iJRASET
International Journal For Research in Applied Science and Engineering Technology

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 13 **Issue:** IX **Month of publication:** September 2025

DOI: https://doi.org/10.22214/ijraset.2025.74409

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 13 Issue IX Sep 2025- Available at www.ijraset.com

IFC Element Positioning and Labeling with Respect to Axes: A Reproducible Methodology Using xBIM

J.N. Zaragoza-Grifé[1], P. Damián-Reyes[2], R.C. López Sánchez[3], A.C. Cabrera Pérez[4]
[1]*Computational Systems PhD Student at DaVinci University EdTech Group, México*[2]
[2]*Professor at DaVinci University EdTech Group, México*
[3,4]*Autonomous University of Yucatán, México*

Abstract: ***This paper presents a unified algorithm for the automatic labeling of BIM elements on floor plans with respect to IFC axis grids. Axes from IfcGrid/IfcGridAxis (letters, numbers, or mixed tags such as [LT-3] or [G'']) are grouped by orientation, assigned a canonical normal, and indexed using an order scalar derived from the projection of axis midpoints. With this one-dimensional indexing, the algorithm identifies point elements (axis intersections), linear elements (nearest host plus orthogonal range), and area elements (ranges in two axis families) without user intervention. The system automatically selects the classification path (area → line → point) according to element size.***
The geometric core relies on projection and bracketing, selecting bounding axes by applying tolerance and proximity criteria. Robustness is achieved through automatic flipping of normal vectors, endpoint selection by 1D distance, and canonical ascending label formatting ([A–B], [9–11], [LT-2–LT-3]). The implementation in .NET (VB) employs xBIM for IFC parsing and footprint extraction. Evaluation on synthetic and real IFC datasets measures label accuracy, endpoint accuracy, and projection error, with ablation studies on tolerance and auto-flip. Limitations include curvilinear grids, nearly parallel families, and complex footprints. The approach is simple, interpretable, and reproducible, supporting traceability in BIM workflows.
Keywords: Automatic grid-based labeling, BIM/IFC (IfcGrid, IfcGridAxis), Projection and bracketing with tolerance, xBIM, .Net (VB)

I. INTRODUCTION

Architectural and engineering drawings are traditionally organized by axis grids, where intersections serve as references for locating columns, beams, walls, and slabs. In BIM (IFC) models, this grid is often explicitly represented as IfcGrid / IfcGridAxis, while structural elements appear with their 3D geometry. Nevertheless, automatically deriving human-readable tags such as [A–B, 9–11], [A, 5–8] or [B–4] remains a common task, often performed manually or handled by fragile routines that are sensitive to tolerances, slight rotations, non-numeric labels (e.g., [B'], [G''], [LT-3]), or geometric discretization. This study introduces a tolerance-aware automatic labeling algorithm that, given the model axes and the floor-plan projection of an element, produces a canonical and human-readable label without manual intervention.

The proposed approach is based on a simple principle: projecting the element geometry onto two approximately orthogonal axis families and bracketing the projection using the nearest axes. To achieve this, axes from the IfcGrid are first grouped by orientation. For each group, a canonical normal is computed, and each axis is assigned an order scalar (orderVal) corresponding to the projection of its midpoint onto that normal. This produces a one-dimensional axis per group, enabling consistent ordering and comparison of axes, even when tags are alphanumeric. Given an element—whether its footprint (polygon), longitudinal axis (segment), or centroid (point)—the algorithm projects it onto the selected group normal, automatically detects the correct sign (auto-flip if all projections fall to one side), and selects the axes that best bracket the projection endpoints within a configurable metric tolerance.

The algorithm unifies three use cases without requiring the user to predefine the element type: (1) point elements (e.g., columns), labeled by the intersection of two axes; (2) linear elements (e.g., beams, walls), labeled by the nearest "host" axis plus a segment along the orthogonal group; and (3) area elements (e.g., slabs, floors, roofs), labeled by ranges across the two orthogonal families. In addition, it addresses two typical error sources: (i) sign inconsistencies between element and axis projections, resolved through auto-flip of the group normal, and (ii) unstable decisions near boundaries, resolved by comparing proximity to projected endpoints and applying tolerance-based containment rules.

II. BACKGROUND AND RELATED WORK

A. Grids and Axes in IFC

In IFC, a grid (IfcGrid) is a planar geometric framework used to locate elements; its individual axes (IfcGridAxis) are 2D curves contained within the XY plane of the grid system and are typically organized into U/V families (with an optional W family for three-direction grids). This modeling is formalized in the buildingSMART specification and establishes, among other rules, that each IfcGridAxis belongs to exactly one of the collections UAxes, VAxes, or WAxes. These definitions enable the systematic extraction of axes and their use as positioning references in floor plans, as described in [1], [2], and [3].

For reading and traversing these entities in open-source software on the .NET platform, the xBIM toolkit [4] provides APIs for IFC2x3 and IFC4 [5], along with type filtering and navigation/query utilities, making it suitable for reproducible implementations of the labeling pipeline based on IfcGrid/IfcGridAxis.

B. Fundamentals of Computational Geometry

Our method reduces localization to projection operations of sets onto directions (group normal vectors) and to the comparison of support functions (minimum and maximum of the dot product $\langle x, y \rangle$ in a direction u). These ideas, together with basic structures such as convex hulls and axis-aligned bounding boxes, are standard in computational geometry; a reference text that systematizes these concepts and their algorithmic analysis is provided in [6].

For the point case (validating whether a grid intersection falls "inside" the footprint), point-in-polygon tests are employed. The two classical formulations—the even–odd rule and the winding number—have been extensively studied and compared; we offer both as options depending on the geometry type and the need for numerical robustness [7].

When a grid is not explicitly available in the IFC, classical works suggest detecting dominant alignments using the Hough transform (parameters ρ–θ), which avoids slope singularities and identifies families of straight lines in vector or raster data. According to [8], this serves as an alternative for recovering "implicit" axes from 2D/3D geometry.

C. Numerical Robustness and Tolerances

Labeling on real BIM models requires robustness to errors and rounding, since small perturbations can alter discrete decisions (e.g., which axis is considered "above/below"). To address this, we employ (i) metric tolerances in comparisons, (ii) an auto-flip rule for the sign of the group normal when the entire projection falls on one side—an indication of inconsistent convention—and (iii) endpoint proximity selection to avoid spurious jumps. From a theoretical standpoint, the literature on robust geometric predicates (orientation, incircle) presented in [9] establishes techniques for consistent decisions under floating-point arithmetic, which underpin the careful handling of nearly degenerate comparisons. In addition, the notion of snap rounding and its variants (e.g., stable snap rounding) provide a framework for understanding how to "adjust" geometric entities to a grid while preserving topological properties. Although our algorithm does not rasterize geometry, the ideas of idempotence and topological consistency inspire the design of our tolerance rules and axis "snapping," as discussed in [10] and [11].

D. Position of the Present Work

In contrast to approaches that depend on the element type (column/beam/slab/wall/chain) or on ad-hoc heuristics by label (e.g., assuming letters are horizontal, and numbers are vertical), we propose a unified framework that:

- Groups axes by orientation directly from IfcGridAxis;
- Assigns each group a canonical normal and an order scalar (orderVal);
- Applies projection and proximity-based bracketing with tolerances and automatic sign detection; and
- Normalizes output into ascending order (letters, numbers, and prefix + number), maintaining independence from tag-specific conventions.

This combination of well-established components (projection/support, point-in-polygon, numerical robustness) with IFC-specific standards and the labeling for elements in floor plans is— to the best of our knowledge— a novel and practical articulation, suitable for open implementation with xBIM. To promote reproducibility, we propose an implementation based on xBIM for reading IFC and extracting IfcGrid / IfcGridAxis, and on standard geometric utilities for 2D projection. The method is independent of the visualization engine and does not require proprietary libraries for its core. Final labeling is normalized into ascending order (letters A→Z, increasing numbers, and prefix + number tags such as [LT - 2, LT - 3]), ensuring consistent outputs such as [A – B, 10 – 11] across equivalent permutations.

Main Contributions:

- A unified framework for labeling point, linear, and area elements with respect to IFC grids, using a single decision routine.
- An orientation-group indexing scheme with canonical normal vectors and order scalars (orderVal), robust to arbitrary tags (letters, numbers, and combinations).
- A tolerant and stable bracketing mechanism combining sign detection (auto-flip), endpoint proximity selection, and tolerance-based enforcement to avoid spurious jumps.
- Canonical ascending formatting of labels (alphabetic, numeric, and prefix + number), aligned with drawing practice.
- An open implementation based on IFC + xBIM, suitable for evaluation and replication.

III.PROPOSED METHODOLOGY

This section formalizes a tolerant grid-based labeling algorithm. We start with axes extracted from IfcGrid / IfcGridAxis (IFC2x3 / IFC4), whose geometry is two-dimensional and lies on the *XY* plane of the grid coordinate system; each axis belongs to one of the grid families U, V (and optionally W). For an open and reproducible implementation, we assume parsing with the xBIM Toolkit (classes IfcGrid [1], IfcGridAxis [2]), although the geometric core is reader-agnostic [5].

A. Inputs and Notation

Axis set $A = \{a_k\}$, each defined by two points in the plan view $p0_k$, $p1_k \in \mathbb{R}^2$ and a free-form tag $tag(a_k)$ (e.g., [A], [11], [LT-3]).
Metric tolerance $\tau > 0$ (in meters).
Element plan-view geometry G:

- Area: simple polygon in $\mathbb{R}^2$, $P = \{v_i\}$, i = 1..m
- Linear: segment S = [s0, s1], both s0, s1 $\in \mathbb{R}^2$
- Point: location c $\in \mathbb{R}^2$

Objective: produce a canonical human-readable label (e.g., [A–B, 10–11]; [A, 5–8]; [B–4]).

B. Axis Indexing by Orientation

1) Direction and Angle (module π): For each axis a, its unit direction is defined as $\mathbf{d}(a) = (p1-p0)/\|p1-p0\|$, with $\theta(a) = \text{atan2}(d_y, d_x)$, $\theta \in [-\pi, \pi]$. Because the underlying line is unoriented, θ and θ + π are equivalent (π-periodic).
2) 1D Angular Clustering: We group axes into sets $G = \{G_j\}$ by similarity of θ (e.g., circular k-means, fine-bin histograms, or 1D-DBSCAN). In practice, for orthogonal grids a fixed threshold $\Delta\theta \approx 5°$ suffices to separate families considered "nearly parallel".
3) Mean Direction and Canonical Group Normal: For each group G, compute the circular mean as a unit vector $\mathbf{d}_G = \Sigma(a \in G)(\cos\theta(a), \sin\theta(a)) / \|\Sigma(a \in G)(\cos(\theta(a)), \sin(\theta(a)))\|$, and define the canonical normal by a 90° rotation $\mathbf{n}_G = \text{rot90}(\mathbf{d}_G) = (-dy, dx)$. This fixes a sign convention for the group (see III.D). The reduction to projections and support functions is standard in computational geometry.
4) Per-Axis Ordering Scalar: We order axes within G along a 1D line via $\text{orderVal}(a) = \text{mid}(a) \cdot \mathbf{n}_G$, where $\text{mid}(a) = ½ (p0 + p1)$. This scalar enables consistent ordering independently of tag type.
Pseudocode (indexing):

```
INDEX_AXIS(A):
 Group by orientation → {G}
 for each G:
   d_G ← circular mean from vectors
   n_G ← rot90(d_G)
   for each a ∈ G:
     mid ← (p0 + p1)/2
     orderVal[a] ← dot(mid, n_G)
   order G by ascending orderVal
 precompute intersections between distinct groups
 return {G}, {n_G}, orderVal
```

C. Bracketing by Projection (Core)

Given a set $X \subset \mathbb{R}^2$ (polygon vertices or segment endpoints) and a direction n, the projected interval is $S_{min}(X, \mathbf{n}) = \min\{x \in X\} \langle x, n\rangle$ and $S_{max}(X, \mathbf{n}) = \max\{x \in X\} \langle x, \mathbf{n}\rangle$, i.e., the support function along ± n. Let G be a group with canonical normal $\mathbf{n}_G$ and axes $\{a_i\}$ sorted by orderVal. Tolerant bracketing selects the axes that best bound $[S_{min}, S_{max}]$.

Projection: $[S_{min}, S_{max}] \leftarrow [S_{min}(X, \mathbf{n}_G), S_{max}(X, \mathbf{n}_G)]$.

Degeneracy checks and auto-flip: If all group orderVals lie on one side of the interval within tolerance τ, i.e., $\forall a \in G$: $orderVal(a) \le S_{min} + \tau$ OR $\forall a \in G$: $orderVal(a) \ge S_{max} - \tau$, then $\mathbf{n}_G$ is inverted relative to the element. Flip $\mathbf{n}_G \leftarrow -\mathbf{n}_G$ and recompute $[S_{min}, S_{max}]$.

Nearest-axis selection with τ: $a_{min} = \arg\min_{\{a \in G\}} |orderVal(a) - S_{min}|$ and $a_{max} = \arg\min_{\{a \in G\}} |orderVal(a) - S_{max}|$, using a snap band τ: if $\exists a$ with $|orderVal(a) - S_{min}| \le \tau$, restrict the minimizer to those within-band candidates; similarly for S_{max}.

Normalization: enforce ascending geometric order. If $orderVal(a_{min}) > orderVal(a_{max})$, swap them. This makes the output independent of sign conventions and prepares the pair for canonical formatting (letters A→Z, increasing numbers, or prefix + number).

Pseudocode (bracketing):

```
BRACKET(X, G, n_G, τ):
  (s_min, s_max) ← project(X, n_G)
  if All(orderVal ≤ s_min+τ) xor All(orderVal ≥ s_max−τ):
      n_G ← − n_G
      (s_min, s_max) ← project(X, n_G)
  a_min ← axis in G minimizing |orderVal − s_min| (with τ-snap band)
  a_max ← axis in G minimizing |orderVal − s_max| (with τ-snap band)
  if a_min = a_max and |G| > 1:
      offset one of them to the coherent neighbor
  reorder(a_min, a_max) by ascending orderVal
  return (a_min, a_max)
```

This rule is stable: it couples a consistent sign (auto-flip) with nearest-axis selection under tolerance, avoiding jumps caused by rounding. For near-degenerate decisions, robust predicates (orientation/collinearity) are recommended to improve numerical reliability.

D. Cases: Area, Linear, and Point (Unified Route)

1) Area (slabs, rooms):

- Choose the pair (G1, G2) with maximal orthogonality: $\arg\min_{\{i \ne j\}} |\langle \mathbf{n}_{\{Gi\}}, \mathbf{n}_{\{Gj\}}\rangle|$.
- For each G_k, run BRACKET(P, G_k, $\mathbf{n}_{\{Gk\}}$, τ) → (a_{min}^k, a_{max}^k).
- Format each range in ascending order (see III.E) and concatenate, e.g., [A–B, 10–11].
- Complexity: O(|P| + |G1| + |G2|).

2) Linear (beams, walls):

For an in-plain segment S = [s0, s1], the label has the form: "tag(a*), RANGE(b_{min}, b_{max})", where a* is the host axis (nearest—and preferably aligned—with the element) and (b_{min}, b_{max}) are the orthogonal-family axes bracketing the projection of S.

Host selection a*: For an axis a with span [a0, a1], distance from a point p is computed as: v = a1 − a0; w = p − a0; t = clamp((w·v)/(v·v), 0, 1); c = a0 + t v; $\mathbf{d}(p, [a]) = \|p - c\|$. Pick $a^* = \arg\min_{\{a \in A\}} \min\{ \mathbf{d}(s0, [a]), \mathbf{d}(s1, [a]) \}$.

Optional alignment criterion: Let u = (s1 − s0)/‖s1 − s0‖ and $\mathbf{d}_{\{G(a)\}}$ be the group statistical mean direction. Require $|u \cdot \mathbf{d}_{\{G(a)\}}| \ge \cos\theta_{max}$ (e.g., $\theta_{max} = 15°$) to avoid nearly orthogonal hosts.

Acceptance threshold: if $d^* = \min\{ \mathbf{d}(s0, [a^*]), \mathbf{d}(s1, [a^*]) \} > \tau$, evidence is weak; degrade to the area case using the segment's bounding box or mark as "Undetermined".

3) Point (columns, piers, posts):

Precompute intersections between axes from different groups. For $a \in G_i$ and $b \in G_j$ ($i \ne j$), the intersection x_{ab} exists if the lines are not parallel. Labeling rule: if $\|c - x_{ab}\| \le \tau$, output "tag(a)–tag(b)". Otherwise, try the nearest pair (one per group) provided both distances are ≤ 2τ. For "inside" tests against a polygon (e.g., validating a column within a bay), use point-in-polygon (even–odd or winding number).

E. Canonical Range Formatting

Given a pair (α, β) with tags:

1) Both numeric (extract value, e.g., [11]→11): sort ascending → [9–11].
2) Both alphabetic (normalize apostrophes/hyphens, e.g., [G"]→[G] for comparison): lexicographic ascending → [A–B].
3) Prefix + number (e.g., [LT – 2]): sort by prefix (text) then number → [LT-2 – LT-3].
4) Mixed types: preserve geometric order (by orderVal). This aligns output with drawing practice (ascending) and decouples tag semantics from geometry.

F. Unified Auto-Type Decision Route

Given element geometry G and tolerance τ:

Pseudocode (locate):

```
LOCATE(G, τ):
  if IsPolygon(G) and Area(G) ≥ A_min:      return LOCATE_AREA(G, τ)
  if IsSegment(G) and Length(G) ≥ L_min:    return LOCATE_LINEAR(G, τ)
  if IsPoint(G) or (Small Area and Small Length):  return LOCATE_POINT(G, τ)
```

Thresholds Amin, Lmin (and τ) robustly control the area → line → point degradation.

G. Robustness and Tolerances

Tolerance τ is applied to orderVal comparisons (selection and snap), host/intersection distances, and containment checks. Auto-flip resolves opposite sign conventions between $\mathbf{n}_G$ and projected geometry. For near-degenerate comparisons (collinearity/parallelism), adaptive robust predicates (e.g., Shewchuk's orientation tests) mitigate rounding errors. Optionally, snap-rounding may regularize systematically noisy data onto a fine grid prior to indexing/projecting; stable variants preserve topology and can justify such preprocessing.

H. No IfcGrid Available: Axis Detection

If the IFC lacks IfcGrid, estimate the grid from observable plan geometry (walls, beams) via the Hough transform (ρ, θ) and cluster accumulator peaks into line families; the ρ – θ parameterization avoids slope singularities. The resulting families are then processed with the same $\mathbf{n}_G$ indexing and the bracketing of this section.

I. Complexity and Costs

1) Indexing: per-group ordering O(|G| log |G|); overall near-linear in |A|.
2) Bracketing: O(|X| + |G|) per group (|X| polygon vertices; 2 for segments).
3) Point case: intersections between dominant families O(|G1|·|G2|), computed once.

J. Operational Summary

These are the main steps for the process: Parse IFC and extract axes (xBIM) → A. Index by orientation, compute $\mathbf{n}_G$, compute orderVal, and sort axes. Auto-select case (area/linear/point). Project and bracket (with τ and auto-flip), then format ascending. (Optional) Validate point case with point-in-polygon (even–odd or winding number). The entire methodology relies on projections, dot products, and orderings—standard computational-geometry tools—paired with robust decisions under tolerance, enabling a reliable mapping from IfcGrid / IfcGridAxis to coherent plan view labels.

IV. IMPLEMENTATION

This section describes a reproducible implementation of the algorithm targeting .NET Framework 4.6.x using VB.NET with free/open-source libraries. The geometric core is graphics-engine agnostic: it only requires the axis set (2D segments) and the element's floor-plan geometry (point, segment, or polygon).

A. Software Stack and Dependencies

1) .NET Framework 4.6.x, VB.Net Programming Language.
2) xBIM Toolkit for IFC parsing (extraction of IfcGrid / IfcGridAxis entities)
3) 2D geometry: minimal types (Point2D, Vector2D, Segment2D, Polygon2D)

4) Reproducibility: the IFC reader can be swapped; the algorithm consumes only (1) a list of axes with endpoints and (2) the element's 2D footprint/segment/point.

B. Minimal 2D Types

```
' -------- Minimal 2D primitives (complete) --------
Public Structure Point2D
    Public X As Double
    Public Y As Double
    Public Sub New(xv As Double, yv As Double)
        X = xv : Y = yv
    End Sub
End Structure

Public Structure Vector2D
    Public X As Double
    Public Y As Double
    Public Sub New(xv As Double, yv As Double)
        X = xv : Y = yv
    End Sub
    Public Function Length() As Double
        Return Math.Sqrt(X * X + Y * Y)
    End Function
    Public Sub Normalize()
        Dim L = Length()
        If L > 0 Then X /= L : Y /= L
    End Sub
    Public Shared Function Dot(a As Vector2D, b As Vector2D) As Double
        Return a.X * b.X + a.Y * b.Y
    End Function
End Structure

Public Class Segment2D
    Public P0 As Point2D
    Public P1 As Point2D
    Public Sub New(a As Point2D, b As Point2D)
        P0 = a : P1 = b
    End Sub
End Class

Public Class Polygon2D
    Public ReadOnly Points As List(Of Point2D)
    Public Sub New(pts As IEnumerable(Of Point2D))
        Points = New List(Of Point2D)(pts)
    End Sub
    Public Sub New(ParamArray pts() As Point2D)
        Points = New List(Of Point2D)(pts)
    End Sub
End Class

' AxisData as input from IFC (swappable reader)
```

ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 13 Issue IX Sep 2025- Available at www.ijraset.com

```
Public Class AxisData
   Public Property Tag As String
   Public Property Start As Point2D
   Public Property [End] As Point2D
   Public Sub New(t As String, s As Point2D, e As Point2D)
      Tag = t : Start = s : [End] = e
   End Sub
End Class
```

C. Data Structures

```
Friend Class Axis2D
   Public Property tag As String
   Public Property p0 As Point2D
   Public Property p1 As Point2D
   Public Property dir As Vector2D          ' unit direction (π-periodic)
   Public Property groupId As Integer       ' orientation group id
   Public Property orderVal As Double       ' <midpoint, group normal>
End Class

Friend Class AxisIntersection
   Public Property tagA As String
   Public Property tagB As String
   Public Property pt As Point2D
End Class

Friend Class AxisIndex
   Public Property Axes As New List(Of Axis2D)
   Public Property Groups As New Dictionary(Of Integer, List(Of Axis2D))   ' axes per orientation group (sorted by orderVal)
   Public Property GroupNormals As New Dictionary(Of Integer, Vector2D)    ' canonical normal per group
   Public Property Intersections As New List(Of AxisIntersection)          ' precomputed pairwise intersections
End Class
```

D. Index Construction

Given a raw list of axes (AxisData) extracted from IFC file as IfcGrid / IfcGridAxis entities, we build an AxisIndex with orientation groups, a canonical normal per group, and an orderVal per axis.

```
Friend Class AxisIndexer

   Public Shared Function BuildIndex(oAxesRaw As IEnumerable(Of AxisData),
                                     oAngThresholdDeg As Double) As AxisIndex
      Dim oIdx As New AxisIndex()

      ' 1) Normalize directions and collect axes
      For Each oA In oAxesRaw
         Dim oAx As New Axis2D()
         oAx.tag = oA.Tag
         oAx.p0 = New Point2D(oA.Start.X, oA.Start.Y)
         oAx.p1 = New Point2D(oA.[End].X, oA.[End].Y)
```



```
    Dim oDir As New Vector2D(oAx.p1.X - oAx.p0.X, oAx.p1.Y - oAx.p0.Y)
    oDir.Normalize()

    ' π-periodic: push to a consistent half-plane
    If oDir.X < 0 OrElse (Math.Abs(oDir.X) < 1.0E-12 AndAlso oDir.Y < 0) Then
        oDir.X = -oDir.X : oDir.Y = -oDir.Y
    End If

    oAx.dir = oDir
    oIdx.Axes.Add(oAx)
Next

' 2) Cluster by angular threshold
Dim angRad As Double = oAngThresholdDeg * Math.PI / 180.0
Dim groups As New List(Of List(Of Axis2D))()

For Each ax In oIdx.Axes
    Dim placed As Boolean = False
    For Each g In groups
        Dim rep As Axis2D = g(0)
        Dim cosang As Double = Vector2D.Dot(ax.dir, rep.dir)
        Dim ang As Double = Math.Acos(Math.Max(-1.0, Math.Min(1.0, cosang)))
        If ang <= angRad Then g.Add(ax) : placed = True : Exit For
    Next
    If Not placed Then groups.Add(New List(Of Axis2D)() From {ax})
Next

' 3) Per group: canonical normal, orderVal, sorting
Dim gid As Integer = 0
For Each g In groups
    Dim sx As Double = 0, sy As Double = 0
    For Each ax In g
        sx += ax.dir.X : sy += ax.dir.X
    Next
    Dim Meander As New Vector2D(sx, sy) : meanDir.Normalize()

    ' canonical normal = rot90(meanDir)
    Dim nG As New Vector2D(-meanDir.Y, meanDir.X) : nG.Normalize()
    oIdx.GroupNormals(gid) = nG

    For Each ax In g
        Dim mx As Double = 0.5 * (ax.p0.X + ax.p1.X)
        Dim my As Double = 0.5 * (ax.p0.Y + ax.p1.Y)
        ax.orderVal = mx * nG.X + my * nG.Y
        ax.groupId = gid
    Next

    g.Sort(Function(a, b) a.orderVal.CompareTo(b.orderVal))
    oIdx.Groups(gid) = g
    gid += 1
```

```
    Next

    ComputeIntersections(oIdx)
    Return oIdx
  End Function

  Private Shared Sub ComputeIntersections(oIdx As AxisIndex)
    ' Precompute intersections between groups (for point labeling)
    Dim keys = oIdx.Groups.Keys.ToArray()
    For i As Integer = 0 To keys.Length - 2
      For j As Integer = i + 1 To keys.Length - 1
        For Each a In oIdx.Groups(keys(i))
          For Each b In oIdx.Groups(keys(j))
            Dim pt As Point2D
            If TryIntersectLines(a.p0, a.p1, b.p0, b.p1, pt) Then
              oIdx.Intersections.Add(New AxisIntersection With {.tagA = a.tag, .tagB = b.tag, .pt = pt})
            End If
          Next
        Next
      Next
    Next
  End Sub

  Private Shared Function TryIntersectLines(a0 As Point2D, a1 As Point2D,
                                            b0 As Point2D, b1 As Point2D,
                                            ByRef oPt As Point2D) As Boolean
    Dim dx1 As Double = a1.X - a0.X, dy1 As Double = a1.Y - a0.Y
    Dim dx2 As Double = b1.X - b0.X, dy2 As Double = b1.Y - b0.Y
    Dim det As Double = dx1 * dy2 - dy1 * dx2
    If Math.Abs(det) < 1.0E-12 Then Return False
    Dim s As Double = ((b0.X - a0.X) * dy2 - (b0.Y - a0.Y) * dx2) / det
    oPt = New Point2D(a0.X + s * dx1, a0.Y + s * dy1)
    Return True
  End Function

End Class
```

E. Projection and Bracketing (Core)

Given a set X of points (polygon vertices or segment endpoints) and a group with canonical normal $\mathbf{n}_G$, the method projects X, checks degeneracy, and selects bracketing axes with tolerance and stabilization.

```
Friend Class Bracketer

  Public Shared Sub BracketByProjection(oIdx As AxisIndex,
                                        oGroupId As Integer,
                                        oAxes As List(Of Axis2D),
                                        oPoly As Polygon2D,
                                        oTol As Double,
                                        ByRef oMinAx As Axis2D,
```

```
                        ByRef oMaxAx As Axis2D)

    oMinAx = Nothing : oMaxAx = Nothing
    If oAxes Is Nothing OrElse oAxes.Count = 0 Then Exit Sub
    If Not oIdx.GroupNormals.ContainsKey(oGroupId) Then Exit Sub

    Dim nG As Vector2D = oIdx.GroupNormals(oGroupId)
    If nG.Length() = 0 Then Exit Sub

    Dim sMin As Double, sMax As Double
    ProjectInterval(oPoly, nG, sMin, sMax)

    Dim allBelow As Boolean = oAxes.All(Function(ax) ax.orderVal <= sMin + oTol)
    Dim allAbove As Boolean = oAxes.All(Function(ax) ax.orderVal >= sMax - oTol)

    ' Auto-flip if degenerate (all to one side)
    If allBelow Xor allAbove Then
        nG.X = -nG.X : nG.Y = -nG.Y
        ProjectInterval(oPoly, nG, sMin, sMax)
    End If

    Dim sorted = oAxes.OrderBy(Function(ax) ax.orderVal).ToList()
    oMinAx = sorted.OrderBy(Function(ax) Math.Abs(ax.orderVal - sMin)).First()
    oMaxAx = sorted.OrderBy(Function(ax) Math.Abs(ax.orderVal - sMax)).First()

    ' Stabilize if both endpoints collapse to the same axis
    If Object.ReferenceEquals(oMinAx, oMaxAx) AndAlso sorted.Count > 1 Then
        Dim i As Integer = sorted.IndexOf(oMinAx)
        If Math.Abs(oMinAx.orderVal - sMin) <= Math.Abs(oMinAx.orderVal - sMax) Then
            oMaxAx = If(i < sorted.Count - 1, sorted(i + 1), sorted(Math.Max(i - 1, 0)))
        Else
            oMinAx = If(i > 0, sorted(i - 1), sorted(Math.Min(i + 1, sorted.Count - 1)))
        End If
    End If

    If oMinAx.orderVal > oMaxAx.orderVal Then
        Dim tmp = oMinAx : oMinAx = oMaxAx : oMaxAx = tmp
    End If
End Sub

Private Shared Sub ProjectInterval(oPoly As Polygon2D,
                           nG As Vector2D,
                           ByRef sMin As Double,
                           ByRef sMax As Double)
    sMin = Double.PositiveInfinity : sMax = Double.NegativeInfinity
    For Each pt In oPoly.Points
        Dim s As Double = pt.X * nG.X + pt.Y * nG.Y
        If s < sMin Then sMin = s
        If s > sMax Then sMax = s
    Next
```

```
  End Sub

End Class
```

F. Labeling: Area, Linear, and Point

```
Friend Class LocatorIFC2D

  Public Shared Function LocatePoint(oIdx As AxisIndex,
                                     oPt As Point2D,
                                     oTol As Double,
                                     Optional oMustBeInside As Polygon2D = Nothing) As String
    Dim cand = oIdx.Intersections.Where(Function(f) Dist(oPt, f.pt) <= oTol)
    If oMustBeInside Is Not Nothing Then
      cand = cand.Where(Function(f) PolygonContains(oMustBeInside, f.pt))
    End If
    Dim hit = cand.OrderBy(Function(f) Dist(oPt, f.pt)).FirstOrDefault()
    If hit Is Not Nothing Then Return $"{hit.tagA}-{hit.tagB}"
    Dim near = oIdx.Intersections.OrderBy(Function(f) Dist(oPt, f.pt)).FirstOrDefault()
    If near Is Not Nothing AndAlso Dist(oPt, near.pt) <= 2 * oTol Then
      Return $"{near.tagA}-{near.tagB}"
    End If
    Return String.Empty
  End Function

  Public Shared Function LocateLinear(oIdx As AxisIndex,
                                      oSeg As Segment2D,
                                      oTol As Double) As String
    Dim bestAxis As Axis2D = Nothing
    Dim bestDist As Double = Double.MaxValue
    For Each kv In oIdx.Groups
      For Each ax In kv.Value
        Dim d0 As Double = DistancePointToSegment(oSeg.P0, ax.p0, ax.p1)
        Dim d1 As Double = DistancePointToSegment(oSeg.P1, ax.p0, ax.p1)
        Dim dd As Double = Math.Min(d0, d1)
        If dd < bestDist Then bestDist = dd : bestAxis = ax
      Next
    Next
    If bestAxis Is Nothing OrElse bestDist > oTol Then Return String.Empty

    Dim orthoId As Integer = GetOrthogonalGroupId(oIdx, bestAxis.groupId)
    If orthoId = -1 Then Return bestAxis.tag

    Dim segPoly As Polygon2D = SegmentBBox(oSeg)
    Dim aMin As Axis2D = Nothing, aMax As Axis2D = Nothing
    Bracketer.BracketByProjection(oIdx, orthoId, oIdx.Groups(orthoId), segPoly, oTol, aMin, aMax)

    Dim rangeTxt As String = TagFormatter.FormatRangeAscending(aMin, aMax)
    If String.IsNullOrWhiteSpace(rangeTxt) Then Return bestAxis.tag
    Return $"{bestAxis.tag}, {rangeTxt}"
```

```
End Function

Public Shared Function LocateArea(oIdx As AxisIndex,
                                  oPoly As Polygon2D,
                                  oTol As Double) As String
    If oIdx.Groups.Count = 0 Then Return String.Empty

    Dim pair As Tuple(Of Integer, Integer) = PickMostOrthogonalPair(oIdx)
    If pair Is Nothing Then
        Dim top = oIdx.Groups.OrderByDescending(Function(kv) kv.Value.Count).
                        Take(2).Select(Function(kv) kv.Key).ToArray()
        If top.Length = 0 Then Return String.Empty
        If top.Length = 1 Then
            Dim aMin As Axis2D = Nothing, aMax As Axis2D = Nothing
            Bracketer.BracketByProjection(oIdx, top(0), oIdx.Groups(top(0)), oPoly, oTol, aMin, aMax)
            Return TagFormatter.FormatRangeAscending(aMin, aMax)
        End If
        pair = Tuple.Create(top(0), top(1))
    End If

    Dim amin1 As Axis2D = Nothing, amax1 As Axis2D = Nothing
    Dim amin2 As Axis2D = Nothing, amax2 As Axis2D = Nothing

    Bracketer.BracketByProjection(oIdx, pair.Item1, oIdx.Groups(pair.Item1), oPoly, oTol, amin1, amax1)
    Bracketer.BracketByProjection(oIdx, pair.Item2, oIdx.Groups(pair.Item2), oPoly, oTol, amin2, amax2)

    Dim t1 As String = TagFormatter.FormatRangeAscending(amin1, amax1)
    Dim t2 As String = TagFormatter.FormatRangeAscending(amin2, amax2)

    If String.IsNullOrWhiteSpace(t1) Then Return t2
    If String.IsNullOrWhiteSpace(t2) Then Return t1
    Return $"{t1}, {t2}"
End Function

' --- helpers ---
Private Shared Function SegmentBBox(oSeg As Segment2D) As Polygon2D
    Dim x0 As Double = Math.Min(oSeg.P0.X, oSeg.P1.X)
    Dim y0 As Double = Math.Min(oSeg.P0.Y, oSeg.P1.Y)
    Dim x1 As Double = Math.Max(oSeg.P0.X, oSeg.P1.X)
    Dim y1 As Double = Math.Max(oSeg.P0.Y, oSeg.P1.Y)
    Return New Polygon2D({New Point2D(x0, y0), New Point2D(x1, y0),
                          New Point2D(x1, y1), New Point2D(x0, y1)})
End Function

Private Shared Function GetOrthogonalGroupId(oIdx As AxisIndex, baseId As Integer) As Integer
    If Not oIdx.GroupNormals.ContainsKey(baseId) Then Return -1
    Dim a As Vector2D = oIdx.GroupNormals(baseId) : If a.Length() > 0 Then a.Normalize()
    Dim bestId As Integer = -1 : Dim best As Double = Double.MaxValue
    For Each kv In oIdx.GroupNormals
        If kv.Key = baseId Then Continue For
```

```
        Dim b As Vector2D = kv.Value : If b.Length() > 0 Then b.Normalize()
        Dim v As Double = Math.Abs(Vector2D.Dot(a, b))
        If v < best Then best = v : bestId = kv.Key
    Next
    Return bestId
End Function

Private Shared Function PickMostOrthogonalPair(oIdx As AxisIndex) As Tuple(Of Integer, Integer)
    Dim keys = oIdx.GroupNormals.Keys.ToArray()
    If keys.Length < 2 Then Return Nothing
    Dim best As Tuple(Of Integer, Integer) = Nothing
    Dim bestVal As Double = Double.MaxValue
    For i As Integer = 0 To keys.Length - 2
        For j As Integer = i + 1 To keys.Length - 1
            Dim a As Vector2D = oIdx.GroupNormals(keys(i))
            Dim b As Vector2D = oIdx.GroupNormals(keys(j))
            If a.Length() > 0 Then a.Normalize()
            If b.Length() > 0 Then b.Normalize()
            Dim v As Double = Math.Abs(Vector2D.Dot(a, b))
            If v < bestVal Then
                bestVal = v : best = Tuple.Create(keys(i), keys(j))
            End If
        Next
    Next
    Return best
End Function

Private Shared Function DistancePointToSegment(oP As Point2D, oA As Point2D, oB As Point2D) As Double
    Dim vx As Double = oB.X - oA.X, vy As Double = oB.Y - oA.Y
    Dim wx As Double = oP.X - oA.X, wy As Double = oP.Y - oA.Y
    Dim ll As Double = vx * vx + vy * vy
    If ll <= 1.0E-16 Then
        Return Math.Sqrt((oP.X - oA.X) ^ 2 + (oP.Y - oA.Y) ^ 2)
    End If
    Dim t As Double = (wx * vx + wy * vy) / ll
    t = Math.Max(0.0, Math.Min(1.0, t))
    Dim proj As New Point2D(oA.X + t * vx, oA.Y + t * vy)
    Return Math.Sqrt((oP.X - proj.X) ^ 2 + (oP.Y - proj.Y) ^ 2)
End Function

Private Shared Function PolygonContains(oPoly As Polygon2D, oPt As Point2D) As Boolean
    ' even-odd rule
    Dim inside As Boolean = False
    Dim n As Integer = oPoly.Points.Count
    For i As Integer = 0 To n - 1
        Dim j As Integer = (i + n - 1) Mod n
        Dim xi As Double = oPoly.Points(i).X, yi As Double = oPoly.Points(i).Y
        Dim xj As Double = oPoly.Points(j).X, yj As Double = oPoly.Points(j).Y
        Dim inter As Boolean = ((yi > oPt.Y) <> (yj > oPt.Y)) AndAlso
            (oPt.X < (xj - xi) * (oPt.Y - yi) / ((yj - yi) + 1.0E-16) + xi)
```

```
            If inter Then inside = Not inside
        Next
        Return inside
    End Function

    Private Shared Function Dist(a As Point2D, b As Point2D) As Double
        Return Math.Sqrt((a.X - b.X) ^ 2 + (a.Y - b.Y) ^ 2)
    End Function

End Class
```

G. Canonical Ascending Range Formatting

```
Friend Class TagFormatter

    Public Shared Function FormatRangeAscending(a As Axis2D, b As Axis2D) As String
        If a Is Nothing AndAlso b Is Nothing Then Return String.Empty
        If a Is Nothing Then Return b.tag
        If b Is Nothing Then Return a.tag
        If String.Equals(a.tag, b.tag, StringComparison.OrdinalIgnoreCase) Then Return a.tag

        Dim cmp As Integer = CompareAxisTagsPreferred(a, b)
        Dim first As Axis2D = a, second As Axis2D = b
        If cmp > 0 Then first = b : second = a
        Return $"{first.tag}-{second.tag}"
    End Function

    ' Preferred ordering: numeric → alphabetic → prefix+number → geometric fallback (orderVal)
    Private Shared Function CompareAxisTagsPreferred(a As Axis2D, b As Axis2D) As Integer
        Dim na As Double, nb As Double
        Dim aNum As Boolean = TryParseNumeric(a.tag, na)
        Dim bNum As Boolean = TryParseNumeric(b.tag, nb)
        If aNum AndAlso bNum Then Return na.CompareTo(nb)

        Dim sa As String = NormalizeLetters(a.tag)
        Dim sb As String = NormalizeLetters(b.tag)
        Dim aAlpha As Boolean = IsAllLetters(sa)
        Dim bAlpha As Boolean = IsAllLetters(sb)
        If aAlpha AndAlso bAlpha Then
            Return StringComparer.InvariantCultureIgnoreCase.Compare(sa, sb)
        End If

        Dim pa As String, ia As Double
        Dim pb As String, ib As Double
        Dim ap As Boolean = TrySplitPrefixNumber(a.tag, pa, ia)
        Dim bp As Boolean = TrySplitPrefixNumber(b.tag, pb, ib)
        If ap AndAlso bp Then
            Dim c As Integer = StringComparer.InvariantCultureIgnoreCase.Compare(pa, pb)
            If c <> 0 Then Return c
            Return ia.CompareTo(ib)
```

```
        End If

        ' Fallback to geometric order
        Return a.orderVal.CompareTo(b.orderVal)
    End Function

    Private Shared Function TryParseNumeric(s As String, ByRef val As Double) As Boolean
        val = 0
        If String.IsNullOrWhiteSpace(s) Then Return False
        Dim t As String = New String(s.Where(Function(ch) Char.IsDigit(ch) OrElse
                                ch = "."c OrElse ch = "-"c).ToArray())
        If t.Length = 0 Then Return False
        Return Double.TryParse(t, Globalization.NumberStyles.Any,
                               Globalization.CultureInfo.InvariantCulture, val)
    End Function

    Private Shared Function NormalizeLetters(s As String) As String
        If String.IsNullOrWhiteSpace(s) Then Return String.Empty
        Return s.Replace("'", "").Replace("""", "").Replace("-", "")
    End Function

    Private Shared Function IsAllLetters(s As String) As Boolean
        If String.IsNullOrEmpty(s) Then Return False
        For Each ch As Char In s
            If Not Char.IsLetter(ch) Then Return False
        Next
        Return True
    End Function

    Private Shared Function TrySplitPrefixNumber(s As String,
                                         ByRef prefix As String,
                                         ByRef num As Double) As Boolean
        prefix = String.Empty : num = 0
        If String.IsNullOrWhiteSpace(s) Then Return False
        Dim i As Integer = s.Length - 1
        While i >= 0 AndAlso (Char.IsDigit(s(i)) OrElse s(i) = "."c)
            i -= 1
        End While
        If i = s.Length - 1 Then Return False
        prefix = s.Substring(0, i + 1).TrimEnd("-"c, "_"c, " "c, "'"c, """"c)
        Dim numStr As String = s.Substring(i + 1)
        Return Double.TryParse(numStr, Globalization.NumberStyles.Any,
                               Globalization.CultureInfo.InvariantCulture, num)
    End Function

End Class
```

H. Unified Route: LocateElement

```
Friend Class AutoLocatorIFC2D

  Public Shared Function LocateElement(oIdx As AxisIndex,
                                       oGeom As Object,
                                       oTol As Double,
                                       Optional oAreaMin As Double = 0.05,
                                       Optional oLongMin As Double = 0.20) As String

    If TypeOf oGeom Is Polygon2D Then
      Dim poly = DirectCast(oGeom, Polygon2D)
      If PolygonAreaAbs(poly) >= oAreaMin Then
        Return LocatorIFC2D.LocateArea(oIdx, poly, oTol)
      End If
      Dim seg As Segment2D = PolyLongestEdge(poly)
      If SegmentLength(seg) >= oLongMin Then
        Return LocatorIFC2D.LocateLinear(oIdx, seg, oTol)
      Else
        Dim cen As Point2D = PolyCentroid(poly)
        Return LocatorIFC2D.LocatePoint(oIdx, cen, oTol, poly)
      End If

    ElseIf TypeOf oGeom Is Segment2D Then
      Dim seg = DirectCast(oGeom, Segment2D)
      If SegmentLength(seg) >= oLongMin Then
        Return LocatorIFC2D.LocateLinear(oIdx, seg, oTol)
      Else
        Dim mid As New Point2D(0.5 * (seg.P0.X + seg.P1.X), 0.5 * (seg.P0.Y + seg.P1.Y))
        Return LocatorIFC2D.LocatePoint(oIdx, mid, oTol)
      End If

    ElseIf TypeOf oGeom Is Point2D Then
      Return LocatorIFC2D.LocatePoint(oIdx, DirectCast(oGeom, Point2D), oTol)
    End If

    Return String.Empty
  End Function

  ' --- polygon helpers ---
  Private Shared Function SegmentLength(seg As Segment2D) As Double
    Dim dx As Double = seg.P1.X - seg.P0.X
    Dim dy As Double = seg.P1.Y - seg.P0.Y
    Return Math.Sqrt(dx * dx + dy * dy)
  End Function

  Private Shared Function PolygonAreaAbs(poly As Polygon2D) As Double
    Dim s As Double = 0
    Dim n As Integer = poly.Points.Count
    For i As Integer = 0 To n - 1
      Dim j As Integer = (i + 1) Mod n
```

```
        s += poly.Points[i].X * poly.Points[j].Y - poly.Points[j].X * poly.Points[i].Y
    Next
    Return Math.Abs(0.5 * s)
End Function

Private Shared Function PolyCentroid(poly As Polygon2D) As Point2D
    Dim a As Double = 0, cx As Double = 0, cy As Double = 0
    Dim n As Integer = poly.Points.Count
    For i As Integer = 0 To n - 1
        Dim j As Integer = (i + 1) Mod n
        Dim cross As Double = poly.Points(i).X * poly.Points(j).Y - poly.Points(j).X * poly.Points(i).Y
        a += cross
        cx += (poly.Points[i].X + poly.Points[j].X) * cross
        cy += (poly.Points[i].Y + poly.Points[j].Y) * cross
    Next
    a *= 0.5
    If Math.Abs(a) < 1.0E-16 Then
        Dim sx As Double = 0, sy As Double = 0
        For Each p In poly.Points
            sx += p.X : sy += p.Y
        Next
        Return New Point2D(sx / n, sy / n)
    End If
    Return New Point2D(cx / (6 * a), cy / (6 * a))
End Function

Private Shared Function PolyLongestEdge(poly As Polygon2D) As Segment2D
    Dim best As Double = -1
    Dim sbest As Segment2D = Nothing
    Dim n As Integer = poly.Points.Count
    For i As Integer = 0 To n - 1
        Dim j As Integer = (i + 1) Mod n
        Dim p0 = poly.Points(i) : Dim p1 = poly.Points(j)
        Dim d As Double = Math.Sqrt((p1.X - p0.X) ^ 2 + (p1.Y - p0.Y) ^ 2)
        If d > best Then
            best = d : sbest = New Segment2D(p0, p1)
        End If
    Next
    Return sbest
End Function

End Class
```

I. Parameters and Recommended Values

Metric tolerance τ = 0.10 m (tune 0.075 – 0.15 m). Angular threshold $\Delta\theta = 5°$ (use 8° for slightly skewed grids). Minimum area A_{min} = 0.05 m². Minimum length L_{min} = 0.20 m. Suggested experiments = PR vs. τ; ablation closest vs. ceiling/floor; impact of $\Delta\theta$.

ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 13 Issue IX Sep 2025- Available at www.ijraset.com

J. Invariants and Tests

Group invariant: projections and orderVal use the stored canonical normal. Auto-flip if all projections fall to one side. Endpoint stability by nearest to s_{min}/s_{max} with neighbor adjustment. Canonical labeling output (A – B, 9 – 11, LT - 2 – LT - 3). Unit tests on synthetic and real IFC plans with annotated ground truth.

K. How to Use

```
' 1) Read IFC → List(Of AxisData) = (tag, p0, p1)
Dim idx = AxisIndexer.BuildIndex(oAxes, 5.0)

' 2) Prepare G (polygon/segment/point) in floor plan, τ = 0.10 m
Dim label = AutoLocatorIFC2D.LocateElement(idx, G, 0.10)   ' → "A-B, 10-11"
```

V. DISCUSSION AND EXTENSIONS

The proposed algorithm demonstrates strong performance for rectilinear grids with straight axes and varied tags. Nevertheless, several limitations and opportunities for extension remain, particularly when the geometry of the grid or the elements departs from ideal assumptions. This section discusses the main challenges and outlines extensions that can broaden applicability while preserving interpretability and reproducibility.

A first challenge arises when grids are oblique, nearly parallel, or composed of multiple families beyond the classical two. The algorithm currently assumes two nearly orthogonal families and selects the pair that minimizes the angular difference between their normal vectors. However, in cases where axis families are close to parallel or when three or more orientations are relevant, the resulting bracketing may become ambiguous. A potential extension is to evaluate all valid pairs and select the one that maximizes the average separation of projected axes, thereby increasing robustness to uneven spacing. Stability criteria, such as enforcing a minimum angular margin, can further prevent degeneracy in family selection.

Curved axes, modeled in IFC as IfcCurve entities (arcs, splines), present a second limitation. The current definition of the order scalar (orderVal) relies on the axis midpoint and a constant group normal, which may not preserve the intended “visual” ordering when axes are curved. A more general formulation involves parameterizing each axis by its curvilinear abscissa along a reference curve and computing orderVal as the median projection onto the local normal. In practice, this requires sampling a set of points along each axis and applying robust statistics such as the statistical median to stabilize the result.

Radial or polar grids introduce a third case. For these, the natural representation is in polar coordinates, with one family defined by angular directions and the other by radial offsets. An appropriate extension is to define order values directly in terms of the radius at the midpoint and the unwrapped angular coordinate, thereby allowing bracketing to operate in $\mathbb{R} \times S^1$. Unwrapping the angle avoids discontinuities at multiples of 2π and ensures continuity of ordering across the circle.

The geometry of the elements themselves also introduces challenges. Complex footprints with holes, multi-polygons, or jagged boundaries can cause bracketing to be governed by spurious extremes. Preprocessing strategies, such as applying Minkowski dilation with a tolerance radius or smoothing the contour, can mitigate noise. Other alternatives include constructing a partial convex polygon, for example using alpha-shapes or concave hulls, with the parameter tied to tolerance τ. When elements are represented as multiple polygons, bracketing should be applied to their union to avoid inconsistencies. Similarly, curved linear elements such as beams or walls require more than endpoint-based bracketing. In such cases, the “host” axis should be defined by minimizing the average distance of a set of sampled points from the element to the nearest axis, which better captures the intended alignment.

Another critical aspect is tolerance selection. While a fixed tolerance value is generally effective, it may prove suboptimal when axis spacing varies significantly across the grid. An adaptive strategy can be employed by estimating the tolerance per axis group, based on the median of one-dimensional spacing differences. This approach normalizes decisions in both dense and sparse regions, preventing false positives near boundaries while maintaining sensitivity in coarse areas. The scaling factor α, within the range [0.05, 0.25], can be tuned according to the model’s scale.

Semantic ambiguities of tags also present difficulties. Labels such as [G'], [G’’], [LT – 3] or mixed language conventions may conflict with lexicographic ordering. Although the proposed method already falls back on geometric ordering when tag types differ, ties may persist. Deterministic tie-breaking policies, such as using orderVal followed by insertion index, and formatting profiles (ascending or descending order per family, or region-specific conventions) provide a structured way to resolve these ambiguities.

In practice, some IFC models lack IfcGrid entities altogether, or include grids of insufficient quality. In these cases, detection of implicit grids becomes necessary.

Classical methods such as the Hough transformation or RANSAC can be applied to walls, beams, or edges to identify dominant alignments. Once directions are established, offsets can be quantified to estimated spacing, and the resulting structure validated through topological consistency criteria, for instance by penalizing impossible crossings and minimizing deviations from alignments. Such recovery procedures would enable the labeling algorithm to remain functional even in incomplete or poorly modeled IFC datasets.

From a computational perspective, the dominant cost of the method is sorting by orderVal and performing linear bracketing. These operations, however, lend themselves to optimization. Indexing can be cached by group in one-dimensional arrays, enabling binary search for the nearest axis. Parallelization can be implemented both at the level of elements (using task parallel libraries or PLINQ) and at the level of axis groups. Furthermore, projection operations can be vectorized using SIMD instructions on .NET, which provides significant speedups. For very large models, spatial blocking by floor sectors can be introduced to distribute computations and reduce memory overhead.

Reproducibility and licensing considerations are equally important. To ensure that the algorithm can be reused and independently verified, the publication of code and parameters (with version hashes or commit identifiers), test IFC files, and evaluation scripts is recommended. Documenting deterministic choices, such as random seeds, and dependencies, such as the IFC reader, ensures transparency. Preference should be given to open-source libraries for the algorithmic core, such as xBIM, while integrations with proprietary visualization engines should be isolated into external layers. This separation ensures that the essential logic remains accessible and reproducible across different environments.

In summary, the algorithm is effective for rectilinear grids with straight axes and heterogeneous tags, but challenges remain in handling curvature, radial layouts, uneven densities, and complex element footprints. The extensions discussed here—adaptive tolerances, curvilinear bracketing, polar coordinates, implicit grid detection, and parallelization—offer a clear path to generalizing the approach to broader scenarios. By maintaining simplicity, interpretability, and traceability, these enhancements align the method with the demands of industrial BIM workflows and academic research.

VI. CONCLUSIONS

This study introduced a unified algorithm for labeling BIM elements on floor plans with respect to IFC grids. The method is capable of handling point, linear, and area elements without user intervention. Its main contributions include an orientation-based indexing scheme with canonical normal vectors and order scalars, projection-based bracketing with endpoint proximity and auto-flip rules, explicit metric tolerances, and canonical ascending formatting of tags. Together, these components enable stable and human-readable labels even in the presence of heterogeneous tags, irregular spacings, and minor geometric misalignments.

The approach provides several practical benefits. First, robustness is achieved by combining projections with endpoint proximity, which prevents unstable jumps caused by discretization or noise, while the auto-flip rule corrects inconsistent sign conventions between axis families and element geometry. Second, neutrality with respect to tags is ensured by relying on the internal orderVal scalar, which is valid for alphanumeric and mixed labels such as A, B, G", LT-3, 11. Third, scalability is supported because indexing is performed once and labeling is linear in the number of vertices, with direct parallelization by element. Finally, the algorithm is independent of any visualization engine, requiring only 2D segments (axes) and element footprints, and can therefore be integrated into batch pipelines or interactive software.

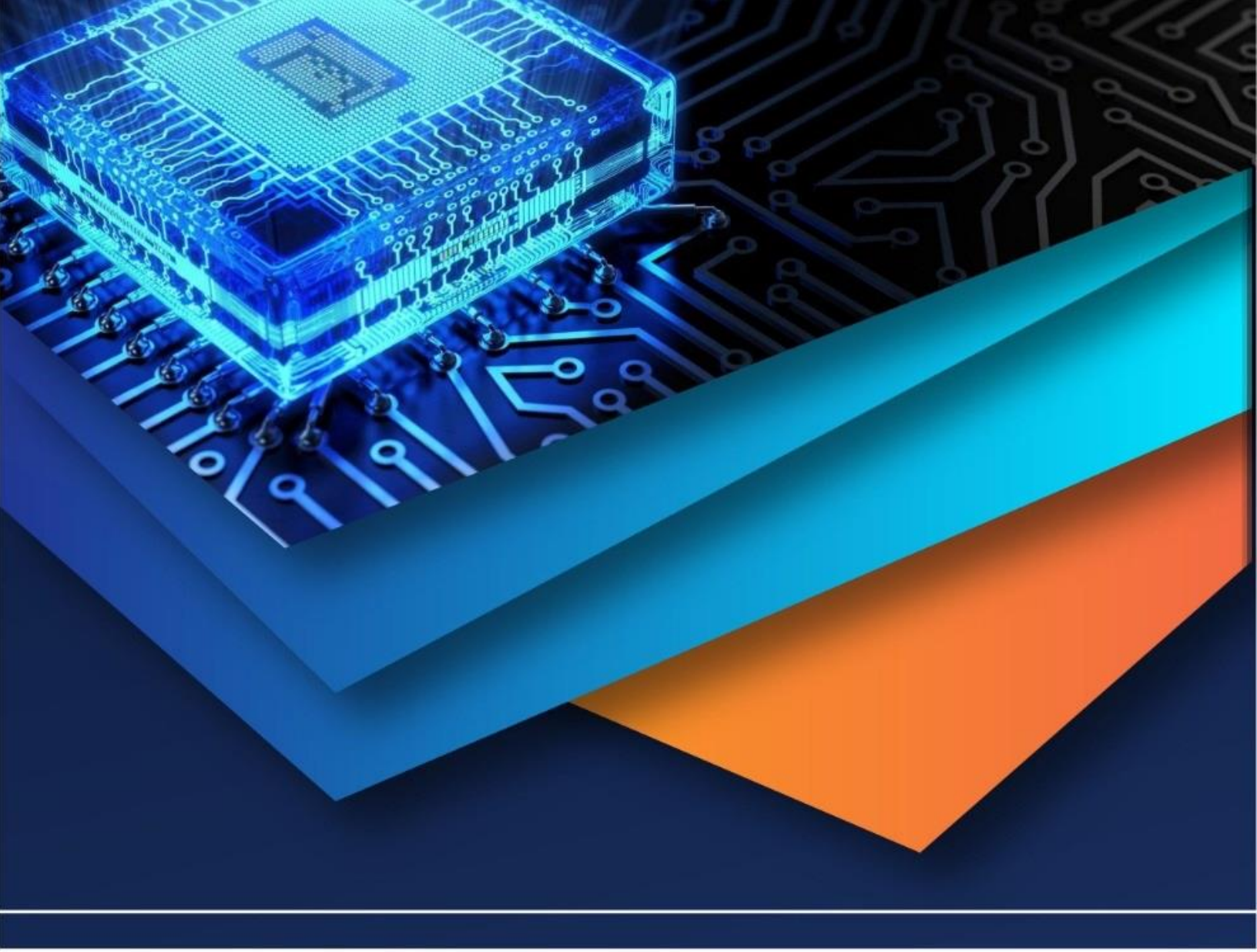
To facilitate reproducibility, we provide guidelines regarding input data (axis lists and floor-plan geometries), reference parameters (metric tolerances, angular thresholds, and minimum dimensions), and software implementation in .Net (VB) with xBIM for IFC parsing. An evaluation protocol is also suggested, combining accuracy metrics with ablation studies to assess the influence of tolerance, auto-flip, and endpoint selection.

Several research directions arise from this work. Adaptive tolerances based on axis spacing statistics may increase robustness, while extensions to curvilinear and radial grids could broaden applicability. Methods such as Hough or RANSAC can serve to recover implicit grids when IfcGrid is not available. In terms of performance, binary search on one-dimensional arrays, SIMD projection routines, and parallel execution provide promising avenues for acceleration.

Overall, the “project and bracket by proximity” approach offers a simple, interpretable, and reproducible solution to the recurring problem of grid-based labeling. Its combination of robustness, neutrality, and scalability makes it suitable for both industrial BIM workflows and academic research. Open implementations and publication of parameters, scripts, and test models will support independent verification and extension to more complex scenarios without sacrificing the traceability required in BIM environments.

REFERENCES

[1] buildingSMART International. (2024). *IfcGrid* (IFC 4.3.2 documentation). https://standards.buildingsmart.org/IFC/DEV/IFC4_2/FINAL/HTML/schema/ifcproductextension/lexical/ifcgrid.htm
[2] buildingSMART International. (2024). *IfcGridAxis* (IFC 4.3.2 documentation). https://standards.buildingsmart.org/IFC/RELEASE/IFC2x3/TC1/HTML/ifcgeometricconstraintresource/lexical/ifcgridaxis.htm
[3] buildingSMART International. (2024). *IFC Schema Specifications*. https://technical.buildingsmart.org/standards/ifc/ifc-schema-specifications/
[4] xBIM Team. (s. f.). *XbimEssentials* (GitHub repository). https://github.com/xBimTeam/XbimEssentials
[5] xBIM Project. (s. f.). *xBIM Toolkit* (site and documentation). https://docs.xbim.net
[6] de Berg, M., Cheong, O., van Kreveld, M., & Overmars, M. (2008). *Computational Geometry: Algorithms and Applications* (3.ª ed.). Springer.
[7] Hormann, K., & Agathos, A. (2001). The point in polygon problem for arbitrary polygons. *Computational Geometry*, 20(3), 131–144.
[8] Duda, R. O., & Hart, P. E. (1972). Use of the Hough transformation to detect lines and curves in pictures. *Communications of the ACM*, 15(1), 11–15.
[9] Shewchuk, J. R. (1997). Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete & Computational Geometry*, 18, 305–363.
[10] Goodrich, M. T., Guibas, L. J., Hershberger, J., & Tanenbaum, P. J. (1997). Snap rounding line segments efficiently in two and three dimensions. En *Proceedings of the 13th Annual ACM Symposium on Computational Geometry* (pp. 284–293). ACM.
[11] Hershberger, J. (2013). Stable snap rounding. *Computational Geometry*, 48(7), 575–585.



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY