)
    for bname, named_vertices in buildings.items():
        is_incomplete = "incomplete" in bname.lower()
        res = classify_vertices(named_vertices, incomplete=is_incomplete)
        if show_building_name:
            print(f"\n{bname}")
        print(f"'Convex' names: {_format_no_quotes(res['convex'])}")
        print(f"'Concave' names: {_format_no_quotes(res['concave'])}")
        print(f"'Flat/collinear' names: {_format_no_quotes(res['flat_collinear'])}")
        print(f"'Endpoints' names: {_format_no_quotes(res['endpoints'])}")

if __name__ == "__main__":
    fname = sys.argv[1] if len(sys.argv) > 1 else "buildings(with_name).txt"
    main(fname)
```