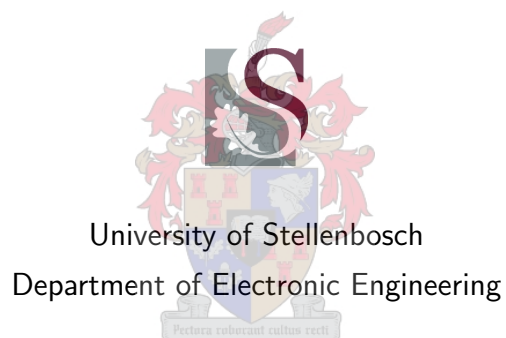


Interactive Recognition of Hand-drawn Circuit Diagrams

Janto F. Dreijer

Thesis presented in partial fulfilment of the requirements for the degree of
Master of Science in Engineering (Electronic Engineering with Computer
Science) at Stellenbosch University



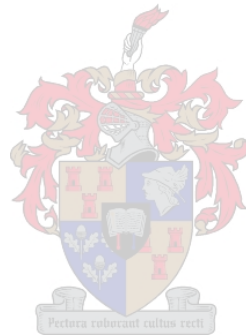
Supervisor: Retief Gerber
Co-supervisor: Thomas Niesler

December 2006

Declaration

I, the undersigned, hereby declare that the work contained in this assignment/thesis is my own original work and that I have not previously in its entirety or in part submitted it at any university for a degree.

Janto Dreijer
November 2006



Abstract

When designing electronic circuits, engineers frequently make hand-drawn sketches of circuits. These are then captured with a computerised design. This study aims to create an alternative to the common schematic capture process through the use of an interactive pen-based interface to the capturing software.

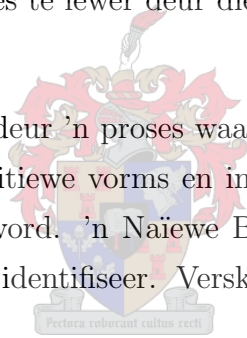
Sketches are interpreted through a process of vectorising the user's strokes into primitive shapes, extracting information on intersections between primitives and using a naive Bayesian classifier to identify symbol components. Various alternative approaches were also considered.

It is concluded that it is feasible to use a pen-based interface and underlying recognition engine to capture circuit diagrams. It is hoped that this would provide an attractive early design environment for the engineer and enhance productivity.

Opsomming

Wanneer ingenieurs elektroniese stroombane ontwerp, word daar dikwels handgetekende sketse gemaak. Die sketse word dan vasgevang in 'n rekenaar vir verdere ontwerp. Hierdie studie beoog om 'n alternatief tot die algemene skematiese vasvangingsproses te lewer deur die gebruik van 'n interaktiewe pen-gebaseerde koppelvlak.

Sketse word geïnterpreteer deur 'n proses waar die gebruiker se penkurwes gevektoreer word na primitiewe vorms en informasie oor die interseksies tussen primitiewe onttrek word. 'n Nuwe Bayesiese klassifiseerder word dan gebruik om simbole te identifiseer. Verskeie alternatiewe benaderings is ook oorweeg.

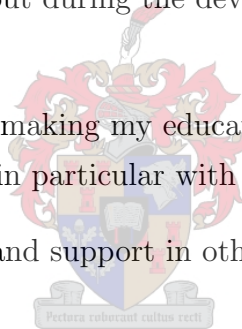


Die gevolgtrekking word dan gemaak dat 'n pen-gebaseerde koppelvlak en onderliggende herkennings algoritmes gebruik kan word om stroombaan-diagramme vas te vang. Dit kan 'n aantreklike omgewing in die vroeë ontwerp stadium vir die ingenieur lewer en produktiwiteit verhoog.

Acknowledgements

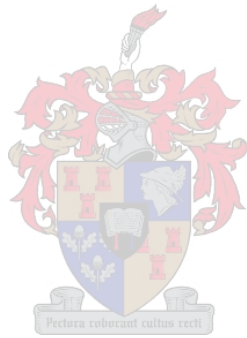
My thanks are expressed to

- My study leader Retief Gerber, Thomas Niesler and fellow students, for their ideas and input during the development of the software and the written text
- My mom and dad for making my education a possibility and helping me finish this report, in particular with proofreading and guidance.
- Carey, for her caring and support in other parts of my life.



Contents

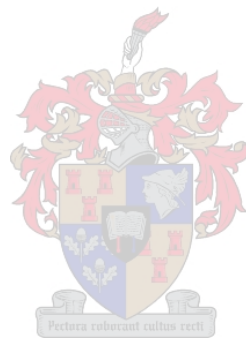
Declaration	i
Abstract	ii
Opsomming	iii
Acknowledgements	iv
Contents	v
List of Abbreviations	ix
List of Figures	x
List of Tables	xii
1 Introduction	1
1.1 Problem Description	1
1.2 Scope of Solution	3
1.3 Study Overview	5



2	Related Research	6
2.1	Intersection Features	6
2.2	Glyph Language	8
2.3	Stroke Order Analysis	10
2.4	Graph Matching	10
2.5	Moments	12
2.6	Fluid Sketches	13
2.7	Research Summary	14
3	Prototype Development	16
3.1	System Requirements	16
3.1.1	Stroke Interpretation	17
3.1.2	Primitive Interpretation	17
3.1.3	Recognition Stages	18
3.2	Design Overview	18
3.3	Stroke Capture	19
3.4	Primitive Identification	20
3.4.1	Opt-out parameters	24
3.5	Primitive Clustering	25
3.6	Symbol Recognition	27
3.6.1	Primitive Intersections	28
3.6.2	Symbol Classifiers	29

3.6.3	Training Data	32
3.7	Symbol Properties	34
3.7.1	Primitive Mapping	34
3.7.2	Symbol Rotation and Translation	35
3.8	Interface	37
3.9	Design Summary	40
3.10	Implementation	40
3.10.1	Tools	40
3.10.2	Implementation Methodology	42
3.10.3	Data Formats	43
4	Evaluation	46
4.1	Recognition Accuracy	46
4.1.1	Primitive Identification Accuracy	47
4.1.2	Symbol Recognition Accuracy	48
4.2	Performance	51
4.3	Usability Analysis	51
5	Recommendations and Conclusions	53
5.1	Future Work	53
5.1.1	Algorithm Enhancements	53
5.1.2	Interface Enhancements	55
5.2	Suggested Applications	56

<i>CONTENTS</i>	viii
5.3 Conclusions	56
Bibliography	58
A Classifier Tutorial	63
A.1 Linear classifiers	63
A.2 Gaussian distributions	65
A.3 Dimensional variance	66
B Intersection Analysis	67
C Code Structure	69



List of Abbreviations

Abbreviation	Description	Definition
CAD	Computer Aided Design	page 1
HMM	Hidden Markov Model	page 10
PDF	Probability Density Function	page 22
SPICE	Simulation Program with Integrated Circuits	page 1
	Emphasis	
XML	Extensible Markup Language	page 41



List of Figures

1.1	Kicad schematic editor	2
1.2	Tablet PC	4
2.1	Gennari interface	7
2.2	LADDER example	9
2.3	Semantic network of square	11
2.4	Fluid sketch	13
3.1	Recognition steps example	18
3.2	Data flow diagram	19
3.3	Primitives	20
3.4	Primitive features histogram	23
3.5	Clustering example.	26
3.6	Symbol definitions.	27
3.7	Diode intersections	29
3.8	Histograms of selected features for <i>box</i> , <i>diode</i> and <i>bjt</i>	31

3.9	Box intersections	32
3.10	Symbol mutation examples	33
3.11	Training a symbol classifier from mutations	34
3.12	Interface example	38
3.13	Interface state machine	39
4.1	Symbols easily confused.	50
A.1	Measurable features of Apples and Oranges	64
A.2	Euclidean and Gaussian distance metrics	65
A.3	Measurable features of Apples and Oranges	66
B.1	Two line primitives intersecting.	67
C.1	Diagram of symbol related classes.	69
C.2	Diagram of primitive related classes.	70
C.3	Directory tree	71



List of Tables

3.1	Intersection types.	28
4.1	Primitive example set size	47
4.2	Primitive confusion matrix	47
4.3	Symbol example set size	48
4.4	Symbol example set classification accuracy	49
4.5	Symbol confusion matrix	50
C.1	Source Files	72



Chapter 1

Introduction

1.1 Problem Description

Circuit diagrams are used to design electronic modules or systems. When designing an electronic circuit, an engineer usually makes a rough initial drawing. This is regularly done on paper and follow various phases of re-drawing. After many versions, when he/she is satisfied with the design, the schematic is captured on a computer using computer aided design (CAD) software¹. From here the circuit can be further edited, refined, integrated, simulated with systems such as SPICE², tested and finally used to generate a physical layout (floor plan) used in the manufacturing process.

Capturing a circuit diagram for use in CAD software can be done with a

¹Examples include products such as Cadence's OrCAD Capture, Altera MaxPlus and the open-source Kicad.

²From <http://bwrc.eecs.berkeley.edu/Classes/IcBook/SPICE>: "SPICE is a general-purpose circuit simulation program for non-linear DC, non-linear transient, and linear AC analyses. Circuits may contain resistors, capacitors, inductors, mutual inductors, independent voltage and current sources, four types of dependent sources, transmission lines, and the four most common semiconductor devices: diodes, BJTs, JFETs, and MOSFETs."

schematic editor (Fig. 1.1). Inserting components using a schematic editor usually involves clicking an icon representing a component or browsing and selecting one from a large library and then clicking the desired placement on a canvas.

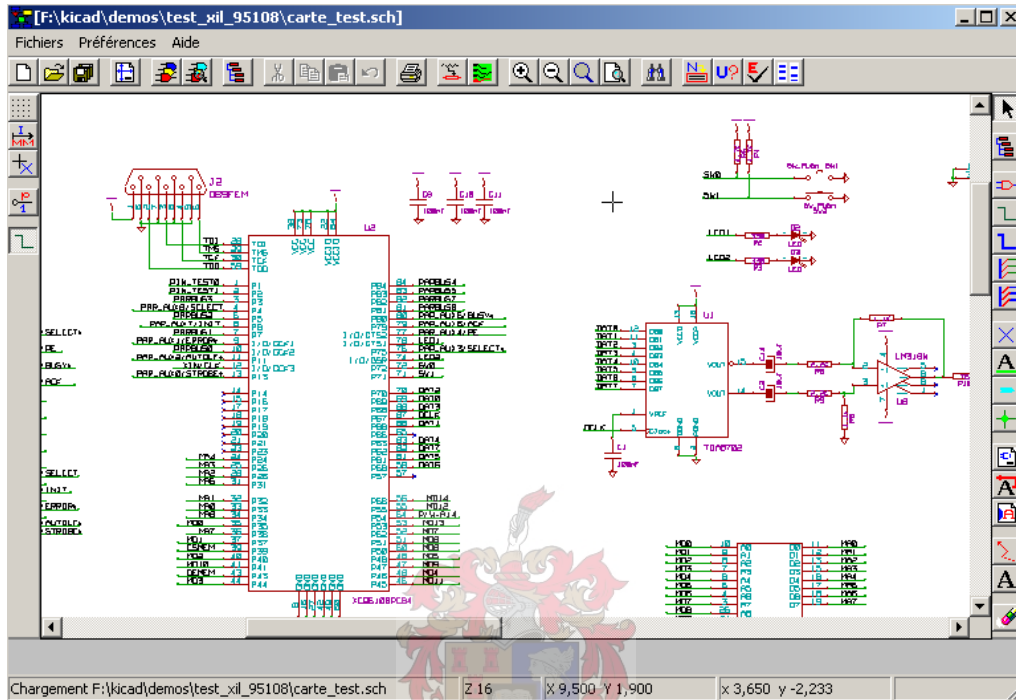


Figure 1.1: Kicad schematic editor.³

Modern CAD software already offers the designer several advantages if used from the earliest design stage [1]. Advantages include fast prototyping, auto-generation of subcircuits and avoiding errors in the capturing stage.

The question of why designers still use pen and paper instead of a computer as the first step in the design process can be raised. Many engineers avoid the use of a schematic editor for non-trivial circuits until the design has become more concrete. Reasons often cited are

- that the designer was away from their desktop computer

³http://www.lis.inpg.fr/realise_au_lis/kicad

- frustration with the speed and intuitiveness of inserting items from the CAD software's component library
- forming connecting wires between components using the software is usually a frustrating experience and not conducive to the creative process

Modern portable technology, however, creates the opportunity for a friendly, familiar and interactive design environment. By leveraging existing electronic input devices it should be possible to create a system where the user is able to select components from a library by simply drawing the corresponding symbol. This would, hopefully, alleviate some of the problems of using a computer in the early design stage, as mentioned above.

1.2 Scope of Solution

A tablet PC (Fig. 1.2) is a fully-functional laptop, but also provides the option of using a stylus or digital pen instead of a keyboard or mouse. In 2002, with the introduction of Microsoft's Windows XP Tablet PC Edition, tablet PCs have entered mainstream usage. At the time of writing, most tablet PCs run Windows XP Tablet PC Edition, which features advanced handwriting recognition technology, but does not include any tools for graphics recognition.

This study aims to create an alternative to the "click-and-drop" capture process through the use of a schematic recognition algorithm and a tablet-based capturing interface. It is hoped that using a portable device, such as a tablet PC, will provide designers with the tools needed to easily sketch circuit diagrams when away from their desktops, and leverage the benefits of using a computer in the early design stage [1, 2]. The extra mobility afforded by electronic tablets should also be a welcome advantage; allowing the designer easier face-to-face collaboration with other designers.



Figure 1.2: Tablet PC

This project also aims to increase productivity in the early design process by enabling a designer to sketch architectures (using a mouse or pen-based electronic tablet) which are then automatically converted into SPICE. This alternative to the current click-and-drop methodology of CAD should feel more familiar to the designer.

Because the Tablet PC does not currently include sketch recognition capabilities, a system that interprets the user's drawing as a circuit diagram needs to be developed. Existing recognition systems can be categorised as ei-

ther off-line or on-line. Off-line recognition involves scanning in a schematic or text that was drawn on paper, and a computer interpreting the resulting raster image. Optical character recognition or OCR is a commonly known example of off-line recognition. On-line recognisers have additional temporal information, as strokes are directly captured as they are drawn. This provides extra accuracy and allows a large degree of interaction between the system and the user.

It should also be possible to use the Tablet PC's handwriting recognition capabilities to add character recognition to this interface.

1.3 Study Overview

The goal of this study is a proof of concept: to prove the viability of using a tablet for schematic capture. It should be possible to use a pen-based interface and underlying recognition engine to capture circuit diagrams and thus provide an attractive early design environment for the engineer.

With this goal in mind it is necessary to analyse work related to on-line schematic recognition and develop and analyse a prototype of a recognition engine. This study will briefly discuss the related algorithms in Chapter 2. Chapter 3 will then detail the design and implementation of the engine. The system is evaluated in Chapter 4 by examining its accuracy and usability. Finally recommendations on possible improvements on and applications for the developed system is discussed in Chapter 5.

Chapter 2

Related Research

Various studies related to on-line drawing recognition have been undertaken by the academia for recognition of flow diagrams [3], UML diagrams [4], Chinese [5] and Japanese characters, musical notation [6] and simple circuit diagrams [7]. These systems provide a basis for this study.

A few major on-line strategies related to the goal of recognising the parts of a circuit diagram were considered: matching intersection features, matching against a linguistic description of a symbol, recognition based on stroke ordering, graph matching, moments as features and finally fluid sketches as a way to recognise primitive shapes. These strategies are discussed and evaluated for applicability in the rest of this chapter.

2.1 Intersection Features

Kara [8] has recently (2004) released his Ph.D. dissertation on automatic parsing and recognition of hand-drawn sketches for pen-based interfaces. He discusses three different strategies for schematic sketch capture. Also detailed is VibroSketch: a sketch-based interface for vibratory symbols and

SimuSketch [3]: a sketch-based interface to Matlab's Simulink.

Together with Kara, Gennari *et al.* [9] developed AC-SPARC, a sketch-based interface for an electronic analysis program. A screen capture of AC-SPARC can be seen in Fig. 2.1. Their technique involves segmenting the user's pen strokes, identifying line and arc primitives, and finally clustering together and recognition of primitives as symbols. Recognition involves pruning the list of candidates based on domain knowledge and classification using a naive Bayesian classifier. Nine geometric properties (number of lines, number of arcs, number of endpoint "L" intersections, point-to-midpoint "T" intersections, midpoint "X" intersections, etc.) are extracted from symbol examples and used to train the classifier. Error correction is then attempted through a rule-based process.

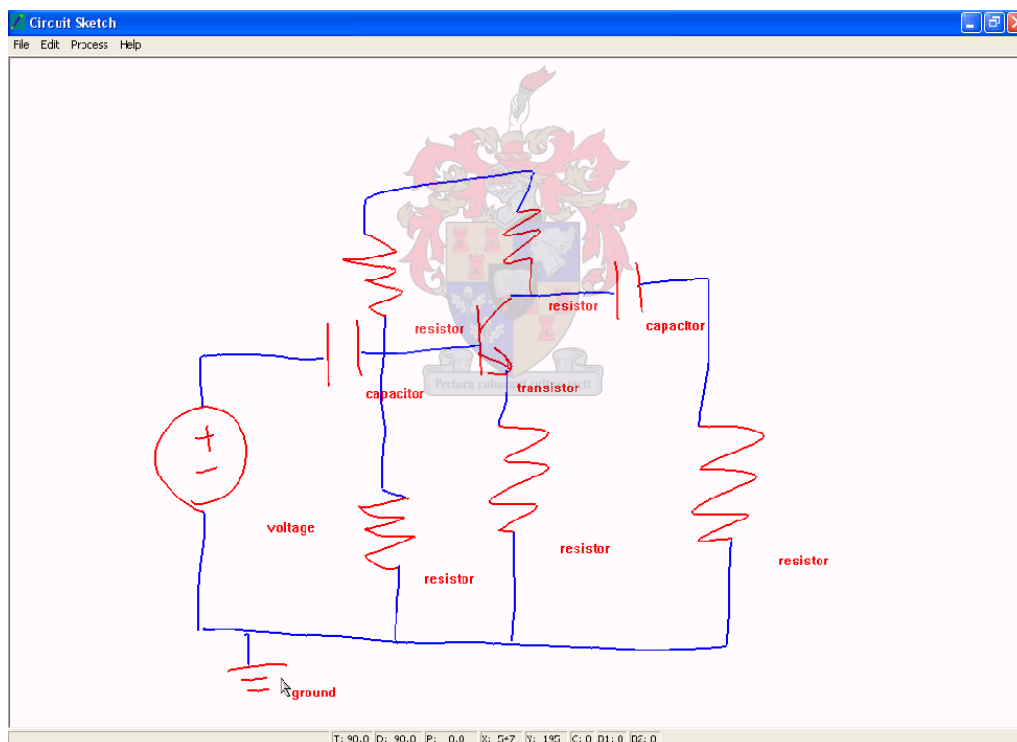


Figure 2.1: Gennari: Interpreted sketch with colour coding and text labels to indicate the identified symbols.

In one of their experiments, six examples for each of 7 symbols were provided by each member of a user group. One of the authors trained the system by

providing 10 examples of each symbol. An average recognition accuracy of 75% across the user examples was reported.

In this author's opinion, requiring a user to train a classifier with a variety of examples for each symbol can be a frustrating experience. It would be preferable to reduce the number of required training examples to as few as one.

The authors of AC-SPARC claim to be able to generate SPICE, however for this to be possible a mapping of component ports is needed. No such algorithm was presented and it is questioned whether using the stated method alone is sufficient to directly derive more than two port mappings per symbol.

2.2 Glyph Language

Alvarado and Hammond [10, 11, 12] uses a Bayes net generated dynamically from a hierarchical description of shapes, in a language called LADDER. This describes a symbol according to a set of hypotheses based on the geometric relationship between the constituent primitives.

Primitive shapes are first identified from a stroke by comparing the least square error between a shape and the pixels of the drawn stroke. Similarity thresholds were determined through manual experimentation and taking measurements from collected data.

Fig. 2.2 shows examples of a few symbols, their primitives and linguistic description of the relationships. The symbol in the first column is specified as a collection of subshapes as shown in the third column. The relative geometric properties between these subshapes are also manually derived and is as shown in the fourth column. These properties can then be used to identify similar symbols.

To address the noise and ambiguities in the drawn schematic the LADDER

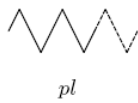
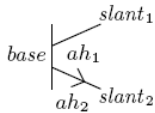
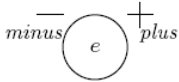
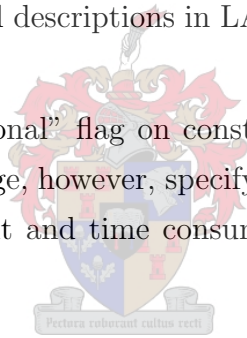
	Resistor	Polyline <i>pl</i>	<code>acuteAngle pl.l[i] pl.l[i + 1]</code> <code>equalLength pl.l[i + 1]</code> <code>pl.l[i + 2]</code>
	BJT	Line <i>base</i> Line <i>slant1</i> Line <i>slant2</i> Line <i>ah1</i> Line <i>ah2</i>	<code>touches slant1.p1 base</code> <code>touches slant2.p1 base</code> <code>coincident ah1.p1 ah2.p1</code> <code>touches ah1.p1 slant2</code> <code>touches ah2.p1 slant2</code> <code>equalLength ah1 ah2</code>
	Voltage-Source	Ellipse <i>e</i> Line <i>minus</i> Plus <i>plus</i>	<code>nextTo plus e</code> <code>nextTo minus e</code>

Figure 2.2: Example symbol descriptions in LADDER. From [11] page 115.

language contains an “optional” flag on constraints and primitives. Even with a well designed language, however, specifying good grammatical shape descriptions may be difficult and time consuming and is something of an art [11, 13].



A hardcoded glyph language would therefore be more suitable when trying to recognise a small, non-expanding symbol set. Other members of their group (particularly Veselova [14, 15]) have developed a system that is able to learn shape descriptions based on few examples. This is done by generating a textual language representation based on properties inferred from human perceptual bias studies.

The author is of the opinion that although the system is able to recognise partially drawn shapes, applicability is questionable when considering the complexity and performance [10] of the algorithm.

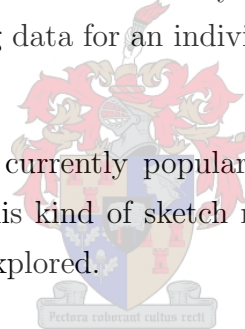
2.3 Stroke Order Analysis

Hidden Markov Models have been extensively used in character recognition.

HMM-based methods have also been investigated for sketch recognition by Sezgin and Davis [16]. Their studies suggest that stroke orderings may be different for each individual, but persist across sketches for each individual. It, however, assumes that stroke ordering for an individual will also persist across multiple sittings. Unfortunately in more complex diagrams (such as circuit diagrams), different orientations of symbols will effect the individual's stroke ordering.

Stroke order analysis is thus more a recognition of the user's drawing style than recognition of the structure of the symbol. It would therefore be necessary to collect training data for an individual over a period of time as opposed to only one sitting.

HMM strategies, although currently popular in handwriting recognition, is thus not applicable to this kind of sketch recognition system and other strategies also have to be explored.



2.4 Graph Matching

Other strategies involve constructing an attributed relational graph of a symbol's geometric properties. This has been done for fingerprint identification [17]. Generally nodes represent the primitives in a symbol and edges in the graph relate the geometric/structural relationship between these primitives (see Fig. 2.3). Unknown symbols are then matched to symbol definitions by measuring the similarity between graphs.

Calhoun *et al.* [18] compares graphs through a process of direct comparison of nodes and edges and computing an error function. The choice of which

primitives are compared depend on the order in which they were drawn. This assumption was relaxed by generating all possible assignments for the first primitive in each definition, allowing more variation in stroke order.

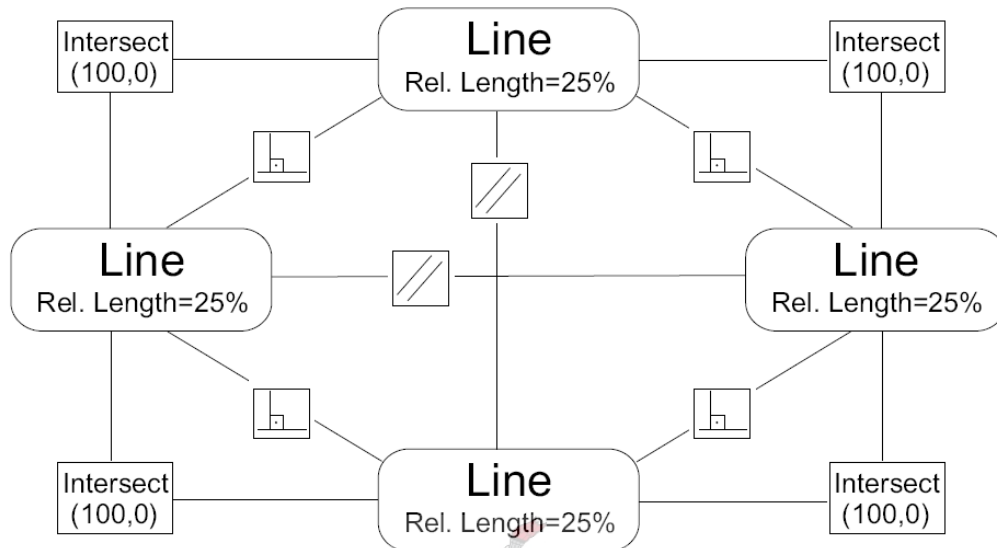


Figure 2.3: The semantic network definition of a square. The edges represent parallel and perpendicular relationships and intersections. From [18].

Line intersection points can be extracted as was done by Gennari *et al.* [9] and then grouped together into nodes according to the distance between these intersections. Graph comparisons or dynamic programming techniques could then be applied to identify symbols.

A planar graph representation could also be constructed from a drawn symbol with intersections between primitives represented by the graph's nodes. Edges between the nodes represent primitive segments between intersections. Matching of symbols would then be equivalent to matching their planar graphs.

The computational complexity of the graph isomorphism problem (determining a permutation of the vertices of one graph so that it is exactly equal to the other) has not yet been classified within a particular type of complexity and remains an open theoretical problem [19]. Fortunately there exist

efficient linear time solutions for some special classes such as planar graphs [20] by partitioning the graph into pieces of small tree-width, and applying dynamic programming within each piece.

Inexact graph matching (when an isomorph does not exist, such as when the number of edges differ), however, is known to be NP-complete. Due to its combinatorial nature inexact graph matching is a computationally expensive process and should be approximated if to be practical. See Bengoetxea [19] and Brooks [21] for an overview.

For this reason graph matching will not be used to compare an unknown symbol to a library of symbols for identification. It will, however, be used to extract certain properties, such as orientation, after a symbol is identified. This will be discussed in more detail later.

2.5 Moments

Moments have been used primarily in the recognition of Chinese characters [22, 23] but also for interpreting hand-drawn musical notation [6] and symbols such as polygons and arc shapes [24].

Liao and Lu [22] combined properties from Legendre moments to form a vector in multidimensional space to represent a Chinese character. Using the root mean square distance between vectors as a difference measure between characters, they noted significant improvements in terms of optical character recognition of Chinese. This is especially true for characters close in shape.

Hse and Newton [24] presented an on-line recognition method for hand-sketched symbols using Zernike moments. Zernike moments are invariant to rotation, scaling and translation of symbols. Using a support vector machine they report a high accuracy for simple polygonal shapes.

Ong and Lee [25] introduced a new set of orthogonal moment functions based on the discrete Tchebichef polynomials. Since the basis set is orthog-

onal in the discrete domain and the method does not involve any numerical approximations, it is superior to the conventional moments such as Legendre and Zernike in terms of preserving the analytical properties needed to ensure information redundancy in a moment set.

Moments, however, will not be used in this study. While moments seem to hold promise for efficient identification, it is based on analysing all of a symbol's pixels and representing it as a single set of features. It does not leverage any information on what primitive shapes were used to construct a symbol. Because the system being designed will have knowledge of the sequence and time that pixels are drawn, information on the primitive shapes can be extracted. For this reason a different path that better leverages the advantages of interactivity between the user and system, was followed.

2.6 Fluid Sketches

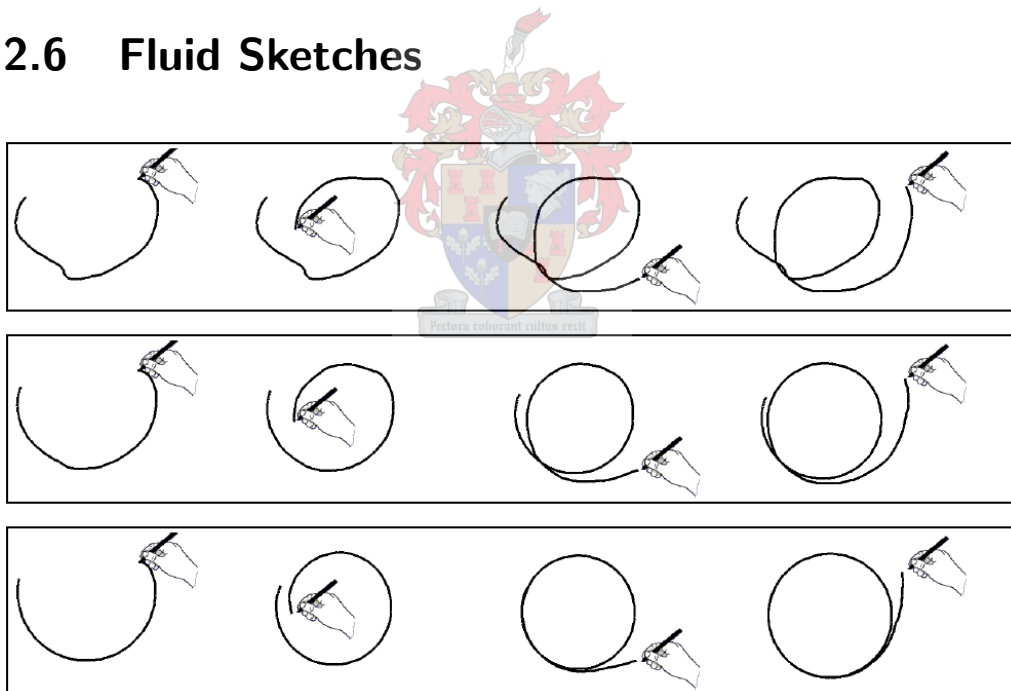


Figure 2.4: Fluid sketch

Fig. 2.4 shows an example of the behaviour of a prototype developed by Arvo and Novins [26] as the user interacts with their system. In the top

row a roughly circular pattern is drawn by the user. With fluid sketching enabled (middle row), the very same trajectory continuously morphs toward the least-squares circle. When the viscosity is decreased (bottom row), the morphing reaches the current optimal shape more quickly.

Their system morphs the user's strokes according to an ordinary differential equation to fit a predicted shape. First the optimal shape from each class is found. The optimal class is then selected as the prediction by comparing the least-squares errors between these shapes. This is relatively easy for classes such as circles and polygons, however, for more complex shapes such as spirals, determining a least-squares error is non-trivial. Also, finding a drawn point's corresponding position on the predicted shape becomes much more difficult.

The interactivity afforded by this system is, however, very pleasing [26] and attempts are made in this study to duplicate a similar effect using a different method of primitive identification.

2.7 Research Summary

The majority of strategies for on-line schematic recognition involve segmenting strokes according to curvature and temporal information, approximating these segments with primitive shapes (lines, arcs, etc.), clustering primitives together into symbols and then comparing the unknown symbols to stored symbol definitions.

Due to possible variance in stroke ordering, HMMs were not further explored. Graph matching is, however, used in this study to determine primitive mappings, which is necessary for calculating symbol rotations and port mappings. While moments could possibly be used, a different path that better leverages the advantages of interactivity between the user and system, was followed.

The recognition algorithm developed in this study is a refinement on work done largely by Gennari (intersection features) and Novins *et al.* (fluid sketches), and is closely related to work done by Alvarado (glyph language). The design of the system will be discussed in Chapter 3.



Chapter 3

Prototype Development

This chapter will detail the design of a system conforming to the stated requirements.

3.1 System Requirements

NioCAD is an electronic CAD environment being developed by Retief Gerber at Stellenbosch University. It was decided to design a prototype system that would be able to interpret a circuit, sketched on a pen-based device, as well as possibly serving as an interface layer on top of NioCAD. With the vision of eventually integrating this recognition engine with NioCAD, some requirements had to be met:

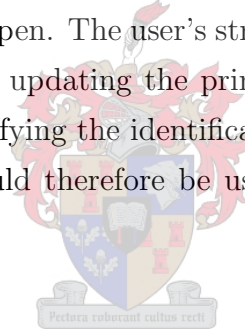
- It was decided that the system must be able to differentiate between at least ten different symbols (listed in Section 3.6).
- The system should be able to learn from only one example of each component. This would allow users to easily define custom component symbols without having to provide a large number of examples.

- Due to its interactive nature, a fast recognition algorithm is vital to the system's usability.

3.1.1 Stroke Interpretation

Many existing on-line recognition systems (see Chapter 2) do not provide user feedback on strokes until a recognition command is issued, after which the user is typically required to correct mistakes made by the engine. By continuously supplying feedback to the user on the recognition of primitive shapes, the user's response could be used for implicit verification. This would lessen the need for this error-correcting stage.

This feedback is made possible by continually updating the identified primitive as the stroke is being made. The decision on the primitive is made final when the user lifts the pen. The user's strokes are then replaced by the primitive. By continuously updating the primitive as the stroke is made, the system is implicitly verifying the identification of the primitive. A simpler primitive identifier could therefore be used than would otherwise be required.



3.1.2 Primitive Interpretation

It is assumed that only one symbol is drawn at a time. After the primitives needed to construct a symbol have been drawn, the user is required to issue a "recognise component" command. This action is triggered from a drop down menu or by pressing a button. This command will group together the primitives and pass it to a second stage where the symbol is matched against definitions stored in a database.

An electronic component has nodes which can be connected to other component nodes (generally called "ports"). Connections between these ports are made by drawing a stroke starting at the one port and ending at the other.

3.1.3 Recognition Stages

Fig. 3.1 pictures a simplified example of the steps such a recogniser should follow. The system is required to convert a user's pen stroke into one of a few primitives (e.g., lines, arcs and circles) as it is being drawn and displayed on the screen. These primitives should then be clustered together, and identified as a specific symbol representing an electronic component.

These recognised symbols are then analysed for rotation and port mapping information. Two symbol ports can be connected together by simply drawing a stroke starting at the one port and ending at the other.

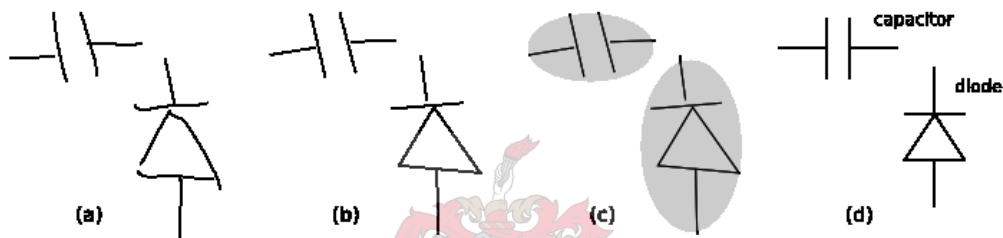


Figure 3.1: Simplified example of recognition steps: (a) Stroke capture, (b) primitive identification, (c) primitive clustering and (d) symbol recognition.

3.2 Design Overview

A simplified data flow diagram for the proposed design is given in Fig. 3.2.

The recognition algorithm relies on a few basic assumptions:

- A symbol is constructed from a set of drawn primitives.
- A single primitive is represented by a single pen stroke.
- One symbol is drawn at a time and completed before the next one is started.

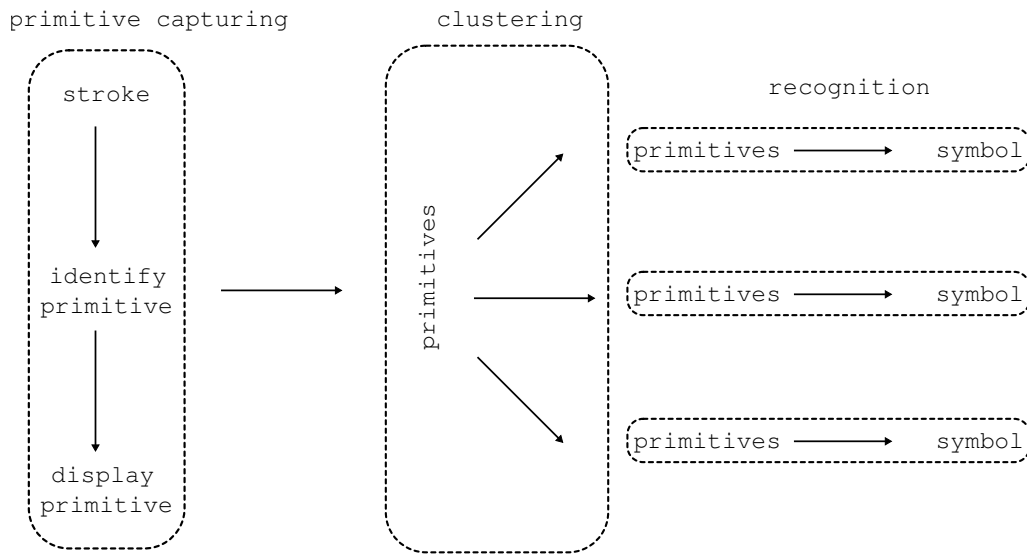


Figure 3.2: Simplified data flow diagram.

- Symbols are drawn and recognised before connections between their ports are made.

These restrictions are seen as acceptable as they appear to superficially correspond to the typical user's drawing style. Some of these restrictions are, however, relaxed significantly through the use of the Corner primitive and clustering discussed in Section 3.5.

3.3 Stroke Capture

A stroke is represented as a collection of points and corresponding time information. More specifically, the output of this capturing stage is a list of sampled points (x -coordinate, y -coordinate and time) which is passed on to the primitive identification stage.

Simply capturing information from “mouse-move” events does not, however, deliver consistent results over different platforms (such as desktop systems and Tablet-like input devices). This is due to the temporal resolution being

inconsistent on some platforms. Like Calhoun [18], a simple mouse position sampler that queries the mouse's position over fixed intervals, was written to compensate for these differences. Running the sampler at 50Hz was found to provide sufficient information to the primitive identification stage.

3.4 Primitive Identification

As a stroke is made, the system attempts to identify the primitive being drawn. A selection is made from a set of primitive types (Line, Jagged, Corner, Crescent, Spiral or Circle) built from the points collected from the stroke thus far, and drawn to the screen. The various primitive types are shown in Fig. 3.3. The selected primitive is recalculated and redrawn with each additional point collected. When the pen is lifted the selection is finalised.

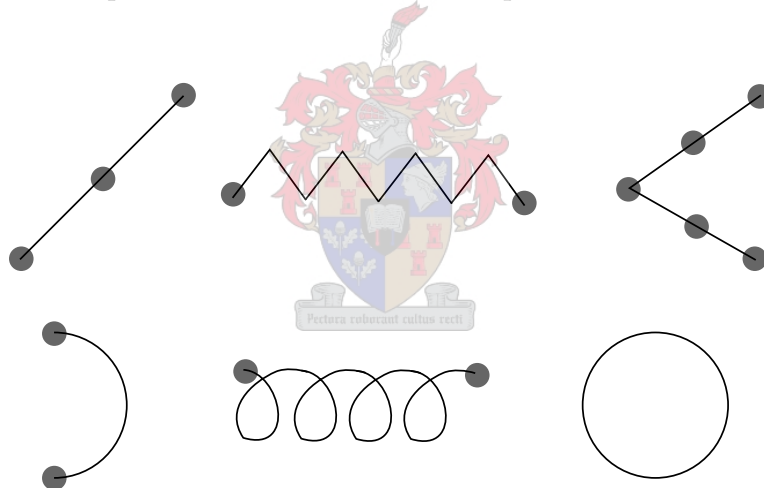


Figure 3.3: The various primitives and their snap points. Line, Jagged, Corner, Crescent, Spiral and Circle.

This method is similar to the one employed by Arvo and Novins (see Section 2.6). However, instead of directly comparing the least squares errors of the primitives, a different primitive identification strategy is used.

In this study a vector of features is extracted from the points collected from a stroke. This feature vector is compared against those of a training library

of stored primitives.

From a stroke's ordered set of points ($\{p_1..p_N\}$), the following properties are derived for use as feature vector dimensions. These features (except for c and d) are also normalised against the length of a line estimate and then upper bounded to the maximum value indicated.

- a) average deviation of points from line estimate. Maximum 2 units.

$$f_a(\{p_1..p_N\}) = \sum_{n=2}^{N-1} \frac{\text{distance}(p_n, \text{line_estimate}(p_1, p_N))}{N-2}$$

- b) average deviation of points from two-line (corner) estimate. Maximum 0.5 units.

$$f_b(\{p_1..p_N\}) = \min(\{C_1..C_N\})$$

where

$$C_x = \frac{f_a(\{p_1..p_x\}) + f_a(\{p_x..p_N\})}{2}$$

- c) average distance from radius of least-square's estimated circle, normalised against radius. Maximum 0.5 units.

$$f_c(\{p_1..p_N\}) = \sum_{n=1}^N \frac{(|\|p_n - c\| - r|)/r}{N}$$

- d) average of angular changes between points (angular changes below $\frac{\pi}{16}$ are ignored).

$$f_d(\{p_1..p_N\}) = \sum_{n=2}^N \frac{(\text{angle}(p_n - p_{n-1}) - \text{angle}(p_{n-1} - p_{n-2})) \bmod \pi}{N-1}$$

- e) total stroke curvature length. Maximum 3 units.

$$f_e(\{p_1..p_N\}) = \sum_{n=2}^N \|p_n - p_{n-1}\|$$

To populate the library, more than 100 examples of each primitive type were collected from five people. The question as to whether this is an

adequate sample to recognise more diverse drawing styles, can be raised. This question is not addressed in this study other than the results provided in Chapter 4. It was assumed that this will provide sufficient information to identify input from any other users.

Each primitive type from the training library is represented by a diagonal¹ Gaussian probability density function (PDF). Fig. 3.4 shows histograms of selected feature distributions for some example primitives. A naive Bayesian² classifier is then used to find a match between the drawn stroke and primitive type.

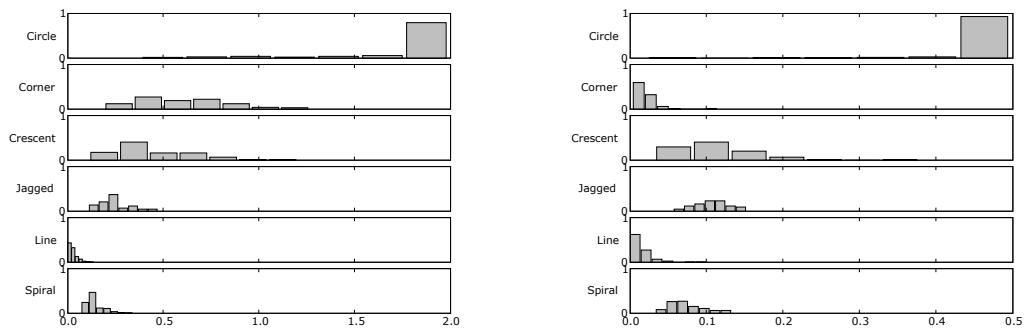
This recognition process can sometimes yield awkward results such as a circle with an abnormally large radius when a line is expected. The constructed primitives therefore also have the ability to “opt-out” of this selection process when it detects strange properties about itself. These properties are listed in Section 3.4.1. Primitives also opt-out if their curvature length differs from that of the drawn stroke beyond a certain margin. If all primitives opt-out, the system defaults to a Line primitive.

Drawing multiple primitives with a single stroke is supported to a limited degree through the use of a “corner” pseudo-primitive. Identification of a corner element is done the same as identification of other primitives, but afterwards is split into two line primitives to form a *V* or *L* shape. The deviation between a stroke and a corner primitive is calculated as the average deviation from the stroke and two line segments. The decision on where to segment the stroke is made by considering all points in the stroke and selecting the one that creates the smallest deviation. This is currently done using a linear walk through the points, but can be improved by employing a binary search.

By converting user strokes into primitives as they are drawn, immediate feedback on the quality of segmentation is given to the user who is then able to correct it. This approach has both advantages and disadvantages.

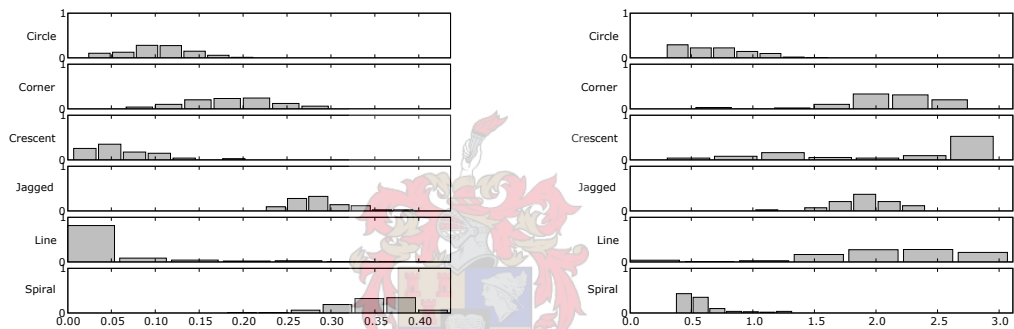
¹A single mean and variance for each dimension.

²See Appendix A for an introductory tutorial.



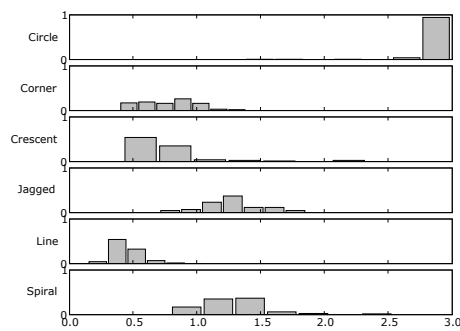
(a) deviation of points from line estimate

(b) deviation of points from corner estimate



(c) distance from radius of estimated circle

(d) angular changes between points



(e) stroke curvature length

Figure 3.4: Histograms of features extracted from primitive examples.

It is hoped this will provide a high quality input to the recognition stage and therefore a simpler classifier, but it also requires more user interaction. This also pushes most of the variability to primitive recognition stage.

Fig. 3.3 also shows the snapping points associated with the primitive types. “Snapping points” are locations on primitives that are more likely to have another primitive starting or stopping at that point than any other location in the vicinity. The start and end of a user’s stroke can thus be modified to fall exactly on these points if close to one of these points. More specifically, a primitive is snapped to one of the points if its snapping points are within a certain range of another snapping point. Snapping is important to differentiate between symbols that can look the same if drawn roughly, such as the *diode* and *box*.

Snapping is not done on strokes with short curvature lengths as this would transform a high percentage of the stroke points and thus interfere with the primitive recognition. Note also that the circle does not have any snap points as no point on the circumference is more likely to have a primitive starting or stopping on them than any other point on the circumference. It could be argued that any point on the circumference (as well as the center point) is more likely than any other location, outside or inside the circle, to have a starting or ending primitive. This is, however, not true for the circuit component symbols being considered and was found to be more of an irritation than a help when drawing symbols such as the dc-source.

3.4.1 Opt-out parameters

This subsection will briefly summarise conditions under which primitives decide to opt-out of the recognition process. Many of the exact parameter values are based on experimental observations. The primitives indicated between brackets are those more likely to have been drawn under the mentioned opt-out condition.

Circle : unnaturally large radius (long Line) *or* very small angular segment (short Line)

Crescent : unnaturally large radius *or* very small angular segment *or* small curvature length (Line)

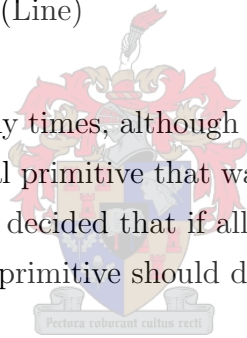
Corner : one of constituent lines decide to opt-out *or* their intersection does not form a *V* or *L* (single Line) *or* their length ratio is larger than 5:1 (Line).

Jagged : small (Line) or very large variation from line estimate *or* too short (Line)

Line : large variation from line estimate

Spiral : small (Line) or very large variation (Circle or Crescent) from line estimate *or* too short (Line)

It can be observed that many times, although not always, when a primitive decides to opt-out the actual primitive that was drawn was in actual fact a Line. For this reason it was decided that if all primitives decide to opt out of the decision process, the primitive should default to a Line.



3.5 Primitive Clustering

Primitives need to be grouped together to form symbols. The simplest and most dependable way is to require the user to issue a “recognise” command before the next symbol is drawn. This can be disruptive to the user’s experience. Instead, primitives can be automatically clustered together based on the order and position in which they were drawn.

Clustering is frequently based on marker symbols and nearest neighbour groupings [8]. A similar, albeit simpler, technique is followed by this system.

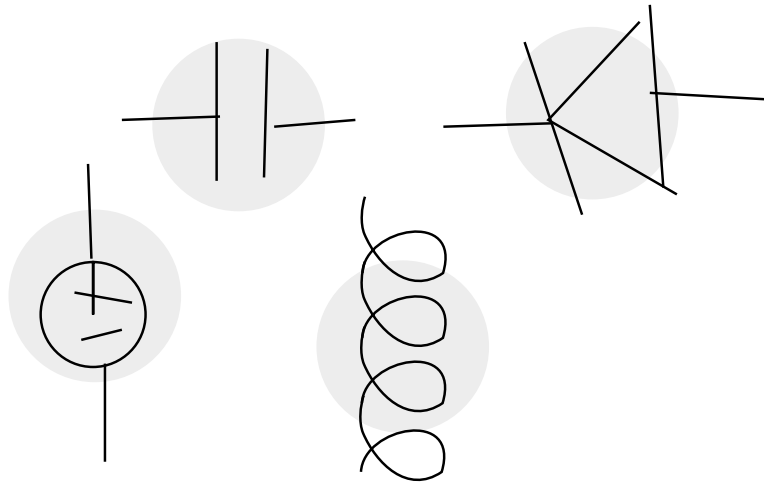


Figure 3.5: Clustering example.

Each primitive is first represented by a corresponding feature vector: the x and y coordinates of the primitive's midpoint, and the time the primitive was drawn. These three features are based on the observation that primitives are not only grouped together in physical proximity, but also in time.

The system groups these points (and thus primitives) together using multiple runs of the k -means algorithm.

The k -means algorithm works by initially partitioning the feature points into k sets (randomly or heuristically determined). The centroids (mean points) of these sets are calculated and the feature points re-assigned to their closest centroids. This process is repeated until points no longer switch between clusters.

The k -means algorithm requires a value for k , the number of clusters (i.e., symbols) to form. The algorithm is thus re-run with values for k ranging from one up to the number of primitives, stopping when forming more clusters does not yield a significantly smaller error.

This still, unfortunately, requires the user to draw symbols and issue a recognise command before being able to form connections between them.

3.6 Symbol Recognition

Drawn symbols are compared against a library of known definitions to find a likely match. The symbol definitions can be seen in Fig. 3.6.

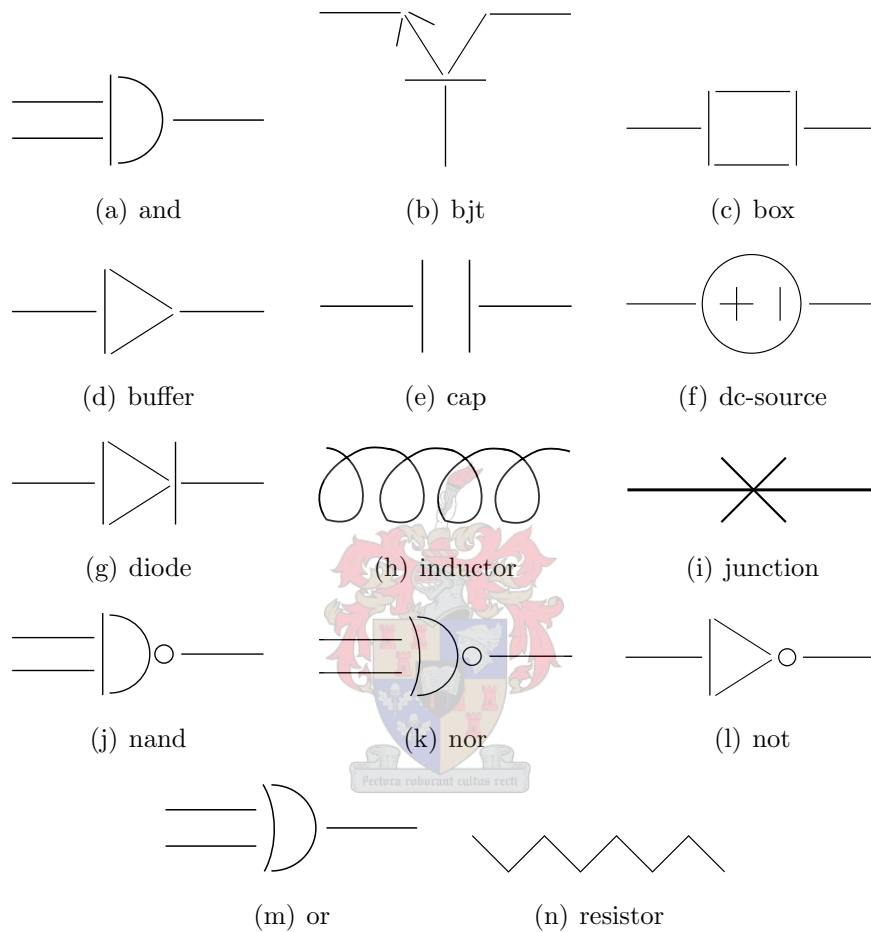


Figure 3.6: Symbol definitions.

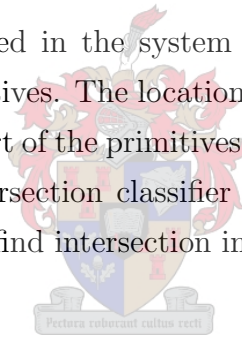
Representing symbols by a set of primitives instead of pixels, effectively transforms the recognition problem from one of matching raster-based images to that of comparing vector-based graphics. This significantly simplifies the symbol recognition.

Table 3.1: Intersection types.

intersection type	intersection location	intersection angle
V	endpoint-to-endpoint	$\approx 45^\circ$
L	endpoint-to-endpoint	$\approx 90^\circ$
y	midpoint-to-endpoint	$\approx 45^\circ$
T	midpoint-to-endpoint	$\approx 90^\circ$
X	midpoint-to-midpoint	$\approx 45^\circ$
+	midpoint-to-midpoint	$\approx 90^\circ$
=	none	parallel
	none	$\approx 0^\circ$
I	any	$\approx 0^\circ$

3.6.1 Primitive Intersections

Many of the algorithms used in the system depend on knowledge about intersections between primitives. The location of intersection, the angle between primitives and the part of the primitives intersected, are all important properties used by the intersection classifier discussed later. The mathematical algorithms used to find intersection information are summarised in Appendix B.



Information on intersections between primitives are generalised into various intersection types according to a simple rule-based system, as summarised in Table 3.1. X and + intersections happen when primitives cross close to each other's midpoints; V and L when they intersect near their endpoints, at an angle; T and y when midpoint and endpoint intersect; and I when they intersect, but not at a significant angle. | and = represent relationships between primitives when they do not intersect, but are co-linear or parallel. This is similar to the technique employed by Gennari and described in Section 2.1. However, their method contains only limited information on the relationship between non-intersecting primitives.

Fig. 3.7 shows an example of some of the extracted intersection types made

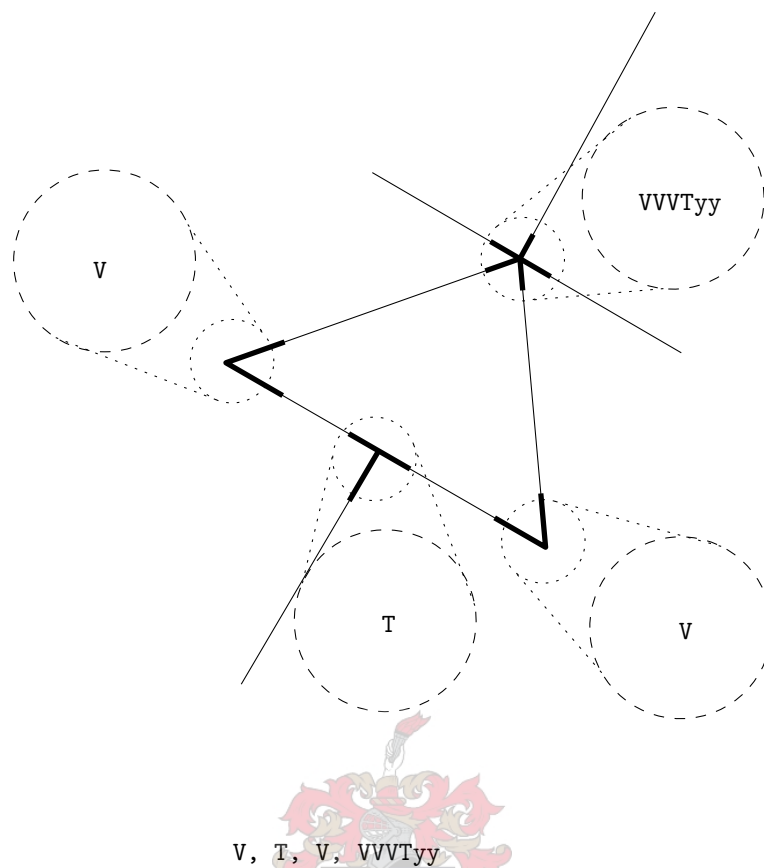


Figure 3.7: Analysis of primitive intersections for a *diode* symbol.

between the primitives that make up a *diode* symbol. L-intersections are illustrated later on in the box example in Fig. 3.9.

Because only a few of the symbols to be recognised contain circles, it was decided to ignore information on intersections between circles and any other primitives. Merely the count of the circles should be adequate in differentiating between those with and without circles.

3.6.2 Symbol Classifiers

A cluster of primitives is classified as a specific symbol type by taking the majority vote of two classifiers: a simple recogniser, built on primitive counts, and an intersection-based recogniser. Through experimentation it

was found that better results are obtained by providing the primitive-count based classifier with two votes.

The results of these classifiers could be fused together in a more sophisticated way, but this simplified approach was found sufficient for the purposes of this study after seeing the results discussed in Chapter 4. Conceptually the simple feature classifier does a rough classification and the full feature classifier incorporating intersection features are used to specialise.

Primitive-count based Recogniser

An elementary representation of a symbol can be found by forming a vector of the number of primitives. E.g. a *nor*-gate consists of 6 primitives: 3 lines, 2 crescents and one circle or $\langle 6, 3, 2, 1 \rangle$; while an *and*-gate consists of 5 primitives: 4 lines, 1 crescent and 0 circles or $\langle 5, 4, 1, 0 \rangle$.

It should be noted that this representation does not provide enough information to differentiate between some symbols such as *diodes* and *boxes* (both having 6 primitives, all lines) or *bjts* (Fig. 3.6(b)) and crystals (8 primitives, all lines). Information concerning the relative arrangement and connectivity of primitives is therefore also taken into account.

Intersection-based Recogniser

This recogniser is based largely on work touched upon by Gennari *et al.* [9]. To classify a symbol, a feature vector is constructed containing information on its intersecting primitives. A naive Bayesian classifier³ is then trained against a set of symbol examples and used for symbol classification.

Analysis of training data (See Fig. 3.8), however, suggests a Gaussian distribution, as was used by Gennari, might not be the best model for this data set. The vector elements are always positive integers and concentrated

³See Appendix A for an introductory tutorial.

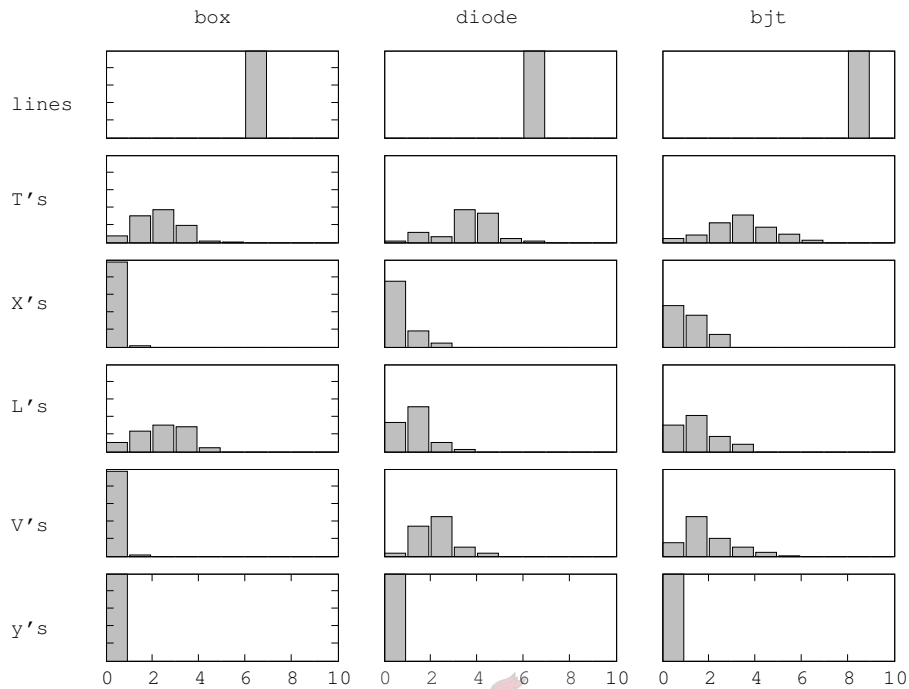


Figure 3.8: Histograms of selected features for *box*, *diode* and *bjt*.

close to the mean (with no deviation in many cases). Better results are obtained if a discrete probability density function is assumed. This is the same as constructing a histogram of the training data's features.

An example will demonstrate how this feature vector is constructed.

Example: Analysis of the box in Fig. 3.9 yields the geometric properties shown in the table. A simplified form of the vector is then the first column or $\langle 6, 6, 0, 0, 2, 0, 4 \rangle$.

There are however more intersection (Table 3.1) types than shown in the example. A count for each of the 9 intersection types, a count for each of the 5 primitive types (a Corner is converted to two Lines) and a total primitive count all contribute to form a vector of 15 elements.

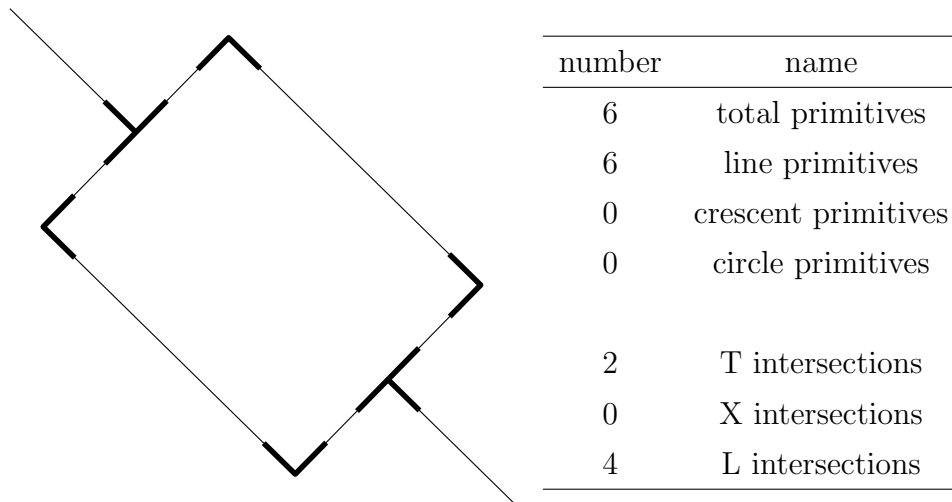
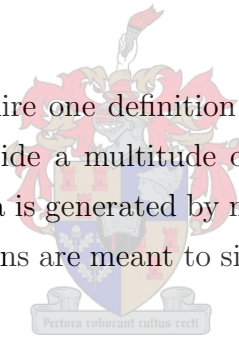


Figure 3.9: An example of a box, its primitive intersections and selected features.

3.6.3 Training Data

It is preferable to only require one definition for each symbol rather than prompting the user to provide a multitude of examples for each symbol. For this reason training data is generated by mutating a single definition of each symbol. These mutations are meant to simulate variability introduced by the user.



For each desired example of a symbol, points in the symbol definition are mutated by adding a Gaussian error. Snapping is then applied to the primitives to imitate corrections made by the user interface (Section 3.4).

The symbol recognition stage can be made more robust against misidentification by the primitive identification stage. These errors are simulated by measuring the accuracy and misidentification tendencies of primitives and applying these type-mutations to generate more representative training data. Examples of the mutations produced by this process can be seen in Fig. 3.10.

Generating training data according to the previous steps can yield rather extreme variations from the definition files. It is unreasonable to expect the

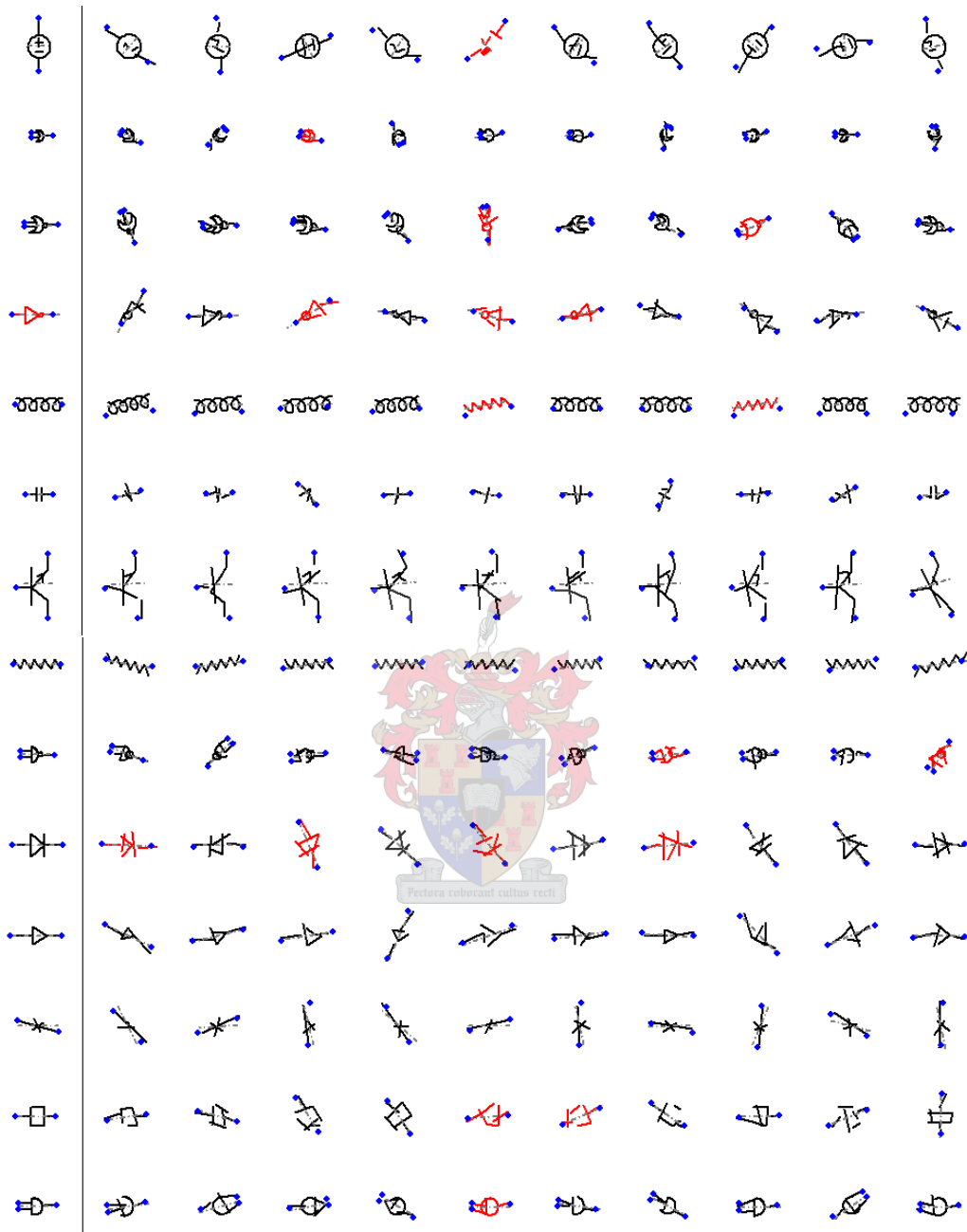


Figure 3.10: Symbol definitions (leftmost column) and examples of generated mutations. Symbols in red where identified by the initial classifier as “unreasonable”.

system to train on these extremes and still produce accurate results.

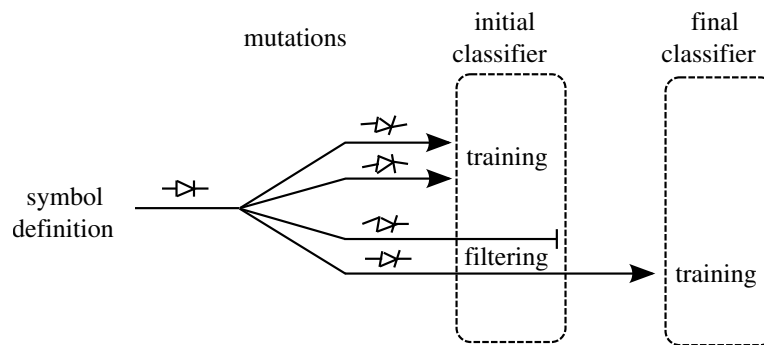
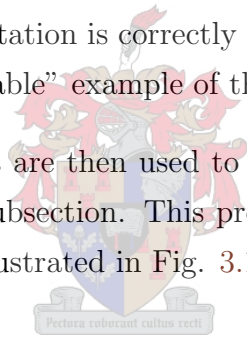


Figure 3.11: Identification of “reasonable” mutations for training a final symbol classifier.

A set of mutations is therefore used to create an initial rough symbol recogniser. Another set of generated mutations is then filtered by a run through this naive classifier. If a mutation is correctly identified by this classifier, it is expected to be a “reasonable” example of the symbol.

These reasonable mutations are then used to train the combined classifier described in the previous subsection. This process of filtering out “unreasonable” training data is illustrated in Fig. 3.11



3.7 Symbol Properties

3.7.1 Primitive Mapping

After a group of primitives is recognised as a specific symbol from the database, it is necessary to know which primitive represents which part of said symbol. This information is needed to determine the rotation and port locations of the symbol.

The group of primitives drawn by the user can be represented as a graph where primitives act as nodes and edges relate the intersection features.

This converts the problem into one of finding an optimal node mapping (or morphism) between the graphs of the drawn symbol and symbol definition.

Inexact graph matching is a computationally expensive process (see Section 2.4). Because this information is needed during time-critical parts of the user's interaction, a very simple method of determining a mapping is used.

A primitive is represented by a feature vector consisting of

- the number of intersections with other primitives
- the normalised distance of the primitive from the symbol midpoint
- the normalised size of the primitive

This feature vector is then used in a naive Bayesian classifier with a diagonal Gaussian probability density function to match up primitives between the symbols.

The drawn symbol is not compared only to the symbol definition, however. Several mutations of the symbol definition are used instead to compensate for too clinical definitions. Two lines might, for example, be exactly parallel according to the definition, but might lose this property very easily when drawn by a user.

The results of these matchings are then combined using a majority vote.

3.7.2 Symbol Rotation and Translation

Information on symbol rotation and translation is needed to correctly draw component parts in the correct locations⁴.

If each primitive's point set (x_n , where x_n is (x, y) coordinate) is rotated and/or reflected (R_n) around the symbol's center point (m), uniformly

⁴For a comparison of different methods of estimating rigid body transformations see [27].

scaled (S_n) and translated to around a different center point (p), to form a new point-set (x'_n) the transform can be modelled as

$$x'_n = T_n(x_n - m) + p, \quad (3.1)$$

where

$$T_n = S_n R_n \quad (3.2)$$

with S_n a positive-definite diagonal matrix, and R_n a unitary matrix.

Assuming the same transform for all primitives yields a simpler overdetermined result:

$$x'_n \approx T(x_n - m) + p + n(t) \quad (3.3)$$

where T is the generalised transform and $n(t)$ is Gaussian noise added to compensate for the assumption.

Since m and p is known, T in (3.3) can be approximated by employing an estimation algorithm such as least-squares. However if the primitive mapping is not known and can only be estimated (Section 3.7.1), a robust estimation technique has to be used to derive meaningful information as linear least squares is very sensitive to outlier data which can occur due to incorrect mappings.

To address this the least-squares estimate is recalculated, ignoring points assumed to be outliers. These outliers are identified as those whose squared residual error differs more than the average squared residual assuming all points contribute equally to the total error.

The general transform, T , is then further factorised into (3.4) using singular value decomposition (SVD).

$$T = R_1 S R_2 \quad (3.4)$$

SVD yields the 2x2 unitary matrices R_1 and R_2 , along with S , a 2x2 matrix with non-negative numbers on the diagonal and zeros off the diagonal⁵. R_1

⁵See Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/Singular_value_decomposition

and R_2 are then rotation/reflection effects and S a scaling in the x and y directions.

Since the scaling of the symbol is not used by the system; the scaling influence of S is removed to produce a pure rotation/reflection transform:

$$R = R_1 R_2. \quad (3.5)$$

The rotation angle (θ) and whether a reflection occurred (f) can then be modelled as the resulting R , a 2x2 unitary matrix:

$$R = \begin{bmatrix} f & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} = \begin{bmatrix} r_1 & r_2 \\ r_3 & r_4 \end{bmatrix} \quad (3.6)$$

where

$$f = \begin{cases} -1 & \text{if reflection} \\ 1 & \text{if no reflection} \end{cases} \quad (3.7)$$

From (3.6) it is then possible to derive θ and whether there was a reflection:

$$r_4 = \cos \theta \quad (3.8)$$

$$\det R = f \cdot (\cos^2 \theta + \sin^2 \theta) = f \quad (3.9)$$

These results determine the orientation in which the component was sketched and how to rotate and/or reflect the corresponding definition to be displayed on the screen.

3.8 Interface

Because the system depends heavily on user interaction, accuracy in determining what was implied by the user depends as much on the quality of the user interface as the quality of the component recognition algorithms. Fig. 3.12 shows an example view of the developed interface.

Tablets typically have a “drawing” mode where the keyboard rotates behind the screen and the user interacts exclusively by means of a pen device. This

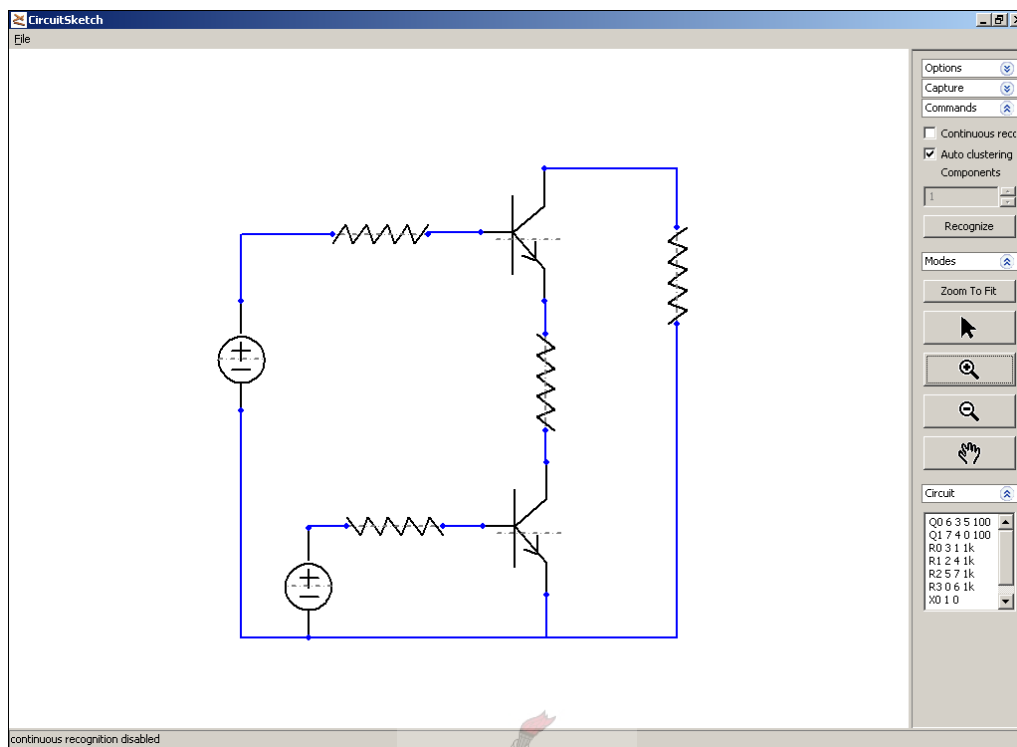


Figure 3.12: Interface example

limits the user to clicking, secondary clicking (sometimes with a button on the side of the pen) and dragging motions. The interface had to be carefully designed to allow the multitude of instructions the user wants to issue when drawing. The state diagram is pictured in Fig. 3.13.

This allows the user to draw using the left mouse button and move/rotate symbols with the right. For a pen/tablet based system this translates to normal pen mode and the pen secondary mode (usually enabled by pressing a button on the pen device).

The interface differentiates between move and rotation actions by determining which is closer when the command was issued: a symbol's midpoint or a port.

When the pen is lifted the user is also informed in the status bar of clustering information and what the current primitives will be recognised as in the

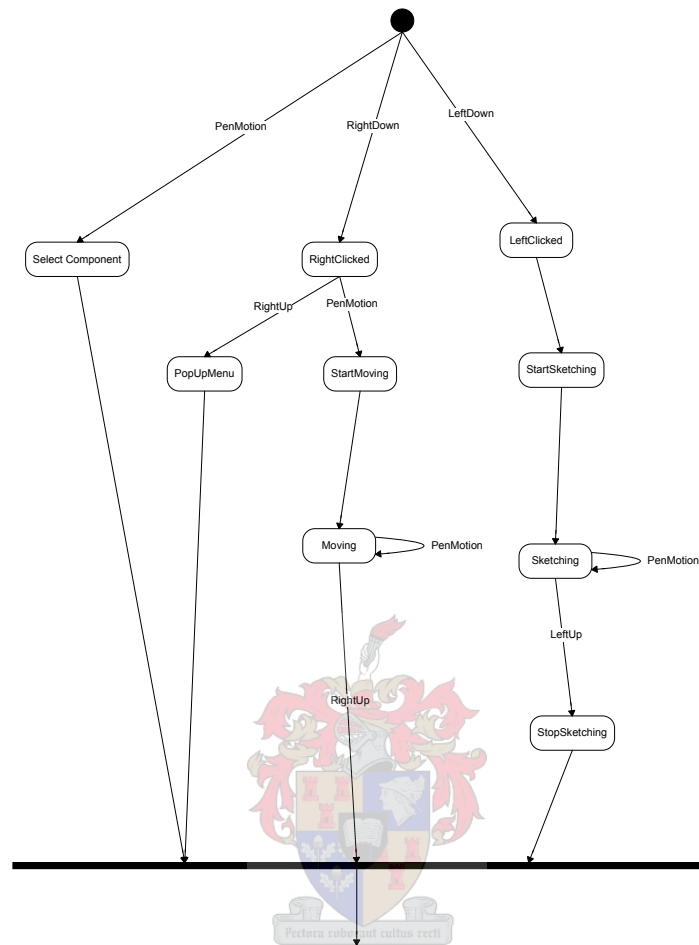


Figure 3.13: State machine diagram of interface reaction to user events.

event that the “recognise” command is issued. This not only allows the user to stop drawing when the system has enough information, but also, and more importantly, prevents the irritating loss of all primitives in case of incorrect recognition.

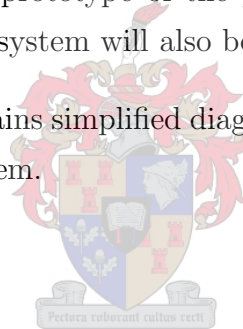
3.9 Design Summary

For readability the design and implementation is presented in such a way as to imply that a segmented process of research and development was undertaken; this was not the case. The development of a recognition engine was largely an experimental process. Ideas were roughly thought through, implemented, tested, removed if applicability seemed questionable and sometimes later integrated back into the system.

3.10 Implementation

This section briefly discusses the development methodology followed and tools used to implement a prototype of the proposed design. Some finer points on the implemented system will also be discussed briefly.

Note that Appendix C contains simplified diagrams and tables of the classes and files related to the system.



3.10.1 Tools

With only a single developer, tools were needed that allowed rapid development. The development tools also needed to facilitate an exploratory coding style, which is necessary for effective research.

The system was designed and developed over a period of 18 months, extensively using a version control system and a multi-paradigm programming language.

Programming language : The system was implemented using Python 2.4⁶ and totals around 4500 lines of code, excluding libraries. Python

⁶<http://www.python.org/>

is an interpreted dynamically typed programming language with automatic memory management and was chosen for its rapid development nature and focus on being fun to use.

Scientific libraries : Numarray is used in many places especially when determining component rotation and representing probability density functions with Gaussian distributions. Numarray's k -means clustering algorithm was used in the initial phases of design but eventually replaced with Pycluster's⁷ implementation for performance reasons.

Graphs are represented by a modified version of the pygraphlib⁸ library, which is unfortunately no longer maintained

Version control : A Subversion⁹ repository was used extensively. This mitigated the risk that comes with the continuous experimentation with source code needed for this research.

Graphics Library : An interface was built around the wxPython¹⁰ Float-Canvas component. This allowed easy conversion between screen and world coordinates.

Database : Primitive and symbol examples were captured and stored in XML structured files. This made it possible to edit data files by hand if need be.

Graphical Analysis : Matplotlib¹¹ is used to produce plots for visual statistical analysis.

Development Environment : SciTE¹², a lightweight and customisable text editor was used. This allowed the dedication of computer resources to the interface and algorithm design.

⁷<http://bonsai.ims.u-tokyo.ac.jp/~mdehoon/software/cluster/software.htm>

⁸<http://pygraphlib.sourceforge.net/>

⁹<http://subversion.tigris.org/>

¹⁰<http://www.wxpython.org/>

¹¹<http://matplotlib.sourceforge.net/>

¹²<http://www.scintilla.org/SciTE.html>

3.10.2 Implementation Methodology

Because the prototype development is primarily research driven and the solution would not be clear until later, the design decisions are very much dependent on the results of the implementation's recognition abilities. This had the effect of making the development very much an exploratory undertaking.

Knowledge of the problem increased as the prototype was implemented, parts were exchanged for different algorithms and functionality was moved from one area of the system to another to improve performance. This made any kind of large up-front design rather irrelevant, as it would most likely be changed as the system grew.

One way of managing these changes is by refactoring often [28]. This involves restructuring code into a better architecture.

Because Python classes are mutable, refactoring is slightly more difficult than in other statically typed languages¹³. Refactoring tools such as Bicycle Repair Man¹⁴ have been developed to help with this, but tests are still needed to ensure code does not break during refactoring.

Unit tests (specifically *doctests*¹⁵) were written for some of the basic functions. Exhaustive unit tests, however, were avoided for most of the system as it would have duplicated code that would have to change often. Instead, it was decided to automate the recognition of example data to exercise the code base. With sufficient code coverage and sanity checks through the source (i.e. *asserts*) it was hoped many of the faults would be discovered as they are introduced. These tests were run regularly, typically several times a day.

It is generally believed that state coverage is superior to code coverage [28]. It is the author's opinion, however, that state coverage is less important

¹³<http://c2.com/cgi-bin/wiki?PythonRefactoringBrowser>

¹⁴<http://bicyclerepair.sourceforge.net/>

¹⁵<http://docs.python.org/lib/module-doctest.html>

in Python than other languages. As Python typically processes collections (such as lists) using an iterator and not incrementing indices, there are less ways in which *out of bounds* errors can occur. Since variables also need not be declared it is also less likely that *null* pointers would be dereferenced. This is, for example, one of the most frequent sources of errors in the Java programming language [29].

3.10.3 Data Formats

Data used by the recognition engine was stored primarily in XML formatted files. This section will briefly illustrate the structures and how they relate to NioCAD's data representation.

Symbols and Primitive XML files have the same structure: a drawing consisting of various primitives and each primitive consisting of a set of points. The following fragment demonstrates the definition of a *dc-source* symbol. Note that the symbol definition also contains information on the port locations.

```
1 <drawing author="janto" host="janto" date="Mon_Feb_06_13
   :29:02_2006">
2 <primitive type="Circle">
3   <point x="-216.75" y="96.83" time="0" />
4   <point x="-211.24" y="93.78" time="0" />
5   ...
6   <point x="-216.75" y="96.83" time="0" />
7 </primitive>
8 <primitive type="Line">
9   <point x="-240.0" y="90.0" time="36.74" />
10  <point x="-240.0" y="50.0" time="37.14" />
11 </primitive>
12 ...
13 <ports>
14   <point x="-240.0" y="180.0" time="83.82" />
15   <point x="-240.0" y="-60.0" time="58.94" />
```



```

16 </ports>
17 </drawing>

```

The *component_properties.xml* file specifies the number of ports and SPICE information for each component type:

```

1 <components>
2 <component name="resistor" spice_label="R" values="1k" />
3 <component name="inductor" spice_label="L" values="1u" />
4 <component name="capacitor" spice_label="C" values="1n" /
  >
5 <component name="bjt" spice_label="Q" port_count="3"
  values="100" />
6 ...
7 <component name="buffer" spice_label="B" values="" />
8 <component name="nand" port_count="3" spice_label="X"
  values="" />
9 <component name="and" port_count="3" spice_label="X"
  values="" />
10 <component name="not" port_count="2" spice_label="X"
  values="" />
11 </components>

```

As NioCAD was still under development during the design of this prototype, NioCAD has its own representation of the same information:

```

1 <Type>
2 <Spice name="capacitor" basetype="PartType" module="
  StdTypes">
3 <PartType>
4 <PortType>
5 <ComponentType>
6 <identifier>C</identifier> <name>Capacitor</name>
  >
7 </ComponentType>
8 <Node aliase="p" /> <Node aliase="n" />

```

```

9      </PortType>
10     <PropertyType>
11       <name>value</name> <value>0</value>
12       <PropertyOptionsType> <showname>>false</showname> <
13         required>>true</required> </PropertyOptionsType>
14     </PropertyType>
15 </PartType>
16 </Spice>
17 <Symbol name="capacitor" basetype="SchemSymbol">
18   <Line> <Point><x>-50</x><y>-10</y></Point> <Point><x>
19     50</x><y>-10</y></Point> </Line>
20   ...
21   <Line> <Point><x>0</x><y>10</y></Point> <Point><x>0</x>
22     <y>80</y></Point> </Line>
23   <Pin> <Point><x>0</x><y>80</y></Point> <Point><x>0</x>
24     <y>100</y></Point> </Pin>
25   <Pin> <Point><x>0</x><y>-80</y></Point> <Point><x>0</x>
26     <y>-100</y></Point> </Pin>
27   <Node alias="p"> <Point><x>0</x><y>-100</y></Point> <
28     /Node>
29   <Node alias="n"> <Point><x>0</x><y>100</y></Point> </
30     Node>
31 </Symbol>
32 </Type>

```

The difference in representation is, however, largely structural and should therefore be possible to eliminate easily.

Chapter 4

Evaluation

In this chapter the prototype described in the previous sections is evaluated to prove the viability of the proposed design. The quality of the implemented system is explored with regards to its recognition accuracy, performance and usability.

4.1 Recognition Accuracy

The system's ability to interpret a user's strokes depends on its accuracy in identifying primitive shapes and component symbols.

The engine was primarily tested against procedurally generated data, but also against data informally collected from a small group of users.

A classifier's recognition tendencies can be represented by confusion matrices, where columns represent instances in the predicted class and each row represent instances in the actual class. It is then relatively simple to see how classes are confused by a classifier. These matrices are shown in Table 4.2 and Table 4.5.

Table 4.1: Primitive example set size

Primitive type	Number of examples
Circle	175
Corner	99
Crescent	74
Jagged	43
Line	384
Spiral	111

Table 4.2: Primitive confusion matrix (values rounded down)

	Circle	Corner	Crescent	Jagged	Line	Spiral
Circle	91	0	5	0	3	0
Corner	0	96	4	0	0	0
Crescent	0	0	100	0	0	0
Jagged	0	0	0	91	4	4
Line	0	0	1	0	98	0
Spiral	0	0	0	0	0	100
Overall accuracy: 96%						

4.1.1 Primitive Identification Accuracy

Although the accuracy of the primitive identification process is not vital as the user implicitly verifies the result of the classifier, it can irritate the user if the system is not consistent with its identification - continuously swapping between primitive types while the stroke is being drawn.

A set of examples was acquired from five users for each primitive type. The number of examples collected for each primitive type is indicated in Table 4.1. Holdout cross-validation, the simplest type of cross-validation, was used: 75% of the examples was used to train the identifier (as described in Section 3.4) while the rest was employed as a test set.

The results of this test can be seen in Table 4.2. With an average class

Table 4.3: Symbol example set size

	and	bjt	box	buffer	capacitor	dc-source	diode	inductor	junction	nand	nor	not	or	resistor	<i>total</i>
eric		5		8	7	6	6					5			37
gerrit	5	2					6			6		2			21
janto	17	11	14	21	21	9	12	16	9	9	10	9	8	12	178
jvdh	5	6	12		13	4	4	6			4	8	6	8	76
mareliz		1			3			1							5
<i>total</i>	27	25	26	29	44	19	28	23	9	15	14	24	14	20	

accuracy of 96% and a minimum of 91%, these results suggest that the primitive identification process is quite accurate in differentiating between shapes.

4.1.2 Symbol Recognition Accuracy

By breaking down recognition into two separate stages with the user implicitly verifying the results of the primitive identification stage, input to the symbol recognition stage can be quite close to the definition file.

It should be noted that symbols consisting of only one primitive (i.e., the resistor and inductor) can, theoretically, be identified perfectly if the primitive was correctly identified. Recognition of these symbols was, however, intentionally left to the same classification strategy as other symbols. This was primarily done to allow the classifier to be as generic as possible.

The symbol recogniser was evaluated against component examples collected from a few users and examples procedurally generated from the symbol definitions.

User Drawn Symbols

A small (Table 4.3) test corpus of symbols was collected from a few students. Classification accuracy results are displayed in Table 4.4.

While drawing symbols, many users expected the system to recognise a larger range of polygons, particularly triangles and squares. After users

Table 4.4: Symbol example set classification accuracy

	and	bjt	box	buffer	capacitor	dc-source	diode	inductor	junction	nand	nor	not	or	resistor
eric		60		37	100	83	0					40		
gerrit	80	0					0			33		50		
janto	88	90	85	95	100	88	25	100	100	100	90	22	100	100
jdvh	60	33	91		92	100	25	83			100	12	83	75
mareliz		100			66			100						
average	76	56	88	66	89	90	12	94	100	66	95	31	91	87

were shown the primitive types the system can identify and allowing them to experiment with the system before collecting test data, recognition accuracy improved significantly.

From Table 4.4 it is clear that the *diode* and *not* symbols were incorrectly recognised most of the time. It is difficult to determine the reason behind incorrect classifications from such a small example set. For this reason the system was also tested against a large corpus of procedurally generated symbol examples.

Automatically Generated Symbols

A test corpus containing 100 examples for each symbol was generated in a way similar to the training data as described in Section 3.6.3, except for not being filtered through an initial classifier. This avoids dependency on the symbol classifier which would cause testing only data correctly classified. This unfortunately means some absurd mutations are tested against the database. It should, however, be sufficient as only the classification tendencies are to be determined and not the actual accuracy values.

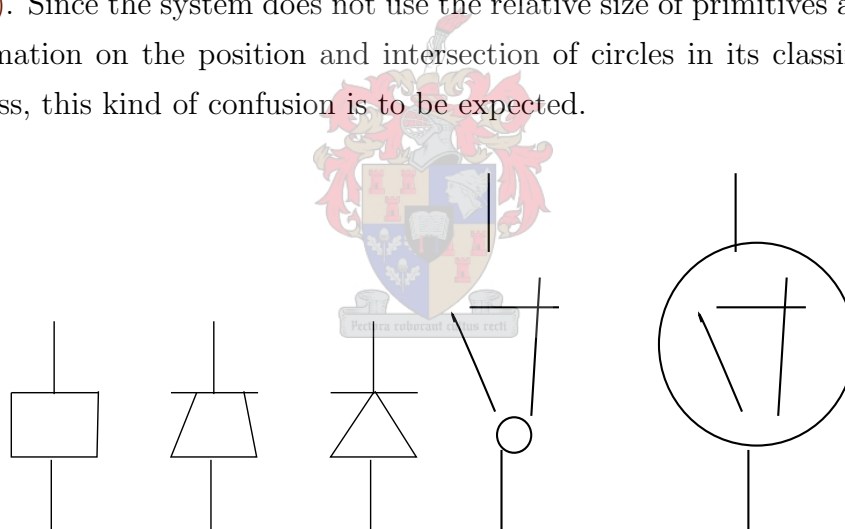
Classification tendencies on the examples can be deduced from the confusion matrix shown in Table 4.5. From this table it is clear to see that although many symbol classes are correctly classified, the system easily confuses *diodes* with *boxes* and *nots* with *dc-sources*.

Some roughly drawn symbols are inherently ambiguous and even difficult for humans to classify. Fig. 4.1(a) shows a symbol that can be classified

Table 4.5: Symbol confusion matrix

	and	bjt	box	buffer	capacitor	dc-source	diode	inductor	junction	nand	nor	not	or	resistor
and	96	0	0	0	0	0	0	0	0	0	0	0	4	0
bjt	0	98	1	0	0	1	0	0	0	0	0	0	0	0
box	0	0	87	0	0	5	2	0	0	0	0	6	0	0
buffer	7	0	0	91	1	1	0	0	0	0	0	0	0	0
capacitor	1	0	0	0	95	1	0	0	3	0	0	0	0	0
dc-source	0	0	7	0	1	87	0	0	0	4	0	1	0	0
diode	0	0	59	0	0	2	29	0	0	0	0	10	0	0
inductor	0	0	0	0	0	0	0	96	0	0	0	0	0	4
junction	0	0	0	0	0	0	0	0	100	0	0	0	0	0
nand	0	0	0	0	0	0	0	0	0	85	9	6	0	0
nor	0	0	0	0	0	4	1	0	0	3	92	0	0	0
not	0	0	6	0	1	50	1	0	0	7	0	35	0	0
or	2	0	0	0	0	0	0	0	0	0	0	0	98	0
resistor	0	0	0	0	0	0	0	15	0	0	0	0	0	85

as either a *diode* or a *box*. The confusion between *nots* with *dc-sources* is slightly more difficult to explain until one looks at the example in Fig. 4.1(b). Since the system does not use the relative size of primitives and any information on the position and intersection of circles in its classification process, this kind of confusion is to be expected.



(a) Confusion between *box* and *diode*. (b) Confusion between *not* and *dc-source*.

Figure 4.1: Symbols easily confused.

4.2 Performance

The most noticeable performance issue is the delay experienced when the system attempts to identify the drawn primitive. Because the system continuously recalculates and displays its interpretation of a user's stroke and heavy processor usage can interfere with the stroke sample rate, and by extension quality of recognition, this part needs to be especially fast. After profiling, it was determined that this delay is primarily due to the primitive recogniser attempting to construct a Corner primitive from the points drawn for the feature vector. This is processor intensive as the system does a linear search through the points to find the optimal point to form the bend. The search time can be reduced to logarithmic by doing a binary search instead.

The training of the primitive classifier also takes a few minutes to run, however, this is not an issue from the user's point of view as the training only needs to be done once and a cached version is used across program runs. Optimising the construction of the Corner primitive, as discussed above, will also significantly improve the training stage.

Surprisingly, training the symbol recogniser only takes a few seconds and is fast enough to regenerate each time the program launches. This is primarily due to using discrete probability functions in the naive Bayesian classifier, and symbols with typically less than seven primitives which reduces intersection calculations.

4.3 Usability Analysis

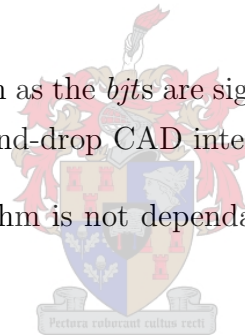
After using the program to draw a few circuits, five users were asked to rate the system's accuracy, ability to draw circuits and their overall impression.

Requiring the user to issue a recognise command after every symbol was found to be disruptive to the design process. In the design of electronic

circuits, groups of components form design patterns and are usually drawn in fast “spurts” without necessarily thinking on the level of the individual components. Interrupting the user after every component is drawn can slow down and frustrate the designer who is trying to accomplish more than simply experiment with the system.

There were several other recurring comments:

- Users especially enjoyed what they described as a natural way of rotating and connecting up components.
- Many believe the system shows promise and would use it after refinements were made.
- Users expected the system to identify various polygons such as rectangles and triangles.
- Complex symbols such as the *bjts* are significantly easier to draw with the traditional drag-and-drop CAD interface.
- The clustering algorithm is not dependable when drawing larger circuits.



The issues mentioned here could be addressed in various ways, some of which are discussed in Chapter 5.

Chapter 5

Recommendations and Conclusions

Refinements to the system's design in terms of its algorithm and interface could be explored in a future study to improve the effectiveness of recognising hand-drawn circuit diagrams. This chapter discusses some suggestions for further research and improvements that could be made. Some suggested applications are also mentioned and this study is concluded in a summary.

5.1 Future Work

5.1.1 Algorithm Enhancements

Users of the system expected it to recognise polygons. This is approached by others by segmenting strokes based on timing information. This would, however, conflict with the jagged primitive, which is necessary for recognising a resistor's symbol in circuit diagrams. Another option is to add additional pseudo-primitives similar to a Corner, such as Triangle and Rectangle which would be decomposed into Line primitives after identification.

Some existing systems report positive results when using a pre-processor to segment strokes into more than one primitive. Each stroke of the user's pen is segmented based on temporal and curvature information. Although this study uses a corner pseudo-primitive, a more generic solution to enable multiple primitives with single stroke (particularly lines) might be useful.

An improved clustering algorithm will improve the practicality of drawing complete circuits. Together with intelligent primitive segmenting based on domain knowledge, connecting wires between components (and thus complete schematics) could even be drawn before issuing a recognition command. Hierarchical clustering should also be explored where primitives are first grouped in time and then position.

Ports drawn close to each other could also automatically form connecting lines.

Planar graph matching could be explored in more depth as a possible symbol recognition strategy. The algorithm used in this study has theoretical limitations when trying to differentiate between symbols that are almost exactly the same, such as a pnp and npn *bjt* (i.e., arrow pointing inwards or outwards).

Using various classification strategies and then fusing together the results can produce better accuracy than the individual classifiers. It has been shown that although there exists ambiguities within the results of an individual classifiers, different classifiers are wrong in different ways and correct in the same way. Combining pixel-based features such as moments, which was mentioned in Section 2, to the classification engine should therefore improve accuracy.

An easier way to incorporate moment information, and improve on the accuracy of symbols that include circles, would be to simply add moment features to the vector used for symbol recognition.

5.1.2 Interface Enhancements

The pen sampler does not react immediately to user input on the tablet PC. This is most likely due to using generic mouse events and not using the Tablet API (wintab) directly.

Gestures such as scratching out a component could be introduced to further simplify the user interface. A character recogniser could also be incorporated, so as to enable users to easily enter values associated with components.

Going back and changing an already drawn component should be possible. It might be desirable, for example, to change a *buffer* into a *diode* or an NPN *bjt* into a PNP *bjt* symbol. This would also allow a user to fix errors in classification.

Many of the interruptions and irritations in using the current system emanate from two of the requirements and assumptions made in Section 3.2:

- A single primitive is represented by a single pen stroke.
- One symbol is drawn at a time and completed before the next one is started.

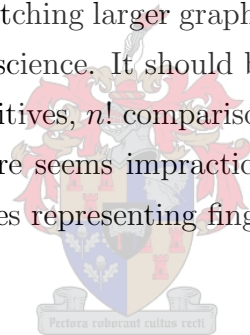
Two phenomena noticed by Oltmans *et al.* [30] directly contradict these assumptions: 1) users do not always draw each object with a sequence of consecutive strokes and 2) users drew more than one object using a single stroke. Although the constraints imposed by these assumptions are relaxed significantly by the prototype, more can be done.

This study only focused on one aspect of making the electronic design process more intuitive. A study of user behaviour when drawing simple circuit diagrams might prove very useful in this goal.

5.2 Suggested Applications

The value of integrating the discussed recognition algorithms with an existing CAD system should be evaluated. Such a system could allow the user to enter components in a hybrid manner. Such a system could, for instance, use sketch recognition to trim the search space for a component before requiring the user to select the desired component. A user could, for example, sketch a *diode* symbol, after which a list containing a generic diode, Zener diode and LED symbol is displayed for final selection. Such a system would only be required to recognise a few basic components, e.g. connections, resistors, capacitors, inductors, op-amps, diodes and boxes.

It would be interesting to explore multi-domain problems using the algorithm employed. It is natural to consider the applicability of the symbol recognition algorithm to matching larger graphs, as it is an important problem still open in computer science. It should be noted, however, that for a symbol consisting of n primitives, $n!$ comparisons are required to determine all intersections. It therefore seems impractical for matching graphs with many primitives such as ones representing fingerprints¹.



5.3 Conclusions

This study explored the viability of using pattern recognition for schematic capture of circuit diagrams.

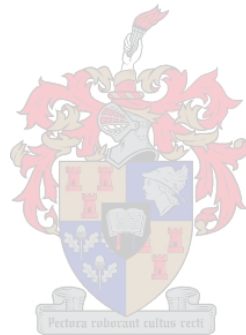
To achieve this goal it was necessary to research work related to on-line schematic recognition. An algorithm was then designed based on these ideas. A prototype of a circuit recognition engine was also implemented to demonstrate a possible approach.

The developed prototype was found to be reasonably accurate and usable when drawing small circuits. Using this system, it is possible to draw sim-

¹Although see [17] for reducing fingerprints into small attributed relational graphs.

ple diagrams of electronic component symbols and connecting wires. It is therefore possible to create a system that can be used to at least draw an approximate circuit in the early design stages.

It should therefore be feasible to use a pen-based interface and underlying recognition engine to capture circuit diagrams and thus provide an attractive early design environment for the engineer. This can be done by creating a plug-in as an initial input stage to an existing CAD system such as NioCAD.



Bibliography

- [1] C. Alvarado, “A natural sketching environment: Bringing the computer into early stages of mechanical design,” Master’s thesis, Massachusetts Institute of Technology, May 2000. [cited at p. 2, 3]
- [2] L. Wenying, “On-line graphics recognition: State-of-the-art.” in *GREC 2003: 5th IAPR International Workshop on Graphics Recognition*, 2003, pp. 291–304. [cited at p. 3]
- [3] L. B. Kara and T. F. Stahovich, “Hierarchical parsing and recognition of hand-sketches,” *ACM Symposium on User Interface Software and Technology (UIST)*, 2004. [cited at p. 6, 7]
- [4] T. Hammond and R. Davis, “Tahuti: A geometrical sketch recognition system for uml class diagrams,” *Papers from the 2002 AAAI Spring Symposium on Sketch Understanding*, pp. 59–68, March 25-27 2002. [cited at p. 6]
- [5] C.-L. Liu, S. Jaeger, and M. Nakagawa, “Online recognition of chinese characters: The state-of-the-art,” *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 26, no. 2, pp. 198–213, 2004. [cited at p. 6]
- [6] G. Taubman, “MusicHand: A handwritten music recognition system,” 2005, Honors thesis. [cited at p. 6, 12]
- [7] O. Ejofodomi, S. Ross, A. Jendoubi, M. Chouikha, and J. Zeng, “Online handwritten circuit recognition on a Tablet PC,” *33rd Applied Imagery Pattern Recognition Workshop (AIPR’04)*, pp. 241–245, 2004. [cited at p. 6]

- [8] L. B. Kara, “Automatic parsing and recognition of hand-drawn sketches for pen-based computer interfaces,” Ph.D. dissertation, Department of Mechanical Engineering, Carnegie Mellon University, Pittsburgh, PA, Sept. 2004. [cited at p. 6, 25]
- [9] L. M. Gennari, L. B. Kara, T. F. Stahovich, and K. Shimada, “Combining geometry and domain knowledge to interpret hand-drawn diagrams,” *Computers and Graphics*, vol. 29, no. 4, pp. 547–562, 2005. [cited at p. 7, 11, 30]
- [10] C. Alvarado and R. Davis, “Dynamically constructed Bayes nets for multi-domain sketch understanding,” in *Proceedings of IJCAI-05*, San Francisco, California, August 1 2005, pp. 1407–1412. [cited at p. 8, 9]
- [11] C. Alvarado, “Multi-domain sketch understanding,” Ph.D. dissertation, Massachusetts Institute of Technology, August 2004. [cited at p. 8, 9]
- [12] T. Hammond and R. Davis, “LADDER: A sketch recognition language,” in *MIT Computer Science and Artificial Intelligence Laboratory Annual Research Abstract*. MIT CSAIL, September 2004. [cited at p. 8]
- [13] C. Alvarado, M. Oltmans, and R. Davis, “A framework for multi-domain sketch recognition,” *AAAI Spring Symposium on Sketch Understanding*, pp. 1–8, 2002. [cited at p. 9]
- [14] O. Veselova, “Perceptually based learning of shape descriptions,” Master’s thesis, Massachusetts Institute of Technology, Cambridge, MA, 2003. [cited at p. 9]
- [15] O. Veselova and R. Davis, “Perceptually based learning of shape descriptions,” *Proceedings of the Nineteenth National Conference on Artificial Intelligence (AAAI-04)*, pp. 482–487, 2004. [cited at p. 9]
- [16] T. M. Sezgin and R. Davis, “HMM-based efficient sketch recognition,” in *Proceedings of the International Conference on Intelligent User Interfaces (IUI’05)*. New York, New York: ACM Press, January 9-12 2005, pp. 281–283. [cited at p. 10]
- [17] D. M. A. Lumini and D. Maltoni, “Inexact graph matching for fingerprint classification,” *Machine Graphics and Vision Special Issue on Graph*

- Trasformations in Pattern Generation and CAD*, vol. 8, no. 1, pp. 231–248, Sept. 1999. [cited at p. 10, 56]
- [18] C. Calhoun, T. F. Stahovich, T. Kurtoglu, and L. B. Kara, “Recognizing multi-stroke symbols,” *AAAI Spring Symposium 2002, Sketch Understanding*, 2002. [cited at p. 10, 11, 20]
- [19] E. Bengoetxea, “Inexact graph matching using estimation of distribution algorithms,” Ph.D. dissertation, Ecole Nationale Supérieure des Télécommunications, Paris, France, Dec. 2002. [cited at p. 11, 12]
- [20] D. Eppstein, “Subgraph isomorphism in planar graphs and related problems,” *Journal of Graph Algorithms & Applications*, vol. 3, no. 3, pp. 1–27, 1999. [cited at p. 12]
- [21] R. Brooks, “Exact probabilistic inference for inexact graph matching,” Apr. 2003, McGill Centre for Intelligent Machines. [Online]. Available: <http://www.cim.mcgill.ca/~rbrook/graphs/graph.pdf> [cited at p. 12]
- [22] S. X. Liao and Q. Lu, “A study of moment functions and its use in Chinese character recognition,” *International Conference on Document Analysis and Recognition*, 1997. [cited at p. 12]
- [23] S. X. Liao, A. Chiang, Q. Lu, and M. Pawlak, “Chinese character recognition via Gegenbauer moments,” *16th International Conference on Pattern Recognition (ICPR’02)*, vol. 3, p. 30485, 2002. [cited at p. 12]
- [24] H. Hse and A. R. Newton, “Sketched symbol recognition using Zernike moments,” *International Conference on Pattern Recognition*, 2004. [cited at p. 12]
- [25] S. H. Ong and P. A. Lee, “Image analysis by Tchebichef moments,” *IEEE Transactions on Image Processing*, Sept. 2001. [cited at p. 12]
- [26] J. Arvo and K. Novins, “Fluid sketches: continuous recognition and morphing of simple hand-drawn shapes,” in *UIST ’00: Proceedings of the 13th annual ACM symposium on User interface software and technology*. New York, NY, USA: ACM Press, 2000, pp. 73–80. [cited at p. 13, 14]

- [27] D. W. Eggert, A. Lorusso, and R. B. Fisher, “Estimating 3-D rigid body transformations: a comparison of four major algorithms,” *Machine Vision and Applications*, vol. 9, pp. 272–290, 1997. [cited at p. 35]
- [28] A. Hunt and D. Thomas, *The pragmatic programmer: from journeyman to master*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999. [cited at p. 42]
- [29] D. Reilly. (2006, June) Top ten errors java programmers make. Last accessed 25 November 2006. [Online]. Available: <http://www.javacoffeebreak.com/articles/toptenerrors.html> [cited at p. 43]
- [30] M. Oltmans, C. Alvarado, and R. Davis, “ETCHA Sketches: Lessons learned from collecting sketch data,” in *Making Pen-Based Interaction Intelligent and Natural*. Menlo Park, California: AAAI Fall Symposium, October 21-24 2004, pp. 134–140. [cited at p. 55]
- [31] R. O. Duda, P. E. Hart, and D. G. Stork, *Pattern Classification (2nd Edition)*. Wiley-Interscience, 2000. [cited at p. 63]
- [32] P. Bourke. (1989, Apr.) Intersection point of two lines. Last accessed 25 November 2006. [Online]. Available: <http://astronomy.swin.edu.au/~pbourke/geometry/lineline2d> [cited at p. 67]
- [33] ——. (1988, Oct.) Minimum distance between a point and a line. Last accessed 25 November 2006. [Online]. Available: <http://astronomy.swin.edu.au/~pbourke/geometry/pointline> [cited at p. 67]

Appendices



Appendix A

Classifier Tutorial

This section gives a brief introduction to linear classifiers and the motivation behind the use of Gaussian distributions. This knowledge is needed to understand the methods employed in this study. For an excellent introduction to the pattern recognition methodology see [31].

In mathematics and computer science a classifier can be described as a mapping from a (discrete or continuous) feature space to a discrete set of labels. If an object can be represented as a vector of feature-values, various methods exist that can be used to “label” the object as of a certain class.

A.1 Linear classifiers

Assume one has the task of creating a system that differentiates between apples and oranges. Further assume one can obtain data from two sensors: one for the weight and one for the colour of a single fruit. To complicate things assume the apples are reddish-yellow¹.

Fig. A.1 shows data that was collected from a few samples of the fruit.

¹*Cripps Pink* cultivar

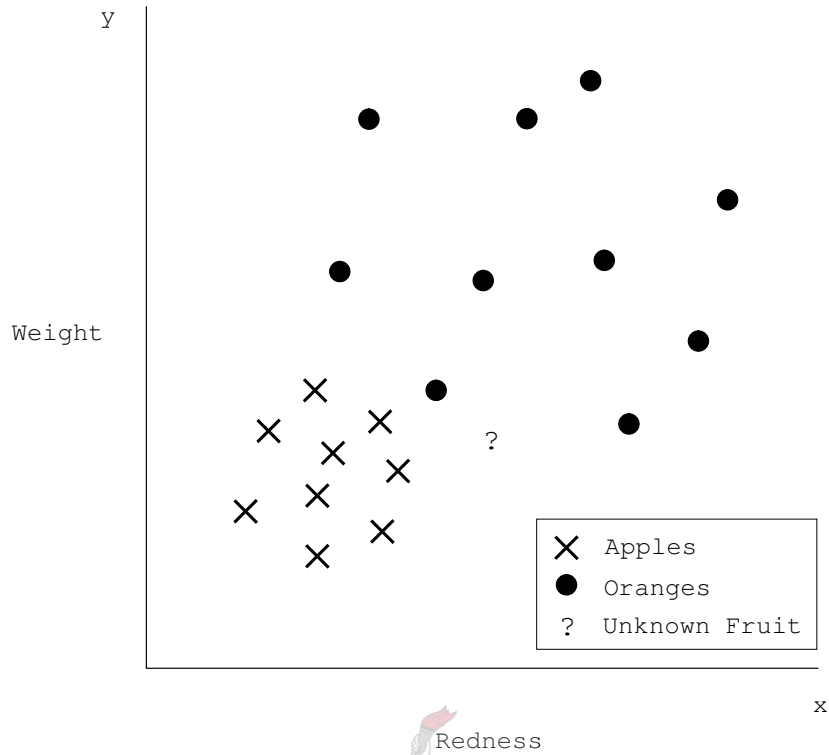


Figure A.1: Measurable features of Apples and Oranges

Identifying an unknown fruit can be done in a number of ways. A simple method would be to determine the euclidean distance of the feature representing the fruit to the mean point of the two classes and labelling it to the closest class. Equations (A.1), and (A.2) would then represent the distance metrics for the classes Apples and Oranges, respectively.

$$\begin{aligned}
 F_A(\langle x, y \rangle) &= \frac{1}{\text{distance}(\langle x, y \rangle, \langle \bar{x}_A, \bar{y}_A \rangle)} \\
 &= \frac{1}{\sqrt{(x - \bar{x}_A)^2 + (y - \bar{y}_A)^2}}
 \end{aligned}
 \tag{A.1}$$

$$\begin{aligned}
 F_O(\langle x, y \rangle) &= \frac{1}{\text{distance}(\langle x, y \rangle, \langle \bar{x}_O, \bar{y}_O \rangle)} \\
 &= \frac{1}{\sqrt{(x - \bar{x}_O)^2 + (y - \bar{y}_O)^2}}
 \end{aligned}
 \tag{A.2}$$

Using this technique to classify the feature vector $\langle x_U, y_U \rangle$ of the unknown fruit, indicated in Fig. A.1, would yield a value of $F_A(\langle x_U, y_U \rangle)$ larger than $F_O(\langle x_U, y_U \rangle)$. The unknown fruit would thus be labelled as an apple.

A.2 Gaussian distributions

To the human mind, classification of the unknown fruit in Fig. A.1 as an apple seems wrong. The examples for the apples are much closer to their average than the oranges are to theirs. This variance from their average suggest that orange measurements are more distributed and the unknown is more likely to be an orange, as the apples have a small variance.

Variances from means should therefore also be considered in the classification process. Variances from measurements in the real world usually follow normal or Gaussian distributions.

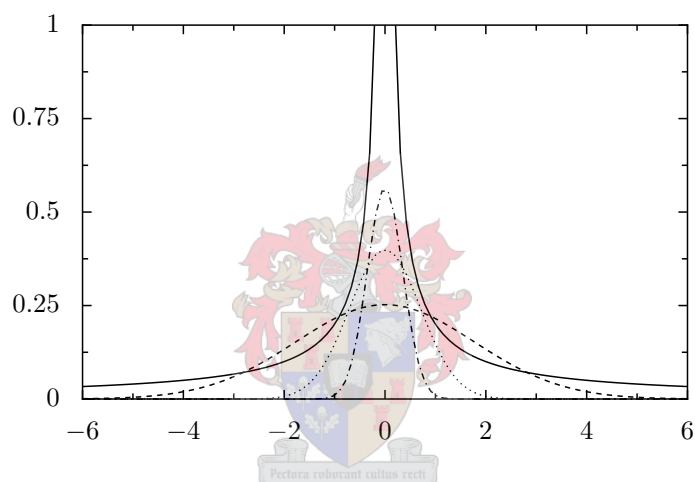


Figure A.2: Euclidean and Gaussian distance metrics

$$G(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-(x-\bar{x})^2/2\sigma^2} \quad (\text{A.3})$$

Equation (A.3) shows the Gaussian function where $\sigma > 0$ and $-\infty < \bar{x} < \infty$ are real constants. x , \bar{x} and σ represent the feature value, average feature value from training data and the variance of training data. Fig. A.2 shows the Gaussian probability density function compared to the reciprocal of the euclidean distance.

The variance parameter in the gaussian function can easily be used to take into account the variances of the fruit features.

A.3 Dimensional variance

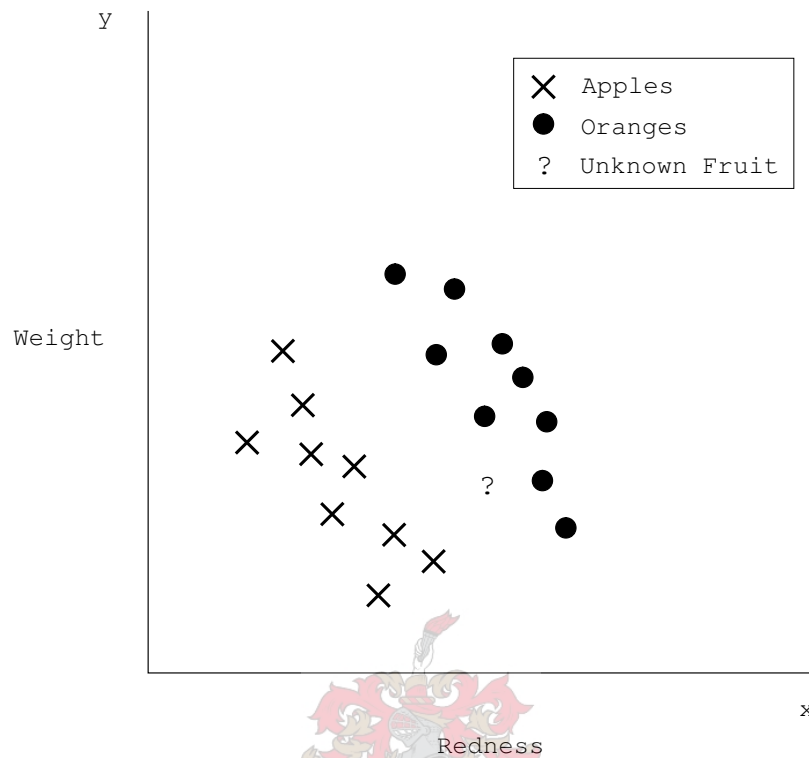


Figure A.3: Measurable features of Apples and Oranges

Fig. A.3 demonstrates a different issue that needs to be addressed by a classifier: The data variance can differ over different dimensions. A simple way to compensate for this is to derive a different deviation for each of the dimensions.

It should be noted that variances do not always differ strictly along dimensional lines. In these cases the co-variances (variances that are not exactly along dimensional lines) could also be leveraged. This is, however, beyond the scope of this simple introduction.

Appendix B

Intersection Analysis

The relatively straightforward mathematical algorithms used to find intersection information are explained in [32] and [33] and will only be summarised here.

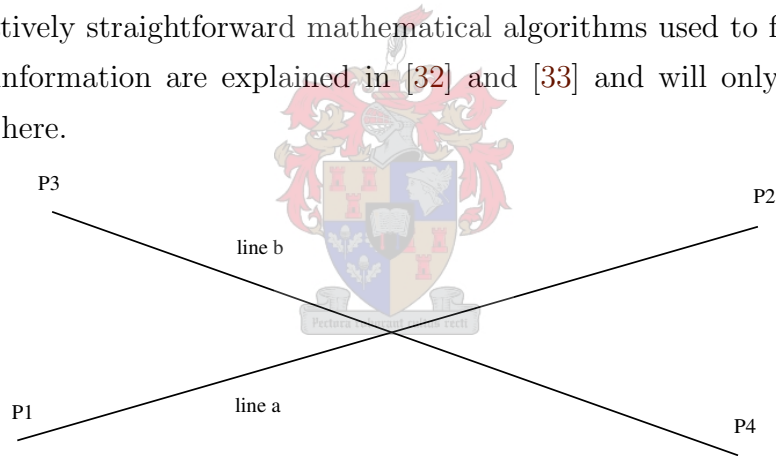


Figure B.1: Two line primitives intersecting.

Given two line segments, as pictured in Fig. B.1, points P_a and P_b that lie on the segments can be defined as

$$P_a = \langle x_1, y_1 \rangle + u_a(\langle x_2, y_2 \rangle - \langle x_1, y_1 \rangle) \quad (\text{B.1})$$

$$P_b = \langle x_3, y_3 \rangle + u_b(\langle x_4, y_4 \rangle - \langle x_3, y_3 \rangle) \quad (\text{B.2})$$

where u_a and u_b are thus scaling factors of the line segments.

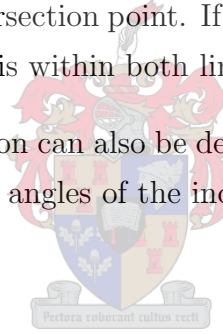
Solving for u_a and u_b where the points intersect (i.e. $P_a = P_b$) yields

$$u_a = \frac{(x_4 - x_3)(y_1 - y_3) - (y_4 - y_3)(x_1 - x_3)}{(y_4 - y_3)(x_2 - x_1) - (x_4 - x_3)(y_2 - y_1)} \quad (\text{B.3})$$

$$u_b = \frac{(x_2 - x_1)(y_1 - y_3) - (y_2 - y_1)(x_1 - x_3)}{(y_4 - y_3)(x_2 - x_1) - (x_4 - x_3)(y_2 - y_1)} \quad (\text{B.4})$$

Various properties on the intersection can then be derived:

- If the denominator for the equations for u_a and u_b is 0 then the two lines are exactly parallel.
- If the denominator and numerator for the equations for u_a and u_b are 0 then the two lines are coincident.
- If u_a or u_b lie inside the range $(0, 1)$ then the corresponding line segment contains the intersection point. If both lie within this range then the intersection point is within both line segments.
- The angle of intersection can also be determined by simply taking the difference between the angles of the individual lines.



Appendix C

Code Structure

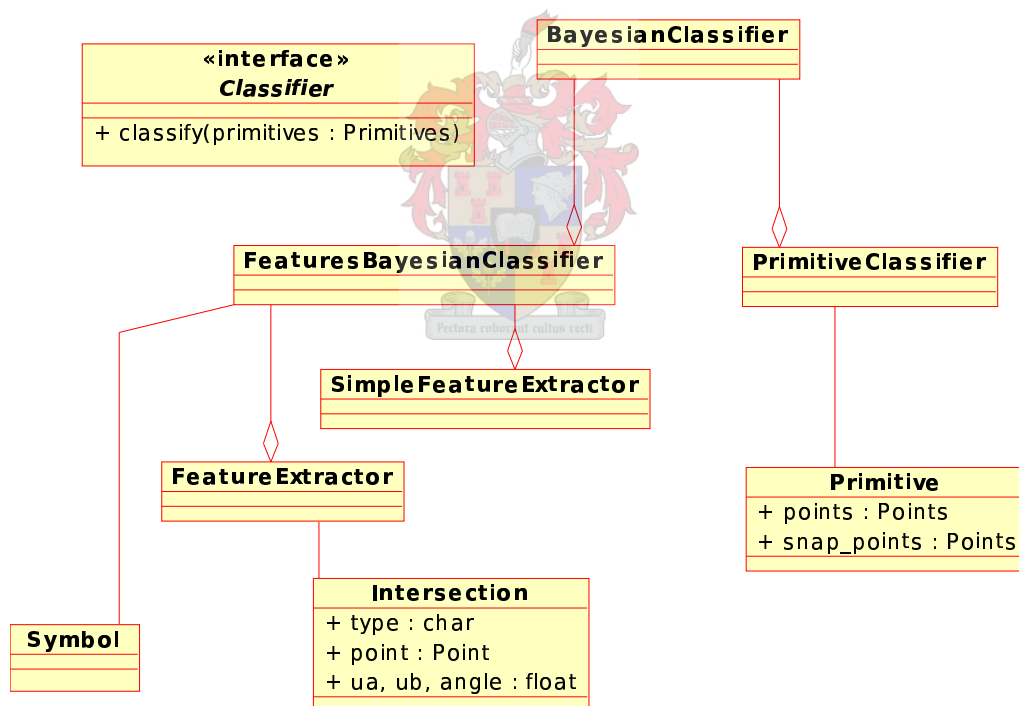


Figure C.1: Diagram of symbol related classes.

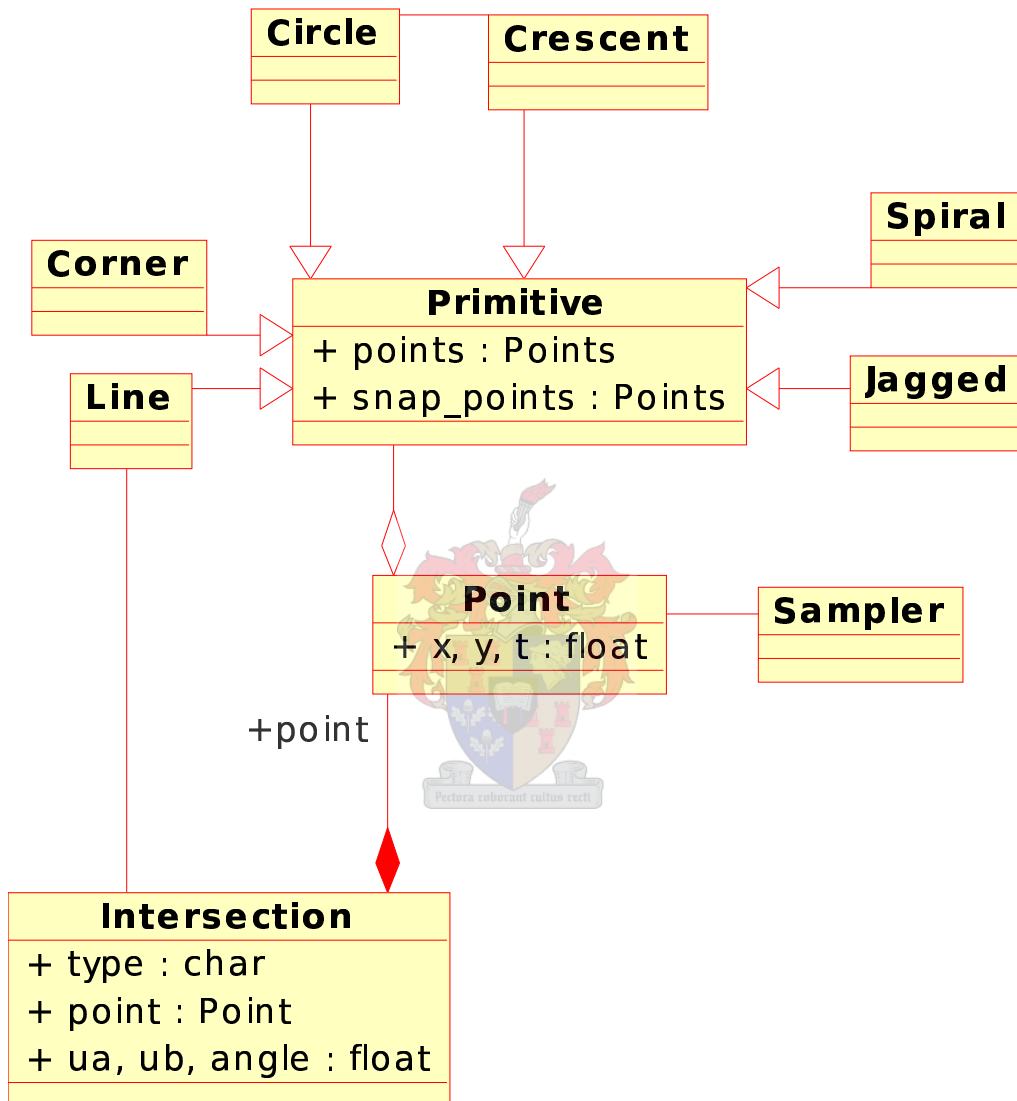


Figure C.2: Diagram of primitive related classes.

<root>	
bayesians.py	python source code
.	
.	
tools.py	
components	XML files representing symbol definitions
and.xml	
bjt.xml	
.	
.	
resistor.xml	
primitives	primitives in XML files
Circle	
001.xml	multiple examples of a circle in a single XML file
.	
.	
nnn.xml	
Corner	
.	
.	
Spiral	
pygraphlib	modified version of the pygraphlib library
reports	various reporting subdirectories
ar_graphs	generated attributed relational graphs
classifier_accuracy	various symbol classifier statistics
planar_graphs	generated planar graphs
primitive_accuracy	various primitive classifier statistics
testdata	examples of symbols
and	
001.xml	a single example of an AND gate in an XML file
.	
.	
nnn.xml	
.	
.	
resistor	
component_properties.xml	XML file with SPICE and port information of symbols

Figure C.3: A summary of the directory tree of the Python implementation. Folders are indicated in bold and italic.

Table C.1: Python source files (excluding modified libraries)

filename	line count	description
bayesians.py	301	Naive Bayesian classifier
circuit_space.py	353	Circuit management and SPICE generation
codestate.py	113	Profiler and code summary tools
event_recorders.py	223	Manages interface between interface and recognition engines
fc_interface.py	659	wxWidgets FloatCanvas interface
feature_graphs.py	119	Feature vector plotting tools
graph_comparison.py	345	Intersection graph matching
linetools.py	1071	Primitive construction and intersection tools
planar_graphs.py	186	Planar graph matching
recognition.py	47	Combined symbol classifiers
test_accuracy.py	224	Symbol recognition accuracy report generation
tools.py	592	Various algorithms used by system
Total	4233	