Wall Extraction and Room Detection for Multi-Unit Architectural Floor Plans

by

Dany Alejandro Cabrera Vargas
B.Eng., University of Cauca, Colombia, 2015

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

Wall Extraction and Room Detection for Multi-Unit Architectural Floor Plans

by

Dany Alejandro Cabrera Vargas
B.Eng., University of Cauca, Colombia, 2015

Supervisory Committee

Dr. Alexandra Branzan Albu, Supervisor
(Department of Electrical and Computer Engineering)

Dr. Maia Hoeberechts, Departmental Member
(Department of Computer Science)

ABSTRACT

In the context of urban buildings, architectural floor plans describe a building's structure and spatial distribution. These digital documents are usually shared in file formats that discard the semantic information related to walls and rooms. This work proposes a new method to recover the structural information by extracting walls and detecting rooms in 2D floor plan images, aimed at multi-unit floor plans which present challenges of higher complexity than previous works. Our proposed approach is able to handle overlapped floor plan elements, notation variations and defects in the input image, and its speed makes it suitable for real applications on both desktop and mobile devices. We evaluate our methods in terms of precision and recall against our own annotated dataset of multi-unit floor plans.

Table of Contents

Supervisory Committee **ii**

Abstract **iii**

Table of Contents **iv**

List of Tables **vi**

List of Figures **vii**

Glossary **ix**

Acknowledgements **x**

1 Introduction **1**

1.1 Focus of Thesis 3

2 Related Work **5**

2.1 Wall Extraction Methods 5

2.1.1 Early Approaches 6

2.1.2 Traditional Machine Learning Methods 8

2.1.3 Unsupervised Multi-Notation Methods 9

2.1.4 CNN-Based Approaches 10

2.2 Room Detection Methods 12

2.2.1 Pixel-Based Methods 13

2.2.2 Geometry-Based Methods 14

2.2.3 Mixed Methods 15

2.3 Relevant Preprocessing Methods 16

3 Proposed Approach **18**

3.1 Wall Extraction 19
3.1.1 Design Assumptions 19
3.1.2 General Method Overview 19
3.1.3 Preprocessing 21
3.1.4 Slice Transform 22
3.1.5 Slice Thickness Filter 28
3.1.6 Angle Matrix Generation 31
3.1.7 Wall Segment Candidate Detection 37
3.1.8 Geometrical Analysis 46
3.2 Room Detection 51
3.2.1 Design Assumptions 51
3.2.2 General Method Overview 52
3.2.3 Wall Mask Generation 53
3.2.4 Gap Closing 54
3.2.5 Closed Region Detection 56
3.2.6 Region Filtering 57

4 Experimental Results **59**
4.1 Implementation Technical Overview 59
4.2 Evaluation of the Angle Matrix Generation 60
4.2.1 Angle Approximation Accuracy 61
4.2.2 Angle Blur Efficiency 63
4.3 Evaluation of the Wall Extraction Method 64
4.3.1 Dataset Generation 65
4.3.2 Quantitative Results 67
4.4 Evaluation of the Room Detection Method 69
4.5 Discussion on Mobile Device Implementation 72

5 Conclusions **75**
5.1 Future Directions 77

A Dataset Image Descriptions **79**

B Technical Specifications **83**

Bibliography **84**

List of Tables

Table 4.1 Angle matrix error measurements when varying line length. . . . 61
Table 4.2 Angle matrix error measurements when varying line thickness. . 62
Table 4.3 Quantitative evaluation of consecutive conditional blur runs. . . 63
Table 4.4 Wall extraction configuration parameters. 65
Table 4.5 Wall extraction quantitative evaluation results 67
Table 4.6 Room detection quantitative evaluation results 70

Table B.1 Desktop PC specifications for experiments. 83
Table B.2 Mobile device specifications for experiments. 83

List of Figures

1.1 Vector to raster image conversion. 2
1.2 Comparison of single vs. multi-unit floor plan. 3

2.1 Example of an early wall extraction method 6
2.2 Example of a 3D model generation experiment. 7
2.3 Real graphical examples of different wall notations. 8
2.4 Patch-based wall extraction method pipeline. 9
2.5 Relevant steps for wall extraction using a CNN to detect junctions. . 10
2.6 Output samples of a FCN-based wall-extraction approach. 11
2.7 Example of a pixel-based room segmentation method. 13
2.8 Example of a geometry-based room segmentation method. 14
2.9 Example of a mixed room segmentation method. 15
2.10 Text/Graphics separation example for illustration. 16

3.1 General steps of proposed wall extraction method 20
3.2 Preprocessing steps on a small floor plan section. 22
3.3 Intuitive illustration of the slice transform. 23
3.4 Parallelized execution diagram for the slice transform 27
3.5 Visualization for line thickness estimation 29
3.6 Example of line thickness histogram 30
3.7 Detail of the effects of slice thickness filtering. 30
3.8 Example of Oriented Bounding Boxes for the slices. 32
3.9 Hue colormap for angle matrix visualization 34
3.10 Angle matrix visualization of a diagonal line. 34
3.11 Example of two conditional blur cases. 35
3.12 Effect of consecutive conditional blurs. 36
3.13 Effect of conditional blur in a floor plan segment. 37
3.14 Angle intervals for line separation. 38
3.15 Example of pixel separation by angle. 39

3.16 Slice group extraction explanation. 41
3.17 Slice group extraction results. 43
3.18 Examples of difficult scanning cases. 44
3.19 Final slice groups for the example map section. 46
3.20 Parallel line segments detected in the example map section. 48
3.21 Line connection analysis results in the example map section. 50
3.22 General steps of proposed room detection method 52
3.23 Example of wall mask generation. 54
3.24 Example of gap closing using virtual walls. 55
3.25 Example of the wall structure for a bigger map section. 56
3.26 Examples of region detection steps. 57
3.27 Examples of region detection steps. 58

4.1 Test image 1: Angle matrix synthetic test. 60
4.2 Angle matrix error areas. 62
4.3 Time vs. Number of pixels plot for angle matrix experiments. 63
4.4 Blur Level vs. Error Reduction Efficiency. 64
4.5 Examples of challenges in our dataset. 66
4.6 Example of dataset ground-truth. 66
4.7 Observations in the wall extraction results. 68
4.8 Room detection GUI. 69
4.9 Error cases in the room detection evaluation. 71
4.10 Frame examples from a real-time application. 73

5.1 Example of artistic detail in a single-unit floor plan. 77

A.1 Test image 1. 79
A.2 Test image 2. 80
A.3 Test image 3. 80
A.4 Test image 4. 80
A.5 Test image 5. 81
A.6 Test image 6. 81
A.7 Test image 7. 81
A.8 Test image 8. 82
A.9 Test image 9. 82
A.10 Test image 10. 82

Glossary

We consider the following concepts in a **computer vision** context:

- **Raster image:** A raster graphics or bitmap image represents an image as a rectangular grid of square pixels of different colors.
- **Vector graphics:** Computer graphics images that are defined in geometrical terms of 2D points, which are connected by lines and curves to form polygons and other shapes.
- **Vectorization:** Also known as raster-vector conversion. The process of converting a vector graphics image to a raster image that portrays the vector image's original visual appearance.
- **Preprocessing:** The preliminary processing steps in a broader computer vision process.
- **Dataset:** A collection of images (or other visual media) and corresponding ground-truth, commonly used to train machine learning methods and evaluate computer vision algorithms.
- **Ground-Truth:** The true, objective data gathered for a particular test, usually created manually to compare against results from automated algorithms.
- **Descriptor:** Visual descriptors or image descriptors are descriptions of the visual features of specific image contents, often used in machine learning methods to summarize visual entities. They can include characteristics of shape, color, texture and motion, among many others.

Acknowledgements

I would like to thank my supervisor **Dr. Alexandra Branzan Albu** for her outstanding leadership, patience and kind support.

I'm also grateful to **Mélissa Côté** for her project leadership and to our industrial partner **Steve Cooke** from Triumph Engineering for his continued collaboration.

Special thanks to Amanda, Ali, Alex and Tunai for being my mentors, research comrades-in-arms, and most importantly, friends.

1 Introduction

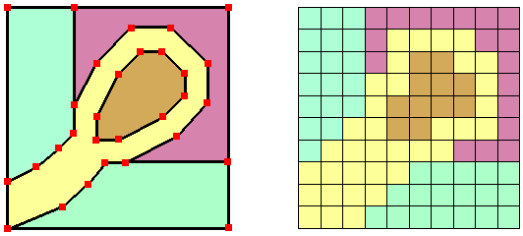
Architectural floor plans are an efficient way to describe aspects of a building using geometric and semantic information. These documents play an important role in the design and construction of buildings and serve as essential communication tools between engineers, architects and clients.

Professional architects usually elaborate floor plans using CAD (Computer Assisted Design) software such as AutoCAD [1], HomeStyler [2], or Sketchup [3] to name a few. These floor plans are internally defined and stored in terms of *geometrical primitives* like points, lines, curves and polygons (a format known as *vector graphics* [4] illustrated in Figure 1.1.a), together with corresponding semantic information.

A common purpose for these design documents is visualization for other professionals, and consequently floor plans are exported to file formats more adequate for sharing (e.g. PDF or TIFF). This process will convert the vector graphics to a raster image: A matrix representation of the original contents in the form of a rectangular grid of square pixels [5], as illustrated in Figure 1.1.b.

This *vector-to-raster* conversion obfuscates the original geometric information from the original CAD file, obstructs the use of these documents for automatic floor plan analysis tasks. Sharing the original CAD file would suffice for a solution to this problem (supposing the receiver has the required software to read it), but in practice, architecture companies might not be willing to freely share their CAD files (or the original file has been permanently lost).

Another obstacle to approaching this problem in the CAD domain, is that combining



(a) (b)

Figure 1.1: Vector to raster image conversion illustration. (a) Original vector graphics: An image built with polygons described by sets of connected points (red) to represent their boundary. (b) A (low resolution) raster image generated from (a) where each pixel has been colored to match the closest polygon region to its position.

data from different CAD software might lead to unexpected compatibility errors: Although most related software claims to be able to read and write DWG files (the native proprietary file format of Autodesk's AutoCAD, the dominant software in this sector), in practice this process often fails (files refuse to open, open but cannot be edited, or have corrupted data). There's historical reasons to believe that Autodesk is not interested in the creation of an universal, open file standard [6].

On the other hand, computer vision researchers have studied for many years the methods to understand document images (architectural floor plans included). Recent research advances have shown promising results guided by the use of machine learning (convolutional neural networks in particular), specially in the case of natural images (e.g. scenes commonly found in nature and everyday human life). However, symbolic images (those created by humans with a communication purpose in mind; most notably documents) pose a completely different set of challenges, often not approachable by methods designed for natural images.

A particular challenge in document understanding is that, contrary to many applications proper to natural images, symbolic entities carry semantic information that requires interpretation: a symbol in a floor plan can be an abstract representation of an element of totally different appearance in the real world, and requires certain prior knowledge (e.g. floor plan notation) to be understood. Symbols might be drawn using combinations of geometrical shapes that look visually similar to other unrelated symbols, requiring contextual information for their accurate interpretation.

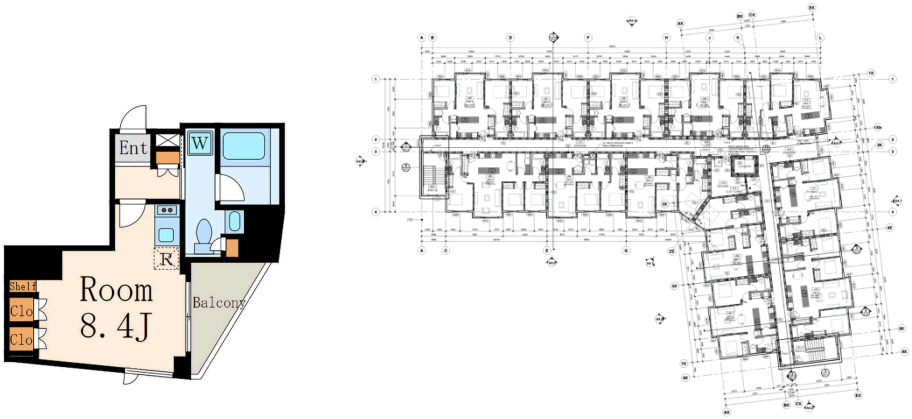
Computer vision methods cope with the complexities of interpreting symbolic images

by not only identifying the objects in the image, but also by attempting to infer their semantic relationships and integrating domain knowledge (context, language, notation) into the process of understanding. In the case of floor plans, symbols are often individually detected in separate categories (e.g. walls, rooms, text labels, symbols) and then their relationships (e.g. hierarchy, connections) and semantic properties are inferred.

Once the underlying document contents are retrieved, many automated applications are made possible: space analysis and optimization, electrical layout generation, water supply planning, to name a few.

1.1 Focus of Thesis

This research work aims to propose a method to extract walls and detect rooms in architectural floor plan images. It focuses on multi-unit floor plan images (e.g. apartment buildings) with overlapped content of the same color. An industrial collaborator provided a set of multi-unit floor plans of different authors and notation, that we used to build our own dataset for quantitative evaluation.



(a) (b)

Figure 1.2: Comparison of single vs. multi-unit floor plan. (a) A single-unit floor plan image from the R-FP dataset [7] (size: 510 x 506 px). (b) A 12-unit floor plan image from our own dataset (size: 9,427 x 5,445 px). (b) Presents a wall structure of higher complexity and portrays more information, omitting artistic detail to focus on content.

Although single-unit and multi-unit floor plans share some visual characteristics,

multi-unit floor plans convey more information within the same image; from a computer vision perspective, they are different in their size and complexity as can be observed in Figure 1.2. Our proposed approach targets specific multi-unit floor plan challenges like same-color content overlapping, line intersection and deceptive visual cues (we further describe these challenges in Section 4.3.1).

We leverage concepts and ideas found in approaches for single-unit floor plans, and adapt them to design our own approach for multi-unit floor plans.

The wall extraction and room detection modules described in this thesis are designed to be included as the first steps of a broader system for floor plan understanding, in particular for automatic electrical layout generation.

2 Related Work

The automatic analysis of architectural floor plans is an important problem that has been researched since the early days of computer vision. In floor plan images, the foreground content is usually easy to differentiate from the background (a common trait for document images) which encouraged early approaches (described in Section 2.1.1) that fit simple, small images and were limited to a single notation.

The recent increase of computational power allowed the proposal of improved methods (some based on machine learning, described in Section 2.1.2 and 2.1.3) that could adapt to multiple notations and more complex contents under certain conditions.

The existing methods for architectural floor plan analysis were mostly designed for single-unit floor plans, which are fundamentally different from the multi-unit floor plan images we focus on. However, both image types share some common elements in context, notation and underlying ideas, and the design of an approach for multi-unit floor plans benefits from a review of methods geared towards single-unit floor plans.

In this section, we review the existing wall extraction and room detection methods, the type of challenges they faced and the solutions they proposed.

2.1 Wall Extraction Methods

Wall extraction methods are concerned with separating the walls from the rest of the content in an architectural floor plan. They have evolved together with changes in

their intended application, the challenges they aim to overcome, the availability of new computer vision methods, and the constant growth of computing power.

2.1.1 Early Approaches

The initial motivation for floor plan understanding systems was the need to automatically digitize massive archives of drawings that were only available as printed or hand-drawn documents (either because the CAD version didn't exist, or because it was expensive / not possible to acquire it). This task was traditionally performed manually by tracing the scanned image into the CAD software, on a computer screen or using specialized tracing tables [8].

The first attempts to solve this problem [9, 8, 10] only tackled images with a predictable drawing notation (usually representing walls as thick lines) and often would be limited to horizontal and vertical lines. Structure-wise, these approaches were mostly concerned with vectorizing the walls, which could be extracted using morphological operations to get rid of thin and small elements; walls were then thinned to 1 pixel thick lines (skeletonized) and traced to obtain their vector representation (Figure 2.1). The traditional process pipeline of *Line extraction* → *Skeletonization* → *Vectorization* became a well-known approach for analyzing various kinds of technical drawings (e.g. topographic maps [5]).

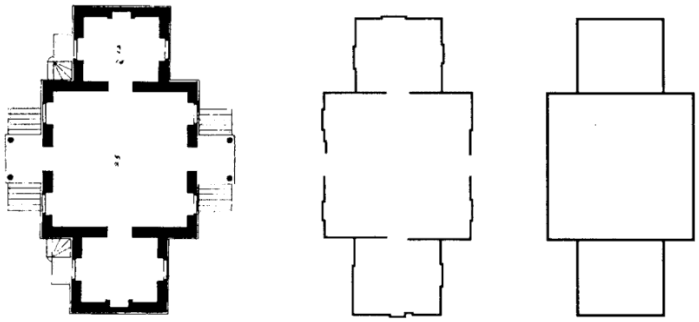
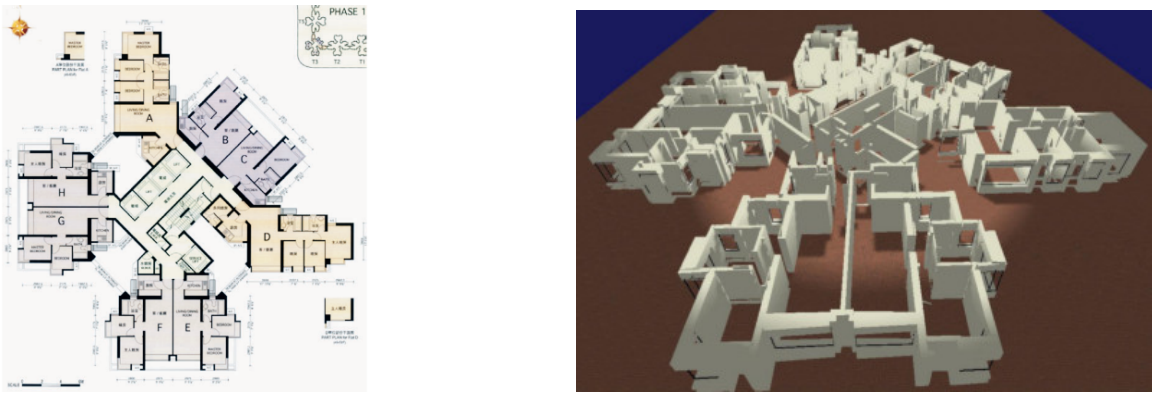


Figure 2.1: Example of an early wall extraction method. Left: Original image after preprocesssing. Center: Thick lines are skeletonized and vectorized. Right: Vector shapes are simplified. Figure reprinted from [9].

As opposed to synthetic architectural images created for academic experimentation, real-world floor plans were drawn and printed as images of bigger size than other image types fit for computer screens, and resizing them to a smaller size would affect

the document's readability. Running image-processing algorithms on these images would raise memory and performance concerns, and Dosch et al. [11] were among the first to address them by proposing a tiled approach where the image was split into partially overlapping tiles which were then processed individually and merged afterwards.

Dosch et al. [11] were also among the first to attach a 3D model generation step to the end of the image analysis pipeline, a trend continued by Or et al. [12] who improved the 3D model output format by generating polygon meshes. Or et al.'s approach [12] presented additional novelties: the support of diagonal walls and experiments on multi-unit floor plans (Figure 2.2) which to the best of our knowledge are the only documented tests against this image type; however, their method relied on user interaction to a degree that processing a single image took multiple hours.



(a) (b)

Figure 2.2: Example of a 3D model generation experiment from a multi-storey floor plan with diagonal walls. (a) Input image; (b) Rendered 3D model. Figure reprinted from [12]

Ahmed et al. [13] proposed one of the first automatic floor plan analysis workflows that included stages for information segmentation (separating text, walls and symbol images), structural analysis (vectorizing walls, closing discontinuities) and semantic analysis (symbol spotting, room detection and labeling). However, their wall extraction method relied on walls being drawn as the thickest lines in the image, limiting their approach to this specific drawing notation.

2.1.2 Traditional Machine Learning Methods

The methods mentioned is Section 2.1.1 were fit for a single notation: walls drawn as the thickest lines of the image. But in reality, there's no standard notation for architectural drawings, as explained by De Las Heras et al. in [14]: walls can be "a single line of different widths, two or many parallel lines or even hatched patterns" (Figure 2.3).

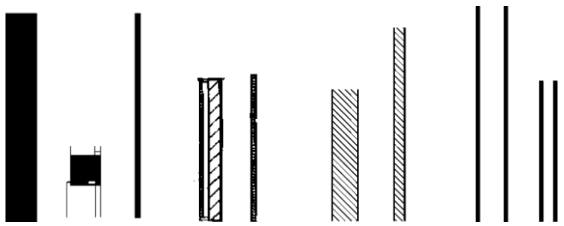


Figure 2.3: Real graphical examples of different wall notations. Figure reprinted from [15]

Towards a method able to handle different notations, De Las Heras et al. proposed the first machine-learning based method for wall extraction (as presented in Figure 2.4), modeled after the classical Bag-Of-Visual-Words [16] model:

1. Divide the image in overlapped square patches.
2. Label each patch as "Wall" or "Not Wall" based on corresponding ground-truth images.
3. Extract patch features using Principal Component Analysis (PCA) [17].
4. Cluster patches via K-Means [18] into a visual words vocabulary.
5. Assign each word's label based on the average label of the cluster patches.

A visual vocabulary trained this way was then used to classify patches in new images. This method was quantitatively evaluated on the CVC-FP Dataset [19] which contained 90 real architectural single-unit floor plans with corresponding ground-truth. Two years later, the same authors published a revision of this method [15], replacing the PCA descriptor with a Blurred Shape Model (BSM) [20] descriptor and K-Means with Support Vector Machines (SVM) [21] for classification. The updated method out-performed the original when evaluated on an augmented version (120 images) of the CVC-FP dataset. This method was later used in [22] as part of a full pipeline of wall extraction, door and window recognition, and room detection.

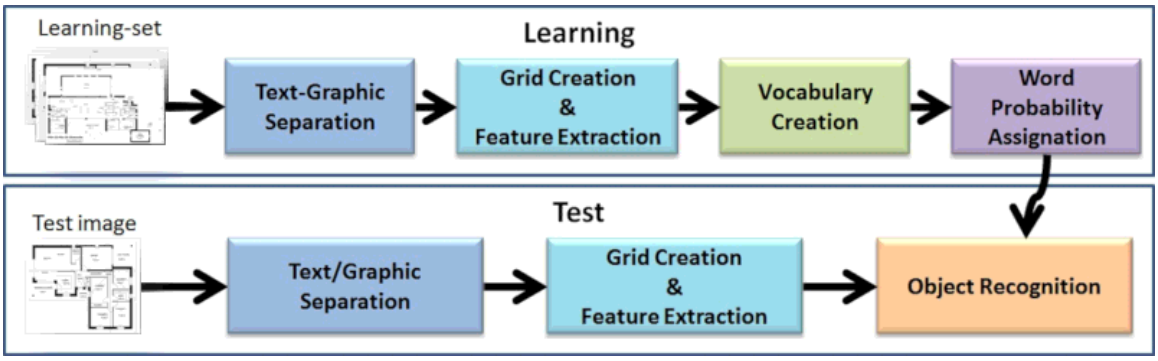


Figure 2.4: Patch-based wall extraction method pipeline. The "Learning" steps are used to train the system, and the "Test" steps extract walls on new images. Figure reprinted from [14].

The main shortcomings of the patch-based approaches mentioned so far (as explained by their own authors in [15]) are that they can only be trained to support one notation at a time and the training process relies on the availability of pixel-wise ground-truth for a sufficiently diverse image dataset. Creating a suitable training dataset is not trivial because the images are big and complex, and their interpretation requires technical knowledge. Although these methods obtained good results in their own controlled dataset, they were not a reasonable solution for the real-world problem.

2.1.3 Unsupervised Multi-Notation Methods

The same year (2013), De Las Heras et al. [15] proposed an unsupervised wall detection method based on 6 general assumptions for characterizing walls:

1. Walls are modeled by parallel lines.
2. Walls are rectangular, longer than thicker.
3. Walls appear in orthogonal directions.
4. Different thickness is used for external and internal walls.
5. Walls are filled by the same pattern.
6. Walls appear repetitively and are naturally distributed among the plan.

Based on these assumptions, this method proposed algorithms for detecting parallel

lines in different image orientations, using statistics to analyze wall candidates, ranking and combining them into an output wall segmentation. It uses statistical analysis to detect the "Thick wall line" notation, which made wall extraction solvable by the earlier methods described in Section 2.1.1. This approach lightly outperformed the recall of previous machine-learning based approaches on different notations in the CVC-FP dataset.

To perform parallel line detection, this method repeatedly rotated the image at a fixed angle interval α and scanned it in horizontal and vertical directions, looking for connected components that might meet the 6 general assumptions. As mentioned in [15], a low α value is required to detect diagonal walls in all orientations, but it also slows down the overall method at a rate of $360/\alpha$; For their experiments, they settled for $\alpha = 15°$ (24 rotations) which obstructed the detection of any diagonal line at unexpected angles, specially at odd multiples of 7.5°. However, it was precise enough for their own dataset, and constituted one of the best approaches in the literature.

2.1.4 CNN-Based Approaches

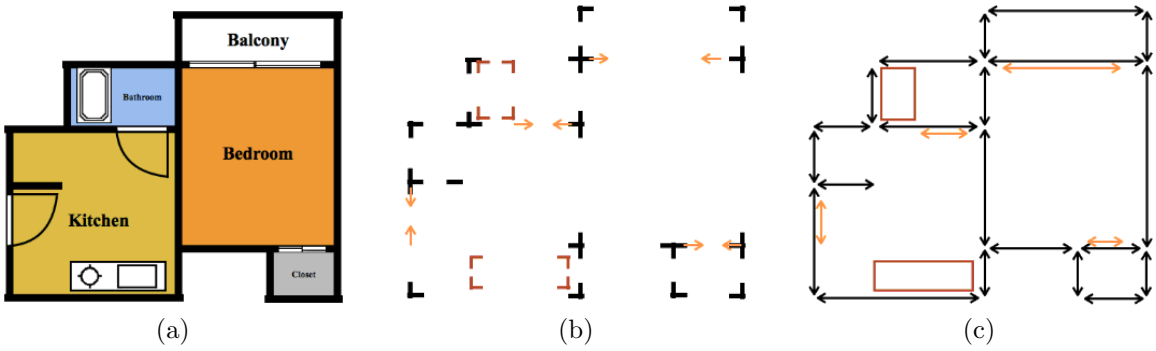


Figure 2.5: Relevant steps for wall extraction using a CNN to detect junctions. (a) Original input example. (b) Junctions detected using a CNN. (c) Connecting junctions into a set of primitives. Figures reprinted from [23].

The continuous growth of computing power enabled the use of Convolutional Neural Networks (CNNs) in modern computer vision, which started to permeate most application domains (architectural floor plan analysis included). In [23], Liu et al. used a straight-forward application of the CNN-based pixel-level heatmap prediction method from [24] to detect the "junctions" (corners and cross-points) in a floor plan

(Figure 2.5), and then used the junctions for reconstructing the original wall structure based on semantic rules. To train and their method, they used the recently published LIFULL HOME dataset [25] to prepare a labeled dataset of 1,000 floorplan raster images which were manually ground-truthed. The size of this dataset made it suitable for deep learning purposes.

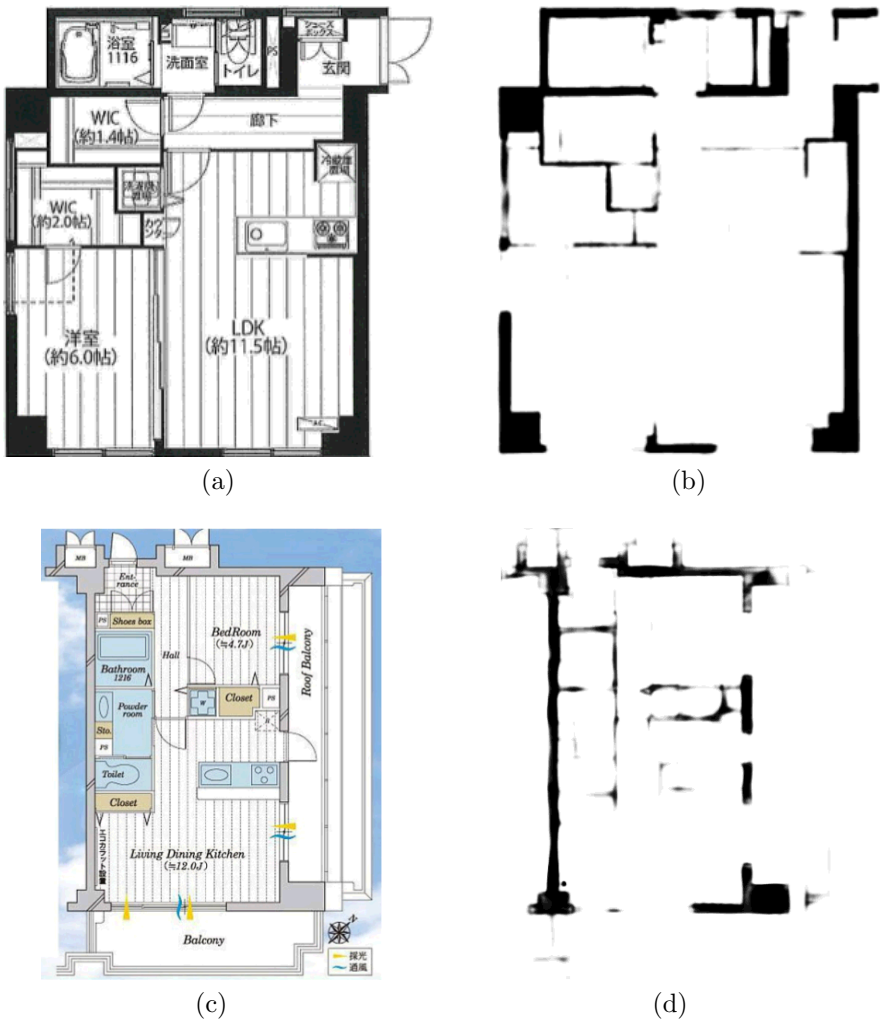


Figure 2.6: Output samples of a FCN-based wall-extraction approach. (a) Original input example A, described as a "simple example". (b) Output for A. (c) Original input example B, described as a "difficult example". (d) Output for B. Figures reprinted from [26].

Liu et al's CNN based approach was valuable for its novelty and current relevance; it also reached average values of precision and recall over 90%. However, their solution was mostly suitable to the images of the LIFULL HOME dataset, which only contain small architectural drawings of low complexity (single floor units, only ver-

tical/horizontal walls at 90-degree angles, predictable, easy to segment notation, no overlapped graphics), adequate to the small Japanese residential units from which the original dataset emerged. It's unclear how this approach would perform wall extraction in our images, since we have to deal with a notable degree of content overlapping which also produces corners and cross-points.

Another CNN based approach was recently proposed by Dodge et al. in [26] based on the method for semantic segmentation using Fully Convolutional Networks (FCNs) described by Long et al. in [27]. They used their own images to train their system: The R-FP-500 dataset [7] which contains 500 floor plan images from Rakuten Real Estate and their corresponding pixel-wise ground-truth. Their system was only trained once for all graphical styles in the dataset, demonstrating capacity to handle the variability present in this dataset. Figure 2.6 shows output examples from their experiments.

Dodge et al. [26] also compared their own methods quantitatively to previous approaches by De Las Heras et al. [14, 28] on their own dataset (CVC-FP) using the Jaccard Index (JI, also known as VOC Score [29]) as measure; However, their JI value (89.2) was not much higher than those obtained in De Las Heras et al.'s previous works on the same dataset (86.1 and 97.14 in [28]). Dodge et al. in [26] argued that their method only requires training once compared to [28], and thus only compared against a modified version of the patch-based approach that was only trained once, thus obtaining a lower JI score.

Although this FCN based method attained high quantitative results, from a qualitative perspective the output in Figure 2.6 showed how this method is unable to guarantee wall connectivity, sometimes missing full walls (making recovery through post-processing impossible).

2.2 Room Detection Methods

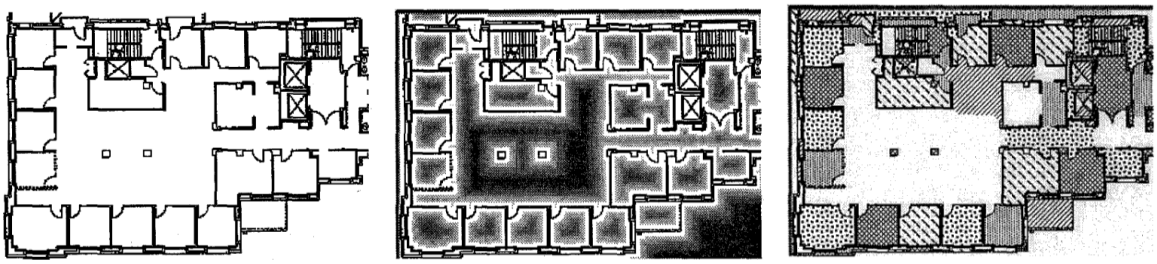
Room detection methods are concerned with detecting and segmenting the different room regions in the floor plan image. They are guided by the wall structure and thus require it to be already extracted. Rooms can be detected using geometrical methods on the vectorized wall structure, or by detecting closed loops in the wall structure

and applying pixel-based region filling methods. Their output can be either a distinct pixel region for each room or a vectorized polygonal region that matches each room's boundary.

Room detection relies on the wall structure it receives being correct and complete, as it can be greatly affected by wall extraction errors (e.g. missing walls, or misreading other elements as walls).

2.2.1 Pixel-Based Methods

Pixel-based methods detect rooms by performing image analysis on a raster image representation of the walls.



(a) (b) (c)

Figure 2.7: Example of an early pixel-based room segmentation method. (a) Input image with extracted walls; (b) A distance map is obtained measuring every pixel's distance to the walls; (c) Result with detected rooms in different fill patterns. Figures reprinted from [30].

Koutamanis and Mitossi [9] proposed one of the earliest room detection methods, by closing holes in a skeletonized version of the walls (last step in Figure 2.1) using fixed rules; however, their approach was unable to handle non-trivial cases or wall extraction errors.

Ryall et al. proposed in [30] the use of a "proximity field" (Figure 2.7, also known as the *distance transform* [31]) to detect the most distant points to any wall as candidate centers of the rooms. Each image pixel was assigned to the closest room center, determined by an algorithm inspired by the concept of "energy minimization". This semi-automatic method was robust to image noise and was able to detect rooms even when their boundary was not clearly surrounded by walls. However, the method

was driven purely by pixel properties, and tended to under-segment or over-segment the room regions.

2.2.2 Geometry-Based Methods

Geometry-based methods detect rooms from a vectorized representation of the walls. As vectorization is often a required step for floor plan understanding systems, geometric algorithms can be applied directly on the wall extraction output to obtain rooms, described as connected line loops.

(a) (b)

Figure 2.8: Example of a geometry-based room segmentation method by Mace et al. (a) Ground-truth with room regions in different colors; (b) Rooms obtained using the polygon partitioning technique proposed by the authors. Figures reprinted from [32].

Mace et al. proposed in [32] a top-down polygon partitioning method (Figure 2.8) based on the assumption that rooms should be nearly convex regions. This method closed gaps between walls by minimizing a measure of concavity in the resulting room polygon. This kind of mathematical approach (another example is the use of Voronoi diagrams [33]) often appeared as the natural solution to theorists, but in practice the lack of contextual knowledge caused their method to over-segment the regions or link the wrong pair of wall vertices.

In [34], Wessel et al. presented a method to detect rooms by performing transversal "cuts" at different heights in a 3D model of the building. These cuts could be seen as 2D floor plan images. They used a graph-centered approach and assumed that rooms could be detected by closing small gaps in the structure. In their method, the decision on how to close a gap depended on information from multiple transversal cuts.

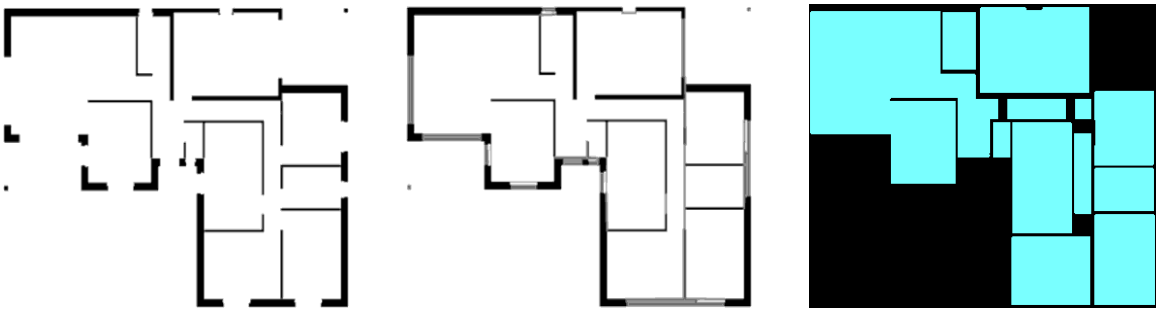
Various authors [8, 12, 34, 22, 23] detect loops in the polygon representation of walls, often closing the holes caused by doors and windows (sometimes detecting their symbols beforehand). These methods require the walls and other symbols to be detected with high accuracy beforehand, which is only possible on certain datasets of low complexity or with the human operator's aid.

2.2.3 Mixed Methods

More elaborate methods like [13] will generate an additional pixel image with the extracted walls and close the gaps in the wall structure, based on information obtained from previous structural analysis steps, including some combination of:

- Wall detection
- Wall edge extraction
- Wall merging
- Detection of the building's external boundary

After closing the gaps, rooms are detected as the pixel connected components that remain in the image as shown in Figure 2.9. Mixed methods combine pixel and semantic information to detect the rooms while keeping a certain degree of robustness.



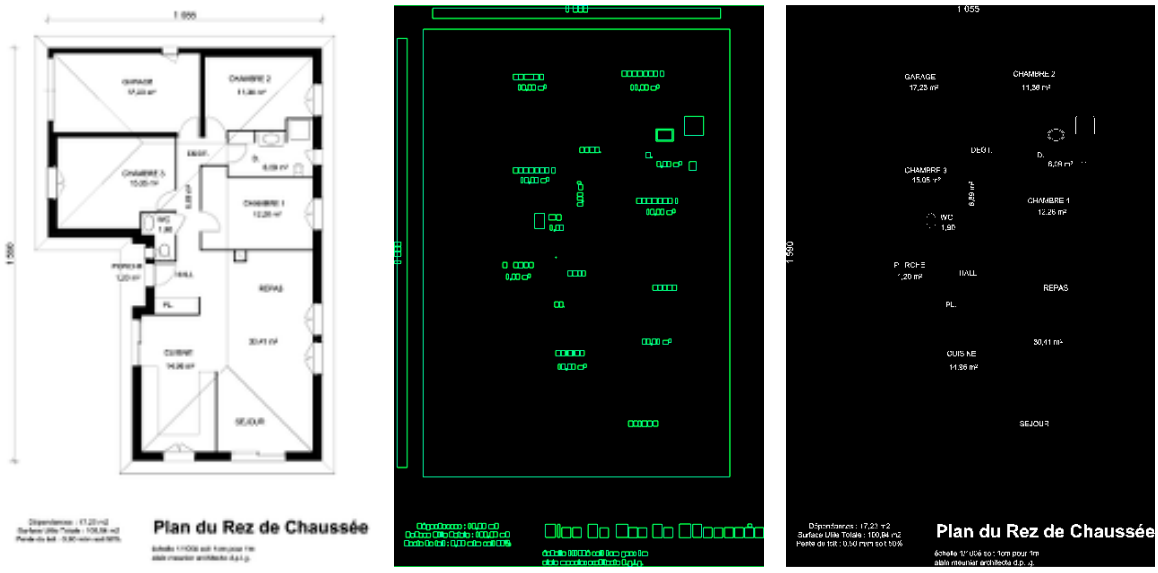
(a) (b) (c)

Figure 2.9: Example of a pixel-based room segmentation method by Ahmed et al. that relies on semantic information. (a) Input image with extracted walls; (b) Gaps are closed using geometrical rules on the vectorized wall polygons; (c) Connected components obtained after closing gaps where door symbols were detected. (Figures reconstructed based on [13])

2.3 Relevant Preprocessing Methods

A common preprocessing step for many wall-extraction approaches [8, 11, 12, 13, 14, 15, 28, 22] is text/graphics separation: detecting and extracting text from the image to reduce its complexity.

Fletcher and Kasturi [35] proposed one of the text/graphics separation methods used in the early approaches for wall extraction (Section 2.1). Their method leveraged the unique characteristics of technical drawings to detect text characters by filtering the image's connected components based on statistical histogram analysis of their area, bounding box's dimensions, aspect ratio and pixel density. This approach was easy to implement and obtained good results, but it failed to detect text characters overlaped to other symbols.



(a) (b) (c)

Figure 2.10: Text/Graphics separation example for illustration, following [36]. (a) Input image from dataset [19]; (b) Bounding boxes for connected components; (c) Text characters extracted based on statistical properties of their bounding boxes.

Ahmed et al. [36] later revisited this topic, improving Fletcher and Kasturi's method [35] to detect overlapped text characters by inferring the position of missing text characters from the position of surrounding detected characters. They also proposed additional statistical filters to separate text characters from dashed lines, which would often be mistaken for the 'I' character.

Both approaches worked under the assumption that text is present on the image, in sufficient quantity to make the area of the text characters the area measure of highest probability among the total connected components on the image.

Dashed line removal was also approached as a separate preprocessing step by Dosch et al. in [11], where they proposed filters for detecting dashed lines and arcs in the image, although their methods are strictly limited to a single notation.

3 Proposed Approach

This research work intends to propose a wall extraction and room segmentation method specifically for *multi-unit architectural floor plans*, as previously discussed in Section 1.1, targeting the particular challenges these types of technical drawings pose (presented in Section 4.3.1).

Multi-unit floor plans resemble single-unit floor plans in many ways (same method of production, created by similar professionals when not the same, very similar notation and visual appearance), which justifies reusing and improving concepts from the related work (Chapter 2). Still, the differences in purpose, complexity and technical challenges motivate the design of a new approach that takes these factors into account.

Following our analysis of related work, and considering the design challenges of the specific problem we want to solve, we propose the use of a combination of techniques that have been tested in images similar to ours *and* our own novel techniques, towards a method created to fit in a complete floor plan analysis system, with reasonable computational requirements (commercially available hardware for home users at the time of this writing) and tolerable run-time to a human operator: Our system should be fast enough to justify not using alternative solutions (e.g. using a semi-automated labeling tool to manually select walls and rooms).

Our method is composed of 2 sequential modules: **Wall Extraction** and **Room Detection**, described in detail in the following sections.

3.1 Wall Extraction

3.1.1 Design Assumptions

Taking into account our problem discussion so far, we adapted De Las Heras's general wall assumptions [15] to our images as follows:

1. Walls are modeled by nearby parallel lines that depict volume between them.
2. Walls are longer than thicker.
3. Wall lines connect to other wall lines through their endpoints.
4. Walls often connect to other walls in right angles.
5. Wall lines don't cross other wall lines.
6. Walls appear repetitively and naturally distributed among the image's main contents.

These become our own **wall assumptions** and drive our algorithmic design. Notice that assumption (1) doesn't hold in case of a common notation: Walls described as single, thick lines. On Chapter 2 we observed how for this particular notation, other approaches simply extract the thicker lines to obtain the walls. We explain our own approach to this notation in Section 3.1.5.

3.1.2 General Method Overview

Figure 3.1 shows the general steps of our wall extraction method. The module receives an image file as input, and outputs the wall structure in a vector format.

We summarize the outline of this method as follows:

1. **Preprocessing**: Binarize the image by removing all color and grey pixels; then remove text and other floor plan elements that don't resemble walls.
2. **Slice Transform**: Gather shape information for every line pixel, to be used in subsequent steps.

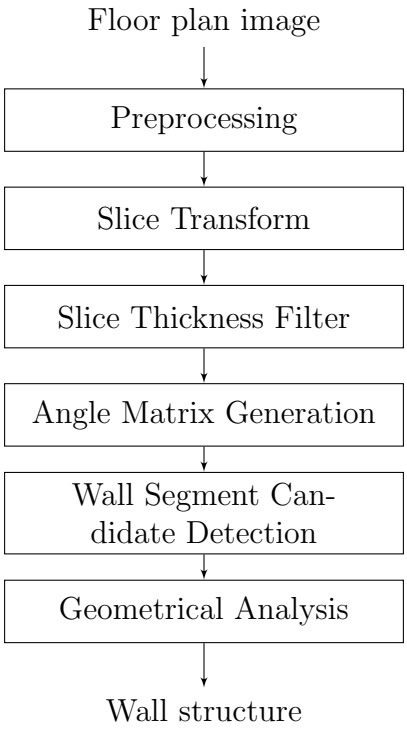


Figure 3.1: General steps of proposed wall extraction method

3. **Slice Thickness Filter**: Analyze line thickness and remove content with unexpected values.

4. **Angle Matrix Generation**: Obtain an approximation of the line angle for every line pixel.

5. **Wall Segment Candidate Detection**: Separate lines by their angle and generate vectorized wall line candidates.

6. **Geometrical Analysis**: A series of geometric algorithms that find line relationships and decide which candidates belong to walls.

Our pipeline is loosely inspired on ideas presented by De Las Heras et al. [15, 22] on unsupervised wall extraction and complete systems for floor plan analysis. However, we propose a novel technique to generate wall-segment candidates (the *Slice Transform*), better suited to the challenges of multi-unit floor plans. The way we make use of the slice transform fundamentally changes how we perform wall extraction.

We present a detailed explanation of this method's steps in the following sections.

3.1.3 Preprocessing

The purpose of this sub-module is to simplify the input image for subsequent sub-modules. It receives a floor plan image file for input and outputs an 8-bit binary {0, 1} image representation.

The steps of this sub-module are described as follows:

1. Load the image file as an RGB 3-channel matrix representation.
2. Remove colored and gray pixels, by detecting and removing any RGB pixel where the average intensity value is greater than a binarization threshold bt (initially configured by the user), satisfying the condition in Equation 3.1:

$$\frac{1}{3}(R + G + B) > bt \tag{3.1}$$

3. Binarize the remaining RGB image to a single channel image with only 2 values: 0 for empty pixels and 1 for non-empty (black) pixels.
4. Assign a value of 0 to the external contour of the image (first and last rows and columns). This helps some of our algorithms avoid scanning invalid positions around the image borders.
5. Apply the Text / Graphics separation method from Ahmed et al. [36] (Briefly described in Section 2.3) and remove the elements that resemble text characters. This step will also get rid of dashed lines and small floor plan symbols that don't resemble walls. Although overlapped text characters are not detected in our implementation, we separate them from the wall lines in later steps.

Figure 3.2 illustrates the steps of this sub-module on a small (roughly 10%) section of one of our floor plans. We reuse this sample image in future sections to give continuity to our method description.

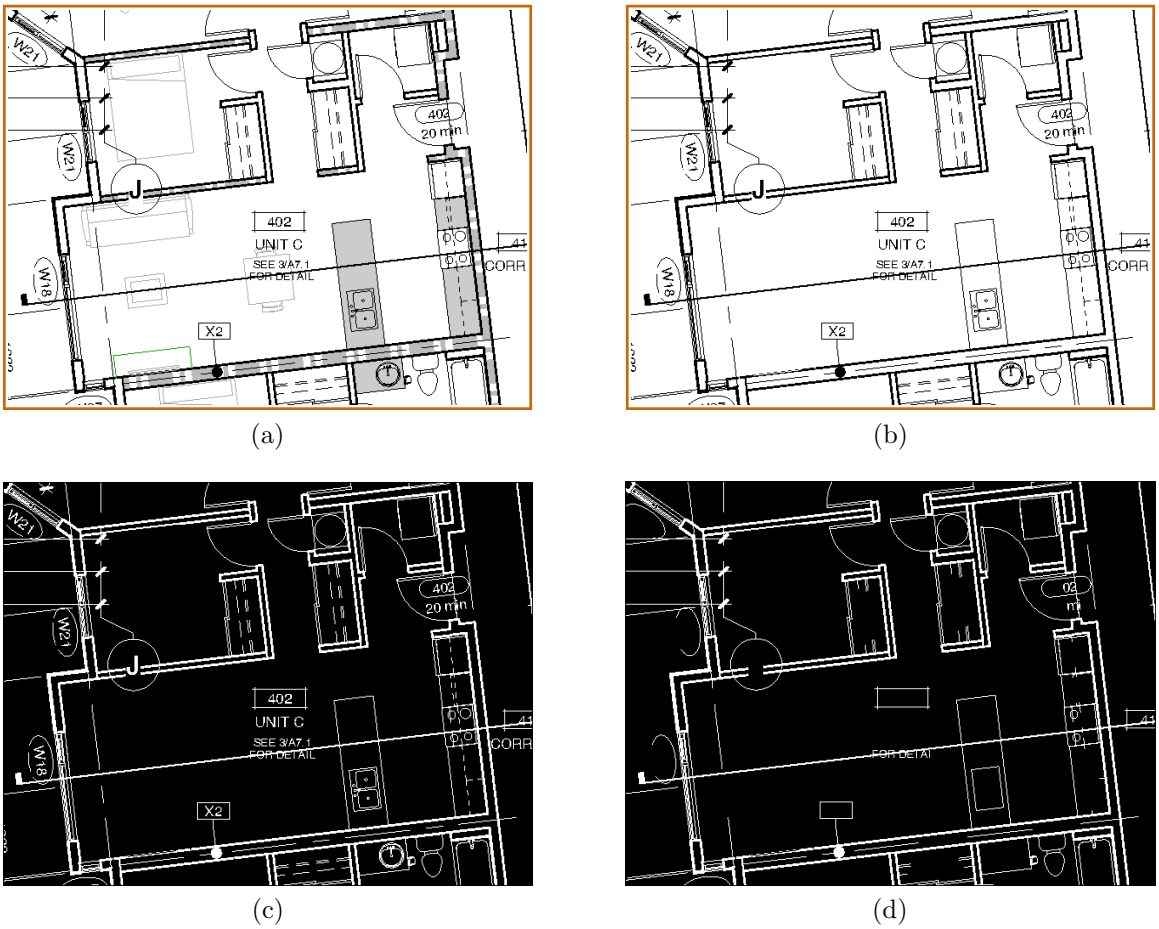


Figure 3.2: Preprocessing steps on a small floor plan section. (a) Input image with a colored border added for illustration purposes. (b) Color and gray pixels removed. (c) Binarized image (normalized for illustration). (d) After Text/Graphics separation.

3.1.4 Slice Transform

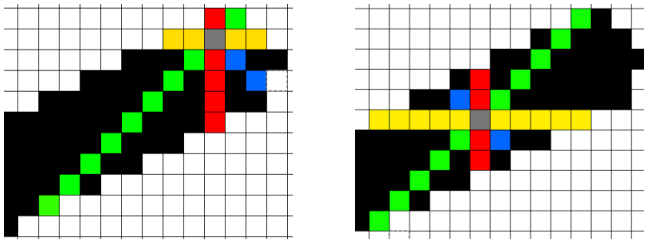
The purpose of this sub-module is to prepare pixel-wise information to be used by other sub-modules. Its input is the binary image acquired after preprocessing, and its output is an 8-bit 4-channel matrix we call the *Slice Matrix*.

The **Slice Transform** is an optimized operation to gather information on the shape surrounding each pixel. It provides a solution to the problem of recognizing overlapped lines in architectural floor plans and is fit for parallelization. We consider it the main original contribution of this thesis. This operation is described in the following sub-sections.

Slice Transform Definition

Following our previous definitions, consider the binary input image A an $m \times n$ matrix, and inside it a pixel a_{ij} in the position with row number $1 \leq i \leq m$ and column number $1 \leq j \leq n$, with a value of either 0 or 1.

For every non-empty pixel a_{ij} on the image, we seek to create a descriptor of the line shape surrounding it, made of 4 quantities obtained from counting the connected non-empty pixels that contain a_{ij} along a line profile in 4 directions: horizontal (0°), vertical (90°), forward diagonal (45°) and backward diagonal (135°), as illustrated in Figure 3.3. We call these 4 sets of pixels *"slices"*.



(a) (b)

Figure 3.3: Intuitive illustration of the slice transform of a pixel (gray) in two different positions. Line slices are colored as yellow (horizontal), red (vertical), green (forward diagonal) and blue (backward diagonal).

Let p_h, p_v, p_d and p_e be the length of the horizontal (h), vertical (v), forward diagonal (d) and backward diagonal (e) slice lengths respectively. We define the pixel descriptor p_{ij} as:

$$p_{ij} = T_s(a_{ij}) = \begin{bmatrix} p_h & p_v & p_d & p_e \end{bmatrix} \quad , \quad p_{ij} \in \mathbb{N}^4 \tag{3.2}$$

Where $T_s(a_{ij})$ is the *Pixel Slice Transform*.

Like i and j, let r and c be a row and a column in A, and a_{rc} the value at such position. We define the component p_h as:

$$p_h = min\{c \mid (c > j) \wedge (a_{rc} = 0)\} - max\{c \mid (a_{rc} = 0) \wedge (c < j)\} \quad , \quad r = i \tag{3.3}$$

Eq. 3.3 can be read as the difference between *"the minimum column to the right of a_{ij} with an empty pixel"* and *"the maximum column to the left of a_{ij} with an empty pixel"*. It describes the horizontal slice length as an interval of columns. Similarly, we describe the vertical slice length as an interval of rows:

$$p_v = min\{r \mid (a_{rc} = 0) \wedge (r > i)\} - max\{r \mid (a_{rc} = 0) \wedge (r < i)\} \quad , \quad c = j \tag{3.4}$$

The length of the backward diagonal slice p_e can be obtained as the euclidean distance between this slice's endpoints, as follows:

$$\begin{aligned} p_e = \|&min\{(r,c) \mid (a_{rc} = 0) \wedge (r > i) \wedge (r - i = c - j)\} \\ &-max\{(r,c) \mid (a_{rc} = 0) \wedge (r < i) \wedge (r - i = c - j)\}\| \end{aligned} \tag{3.5}$$

Eq. 3.3 can be read as the magnitude of the vector between two points: *"the minimum coordinates to the bottom-right of a_{ij} with an empty pixel"* and *"the maximum coordinates to the top-left of a_{ij} with an empty pixel"*, both along the backward diagonal line profile since $(r - i) = (c - j)$.

By rotating the matrix 90° degrees, we can obtain the length of the forward diagonal slice p_d in a similar way:

$$\begin{aligned} p_d = \|&min\{(r,c) \mid (\tilde{a_{rc}} = 0) \wedge (r > i) \wedge (r - i = c - j)\} \\ &-max\{(r,c) \mid (\tilde{a_{rc}} = 0) \wedge (r < i) \wedge (r - i = c - j)\}\| \end{aligned} \tag{3.6}$$

Where $\tilde{a_{rc}}$ is a value in $\tilde{A}$, a 90° rotated version of A obtained from the dot-product of A with the anti-diagonal identity matrix $\tilde{I}$:

$$\tilde{A} = A \cdot \tilde{I} \tag{3.7}$$

By applying $T_s(a_{ij})$ to every non-zero pixel in A, we obtain its *Slice Matrix* M_s:

$$M_s(A) = \begin{cases} T_s(a_{ij}) & \text{if } a_{ij} \neq 0 \\ [0,0,0,0] & \text{otherwise} \end{cases} \tag{3.8}$$

Slice Transform Implementation

The *Slice Transform* is a full image per-pixel operation that requires the exploration of multiple positions for every non-empty pixel. This is a performance concern because one of the well-known factors that affect the speed of image processing algorithms is the number of times we access image pixels.

We address this issue by obtaining $M_s(A)$ from scanning full line profiles on the image in the four slice directions. We scan a single line profile using Algorithm 1, where $1 \leq i \leq m$ and $1 \leq j \leq n$ represent the initial scan position and d_i, d_j are the displacement values that set the scan direction.

The displacement values (d_i, d_j) in Algorithm 1 match the 4 directions of the Slice Transform using the following values: $(1, 0)$ horizontal, $(0, 1)$ vertical, $(1, -1)$ forward diagonal and $(1, 1)$ backward diagonal. This algorithm only needs to access each pixel position 2 times at most (first time to read a_{ij}, second time to write b_{ij} if needed), independent of the scan direction.

Since the algorithm can be implemented using integer programming and only requires addition and subtraction operations (both optimized at CPU level), it allows strong-typed programming languages to minimize CPU and memory demands, and both input and output could be stored as 8-bit matrices to minimize memory consumption (limiting slice length to a maximum of 255, suitable for our purposes).

Pixel I/O operations have the highest cost in Algorithm 1. Supposing a single *Scan-Run()* operation runs through s_r pixels, in the best-case scenario all pixels in the line profile are empty (s_r reads, 0 writes); while in the worst-case scenario all pixels are occupied (s_r reads, s_r writes) for a total of $2 \times s_r$ pixel I/O operations, presenting a **linear complexity** relationship.

Algorithm 1: ScanRun

Data: A ($n \times m$ input matrix) $\quad i, j \in \mathbb{N} \quad d_i, d_j \in \{-1, 0, 1\}$
Result: B ($n \times m$ matrix)

```
begin
    B ← 0
    count ← 0
    inSlice ← false
    while (1 ≤ i ≤ m) ∧ (1 ≤ j ≤ n) do
        if a_ij ≠ 0 then
            count ← count + 1
            inSlice ← true
        else if inSlice then
            inSlice ← false
            i_s ← i
            j_s ← j
            steps ← count
            repeat
                b_ij ← count
                i_s ← i_s − d_i
                j_s ← j_s − d_j
                steps ← steps − 1
            until steps = 0
            count ← 0
        i ← i + d_i
        j ← j + d_j
    end
end
```

We obtain the components of $S_t(A)$ by repeatedly scanning line profiles in A in the desired direction:

- *ScanRun()* horizontal on every row of A obtains the p_h component of $M_s(A)$.
- *ScanRun()* vertical on every column of A obtains the p_v component of $M_s(A)$.
- *ScanRun()* in the forward diagonal direction starting from every pixel at the first row and the first column obtains the p_d component of $M_s(A)$.
- *ScanRun()* in the backward diagonal direction starting from every pixel at the first row and the first column obtains the p_e component of $M_s(A)$.

Note that we only perform operations on non-empty pixels, which in floor plan images (and technical line sketches in general) appear in lesser quantity than empty pixels, specially after preprocessing. This is another reason why w is a more realistic measure for complexity analysis than the total image area.

Parallelization

We can obtain the 4 components of A in parallel, running the algorithm for intervals of i and j in the desired scan direction as shown in Figure 3.4, where we scan each direction in a separate thread.

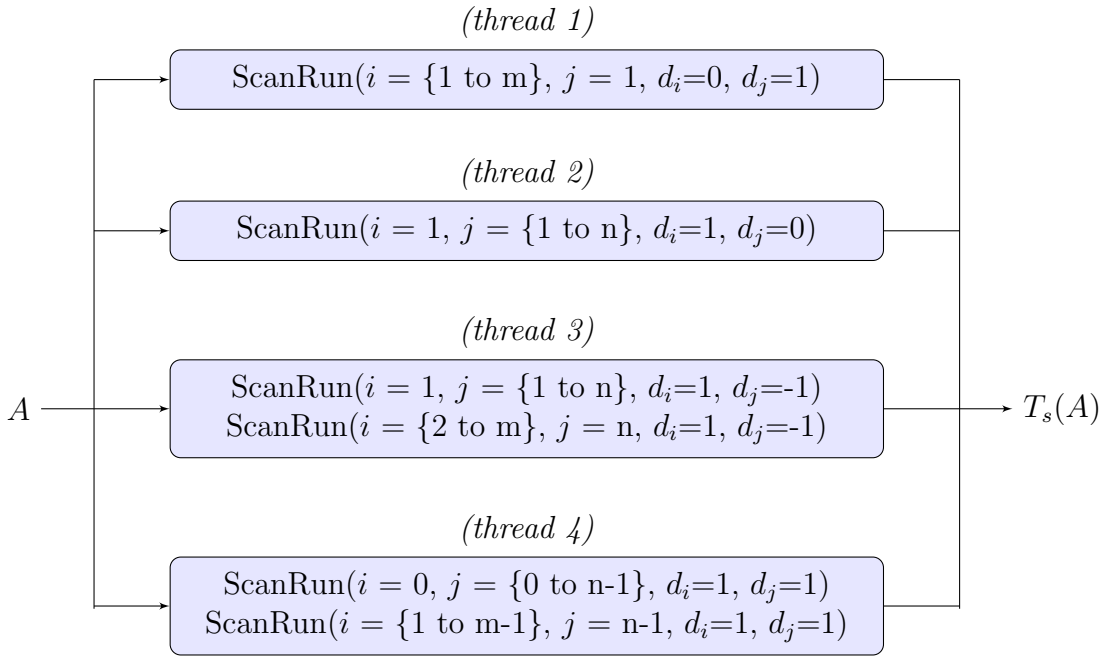


Figure 3.4: Generation of $T_s(A)$ with parallelized runs of Algorithm 1 in four threads, corresponding to the (1) horizontal, (2) vertical, (3) forward diagonal and (4) backward diagonal scan directions.

The method can be safely run in parallel because it only performs read operations in A and the output of each thread can be stored in a separate 2D matrix (no race conditions); the 4 outputs can then be joined into a 4-channel matrix representing $T_s(A)$. Algorithm 1 only requires algebraic operations and since the 4 threads perform the same amount of image read & write operations, a single thread won't become a noticeable bottleneck.

Future optimizations of this method should consider effects on race conditions and memory requirements. We further discuss the performance characteristics of the Slice Transform in Sections 4.2 and 4.5.

3.1.5 Slice Thickness Filter

This sub-module finds the approximate transversal thickness of the lines present on the image, and uses it to remove content of unexpected thickness. It receives the Slice Matrix and the preprocessed image A as input, and returns a 2D matrix M_t with the approximate line thickness for every pixel and the modified A.

Obtaining the line thickness

One of the Slice Matrix applications is obtaining a fast approximation of the line thickness for every pixel on the image. We can do this by combining the slice information of every pixel and its neighbors in a voting system. First, we gather the first line thickness approximation using Algorithm 2.

Algorithm 2: PixelLineThickness

Data: A ($n \times m$ input matrix) M_s (slice matrix)
Result: M_t ($n \times m$ thickness matrix)

```
begin
    M_t ← 0
    foreach m_ij ∈ M_s | m_ij ≠ [0,0,0,0] do
        {p_h, p_v, p_d, p_e} ← m_ij
        slices ← sort({p_h, p_v, p_d, p_e})
        if slices[0] = slices[1] ∨ (slices[0] > 1 ∧ 2 × slices[0] < slices[1]) then
            M_t(i, j) ← slices[0]
        else
            M_t(i, j) ← average(slices[0], slices[1])
    end
end
```

Notice that Algorithm 2 will return a lower value than expected at pixels positioned at corners and edges of diagonal lines, where one of $\{p_h, p_v, p_d, p_e\}$ will likely get cut by the jaggedness of the pixel representation. We solve this problem by replacing

every pixel where $M_t(i, j)$ is different from the majority of its 8 immediate neighbors, by the neighbors' average thickness value (a conditioned version of the Median Filter). Figure 3.5 presents a visualization the results obtained for our example map section.

Figure 3.5: Visualization for line thickness estimation using a color map {2:yellow, 3:blue, 4:orange, 5:purple, 6:cyan, 7:pink, 8:line, 9:skin, 10:dark cyan}. Pixels of thickness 1 have been removed.

We base our thickness approximation on the 2 shortest slice thickness lengths, which in practice are insufficient to determine the exact line thickness value (we later obtain a reliable value for line thickness in Section 3.1.7). For this reason, $M_t(i, j)$ only holds a rough approximation of the line thickness, accurate for line angles near multiples of 45° (angles that ressemble the 4 possible orientations for our slices). This approximation is enough to fulfill the original purpose of this sub-module: the detection and removal of elements with abnormal thickness values.

Line Thickness Filter

After obtaining $M_t(i, j)$, we use statistical analysis to remove contents with unexpected line thickness, under the assumption that most of the lines on the binary image have a line thickness that occurs repeatedly. We perform the following steps:

1. Gather a histogram of bin size 1 for the thickness values found in $M_t(i, j)$, as shown in Figure 3.6.

2. Find the most common slice thickness t_c.

3. To remove outliers, remove every pixel where $M_t(i, j) > f_t \times t_c$, where f_t is a *Thickness Factor* configured by the user (a value of 2 works in our dataset).

A difficulty present in our multi-unit floor plan images is the presence of overlapped symbols on the wall lines; these symbols are sometimes drawn as dense connected components. After removing pixels of unexpected thickness, we manage to get rid of these overlapped symbols as shown in Figure 3.7.

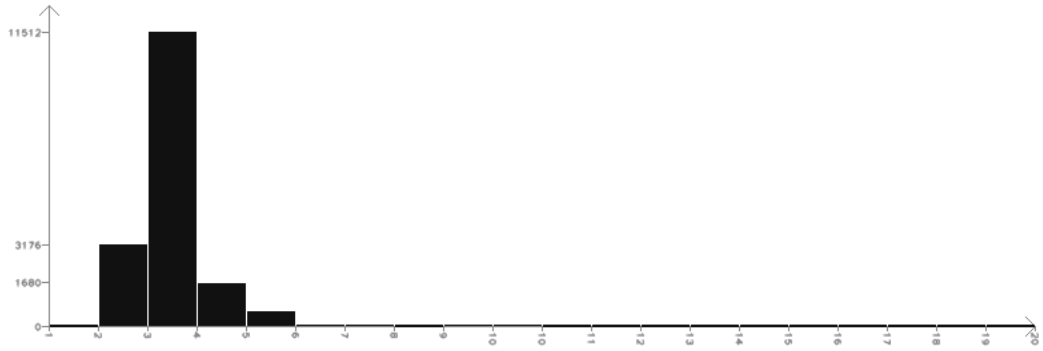
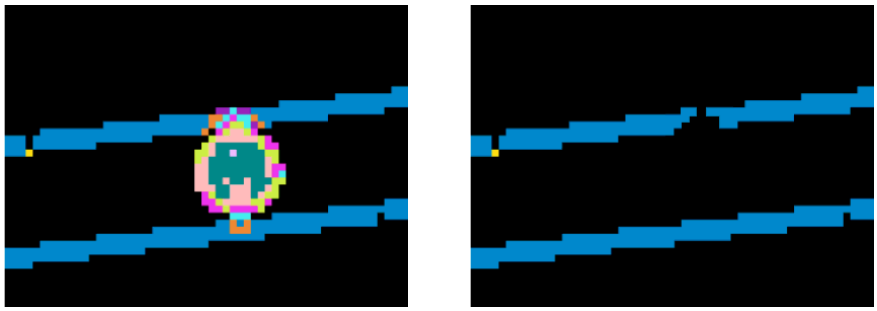


Figure 3.6: Line thickness histogram for the example image in Figure 3.5. Horizontal axis: Line thickness value; Vertical axis: pixel count.



(a) (b)

Figure 3.7: Detail of the effects of slice thickness filtering. (a) Zoomed-in section of Figure 3.5 centered around an overlapped symbol. (b) After the filter, the element is removed. The new line discontinuity is reconstructed in a later step.

After applying the slice thickness filter, this operation will usually miss some of the pixels, creating small blobs of isolated pixels which can be removed using morphological operations, or by removing connected components that don't look like walls

as we did in Section 3.1.3. We perform the second clean-up step mentioned before returning the filtered binary image.

We also remove 1 pixel thick lines as they are too thin to represent wall lines both in our images and other datasets [19, 7, 25]. Our method is thus limited to a minimum wall line thickness of 2 pixels, but the vectorization of 1 pixel thick lines is a solved problem (e.g. using Greenlee's method [5]) and could be added to the submodule in Section 3.1.7 if needed. There's no maximum limit to the thickness detected by our method, but our implementation introduces a limit at 255 pixels to minimize memory requirements (8 bits per pixel), which suits all the datasets reviewed.

If needed, a stricter threshold interval could be obtained using outlier detection techniques like the InterQuartile Range method (IQR) [37].

We expect architectural floor plans to be predominantly drawn using lines, so that they will present a histogram shape similar to Figure 3.6, except when walls are represented as very thick single lines. In the later case, the same histogram will show two separable peaks (as the thick wall notation intentionally makes walls easy to differentiate) and the walls can then be extracted by application of threshold selection techniques on the thickness histogram (like the proposed by Otsu [38]), or most of the approaches described in Section 2.1 which generally present the best quantitative results on this "thick wall" notation.

3.1.6 Angle Matrix Generation

The purpose of this sub-module is to obtain for every non-empty pixel in A, an approximation of the angle θ of the line that contains said pixel. It receives the Slice Matrix S_m and the preprocessed image A as input, and returns a 2D matrix M_a with the corresponding angle θ_{ij} for every pixel a_{ij} in a, such that:

$$M_a(a_{ij}) = \begin{cases} 0° \leq \theta_{ij} < 180° & a_{ij} \neq 0 \\ \varphi & \text{otherwise} \end{cases} \tag{3.9}$$

Where φ is the default angle value in M_a for empty pixels in A (we use -1 in our specific implementation). We approximate the line angle by analyzing trigonometric

relationships between the components of the Slice Matrix, based on the observation presented at Figure 3.8 related to the Oriented Bounding Box (OBB) that contains a pair of opposite slices.

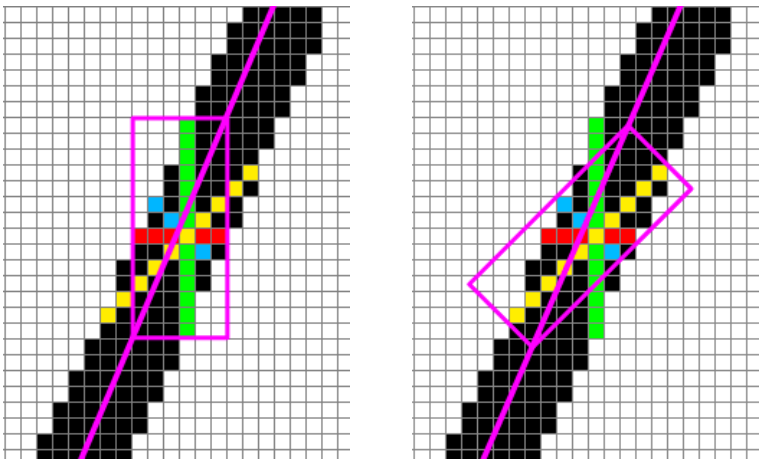


Figure 3.8: Example of OBBs for the slices of the slice transform at pixel a_{ij}. (a) For the OBB that contains the vertical and horizontal slice (green and red respectively). (b) For the obb that contains both diagonal slices (yellow and blue). OBBs appear in purple, as well as one of their diagonal projections.

The line angle θ_{ij} for pixel p_{ij} can be approximated as the diagonal angle of one of the OBBs created by slices of opposing directions ($p_h \wedge p_v$ or $p_d \wedge p_e$). Since each OBB has two possible diagonals, the best diagonal can be chosen using observations on the OBB's aspect ratio and the slope of the predicted angle as shown in Algorithm 3. Often both OBBs will describe a similar angle, but in difficult pixel positions (e.g. corners, edges, connections) we prioritize the OBB with the better chance of success.

The *AngleMatrix()* algorithm initially exploits some common conditions for perfectly horizontal, vertical and diagonal lines, which reduces the computation time and increases precision. Then if required, it approximates the angle from the OBB that has the greatest length difference between their opposing slices. We take this difference as a measure of reliability because when opposing slices match each other's lengths, it becomes harder to decide if the OBB is more horizontal than vertical, or if the line slope is positive or negative. If both differences are similar, we average the angle obtained from both OBBs.

We can visualize the angle matrix M_a using a colormap to represent angle values, by

Algorithm 3: AngleMatrix

Data: A ($n \times m$ input matrix) M_s (slice matrix)
Result: M_a ($n \times m$ angle matrix)

begin
  $M_a \leftarrow -1$
  foreach $m_{ij} \in M_s | m_{ij} \neq [0,0,0,0]$ **do**
    $\{p_h, p_v, p_d, p_e\} \leftarrow m_{ij}$
    if $p_d = p_e \wedge p_d = p_h \vee p_d = p_e \wedge p_v > p_h$ **then** $M_a(i,j) \leftarrow 90°$
    else if $p_d = p_e \wedge p_d = p_v \vee p_d = p_e \wedge p_h > p_v$ **then** $M_a(i,j) \leftarrow 0°$
    else if $p_h = p_v \wedge p_e < 0.1 \times p_d$ **then** $M_a(i,j) \leftarrow 45°$
    else if $p_h = p_v \wedge p_d < 0.1 \times p_e$ **then** $M_a(i,j) \leftarrow 135°$
    else
      if $p_d > p_e$ **then** $\alpha_0 \leftarrow \tan^{-1} p_v/p_h$
      else $\alpha_0 \leftarrow -\tan^{-1} p_v/p_h$
      if $p_h > p_v$ **then**
        if $p_d > p_e$ **then** $\alpha_1 \leftarrow \tan^{-1} p_d/p_e - \pi/4$
        else $\alpha_1 \leftarrow \tan^{-1} p_d/p_e + 3\pi/4$
      else
        $\alpha_1 \leftarrow \tan^{-1} p_e/p_d + \pi/4$
      end
      if $\alpha_0 < 0$ **then** $\alpha_0 \leftarrow \alpha_0 + \pi$
      if $\alpha_1 < 0$ **then** $\alpha_1 \leftarrow \alpha_1 + \pi$
      if $0.5 \times |p_h - p_v| > |p_d - p_e|$ **then** $M_a(i,j) \leftarrow degrees(\alpha_0)$
      else if $0.5 \times |p_d - p_e| > |p_h - p_v|$ **then** $M_a(i,j) \leftarrow degrees(\alpha_1)$
      else $M_a(i,j) \leftarrow degrees(average(\alpha_1, \alpha_0))$
  end
end

associating the angle interval $[0°, 180°)$ to a the Hue channel in the HSV color space, obtaining the angle colormap in Figure 3.9.

One may note how the colormap has similar colors for angles near $0°$ and $180°$; this accurately represents how we measure the line angle (scalar) instead of the direction (vector), thus all angles in the interval $[180°, 360°)$ are equivalent to their opposites in $[0°, 180°)$. We maintain every angle in our algorithms inside this interval by keeping Equation 3.10 valid through every angle calculation.

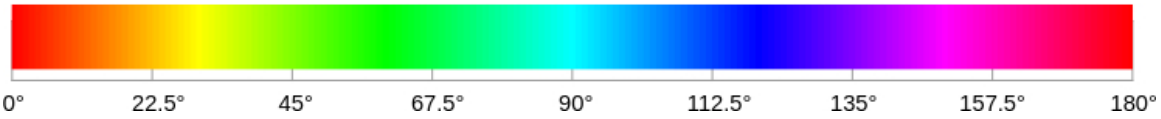


Figure 3.9: Hue colormap for angle matrix visualization.

$$\theta_{ij} = \begin{cases} \theta_{ij} - 180° & \theta_{ij} \geq 180° \\ \theta_{ij} + 180° & \theta_{ij} < 0° \\ \theta_{ij} & \text{otherwise} \end{cases} \tag{3.10}$$

For illustration purposes, we use the hue colormap to generate the angle matrix visualization of a diagonal line at 25° degrees in Figure 3.10, which would ideally have a solid yellow mustard color to match the corresponding color in Figure 3.9. Instead, we observe different shades of orange in the line body and some unexpected green and red colors near the edges, which we recognize as angle approximation errors.

Algorithm 3 is usually not accurate at line corners. In these cases, the pixel descriptor alone doesn't hold enough information to perform a reliable angle approximation. We observe that most pixels that were correctly approximated have a similar color to their neighbors, while pixels that caused errors are different to their neighbors. Following this rationale, we improve the angle approximation accuracy by combining the information of nearby pixels, applying a non-linear blur filter on M_a. This filter is called *Conditional Blur* and is described in Algorithm 4.

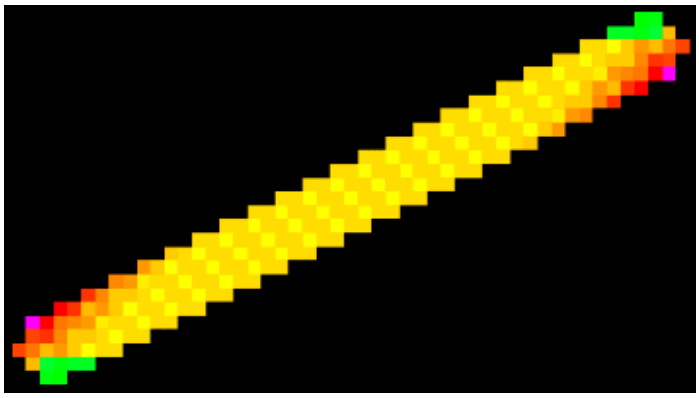


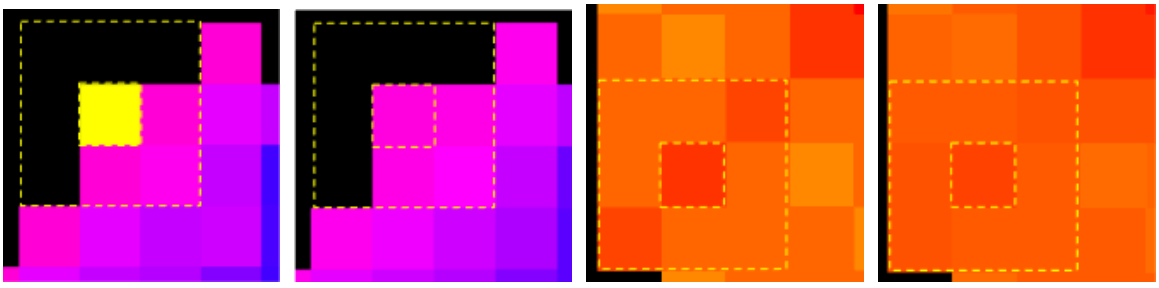
Figure 3.10: Detail of the angle matrix visualization for a diagonal line of thickness 4 px, length 50 px and angle 25°.

Algorithm 4: ConditionalBlur

Data: A ($n \times m$ input matrix) M_a (angle matrix)

Result: M_a ($n \times m$ angle matrix)

```
begin
    foreach θ_ij ∈ M_a | θ_ij ≠ φ do
        neighbors ← 8-connected neighbors of θ_ij
        similars ← {∅}
        differents ← {∅}
        foreach n_k ∈ neighbors | n_k ≠ φ do
            if isSimilar(n_k, θ_ij) then similars.add(n_k)
            else differents.add(n_k)
        end
        if similars.size > differents.size then
            θ_ij ← averageAngle(similars, θ_ij)
        else if similars.size < differents.size then
            θ_ij ← averageAngle(differents)
    end
end
```



(a) (b) (c) (d)

Figure 3.11: Example of two conditional blur cases. (a) A pixel whose neighbors are different; (b) Pixel in (a) is replaced by the neighbors' average. (c) A pixel whose neighbors are similar; (d) The pixel in (c) is averaged with its similar neighbors. The pixel's neighborhood is indicated in yellow dashed lines.

The Conditional Blur filter in Algorithm 4 compares pixel angle θ_{ij} with its non-empty neighbors to determine which are *"similar"* and which are *"different"* from it. If most neighbors are "different" then θ_{ij} is replaced by the average of these "different" neighbors (Figure 3.11[a,b]). On the contrary, If most neighbors are "similar" then θ_{ij}

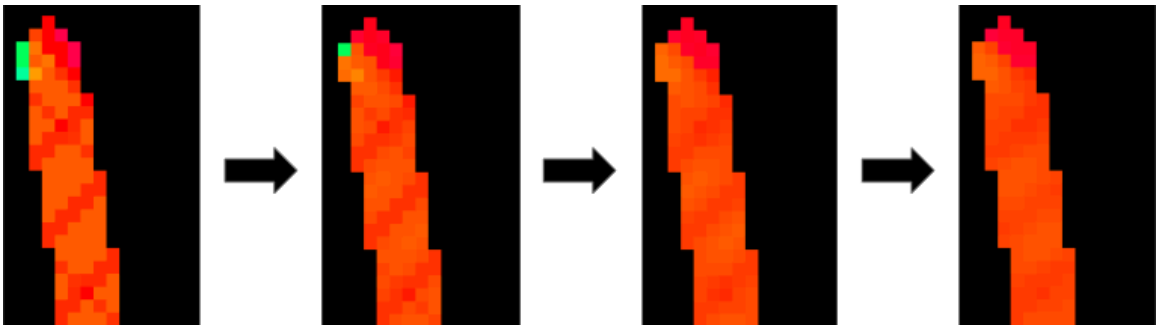


Figure 3.12: Effect of consecutive conditional blurs. Starting with the original section (left image) followed by the results after 1, 2 and 3 blur applications.

is averaged together with these "similar" neighbors (Figure 3.11[c,d]). This requires us to define a measure for angle similarity.

In Algorithm 4 we compared two angles using the *isSimilar*(θ_1, θ_2) function, which must be implemented while making sure that Equation 3.10 holds. We decide if two angles are similar by measuring their difference using Eq. 3.11 and comparing it to a configurable angle similarity threshold θ_{sim} in degrees.

$$dif(\theta_1, \theta_2) = min(|\theta_1 - \theta_2|, |\theta_1 - (\theta_2 + 180°)|) \tag{3.11}$$

We require similar considerations when calculating the average between angles, as shown in Equation 3.12.

$$avg(\theta_1, \theta_2) = \begin{cases} \frac{1}{2}(\theta_1 + \theta_2) & \text{if} \quad dif(\theta_1, \theta_2) \leq dif(\theta_1, \theta_2 + 180°) \\ \frac{1}{2}(\theta_1 + \theta_2 + 180°) & \text{otherwise} \end{cases} \tag{3.12}$$

Successive applications of *Conditional Blur* reduce the errors caused by unexpected slice values at corners and edges, and improves the overall line angle approximation by spreading local information. Figure 3.12 shows the effect of conditional blur runs.

Although increasing the blur iterations reduces the overall error quantity, the error reduction rate diminishes with every new blur iteration as shown in Section 4.2. Being a full image operation, the number of conditional blur levels presents a trade-off for performance. Blurring also poses a risk of information loss when most of the pixel

neighborhood has wrong values. However, in our experiments (Section 4.2) the overall effect of the conditional blur is a quantitative reduction in errors.

When we obtain the Angle Matrix (and apply conditional blur) to our example floor plan section, individual lines mostly show a homogeneous line angle as shown in Figure 3.13, making them easier to differentiate even when they intersect or connect to other elements. Although the line angle varies around overlapped or connected line sections, we assume that these problematic areas are only a small portion of the total line length and reconstruct them in later steps.

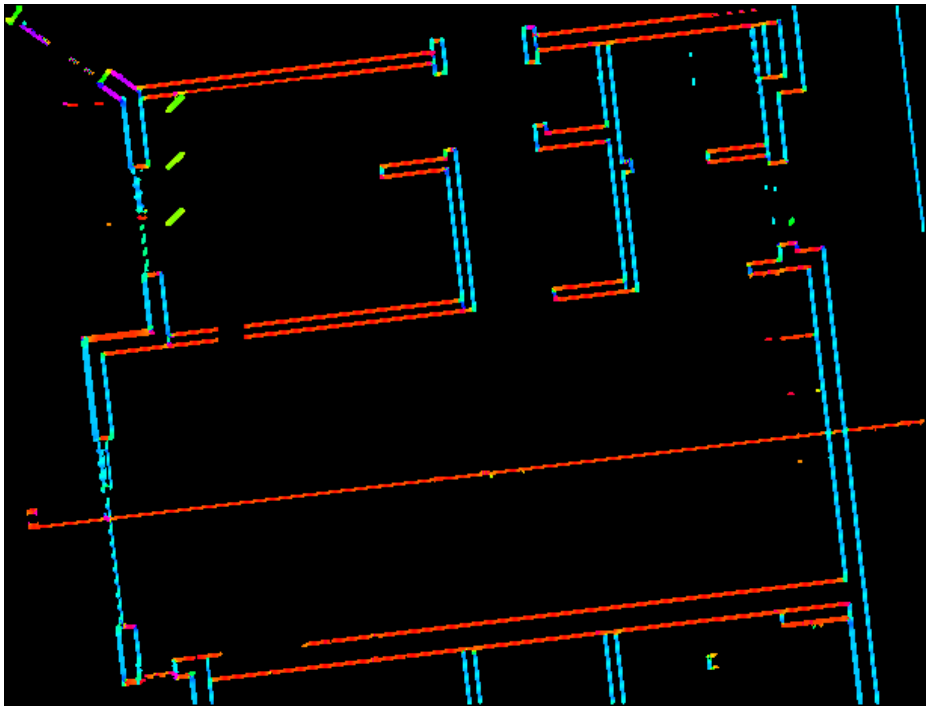


Figure 3.13: Effect of conditional blur in our floor plan segment. The blurred Angle Matrix obtained appears colored with the hue colormap from Figure 3.9.

3.1.7 Wall Segment Candidate Detection

The purpose of this sub-module is to detect line segments that might belong to walls and transform them into a mathematical representation (vectorization). It receives the preprocessed image A and the angle matrix M_a as input, and outputs a list of vectorized line segments L_s.

Line separation

One of the main challenges this thesis addresses is performing wall extraction in the presence of overlapped map elements, specially when they share visual characteristics with walls (e.g. wall lines intersected by other solid lines of the same thickness). We approach this problem (considering our wall assumptions mentioned in Section 3.1.2) with a "divide-and-conquer" strategy: We separate the image pixels by their line angle into 4 binary images $\{H, V, D, E\}$, corresponding to the angle intervals in Figure 3.14.

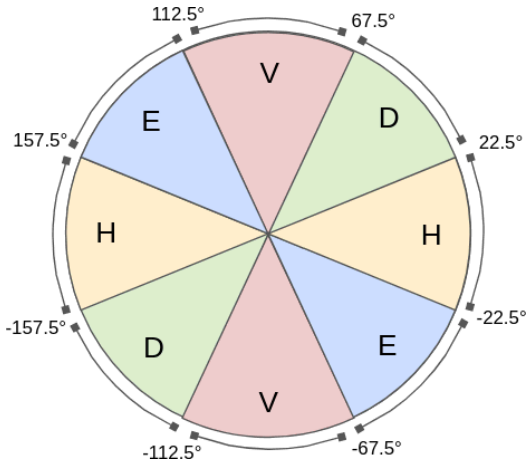
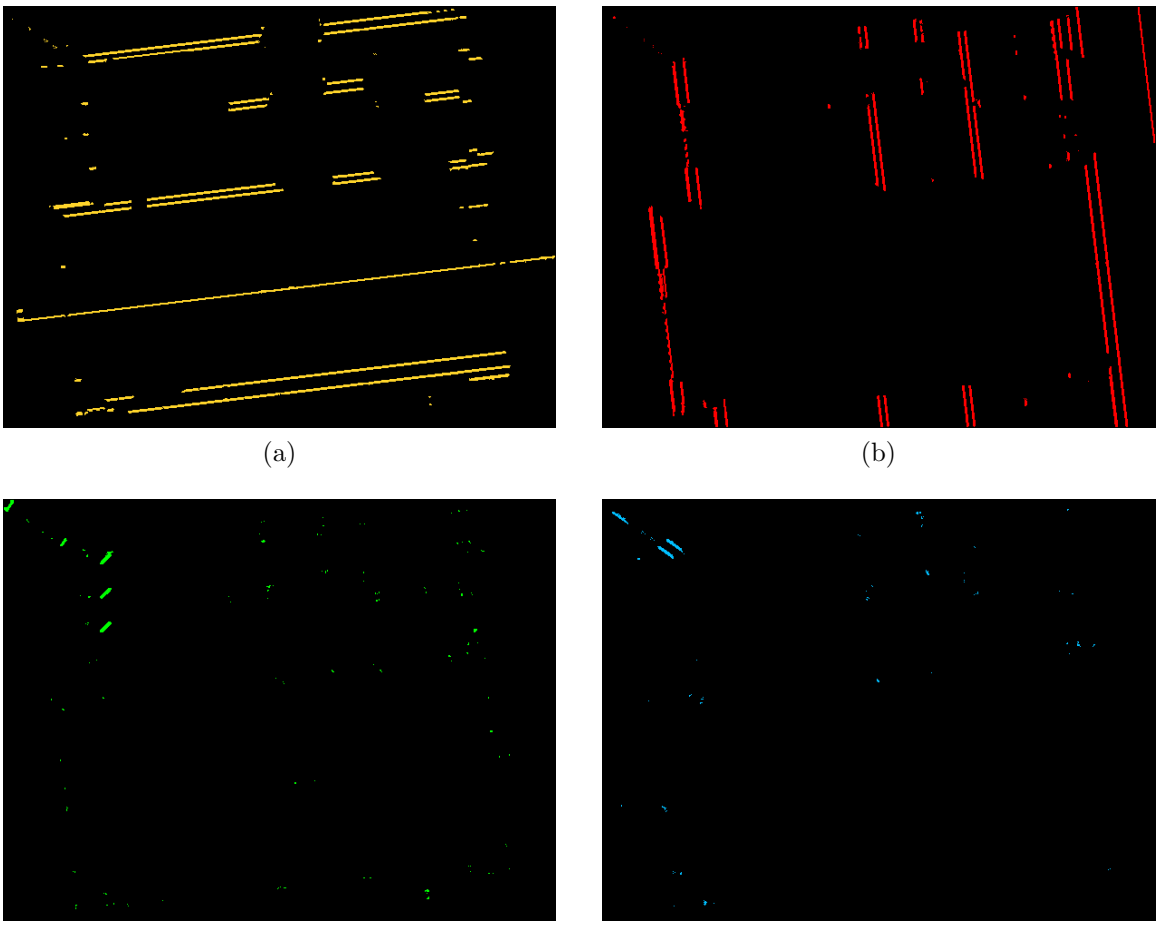


Figure 3.14: Angle intervals for line separation into 4 different images: H, V, D and E.

The angle intervals were picked to match common floor plan images; horizontal and vertical $\{H, V\}$ walls are very common, and when diagonals $\{D, E\}$ occur they also connect to other walls in right angles (one of our wall assumptions). Pixels with angles near the border between 2 intervals (odd multiples of 22.5°) are repeated in both separated images; we decide this by comparing the pixel angle to a configurable threshold we call *Angle Border Distance* d_{border}.

Separating the image pixels in our example floor plan segment produces the 4 binary images in Figure 3.15. This process successfully separates lines even if they are connected or overlapped, as long as they belong to different angle intervals, simplifying the line vectorization process. Additionally, most elements that don't resemble a straight line (e.g. small curves) will split into smaller connected components, easier to detect and filter.

To compensate for the content destroyed during this process, we project every non-



(a) (b)

(c) (d)

Figure 3.15: Example of pixel separation into binary images $\{H, V, D, E\}$ shown in (a), (b), (c) and (d) respectively. Pixel colors were added to match the intervals in Figure 3.14.

empty pixel in $\{H, V, D, E\}$ in the horizontal, vertical, forward diagonal and backward diagonal directions respectively, as long as the projected pixel is not empty in the original image A. We use Algorithm 5 for this purpose, specifying d_i and d_j to determine displacement direction the same way we did in Algorithm 1. We limit the use of this algorithm to pixels with at least 2 non-empty connected neighbors to avoid projecting noise.

Algorithm 5: PixelProjection

Data: M ($n \times m$ matrix) A (binary image) $i, j \in \mathbb{N}$ $d_i, d_j \in \{-1, 0, 1\}$
Result: M (with pixels projected)

begin
  do
    $m_{ij} \leftarrow 1$
    $i \leftarrow i + d_i$
    $j \leftarrow j + d_j$
  while $i \geq 1 \wedge i \leq m \wedge j \geq 1 \wedge j \leq n \wedge a_{ij} = 1 \wedge m_{ij} = 0$
end

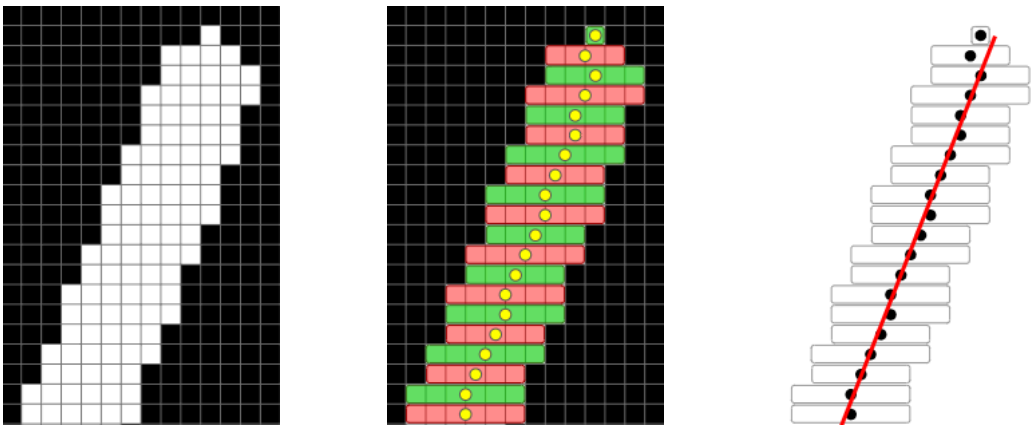
Line extraction using Slice Groups

After obtaining the $\{H, V, D, E\}$ binary images and projecting their pixels, we extract line segments from each one of them separately. There are still significant challenges to consider:

- The content has suffered some degree of destruction in previous steps.
- Undesired (e.g. noise) content that looks like a line must be avoided.
- Connected lines of similar angle must be considered as distinct objects.

Each one of the connected components in $\{H, V, D, E\}$ could be either a line, undesired noise, or combined groups of connected lines and/or noise. In our approach, we make the assumption that under this combination of content (some of it destroyed) lies a clean structure of straight lines, that can be subdivided into connected **"Slices"** that behave in a regular and predictable way, as illustrated in Figure 3.16. We call the set of slices that compose an individual line a **"Slice Group"**.

The more slices we use to represent a line, the more precisely we can approximate the line slope; thus we represent lines with angles in the interval $[45°, 135°]$ ("vertical looking" lines) using horizontal slices, and we use vertical slices at any other angle range. Since pixels are strictly square shaped, the pixel image representation is better suited to rectangular slices and any other kind of slice (e.g. diagonal) will increase the slice group's slice length variations and the slice center's distance to the original



(a) (b) (c)

Figure 3.16: Slice group extraction explanation. (a) Original zoomed-in segment of a line. (b) The line can be described as series of horizontal slices (colored red and green to be made easy to distinguish) with their center (yellow) at sub-pixel precision coordinates. (c) The original line segment (red) can be recovered using statistical techniques.

line segment.

We can determine which type of slice to use from the average pixel angle $\theta_{avg} \in [0°, 180°)$ of the connected component, using eq. 3.13.

$$\text{Slice type} = \begin{cases} Horizontal, & \text{for } 135° \leq \theta_{avg} < 45° \\ Vertical, & \text{for } 45° \leq \theta_{avg} < 135° \end{cases} \tag{3.13}$$

Given a non-empty pixel p at position (i, j) in the binary image $B \in \{H, V, D, E\}$, we can scan the horizontal slice that contains p using Algorithm 6.

Each obtained slice object contains its first (p_0) and last (p_1) pixel positions, center position and length. We can also scan vertical slices using an analogous procedure to Algorithm 6, by displacing p_0 and p_1 between rows instead of columns. Note that we change the value of every visited pixel in B to 2 to avoid re-scanning pixels.

After scanning a slice, we can scan the next connected slice by repeating Algorithm 6 at $p = slice.center + (1, 0)$, and similarly the previous slice by scanning the position $p = slice.center + (-1, 0)$. Applying these offsets can be seen as moving the slice center *forward* or *backward* respectively. Analogous rules are applied to vertical slices

Algorithm 6: ScanHorizontalSlice

Data: B ($n \times m$ matrix $\mid b_{ij} \in \{0, 1, 2\}$), $i, j \in \mathbb{Z}$, type $\in \{horizontal, vertical\}$
Result: s (slice object)

begin
  $slice.p_0 \leftarrow (i, j)$
  $slice.p_1 \leftarrow (i, j)$
  $b_{ij} \leftarrow 2$
  do
    $slice.p_0 \leftarrow slice.p_0 + (0, -1))$
    $B(slice.p_0) \leftarrow 2$
  while $B(slice.p_0 + (0, -1)) = 1$
  do
    $slice.p_1 \leftarrow slice.p_1 + (0, 1))$
    $B(slice.p_1) \leftarrow 2$
  while $B(slice.p_1 + (0, 1)) = 1$
  $slice.center \leftarrow \frac{1}{2}(slice.p_0 + slice.p_1) + (0.5, 0.5)$
  $slice.length \leftarrow \|slice.p_0 - slice.p_1\|$
end

by using the displacements $(0, 1)$ and $(0, -1)$.

We detect all slice groups in $B \in \{H, V, D, E\}$ by repeatedly scanning slices while moving forward and backward from the starting point, as shown in Algorithm 7. Note that this algorithm stops scanning slices if the *shouldContinue(group, s)* function returns *false*, which occurs on any of the following events:

1. The last slice scanned touches the border of the image.
2. The last slice scanned has a length of 1.
3. The portion of the last slice that is directly connected (4-connection) to the previous slice is less than half the minimum of their lengths.
4. The distance between the line equation of the group so far and the center point of the last slice scanned is greater than half the slice's length.

Figure 3.17 shows the slice groups detected on the V angle interval of our example image. The stop conditions manage to distinguish overlapped elements even if they are connected to the endpoints of real lines (Figure 3.18). Condition (4) is only evaluated if an n_{min} sufficient number of slices has been scanned, and only every n_T

Algorithm 7: SliceGroupDetection

Data: B ($n \times m$ matrix $\mid b_{ij} \in \{0, 1\}$)
Result: *slice* (object)

```
begin
    L_G ← {∅}
    foreach b_ij ∈ B | b_ij = 1 do
        group ← {∅}
        s_0 ← scanSlice(B, i, j, type)
        s ← s_0
        while s.scan(B, type) ≠ ∅  ∧ shouldContinue(group, s) do
            group.add(s)
            moveForward(s)
        end
        s ← s_0
        moveBackward(s)
        while s.scan(B, type) ≠ ∅  ∧ shouldContinue(group, s) do
            group.add(s)
            moveBackward(s)
        end
        L_G.add(group)
    end
end
```

slices to improve performance.

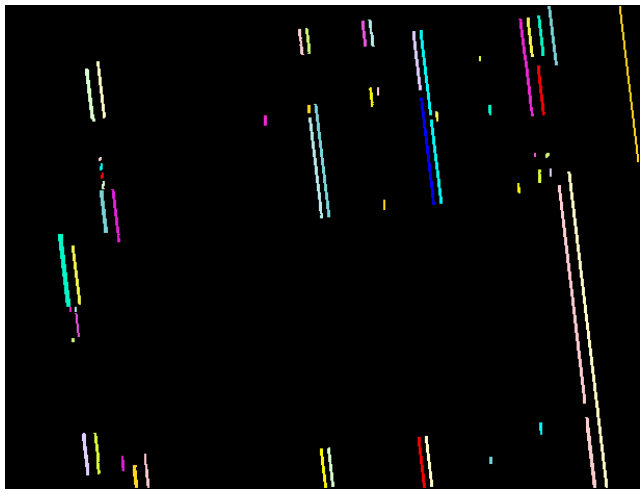
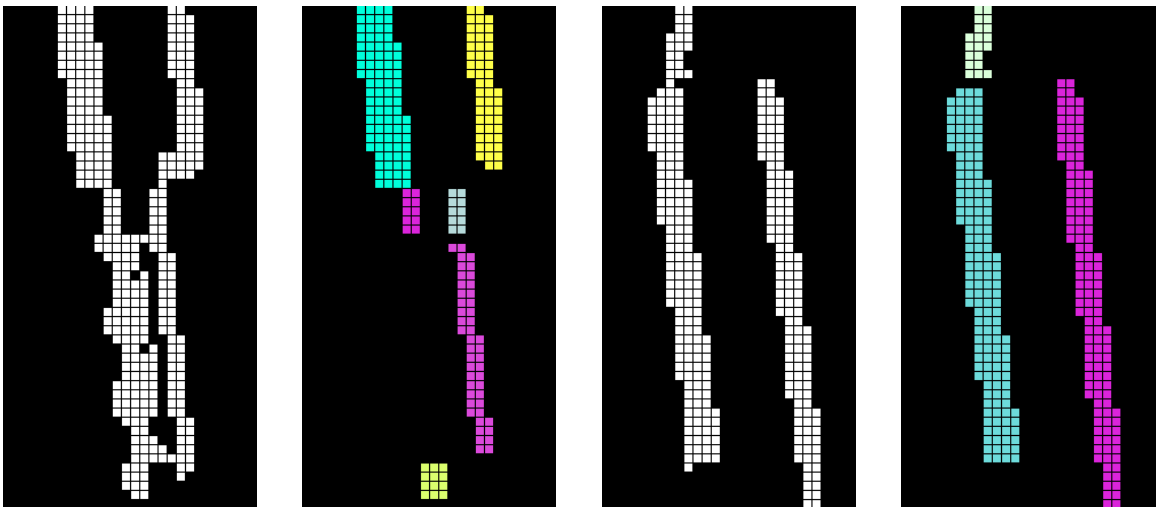


Figure 3.17: Slice group extraction results for V in our example image. Individual slice groups are colored for illustration.



(a) (b) (c) (d)

Figure 3.18: Examples of difficult scanning cases from Figure 3.17. (a) (Original) Wall lines connected to noisy portions of a window. (b) Slice groups detected; the noisy component is removed due to low correlation and the wall lines were disconnected from the window. (c) A similar wall-noise connection case. (d) The groups are separated via stop conditions.

Slice group vectorization via linear regression

After we detect a slice group, we can use linear regression by least squares fitting on the n slice center points to approximate the original line equation $y = mx + b$ shown in eq. 3.14 and obtained by direct application of eqs. (3.15) to (3.17), where m is the line slope and b is the line's intercept with the x axis.

$$y = mx + b \quad , \quad m = \frac{S_{xy}}{S_{xx}} \quad , \quad b = \bar{y} - m\bar{x} \tag{3.14}$$

$$\bar{x} = \frac{\sum x_i}{n} \quad , \quad \bar{y} = \frac{\sum y_i}{n} \tag{3.15}$$

$$S_{xy} = \frac{\sum (x_i - \bar{x})(y_i - \bar{y})}{n} \tag{3.16}$$

$$S_{xx} = \frac{\sum (x_i - \bar{x})^2}{n} \tag{3.17}$$

To avoid division by zero, our regression algorithm will first check if the center points

describe a perfect (or almost perfect) horizontal or vertical line, and in such case return the ideal line equation ($y = k$ or $x = k$, $k \in \mathbb{R}$) respectively.

We also obtain a statistical correlation coefficient $-1 \leq r \leq 1$ for the linear regression using eqs. (3.18) and (3.19):

$$S_{yy} = \frac{\sum (y_i - \bar{y})^2}{n} \tag{3.18}$$

$$r = \frac{S_{xy}}{\sqrt{S_{xx}}\sqrt{S_{yy}}} \tag{3.19}$$

The center points of a slice group representing a straight line are expected to have a strong correlation, while curves and noisy content will show weak correlation. We follow the well-known interpretation of r by Evans [39] and consider values of $|r| < 0.5$ as weak correlation, and ignore any slice group that meets this condition. As long as the slice group only contains pixels from one map element, this filter successfully removes curves and noisy connected components that look like lines.

Finally, we consider the external (first and last) slices in the slice group and obtain their closest position in the $y = mx + b$ line equation; these 2 points become the endpoints of the **vectorized line segment** equivalent to this slice group.

Clean-up and output

After detecting and vectorizing all the slice groups from each of the $\{H, V, D, E\}$ binary images, we merge every pair of slice groups that have one of their endpoints close to each other (under a small euclidean distance threshold d_{merge}) that have approximately the same line equation (under small tolerance thresholds m_d and b_d). We keep these thresholds very small because we will later repeat similar operations in the geometric domain. We also remove slice groups with less than 3 slices as they are usually noise. Figure 3.19 shows the final slice groups for our example floor plan section.

For every line segment, we approximate its transversal thickness t_t from its line angle

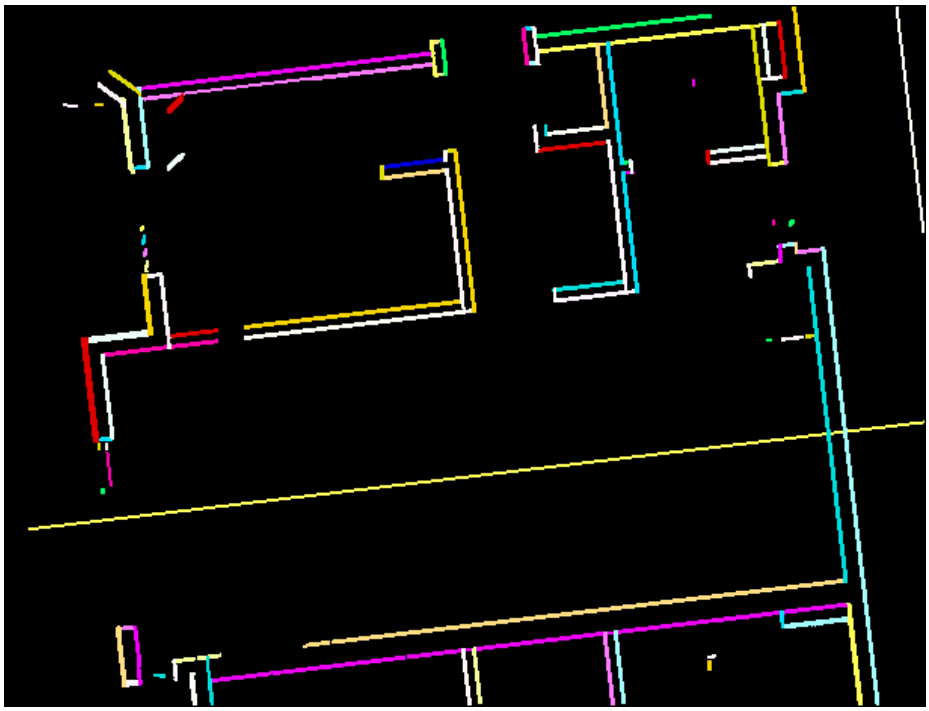


Figure 3.19: Final slice groups for the example map section.

$0° \leq a_l < 180°$ and average slice length l_{avg} using eq. 3.20 for horizontal slices and eq. 3.21 for vertical slices.

$$t_t = l_{avg} \times \sin a_l \tag{3.20}$$

$$t_t = l_{avg} \times \cos a_l \tag{3.21}$$

The concatenated collection of all vectorized line segments in $\{H, V, D, E\}$ is the final output of the *Wall Segment Candidate Detection* sub-module.

3.1.8 Geometrical Analysis

The purpose of this sub-module is to construct and return the final vectorized wall structure. It uses the total list of vectorized line segments L_s and the preprocessed binary image A as input.

The collection L_s can be manipulated using well-known vector operations and 2D

geometry. Although we implemented our own geometry library for this thesis, most programming languages already offer powerful and reliable computational geometry libraries (e.g. CGAL [40]) that can be used to implement this sub-module's functionality.

We execute the two sets of geometrical operations: **Parallel Segment Detection** and **Line Connection Analysis**.

Parallel Segment Detection

For every line segment $s \in L_s$ with line thickness t_0 and length l_0, we find the closest parallel line segment s_p with line thickness t_p, such that the perpendicular distance d_p between both segments meets:

$$max(t_0, t_p) \leq d_p \leq f_r \times l_0 \tag{3.22}$$

The lower limit of this interval avoids parallel lines so close to each other that they don't visually depict a volume between them, and the upper limits avoids lines too distant from each other, using a multiplication factor $0 < f_r \leq 1$ that guards our assumption that walls appear visually longer than thicker. Given the subjective nature of f_r, we prefer to use the conjugate of the golden ratio (0.61803) [41], a well-known aspect-ratio for rectangles that is successful in our dataset.

Additionally, to be considered a valid parallel segment, the length l_{proy} of the perpendicular projection of s_p into the line segment s must meet:

$$l_{proy} \geq f_{proy} \times l_0 \tag{3.23}$$

Where the multiplication factor $0 < f_{proy} \leq 1$ is the minimum portion of s that must be covered by the projection of s_p to consider both segments as part of the same wall. Lines of complex wall shapes often won't meet this criteria, but they can still be detected by other geometrical filters.

Figure 3.20 shows the result of the parallel segment detection. Most of the parallel

pairs detected will have a similar distance between them, except in cases where pairs of long unrelated lines are coincidentally parallel and close to each other.

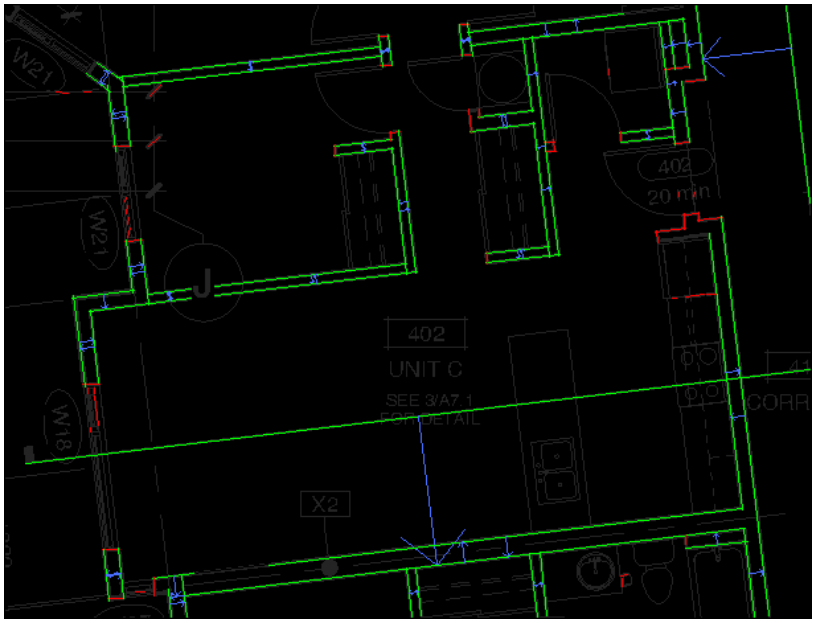


Figure 3.20: Parallel line segments (green) detected in the example map section. Line segments without a valid parallel segment appear in red. A blue arrow has been drawn between the s and s_p pairs found.

We assume that walls are the most common occurrence of nearby parallel lines of similar thickness in the image. Since we are only interested in parallel lines that are part of walls, we use the InterQuartile Range (IQR) [37] statistical method to detect and filter outliers in our detected pairs of parallel lines by their perpendicular distance d_p, as follows:

1. Create an array D_p with all the distances d_p between parallel segments found (length of all blue arrows in Figure 3.20).

2. Find the average line thickness t_{avg} for all parallel segments.

3. Obtain a histogram of d_p measurements.

4. Obtain the first and third quartiles Q_1, Q_3 for D_p, and the range length $IQR = Q_3 - Q_1$.

5. Calculate the minimum and maximum thresholds for parallel distance as follows:

$$dp_{min} = max(Q_1 - IQR, 2 \times t_{avg}) \tag{3.24}$$

$$dp_{max} = max(Q_3 + 1.5 \times IQR, 2 \times Q_3) \tag{3.25}$$

We suppose that any pair of detected parallel lines with a perpendicular distance d_p outside of the interval $[dp_{min}, dp_{max}]$ is a false detection and no longer consider them as parallel lines.

The original IQR rule for outliers [37] uses the interval $[Q_1 - 1.5 \times IQR, Q_3 + 1.5 \times IQR]$, which we have modified to meet some of our wall assumptions: Walls must depict a volume between them, and as they appear repetitively, we allow distances $\leq 2 \times Q_3$ for the common case of walls between units being 2 times thicker than normal walls.

Line Connection Analysis

For every line segment $s \in L_s$ with thickness t_s, we detect and store the segment s_{end} (with thickness t_{end}) each endpoint of s is connected to; such connection only exists if the distance d_{end} between their connected endpoints meets:

$$d_{end} \leq 2 \times max(t_s, t_{end}) \tag{3.26}$$

We set 2 times the line thickness as a threshold reference for deciding if two endpoints are connected because, due to the way we scan slice groups, both vectorized connected line segments will cross each other inside their own line thickness.

During this step we also detect if s connects to the body of s_{end} instead of its endpoints, or if the two line segments cross each other away from their endpoints by examining the minimum euclidian distance between the 4 endpoints involved and both line segments.

After detecting all line connections, we assume that wall lines connect to other wall lines (Section 3.1.1) and mark every line segment with an endpoint connected (directly or trough another segment) to a parallel line segment as a wall. Figure 3.21.(a) shows the result of this operation.

With the information gathered so far, we execute a series of connection filters in the following order:

1. Remove parallel lines (and any line connected to them) that cross the body of at least c_{max} other parallel lines.

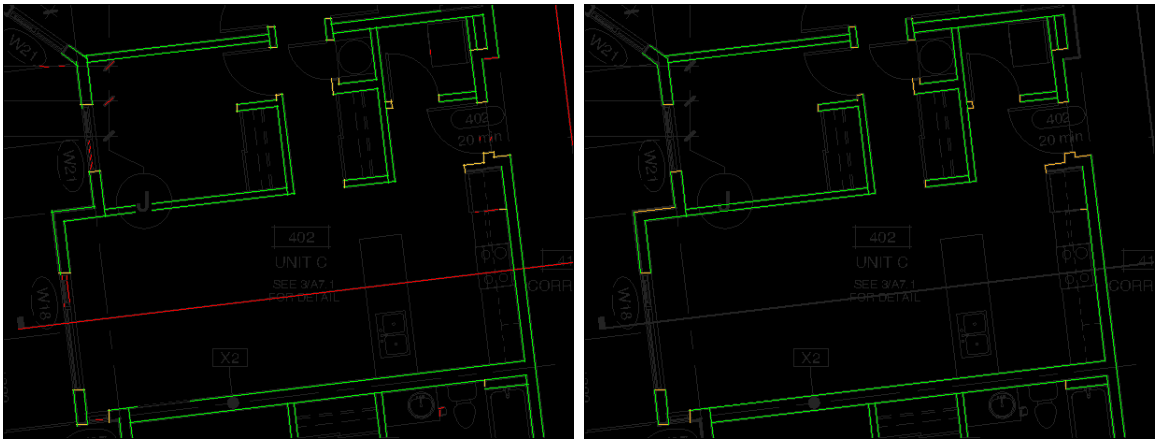
2. Remove parallel lines with both endpoints not connected to anything.

3. Merge co-linear segments that have disconnected endpoints, if said endpoints can be connected through a line of non-empty pixels in the binarized image A.

4. Remove any segment that is not parallel and is not connected (directly or indirectly) to a parallel segment.

5. For every endpoint that is connected to another endpoint, find the cross-point between both lines and consider it a *"corrected position"* for the endpoints.

After executing the connection filters we obtain the final wall extraction result in Figure 3.21.(b); This collection of line segments is the final **Wall Structure** output of the Wall Extraction method. We save this vector structure (in CSV standard format [42]) to re-use it in our Room Detection module and correction GUI.



(a) (b)

Figure 3.21: Line connection analysis results in the example map section. (a) Line connections detected. The long red line that crosses the right wall was rejected by the IQR method. (b) Final module output with line end-points corrected to match connection positions. Parallel lines appear in green and lines connected to parallel lines in yellow.

3.2 Room Detection

3.2.1 Design Assumptions

In the context of architectural floor plans, a room is a region of free space with a defined boundary that distinguishes it from other regions. Usually rooms are separated from other rooms by walls, windows and/or doors; On the other hand, rooms can be defined as well as a space region with a common purpose (e.g. kitchen, living room, bathroom, passageway, to name a few) and in many cases there's no element that separates one room from another (e.g. rooms shaped as perfect rectangles might contain a kitchen and a living room).

When the room boundary coincides with a closed loop of walls, windows and doors, they can be accurately detected using the approaches reviewed in Section 2.2. If these structural loops are not perfectly closed but their gaps are small enough, closing them (e.g. using geometry-based approaches) can still enable their detection.

However, when room boundaries don't match the wall structure, the missing boundaries (if ever detected) can be chosen from a theoretically infinite number of possible ways one could divide 2 or more regions, only guided by assumptions of what function the rooms are expected to fulfill. In practice, this leads to problems like under-segmentation, over-segmentation and wrong space divisions.

In practice, human operators will find different room shapes (all valid) when the space boundaries are not evident [32]. The problem of room detection in these cases is still unsolved and is addressed in literature with sentences like *"very subjective"* [13], *"splitting open rooms may fall into subjective decisions"* [32] and *"cannot be solved perfectly by purely syntactic methods"* [30].

This subjectivity aspect cannot be solved relying only on the wall structure, because in many cases there's no feature in the walls that hints the change from one room to another. Sometimes, the required information to discover the missing room boundary might not be present in the floor plan at all. In these cases, a human reader is able to identify the room regions by finding visual patterns in the floor plan symbols, text labels and other indications (e.g. dashed line regions), combined with their own contextual knowledge and natural brain mechanisms like the perception of illusory

contours [43].

A data set of map symbols in multi-unit floor plans doesn't exist at the time of this writing and creating one is a complex task that is beyond the scope of this thesis. Room labels are unreliable as (contrary to single unit floor plans) they don't appear so often in multi-unit floor plans.

Considering that our modules are parts of a bigger architectural floor plan understanding system, where other modules are dedicated to symbol spotting and text recognition, we decided to develop a method that detects under-segmented rooms and provides them to other modules that can later sub-divide them from a higher and more complete semantic perspective.

3.2.2 General Method Overview

Figure 3.22 shows the general steps of our room detection method, which receives the wall structure as input (obtained by the wall extraction module) and outputs the room regions in vector format.

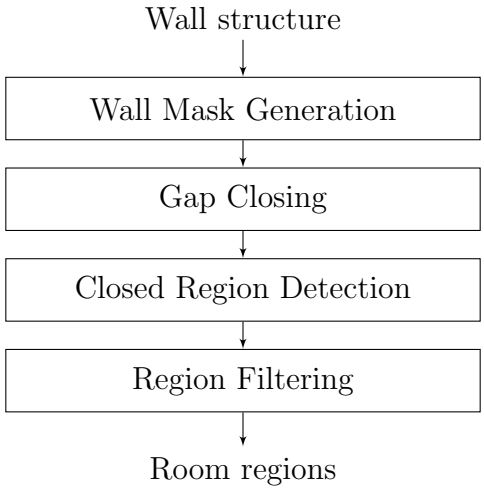


Figure 3.22: General steps of proposed room detection method

Our room detection pipeline is based on general concepts of mixed room detection methods referenced at Section 2.2.3, which combine geometric and pixel operations to detect rooms while keeping a degree of robustness against wall extraction errors. The outline of our method can be summarized as follows:

1. **Wall Mask Generation**: Use the wall structure to draw a binary mask of the walls. Our approach is robust to certain cases of wall extraction errors.

2. **Gap Closing**: Close gaps in the wall structure using geometrical algorithms to create closed regions.

3. **Closed Region Detection**: Detect significant connected components in the wall mask as room candidates.

4. **Region Filtering**: Decide which candidates are possibly rooms, and vectorize their region contours.

We present a detailed discussion of our room detection steps in the following sections.

3.2.3 Wall Mask Generation

The purpose of this step is to obtain a pixel-wise mask for the walls. Its inputs are the preprocessed binary image A ($n \times m$ matrix) and the list of line segments L_s obtained from the wall extraction module, and outputs a binary image $M_w, m_{ij} \in \{0, 1\}$ of the same size as A, containing the wall mask.

We produce the wall mask image in the following steps:

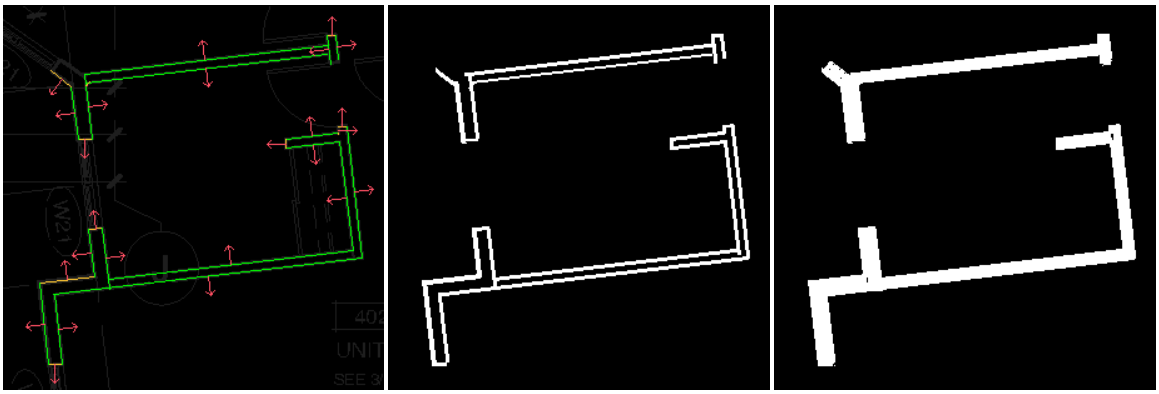
1. For every $s \in L_s$, obtain the normal unit vector $\bar{n_s}$ that points to the "outside" of the wall. We start by calculating $\bar{n_s}$ for parallel line segments since $\bar{n_s}$ will always point away from the other line in a parallel line pair, then we recursively propagate this orientation to all non-parallel connected lines until we find a segment that already has a value for $\bar{n_s}$.

2. Re-draw all line segments $s \in L_s$ into the empty matrix M_w, by repeatedly drawing slices of the same length and type as their original slice group. We draw all slices using a unique slice length value (the average slice length obtained on Section 3.1.8) to get rid of any overlapped noise, and generate any extra slices required after the geometrical modifications.

3. Fill the inside of the walls. For this purpose we project every pixel that belongs to the line segments in their respective -$\bar{n_s}$ direction (towards inside the wall), a maximum distance equal to the distance between the wall's parallel line

segments, until colliding with a non-empty pixel. If the line segment doesn't have a parallel line, use the distance from the nearest parallel pair that connects (directly or indirectly) to this segment.

Figure 3.23 illustrates each one of these steps in the top-left portion of our floor plan example. Note that the generated wall mask is able to reproduce the wall body even when the wall extraction module failed to detect both parallel line segments of a wall, which is a robustness feature of pixel-based approaches that we bring into our mixed method.



(a) (b) (c)

Figure 3.23: Example of wall mask generation. (a) Find the normal vector $\bar{n_s}$ for every line segment (red). (b) Redraw all line segments. (c) Project all line segment pixels inside the walls to fill them.

3.2.4 Gap Closing

The purpose of this step is to produce closed pixel regions in M_w by closing the gaps in the wall structure. Its inputs are the list of line segments L_s and the wall mask M_w, and outputs a binary image $M_r, m_{ij} \in \{0, 1\}$ of the same size as M_w, containing the closed regions.

We fill the gaps in the wall structure we create what we call *virtual walls*, by linking pairs of segments by their closest endpoints as long as:

- Both endpoints are collinear to the longest segment between the two.
- The line contained between them is empty in the wall mask.

- The distance between them is not longer than 2 times the length of the longest segment of the two.

To avoid connecting distant segments, we also consider a maximum gap distance equal to the length of the longest wall segment found. Additionally, to add tolerance to collinearity conditions we consider the line equation of the longest segment of the two and allow the closest endpoint of the other segment to be a maximum distance t_d from said line equation, with t_d being their average line thickness.

Figure 3.24 shows the connection results for our example image. Some approaches in our related work [32, 34, 12] prefer more aggressive gap closing routines, but considering the subjectivity component in room detection, we decided to leave further subdivision of our rooms to subsequent floor plan understanding modules that have also detected the map symbols and text labels (outside of the scope of this thesis).

Figure 3.24: Example of gap closing using geometric virtual walls, drawn in blue.

To finalize this step, we draw all the virtual walls into M_w, as lines of thickness 2. Since virtual walls are non-existent we want them to be as thin as possible, but we need a thickness greater than 1 to guarantee separated 4-connected components [44].

3.2.5 Closed Region Detection

In this step we obtain the closed pixel regions in M_w which are our room candidates. Its inputs are the list of line segments L_s, the wall mask M_w and the list of virtual walls L_v previously obtained.

To better explain this module's steps, we use the floor plan section in Figure 3.25 which covers a bigger area and has more diverse contents than the example from the previous chapter.

Figure 3.25: Example of the wall structure for a bigger map section.

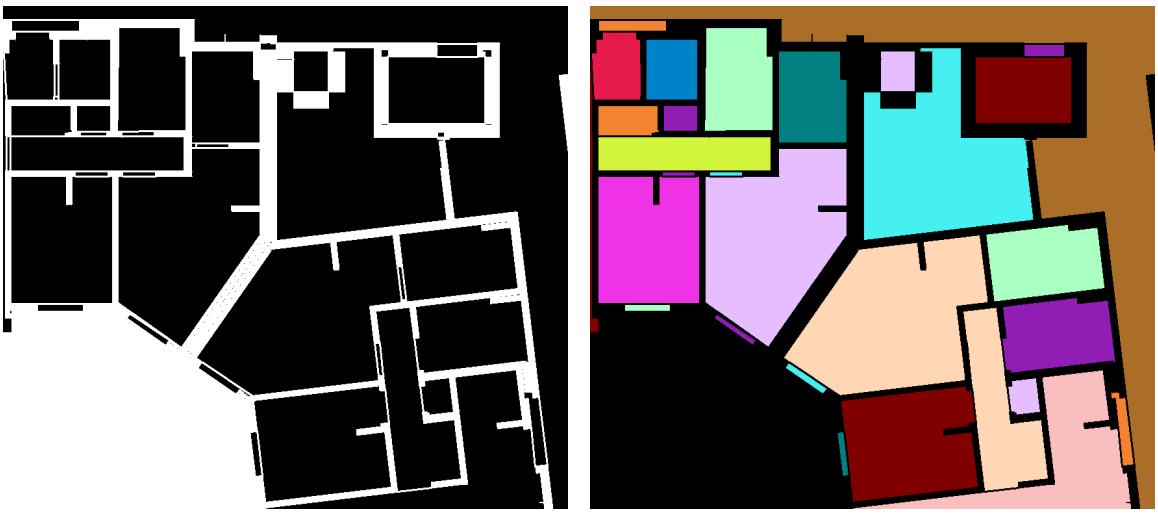
We follow this procedure:

1. Apply the flood fill algorithm [44] on the coordinate (0,0) of M_w with value 1; this will make the exterior of the building have the same value as the walls, as shown in Figure 3.26(a).

2. Invert M_w (switch ones and zeroes).

3. Detect all connected components in M_w with their area (total pixel count), as in Figure 3.26(b).

The connected components obtained this way include the rooms and other areas, which we filter in the next step.



(a) (b)

Figure 3.26: Closed region detection steps. (a) Flood-filled inverted wall mask. (b) Connected components.

3.2.6 Region Filtering

In this final step we filter the connected components so that only the room regions remain. The vectorized room regions obtained are the final module output. This sub-module has the following steps:

1. Filter small connected components by removing those with an area inferior to:

$$a_{min} = (d_{avg} + 2 \times t_{avg})^2 \tag{3.27}$$

   Where d_{avg} is the average distance between parallel segments and t_{avg} is the average line threshold. We consider that rooms cannot be smaller than a wall of the minimum possible length (a square wall).

2. Obtain the external contour of the remaining connected components (we use the border following method by Susuki & Abe [45]).

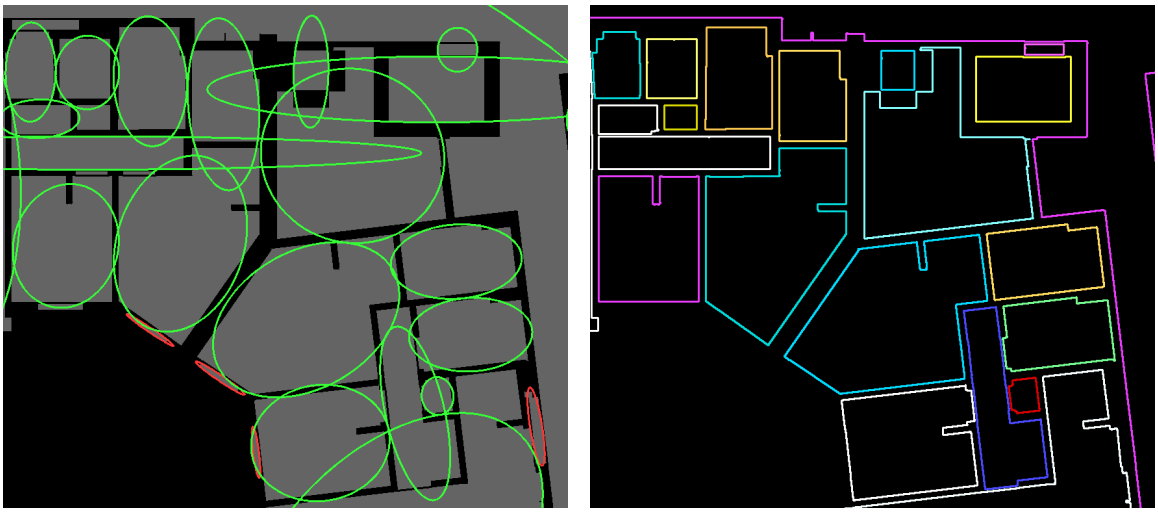
3. Vectorize the contours obtained, using an optimized vectorization algorithm (we use the approximation method by Teh & Chin [46]).

4. Remove any contour where the rotated height or width of the ellipse that fits

the contour points, is inferior to:

$$w_{min} = 2 \times d_{avg} + 2 \times t_{avg} \tag{3.28}$$

We use the Direct least square method [47] for fitting ellipses as shown in Figure 3.27(c).



(a) (b)

Figure 3.27: Examples of region filtering steps. (a) Ellipse approximation (green) with some filtered (red) via eq. 3.28; (b) Final vectorized room contours, re-drawn for illustration purposes.

The remaining vectorized region contours (as illustrated in Figure 3.26(d)) are the **final output** of the room detection module. We export these regions to a CSV file (as we did for the wall extraction module output) so that other modules and GUIs can make use of them.

4 Experimental Results

This chapter presents the experiments we carried out to evaluate the main sections of our approach. We first provide the technical context and configuration for our experiments in Section 4.1. Section 4.2 describes experiments on the Slice Transform and Angle Matrix generation methods. We present experiments evaluating our Wall Extraction and Room Detection modules against our own image dataset in Sections 4.3 and 4.4. Finally, in Section 4.5 we describe an experiment using a mobile device application to explore a secondary result that might motivate future work.

4.1 Implementation Technical Overview

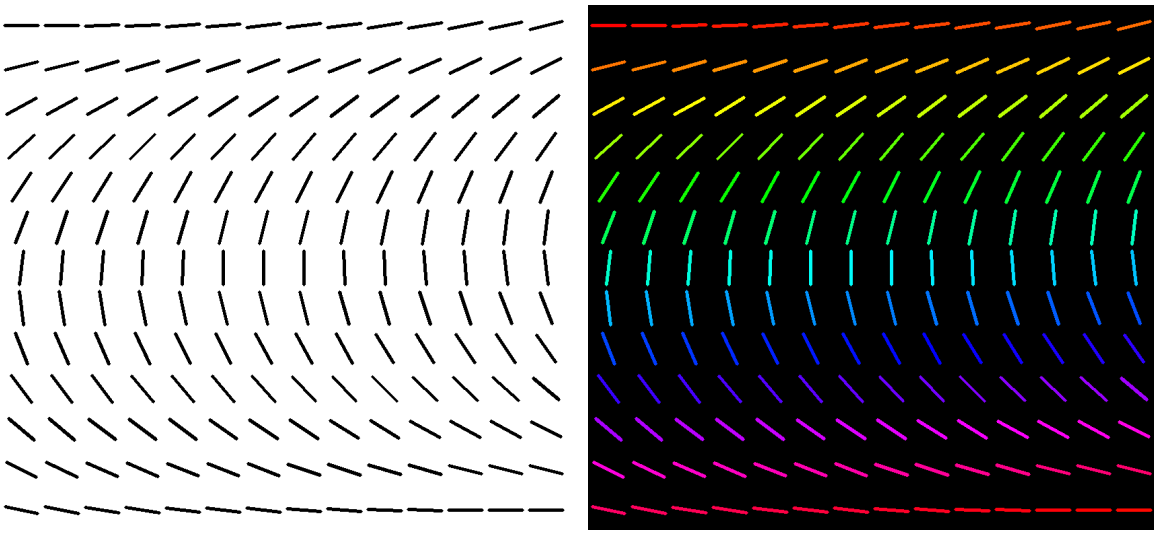
For experimentation, the wall extraction and room detection methods were implemented in C++ using the OpenCV (3.4.1) computer vision library, and executed on the desktop PC and mobile device with the technical specifications described in Appendix B.

An additional experiment performed on a mobile device was implemented using native C++ for Computer Vision and Java for GUI and OS-specific interaction, linked using the Android Native Development Kit (NDK) and the Java Native Interface (JNI).

The GUI prototype tool used during our experiments was developed using MATLAB R2017b.

4.2 Evaluation of the Angle Matrix Generation

Our wall extraction method is based on the results obtained from the slice transform and particularly depend on the angle matrix (Section 3.1.6), responsible for finding the approximate pixel-wise line angle. We test its performance by quantitative evaluation against synthetic images of lines at angles $\in [0°, 180°)$ like the one shown in Figure 4.1.



(a) (b)

Figure 4.1: Test image 1: Angle matrix synthetic test. (a) 180 lines at different angles, of length 50px and thickness 3px (a common thickness in our dataset). (b) Angle hue map of the ground-truth.

Although none of the test lines are connected or intersected (as is often the case in real floor plans), we discussed in Section 4.2 how the areas prone to errors gather around the line corners and overlapped line sections, which we assume are not a significant portion of the total line length (otherwise even humans would have problems perceiving the line, and we presume a communication purpose on the floor plan). We're interested in the non-overlapped sections of the lines, as they can be used by other sub-modules to reconstruct missing or noisy sections.

To compare M_a to the ground-truth, we measure the *Accumulated error* as the total sum of the per-pixel angle difference to the ground-truth and *Average error* as the global average of said per-pixel difference. We calculate the difference between angles using Eq. 3.11 of Section 3.1.6.

In our quantitative results, accumulated error measures appear rounded for simplification. All time measurements are the average of 5 task executions. In all cases, we run the Slice Transform in a single thread (no parallelism) to facilitate time measurement.

4.2.1 Angle Approximation Accuracy

In this experiment we evaluate the accuracy of the method against lines of different length, angle and thickness. All experiments have blur level 3 and $d_{border} = 2$ for 180 lines with angles $\in [0°, 180°)$.

Length Variations

We perform multiple experiments by varying line length (all other parameters unmodified) and obtain Table 4.1.

Length	Time (ms)	Total pixels	Accumulated error	Average error
40	133	21592	36349	1.68°
50	174	27034	41071	1.52°
60	190	32394	46713	1.44°
70	221	37850	49932	1.32°
80	264	43200	55887	1.29°
90	311	48571	60080	1.24°
100	340	53955	63290	1.17°
110	383	59226	66991	1.13°

Table 4.1: Angle matrix error measurements when varying line length.

We observe how the *Average Error* stays around 1°-1.5° and diminishes as we increase the *Length* of the lines. We mentioned in Section 3.1.6 that the AngleMatrix algorithm has difficulties near the line end-points (as shown in Figure 4.2), and the longer the test lines, the less pixels near the end-points make a difference on the total error measurement.

Thickness Variations

We perform multiple experiments by varying line thickness (all other parameters unmodified) and obtain Table 4.2.

Thick-ness	Time (ms)	Total pixels	Accumulated error	Average error
2	97	14456	29385	2.03°
3	134	21592	36349	1.68°
4	180	28849	69319	2.40°
5	202	36036	111761	3.10°
6	221	43168	164275	3.81°
7	256	50500	244478	4.84°
8	304	57756	334225	5.79°
9	344	64951	437667	6.74°

Table 4.2: Angle matrix error measurements when varying line thickness.

We observe how the *Average Error* increases with the *Thickness* of the lines, except at thickness 2 where the lines are so thin it becomes harder to gather complete slices. Increasing the line thickness without increasing the length makes the problematic areas near the line corners bigger (more pixels) compared to the rest of the line, affecting the global result. However, bodies as thick and short as in our last experiment (Figure 4.2) don't usually represent lines in floor plans, but rather polygons.

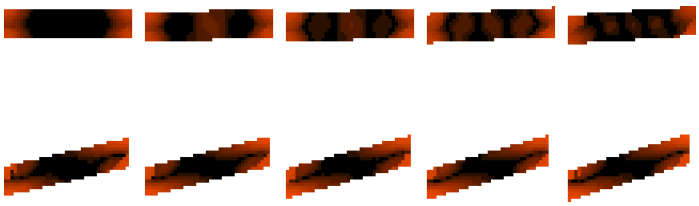


Figure 4.2: Average error values tinted in orange for lines with thickness 9 and length 40 px. Average error is higher on pixels near each line's endpoints (line corners).

The data in tables 4.1 and 4.2 includes time measurements for both experiments and the total number of non-empty pixels. As shown in Figure 4.3, even when considering the differences between both experiments, the time taken to obtain the AngleMatrix appears to relate to the number of non-empty pixels in the image with a linear relationship, which agrees to our discussion on complexity in Section 3.1.4.

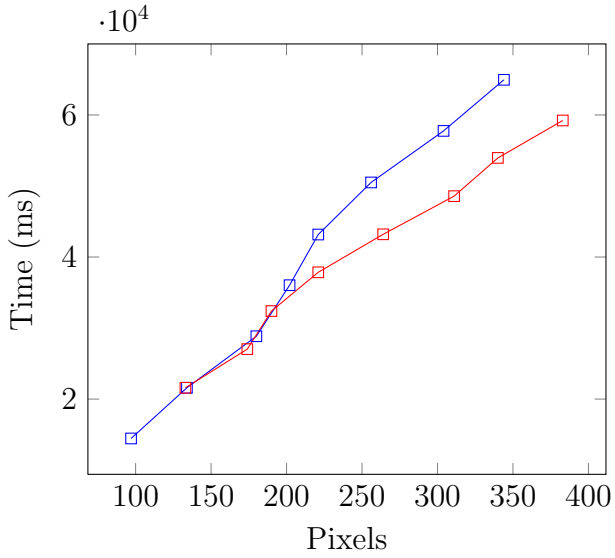


Figure 4.3: Time vs. Number of pixels for 2 experiments: Thickness change (blue) and length change (red).

4.2.2 Angle Blur Efficiency

To check the performance impact of the angle blur filter, we compare the output M_a of the Angle Matrix Generation sub-module to our Test Image 1 ground-truth in different blur levels and obtain the data in Table 4.3.

Blur level	Time (ms)	Accumulated error	Average error	Error reduction	ERE
0	82.1	210660	4.46	-	-
1	127.5	136173	2.88	35.36°	0.278%
2	192.3	112337	2.38	17.50°	0.091%
3	260.8	96671	2.04	13.95°	0.053%
4	312.1	84913	1.79	12.16°	0.039%
5	395.0	76926	1.62	9.41°	0.024%

Table 4.3: Quantitative evaluation of consecutive conditional blur runs. Time measurements obtained from the average of 10 iterations.

In Table 4.3, *Accumulated error* is the total sum of the per-pixel difference to ground-truth. *Average error* is the average per-pixel difference to ground-truth. *Error reduction* is the reduction percentage in accumulated error since the last blur level. *ERE* is the *Error Reduction Efficiency*, obtained from Eq. 4.1 and plotted in Figure 4.4.

$$ERE = \frac{ErrorReduction}{time} \tag{4.1}$$

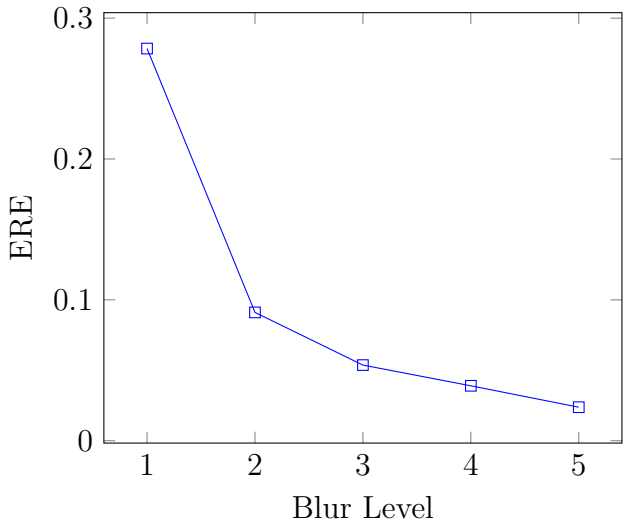


Figure 4.4: Blur Level vs. Error Reduction Efficiency.

In this experiment we observe that applying the ConditionalBlur algorithm multiple times will reduce the total error measurements, but the error reduction percentage obtained with each new ConditionalBlur application will diminish in an exponential rate, while the time required for the full process increases linearly. The blur level for the wall extraction module should be chosen while considering this trade-off between speed and approximation accuracy.

The *Average Error* column at Table 4.3 gives us a reference for configuring the *Angle Border Distance* (d_{border}) threshold described at Section 3.1.7.

4.3 Evaluation of the Wall Extraction Method

In this experiment we evaluate the wall extraction method against our own labeled dataset of multi-unit floor plan images. We perform quantitative evaluation in terms of precision and recall, by executing the full method pipeline with the following configuration:

Binarization Threshold (b_t): 20	Blur level: 3
Thickness factor: 2	Angle Border Distance (d_{border}): 2
Slice group min slices (n_{min}): 10	Slice group slice period (n_T): 20
Slice group merge distance (d_m): 2	Line equation tolerance (m_d, b_d): 0.02, 1
Parallel ratio factor (f_r): 0.61803	Parallel projection factor (f_{proy}): 0.3

Table 4.4: Wall extraction configuration parameters.

These configuration values were obtained by manual adjustment for the images in our dataset (which differ from each other in resolution, notation, complexity and challenges as explained in the next section), and would be user-adjustable in a final user product.

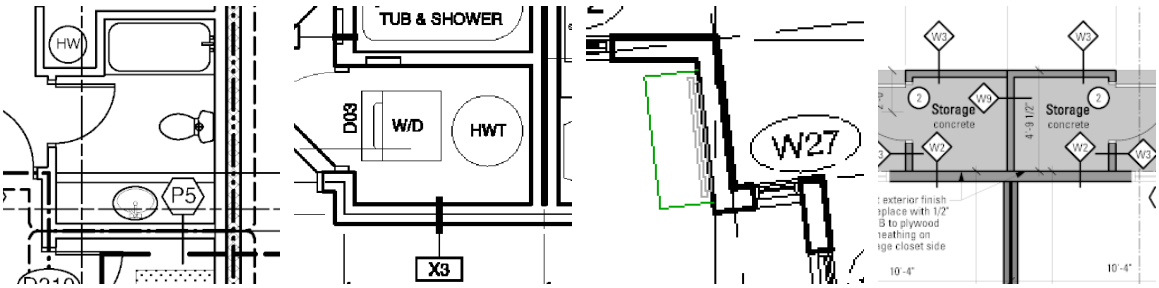
4.3.1 Dataset Generation

Our industrial partner has provided (under confidentiality agreements) a diverse set of real multi-unit architectural floor plan images from different firms and characteristics. From these images, we selected a subset of 10 images (further described in Appendix A) that exhibit the following challenges:

- Horizontal, vertical and diagonal walls.
- Different authors and notations.
- No walls represented as single thick lines (as the methods in Section 2.1 already provide solutions for extracting this wall notation).
- Overlapped elements of different thickness and at different angles.
- Non-wall lines with the same thickness as wall lines.
- Errors in the drawing technique and small content destruction (e.g. line discontinuities due to overlapping gray elements).

Figure 4.5 shows examples of challenging sections in our dataset.

The original floor plans are surrounded by a rectangular frame and text legend; We trimmed the main floor plan image area manually. By design, we expect this image trimming process to happen in an external module prior to wall extraction.



(a) (b) (c) (d)

Figure 4.5: Examples of challenges in our dataset. (a) Section from image 3 with content overlapping and different wall patterns. (b) Section from image 9 with a notation that represents walls with a line in the middle. (c) Section from image 1 with diagonals and content destruction caused by overlapping black & grey symbols. (d) Section from image 4 with different notation and background.

For every dataset image, we manually created a corresponding ground-truth image, using an image editor to paint all pixels that belong to wall lines. The ground-truth creation task is time-consuming, as differentiating the walls can be difficult, the number of lines to paint (per image) can reach the hundreds and drawing pixel-wise ground-truth requires attention to detail. Figure 4.6 shows a ground-truth image example.

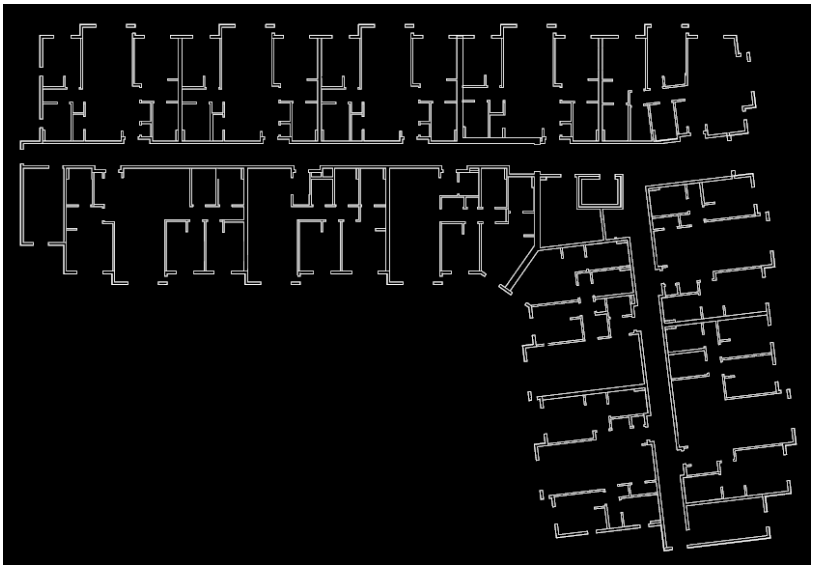


Figure 4.6: Example of dataset ground-truth image for test 1; White pixels represent wall lines. Original image size: 4,162 x 3,095 pixels.

4.3.2 Quantitative Results

We execute the Wall Extraction pipeline on the 10 test images in our dataset and compare the result to our ground-truth in terms of Precision and Recall, obtained via 4.2 where TP *(True Positives)* are the wall pixels correctly detected, FP *(False Positives)* are the pixels incorrectly detected as part of a wall, and FN *(False Negatives)* are the wall pixels not detected. Table 4.5 shows our final quantitative results.

$$Precision = \frac{TP}{TP + FP} \qquad Recall = \frac{TP}{TP + FN} \tag{4.2}$$

No	Time(ms)	Width	Height	Wall Pixels	Precision	Recall
1	11,144	4,162	3,095	235,178	0.964	0.953
2	55,335	9,427	5,445	708,253	0.424	0.941
3	18,647	2,618	5,709	245,541	0.574	0.701
4	39,137	9,779	4,961	1,081,995	0.905	0.901
5	41,951	9,053	6,171	1,083,540	0.770	0.950
6	10,458	4,774	3,212	258,132	0.723	0.953
7	18,091	5,544	5,995	416,496	0.809	0.878
8	4,019	2,679	1,338	162,375	0.844	0.879
9	12,076	4,603	3,201	367,189	0.774	0.971
10	6,191	3,404	2,756	207,492	0.880	0.966

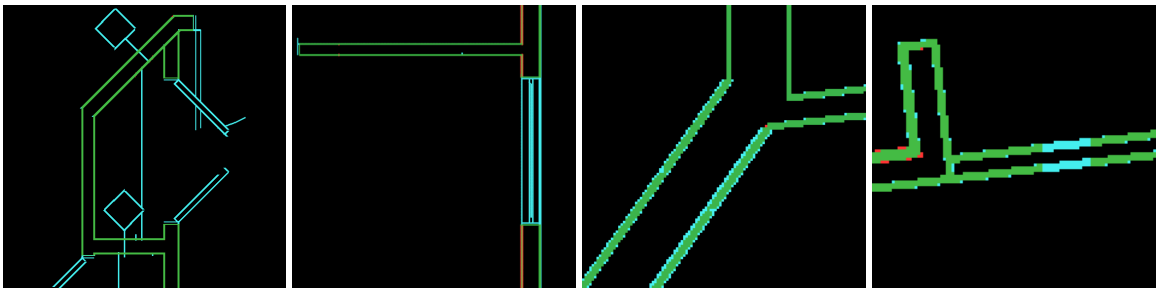
Table 4.5: Wall extraction quantitative evaluation results. Time was obtained as the rounded average value of 5 experiment repetitions.

We make the following observations from our results:

- Our values of recall are always higher than our precision. This is a desirable characteristic for our method as in a real application (which in this context has no tolerance for errors), it's easier for the user to delete existing lines than to draw non-existing ones.

- Our best results are obtained on images exported at a resolution of 150 DPI. In contrast, our worst results occur on images exported at resolutions of 300 DPI; At this resolution, lines are so big that even small details in floor plan symbols are detected as lines (as they often contain rectangles). Figure 4.7.a illustrates this problem, which we presume could be approached with additional geometrical filters.

- Some of our false positives are caused by wall windows in certain notations (Figure 4.7.b) that are mistaken as part of a wall; in our approach this error facilitates detecting rooms because no gap in the wall needs to be closed.
- There are numerous false positives and false negatives appearing around diagonal walls, due to our diagonal line drawing algorithm not being exactly the same originally used to draw the floor plans (Figure 4.7.c). Although Bresenham's line drawing algorithm [48] is a well-known method adopted in many real applications, there are other techniques (like Wu's [49]) to consider.
- A small quantity of false positives are caused by our geometric algorithms solving line discontinuities (Figure 4.7.d), which could be a desirable result because filling gaps in the wall structure contributes to the room detection process.
- Compared to the AngleMatrix experiments in Section 4.2.1, the complexity of real floor plans is harder to measure using global quantities (like processing time vs. number of non-empty pixels) because the wall extraction module includes a broader series of steps on both image and geometrical analysis. However, we still observe that higher amounts of wall pixels are associated with higher processing times.



(a) (b) (c) (d)

Figure 4.7: Observations in the wall extraction results, colored: Green (True Positives), Cyan (False Positives), Red (False Negatives). (a) Doors of sufficient size and thickness can be mistaken for walls. (b) Windows of certain notations are detected as part of the wall. (c) Diagonal lines don't exactly match the originals as they are drawn with different algorithms. (d) Some discontinuities are solved during geometrical analysis.

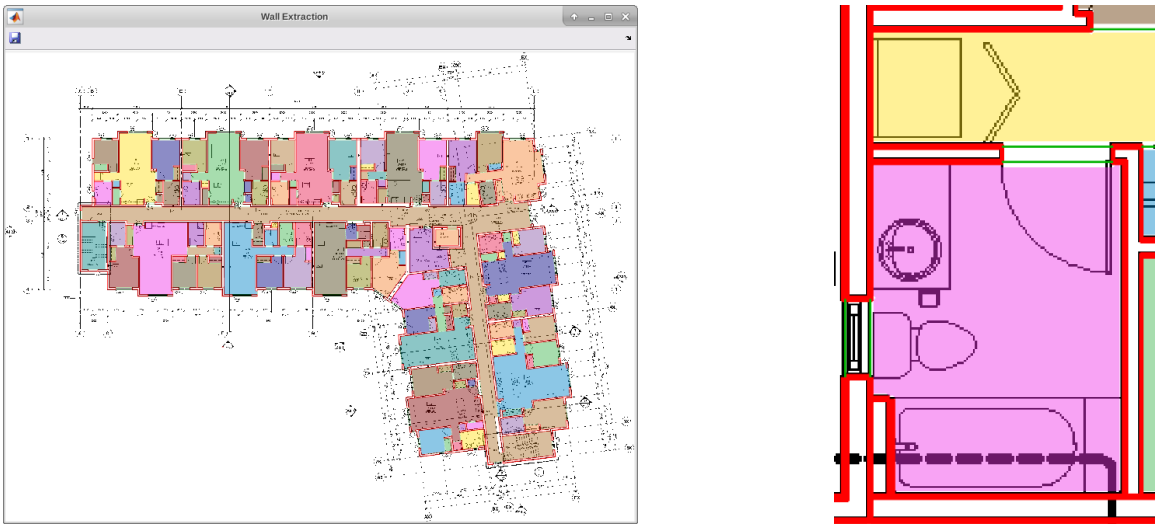
4.4 Evaluation of the Room Detection Method

Evaluating room detection is challenging because defining the "correct" boundaries for rooms can be subjective (discussed in Section 3.2.1). Instead, we evaluate our room detection method using a GUI tool to measure the minimum number of user corrective actions required to detect the rooms under the following criteria:

1. Rooms are regions of empty space inside the building.
2. Room regions are divided by doors (closet and cabinet doors included).
3. Rooms are always surrounded by walls, even if not in direct contact with them.

To avoid the propagation of wall-extraction related errors into this experiment, we required an accurate vectorized wall structure to serve as starting point, which we obtained from the pixel-wise wall ground-truth created for the experiment in Section 4.3.

Figure 4.8 shows the correction GUI with an example image loaded, with the corresponding walls and rooms overlaid on top of it. Wall lines appear in red, virtual walls in green and rooms appear as semi-transparent areas in different colors.



(a) (b)

Figure 4.8: Room detection GUI with test image 1 loaded. (a) General view. (b) Zoomed region with wall lines in red, semi-transparent colored rooms and green lines as virtual walls. Note that the cabinet in the yellow room of (b) was not detected as a separate room.

Since the initial wall structure is correct, only the virtual walls need modifications. The GUI has the following interaction functionality:

- The image can be zoomed & panned using the mouse wheel and right button.
- Virtual walls can be selected with a mouse click, revealing their endpoints.
- A selected virtual wall's endpoints can be dragged around.
- Selected virtual walls can be deleted with the Delete key.
- A new virtual wall can be added at the mouse position by pressing *Ctrl+N*.

Internal counters keep track of the number and type of user actions:

- *Additions*: Times the user added a new virtual wall.
- *Edits*: Times the user modified a virtual wall's endpoints.
- *Deletions*: Number of virtual walls deleted.

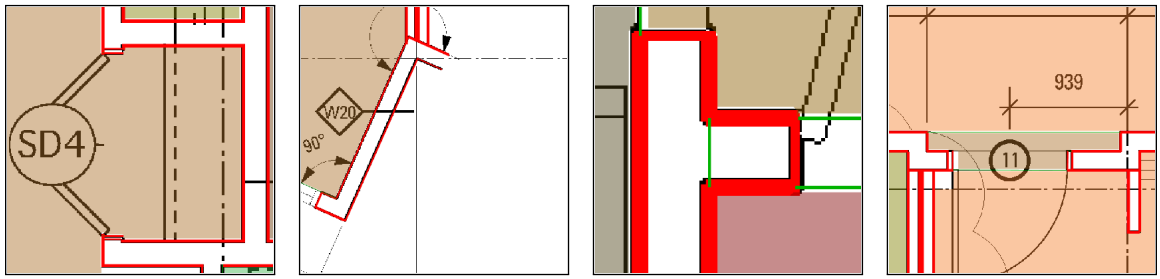
This experiment's results are shown in table 4.6 for each test case, where *Actions* is the sum of duplicates, edits and deletions, and *Change* is a quantity obtained by dividing the virtual walls by the number of actions, which we use as a relative error measure. We also include the count of cabinets and closets because most of the additions were performed to close these regions.

No.	Cabinets & Closets	Virtual walls	Additions	Edits	Deletions	Actions	Change
1	72	405	24	18	24	66	0.163
2	21	123	10	8	17	35	0.284
3	13	135	1	9	6	16	0.118
4	8	161	4	5	16	25	0.155
5	16	135	12	9	20	41	0.303
6	22	161	4	3	33	40	0.248
7	4	98	9	6	5	20	0.204
8	8	144	8	7	12	27	0.187
9	18	203	14	1	15	29	0.142
10	15	146	3	2	19	24	0.164

Table 4.6: Room detection quantitative evaluation results, where lower Change is better. Virtual walls is the number of virtual walls before manual correction.

We make the following observations from this experiment:

- Additions & edits were often required to separate cabinets and closets from the containing room (e.g. the cabinet at Figure 4.8.b and the closet at Figure 4.9.a). Detecting their door symbols would improve our quantitative results.
- Additions were required in cases where the the wall structure has been cut by the original author (e.g. Figure 4.9.b) since the missing boundaries had to be drawn manually. This reveals a limitation of our room detection method since it assumes rooms are always surrounded by walls.
- Edit actions were caused by virtual walls not connecting to the right wall endpoint, in cases where the wall structure was symmetrical enough to confuse the gap closing algorithm.
- Most deletions removed undesired virtual walls connecting small symmetrical gaps in the wall structure, for example at "T shaped" junctions (e.g. in Figure 4.9.c). These small virtual walls could be automatically removed with additional geometrical filters on the minimum virtual wall length.
- Some deletions removed virtual walls created to connect parallel walls of similar characteristics that faced each other (Figure 4.9.d). These actions could be avoided with a more robust set of geometrical rules for creating virtual walls.
- There's no clear relationship between these results and our wall extraction evaluation. We observe that although our room detection module relies only on the wall structure, it's insufficient for a more accurate detection.



(a) (b) (c) (d)

Figure 4.9: Common error cases observed. (a) A cabinet door gap not detected. (b) Intentionally cut walls fail to produce rooms. (c) A virtual wall created inside a "T" wall intersection. (d) Wall corners incorrectly connected.

For this experiment, we performed the operations without time limit, aiming for the

least operations possible to satisfy the room criteria. We leave for future work a more complete evaluation experiment with a group of operators of different training levels and a bigger dataset.

4.5 Discussion on Mobile Device Implementation

This experiment aims to gain insight into a secondary result not related to the focus of this thesis.

In Section 3.1.4 we described the Slice Transform as robust to image noise and fast enough for usage in real-time applications on mobile devices. In this context we consider *"real-time"* as being able to process every frame provided by the device's camera before the arrival of the next frame, avoiding latency problems perceptible for the user. This would mean that our theoretical contributions could be relevant in other applications that deal with line drawings under performance considerations.

In support of this claim, we implemented a mobile application to demonstrate the slice transform's capability to perform image processing in real-time, using the device described in Appendix B.

This simple application would read the camera video and color lines on each frame a different color depending on their angle. On every camera video frame M ($m \times n$ matrix, $m_{ij} \in \mathbb{R}^3$) this application performs the following process:

1. Threshold M by gray intensity level on a fixed low threshold (leaving only the darkest pixels) into a binary image B.

2. Execute Slice Transform and Angle Matrix Generation on B with blur level 3. We obtain pixel-wise line angle values a_{ij} for every $b_{ij} \neq 0$.

3. Modify the original M color image, so that every pixel position that is not empty in B, gets replaced by a color as follows:

$$M_{ij} = \begin{cases} Red, & \text{for } 157.5^\circ \leq a_{ij} < 22.5^\circ \\ Cyan, & \text{for } 22.5^\circ \leq a_{ij} < 67.5^\circ \\ Green, & \text{for } 67.5^\circ \leq a_{ij} < 112.5^\circ \\ Yellow, & \text{otherwise} \end{cases} \tag{4.3}$$

We test this mobile application on 3 line drawings: a sharp sign, a simple house drawing and a printed section of a floor plan. Figure 4.10 shows some frames from a live screen recording of this experiment.

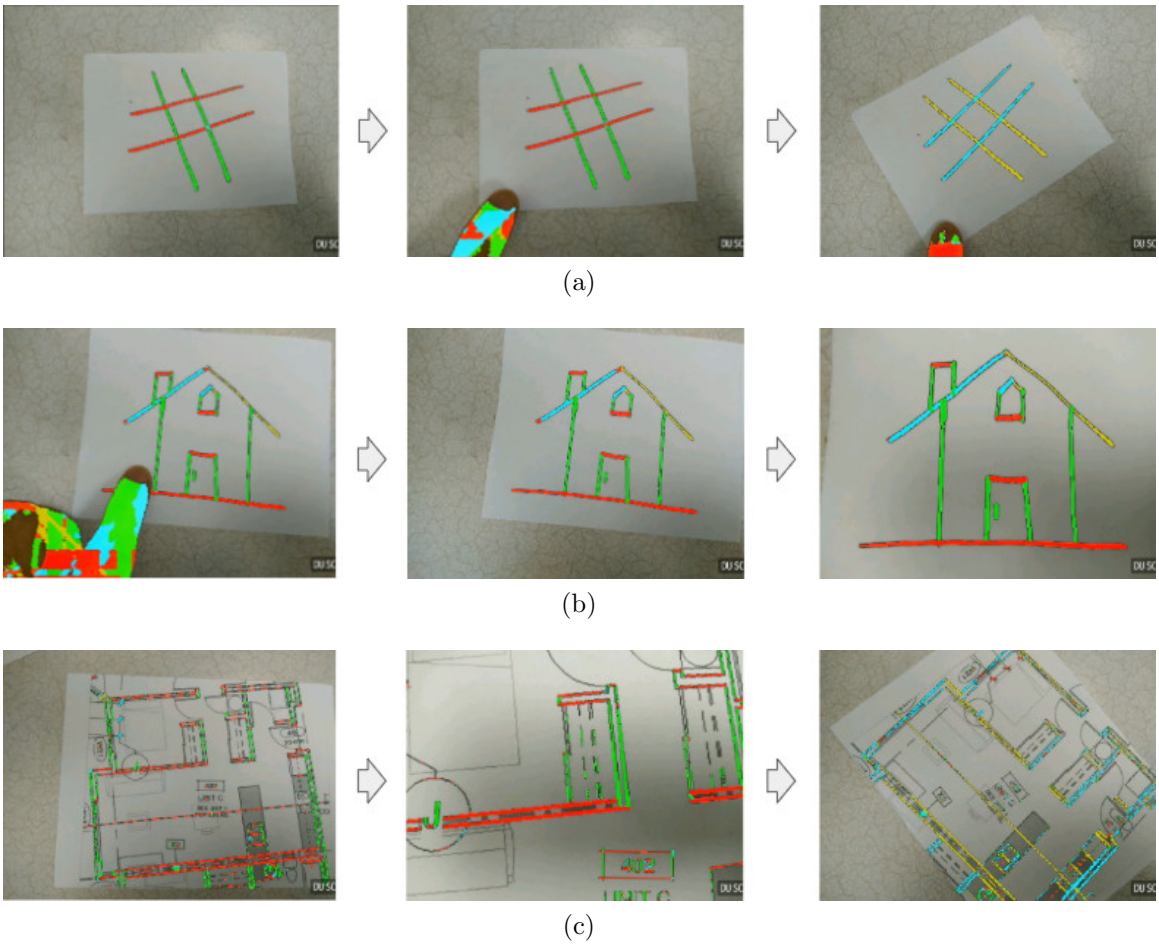


Figure 4.10: Frame examples from the real-time mobile application. (a) A sharp sign drawing rotated. (b) A simple line drawing zoomed-in. (c) A printed floor plan section zoomed-in and rotated.

The experiment was recorded in the same mobile device, using a commercial screen-recording tool [50]. We gathered the following findings from this experiment:

- Camera video resolution was a major component in maintaining the application responsive and fluid. We used lower camera video resolutions (640 x 480 pixels and below) to allow the algorithm to keep up with every new frame captured.
- Although most of the experiment was fluid, we encountered some latency issues (only a few frames skipped) when the number of thresholded dark pixels

increased (e.g. when a finger got processed); this agrees with our discussion on the algorithm's performance (Section 4.2).

- Major latency issues occurred when the camera got partially or completely covered (dark); these situations should ideally be detected before processing the video frames.
- Our algorithms are able to process images from a noisy origin (dark lightning, shaky camera, different distance and rotations). This suggests a possible future application in real-time video processing of hand-drawn line drawings.

5 Conclusions

This study proposes a novel method for performing wall extraction and room detection on architectural floor plan images. Our method is aimed at solving the particular challenges of multi-unit (e.g. Residential condominiums) floor plans, which are created with a different purpose and complexity from single-unit floor plans. These challenges include a bigger image size, notorious presence of same-color overlapping elements, visual similarities between unrelated floor plan elements and a higher relevance of walls at diagonal angles. Our methods are designed to be implemented as the initial modules of a broader floor plan understanding system.

The steps of our wall extraction method are based on diverse applications of a novel image transform operation we call the *Slice Transform*. We use the Slice Transform to solve the problem of recognizing overlapped lines and reuse its output to approximate the angle and thickness of the lines on the image, which allows us to design a set of algorithms for extracting the wall structure. We consider the slice transform and its applications as the main theoretical contribution of this thesis.

The multi-threaded version of the slice transform is fast enough to suggest its possible usage on desktop software and mobile applications related to line drawings. We further explored this secondary result by successfully implementing our algorithms in a mobile application capable of real-time image processing.

Our room detection method combines the robustness of pixel-based methods and the informational richness of geometry-based methods.

The key advantages of our methodology for floor plan analysis are:

1. The use of the slice transform to approximate line angles is suited to diagonal lines of any angle and allows us to separate intersected lines without losing their original shape. The same data can be reused to filter overlapped elements not deemed as wall lines by their thickness.

2. Our room detection method is able to recover from certain types of wall extraction errors, by relying on information from connected lines to reconstruct the wall's original body.

3. The speed of both wall extraction and room segmentation makes them suitable for end-user applications, and their output in vector format allows further geometrical analysis and direct connection with other floor plan analysis modules.

The identified limitations are:

1. Our room detection method only relies on the wall structure for proposing an initial set of regions, which is insufficient to accurately detect all rooms. This could be improved with a higher quantity of available semantic information, including the localization of symbols, text labels when available and other visual indications of space division.

2. The gap closing method in our room detection method is sensitive to symmetrical structures, where it tends to create unnecessary virtual walls.

3. Our room detection method requires the wall structure in the floor plan to be complete (e.g. the image contents haven't been intentionally cut) and rooms to be surrounded by walls.

4. Our wall extraction method was designed to prefer mistaking non-wall elements as walls (rather than the opposite), maximizing recall while sacrificing precision. This is motivated towards easing the correction process for the user (as real applications don't tolerate any error on the wall structure) but can cause lower precision in cases of heavy overlapping.

5. We require additional geometric analysis to support images exported at high resolutions (e.g. our dataset image 2 exported at 300 DPI), since certain rectangular floor plan elements may appear big enough to resemble small walls.

5.1 Future Directions

Our approach could also be tested against single-unit floor plan datasets [19, 7, 25], after adding the geometrical filters required to make it robust to their particular characteristics (e.g. Elements of increased artistic detail as illustrated in Figure 5.1), considering that these images follow a different end purpose.

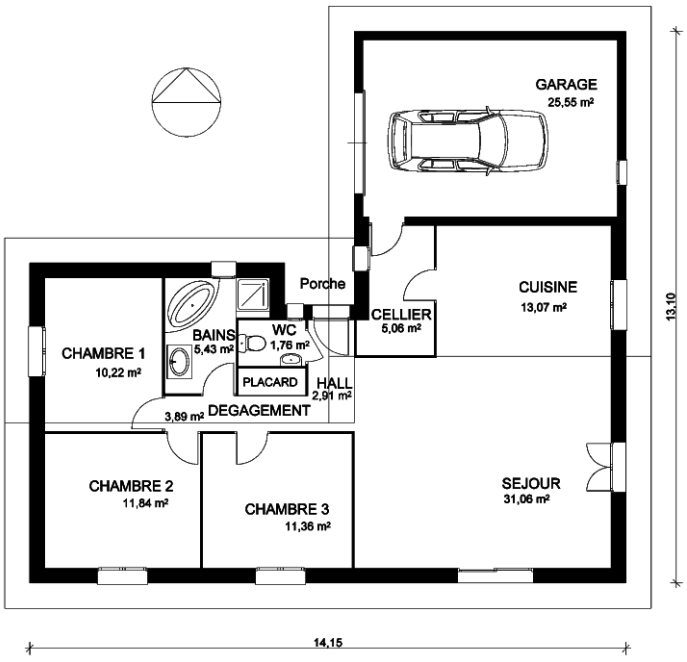


Figure 5.1: Example of artistic detail in a single-unit floor plan. The empty space available in single-unit floor plans allows the designer to add decorative symbols like cars. Section of an image from the CVC-FP dataset [19].

During our angle matrix generation step (particularly Algorithm 3 in Section 3.1.6) we noticed the possibility of using a neural network as an alternative way of finding the line angle, using a descriptor that would contain the slice data $\{p_h, p_v, p_d, p_e\}$ for the pixel and its neighborhood. Further experimentation would reveal if such an approach would be more effective and what trade-offs it brings.

A machine learning method could also improve our geometrical analysis wall extraction sub-module (Section 3.1.8), by helping to filter non-wall line segments using a descriptor with the characteristics of its connections, parallel lines, lines crossed, and other semantic information we gathered in this step.

As discussed in our conclusions, the output of our room detection could be further

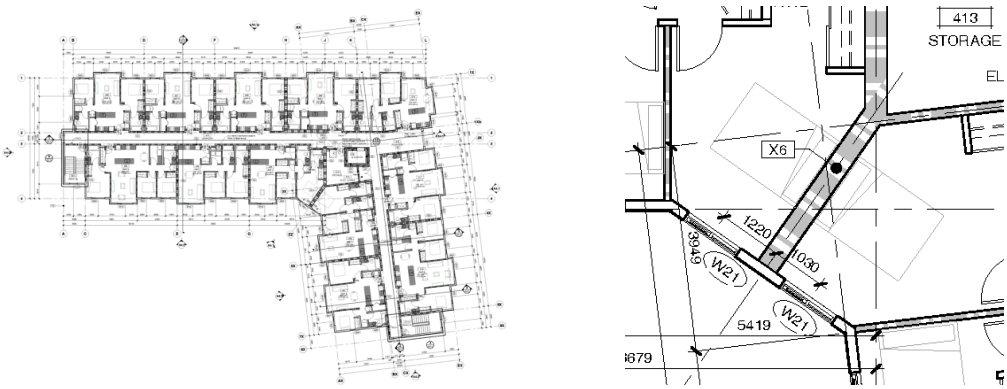
improved with knowledge of the floor plan symbols (doors in particular would improve our quantitative results). The subjective components of the room detection task (as discussed in Section 3.2.2) suggest that the room regions could be better described as fuzzy probability regions instead of polygons.

The experiment in Section 4.5 proved the feasibility of using our algorithms for real-time image processing in mobile devices. This suggest possible applications related to line drawings (e.g. Interpretation of floor plan images on mobile devices and educational mobile applications for children).

A

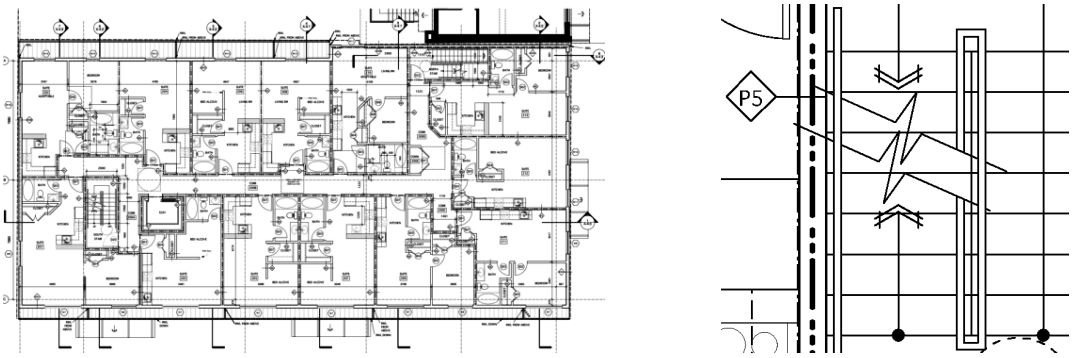
Dataset Image Descriptions

Figures A.1 to A.10 show a down-scaled version of the 10 images selected for our multi-unit floor plan dataset, with a zoomed-in section and a short description of its contribution to the dataset diversity. Except for cropping the external frame, the images have not been modified.



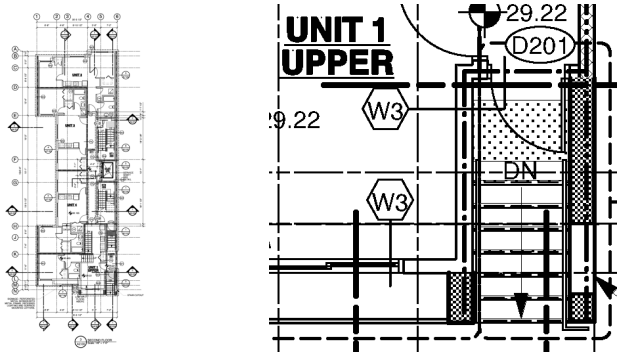
(a) (b)

Figure A.1: Test image 1 (size: 4,162 x 3,095 px). A 12-unit floor plan with diagonal walls filled with gray patterns. (a) Overview. (b) Zoomed section.



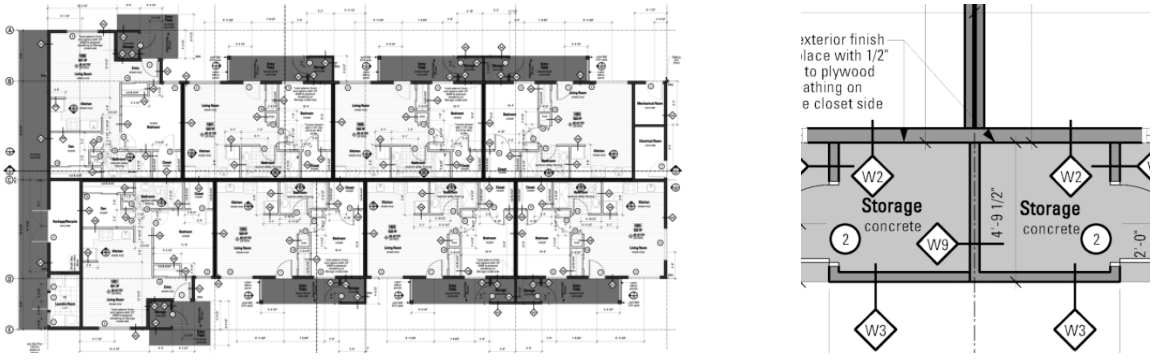
(a) (b)

Figure A.2: Test image 2 (size: 9,427 x 5,445 px). A 13-unit floor plan exported at 300 DPI (big image, every line is thicker) with many overlap cases. (a) Overview. (b) Zoomed section.



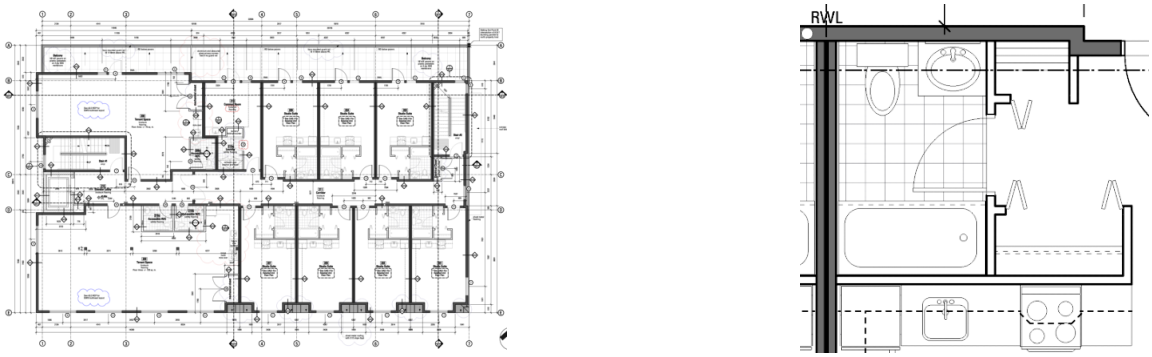
(a) (b)

Figure A.3: Test image 3 (size: 2,618 x 5,709 px). A 4-unit vertical floor plan with heavy overlapping, pattern backgrounds and many staircases. (a) Overview. (b) Zoomed section.



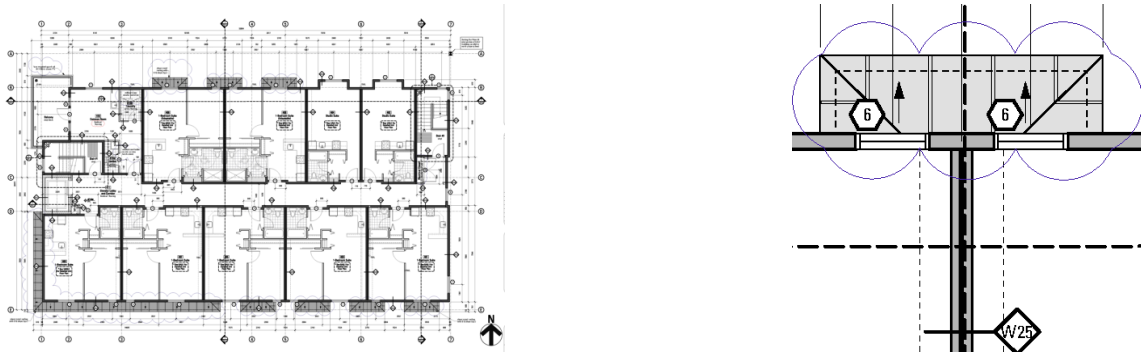
(a) (b)

Figure A.4: Test image 4 (size: 9,779 x 4,961 px). An 8-unit 300 DPI floor plan with walls and regions in shades of gray. (a) Overview. (b) Zoomed section.



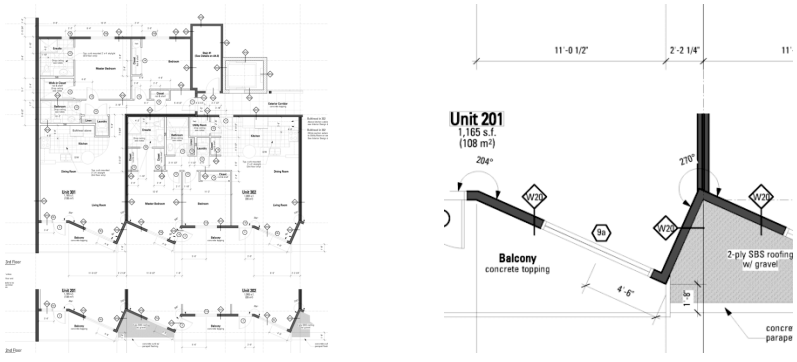
(a) (b)

Figure A.5: Test image 5 (size: 9,053 x 6,171 px). A 10-unit 300 DPI floor plan with gray, rectangular walls and pattern backgrounds. (a) Overview. (b) Zoomed section.



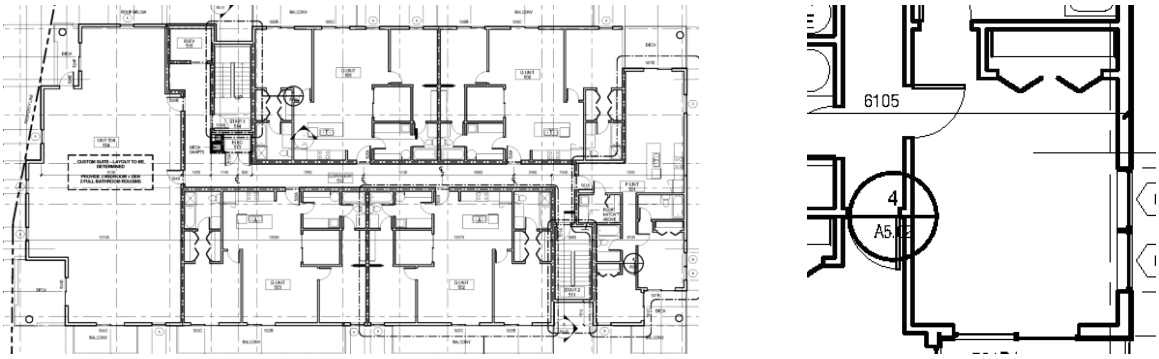
(a) (b)

Figure A.6: Test image 6 (size: 4,774 x 3,212 px). A 10-unit floor plan with the same notation as test image 5, but exported at half the resolution (150 DPI) for thinner lines. (a) Overview. (b) Zoomed section.



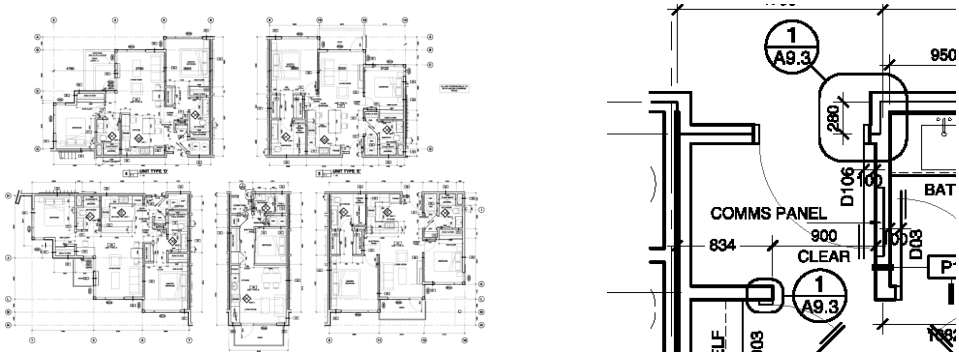
(a) (b)

Figure A.7: Test image 7 (size: 5,544 x 5,995 px). A 4-unit square floor plan with isolated diagonal walls at unexpected angles. (a) Overview. (b) Zoomed section.



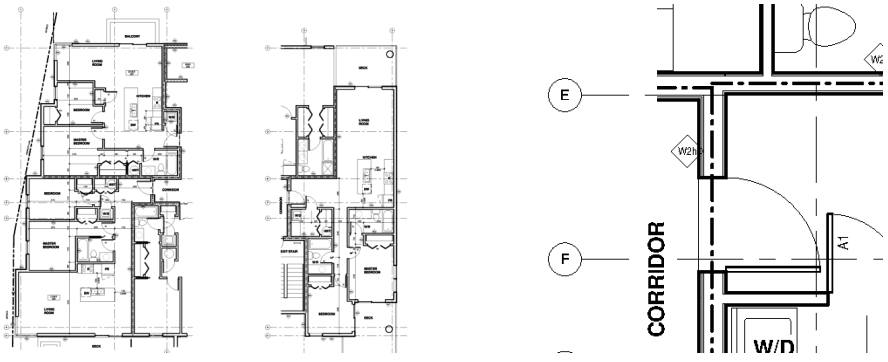
(a) (b)

Figure A.8: Test image 8 (size: 2,679 x 1,338 px). A 4-unit floor plan exported at a low resolution (100 DPI) with different notation for doors and windows. (a) Overview. (b) Zoomed section.



(a) (b)

Figure A.9: Test image 9 (size: 4,603 x 3,201 px). A 5-unit 150 DPI floor plan with split units and incomplete (cut on purpose) walls. (a) Overview. (b) Zoomed section.



(a) (b)

Figure A.10: Test image 10 (size: 3404 x 2756 px). A 3-unit 150 DPI smaller floor plan with split units. (a) Overview. (b) Zoomed section.

B Technical Specifications

The experiments in sections 4.2, 4.3 and 4.4 were compiled and executed in a desktop computer with the following specifications:

CPU	Intel Core i7-4770 3.40GHz
RAM	32 GB DDR3
HDD	256 GB SSD
SO	Elementary OS 0.4.1 built on Ubuntu 16.04.3 LTS
Compiler	Bundled c++, C++ standard version 17

Table B.1: Desktop PC specifications for experiments.

The experiment in Section 4.5 was compiled in the desktop PC previously mentioned, and deployed in a mobile device with the following specifications:

Brand & model	BLU Life One X2
CPU	1.4 GHz Cortex-A53
RAM	4 GB DDR3
Internal Memory	64 GB
SO	Android 6.0.1 (Marshmallow)
Compiler	Android Studio 3.1.3

Table B.2: Mobile device specifications for experiments.

Bibliography

[1] AutoDesk, "AutoDesk AutoCAD." https://www.autodesk.com/products/autocad/overview. [Online; accessed 2018-08-08].

[2] EasyHome, "EasyHome HomeStyler." https://www.homestyler.com. [Online; accessed 2018-08-08].

[3] Trimble, "Trimble SketchUp." https://www.sketchup.com. [Online; accessed 2018-08-08].

[4] J. Harris and S. Withrow, *Vector graphics and illustration : a master class in digital image-making*. Mies, Switzerland : Rotovision, 2008.

[5] D. Greenlee, "Raster and vector processing for scanned linework," *Photogrammetric Engineering and Remote Sensing*, vol. 53, pp. 1383–1387, 01 1987.

[6] K. Wong, "Open design alliance caught in the Autodesk-SolidWorks legal tangle." http://www.digitaleng.news/virtual_desktop/2009/07/open-design-alliance-caught-in-the-autodesk-solidworks-legal-tangle/. [Online; accessed 2018-08-13].

[7] Rakuten Institute of Technology, "R-FP-500 Dataset." https://rit.rakuten.co.jp/data_release. [Online; Accessed: 2018-07-24].

[8] C. Ah-Soon and K. Tombre, "Variations on the analysis of architectural drawings," in *Document Analysis and Recognition, 1997., Proceedings of the Fourth International Conference on*, vol. 1, pp. 347–351, IEEE, 1997.

[9] A. Koutamanis and V. Mitossi, "Automated recognition of architectural drawings," in *[1992] Proceedings. 11th IAPR International Conference on Pattern Recognition*, pp. 660–663, Aug 1992.

[10] Y. Aoki, A. Shio, H. Arai, and K. Odaka, "A prototype system for interpreting hand-sketched floor plans," in *Proceedings of 13th International Conference on Pattern Recognition*, vol. 3, pp. 747–751 vol.3, Aug 1996.

[11] P. Dosch, K. Tombre, C. Ah-Soon, and G. Masini, "A complete system for the analysis of architectural drawings," *International Journal on Document Analysis and Recognition*, vol. 3, no. 2, pp. 102–116, 2000.

[12] S.-h. Or, K.-H. Wong, Y.-k. Yu, M. M.-y. Chang, and H. Kong, "Highly automatic approach to architectural floorplan image understanding & model generation," *Pattern Recognition*, pp. 25–32, 2005.

[13] S. Ahmed, M. Liwicki, M. Weber, and A. Dengel, "Improved automatic analysis of architectural floor plans," in *2011 International Conference on Document Analysis and Recognition*, pp. 864–869, Sept 2011.

[14] L.-P. de las Heras, J. Mas, G. Sánchez, and E. Valveny, "Wall patch-based segmentation in architectural floorplans," in *2011 International Conference on Document Analysis and Recognition*, pp. 1270–1274, Sept 2011.

[15] L.-P. de las Heras, D. Fernández, E. Valveny, J. Lladós, and G. Sánchez, "Unsupervised wall detector in architectural floor plans," in *2013 12th International Conference on Document Analysis and Recognition*, pp. 1245–1249, Aug 2013.

[16] F. Perronnin, "Universal and adapted vocabularies for generic visual categorization," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 30, pp. 1243–1256, jul 2008.

[17] I. T. Jolliffe, "Principal component analysis and factor analysis," in *Principal Component Analysis*, Springer Series in Statistics, pp. 115–128, Springer, New York, NY, 1986.

[18] S. Lloyd, "Least squares quantization in PCM," *IEEE Trans. Inf. Theory*, vol. 28, no. 2, pp. 129–137, 1982.

[19] L.-P. de las Heras, O. Terrades, S. Robles, and G. Sánchez, "CVC-FP and SGT: a new database for structural floor plan analysis and its groundtruthing tool," *International Journal on Document Analysis and Recognition*, 2015.

[20] S. Escalera, A. Fornés, O. Pujol, P. Radeva, G. Sánchez, and J. Lladós, "Blurred shape model for binary and grey-level symbol recognition," *Pattern Recognit. Lett.*, vol. 30, pp. 1424–1433, nov 2009.

[21] C. Cortes and V. Vapnik, "Support-vector networks," *Mach. Learn.*, vol. 20, pp. 273–297, sep 1995.

[22] L.-P. de las Heras, S. Ahmed, M. Liwicki, E. Valveny, and G. Sanchez, "Statistical segmentation and structural recognition for floor plan interpretation," *IJDAR*, vol. 17, pp. 221–237, Sept. 2014.

[23] C. Liu, J. Wu, P. Kohli, and Y. Furukawa, "Raster-to-vector: Revisiting floorplan transformation," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2195–2203, 2017.

[24] A. Bulat and G. Tzimiropoulos, "Human pose estimation via convolutional part heatmap regression," in *Computer Vision – ECCV 2016*, pp. 717–732, Springer International Publishing, 2016.

[25] Y. Kiyota, "Promoting open innovations in real estate tech: Provision of the lifull home's data set and collaborative studies," in *Proceedings of the 2018 ACM on International Conference on Multimedia Retrieval*, ICMR '18, (New York, NY, USA), pp. 6–6, ACM, 2018.

[26] S. Dodge, J. Xu, and B. Stenger, "Parsing floor plan images," in *2017 Fifteenth IAPR International Conference on Machine Vision Applications (MVA)*, pp. 358–361, may 2017.

[27] J. Long, E. Shelhamer, and T. Darrell, "Fully convolutional networks for semantic segmentation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 3431–3440, 2015.

[28] L.-P. de las Heras, J. Mas, G. Sánchez, and E. Valveny, "Notation-Invariant Patch-Based wall detector in architectural floor plans," in *Graphics Recognition. New Trends and Challenges*, Lecture Notes in Computer Science, pp. 79–88, Springer, Berlin, Heidelberg, 2013.

[29] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman, "The Pascal Visual Object Classes (VOC) Challenge," *Int. J. Comput. Vis.*, vol. 88, pp. 303–338, jun 2010.

[30] K. Ryall, S. Shieber, J. Marks, and M. Mazer, "Semi-automatic delineation of regions in floor plans," in *Proceedings of 3rd International Conference on Document Analysis and Recognition*, vol. 2, pp. 964–969 vol.2, Aug. 1995.

[31] A. Rosenfeld and J. L. Pfaltz, "Distance functions on digital pictures," *Pattern Recognit.*, vol. 1, pp. 33–61, jul 1968.

[32] S. Macé, H. Locteau, E. Valveny, and S. Tabbone, "A system to detect rooms in architectural floor plan images," in *Proceedings of the 9th IAPR International Workshop on Document Analysis Systems*, DAS '10, (New York, NY, USA), pp. 167–174, ACM, 2010.

[33] S. Fortune, "Voronoi diagrams and delaunay triangulations," in *Computing in Euclidean Geometry*, vol. 4 of *Lecture Notes Series on Computing*, pp. 225–265, WORLD SCIENTIFIC, jan 1995.

[34] R. Wessel, I. Blümel, and R. Klein, "The room connectivity graph: Shape retrieval in the architectural domain," in *The 16-th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision'2008* (V. Skala, ed.), UNION Agency-Science Press, Feb. 2008.

[35] R. Kasturi, S. Bow, W. El-Masri, J. Shah, J. Gattiker, and U. Mokate, "A system for interpretation of line drawings," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 12, pp. 978–992, Oct. 1990.

[36] S. Ahmed, M. Weber, M. Liwicki, and A. Dengel, "Text/graphics segmentation in architectural floor plans," in *2011 International Conference on Document Analysis and Recognition*, pp. 734–738, Sept 2011.

[37] G. Upton and I. Cook, *Understanding Statistics*. OUP Oxford, 1996.

[38] N. Otsu, "A threshold selection method from gray-level histograms," *IEEE Trans. Syst. Man Cybern.*, vol. 9, no. 1, pp. 62–66, 1979.

[39] J. D. Evans, *Straightforward statistics for the behavioral sciences*. Brooks/Cole, 1996.

[40] C. E. Board, "CGAL, Computational Geometry Algorithms Library." `http://www.cgal.org`. [Online; accessed 2018-08-01].

[41] R. A. Dunlap, *The Golden Ratio and Fibonacci Numbers*. World Scientific, 1997.

[42] Y. Shafranovich, "Common format and MIME type for comma-separated values (CSV) files." `https://tools.ietf.org/html/rfc4180`. [Online; accessed 2018-08-01].

[43] G. J. Carman and L. Welch, "Three-dimensional illusory contours and surfaces," *Nature*, vol. 360, p. 585, dec 1992.

[44] S. V. Burtsev and Y. P. Kuzmin, "An efficient flood-filling algorithm," *Computers and Graphics*, vol. 17, no. 5, pp. 549 – 561, 1993.

[45] S. Suzuki and K. Abe, "Topological structural analysis of digitized binary images by border following," *Computer Vision, Graphics, and Image Processing*, vol. 30, no. 1, pp. 32 – 46, 1985.

[46] C.-H. Teh and R. T. Chin, "On the detection of dominant points on digital curves," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 11, pp. 859–872, Aug 1989.

[47] A. Fitzgibbon, M. Pilu, and R. B. Fisher, "Direct least square fitting of ellipses," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 21, pp. 476–480, may 1999.

[48] J. Bresenham, "Algorithm for computer control of a digital plotter," *IBM Syst. J.*, vol. 4, no. 1, pp. 25–30, 1965.

[49] X. Wu, "An efficient antialiasing technique," in *Proceedings of the 18th annual conference on Computer graphics and interactive techniques*, vol. 25, pp. 143–152, ACM, jul 1991.

[50] DU Group, "DU Recorder." `http://www.duapps.com/product/du-recorder.html`. [Online; accessed 2018-08-08].