Investigation of Sketch Interpretation Techniques into 2D and 3D Conceptual Design Geometry

Sheng-Feng Qin, BEng., MSc.

A thesis submitted to the University of Wales for the degree of Doctor of Philosophy

February 2000

Director of Studies: Dr. David K. Wright
Second Supervisor: Dr. Ivan N. Jordanov

School of Product & Engineering Design
University of Wales Institute, Cardiff
United Kingdom
To my family

[Signature]

[Signature]
DECLARATION

This work has not been previously accepted in substance for any degree and is not being concurrently submitted in candidature for any degree.

Signed: [Signature]
Date: 28/02/2000

STATEMENT 1

This thesis is result of my own investigations, except where otherwise stated. Other sources are acknowledged by footnotes giving explicit references.

Signed: [Signature]
Date: 28/02/2000

STATEMENT 2

I hereby give consent for my thesis, if accepted, to be available for photocopying, for inter-library loan, and for the title and summary to be made available to outside organisations.

Signed: [Signature]
Date: 28/02/2000
ACKNOWLEDGEMENTS

Firstly, I would like to thank the supervisors of my study, Dr. D. K. Wright and Dr. I. N. Jordanov, for their encouragement, guidance and strong supports during the course of this work.

I also would like to thank the School of Product and Engineering Design, at the University of Wales Institute Cardiff, and the Department of Design, at the Brunel University, London, for offering the place of my study and the financial support.

At the University of Wales Institute Cardiff, I would like to thank Professor R. Brown, Mr. K. Roberts, Mr. T. J. Nam, and all the members of the School of Product & Engineering Design, for their support.

My hearty thanks go to my wife Sui-Ying Yang and my children Miao Qin and Deming Qin for their full support and understanding over the past years.

Finally, I would like to express my gratitude to my parents for their encouragement throughout my study.
ABSTRACT

This thesis presents the results of new techniques investigated for applying on-line sketching into 2D and 3D conceptual design geometry throughout a whole development process: data collection, concrete curve segmentation and fitting, 2D geometric constraint extraction and solver, and 3D feature recognition and modelling. This is a new approach. A real time sketch and fuzzy knowledge-based prototype system has been developed in four phases. In the first phase, the segmentation approach investigated accepts the input of on-line free-hand sketch, and segments them into meaningful parts, by using fuzzy knowledge in terms of sketching position, direction, speed and acceleration. During the second phase, a parallel curve classification and identification method is studied by employing fuzzy heuristic knowledge in terms of curve linearity and convexity, in order to quickly classify and identify a variety of 2D shapes including straight lines, circles, arcs, ellipse, elliptical arcs, and free-form curves. Afterwards, a geometric constraint inference engine and a constraint solver are utilised according to degrees of freedom analysis, to capture a designer’s intention, to infer geometric constraints simply and automatically, and to generate a possible solution without involving iterative computing. The solver also supports variational geometry in 2D and 3D. In the last phase, rule-based feature interpretation and manipulation techniques are investigated. While drawing, the 2D geometry is accumulated until it can be interpreted as a 3D feature. The feature is then placed in the 3D space and a new feature can be built incrementally upon previous versions.

The given examples and case studies show that the system can interpret users’ intention on 2D and 3D geometry satisfactorily and effectively. It can not only accept sketched input, but also users’ menu-based interactive input of 2D primitives and 3D projections. This mixed automatic feature interpretation and interactive design environment can encourage designers with poor sketching skills to use it for creative design tasks.
## CONTENTS

Chapter 1  Introduction

1.1 Computer Aided Conceptual Design  
1.2 Objectives  
1.3 Related work

1.3.1 Sketch Input and 2D Interpretation  
1.3.2 3D Reconstruction  
1.3.3 Inferring 3D models with Sketching  
1.3.4 Modelling with 3D Input Devices  
1.4 Structure of Proposed System

Chapter 2  Sketched Input and Fuzzy Knowledge-Based On-Line Curve Segmentation

2.1 Input of Design Sketches

2.1.1 Image Input  
2.1.2 On-Line Sketching

2.2 Introduction of Curve Segmentation

2.2.1 Off-Line Curve Segmentation  
2.2.2 Discrete Curvature  
2.2.3 On-Line Curve Segmentation

2.3 Data Collecting and Filtering

2.4 Fuzzy Knowledge-Based Segmentation

2.4.1 Types of Segmentation Points
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.4.2 Segmentation Procedure</td>
<td>35</td>
</tr>
<tr>
<td>2.4.3 Computing Directional Deviation</td>
<td>35</td>
</tr>
<tr>
<td>2.4.4 Finding Obtuse Corner Points</td>
<td>40</td>
</tr>
<tr>
<td>2.4.5 Detecting Acute Corner Points</td>
<td>40</td>
</tr>
<tr>
<td>2.4.6 Detecting Inflection Points</td>
<td>49</td>
</tr>
<tr>
<td>2.5 Case Studies</td>
<td>53</td>
</tr>
<tr>
<td>2.6 Discussion</td>
<td>59</td>
</tr>
</tbody>
</table>

Chapter 3 Parallel Classification and Identification of 2D Primitives

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Introduction</td>
<td>62</td>
</tr>
<tr>
<td>3.1.1 General Fitting</td>
<td>62</td>
</tr>
<tr>
<td>3.1.2 Split-and-Merge Based Fitting</td>
<td>63</td>
</tr>
<tr>
<td>3.1.3 Segmentation Based Fitting</td>
<td>63</td>
</tr>
<tr>
<td>3.2 Classification</td>
<td>66</td>
</tr>
<tr>
<td>3.2.1 Classification of Lines</td>
<td>66</td>
</tr>
<tr>
<td>3.2.2 Classification of Free-Form Curves</td>
<td>67</td>
</tr>
<tr>
<td>3.2.3 Classification of Conics</td>
<td>70</td>
</tr>
<tr>
<td>3.3 Identification and Generation of 2D Primitives and B-spline Curves</td>
<td>70</td>
</tr>
<tr>
<td>3.3.1 Generation of Straight Lines</td>
<td>70</td>
</tr>
<tr>
<td>3.3.2 Generation and Classification of Conic Curves</td>
<td>72</td>
</tr>
<tr>
<td>3.3.3 B-spline Fitting</td>
<td>79</td>
</tr>
<tr>
<td>3.4 Examples</td>
<td>82</td>
</tr>
<tr>
<td>3.5 Discussion</td>
<td>85</td>
</tr>
</tbody>
</table>
Chapter 4    Geometric Constraint Solver and 2D Geometry

4.1 Introduction

4.2 Related Work
   4.2.1 Equational Solvers
   4.2.2 Constructive Solvers
   4.2.3 Degrees of Freedom Analysis

4.3 Overview of Proposed Method

4.4 Constraint Inference Engine
   4.4.1 Type-1 Connections
   4.4.2 Type-2 Connections
   4.4.3 Type-3 Connections
   4.4.4 Type-4 Connections
   4.4.5 Unitary Relations
   4.4.6 Pairwise Relations

4.5 Analysis of Object Degrees of Freedom
   4.5.1 Definitions
   4.5.2 Object Degrees of Freedom
   4.5.3 Constrained Degrees of Freedom

4.6 The Constraint Solver
   4.6.1 General Construction Strategies
   4.6.2 Generation of Construction Steps

4.7 Geometry in 2D
   4.7.1 From Sketches to 2D Geometry
   4.7.2 Combination of Sketches with Interactive Input into 2D Geometry
4.7.3 Variational Geometry in 2D Geometry

4.8 Examples and Discussion

Chapter 5  Feature Interpretation and 3D Geometry

5.1 Introduction

5.1.1 Feature Recognition

5.1.2 Design with Features

5.2 Overview of Proposed Method

5.3 Feature Design

5.3.1 Feature Types

5.3.2 Internal Representations of Features

5.4 Inference Engine in 3D

5.4.1 General Interpretation Assumptions

5.4.2 Inferring Features

5.4.3 Finding Closed Profiles and Extrusion Edges

5.4.4 Determining Extrusion Directions

5.4.5 Looking for Reference Planes

5.4.6 Discussion on Computation of Rotation Angles

5.4.7 Computing Modelling Parameters

5.5 Menu-Based Interactive Input of 3D Object Projections

5.6 Variational Geometry in 3D

5.7 Examples and Discussion

Chapter 6  Case Studies and Conclusions
Chapter 1 Introduction

1.1 Computer Aided Conceptual Design

Design activity plays a core role in a product development process (PDP), which is illustrated by total design activity model shown in Figure 1.1. This model is widely accepted as a basis for teaching design [Pugh 1990].

![Total design activity model](image)

Figure 1.1 Total design activity model

Conceptual design is at an early stage of the design process with characteristics of fuzzy problems, tolerating high degrees of uncertainty and vague ideas. Decisions
made during this stage have significant influence on the cost, performance, reliability, safety and environmental impact of a product. Even the highest standard of detailed design can never compensate for a poor concept. Early design ideas are represented as 2D and 3D sketches, and are vague and imprecise [Tovey 1989, 1997], [Eisentraut 1997]. During this stage, designers search for structures, generate ideas, and try out different solutions, while at the same time these activities are guided by sketching out ideas [Van-Dijk 1995]. Thoughts come to mind as designers view a drawing or model in progress, which can alter their perceptions and suggest new possibilities. The emerging representation allows them to explore avenues that could not be foreseen, and ideas are generated along the way [Dorsey 1998]. Currently, there exists little computational support for the early stages of geometric design. Various CAD systems, such as advanced parametric and feature based systems, have been developed to support the 2D drafting and 3D modelling of products, but they usually require complete, concrete and precise definitions on the geometry which are only available at the end of the design process. Also, conceptual designers still tend to prefer paper and pencil to CAD systems for effective expression, communication and recording of new ideas. The reasons for this include the low overhead of a single-tool interface (pencil), the lack of special knowledge needed to draw, the ease with which many kinds of changes can be made, and the fact that precision is not required to express an idea. Pencils are cheap. Nevertheless, pencil and paper are still imperfect. After many changes, the paper can become cluttered. Drastic alterations such as showing the model from different viewpoints require new drawings, and collections of drawn objects cannot be transformed as a unit. In contrast, computer models do not have these disadvantages [Zeleznik 1996].
To support the early stage of geometric design and to improve the speed, effectiveness and quality of the design decision, studies [Eggl 1997], [Hwang 1994], [Lipson 1995, 1996] indicate that Computer Aided Conceptual Design (CACD) systems must allow sketched input, have a variety of interfaces, recognise features, and manage constraints.

1.2 Objectives

This research is aimed at developing a sketch based CAD system interface for assisting designers during conceptual design stages. It captures the designers’ intention and interprets the input sketch into more geometrically exact 2D vision objects and further 3D models. It could also allow designers to specify a 3D object or a scene quickly, naturally, and accurately.

The research objectives are

- To compare and evaluate alternative approaches for on-line sketching, recognition and interpretation;
- To investigate and implement techniques in conceptual design in both 2D and 3D geometry;
- To evaluate the prototype system;
- To develop C/C++ software, which will enable the freehand drawing system to be used on both, UNIX and MS-Windows based platforms.

1.3 Related Work

To allow the input of sketches to a geometric modeller, sketch interpretation and recognition must be supported. A variety of techniques to support sketched input, 2D interpretation and recognition, and 3D reconstruction have been developed.
1.3.1 Sketch Input and 2D Interpretation

Designer’s sketches are imprecise in geometry, with non-identical line thickness, and various shadows and textures. Furthermore, designers range from new graduates to very skillful design engineers, and have different sketching skills, such as sketching speed and geometry accuracy. The problem of sketched input and its interpretation into basic 2D primitives such as line, arc, circle, ellipse, ellipse arc, rectangle and curve is important and challenging. Note that sketch interpretation here is limited to only geometry sketches, and does not include interpreting on-line handwriting and textures.

The first interactive CAD system developed by Ivan Sutherland [1963] at MIT allowed sketched input and included a recognition system. It was able to get input from an image drawn on the screen with a lightpen, and then to use the sketch to perform a variety of analyses. During the 1960s, and 1970s, the main focus of attention moved from sketch-based systems to traditional CAD systems. With the development of computer vision, artificial intelligence, and design studies, sketch input and recognition received attention again in the 1980’s. Recent literature [Chen 1996], [Maki 1997] shows that fuzzy logic and knowledge-based methods are quite effective for on-line sketching and its 2D interpretations and recognition.

A mouse-based drawing system using a fuzzy logic concept has been developed by Chen and Xie [1996]. The system used fuzzy membership functions for evaluating sketching manners, e.g., changes of sketching direction are large, and the sketching speed is very slow. The system is a real time shape recognizer and generator. It can infer human drawing intentions and generate the corresponding geometric primitives such as line, circle, circular arc, ellipse, elliptic arc and smooth B-spline curves. A
A freehand curve is created by pressing a button, moving the mouse while the button is still pressed and then releasing the button. Each freehand curve is processed by a fuzzy processing engine. Fuzzification is the first procedure that fuzzifies each point. It calculates the degrees of membership functions of speed, acceleration, change of angle and change of vector for each point. A fuzzy filter checks each point, and filters out unintentional points with fuzzy inference rules. Since a hand drawing may consist of more than one curve segment, a fuzzy separator finds the turning points and divides the drawing into some subcurves. A fuzzy identifier checks all the possible points for each possible reference model, obtains the possibility value for the model, and uses a set of fuzzy inference rules to decide which model it belongs to. It may not belong to any model if it is just a free curve. A fuzzy generator finds the corresponding geometric representation of the curve, draws it and erases the original freehand drawing.

A fuzzy knowledge-based system for design feature recognition of freehand illustration [Maki 1997] has been developed for a Japanese sewing shop. It thus has been confirmed that the fuzzy knowledge-based method is effective.

### 1.3.2 3D Reconstruction

Reconstructing 3D object from line drawings interpreted from sketches is very interesting, since, it includes areas of machine vision, artificial intelligence and engineering. It is well known that a 2D point in a given projection plane can have an infinite number of corresponding 3D points on the projection line.

The reconstruction problem started to gain attention only in the late 1960s. Today it is considered an important research area. Much work has been devoted to reconstructing 3D information from a 2D projection including multi-views (top, front
and side view) [Nagendra 1988], and a single view: axonometric view [Wang 1993] or perspective view [Horad 1987]. For reconstruction from a multi-view, Wesley and Markowsky [1981] presented their wire frame algorithm: first 3D vertices are generated from 2D projected ones, 3D edges generated from 3D vertices, then 3D sub-faces generated from 3D edges, 3D sub-faces assembled to form 3D sub-objects, then the 3D sub-objects assembled to form the objects matching 2D projections given. Later researchers, such as [Sakurai 1983], [Yan 1994], [Masuda 1997], [Dori, 1996], focused on extending Wesley and Markowsky’s algorithm to introduce more curved objects with less restriction on their placement. In engineering applications, the above approaches provide complete solutions with either two or three views of the object. These methods aimed to produce a solid model, given a complete drawing of the target object which contains depth information, and concentrated on matching vertices between views, or just producing face information. However, my aim is to allow designers to input a quick sketch for just a single view. I do not require them to draw several views for expressing 3D design ideas.

Some works [Horad 1987],[Hale 1992], from the computer vision community, aim to reconstruct objects from line drawings extracted from a single perspective view image, rather than sketched by a user. Perspective projections are hard to sketch, and too error prone to be used for quick sketching by hand.

For reconstruction from a single isometric projection, two main methods, namely labelling schemes and optimisation approaches, have received considerable attention in the computer vision area [Wang 1993]. In the case of interpreting line drawings representing a single view of a 3D scene, there is still a very large gap between the human’s ability and that of the best existing computer programs. For the limited
problem of perfect line drawings of polyhedral scenes, there is an elegant collection of mathematical theories concerned with deducing 3D structure from 2D projections. Gurnan [68] first demonstrated that a few simple heuristics could be used to decompose a single drawing of a complex collection of polyhedral objects into its separate parts. Huffman [1971] and Clowes [1971] set forth the first labelling scheme valid for polyhedra. The Huffman-Clowes labelling method classifies line segments into three categories: convex edges, concave edges and occluding edges. In the trihedral world in which exactly three planes are assumed to meet at every corner, the possible combinations of line labels for each junction type can be catalogued in a dictionary. Given this labelling, there is only a finite number of ways in which lines can meet at a junction. Possible labelled-line configurations around a node are shown in Figure 1.2. Labels in Figure 1.2 can be concluded as follows:

(a) a “+” line represents a convex edge which has both of its corresponding planes visible from the camera;

(b) a “-” line represents a concave edge which has both of its corresponding planes visible from the camera;

(c) an arrow represents convex edges in the scene which have both of their associated planes on the same side of the edge as viewed from the camera, one hiding the other. An arrow is used with the convention that, as one moves in the arrow, the pair of associated planes it to the right.

(d) the identifying numbers are determined in accordance with the number of octants which are occupied by solid material at the vertex, since three planes which meet at a vertex partition the surrounding space into eight octants. the integer shown is the associated vertex-type. All four vertex-types are illustrated in Figure 1.3.
(e) top row in the Fig. 1.2 represents 'V' junctions where only two edges are visible; e.g., top junctions shown in Figure 1.3.

Figure 1.2. Possible labelled-line configurations around a picture node

Figure 1.3 Four vertex types

It is convenient to call each junction either a 'V', 'W', or 'Y' junction. The labelling scheme is based on line labels and junction library to interpret line drawings. The proper interpretation of the configuration of lines incident at a given node cannot
be determined, unless the surrounding ‘context’ is taken into consideration. Figure 1.4 gives an example of labelled drawings.

Figure 1.4 Labelled drawings

Mackworth [1973] introduced a new representation called “gradient space”, a 2D space in which every point represents the slope of some family of parallel planes. He took special advantage of the fact that if the axes of the gradient space are aligned with the x, y axis of a picture, then the line in gradient space connecting the two points corresponding to the slopes of two intersecting planes will be normal to the line of intersection of these planes in the picture. Mackworth showed that the gradient space constructions could be used to generate improved interpretations of line drawings in terms of the Huffman-Clowes labels. Waltz [1972], Kanade [1980], Sugihara [1982, 1984, 1986], and several others [Wang 1993], [Grimstead 1995] extended the Huffman-Clowes labelling scheme based on junction libraries and gradient space. Waltz extended line labelling to permit handling drawings containing shadows, cracks, missing and accidental alignments of edges, and non-trihedral
vertices. Kanade explicitly exploited the planarity of the faces of blocks world and “Origami” objects (by employing a ‘gradient space’ representation) to accomplish a form of semi-quantitative recovery. In addition to consistent edge labelling, he could also constrain the relative orientation of the faces of the target 3D model. In an important sense, Kanade shifted the shape from line drawings paradigm away from qualitative description, toward quantitative reconstruction of 3D shape. Sugihara reformulated the realizability and recovery problems for line drawings of polyhedra (both with and without hidden lines removed) in purely algebraic terms. He required as input a specification of vertexes defining each of the individual planar faces of the polyhedra, and also required the implied line drawing to be a general position projection of the polyhedra. With this approach, he succeeded in providing an algebraic criterion as a necessary and sufficient condition for a line drawing to represent a physically realizable polyhedral object. He could also constrain the space of feasible solutions, and obtain a unique solution if enough additional constraints were provided. These additional constraints were obtained from information beyond that provided by the line drawing (e.g., shading or texture information). Lamb and Bandopadhay [1990] presented a system for interpreting a 3D object from a rough line drawing. Their system uses heuristic rules plus labelling information. The idea is to provide CAD designers with the capability of interactively viewing the most likely interpretation from currently available drawing, which may be rough, and to make changes if it is incorrect. This approach provides a good user interface for CAD designers to view 3D objects, but it suffers from its use of heuristic perceptive rules. These rules are true in many cases, but not always. This approach is composed of six steps:
Step 1: Convert the input into an adjacency graph;

Step 2: Apply a labelling to each vertex and edge in the graph;

Step 3: Select the "best" region for coordinate assignment;

Step 4: Select a reference junction for the region;

Step 5: Assign coordinate values to all junctions;

Step 6: Repeat steps 2-5 until all visible junctions have been assigned coordinates;

Finally, a wireframe model of an object will be formed.

In general, labelling methods require a hidden-line removed 2D view of a 3D object, and are not suitable for handling inaccurate drawings and possible missing entities [Lipson 1996].

An optimisation approach requires a complete wireframe drawing as input, and the use of an optimisation method gradually assigning the depth of each vertex from an initially flat drawing to a 3D wire-frame and minimising the standard deviation of angles between connected lines, as in the work by Marill [1991] and Leclerc [1992]. Marill focused on the original problem of human interpretation of single line drawings as 3D structure. He did not restrict his universe to block world objects, nor did he demand that the line drawings should be complete. He used an optimisation approach to find a solution. His algorithm consists of two components, an objective function and a simple descent optimisation procedure for finding a local minimum of this objective function. The objective function is simply the standard deviation of all of the angles (SDA) in the recovered 3D object with respect to their common mean. Marill calls this the minimisation of the SDA or MSDA principle. The input line drawing is specified as a set of points (vertexes) and lines; each point is represented by an \((x, y)\) coordinate pair, and each line is represented by an integer pair.
corresponding to the sequence numbers of the two points it joins. The representation of the recovered 3D object involves supplying a third (z) coordinate for each of the originally specified points. This is the orthographic extension of the line drawing. The result is actually a wire frame, rather than a solid object. To evaluate the objective function, every pair of lines terminating on a point (as defined in the input specification) is considered to form a separated angle. Note that the intersection, between two lines that happen to cross at intermediate points of their extent in the line drawing, is not treated as a vertex, and does not contribute to the objective function (even if the lines were to lie in the same plane in the 3D reconstruction). Similarly, two distinct vertexes can have the same (x, y) coordinates in the line drawing, but the line segments terminating on the distinct vertexes do not interact to form angles (even if the vertexes coincide in the 3D reconstruction). Thus, given a line drawing with n vertexes, each possible orthographic extension is represented as a z vector having n components; the corresponding angles and SDA are computed to evaluate the proposed solution. Marill uses a descent technique to search for a best answer, recognising that this is simply heuristic and that this approach will find only a single local minimum of his function. The input object has all of its z values initially set to zero; that is, it is a flat object lying in the (x, y) plane. At each stage of the search, the SDA of the current z vector is computed and then the program looks at the children of the current vector. These 2n children are all vectors one step size away from the current vector, and are formed by both, adding and subtracting a specified step value to each of the n components in the current vector. The value of the SDA is computed for each of these 2n children, and the child with minimum SDA is selected as the new current vector. This process is repeated until no improvement in the SDA is obtained,
and the resulting vector is returned as solution for the first of the three rounds of descent. Each additional round uses a smaller step and begins with the result obtained from the previous round. Leclerc [1992] critically examined the merits of Marill’s algorithm and improved the performance of this approach, by modifying Marill’s objective function to explicitly favour planar-faced solutions, and by using a more competent optimisation technique.

Lipson and Shpitalni [1995,1996] extended Leclerc and Marill’s work by identifying and formulating geometrical regularities and seeking their associated 3D configuration, and presented an optimisation-based algorithm for reconstructing a 3D model from a single, 2D edge-vertex graph. The graph, which serves as input for the reconstruction process, is obtained from an inaccurate free-hand sketch of a 3D wire frame object. The system has a number of assumptions:

(1) The input to the system consists of a single 2D line drawing only, which is given as a graph of connected entities;

(2) The input projection represents a wireframe model of a generated object that may be manifold, non-manifold for an assembly of such objects;

(3) The projection is drawn from a general non-accidental viewpoint that reveals all edges and vertices. That is, none of the edges or vertices coincide accidentally, and none of them accidentally appear to be joined in the projection;

(4) All drawn lines and curves in the projection represent real edges, silhouette curves or intersections of faces in the 3D object;

(5) The sketch is assumed to depict the object in a parallel (or nearly parallel) projection.

The procedure of the algorithm includes three steps:
(1) The initial sketch strokes are smoothed, classified into geometrical entities and then linked together at their end-points to form a projected topological edge-vertex graph. The original sketch is obtained from an on-line sketching interface, and it is assumed that each edge is drawn as a continuous sketch stroke;

(2) Faces of the depicted 3D wireframe object are determined in the 2D topological graph;

(3) Reconstructing a 3D object using implicit 3D information originated from three sources: image regularities, face topology, and statistical configuration of entities. Image regularities are special geometrical relationships between individual entities or within groups of entities. The image regularities are based on heuristic rules, e.g., the heuristic rule for parallelism regularity is that if two lines are parallel in the sketch plane, they probably represent parallel lines in the 3D object, although mathematically this is not necessarily the case. A variety of image regularities are employed in this system in terms of face planarity, line parallelism, line verticality, isometry, corner orthogonality, skewed facial orthogonality, skewed facial symmetry, line orthogonality (termed MSDP for minimum sum of dot products), minimum standard deviation of angles (MSDA), face perpendicularity, prismatic face, line collinearity and planarity of skewed chains. For each regularity, the corresponding weighting coefficient is computed according to its mathematical formulation. A 3D configuration can be represented by a vector \( Z \) containing the z coordinate variable of the vertices. A compliance function \( F(Z) \) can be computed for any 3D configuration by summing the contributions of the regularity terms. Regularities are prefixed by a global balancing coefficient vector \( W \). This process of manipulating the z coordinates
while seeking the best reconstruction is a $n$-dimensional non-linear optimisation, where $n$ denotes the number of vertices. This method supports a wide scope of general (manifold and nonmanifold) objects containing flat and cylindrical faces. But it cannot support free-form surface objects. The final compliance function to be optimised takes the form

$$F(Z) = W^T \sum[\alpha]$$

Where

$$[\alpha] = \begin{bmatrix}
  \alpha - \text{planarity} \\
  \alpha - \text{parallel} \\
  \alpha - \text{vertical} \\
  \alpha - \text{isometry} \\
  \alpha - \text{corner} \\
  \alpha - \text{skewed orthogonality} \\
  \alpha - \text{skewed symmetry} \\
  \alpha - \text{minimum sum of dot products} \\
  \alpha - \text{minimum standard deviation angles} \\
  \alpha - \text{perpendicular faces} \\
  \alpha - \text{prismatic} \\
  \alpha - \text{collinear}
\end{bmatrix}$$

These optimisation-based approaches do not take into account drawing errors, nor do they attempt to tidy up the drawings.

1.3.3 Inferring 3D Models with Sketching

In 3D design, reconstructing 3D objects from an arbitrary 2D input is in general too ambiguous. So, some experimental systems like ‘Design Capture System’ [Hwang 1994], IDeS [Branco 1994] and SKETCH [Zeleznic 1996], ‘Quick-sketch’[Eggl 1997], are based on perceptual analysis and constraints to infer 3D feature based models from 2D sketching.
Hwang and Ullman [1994] developed a design capture system, which has two phases: 2D stroke recognition and 3D feature recognition. In the first phase, sketched strokes are interpreted as lines, arcs, circles, ellipse, etc. These primitives are accumulated, until they can be recognised as a 3D feature. New features can be built upon previous ones. Their system still uses some junction features such as “ARROW HEAD” and “DUCK_CLAW” to inference a box feature. They did not employ a general modelling feature of an extrusion object in terms of a closed profile with an extrusion edge. Thus, their system has difficulty in applying an inference knowledge method for a box structure to a general combined extrusion object. The weakness of the system is that it recognises a finite number of features (actually 2 features): box and cylinder. To construct a complicated design, a larger number of features are required.

The IDeS prototype system allows the creation of sketched 3D models by drawing them directly, or by combing drawings with common solid modelling techniques. It tries to combine simplicity and intuition, while visual thinking and drawing with the useful features of the solid modeller, like Constructive Solid Geometry (CSG) operators. However, this system is limited by a menu-oriented interaction style and does not consider constructing and editing full 3D scenes.

The SKETCH system developed by Zeleznik [1996] attempts to bridge the gap between hand sketches and computer-based modelling programs, combining some of the features of pencil-and-paper sketching and some of the features of CAD systems to provide a lightweight, gesture-based interface to “approximate” 3D polyhedral modelling. SKETCH uses a gestural mode of input in which all operations are available directly in the 3D scene through a three-button mouse. The user sketches the
salient features of any variety of 3D primitives and, following four simple placement rules, SKETCH instantiates the corresponding 3D primitives in the 3D scene. SKETCH allows both geometry and the camera to be gesturally manipulated, and uses an automatic grouping mechanism to make it easier to transform aggregates of geometry. SKETCH can create and edit 3D models. However it has many flaws, many of the gestures being based on an ad hoc trial and error approach, some of which do not satisfy users. For example, two parallel lines drawn in the same direction create a cylinder, and two non-axis aligned lines that do not meet at a point create a truncated cone. They seemed too far from sketching a cylinder and a truncated cone in a natural way. It is also difficult for the users to memorise the gesture rules and object-generating rules.

Eggli, Hsu, Bruderlin and Elber [1997] described a ‘Quick-sketch’ system, which is a 2D and 3D modelling tool for a pen-based computer. They developed specific drawing techniques that have an unambiguous interpretation in 3D. These techniques are partly adaptations of conventional 3D techniques for a sketch-based environment, in combination with the new 2D sketching and manipulation techniques. Their system can interpret 2D sketch into 2D lines, circles, arcs or B-spline curves, and build up geometric relationships and constraints. Then it infers 3D models (3D solid objects and B-spline surfaces) from 2D shapes and constraints. These objects may be refined by defining 2D and 3D geometric constraints. A graph-based constraint solver is used to establish the geometric relationships or to maintain them when manipulating the objects interactively. To model 3D objects by sketching, the system currently allows the following techniques: extrusion surfaces can be generated by sweeping a 2D profile along a straight line. Ruled surfaces can be defined between two sketched
curves. A sketched cross-section can be swept along a sketched curve, creating a sweep surface. A surface of revolution can be created by simply sketching two approximately symmetric silhouette lines. Lines and curves can be also sketched on planar faces of existing objects. In this way, features can be quickly added to the objects. It seems doubtful whether their system could successfully infer large complicated objects, because it does not interpret 2D ellipses and uses a vague projection co-ordinate system.

The direct input of depth, whilst creating the sketches, is also investigated. Fukui [1988] developed a system for transforming 2D data into 3D data face by face, referring to the geometry of connected faces that were transformed before. If there is only one, or no adjacent face, the viewing direction is referred to. Topologically, this sketching process is interpreted as a sequence of Euler operations, and, consequently, The topological consistency holds at any stage of the drawing process. In principle, curved shapes cannot be input directly by this method. Pugh [1992] proposed an algorithm which applies geometric constraint satisfaction to the labelling scheme to generate a three dimensional object. This system is a solid modelling system whose user-interface is based on interactive sketch interpretation. Interactive sketch interpretation lets the designer create a line drawing of a desired object while the system (Viking) produces a 3D object description. This description is consistent with both the designer’s line-drawing and a set of geometric constraints, either derived from the line drawing, or placed by the designer. Sketch interpretation divides the task of interpreting a line-drawing into two parts: finding a surface-topology and solving for a vertex geometry. The first part is done by generating surface-topologies that are consistent with the line-drawing until one acceptable for the user is found. The second
is done by using a geometric constraint solver to find a vertex geometry that satisfies a system of constraints either derived from the line-drawing and the proposed surface topology, or placed by the user. The surface topology and vertex geometry are combined to form a three-dimensional object description that is consistent with both the line drawing and the constraints. Sketching is traditionally done in only two dimensions. However, the user must specify the location of each vertex in three dimensions when sketching with Viking. This system produces solid models directly, but not in a natural way for the users. Furthermore, it seems unsuitable for the interpretation of large complicated drawings all at once.

1.3.4 Modelling with 3D Input Devices

There is a very different approach to constructing 3D models that requires 3D input devices as the primary input mechanism. Butterworth [1992] developed a system 3DM, which is a three-dimensional surface modelling program that draws techniques of model manipulation from both CAD and drawing programs and applies them to modelling in an intuitive way. 3DM uses a head-mounted display (HMD) to simplify the problem of 3D model manipulation and understanding. A HMD places the user in the modelling space, making three-dimensional relationships more understandable. However, 3DM has shown weakness in the area of constraints and models where traditional CAD and drawing programs perform well. For instance, 3DM has no way of keeping two polygons parallel, causing some models to appear irregular.

Among the modelling applications using 3D input devices, the system JDCAD is one of the latest and most advanced [Liang 1994]. In the system, the user is equipped with a pair of six degrees of freedom tracking devices. The system uses a tracking device ‘bat’ for 3D input and a head tracker to realise a display. JDCAD supports
creating 3D primitives and performing Boolean operations. The user can select an object type from a 3D menu and then create a bounding box with a 3D stroke. The primitive is centred at the middle of the bounding box and the parameters are derived from its size. With JDCAD the user can create imprecise 3D sketches quickly, but it seems to have limited support for constrained manipulation and for generating precisely dimensioned objects interactively.

Sachs [1991] presented a free form sketcher using two-handed interaction and 3D input devices. With their 3-Draw system 3D surfaces can be input quite naturally by holding a tablet in the left hand and a stylus in the right. Now, one can sketch a line with the stylus while the tablet is moved through space, thus creating a 3D spline. 3-Draw is very specifically tailored to rough surface sketching.

A New virtual reality-based sketching system has been developed by Deering, [1995], showing that the simple 2D sketch-draw paradigm can be extended to 3D. Using head-tracked stereo shutter glasses and desktop CRT display configuration, virtual objects can be created with a 3D wand manipulator directly in front of users. The entire system is controlled through the use of a 3D multilevel pie menu. The system also supports animation objects through the generalisation of grouping operations when elemental animation objects are grouped with static objects.

Stork and Maidhof [1997] developed a solid modelling system using a 3D Space mouse. They introduced the topological-context-based modification technique that uses degrees of the 3D mouse advantageously, for intuitive CAD-specific manipulations. This technique determines the way an object is modified during further interactions, based on the picked topological element and subsequent 3D gesture, considering local geometrical properties and object type.
These modelling approaches that require 3D input devices are out of our interest. The simple reason for this is that sketching with 3D input devices is not an natural or traditional way.

1.4 The Structure of Proposed System

In this thesis, the development of the proposed system for conceptual design is described. The system flow chart is shown in Figure 1.5. In the first phase, the system gets a sequence of input data from mouse button presses, mouse motion and mouse button release events. From this data, information about the speed, acceleration, direction, angle, and accumulative chord length is extracted. These information is used in the following processes to infer users drawing intentions, and then to filter unintentional and redundant points. During the segmentation phase, the input sketch is divided into several sub-curves by locating segmentation points (points connecting two meaningful sub-curves). In the third stage, each of the curve segments is classified and recognised. Then, the corresponding precise 2D primitives, or B-spline curves are identified and generated. At this point, 2D primitives can also be quickly inputted by selecting 2D menu. After that, a 2D inference engine is performed for extracting geometric constraints (connectivity, parallelism or perpendicularity). 2D geometry can be received then by a 2D geometric constraint solver, based on the degrees of freedom analysis. Finally, this 2D geometry (primitives and connections) is accumulated until it can be recognised as a 3D object or feature. The features are placed in 3D space and new features can be built upon previous ones.

![System flowchart](image-url)

Figure 1.5 System flowchart
This thesis first presents a profile of the developed system. In the Chapter 2, description of fuzzy knowledge-based sketch segmentation is given. Then, a parallel classification and identification of 2D primitives, or B-spline curves are studied in Chapter 3. Afterwards, the 2D inference engine and the solver are described in Chapter 4. 3D feature interpretation and 3D manipulation are presented in Chapter 5. Case studies and conclusion are shown in Chapter 6. Appendices give software structures. My research has produced a new CACD design tool by combining interactive 2D primitive and 3D projection input with on-line sketch 2D and 3D interpretation based on fuzzy knowledge.
Chapter 2 Sketched Input and Fuzzy Knowledge Based On-line Curve Segmentation

2.1 Input of Design Sketches

In general, there are two ways to enter sketch information into 3D reconstruction/interpretation systems. Designers can either sketch their idea on paper and then put the sketched drawings into a computer by scanning the drawing, or can draw on-line sketches.

2.1.1 Image Input

For the first method, images may be inputted by scanners either coloured or grey images. Surely colour could give a lot of cues on segmentation. Also, shadow and texture information could be used to provide some cues on segmentation in computer vision or image understanding areas. For simplicity, here, the images can be assumed as grey ones. These images can be transformed into other images with desirable properties by a number of low-level image processing techniques, such as filtering the image in order to remove noise, or at least to partially suppress it. Then, those images can be further treated by image segmentation techniques (or feature-extraction) to form boundary information, such as boundary curves represented by chain-codes. By definition, segmentation can fundamentally be regarded as a process of pixel classification. In low level image processing, segmentation is concerned with splitting up an image into meaningful segments (also called regions or areas), that is, separation of objects from the background. Segmentation is a basic requirement for the identification and classification of objects in a scene. This segmenting operation can be approached as either

- An edge-based method for locating discontinuities in certain properties in the image, or
- A region-based method of grouping of pixels according to certain similarities.
In the edge-based approach, the boundaries of objects are used to partition an image. Edge points that lie on the boundaries of an object must be marked, and they often are detected by analysing the neighbourhood of the point. The regions on either side of an edge point (i.e., the object and the background) have dissimilar characteristics. Thus, in edge detection, the emphasis is on detecting dissimilarities in the neighbourhoods of points.

In the region-based approach, all pixels corresponding to a single object are grouped together and marked to indicate that they belong to the same object. The splitting and merging of the points belonging to the same object from all other points are based on some criteria. Two very important considerations are spatial proximity and intensity similarity.

It should be emphasised that there is no single standard approach to segmentation. There are many different ways in which one can attempt to extract meaningful parts from a scene, based on different descriptions [Rosenfeld 1976]. One of the reasons that we do not have a general image understanding system is that a two dimensional image can represent a potentially infinite number of possibilities [Fu 1981]. Pavlidis [1977] commented that an image segmentation problem is basically one of psychophysical perception, and therefore, not susceptible to a purely analytical solution. Any mathematical algorithms must be supplemented by heuristics, usually involving semantics for the class of pictures under consideration.

After segmenting an image into meaningful boundaries, another low-level image processing technique, namely boundary tracking (edge tracking), can be used to reduce edges to unit thickness and to make a connection between two successive edge points. Finally, a boundary curve can be received as a set of sequence points (or digitised curve).

These low-level image processing techniques, such as filtering, segmenting, and boundary tracking will not be explained here, as they can be found in the related literature [Low 1991], [Davies 1997], [Kasturi 1991], [Fischler 1987], [Shirai 1987], [Nazif 1984].
During conceptual geometric design stage, this image input method can only enter final design results in the design sketch form for 3D reconstruction, since no designer likes to scan sketches stroke after a stroke, or even a feature after a feature. Thus, this input method cannot produce real time 3D recognition feedback to the designer, although this feedback is very useful for conceptual design. So, the image input method is not adopted in my proposed system.

2.1.2 On-Line Sketching

Alternatively, designers can draw on-line sketches to express their design ideas. In this method, information about drawing position, sequence (chain information), and dynamic parameters in terms of speed and acceleration can be received at real time. One stroke is represented by a set of sequence points (or digitised curve) with some dynamic parameters.

As for input devices, some researchers suggest that a stylus or drawing tablet should always be used for this kind of sketching, because the use of a mouse as a drawing implement seems a potential disaster, as the mouse is well known to interfere severely with freehand sketching. But others found that it is inconvenient to sketch on the tablet with a stylus, while the sketched object is displayed on the screen. Designers felt awkward because they had to watch the object on the screen while performing sketching on “remote” tablet [Hwang 1994]. So, another suggestion is to sketch directly on the screen with a light pen. Actually, sketching with a light pen may still have some inconvenience, for example, a moving hand between the screen and eyes would block the user’s vision.

For this prototype system, a conventional mouse is selected as input device as in references [Zeicznik 1996], [Chen 1996], [Jenkins 1992], [Shpitalni 1997].
For on-line sketching and interpreting, a two-phase system (sketched stroke classification and interpretation) is developed by Eggli, Hsu, Bruderlin and Elber [1997]. Their Quick-sketch system applies some mode-dependent preference functions to interpret types of shapes, which include lines, circular arcs, full circles and B-spline curves, on condition that each stroke represents one entity. In their system, one of the basic 2D primitives, the ellipse, was not included. Another two-phase system, which uses a conic equation for best-match fitting of sketch strokes is described in [Shpitalni 1997]. The result of this matching is a group of conic sections representing lines, arcs, elliptic arcs and hyperbolas without spline curves. In the system, user’s intentions in terms of speed, acceleration and stroke length is not considered at all; moreover, each stroke is assumed to correspond to a single entity (line/arc), which is not a general case in a variety of applications.

Three-phase systems (data filtering and curve segmentation, identification, and generation of primitives and B-spline curves) were established in [Jenkins and Martin, 1992], [Chen 1996]. The system [Jenkins 1992] first finds the corner points, and then employs lines and circles/arcs to describe sketches by applying constraints to enforce users’ intention. The other system [Chen and Xie, 1996] can infer users’ drawing intentions using the drawing speed, acceleration, the changes of angles and changes of vectors at each point, and can generate the corresponding geometric primitives and B-spline curves. This system first finds the corner points and divides the drawing into sub-curves by a fuzzy separator [Yager and Zadeh, 1992], then analyses the sub-curves with different reference models, obtains the corresponding possibilities and identifies the types of shapes. The reference models seem complicated and time-consuming without sufficient robustness for ellipses and elliptical arcs. Approaches in pen stroke recognition can also be found in the context of manuscript character recognition literature [Tappert 1990], [Li 1998].

My system is implemented on a PC, using Visual C++ and the Open GL graphics library. It is composed of four functional phases: data collecting and filtering; segmentation; classification and
identification; and generation (Figure 2.1). In the first phase, the system gets a sequence of input data from mouse button presses, mouse motion and mouse button release events. From this data, information about the speed, acceleration, direction, angle, and accumulative chord length is extracted. This information is used to infer the user drawing intentions, and then unintentional and redundant points are filtered. During the segmentation phase, the input sketch is divided into several sub-curves by finding corner points and inflection points. In the next stage, each of the curve segments is classified and recognised. Finally, the corresponding precise 2D primitives: straight lines, circles, arcs, ellipses, and elliptical arcs, or B-spline curves: spiral lines, spring lines, and general curves, are generated and displayed.

![2D recognition flowchart](image)

**Figure 2.1. 2D recognition flowchart**

### 2.2. Introduction of Curve Segmentation

To allow sketch input in more natural way, one stroke input that includes more than one geometric primitive is acceptable in our system. Thus, perfect segmentation of sketch strokes into straight lines and other sub-curves is a prerequisite for obtaining the best sketch recognition and interpretation, because errors in segmentation might propagate to feature extraction and classification.

From a general perspective, the task of grouping elements of a curve under a given criterion is usually called *curve segmentation*. Describing each sub-segment in a compact mathematical way is useful towards at least three goals:

1. feature extraction (shape descriptors);
2. data compression; and
2.2.1 Off-line Curve Segmentation

Off-line curve segmentation comes from feature extraction and matching in the fields of image processing and computer vision. A set of sequence points (a boundary curve) from low-level image processing can be further segmented (or represented) into several meaningful parts (2D sub-curves).

For the past two decades, many off-line algorithms based on polygonal approximation and dominant point detection have been developed, most of which use straight lines to approximate the edge pixels. The fit is made to reduce a chosen error criterion between the approximation and the original curve. In the iterative endpoint-fit algorithm [Ramer 1972][Pavlidis 1974], the first step is to make a straight line segment between the two farthest points on the curve. The perpendicular distances from the segment to each point on the curve are measured. If any distance is greater than the chosen threshold, the segment is replaced by two segments; one of each from a segment endpoint to the curve point where the distance to the segment is greatest. This process is iterated until all segments are within the threshold. A straight-line fit also can be constrained to grow within a radius around each data point [Pavlidis 1982]. The line segment is grown from the first point, and when further extension of the line segment causes it to fall outside the radius of a point, a new line is started. Kurozumi and Davis [1982] employed a minimax approach, in which the line segment approximations are chosen to minimize the maximum distance between the data points and the approximating line segment. Dominant point based approaches first detect critical points in curves, and then connect these points to represent a polygonal approximations [Teh 1989], [Wu 1993] and [Inesta 1998]. Dominant points are regarded as the local curvature maxima points, such as corners.
(an isolated curvature change, for which the tangent to the curve is discontinuous), inflection points (zero-crossing in curvature), and smooth join points (points of transition from zero curvature to a significant value). To find dominant points, some smoothing techniques (Gaussian smoothing or adaptive filtering) are adopted to reduce the noise, because the curvature is sensitive to noise [Saint-Marc 1989], [Inesta 1998]. However, the piecewise linear approximation of digital curves is scarcely useful for segmenting curves in a meaningful and compact form [Chen 1996], since common used circular and elliptical arcs are not represented properly. Several works have been published on the problem of segmenting planar curves into straight lines and conic arcs. The approaches can be briefly divided into two major categories: edge approximation, and break point detection. In the first method, curves are segmented into straight lines and conic arcs by repeatedly fitting and segmenting, based on some “goodness-of fit”. Albano [1974] published one of the first papers on curve segmentation by edge approximation, in which planar curves are segmented into straight lines and conic arcs. The edge approximation procedures employed in recent research are mainly splitting (or region decomposition) and merging (or region growing), and split-and-merge [Rosin 1989], [Ichoku 1996]. By generalisation and extension of Lowe’s technique [Lowe 1987], Rosin and West [1995] developed a nonparametric procedure for segmenting 2D Curves into combinations of various features, mainly straight lines and elliptical arcs. The procedure can be summarized as: (1) segment the curve into straight-line segments by first forming a binary tree with recursive subdivision of the curve, and then traversing the tree to select the best representation; (2) extend the line segments by combining with adjacent line segments; (3) replace some line segments by elliptical arcs; (4) extend elliptical arcs by combining with adjacent line segments and/or elliptical arcs; (5) refine elliptical fit. In break point detection, the break points are usually detected at curvature extrema or points of rapid curvature change. Chen, Venture and Wu [1996] developed a procedure for segmenting a planar curve into lines and circular arcs, in which the number of entities
(or break points) of the curve is given. This procedure can be divided into two stages: (1) to obtain a starting set of break points, and determine the approximation functions (lines and arcs) for the data intervals that are separated by the break points; and (2) adjust the break points until the error norm is locally minimised. The first stage is based on the detection of significant changes in curvature using the chain-code and differential chain-code techniques, and the second stage is an optimization curve line fitting scheme. Wan and Venture [1997] gave a procedure for segmenting planar curves, mainly the projected boundary contours of machined parts, into straight-line segments and elliptical arcs. The break points are divided into two types: corners and smooth points. The corners are detected by first applying adaptive smoothing the tangent orientation along the curve, then taking the derivative of the smoothed tangent orientation, and finally locating the high spikes on the derivative. The smooth joins are first roughly located by a dynamic focusing fitting technique and then refined by an adjustment algorithm. The dynamic focusing fitting technique holds one end of a curve segment (which is bounded by a pair of adjacent corners) fixes and scans it from the other end until it focuses on a component segment which fits either a straight line or an elliptical arc. This component segment is identified and the process is repeated in the same manner for the rest of the curve. In the refining stage, each smooth join is adjusted to the left, or to the right, point by point, until the measurement of goodness of the fit for the curve segment is optimised. Generally speaking, the edge approximation method leads heavy computation, whether the break point detection approach is sensitive to noise. Also, all the above methods are focused on either line-fit or combinations of lines and general elliptical arc fitting, which are difficult to be used for applications with free form curve descriptors.

2.2.2 Discrete Curvature

In digitised curve segmentation, a contour will be regarded as a sequence of points,

\[ C = \{ P_i = (x_i, y_i): i=1, \ldots, N \} \]
To find dominant points, or break points in the curve, the first problem in detecting a local maximum of the curvature is to establish a precise definition of a discrete curvature. In the Euclidean plane, the curvature $\tau$ of a 2D curve in point $P_i$ of it is defined as the rate of change of the angle $\psi$ of the slope of the tangent at $P_i$ versus the traversed arc length $s$:

$$\tau = \frac{d\psi}{ds}.$$ 

$\tau$ also can be computed if it is expressed in terms of derivatives of the functions $x(t), y(t)$.

Define

$$y' = \frac{dy}{dx},$$

$$y'' = \frac{d^2y}{dx^2}.$$ 

It is known that

$$\tau = \frac{y''}{(1+(y')^2)^{3/2}}.$$ 

Here, $\tau$ is called the continuous curvature. Nevertheless, for a digital curve, this formula cannot be directly applied, since slope changes are not arbitrarily small. To overcome this problem, some significant measurements [Rosenfeld 1973], [Rosenfeld 1975], [Freeman 1977][Teh 1989], [Inesta 1998] in accordance with the curvature concept are developed for digital (or discrete) curves. These measurements are called discrete curvatures. One of them is called as $k$-cosine curvature. It can be defined as the following:

If two $k$-vectors are defined at $P_i$ as

$$a_i = (x_i - x_{i-k}, y_i - y_{i-k}),$$

$$b_i = (x_i - x_{i+k}, y_i - y_{i+k}),$$

then, the $k$-cosine can be defined at $P_i$ as:
\[ \cos_k(P) = \frac{a_k \cdot b_k}{|a_k| |b_k|} \]

Here, \( k \) is a smoothing parameter \((k>1)\), called the supporting region length.

The problem of feature point detection and curvature computation in digital curves may also be considered as a scale-space problem [Mokhtarian 1986], [Asada 1986]. In this approach, the curvature as a function of the position is computed at multiple scales; these are convolved with a Gaussian function, and second derivative of the result is computed.

### 2.2.3 On-line Curve Segmentation

Although many off-line algorithms have been proposed, most of them are not accurate enough or fast enough to be used in on-line systems [Wan 1997]. On-line sketches have not only the properties of low-level images, but also have some dynamic features in terms of drawing direction, speed, and acceleration. One advantage of on-line curve segmentation over off-line segmentation is that dynamic information can be employed to assist the segmenting process. Another advantage is that there is close interaction between users and machines. The users can thus correct any recognition error immediately as it occurs. For on-line curve segmentation, there is lack of literature available. Although there are some papers on segmentation of on-line handwriting script, they mainly focus on applying the writing direction to recognise characters [Tappert 1990], [Li 1997, 1998]. Roughly speaking, one of the differences between on-line and off-line segmentation is that the problem in on-line segmentation is how to cope with large deformations, as in on-line character recognition [Berthod 1979], while the problem in off-line segmentation is how to cope with noisy pictures. For an on-line sketch, some systems [Eggl 1997], [Lipson 1996], [Zeleznik 1996] simply avoid this curve segmentation problem by assuming that one stroke is one entity, whereas others [Chen 1996], [Jenkins 1992] just roughly find corners. According to my observation, in paper and pencil-based sketches, designers often draw one stroke containing more than one entities. In my system, an intelligent and adaptive threshold segmentation technique is used on the basis of fully exploiting the
properties of dominant points and fuzzy heuristic knowledge in terms of sketching speed and acceleration.

2.3. Data Collecting and Filtering

While sketching, as input data we obtain a sequence of mouse positions from pressing a mouse button, moving the mouse while the button is still pressed and releasing it. The mouse positions are measured in the screen co-ordinate system with the lower-left corner as origin. A unit is a screen pixel. This sequence of data represents a freehand curve. In order to obtain the speed, I use a distance between two adjacent points, and a constant time interval for the machine capturing events. The time interval can be affected and adjusted by setting bauds, e.g., 9600, on communication ports. If I assume that the value of the time interval is unity, then the value of the distance can be used as a speed measure. Subsequently, the acceleration is received on the basis of the speed.

![Speed vs Time](image1)

(a) Speed cycle

![Acceleration vs Time](image2)

(b) Acceleration cycle

Figure 2.2. Speed and acceleration cycles

If a free hand sketch comprises several sub-curves, the drawing speed for each sub-curve will usually increase from zero to a normal steady speed, remain level and finally will decrease. This forms a speed cycle and a corresponding acceleration cycle for each sub-curve (Figure 2.2).

The drawing speed will decrease when sketching is approaching a segmentation point and then speed up just after the segment point. Thus, the density of sample points close to a segment point will be bigger than that in the middle of the sub-curves. When drawing is very slow, two
consecutive digitised points are very close, because of the finite pixel spacing. For this reason, a minimum distance threshold such as, 3.5 units, is chosen for the distance constraint between them. All points, which are within the minimum distance, should be eliminated. Nevertheless, if one applies this rule to the points near a segmentation point, one may miss some candidates for a segmentation point. To obtain these candidates and filter high density points, a knowledge-based filtering rule is applied. That is, if a distance between two successive points is less than the minimum threshold and the acceleration at current point is negative, the current point will replace the previous one, or will be eliminated.

This early filtering reduces the number of points to be restored and treated, and ensures that a vector between two filtered successive points can give a good approximation of the sketching direction. The points derived from this filtering are the input to other treatments. In this way, we obtain a sequence of points \( \{Q_j \mid j=1, 2, 3, ..., N\} \) with attributes of speed, acceleration and accumulative chord length. For convenience, the data collection is also denoted by \( \{Q_j\} \).

2.4 Fuzzy Knowledge Based Segmentation

2.4.1 Types of Segmentation Points

The segmentation points in sketch strokes can be divided into three types: obtuse corner points; acute corner points; and inflection points. An important property of a corner point is that the tangent to its contour is discontinuous. Inflection points have continuous tangents, but discontinuous curvature. Correspondingly, the directional deviation \( \theta \) between stroke vectors \( \mathbf{a} \) and \( \mathbf{b} \), shown in Figure 2.3 (a) and (b), features a dramatic change for corner points. For an obtuse corner point, \( \theta \) ranges from 90° to 180°. For an acute corner point, the deviation \( \theta \) is between parameter \( \theta_1 \) and 90°. The value of \( \theta_1 \) is set to 15°. A feature for recognising an inflection points is a change of curve convexity as shown in Figure 2.3(c), which in this example changes from counter-clockwise to clockwise.
2.4.2 Segmentation Procedure

My fuzzy knowledge based segmentation algorithm can be briefly described in 4 steps:

Step 1: compute directional deviation $\beta_i$ at point $i$ with an adaptive support region, based on k-cosine curvature measure [7], and perform nonmaxima suppression;

Step 2: find obtuse corner point; if $\beta_i > 90$ degree;

Step 3: detect acute corner points between two adjacent obtuse corner points by applying adaptive threshold and fuzzy knowledge, with respect of drawing speed, acceleration, and curve's linearity;

Step 4: determine whether a sub-curve between two neighbouring corner points, can be fitted with a straight line by computing its linearity. If yes, do line fitting, and then go to the next sub-curve. If not, detect inflection points within this sub-curve. If the sub-curve does have some inflection points, it will be treated as a B-spline, or it will be further classified.

2.4.3 Computing the Directional Deviation

A directional deviation $\beta_i$ at point $i$, ranging from 0 to 180 degrees, is defined as a dot product of two unit directional vectors:

$$\beta_i = \arccos (V_{i,i-m} \cdot V_{i,i+m}). (i = m, m+1, m+2,..., N-m),$$
where

\[ V_{i,-m} = (\mathbf{Q}_i - \mathbf{Q}_{i-m}) / \| \mathbf{Q}_i - \mathbf{Q}_{i-m} \|, \]

\[ V_{i,+m} = (\mathbf{Q}_{i+m} - \mathbf{Q}_i) / \| \mathbf{Q}_{i+m} - \mathbf{Q}_i \|. \]

and \( m \) is defined as a support length (number of points from the central point \( \mathbf{Q}_i \)). The theoretical value of \( \beta_i \) for lines is zero, and its value for curves is equal to that of an angle between two tangent lines at two adjacent points of the curve.

The deviation \( \beta_i \) is considered here only as depending on the relative positions over the support length \( m \), so, more robustness to noise effects and quantization is expected for this measurement, since \( m \) can be set locally and automatically. Obviously, if a fixed value of \( m \) is applied, it may be too large for some cases, missing fine details, or may be very small and more sensitive to noise. Moreover, each point will be evaluated independently of the characteristics of the whole contour, taking only local properties into account. To achieve this, it is necessary to make the support length determination automatically. The value of \( m \) can be given from a fuzzy function of a local average step length:

\[
m = \text{round} \left( 2 \frac{S_a}{S_i} + 0.5 \right);
\]

where \( S_a \) is a average speed (arc length), which equals the total accumulative arc length divided by the number of steps \((N-1)\); \( S_i \) is the speed at \( \mathbf{Q}_i \), which equals the average of step length between points \( i \) and point \( i-1 \), and step length between points \( i \) and point \( i+1 \). From this formula, \( m \) will be 1 if \( S_i \) is much larger than \( S_a \); and \( m \) will be 3, if \( S_a \) is equal to \( S_i \), which avoid taking the neighbouring step point to evaluate the angle change. The self-adjustment of this support length implies some advantages: (a) the algorithm works well for features of different sizes in the same contour, and (b), it works automatically. Otherwise, if the support length is set to be very large, the algorithm will miss fine details, and if it is very small, the algorithm will be more sensitive to noise.
After determining the support region \( m \) and, computing angle \( \beta_i \), system performs nonmaxima suppression as follows:

a) keep all original angles \( \beta_i \) for point \( i \) in an array ORG, which will be used in the next section;

b) 1\textsuperscript{st} pass: \texttt{for}\((j=1; j<N; j++)\) if \((p[j].\text{angle}\leq p[j+1].\text{angle})\) \( p[j].\text{angle}=0.0; \)

c) 2\textsuperscript{nd} pass: \texttt{for}\((j=n; j>1; j--)\) if \((p[j].\text{angle}\leq p[j-1].\text{angle})\) \( p[j].\text{angle}=0.0; \)

After the suppression, the angle for point \( i \) will be greater than zero, only if the point \( i \) is a local dominant point.

The adaptive support region, \( m \), varies quite stably. Figure 2.4 shows changes of \( m \) against accumulative arc length (\( S \)). In Figure 2.4, the vertical axis is scaled to 5 times \( m \); the horizontal axis represents the arc length, corresponding to individual points. Note that the diagonal lines are drawn by the system to assist designers sketching. On the upper-right side, there are original sketches and corresponding fitting. Comparing Figure 2.4 (a sketched elliptical arc) with Figure 2.5 (a sketched line), it is becoming clear that the changes of \( m \) are insensitive to the different types of primitives. Changes in \( m \) for different sizes (or scales) of primitives are insensitive either. It can be seen from Figure 2.6 and Figure 2.7, which show changes of \( m \) for a small, and a big arcs, respectively. On the other hand, change ranges for \( m \) are not big. From Figure 2.4 to Figure 2.7, \( m \) may ranges from 1 to 5. This property of \( m \) makes the angle detection stable.
Figure 2.4. Changes of $m$ over an arc

Figure 2.5 Changes of $m$ over a straight line
Figure 2.6. Changes of $m$ over a small arc.

Figure 2.7. Changes of $m$ over a big arc.

An example of angle detection and nonmaxima suppression is given in Figure 2.8. Original sketches and corresponding fits are given in the upper right side. The top graphics show the original computation of directional deviation. The bottom one gives the result of nonmaxima suppression.
After this suppression, all angles at nonmaxima points are assigned a value of zero. There are only 7 local maximum points left to be further treated. One obtuse corner point exists on the sketched input. Its direction change is greater than 90°.

2.4.4 Finding Obtuse Corner Points

When $\beta_i > 90°$, it means that sketching will make a “U” turn at point $i$. This geometric feature strongly suggests that point $i$ is a corner point. The first point and the last point in $\{Q_j\}$ are assumed to be obtuse corner points. Then, point $i$ is detected as an obtuse corner point when $\beta_i > 90°$.

2.4.5 Detecting Acute Corner Points

In this step, acute corner points are detected between two neighbouring obtuse corner points. An intelligent and adaptive threshold segmentation technique is applied, based on fuzzy heuristic knowledge, in terms of sketching skill, speed and acceleration, and curve linearity. An adaptive threshold for the directional deviation $\beta_i$ at point $i$ consists of three parts: basic threshold $\beta_0$, adaptive speed tolerance $\beta_s$, and adaptive linearity tolerance $\beta_l$. Point $i$ will be treated as an acute
corner point, if the angle $\beta_i > (\beta_0 + \beta_s + \beta_v)$, and the speed and acceleration at this point meet corresponding constraints.

The basic threshold $\beta_0$ reflects the drawing skills and pixel-based drawing features. We use an average angle between two adjacent obtuse corner points as $\beta_0$. When computing the average angle from the original angle array ORG, we do not count angles at the two obtuse corner points, because the angles at obtuse corner points are larger than the angles at middle points. Then, the adaptive speed tolerance $\beta_s$ is applied to imply the effects of drawing speed. According to our observations, the faster the sketching, the smoother the curve. In other words, the value of $\beta_i$ will give more accurate estimation of the true geometric parameters when drawing is fast. So, an adaptive speed tolerance $\beta_s$ should be a function of the drawing speed. The faster the sketching, the smaller the tolerance. As a result, we use a fuzzy function shown in Figure 2.9(a) to obtain $\beta_s$. That is: $\beta_s = (10-S_i)*2$; and, if $S_i > 10$, $\beta_s = 0$; $S_i$ is the average of the speeds at point $i$ and point $i+1$. The numbers of 10 and 2 are obtained from tests.

Another adaptive tolerance, that is considered, is the linearity tolerance $\beta_l$ because the theoretical value of $\beta_l$ is different for lines and curves. Thus, the tolerance $\beta_l$ should have a smaller value between two lines, than between one line and one curve, or between two curves. Before the sketches are interpreted, we use the linearity of a sub-curve to evaluate the possibility of being a straight line.
The linearity is the value of the distance between two points divided by the accumulative chord length between them. For example, if the linearity value is 0.98, there is a 98% possibility of the corresponding curve being treated as a line. To compute $\beta_l$, we employ a function shown in Figure 2.9(b) with product of $L_p * L_n$, where $L_p$ and $L_n$ are linearities for the curve segments before and after the point $i$, respectively. That is: $\beta_l = 10 + (1 - L_p * L_n) * 50$. The numbers of 10 and 50 come from experimentations.

Considering the speed and acceleration constraints, we found that when the drawing approaches a corner point, the speed is lower than, or equal to the average speed ($S_a$) over this sub-curve. The value of the speed also varies with the sketched objects. To capture these features, an adaptive speed constraint is included by using a fuzzy function with $\theta = S_a * L_p * L_n$, shown in Figure 2.9(c). For the acceleration, as shown in figure 2.2(b), sketching at the segmentation point should be with negative or zero acceleration. This forms the acceleration constraints.

As noted, all of the adaptive functions, shown in Figure 2.9, are triangular functions. This is because those functions can reflect heuristic knowledge simply, and have good computational efficiency.

Finally, we apply the following steps and rules to find the acute corner points:

1. Compute an average speed $S_a$ for current segment between two obtuse corner points;
2. Produce an adaptive tolerance $\beta_S$ and basic tolerance $\beta_0$;
3. Compute the linearities $L_p$ and $L_n$;
4. Determine an adaptive tolerance $\beta_l$;
5. Give angle threshold by $\beta = \beta_0 + \beta_S + \beta_l$;
6. Compute adaptive speed threshold $\theta$;
7. Judge if point $i$ with a suitable interval to adjacent corner points is an acute corner point by the conditions: $\{ (\beta_l >= \beta) \text{ and } \text{acceleration} <= 0 \text{ and } \text{speed} < \theta \}$.
Determination of the fuzzy functions is discussed below. Determining $\beta_0$ is first considered, based on Figure 2.10 to Figure 2.13, in which $s$ stands for arc length against angular deviation.

Figure 2.10  Digitised features of a straight line ($D = 20^\circ$)

Figure 2.11. Digitised features of a straight line ($D = 5^\circ$)
To examine $a$, a straight line equation is used for producing standard data. The line equation is given by

$$p = p_0 + D t, (0 \leq t \leq 150),$$

$$p_0 = (300, 200)$$

A sequence of points can be received by a for-loop computation by increasing $t$ from 0 to 150 with step length 1. The points represent a digitised straight line. This data, after initial filtering (threshold = 3.5), is sent to be checked for directional derivation. Theoretically speaking, the directional derivation should be zero for a straight line. But the derivation is usually $\neq 0$ for a digitised straight line, because a digitised curve is just an approximation of its corresponding mathematical curve, and it has some round-off errors. For example, when $D$ is a unit vector along $20^\circ$ direction, the graph of the directional derivation over its arc length can be obtained as shown in Figure 2.10. In this figure, the digitised straight line is shown to the right, the graph is displayed to the left.
changing D to 5° and 35° directions, the corresponding graphs are shown in Figure 2.11 and Figure 2.12, respectively. It is clear from these graphs that different directional straight lines after digitising will have different features of directional derivation in terms of distributions and sizes, although they have the same theoretical value of the derivation. Thus, an average of the derivation is selected as β₀ to enable it to be adapted to different line. It is believed that using the average as β₀ is better than using a fixed value, such as 15° as in other systems [Lee 1998].

Figure 2.13. Fast sketching along a straight line

Some experimental tests are also conducted for consideration of drawing speed. Figure 2.13 shows a graph of directional derivation against the arc length, which corresponds to a fast drawn straight line. For comparison, Figure 2.14 gives similar graph corresponding to a slow drawn straight line. The sketched lines in both figures have the same direction (along 135° direction). It can be seen from Figure 2.13 and Figure 2.14, that the fast sketched line has relatively smaller
directional derivation than the slow sketched curve. Therefore, it is reasonable to take an adaptive speed tolerance $\beta_s$ into account in this on-line application. Figure 2.15 illustrates results of adaptive speed tolerance for a given sketch comprising two lines, drawn from left to right. Initial sketch and corresponding fits are displayed at the upper-right corner. From the density of the sketch points, it can be seen that the drawing speed is relatively low at the begin points, the middle points and the end points of the first line. The drawing speed for the second line is relatively high, excluding the end points. On the left side of Figure 2.15, the top graph gives changes of directional derivation against the arc length. It roughly reflects the speed effects on directional derivation. The bottom graph shows changes of adaptive speed tolerance over the arc length. From this graph, it is clear that the graph has four peaks corresponding to the four low speed parts of the sketch. Furthermore, by comparing the top graph with the bottom one, it can be observed that the adaptive speed tolerance function can produce satisfactory tolerance values for the corresponding directional derivation. So, the adaptive speed tolerance function can be of practical use.

![Figure 2.14 Slow sketching along a straight line.](image)
Figure 2.15 Speed tolerance $\beta$, for specific input.

Figure 2.16 Linearity tolerance

Obtained testing results of linearity tolerance for a sketched line and an arc are shown in Figure 2.16. The two sketched primitives are displayed on the right side, and the linearity tolerance graph against arc length is on the left side. The bottom graph corresponds to the line drawing. The
linearity tolerance value for the line is nearly constant. This means that there are hardly any changes of the curve’s curvature. The top graph represents the tolerance for the arc drawing. This graph is with higher tolerance, which means that the system can give correct measurements of the curve linearity at a large scale. It also can be found from this graph that the tolerance value at the two end parts is gradually greater than that at the middle part. This property enables the system to overcome easily the occurrence of unintentional corner points during processing the two end parts. So, the linearity tolerance function is usable.

Properties of the adaptive speed threshold function \( \theta \) are demonstrated in Figure 2.17 and Figure 2.18. In Figure 2.17, the graph of the speed threshold \( \theta \) over arc length corresponds to a given circular curve. The values of \( \theta \) changes smoothly around the average speed. The graph also shows that values of \( \theta \) at two end parts are relatively smaller than that at the middle parts. This property is helpful to overcome occurrences of unintended corner points at the end parts of conic curves.

Figure 2.17. Adaptive speed threshold for a conic
Similarly, the graph corresponding to a given free-form curve in Figure 2.18, shows that values of \( g \) at the middle parts of the curve are relatively smaller, than the values at the two end parts of the free form curve. This property gives a more strict check of the corner points while processing the middle parts, than for the end parts. For an intentionally free-form curve, this property is useful and welcomed. It is believed from the two examples that the adaptive speed threshold function is of practical use, and can reflect user's intention for different type curves.

![Figure 2.18. Adaptive speed threshold for a free-form curve](image)

It can be concluded from the above discussion, that selected fuzzy functions in terms of drawing speed tolerance, linearity tolerance and speed constraints are reasonable, practical, and useful.

### 2.4.6 Detecting Inflection Points

If a sub-curve between two neighbouring corner points is not well fitted by a straight line, we continue to detect inflection points between the two corner points. We define a convexity vector as a cross product vector of two unit vectors:

\[
K_i = V_{i,i-m} \times V_{i,i+m}.
\]
where $V_{i-i_m}$, $V_{i+i_m}$ and $m$ are defined in section 3.2, index $i$ is between two adjacent corner points.

From vector $K_i$, the drawing direction and convexity of the shape can be received. If all products $K_i > 0$, they point out of the screen, which means the drawing direction is anti-clockwise (CCW) along with a left-hand bend. If all $K_i < 0$, the drawing direction is clockwise (CW) along with a right-hand bend. If $K_i = 0$, the drawing direction is along a straight-line. At inflection points, the adjacent $K_i$'s should change their signs, which means the convexity of curves changes from convex to concave or vice versa. Some examples of the $K_i$'s changes are shown in Figure 2.19, and Figure 2.20. Note that the fitted curves are right-shifted to 2 units in order to see the initial sketches clearly.

Figure 2.19 Effects of drawing direction on $K_i$'s
Figure 2.20 $K_i$'s changes for a free-form curve with two inflection points.

We use the dot product of $K_i$ and $K_{i+1}$ to detect candidates for inflection points for each span between two adjacent corner points. Then we utilise the refinement process to further deal with those candidates, because some candidates are introduced because of sketching defects, such as hand vibration, which does not reflect the user's intention. The point $i$ is a candidate for inflection point if the following conditions are sufficed:

1. The dot product of $K_i$ and $K_{i+1}$ is equal to or less than zero;
2. The linearity of the curve segment between point $i$ and previous segmentation point is less than an adaptive threshold value such as 0.95;
3. The arc length between point $i$ and previous segmentation point is greater than 5% of the arc length between the two adjacent corner points. This threshold can be changed by users according to their sketching skills and individual requirements.
Conditions (2) and (3) are applied to filter most of the high frequency inflection points due to hand or mouse vibrations. Figure 2.21(a) shows sketches with high frequency inflection points along with an intended elliptical arc (dash line), and finds some suitable candidates such as $SP_s$ in Figure 2.21(b). After detecting all candidates for inflection points of the current span, the following refinement process is conducted:

1. Sub-divide the current span into sub-spans by dividing at all candidates. Then divide each sub-span by a step specified by the user, from 5% to 25% arc length of each sub-span, depending on the user’s sketching skills and the precision requirement. Consequently, five sub-dividing points around the candidates can be obtained. Figure 6(b) shows this process, $SP_{s-1}$, $SP_s$, and $SP_{s+1}$ are three adjacent candidates, five points $\{P_j | j = 0, 1, 2, 3, 4\}$ around the candidate $SP_s$ are received by setting the step=25% arc length of each sub-span;

2. Calculate the convexity vectors $K_f = U_j \times U_{j+1}$, $(j = 1, 2, 3)$ at a large scale, where $U_j$ is a unit vector along with $(P_j-P_{j-1})$;

3. The candidate $SP_s$ can be determined as a meaningful inflection point, only if the dot product of $K_f$ and $K_{f+1}$ is less than zero.

Finally, we have obtained all segmentation points (obtuse corner points, acute corner points and inflection points).
2.5 Case Studies

Both, synthetic data and real-time sketches were generated to test the proposed procedure. Two types of synthetic contours were created: a torch-like shape of Figure 2.22, and a lamp-like shape shown in Figure 2.24. The first synthetic shape is a combination of fifteen lines, consisting of seven obtuse corners and seven acute corners. To obtain the synthetic data, a sequence of straight line segments are generated, and then filtered by distance threshold 3.5. The line sequence can be specified by Table 2.1. Figure 2.22 shows the result after the filtering. A pre-assumed value for speed information such as zero is assigned to each point. Accordingly, the adaptive speed tolerance is ignored. These testing data have a segmentation result shown in Figure 2.23.

Table 2.1 Sequenced line segments

<table>
<thead>
<tr>
<th>Number</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direction (degrees)</td>
<td>105</td>
<td>180</td>
<td>130</td>
<td>0</td>
<td>140</td>
<td>10</td>
<td>85</td>
<td>-85</td>
<td>-10</td>
<td>-140</td>
<td>0</td>
<td>-140</td>
<td>180</td>
<td>-90</td>
<td>-125</td>
</tr>
<tr>
<td>Length</td>
<td>50</td>
<td>30</td>
<td>40</td>
<td>50</td>
<td>45</td>
<td>50</td>
<td>70</td>
<td>70</td>
<td>50</td>
<td>45</td>
<td>50</td>
<td>30</td>
<td>30</td>
<td>40</td>
<td>25</td>
</tr>
</tbody>
</table>

Figure 2.22 Torch-like shape
In Figure 2.23, all acute corner points are marked by small circles (the fitted curves are right-shifted to 2 units for a clear view of initial data). The other segmentation points are obtuse corner points (unmarked). The result shows that the system successfully detects all obtuse corners and acute corners.

![Figure 2.23 Segmentation result of torch curves](image)

The second synthetic figure is referenced from Ten and Chin [1989] and Chen [1996], consisting of four semi-circles having different radii. It represents a typical example of a curve that has features of multiple scales. The reason for choosing this figure for the experiment is mainly to compare the results produced by my procedure with those from the dominant-point detection methods presented by Teh and Chin [1989]. The largest circle is set with a radius of 100. The synthetic data are produced from the largest arc in an anti-clockwise direction, and then filtered. Similarly, a default value for speed information such as zero is assigned to each point, and the adaptive speed tolerance is ignored. The segmentation results are shown in Figure 2.24 and Figure 2.25.

54
Figure 2.24 Four arcs with multiple scales (starting point marked with a triangle)

Figure 2.25 Segmentation result of connected four arcs.

In Figure 2.25, one acute corner point is marked with a small circle. The other segmentation points are obtuse corner points (unmarked). The result shows that the system can successfully find all the segmentation points for different scale features.
Figure 2.26 illustrates the detection of obtuse corner points of on-line sketching. Dot graphs display initial sketches. Line graphs (right-shifted 2 units for a clear view of initial sketches) show results of segmentation and corresponding fits. Drawing speed varies from very slow (drawings on the lower-left side) to very fast (sketches on the lower-right side). Primitives connected to the obtuse corner points include straight lines, arcs, elliptical arcs, and free-form curves. In the two left columns, drawings are produced by sketching in different directions (clockwise vs. anticlockwise, top-to-bottom vs. left-to-right). This indicates that this system can work in direction-independent way. As it can be seen from this figure, all obtuse corner points are found. At the lower-right corner, the middle line in the ‘Z’ shape is recognised as a straight line, since the drawing speed was fast, the user paid less attention to it. If the speed was slow, it might be an arc.

Figure 2.26 Detection of obtuse corners

56
Some examples given in Figure 2.27 demonstrate the system's ability to determine acute corner points. In the graphs (fitted curves are shifted 2 units to the right), small circles are used to mark the acute corners. They indicate that all intentional acute corners can be obtained, regardless of changes in the drawing speed, and that connected primitives have different types.

Figure 2.27 Detection of acute corners

Figure 2.28 is employed to give examples of finding inflection points (fitted curves are shifted 2 units to the right). In this figure, the first drawing at the upper-left corner gives an example of the combination of detecting corners and finding inflection points. Acute corners are marked with the smaller circles. Candidates of inflection points are signed with short-vertical line bars. If the refinement process confirms the candidates as real inflection points, the inflection points will be
marked by two concentric circles. The graphs at the bottom show that some candidates of inflection points are not real inflection points, because they can not match users' intention. In the middle column, there are two drawings with extra acute points. These acute points are detected as unintentional, caused by fast drawing or hand vibration. This implies that the system can not guarantee 100% accuracy in the detection of segmentation points.

![Figure 2.28 Detection of inflection points](image)

Actually, it is hard for any system to find a perfect solution, because segmentation techniques are basically 'ad hoc' and highly problem dependent. Thus, it is very demanding for a system to provide a way for correcting a wrong solution. In our system, users can correct unintended segmentation and corresponding fits to their intentional primitives by interaction with the system. For example, if the system gets segmentation result as shown in Figure 2.29, not relevant to the
users' intentional shape of an ellipse, then the user can simply click the “X” icon on the toolbar menu to undo the segmentation, and click the ellipse icon menu to give a correction. This interaction can produce a correct answer, as shown in Figure 2.30.

![Figure 2.29 Unintended segmentation points](image)

![Figure 2.30 Interactive correction of segmented curve](image)

2.6 Discussion

For a comparative study of curve segments, it is difficult to give direct comparison with previous methods, since most of them are based on either chain-codes, or polygon approximation. On the
other hand, they do not use any dynamic information. Here, some comparison is made in a qualitative way.

(1) In angle or curve curvature detection, this system can represent and accept arbitrary angles to be examined, whether chain code based system [Freeman 1977] can only present limited directions (4 or 8). This implies that my system can work more accurately.

(2) Dominant points based methods [Rosenfeld 1973] [Teh 1989], are concentrated at local maxima points of curvatures. Once the local maxima points are obtained, they are automatically regarded as segmentation points. Actually, some of them are not real segmentation points if segmented curve is considered at a large scale. The problem is that there is no adaptive evaluation measurement for an individual dominant point to check if it is a real segmentation point. As a result, these approaches produce too many segmentation points. In my system, an adaptive evaluation measurement (threshold: $\beta_0 + \beta_5 + \beta_I$, and speed constraint $\theta$) for each point is produced to check if a local maximum point is a real corner considering the drawing speed, acceleration, and linearity. Thus, this system has good potential to capture real segmentation points and users' intention. The segmentation points determined by my system are just a sub-set of dominant points.

(3) Previous break-points based system [Teh 1989] are probably more sensitive to noise, because they only focus on the detection of local maxima curvatures.

(4) Edge approximation based approaches [Alban, 1974], [Rosin 1995] are dependent on how many kinds of primitives have to be recognised. If a free-form curve is included, these methods would be difficult to apply. Moreover, these approaches lead to heavy computation, which is not applicable for online applications.

By comparing mine with previous approaches, it can be concluded that my corner point detection algorithm does not have the disadvantage of requiring a set of fixed input parameters, and does nor
involve heavy computation, as in [Teh 1989]. Its ability to automatically vary the region of support, $m$, and the adaptive threshold, $(\beta_0 + \beta_s + \beta_v)$, enables the system to perform reliably even when the object consists of multiple size features. This can be seen from the result shown in the case studies section. Furthermore, its ability to interactively correct unintentional segmentation points makes it very practical.

Generally speaking, this fuzzy knowledge based system can interpret user's intention correctly and effectively. From real-time sketches, the system can give proper segmentation and curve fitting in a variety of 2D shapes: straight lines, circles, arcs, ellipses, elliptical arcs, spiral lines, and free-form curves. This system is suitable for dealing with vague and imprecise sketching information.
Chapter 3. Parallel Classification and Identification of 2D Primitives

3.1 Introduction

After segmentation processing, the problem remaining is how to describe a set of discrete data points within one segment, into a meaningful form. This task includes two aspects: (1) classifying the set of data into different types of 2D primitives, and (2) identifying a group of parameters to determine the classified 2D primitives. In general, this task can be referred to curve fitting. It has found many applications in image processing [Rosenfeld 1976], computer vision [Kanatani 1993], pattern recognition [Duda 1973], computer graphics and computer-aided geometric design [Piegl 1995], and data analysis and presentation [Dixon 1969]. Curve fitting may refer to either interpolation or approximation. In the case of interpolation, constructing a curve satisfies the given data precisely, i.e., the curve passes through the given points and assumes the given derivatives at the prescribed points. In approximation, constructing curves do not necessarily satisfy the given data precisely, but only approximately. According to the types of 2D curves, curve fitting can be further classified into straight line fitting [Stroud 1995], [Schied 1968], conic fitting [Bookstein 1979], [Rosin 1993], free-form curve fitting [Shao 1996], [Plass 1983]. A general method for curve fitting is least squares (LS). The LS principle can be found in mathematical literature [Scheid 1968], [Stroud 1995]. From a strategic view, curve fitting can be divided into three categories: general fitting, split-and-merge based fitting, and segmentation-based fitting.

3.1.1 General Fitting

Theoretically, a curve segment can be approximated by a piece of polynomial [Scheid 1968], [Birkhoff 1965], [Rice 1969], or B-spline [Esch 1969], [Pavlidis 1983], [Plass 1983], [Shao 1996], or
Non-Uniform Rational B-Splines (NURBS) [Piegl, 1995]. One can use this kind of general LS fitting to represent a curve in a compact form, instead of suitable 2D primitives fitting. But this general fitting may result in forms other than the clearly meaningful form such as straight lines, circles or ellipses. This fitting is difficult to understand in computer vision systems. Actually, curve fitting techniques are similar to segmentation techniques, and are application or problem dependent. For example, lines, circles, and ellipses are very useful features used to recognise the machined parts in 3D computer vision systems, A piece of polynomial or free-form curve is not a very helpful feature in these systems, and thus ‘general fitting’ is not my choice.

3.1.2 Split-and-Merge Based Fitting

The ‘split-and-merge’ based fitting method is actually embraced in the edge approximation based segmentation process [Albano, 1974], [Rosin and West, 1995]. For example, Etemadi [1992] proposed a three pass segmentation techniques for segmenting a planar curve into straight lines and circular arcs, based on the principle of splitting and merging. On the first pass, the curve to be segmented is divided into small segments. Each segment is almost symmetric about an axis passing through the midpoint of the segment in a direction perpendicular to the line joining its end points. On the second pass these segments are linked to form co-curvilinear and collinear segments based on curve fitting. Finally, each merged segment is classified as a straight line or a circular arc depending on the deviation of the midpoint of the segment from the line joining the two end points. This approach has some shortcomings, one of which is the heavy computational requirement as discussed in Chapter 2.

3.1.3 Segmentation Based Fitting

This kind of curve fitting is based on the segmentation processing. Before the curve fitting, it is assumed that each segment has been segmented properly, or that it represents only one entity. When
making curve fits, it is not necessary to consider adjustments to the end points in accordance with segmentation concerns as in the edge approximation approaches.

For the on-line classification of sketch strokes, some mode-dependent preference functions have been employed for the interpretation of shapes including lines, circular arcs, full circles, and B-spline curves [Egglı et al 1997]. One of the basic 2D primitives, the ellipse, is not included in their system. Furthermore, they did not give details about the mode-dependent preference functions.

The use of a conic equation for best-match fitting of sketch strokes is described by Shpitalni and Lipson [1997]. The result of this matching is a group of conic sections representing lines, arcs, elliptical arcs and hyperbolas. The system ignores all other possible convex curves, such as free-form curves. Moreover, the system can hardly distinguish straight lines from low curvature elliptical arcs, because an exact line is a rare form of a conic curve. Thus, I found the use of the general conic fitting is not useful for my system.

Jenkins and Martin [1992] use a fit-and-test method to classify sketches according to the complexity of the curves. Here, the complexity of a shape is judged on the amount of information that must be given in order to specify its shape. The simplest primitive types are considered first, so the system initially tests if a straight line is a good fit to the stroke data. If this fails, a circular arc is next considered, and if this fails too, a composite Bezier curve is used to represent the sketches. The system excludes ellipses and elliptical arcs. This fit-and-test approach has a heavy computational load.

A fuzzy logic based system presented in [Chen 1996] uses a different reference model to obtain corresponding possibilities, identify the types of shapes, and generate the corresponding geometric primitives and B-spline curves. The reference models are adopted in preference order from lines, circles, and ellipses to free-form curves. Their classification procedure for a curve can have the following steps:
Step 1: evaluate the curve in a line reference model to obtain a possibility value of “the curve is a line”;

Step 2: evaluate the curve in a circular reference model to obtain a possibility value of “the curve is a circle or circular arc”;

Step 3: evaluate the curve in an elliptic reference model to obtain a possibility value of “the curve is an elliptical arc or ellipse”;

Step 4: evaluate the curve in a closed curve reference model to obtain a possibility value of “the curve is a closed curve”;

Step 5: apply fuzzy inference rules to determine which geometric primitives the curve belongs to. If it does not belong to any primitives, it is a free-form curve. These rules also determine if the curve is open or closed.

Step 6: the curve is generated as one of the geometric primitives or B-spline curve;

Step 7: repeat step 1 to step 6 for another segment.

This method seems complicated and time-consuming, since each type of curve needs to go through all of these steps, no matter how simple it is. This system is also without sufficient robustness for ellipses and elliptical arcs [Qin 1999].

This chapter presents a method for classifying sketch strokes used in our on-line sketching system. The method is based on fuzzy knowledge with respect to the curve’s linearity, convexity, and drawing speed. It proposes the use of inferring rules to classify sketch strokes (curves) and identify lines, circles, arcs, ellipses, elliptical arcs, spring lines and free-form B-splines. The proposed technique is tested and proven to be fast, and suitable for real-time classification and identification.
3.2 Classification

To find suitable 2D primitives, it is very important for the system to be able to correctly classify a sub-curve as a line, a conic curve or a free form curve. Curve classification follows a linear four-step procedure (Figure 3.1).

![Diagram of curve classification procedure](image)

**Figure 3.1 A curve classification procedure**

3.2.1. Classification of Lines

We classify a curve according to three preference orders: linearity (or straightness), convexity, and complexity of a shape, not only by complexity, as in [Chen 1996], [Jenkins 1992]. We consider the curve classification as an ill-conditioned problem, depending on applications. For example, the curve shown in Figure 3.2(a) represents a straight line. But, if it is viewed at a fine scale, changes of convexity will be seen. Theoretically speaking, a curve with some changes of convexity cannot be a straight line or any conic section. To avoid this scale problem and to quickly classify curves as straight lines, we firstly compute the curves' linearity, and then classify the curves using it. The linearity is the value of the distance between two end points divided by the accumulative chord length between them. For example, the linearity of a strictly straight line segment should be 1. So,
before a sub-curve is interpreted, its linearity is used to evaluate the possibility to be a straight line. For instance, if the linearity is 0.95, it means that the corresponding curve has a 95% possibility of being treated as a line. In our system, if a curve's linearity is greater than an adaptive threshold, such as 0.98, the curve will be identified as a straight line. In contrast to a least square line fitting and testing method, computing linearity is easier. This linearity threshold can be computed by an adaptive threshold function of the average drawing speed. If users draw the sketch very fast or roughly, the lower threshold is expected. If the users draw the sketch carefully and slowly, the higher threshold should be exploited. This is because slow drawing implies that the users are paying more attention to each point, and hence, each point has a more accurate geometric meaning. Furthermore, the better the drawing skills, the lower the threshold. This linearity threshold \( T_s \) can be expressed as \( T_s = f_s + 0.9 + (10 - \text{mean}_{\text{speed}}) \times 0.5\% \). The drawing skill factor \( f_s \in [0.01, 0.05] \) can be selected by the users from a system menu. The mean speed is the average speed over the sub-segment, which is normally less than 10, according to tests. If the mean speed is greater than 10, \( T_s = f_s + 0.9 \). All constants come from tests. For future work, all these constants could be learnt from an intelligent machine learning (or training) process.

3.2.2. Classification of Free-Form Curves

Secondly, if a curve is not a straight line, we check whether it has changes of convexity from convex to concave, or vice versa, by detecting inflection points and changes in the drawing directions. If the curve has some inflection points, it will be classified as a free-form curve. Sometimes, the system may fail to find inflection points; in this rare case, it will further detect changes of convexity by the following steps (Figure 3.2(b)):

- Sub-divide the curve by an arc length. The stepping length is 5~10% of the total accumulative chord length, and could be selected by the users in accordance with their precision requirement and skills;
- Compute a convexity vector as a cross product of two vectors; that is \( \mathbf{K}_i = \mathbf{V}_{i-1} \times \mathbf{V}_{i+1} \), where \( \mathbf{V}_{i-1} = \mathbf{P}_i - \mathbf{P}_{i-1} \), \( \mathbf{V}_{i+1} = \mathbf{P}_{i+1} - \mathbf{P}_i \). From the vector \( \mathbf{K}_i \), the drawing direction and convexity of the shape can be received. If all \( \mathbf{K}_i > 0 \), they point out of the screen, which means the drawing direction is anti-clockwise along with a left-hand bend. If all \( \mathbf{K}_i < 0 \), the drawing direction is clockwise along with a right-hand bend. If \( \mathbf{K}_i = 0 \), the drawing direction is along a straight-line. At convexity change points, the adjacent \( \mathbf{K}_i \)'s should change their signs, which means that curve convexity changes from convex to concave, or vice versa;
- Use the dot product of \( \mathbf{K}_i \) and \( \mathbf{K}_{i+1} \) to detect changes of convexity, if the product is negative;
- Classify a curve as a free-form curve if it has some changes of convexity.

![Diagram](image)

(a) Linearity  
(b) Convexity

Figure 3.2. Linearity and convexity

Thirdly, if there is no inflection point and there is no change in the curve's convexity, but some self-intersection points (Figure 3.3) exist, the curve would be identified as a free-form curve representing a spring line. If any of the arc lengths, which are out of the closed segment, is greater than a threshold of 20% of the total arc length, the curve is considered as a free-form curve. Figure 3.3(a) shows a normal over-drawn case, in which the two arc lengths, out of the closed segment, are
less than the threshold. Figure 3.3(b), 3.3(c), and 3.3(d) give examples of spring lines, in which arc lengths, out of the closed segment, are greater than the threshold on one or both sides.

To find if the curve has self-intersections, a simple procedure with two nested loops is adopted. For the outer loop, let the variable $i$ go from start points to end points with a step of 1. For the inner loop, let the other variable $j$ go from $i+2$ and increments by 1, until the end point is reached. Within the inner loop, I first find the intersection point of line segments $(P_i, P_{i+1})$ and $(P_j, P_{j+1})$. If the intersection point is on either of the two line segment, a self-intersection point is found. Then we record the intersection position at $(i, j)$, and assigning $i=j+2$ to break the inner loop and continue in the outer loop. Finally, all self-intersection points are extracted. This procedure is quite simple and practical.

![Diagram](image)

(a) normal  (b) one side  (c) two sides  (d) multiple-self intersections

Figure 3.3. Classification of conic sections and spirals
3.2.3. Classification of Conics

Finally, if a curve is not a spiral line, it will be fitted with a general conic equation (see details in next section). It then can be further classified as a free form curve, an ellipse (including a circle), a arc, and a hyperbola, or a parabola. This step mixes the identification and classification.

3.3 Identification and Generation of 2D Primitives and B-spline Curves

After the classification, each curve should be identified and fitted with a meaningful 2D primitive or a B-spline segment to represent its corresponding sketching points. In general, let the number of the points be $n$.

3.3.1 Generation of Straight Lines

Once we have classified a curve as a line, we need to generate it from a sequence of $n$ corresponding sketching points, with the best approximation. The problem is to find the best coefficients $(a, b)$ for the normalised line equation: $y = ax + b$. A weighted Least Square (LS) routine is used [Dixon and Massey, 1969]. The sum of the squares of the deviation is obtained by

$$f(a, b) = \sum_{i=1}^{n} w_i (y_i - ax_i - b)^2,$$

Where $w_i$ is a weight for point $(x_i, y_i)$. We assume that $w_i$ is positive, and relatively large, for low speed drawing, and relatively small for high speed drawing, because human intention represented at each point, in terms of speed, is different. Normally, the higher the drawing speed, the less the contribution of the intention. I apply a triangular distribution of weights for each speed. When the drawing speed is within the speed limits, the ratio $w_i$ of average speed to the drawing speed at point
\((x_i, y_i)\) can be used as a weight value. Thus, when the drawing is with a constant speed, the weight for each point is 1, no matter how high, or low, the average speed is.

The necessary conditions for minimisation of function \(f(a, b)\) are its partial derivatives for \(a\) and \(b\) to vanish. That is,

\[
\frac{\partial f}{\partial a} = -\sum 2w_i(y_i - ax_i - b)x_i = 0,
\]

\[
\frac{\partial f}{\partial b} = -\sum 2w_i(y_i - ax_i - b) = 0.
\]

From these equations, we receive the following concurrent equations:

\[
a\sum w_ix_i^2 + b\sum w_ix_i = \sum w_ix_iy_i,
\]

\[
a\sum w_ix_i + b\sum w_i = \sum w_iy_i.
\]

These equations are the “normal equations”. Introducing the symbols

\[
s_0 = \sum w_i x_i^2, \quad s_1 = \sum w_i x_i, \quad t_1 = \sum w_i x_i y_i,
\]

\[
s_2 = \sum w_i, \quad t_2 = \sum w_i y_i,
\]

these equations may be solved in the form

\[
a = \frac{t_1 s_2 - t_2 s_1}{s_0 s_2 - s_1^2}, \quad b = \frac{t_2 s_0 - t_1 s_1}{s_0 s_2 - s_1^2}
\]

Finally, the coefficients \((a, b)\) can be received by solving the above equations, and the line can be generated. When a sketch is a vertical line exactly, the above equation will become singular. Thus, when I find that the line is vertical, I skip this LS fitting process and directly find a properly vertical line for the sketch.

The fitted line may not pass through the first and last points of the sketch. To determine the starting and ending points of a fitted line segment, the intersection point of the generated line and a line, which passes the first point and is perpendicular to the generated line, is produced as the starting point. The ending point is defined in the same way.
3.3.2 Generation and Classification of Conic Curves

After a curve is considered as a conic curve, the problem remaining is to fit a conic section to a set of sketching points \( \{S_j, j=1, \ldots, n\} \). A general conic section can be described by the following equation [Bowyer and Woodwark, 1983]:

\[
Q(x, y) = ax^2 + 2hxy + by^2 + 2gx + 2fy + c = 0. \tag{1}
\]

Many parameter estimation techniques for finding the coefficients of this equation have been developed in the vision community. These include various LS schemes [Gander 1994], [Bookstein 1979], [Rosin 1993], Kalman filtering [Zhang 1992], [Porrill 1992], and Hough transform based methods [Huang 1989], [Tsuji 1978], [Ballard 1995]. Generally speaking, The Kalman filtering approach is iterative and computationally intensive and the Hough transform requires a large number of data points for a good result [Rosin 1993]. In our system, we investigate the weighted LS fitting, based on algebraic distances, because geometric distances are difficult to evaluate. Thus, the fitting problem reduces to minimising a function,

\[
E = \sum_{i=1}^{n} w_i Q(x_i, y_i)^2 \tag{2}
\]

Equation (1) has 6 degrees of freedom, with one of them redundant, due to the scaling possibility of this implicit form. To avoid a trivial solution \( a = h = b = g = f = c = 0 \) of equation (2), \( Q(x, y) \) should be normalised. There are many different normalisations proposed in the literature [Bookstein 1979]. Here, only two common normalisations are discussed. Care has to be taken in selecting the normalisation procedure because most normalisations create singularities. These prevent a set of conics being fit, and make the neighbouring conics difficult to fit. The equation (1) can be normalised with \( a + b = 1 \). Setting \( a + b = 1 \), the equation (1) has no singularities involving ellipses, but only rectangular hyperbolae will be excluded. This normalisation is sensitive to eccentricity for
short sections. Experiments [Rosin, 1993] show that increased eccentricity implies a tendency to fit hyperbolae instead of ellipses.

In practice, equation (1) can be assumed with \( c \neq 0 \), and therefore can be normalised with \( c = 1 \). Note that this normalisation (\( c = 1 \)) has singularities for all conic sections crossing the origin. However, as stated in [Rosin 1993], the singularity problem can be overcome by shifting the data. This normalisation is not sensitive to eccentricity for short sections. Because of these advantages, we decided to use this normalisation in our system. After the normalisation, the fitting problem is to minimise the following function:

\[
E = \sum_{i=1}^{n} w_i (ax_i^2 + 2hx_iy_i + by_i^2 + 2gx_i + 2fy_i + 1)^2 ,
\]

where weight the \( w_i \) is received in the same way as in section 3.3.1. To obtain the minimum value of \( E \), the following five partial derivatives are set to 0,

\[
\begin{align*}
\frac{\partial E}{\partial a} &= \sum 2w_i x_i (ax_i^2 + 2hx_iy_i + by_i^2 + 2gx_i + 2fy_i + 1) = 0, \\
\frac{\partial E}{\partial h} &= \sum 2w_i x_i (ax_i^2 + 2hx_iy_i + by_i^2 + 2gx_i + 2fy_i + 1) = 0, \\
\frac{\partial E}{\partial b} &= \sum 2w_i y_i (ax_i^2 + 2hx_iy_i + by_i^2 + 2gx_i + 2fy_i + 1) = 0, \\
\frac{\partial E}{\partial g} &= \sum 2w_i x_i (ax_i^2 + 2hx_iy_i + by_i^2 + 2gx_i + 2fy_i + 1) = 0, \\
\frac{\partial E}{\partial f} &= \sum 2w_i y_i (ax_i^2 + 2hx_iy_i + by_i^2 + 2gx_i + 2fy_i + 1) = 0,
\end{align*}
\]

where the symbol \( \Sigma \) implies summation for \( i \) from 1 to \( n \). We then obtain the system of five simultaneous linear equations:

\[
\begin{align*}
(\sum w_i x_i^4) a + (\sum w_i x_i^3 y_i) h + (\sum w_i x_i^2 y_i^2) b + (\sum w_i x_i y_i) g + (\sum w_i y_i) f + \sum w_i x_i^2 &= 0, \\
(\sum w_i x_i^3 y_i) a + (\sum w_i x_i^2 y_i) h + (\sum w_i x_i y_i) b + (\sum w_i x_i y_i) g + (\sum w_i y_i) f + \sum w_i y_i^2 &= 0, \\
(\sum w_i x_i^2 y_i^2) a + (\sum w_i x_i y_i) h + (\sum w_i y_i) b + (\sum w_i x_i y_i) g + (\sum w_i y_i) f + \sum w_i y_i^2 &= 0, \\
(\sum w_i x_i y_i) a + (\sum w_i x_i y_i) h + (\sum w_i x_i y_i) b + (\sum w_i x_i y_i) g + (\sum w_i y_i) f + \sum w_i x_i &= 0, \\
(\sum w_i x_i^3 y_i) a + (\sum w_i x_i^2 y_i) h + (\sum w_i x_i y_i) b + (\sum w_i x_i y_i) g + (\sum w_i y_i) f + \sum w_i y_i &= 0.
\end{align*}
\]

By solving the above equations, the coefficients \( (a, h, b, g, \text{ and } f) \) can be obtained. Once these coefficients are found, we use formula (3) for computing the LS fitting error. If the mean fitting error
over $n$ points is greater than a threshold value of 0.001 (obtained from tests), the sub-curve will be classified as a free-form curve. If not, categorising a given conic (1) into one of the three possible forms, can be carried out using the following three invariants [Bowyer 1983]:

$$
\Delta = a(bc - f^2) - h(hc - gf) + g(hf - gb);
$$

$$
\delta = ab - h^2;
$$

$$
s = a + b.
$$

If $\Delta = 0$, then the conic degenerates into a line(s), or a point (which may not always exist), otherwise:

if $\delta < 0$, the conic is a hyperbola;

if $\delta = 0$, the conic is a parabola;

if $\delta > 0$ and $\Delta < 0$, the conic is an ellipse.

The five parameters of a general ellipse: the central point $(X_c, Y_c)$, the two radii $(R_a, R_b)$ and the rotation angle $\theta$, could be received simply from: (a) translating the ellipse to the origin to eliminate coefficients $g$ and $f$; and (b) rotating through an negative $\theta$ angle to eliminate coefficient $h$. Finally, we have [Qin 1999]

$$
X_c = (bg - fh) / (h^2 - ab),
$$

$$
Y_c = (af - gh) / (h^2 - ab),
$$

$$
= \frac{1}{2} \arctg \left( \frac{2h}{a - b} \right),
$$

$$
R_a = (-c' / a')^{1/2}
$$

$$
R_b = (-c' / b')^{1/2}
$$
where
\[ a' = a \cos^2 \theta + b \sin^2 \theta + 2h \sin \theta \cos \theta, \]
\[ b' = a \sin^2 \theta + b \cos^2 \theta - 2h \sin \theta \cos \theta, \]
\[ c' = aX_C^2 + bY_C^2 + 2hX_CY_C + 2gX_C + 2fY_C + c. \]

A second way [Qin 1999] of obtaining these parameters can be rotation about origin through \(-\theta\) angle (see Figure 3.4) to eliminate coefficient \(h\). After the rotation transformation, the ellipse will be placed at a special position, centred at \((X_e, Y_e)\), where the ellipse is parallel to either the x-axis or the y-axis; a ray from origin to new centre \((X_e, Y_e)\) has a direction of \(\theta\). From this rotation transformation, I can obtain the above five parameters as follows:

\[ \theta = \frac{1}{2} \arctan \left( \frac{-2h}{a-b} \right), \]
\[ R_a = \left( \frac{g' + f' \sqrt{b' - c}}{b'} \right)^{1/2}, \]
\[ R_b = \left( \frac{g' + f' \sqrt{b' - c}}{b'} \right)^{1/2}, \]
\[ X_C = X_e \cos \theta - Y_e \sin \theta, \]
\[ Y_C = X_e \sin \theta + Y_e \cos \theta, \]

where
\[ X_e = g' / a', \]
\[ Y_e = f' / b', \]
\[ a' = a \cos^2 \theta + b \sin^2 \theta + 2h \sin \theta \cos \theta, \]
\[ b' = a \sin^2 \theta + b \cos^2 \theta - 2h \sin \theta \cos \theta, \]
\[ g' = g \cos \theta + f \sin \theta, \]
\[ f' = -g \sin \theta + f \cos \theta. \]
Here, we describe the second method just for correction purposes, because the reference [Chen 1996] used this approach in improper way [Qin 1999].

Translation and orientation for hyperbolae, with respect to their canonical position, can be determined by the same centre \((X_c, Y_c)\) and \(\theta\), as for ellipses. The length of their main axes can also be calculated by \(R_a\) and \(R_b\). In the rare case of obtaining a fit for a parabola, we just record its classification information, and then use a free form curve to fit it (see next section).

If the ratio of \(R_a/R_b \approx 1\), we classify the ellipse as a circle, and simply take the centre of the ellipse as its centre. The average of \(R_a\) and \(R_b\) is then the circle radius.

Figure 3.4 Ellipse transformation

Figure 3.5 Finding start and end angles
Furthermore, a start angle and an end angle have to be computed in order to specify elliptical arcs
(or circular arcs). Firstly, drawing direction information can be received from the convexity vector
$K_i$. If $K_i > 0$, the drawing direction is anti-clockwise, and if $K_i < 0$, the drawing direction is
clockwise. To find the start and end angles, three points are taken into account: first point $S_I$, last
point $S_n$, and middle point $S_m$, as shown in Figure 3.5. For an anti-clockwise drawing (Figure
3.5.(a)), the start angle $\phi_i$ corresponds to the first point $S_I$, the end angle $\phi_n$ corresponds to the last
point $S_n$, and the middle point $S_m$ corresponds to angle $\phi_m$. For a clockwise drawing, it is necessary
to exchange $S_I$ and $S_n$, then the start and end angles can be obtained as for anti-clockwise drawing
by:

$$\phi_i = \arccos((x_i - X_c)/r_i), \text{ and}$$

$$\phi_i = 2\pi - \phi_i, \text{ if } (y_i - Y_c) < 0;$$

where $r_i$ is a distance between $S_I$ and the centre point.

Secondly, angles $\phi_m$ and $\phi_n$ can be received in the same way, and further processing is conducted
to get correct angles. The over-drawn case shown in Figure 3.5. (b) is treated as:

$$\phi_m = \phi_m + 2\pi, \text{ if } \phi_m < \phi_i;$$

$$\phi_n = \phi_n + 2\pi, \text{ if } \phi_n < \phi_i;$$

$$\phi_n = \phi_i + 2\pi, \text{ if } \phi_n > \phi_i \text{ and } \phi_n < \phi_m \text{ (over-drawn case);}$$

Finally, the start angle $\phi_i$ and end angle $\phi_n$ are received. The value of $\phi_i$ is between 0 and $2\pi$;
$\phi_n$ can however range from 0 to $4\pi$, because the extent may be $2\pi$ or more. If $\phi_n - \phi_i \approx 2\pi$, the
fitting curves are closed (ellipses or circles), otherwise, they are open (elliptical arcs or circular arcs).

Here, start angles and end angles are geometric angles.
Parametric forms for conics are employed in the system in order to display arcs. For circular arcs, the equations with a parameter $\beta$ are:

\[ x = X_c + r \cos \beta, \]
\[ y = Y_c + r \sin \beta. \]

For elliptical arcs, equations with a parameter $\beta$ can be expressed as

\[ x = X_c + R_a \cos \beta \cos \theta - R_b \sin \beta \sin \theta, \]
\[ y = Y_c + R_a \cos \beta \sin \theta - R_b \sin \beta \cos \theta. \]

For circular arcs, the angular parameter $\beta$ is the same as a geometric angle $\varphi$. Thus, the angles of $\varphi_1$ and $\varphi_n$ can be directly used as parameters to form a range when generating arcs, while for elliptical arcs, they cannot be automatically employed as parameter angles, because $\beta$ has a fixed relationship with the corresponding geometric angle $\varphi$, which is $\tan \varphi = (R_b / R_a) \tan \beta$ (see Figure 3.6).

When generating an elliptical arc, we need to convert a geometric angle to a parameter angle first. This transfer can be described with the following procedure:

```
{ 
initialise $\xi = 0$ and $\beta = 0$; 
if $\varphi > 2\pi$, then $\xi = 2\pi$ and $\varphi = \varphi - \xi$; 
if $\pi / 2 > \varphi \geq 0$, $\beta = \arctan((R_n / R_b)\tan \varphi)$; 
if $\varphi = \pi / 2$, $\beta = \pi / 2$; 
if $3\pi / 2 > \varphi > \pi / 2$, $\beta = \pi + \arctan((R_n / R_b)\tan \varphi)$; 
if $\varphi = 3\pi / 2$, $\beta = 3\pi / 2$; 
if $2\pi \geq \varphi > 3\pi / 2$, $\beta = 2\pi + \arctan((R_n / R_b)\tan \varphi)$; 
finally, let $\beta = \beta + \xi$; 
}
```
After this conversion, a pair of parametric angles \( \beta_1 \) and \( \beta_n \) can be received as range values to generate arcs, corresponding to \( \varphi_1 \) and \( \varphi_n \), and the \( \beta_1 \) and \( \beta_n \) will finally become model values for elliptical arcs in the system.

### 3.3.3 B-Spline Fitting

For curve fitting, B-splines are usually used to fit a set of points, since B-splines have flexible local control properties. There are two main ways for curve fitting: interpolation and approximation. In general, the interpolation is easier than the approximation. When interpolating, the number of control points is automatically determined by a chosen degree and the number of data items. The corresponding knot vector can be obtained in advance. Also, there is no curve error to be checked. When approximating, we don’t know in advance how many control points are required to obtain the desired accuracy \( \varepsilon \); the value of \( \varepsilon \) is usually not known, and hence the approximation methods are
generally iterative [Piegl L., 1995]. For real-time application, interpolation approaches are more suitable.

For a given set of sketch points \( \{S_i\} \), we intend to interpolate several key points \( \{D_i\}, i = 0, 1, \ldots, m \) of \( \{S_i\} \) with a \( p \)th degree B-spline curve, that is

\[
C(u) = \sum_{i=0}^{m} N_{i,p}(u) P_i.
\]

where \( P_i \) are the \( m+1 \) unknown control points. \( N_{i,p}(u) \) is \( p \)th-degree B-spline function, which is defined by the following recurrent formula:

\[
N_{i,0}(u) = \begin{cases} 
1, & \text{if } u_i \leq u < u_{i+1} \\
0, & \text{otherwise,}
\end{cases}
\]

\[
N_{i,p}(u) = \frac{u-u_i}{u_{i+p}-u_i} N_{i,p-1}(u) + \frac{u_{i+p+1}-u}{u_{i+p+1}-u_{i+1}} N_{i+1,p-1}(u).
\]

If we give a parameter value \( \hat{u}_k \) for each key point \( D_k \), and select an appropriate knot vector \( U = \{u_0, u_1, \ldots, u_{m+p+1}\} \), we can build up \( (m+1) \times (m+1) \) system of linear equations

\[
D_k = C(\hat{u}_k) = \sum_{i=0}^{m} N_{i,p}(\hat{u}_k) P_i
\]

(4)
In our application, I chose cubic B-spline fitting, that is, $p=3$. For the key points, I prefer to use as few as possible, in order to fit the sketched curve with respect to the computational efficiency. Each curve between two adjacent inflection points (including two end points and change points in convexity) is sub-divided into $SEG$ segments with an equal arc length. Consequently, two end points and all $(SEG-1)$ sub-dividing points are considered as key points. Clearly, for each sub-curve, there are $(SEG+1)$ key points. Thus, the number of key points for a whole curve, $m$, can be determined by the number of sub-curves. The default value for $SEG$ is 8, but it can be selected by the user.

To solve equation (4), a knot vector must be determined in advance. We assume that the parameters in the knot vector lie in the range $u \in [0, 1]$. In this system, we use a chord length based method with average technique, recommended by [Piegl 1995], to construct the knot vector. First $u_k$ is computed by

\[
\begin{align*}
\hat{u}_0 &= 0, \quad \hat{u}_n = 1, \\
\hat{u}_k &= \hat{u}_{k-1} + \frac{|l_k \cdot l_{k-1}|}{d}, \quad k = 1, 2, \ldots, m-1, \\
\end{align*}
\]

where $d$ is the total chord length, and $l_k$ is accumulative chord length at point $k$. Then the following technique of averaging is adopted

\[
\begin{align*}
\hat{u}_0 = u_1 = \ldots = u_p = 0, \quad u_{m+1} = \ldots = u_{m+p+1} = 1, \\
\hat{u}_j + p &= \frac{1}{p} \sum_{i=j}^{j+p-1} \hat{u}_j, \quad j = 1, 2, \ldots, m-p. \\
\end{align*}
\]

Finally, I combine equations (4), (5) and (6) to determine the control points and fitted cubic B-spline.
3.4. Examples

Figure 3.7 is employed to demonstrate identification of lines, arcs, circles, ellipses and elliptical arcs (sketches to the left and fitted curves to the right). The top sketches show examples of straight lines and a whole circle. The next two sketches illustrate the application of my approach for general conics fitting and classification of arcs and elliptical arcs. The last one gives an over-drawn case of an ellipse with an open gap between starting part and ending part. Classifications by using self-intersection points are shown in Figure 3.8. The top sketch is with short arc lengths, out of the closed section, it is treated as a normal over-drawn case, and classified as an ellipse. The next from the top is with long arc lengths, out of the closed section, it is regarded as a free form curve. The middle one demonstrates spring line fitting with multiple self-intersections. The bottom sketch is without any inflection or self-intersection point, and would be fitted with a LS ellipse, but the error of this fitting is greater than a given threshold 0.001, thus, it is finally fitted with a B-spline. The sketch above the bottom one, is with some inflection points or changes of convexity, and is identified as a free-form curve with a B-spline fitting.

Figure 3.7 Conic curves.
Figure 3.8 Free-form B-splines.

Figure 3.9 Fast sketching
Figure 3.10 Slowly sketching

Figure 3.9 and Figure 3.10 are used to show the effects of the drawing speed. The two sketches look like arc shapes with similar curvatures. However, the first sketch is drawn very fast, which usually means that users pay relatively little attention to it. The threshold of linearity for it will be a lower value and finally, it is classified as a straight line. The second sketch is produced slowly, which means that users intended to draw an arc. Thus, a bigger threshold of linearity is applied to it, and it therefore is fitted with an arc. In this way, the system can intelligently interpret the users’ intention.

However, no system of classification can guarantee that it works with 100% reliability of interpreting the uses’ intention. Our system can accept interactively corrections from the users, in order to allow for mistaken interpretation. For example, one would like to sketch an elliptical arc, while the system gives a free-form curve as a response (see Figure 3.11.). In this case, the user can
simply click on the “X” icon and then choose the ellipse-shape icon. The system will accept this new instruction and will produce an elliptical arc instead (see Figure 3.12).

![Figure 3.11 Improper classification case](image1)

![Figure 3.12 Interactive correction of classification](image2)

3.5. Discussion

This parallel classification and identification method is based on segmentation processing. It can significantly save computing time, in comparison with the approximation based curve segmenting and fitting processing. Moreover, if it is compared with complexity-based classification approaches [Chen 1996], [Jenkins 1992], assuming $N$ types of curves to be classified, it still can roughly improve computing efficiency by roughly $(N-1)$ times, since the complexity-based methods would try fitting and classification $N$ times, while the parallel approach only does once. This advantage is very crucial for on-line applications.
This parallel method employs some heuristic knowledge in terms of linearity and existence of inflection points, or changes of convexity, in order to quickly classify straight lines and free form curve, and then let only conics alone. The results show that this method is very practical. Adaptive threshold of linearity in accordance with the drawing speed is applied to the classification of straight lines. This also adds a useful feature for capturing user intent. Thus, the system can intelligently classify curves.

Furthermore, the proposed method can distinguish real free form curves from various over-drawn cases of conics, by checking self-intersection conditions. This allows users to draw in more natural ways in terms of various over-drawn sketching. In the published work of [Chen 1996], [Jenkins 1992], It seems that no attention is given to over-drawn cases.

Similarly for segmentation processing, this classification and identification processing can accept users’ interactive correction. If the system misinterprets users’ intent, or fails to give a proper classification, the users have the opportunity to correct the sketch interactively. This makes the system very flexible, user-friendly and practical.

The examples show that this intelligent system can interpret the user’s intention properly, effectively and robustly, based on adaptive linearity thresholds and existence of changes of curves’ convexity. From real-time sketches, the system can give proper classification and curve fitting in variety of 2D shapes: straight lines, circles, arcs, ellipses, elliptical arcs, spring lines and free-form curves. This technique is therefore suitable for dealing with vague and imprecise information from sketching.
Chapter 4 Geometric Constraint Solver and 2D Geometry

4.1 Introduction

As a result of the segmentation, classification and identification processing, a set of 2D primitives including straight lines, circles, circular arcs, ellipses and elliptical arcs, or B-splines is obtained. These 2D entities are roughly placed at their proper positions and directions. But in general, they are not connected together in a way that reflects a user’s intent. This task will be dealt with within this chapter. Our aim is to capture the designers’ intentions and to replace and modify those initial sketches with a more geometrically exact version as a generic geometry, which can be further modified based on variational geometry. The task includes three parts: (1) inferring constraints that are desirable for a given rough sketch; (2) finding a solution to satisfy the constraints wherever possible; and (3) finally modifying drawings to desired 2D geometry. The results from this processing can be further manipulated or even passed directly to other CAD packages.

4.2. Related Work

A variety of related work concerning geometric constraint solvers for finding the configurations of a collection of geometric entities that meet a set of geometric constraints has been studied.

Solving a geometric constraint system is a problem that has been considered by several communities, ranging from automated geometry theorem proving [Chou 1987], [Wu 1986], describing mechanical assemblies [Roller 1998], constraint-based sketching and design [Sutherland 1963], [Shpitalni 1997], geometric modelling for CAD [Hoffmann 1996], and kinematics of robots and other mechanisms [Hartenberg 1964], [Kramer 1992]. In constraint-based design and modelling systems, constraint solvers allow designing generic objects, rather than explicit ones. Informally, a geometric constraint problem (model) comprises a (finite) set of geometric elements $p$ and a (finite)
set of constraints between them. The constraints are relationships between geometric elements, which limit translational, rotational and dimensional degrees of freedom for individual elements. The geometric elements are drawn from a fixed universe, such as points, lines, circles and conics in 2D or points, lines, planes, cylinders and spheres in 3D. The constraints can be topological (structural) constraints, such as incidence, tangency, parallelism, perpendicularity, etc., or metric (dimensional) constraints such as distance or angle [Hoffmann 1998]. The usual geometric constraints can be represented either by algebraic equations, or by predicates (declarative languages). Generally speaking, predicate formulation is suited to qualitative approaches, and the other one is suited to numerical methods [Michelucci 1998].

Geometric constraint based models can only be solved if they are well-constrained, which means that there are exactly enough constraints in the model to uniquely determine the parameters of all geometric elements in it. When some parameters are determined in different ways by multiple groups of constraints, the model is called overconstrained. When some parameters can not be decided by the constraints, the model is called underconstrained [Noort 1998]. In principle, an overconstrained situation can be repaired by choosing some of the involved constraints to be adjusted, or removed from the model, while an underconstrained situation can be repaired by adding constraints, such as default constraints.

When solving geometric constraint problems, a solver must produce an instance of a given topology that exactly satisfies given constraints. Approaches can be divided into two main categories: (1) parametric geometry, in which dimensional constraints are dealt as parameters of a design so that desired shape can be constructed sequentially according to predefined scheme and order with topological constraints [Roller 1991]; and (2) variational geometry, where constraints are given by an arbitrary scheme in no particular order, and the solver must then automatically derive a solution strategy for constructing the desired shape. The parametric approach is more stable and
controllable for finding a solution, since it uses a predefined scheme of dimensions and a predefined evaluation order. Thus, it is more common in commercial CAD systems. However, demands for a pre-specified dimensioning scheme and order limit the freedom of the design to modify the shape definition. The variational geometry approach has none of the above limitations, but it is more difficult to apply because the solver must be capable of: deriving a solution sequence; dealing with large sets of simultaneous non-linear equations; managing multiple solutions; and identifying user intent as to the most plausible solution [Shpitalni 1997].

Either equation or constructive solvers are usually used for tackling the variational geometry problems.

4.2.1 Equation Solvers

The equation solvers translate geometric constraints into a system of algebraic equations which are then solved using different iterative techniques.

Numerical Approaches

Numerical constraint solvers are quite general. However, most numerical methods have problems in handling generic overconstrained and underconstrained tasks.

Early systems [Sutherland 1963], [Borning 1981] used relaxation methods to solve the system equations. Relaxation methods work by perturbing the values assigned to variables in such a way that the total error is minimised. The main problem is that convergence is rather slow.

One of the most widely used numerical techniques is the well-known Newton-Raphson iterative method. When based on Newton-Raphson iteration, solvers require a good initial value. If, as is usual, the initial values are taken from a rough sketch defined by users, the sketch must almost satisfy all the constraints. The Newton-Raphson method will find one solution, closest to the initial guess, since it is a local method and cannot find a more suitable alternative. It is, therefore,
inappropriate when the initial sketch may lead to an unwanted solution. Solvers as those of [Nelson 1985], [Lin 1981] are based on a Newton iteration technique.

Solano and Brunet [1994], proposed solutions for constraint-based modelling, which also uses a numerical solver. Their system first deals with sequential constraints and then solves circularly interdependent constraints.

**Symbolic Approaches**

In the symbolic approach [Wu 1986], [Buchberger 1985], geometric constraints are also translated into algebraic equations (polynomials) with the same roots as the original problem. The symbolic approach first uses general symbolic methods, such as Wu-Ritt’s characteristic method, or the Gröbner basis method to change the equation set to new forms that are easy to solve, and then to solve the new equations numerically. Both methods can solve general nonlinear systems of algebraic equations, but they require exponential running times. In general, the problem of simultaneously solving \( n \) polynomials with \( n \) parameters is reduced to \( n \) steps of solving one polynomial with one parameter, and all possible solutions can be computed.

In Reference [Kondo 1990], [Kondo 1992], Kondo considers addition and deletion of constraints using Buchberger’s algorithm to derive a polynomial that gives the relationship between the deleted and added constraints.

**Propagation Methods**

Constraints propagation was a popular approach in early constraint solving systems. The constraints are first converted into a system of equations, and an undirected graph whose nodes are the system variables and constants, and whose edges represent equations relating these variables and constants is generated. The propagation method attempts to direct the graph edges so that every
equation can be solved incrementally, initially only from the constants. To succeed, various propagation techniques [Latham 1996], [Steele 1980], [Gao 1998] have been tried, but none of them is guaranteed to derive a solution when one exists, and most have failed when cyclic interdependence situations occur.

4.2.2 Constructive Solvers

Constructive constraint solvers take advantage of the fact that most configurations in an engineering drawing are solvable by ruler, compass, and protractor, or by using another less classical repertoire of construction steps. The constraints in this method are satisfied constructively by placing geometric elements in some order that may, or may not be fixed. This is more natural for the user, and it makes the approach suitable for interactively debugging a sketch. The two main approaches commonly are classified as: rule-based; and graph-based constructive solvers.

Rule-Based Approach

In this approach, constraints are expressed by predicates, and geometric construction operations by functional expressions. Rule-based solvers construct a symbolic solution of the constraint problem, using a rewriting rule system to discover and execute a sequence of geometric operations, which build the object that satisfies all the constraints. If the constraints consistently describe the position and orientation of the object, then the constraint problem can be solved. This approach needs extensive computations to search and match rewritten rules [Joan-Arinyo 1997], [Bouma 1995], [Verroust 1992], [Aldefeld 1988].

Graph-Based Approach

Graph-based solvers have two phases [Fudos 1997]. First, a graph representing the constraints is analysed, and a sequence of construction steps is derived. Second, the construction steps are executed to derive the solution. DCM, a commercial constraint solver described in [Owen 1991] uses this approach: a graph is broken up into a set of subgraphs, such that an algebraic solution for
each subgraph exists. Then, the subgraphs are positioned using rigid body transformation to all geometries that belong to the subgraph.

This approach is faster and more methodical than the rule constructive approach. However, as the repertoire of possible constraints increases, the graph analysis algorithm has be to modified.

4.2.3 Degree of Freedom Analysis

In this method, the notion of degrees of freedom is associated to primitive geometric objects and constraints. Any geometric element has a number of degrees of freedom in its embedding space, while any constraint will reduce the degree of freedom of an object. Many geometric constraint problems can be solved by reasoning symbolically the geometric entities themselves, using the degrees of freedom analysis technique. In this approach, a plan of measurements and actions [Kramer 1992] is devised to satisfy each constraint incrementally, thus monotonically decreasing the system’s remaining degrees of freedom. This plan is used to solve, in a maximally decoupled form, the equation resulting from an algebraic representation of the problem. Degrees of freedom analysis results in a polynomial time. This approach seems more robust and more efficient in comparison with numerical solvers. In Reference [Noort 1998], this approach is applied to deal with over-and under-constrained geometric models.

Generally speaking, the above systems rely heavily on user interaction to produce the constraints either by stating relations, or by adding dimensions. These systems also focus on *rc-configuration* problems in which a diagram can be drawn using ruler and compass.

4.3. Overview of the Proposed Method

In general, our system uses not only constraints as a design aid in the creation of a 2D geometry, as in [Pavlidis 1985], [Jenhins 1992], but also uses geometric constraints for modification of the geometry after it is created. The system can first automatically infer constraints during sketching,
using its inference engine, and then modify the sketches to a generic model to satisfy the constraints, using its constraint solver. The solver is based on several constructive principles, degrees of freedom analysis, and scenario analysis. After receiving the generic geometry, its variation corresponding to changes of geometric element(s) can be obtained by re-applying the solver. This method converges quickly to one of many possible solutions, and therefore, it is very suitable for on-line applications.

### 4.4 Constraint Inference Engine

Once the closest fitting primitives have been found, the system constraint inference engine tries to infer certain relationships between them. Relationships can be classified into three categories: connectivity relations, unitary relations and pairwise relations [Jenkin 1992]. In the following terminology, we use terms: entities or elements instead of primitives.

#### 4.4.1 Type-1 Connections

The inference engine searches for connectivity relations first. From the current primitive to a previous one, it looks at the end points of this pair of primitives to determine whether they are within a certain adaptive distance tolerance. If they are, the two end points of the two primitives are connected. The adaptive distance tolerance is related to the lengths of lines or radii of arcs, e.g., 20% of the line lengths or 30% of the radii. In this case, relation code 1 is assigned (default is 0, meaning free end). If a connection relation is found, the system will record this connection information, the involved element index and end point codes, such as starting points (coded 0) or ending points (coded 1). Obviously, one end of a primitive may have several type-1 connections. If one end of a primitive may have a zero type-1 connection, the end of the primitive is called a free-end. The type-1 connections are shown in Figure 4.1, in which lines, circular arcs, elliptical arcs and
free-form curves are linked together, forming a closed contour. Here, an adaptive distance tolerance is applied, since a simple approach of joining the end points that are closer than a minimal threshold distance will not suffice; a threshold that is too large may eliminate fine details in connections, and a too small threshold may leave adjacent end points unlinked. Different tolerances may be needed for different parts of the sketch, and certainly for different users. Using adaptive distance tolerance can roughly satisfy these criteria.

![Figure 4.1 Type-1 connections](image1)

![Figure 4.2. Type-2 connections](image2)

### 4.4.2 Type-2 Connections

After establishing the type-1 connections, the inference engine tries to infer second or third type relations for a free end. A second type connection is a touching relation, in which an end point of a primitive falls on the path of another primitive. The relation code in this case is 2, shown in Figure 4.2. To detect the second type connection, the following procedure is conducted:
Step 1: Compute the distance between a free-end point and an involved geometric primitive;

Step 2: Check if the distance is within an adaptive threshold related to the pair of primitives; if not, search for next primitive;

Step 3: Find an intersection point near the free end point;

Step 4: Determine whether the intersection point is on the involved primitive segment, if so, the free end point has a type-2 connection with the involved primitive, if not, search for another primitive.

Currently, most commonly used type-2 constraints, shown in Figure 4.2, are implemented in our system. The adaptive threshold is also employed here, with the same consideration as in detecting type-1 connections.

4.4.3 Type-3 Connections

The third type of connection (relation code 3) is a tangent relation shown in Figure 4.3, in which one end of a primitive is tangent to another, as between lines, circles, ellipses, arcs, or elliptical arcs.

![Type-3 connections](image)

Figure 4.3 Type-3 connections

To determine the type-3 constraints, two steps are applied. First step is to check if a free end of a primitive is on another primitive, the procedure is the same as in detecting type-2 constraints. The
The second step is to further determine whether the connection is satisfied by the tangency constraints. The tangency constraints vary for different pairs of primitives. Taking a pair of a line and an arc, as an example, we can recognise if the connection is a tangent, using the distance between the line and the centre of the arc. If the distance is close to the radius length of the arc (adaptive threshold), the connection is a type-3 one.

![Figure 4.4 type-4 connections](image)

### 4.4.4 Type 4 Connections and Discussion

The fourth type constraints apply when an ellipse is tangent to, or on the path of another primitive as illustrated in Figure 4.4, and the type relation code is 4. This constraint is a special case of a general tangency relationship, in which two elements can be tangent at a general position for both of them. The type-4 constraints are application oriented. This type of relations is used specifically for inferring objects formed by rotations. When checking the type-4 constraints, the system first computes an intersection point of the involved primitive and the ellipse’s major axis, and then determines the distance between the intersection point and the ellipse centre. If the distance is near to the radius length of the major axis, the connection is regarded as of type-4.

Figure 4.5 gives some examples of applications of different connections. When sketching slot features from a box or a cylindrical object, users will meet first and second type constraints (Fig. 4.5...
After feature recognising, the top-lines in Fig. 4.5(a) will be broken (partially erased) by solid ‘—’ operations. When silhouette lines are drawn to express a cylindrical object or a feature (Fig. 4.5(b)), the third type constraints will be obtained. The type-4 constraints will be met, when an ellipse from a projection of a section circle touches the path of a silhouette curve, to express a revolution feature (Fig. 4.5(c)).

![Fig. 4.5 Examples of different connections](image)

(a) type-2  (b) type-3  (c) type 4

4.4.5 Unitary Relations

The unitary constraints are properties of a single primitive on its own. The unitary constraints apply to lines, ellipses, arcs, and elliptical arcs. For the lines, the engine examines the slope of the straight line to classify it as one of a set of special directions: horizontal, vertical, or isometric projection of the principle axes, as shown in Figure 4.6. If it is classified, the straight line will be assigned corresponding unitary constraint code: HOR, VER (or ISO-Y), ISO-X, and ISO-Z. Subsequently, this line will be changed to its corresponding direction. For an ellipse, the system checks if the direction of one of its axes is close to one of the special directions. If so, the ellipse gets the same unitary relation code as in the line case. In case of circular arc, the angles between the
horizontal line and the lines from the centre of the arc to its two end points are examined and checked for matching of the special direction angles. If so, the angle subtended by the arc will be changed. For an elliptical arc, the system first checks its directions, as for an ellipse, and then examines its start and end angles, as for a circular arc. The unitary constraints are directional constraints.

![Special Directions](image)

**Figure 4.6 Special Directions**

### 4.4.6 Pairwise Relations

The pairwise constraints are geometric relations shared by two primitives [Jenkins 1992]. Currently, the system supports parallelism and perpendicularly relations between pairs of lines, ellipses, or elliptical arcs. Each line, or ellipse, may have one of the pairwise relations: either parallelism, or perpendicularity, with a number of previous primitives. The system searches backwards for parallelism and perpendicularity relations (codes are PRAR and PERP, respectively). Once this relation is found, the system will stop further search, if not, the search will stop when the first element is reached. These pairwise relations also provide directional constraints to the system.
Another consideration for finding constraint relationships is to reduce the number of relations generated. Obviously, this can be great, even for moderately simple sketches. A number of methods have been employed to reduce the number of relations produced, for example, timing stamping and sequence stamping techniques [Jenkins 1992]. In the timing stamping, if the time difference between each start time is greater than a given threshold, this pair of elements will be prevented from analysing the pairwise relations. Similarly, in sequence stamping, if the difference between two primitives' sequence numbers (assuming each primitive has a unique successive number) exceeds a predefined range, the pair of elements is prevented from entering the relation engine. As Jenkins and Martin stated in [1992], both time and sequence stamping try to exploit a habit of designers when drawing, which is that related details of a sketch tend to be drawn in roughly the same time frame, and both techniques can reduce significantly the number of relations generated. But, they suffer if the user changes his/her attention from the current to another part of the sketch. In our system, one method to reduce the number of connection relations is to use backward search by exploiting the fact that sketches are drawn one stroke after another, in order. A current stroke refers to previous ones. For the current element, the search for connection relations goes only backward from previous element until the first one is reached. The second method is to build up a stop-search mechanism during pairwise relation backward detection, since if one element has a pairwise relation already, the directional change is well-constrained and any more pairwise relation will become redundant, or over-constrained. These two methods can reduce the number of relations satisfactorily and do not have drawbacks of the time stamping and sequence stamping techniques.

4.5 Analysis of Object Degrees of Freedom

Once the variety of constraints (relations) is obtained, the next step is to modify individual primitives to satisfy all constraints, or to find a satisfactory solution. The system first analyses the
Table 4-1 Object degrees of freedom

<table>
<thead>
<tr>
<th>Constraint Type</th>
<th>Translational DF</th>
<th>Rotational DF</th>
<th>Dimensional DF</th>
<th>Total ODF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Arc</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Elliptical arc</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Circle</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Ellipse</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>B-spline</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

In my system, there are no explicit dimensional constraints. Thus, each primitive may vary in location or size. This means that objects in the system are not rigid-objects. Different primitives have different object degrees of freedom, in accordance with different construction limitations. The degrees of freedom (DF) for each primitive are shown in Table 4-1. For example, a circle has no rotational DF, because it is a perfect symmetry; it also has no dimensional DF, which means that its radius is constant during construction processing. This limitation is applied in most of the constraint solvers. Similarly, this dimensional restriction is applied on ellipses. For a circular arc, or elliptical arc, one dimensional DF is given to allow the system to change its extended angle, but not for changing its radius (or radii). I fix the radii of circles and ellipses, since changes of radii may make extra movements for their connected primitives.

4.5.3 Constrained Degrees of Freedom

Different constraints have different constrained degrees of freedom. The CDF for each type of constraints are given in Table 4-2. For the type-1 constraint relations, they are incidence constraints, which restrict two object’s translational degrees of freedom. The second type constraints requires
one end point of a primitive to move on its partner curve. So, this type constraint eliminates one object’s translational degree. The third type constraints remove one object’s translational degree of freedom, as in the second type, and exclude one object’s rotational degree of freedom by requiring a tangency relation. The \textit{type-4} constraints eliminate only one object’s rotational degree of freedom. The last two types of constraints (unitary relations and pairwise relations) only discharge one object’s rotational degree of freedom.

Table 4-2 Constrained degrees of freedom

<table>
<thead>
<tr>
<th>Constraints</th>
<th>CDF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type 1 relation</td>
<td>2</td>
</tr>
<tr>
<td>Type 2 relation</td>
<td>1</td>
</tr>
<tr>
<td>Type 3 relation</td>
<td>2</td>
</tr>
<tr>
<td>Type 4 relation</td>
<td>1</td>
</tr>
<tr>
<td>Unitary relation</td>
<td>1</td>
</tr>
<tr>
<td>pairwise relation</td>
<td>1</td>
</tr>
</tbody>
</table>

4.6 The Constraint Solver

To configure geometry for all sketched 2D primitives that satisfy involved constraints for the different types of primitives, the system first analyses configuration degrees of freedom, and then produces construction rules (or steps) according to several general construction strategies. All these construction rules are pre-coded in the system to give a local solution by internally solving the necessary constraint equations.

4.6.1 General Construction strategies

When solving the system constraints, the following general construction strategies are applied to all types of primitives to generate construction steps.
(1) **One-stop policy:** the constraint solver tries to find a possible configuration for one primitive each time, incrementally from the first element to the last one. Therefore, the solver only considers a local configuration problem related to the current primitive, which means that only constraints concerning the current element will be treated. In comparison with considering a global configuration, this local constraint problem is simpler and more easily solved. Moreover, the system constructs each primitive in a sequence order, which is quite suitable for on-line applications. When the users finish drawing the last stroke, the system will complete the configurations. The solver can find only one possible solution for an overall constraint problem.

(2) **One-side policy:** when dealing with the current element, the solver ignores all constraint relations between the current entity and primitives generated after it. This means that only constraints between the current element and previous ones (on one-side of it) will be solved. This strategy further reduces the number of constraints to be treated, and complexity of the problem. This policy considers the fact that when sketching a current stroke, the user mainly takes previous elements as reference objects to form new constraints, although some intentions might be born at this moment. If the user stops the drawing after the current stroke, the system would still give a possible solution.

![Figure 4.7 Applications of minimal movement policy](image)
(3) **Minimal movement policy**: if the current element is required to change its location, or size to satisfy one or more constraints, the solver should try its best to keep movements minimal, since original position and size of the element represent users' initial intent. This policy attempts to capture users' intent more accurately. Figure 4.7 gives two examples of applications of this minimal movement strategy. A straight line AB is given a constraint: point B incidence of a fixed point C (see Fig. 4.7(a)). To meet this constraint, the solver just moves point B to point C, keeping point A unmoved, as shown in Fig. 4.7(b). This solution obeys the minimal movement policy. An alternative solution is given in Fig. 4.7(c), by translating point B and point A simultaneously to meet point B and point C. Although this solution can satisfy the constraint, it is still banned because it moves point A unnecessarily, damaging the minimal movement policy. In Fig. 4.7(d), a straight line (dashed line) needs to be modified to a vertical line. In accordance with the current policy, the solver rotates it about its middle point, to a vertical line (solid line). The system does not take into account the second solution (Fig. 4.7(e)), which rotates the line about its one end to form a vertical line, because that places the resultant line too far from the original one.

(4) **Use of default constraints**: when a primitive is underconstrained, the solver should always make use of default constraints to give a configuration, subjected to the above general policies. For instance, if a straight line is free of any constraints, the solver simply takes two current end positions as its two default incidence constraints of its two ends. If a primitive is constructed under some default constraints, it still can be further modified if this modification is necessary.

(5) **Backward propagation**: if the current entity has some constraints with previous elements, the solver first tries to modify the current entity to satisfy the constraints, and tries to keep previous entities unchanged, although the constraints could be met by changing previous elements, either. Otherwise, once a new constraint is added, all previous elements will be changed one-
by-one to re-satisfy constraints in a constraint chain from the current entity to the first one. This will not only lead to heavy computation and instability, but will also harm the minimal movement policy. Actually, the backward propagation strategy can be introduced from the minimal movement policy.

![Diagram](image)

Figure 4.8 Fixed joint policy

(6) **Fixed joint policy**: if any two elements meet at their end points by solving some related constraints, the common point will become a fixed joint. Once a fixed joint is formed, it can not be further modified. In Figure 4.8 we give examples of dealing with fixed joints and default constraints. Geometric primitives are generated in sequence order, the sequence numbers are displayed in Fig. 4.8(a). Line 1 and line 2 form one joint, and line 3 and line 4 yield the other joint. Now, line 5 comes to link two joints, with another HOR directional constraint. This is an overconstrained case, since fixed joints can not be moved. In order to give a solution, the solver simple removes the HOR directional constraint to make line 5 well-constrained, although it seems unwise, and finally connects line 5 between the two fixed joints from Fig. 4.8(b). Line 6 is different, although it has two type-1 constraints with two end points of line 1 and line 4, and a HOR directional constraint, which seems the same as for line 5. The difference between line 5 and line 6 is that the two end points to be linked by line 6 are both determined by default constraints. Both end points can be changed to further satisfy the constraints for a horizontal
line. So, we can first change the line to a horizontal one, then compute new intersection points between the temporary horizontal line, and line 1 or line 4, respectively Fig. 4.8(b). In this system, I put connection constraints higher priority than unitary constraints, since connection constraints are more important than unitary constraints for interpreting 3D objects. On the other hand, if the system is further developed with more constraint types such as snapping to the nearest grid point, the two bottom ends of line 2 and line 3 would be at the same height level. As a result, line 5 and line 6 would be both horizontal lines.

4.6.2 Generation of Construction Steps

Before considering how to configure a new element, analysis of its constraint and degrees of freedom is performed. Then a decision of how to construct the new entity in steps is made according to the analysis result. Tables 4-3 (a), (b) and (c) illustrate a case study for a line configuration. In the scenario column of Table 4-3 (a), the symbol ‘ ’ means a default fixed point. In general, there are 14 combinations of different constraints. Only three of them are well-constrained (Table 4-3 (b)), two cases are over-constrained (Table 4-3 (c)), and most of them (nine cases) are under-constrained (Table 4-3 (a)). This means that in most of the cases a possible solution can be found easily, under the current solving strategies. The main concern is how to add default constraint(s) and solve constraint equations, since after choosing the default constraints, the construction steps become very clear. Furthermore, most of the cases need only one-step construction, in accordance with the added default constraints. If the constraints include one unitary, or pairwise (parallelism or perpendicularity) relation, in general, two steps are needed. The two steps can be separately performed, by firstly modifying the current entity to meet the constraint and then simply focusing on the predefined one-step construction.

As a result of the above analysis, construction steps under the general strategies can be given in the following order:
- Change the current element to satisfy its unitary or pairwise relation, if necessary;
- Select one construction step, according to the remaining constraint, and solve its corresponding constraint equations.

For example, in case No 5 (Table 4-3 (a)), a line is constrained with a unitary relation (VER) and one second type relation. We can solve the problem in two steps. First rotating the line around its middle point to meet the unitary relation. This operation is the same as in case No 2. After that, the problem will become the same as in case No 3. Thus, the second step is to extend the line to the partner object, by finding an intersection point. In these three cases, all program routines for changing direction and obtaining intersection are separately reusable and combinable. This can not only save development time, but also reduces the number of constructive methods. Case No 4 can be solved by finding a tangent line (from the two possible) which is closer to the initial line, similarly to the solution of case No 7, and further extending the line to the default length. In cases No 8 and No 9, after treating a unitary or pairwise relation, the following step just moves the line in parallel to an incident point or a tangent point. The next three cases (Table 4-3 (b)) are well-constrained. They need only one step to solve their constraint equations, which depends on the types of involved primitives. The last two cases (Table 4-3 (c)) are over-constrained, and the construction methods for them are the same as stated in previous section (Fig. 4.8).

B-spline curves have been restricted to have only type-1 constraints. Their constraint problems are just special cases of lines, as in cases No 1, No 6 and No 10 (Table 4-3 (a), (b)). The solver simply assigns incident points to their end points.

Circular and elliptical arcs are open sections with two ends, and have the same 4 object degrees of freedom as lines. They also have the same types of constraints to be applied as lines. From a degrees of freedom analysis point of view, topologically speaking, they are from the same class of objects as lines, although they are different dimensional objects, with respect to the number of their
representing parameters. Therefore, construction steps for arcs are similar to those for lines under the same constraint combination. Each line case has a corresponding case for arcs. Considering case No 3 (Table 4-3 (a)) as an example, the construction method for lines finds intersection point between two lines with one default end. For arcs (circular or elliptical), the construction method still finds an intersection point, but between a circular arc, or an elliptical arc, and a line with one default end, as well. The only difference is the use of different equations to obtain the intersection point.

The construction method is the same. However, difficulties of solving different constraint equations may arise, especially when elliptical elements are involved. To some extent, this prevents the system from implementing some constraints attached to elliptical elements. This also may be the reason why most of the referenced systems consider only rc-configuration problems. Those systems involve only lines, circles and arcs. For example, finding a tangent line between a point and a circle is much easier than finding such between a point and an arbitrary ellipse.

A circle can only move in 2D under current assumption with a constant radius. So, its construction is always to find a displacement of its centre point, and then shift it to a desired position. An ellipse direction can be changed under a unitary or pairwise constraint, and also its centre points can be shifted as in the case of a circle.
<table>
<thead>
<tr>
<th>Scenario</th>
<th>No.</th>
<th>Constraints U-unitary, P-pairwise</th>
<th>ODF</th>
<th>CDF</th>
<th>CF</th>
<th>Added Default constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td></td>
<td>4</td>
<td>0</td>
<td>4</td>
<td>2 ends</td>
</tr>
<tr>
<td>2</td>
<td>1 U or 1 P</td>
<td></td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>1 end, 1 dim.</td>
</tr>
<tr>
<td>3</td>
<td>1 type-2</td>
<td></td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>1 end, 1 rot.</td>
</tr>
<tr>
<td>4</td>
<td>1 type-4</td>
<td></td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>1 end, 1 dim.</td>
</tr>
<tr>
<td>5</td>
<td>1 type-2 and 1 U. or 1 P</td>
<td></td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>1 end</td>
</tr>
<tr>
<td>6</td>
<td>1 type-1</td>
<td></td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>1 end</td>
</tr>
<tr>
<td>7</td>
<td>1 type-3</td>
<td></td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>1 end</td>
</tr>
<tr>
<td>8</td>
<td>1 type-1 and 1 U. or 1P</td>
<td></td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>1 dimensional</td>
</tr>
<tr>
<td>9</td>
<td>1 type-3 and 1 U. or 1 P</td>
<td></td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>1 dimensional</td>
</tr>
</tbody>
</table>
4.7 Geometry in 2D

In our system, 2D geometry can be obtained in three different ways. This gives users a lot of freedom to quickly specify their geometric design ideas.

4.7.1 From Sketches to 2D Geometry
After solving the constraints inferred by the system inference engine, 2D primitives input by sketches can be properly configured as 2D geometry, using our constraint solver based on construction rules and degrees of freedom analysis. This method lets the users to work with the system in a more natural way. Figure 4.9 illustrates original input of sketches, which consists of several lines, arcs and a B-spline. Constraints involved in this case include type-1 and type-2 connections, e.g., a line touching an arc, and unitary relations, e.g., vertical lines. These constraints are detected successfully by the inference engine, and then are solved properly by the constraint solver. Figure 4.10 gives the result of this 2D configuration. It can be seen that the ellipse in the middle and the two vertical lines are changed under unitary constraints; the three lines are modified to touch the other primitive under type-2 constraint. All type-1 constraints are linked correctly.

Figure 4.9 Input of sketches
4.7.2 Combination of Sketches with Interactive Input to 2D Geometry

Besides sketched input, the system provides another menu-driven interactive input as well. The reason for this is either because some sketches are hard to draw, e.g., a line tangent to an ellipse, or because sometimes it is hard for the system to detect the constraints correctly, based on the sketches. For example, if two quite short lines are designed to meet, a small threshold value will be applied to detect their type-1 relation, due to their short lengths. However, the result of this detection may be wrong, because the distance between the two ends of LS fitted lines may be greater than the threshold. The reason for this is that the end points of fitted lines, normally are different from the end points of sketches, although sketches may be met at their ends. In this case, if 2D primitives can be input directly and interactively, it will be better for both the users and the system, since the interactive inputs, in general, is more accurate than the sketching.

Moreover, from a need to infer 3D objects from rough 2D input, this additional input method will give the users more freedom when working with the system. They can sketch 2D primitives, enter 2D elements interactively, or combine the two methods at any points. This makes the system user friendly.
Figure 4.11 shows an example of combination of sketch with menu-based interactive input. The two lines tangent to the circle are drawn interactively and others are drawn by sketching. The system first collects all input together and then sends them to an inference engine and the solver. The result of this combined input is given in Figure 4.12. From this example, it can be seen that the system can capture type-3 relations (tangency), and the solver can give a correct answer.
4.7.3 Variational Geometry in 2D

The second method of obtaining 2D geometry is using variational geometry, based on the constraint solver. After the drawing input stage, a generic geometry of the input can be obtained by using the inference engine and the solver. Afterwards, one can change the generic geometry to a specific one, by modifying desired positions of the primitives, and applying the solver again. Obviously, the newly created geometry has the same constraints as the generic one. In most commercial CAD systems, the process of geometry modification is required to select the dimension(s) to be changed, and to enter new dimensional value(s). In our system, one can simply select element(s) and change it (them), since dimensional constraints are integrated with construction rules and default constraints. In other words, the geometry in our system is derived from the current geometric primitives themselves and their constraints, and not from the explicit dimensions of generic objects as in commercial CAD systems. Thus, the system interface becomes very simple and friendly.

Figure 4.13 Sketches

Figures. 4.13 - 15 demonstrate the system's capability for geometry modification. In Fig. 4.13, a set of steps is represented by several lines. The inference engine finds their connection relations and
unitary constraints. Then the constraint solver gives a generic object as shown in Fig. 4.14. All lines are changed to either horizontal or vertical due to the corresponding unitary constraints. The graph is modified as a exactly closed section under the connection constraints. The last line (the horizontal bottom line) in the sketch is an over-constrained case (two connection constraints and one unitary constraint), the solver can give a correct answer to it. After receiving the generic object, we change the second vertical line (from left to right) to a slant line by moving its lower end point A roughly along northeast direction. Then the variational geometry is obtained by applying the solver again. As the result shows in Fig. 4.15, the overall horizontal size (as length of the bottom horizontal line) and the height of the first step become a bit bigger than the original ones. This is because the solver tries to keep the constraints and the default dimensions, after moving the end point A.

This variational geometry may be useful not only at the conceptual design stages, but also at the detailed design stages.

![Figure 4.14 Solving over-constrained case](image-url)
Figure 4.15 Variational geometry from the original sketches

This variational geometry is maybe useful not only at conceptual design stages, but also at detailed design stages. Combining with sketch and interactive input, 2D geometry in the system can be created in more flexible ways.

(a) Generic object

(b) After changing B

(c) After applying variable topology

Figure 4.16 One limitation of variable geometry
As Gossard et al. [1988] stated, one of the fundamental limitations of the variational geometry is that it can only produce a specific geometry with the same topology of the generic one. If an object is constrained to a fixed topology, sometimes, unrealisable objects may be generated. For example, a slot is located in a block shown in Fig. 4.16(a), with dimensional constraints $A$ and $B$ (as well as the default $C$). When the dimension $B$ is increased, so that $A+B>C$, the result is unrealisable (Fig. 4.16(b)). In this case, variable topology is recommended to eliminate the unrealisable object [Gossard 1988], and their result is shown in Fig. 4.16(c).

In contrast, our constraint solver generates a new geometry object based on the same constraints (topology), but it can configure over-constrained elements properly, according to the fixed joint policy. It can automatically keep or adjust some default dimensions. A similar case study is illustrated in Fig. 4.17. Fig. 4.17(a) gives an initial sketch and the corresponding generic object is shown in Fig. 4.17(b). After changing the dimension similar to $A$ from Fig. 4.16, the result is given in Fig. 4.17(c). Here, in difference of the case from Fig 4.16, the dimension similar to $B$ is not changed. Then, it is increased, and while increasing $B$, the plat on the right side keeps moving as in the case of increasing $A$. Thus, the final result is given in Fig. 4.17(d). So, my system interprets this case better (here is neither an unrealisable object, nor a topology change), comparing with the solution produced from Gossard’s et al. system [1988] (Fig. 4.16).

![Initial Sketch](image-url)
Figure 4.17 Comparative study of a conflict case

(b) a generic object

(c) changing position

(d) result
4.8 Examples and Discussion

Figure 4.18 shows sketches of a house with a square door, an elliptical window, etc. Before cleaning up, the primitives are not properly connected. For example, the elliptical window is not quite vertical or horizontal, the left wing line of the roof is not parallel to grid lines, the sun ray lines do not start exactly from the circle to outwards, etc. Nevertheless, their geometric constraints can be inferred by the inference engine, and the constraints can be solved by the system solver. Therefore, after tidying up, the 2D geometry in Figure 4.18 becomes more exact, which is shown in Figure 4.19.

Figure 4.18. 2D primitives before cleaning up
From all the above examples, we can draw the following conclusions:

(1) The system can infer geometric constraints automatically. Its inference engine works well with pre-defined constraint types, based on adaptive threshold criteria. In comparison, the adaptive tolerance threshold related only to a pair of primitives to be checked is better than a fixed tolerance to detect connection relations. It is also better than the clustering approach [Shpitalni M. 1997] in terms of computational load;

(2) The constraint solver based on construction rules and degrees of freedom analysis can quickly and properly gives one of many possible solutions. It determines primitives one by one, and does not need to solve a system of simultaneous non-linear equations iteratively;

(3) The inference engine and the constraint solver can deal with elliptical primitives and free-form curves, although some types of constraints are not currently implemented. However, most solvers only focus on rc-configuration problems [Gao 1998], [Aldefeld 1988], [Jenkins 1992], excluding elliptical primitives or free-form curves;
(4) The solver works directly on current sketches. No dimensional schema is required and users are not asked to add dimensions to the sketches, as in most commercial parametric CAD systems. The solver treats 2D primitives as semi-rigid-objects with a dimensional degree of freedom. For example, a line in 2D space has 4 object degrees of freedom in stead of 3 for a rigid-body. In this way, the solver treats the dimension information either as default constraint or as a changeable constraint, depending on object’s configuration degrees of freedom. In contrast, most geometric constraint solvers [Latham 1996], [Roller 1990], [Bouma 1995]. regard dimensional constraints as rigid constraints;

(5) The solver can give variational geometry in 2D, based on generic geometry and its constraints. It does not require users to specify dimensions, as most variational system [Lin 1981]. In comparison with similar systems Easel [Jenkin 1992] and Beautifier [Pavlidis 1985], they made use of constraints as a design aid in the creation of a part but did not use geometric constraints for geometry modification;

(6) The user can combine sketched input with menu-based interactive input, to quickly and effectively specify 2D primitives. This makes the system suitable for different users with different sketching skills or even preferences;

(7) The solver will require more precoded construction rules, if the number of constraint types is increased. This may be one drawback, but it can be offset against faster performance.
Chapter 5. Feature Interpretation and 3D Geometry

5.1 Introduction

After correction of 2D primitives, 2D geometry has its correct topology connections and correct primitives in terms of sizes and directions. The problem remaining is to recognise 3D objects from 2D topology and geometrical information. As studied in Chapter 1, labelling methods and optimisation schemes for a 3D reconstruction from a single view are not suitable for handling inaccurate drawings and on-line applications. Therefore, feature interpretation techniques have been investigated, based on perceptual analysis and 2D constraints, in accordance with an observation that when sketching, designers intend to draw related details of a design feature in roughly the same sequence frame. The design feature may be a modelling primitive such as a cylinder or a functional feature such as a slot. While drawing, the 2D geometry is accumulated until it can be recognised as a 3D feature. The feature is then placed in the 3D space, and a new feature can be interpreted and built upon previous ones incrementally. This method has very good performance in interaction, although it suffers from the problem of users moving their attention to another feature before entering enough information for its interpretation.

Related work on feature-based modelling can be addressed in two categories: feature recognition, and design with features.

5.1.1 Feature Recognition

Features are generic shapes. They can encapsulate design functions along with geometry which provides the essential information for manufacturing [Zulkifli 1999]. Hence, features have been identified in the engineering community as key elements in the integration of design and manufacturing. Obviously, a feature’s properties are application dependent. For example, shapes
such as holes, slots, and pockets are regarded as typical manufacturing features in their B-Rep description.

Feature based-models represent parts in terms of high-level functional entities, e.g., holes and grooves. By contrast, geometric models represent parts in terms of low-level geometric entities such as faces, edges, and vertices. The feature recognition [Nezis 1997], [Marefat 1990] involves identification and conversion of geometric/topological elements from an existing solid model, into predefined sets of features suitable for a particular application, for instance, computer aided process planning (CAPP) [Xu 1998]. Most of work in this field is based on the ideas of searching an edge graph of the model, and looking for patterns. The others use the idea of matching templates with parts of models to find features [Woodwark 1988], [Parry-Barwick and Bowyer 1995]. The principle advantage of feature recognition is the possibility of using current CAD database and geometric modellers for design. The problem of feature recognition is its limited domain of recognition techniques.

5.1.2 Design with Features

In the design with features approach, part models are defined directly with features. A typical feature modeller allows designers to add, subtract and manipulate features created as instances of predefined feature types. The feature-based models are therefore created during the design stage. However, the set of features used in design is limited, and the results are strictly context-dependent. Furthermore, the designers are likely to prefer the simpler user interface of drafting-based geometric modellers to the complex menu-based interface of a ‘pure’ design with features system. Therefore, most feature-based systems [Laakko 1993], [Martino 1994] combine ‘feature recognition’ with ‘design with features’, e.g., recognising simple features and interactively entering complex features.
5.2 Overview of Proposed Method

As noted, the feature recognition requires 3D geometric models as its input, whereas the design-by-feature requires explicitly pre-defined individual features. By contrast, in our system, 3D models are not available before feature interpretation, which is based on 2D input. Thus, feature recognition techniques can’t be applied directly in my system.

A hybrid technique is studied to combine feature interpretation with feature based design methods to develop a 3D inference (or modelling) engine for machined parts. The 3D inference engine analyses current accumulated 2D elements (geometric and topological information), and infer the elements as a 3D design feature, based on their 2D configuration and already - inferred 3D objects, incrementally. This method brings the following advantages:

- It makes input precise and efficient. Once a specific feature is recognised, it is not necessary to further input complete model data. For example, if one closed 2D profile with one extrusion edge is recognised as an extrusion object, further input of any edges will become unnecessary. In this case, the user can input any extrusion edge, no matter whether it is visible or not. So the user does not have to be interrupted in order to determine which edges should be inputted;

- Once the design is finished, the feature model is available. This makes it unnecessary to create a feature model by decomposing the solid model for further manufacturing process;

- It simulates parametric modelling methods in the design process. Design is conducted feature by feature in temporal order. The constraints and geometric variables are easy to set up and if using the design system as an interface to a commercial parametric CAD system, such as, Pro/Engineer, one can effectively integrate the conceptual design process with more detailed design.

Although the number of features required for modelling complex mechanical parts is huge, it could be reduced significantly if the main focus is on conceptual design, because conceptual design just
expresses rough design ideas by mainly solid primitives and their Boolean operations: union, subtraction and intersection.

In order to apply features to the design process, the system first recognises a feature from 2D freehand sketches. It then transforms the feature recognised to its proper 3D position. Once a feature is created, the user can examine it in a wireframe model or in a shaded solid model. The user can continue to add features, based on the wireframe or shaded model, by sketching in the 3D world. In this way, the system can support an iterative creative design process: thinking, creating, evaluating.

The feature interpretation is based on the knowledge of how to create 3D objects in current CAD systems. A common way for building cross-section 3D solid is to define a closed profile, then pull the profile along a path. For creating 3D open surfaces, several curves and predefined methods are needed. Currently, the focus is on 3D solid modelling.

5.3 Feature Design

5.3.1 Feature Types

Feature design is based on mechanical engineering applications. In general, the system may recognise any of the following features:

- Box feature, which can be transformed into protrusion of a rectangular shift (Fig. 5.1(a)), or depression of a rectangular hole, or slot (Fig. 5.1(b));

- Cylindrical feature, which can be recognised as a cylindrical shaft (Fig. 5.1(c)) or a hole including blind and through holes (Fig. 5.1(d)); For a hole, the system currently know its information, but the system do not display this feature by solid ‘-‘ operations.

- Revolution feature, which can be any revolution, solid or hollow (Fig. 5.1(e));

- Ruled surface feature (Fig. 5.1(f)); if two curves are drawn on two known reference planes and a straight line links the ends of the curves, a ruled surface can be produced;
- Swept surface (Fig. 5.1(g)); if two curves are drawn on two known reference planes and they meet at their ends, a swept surface can be produced.
- Modified feature, which can be a chamfer or a fillet (Fig. 5.1(h));
- Complex extrusion feature.

Due to limited developing time, this prototype system has incorporated box features, cylindrical features and simple revolution features.

![Diagram of various features](image)

Figure 5.1. Features: (a) box solid; (b) rectangular hole; (c) cylindrical shaft; (d) cylindrical hole; (e) revolution; (f) ruled surface; (g) swept surface; (h) chamfer

### 5.3.2 Internal Representation of Features

In this system, a box feature is defined as a generic cube. Its centre is located at origin of a local co-ordinate system with one unit size (see Figure 5.2(a)). For a general box with different
dimensions in length, width and height, it may be located at any position with different directions. This general box feature can be represented by applying three different scale transformations to its corresponding axes, shown in Figure 5.2(b), employing three different rotation transformations about its axes, and finally shipping to its correct positions.

Figure 5.2 Box feature; (a) generic feature; (b) general feature

A frustum feature is defined as a truncated cone along Z axis with base at \( z = 0 \), and top at \( z = \text{height} \) (see Figure 5.3). The base radius is at \( z = 0 \), and the top radius is at \( z = \text{height} \). After shipping and rotation transformation, it can be located at a general position.

Figure 5.3 A frustum feature
A free-form surface can be defined directly by 3D curves drawn on the reference planes. It is not necessary to apply a 3D transformation to it. The 3D curves can come from 2D sketches on a given reference plan or surface.

Each type of feature is internally represented by an object-oriented class, which encapsulates modelling data: dimensional and positional parameters, derived data from the model such as B-rep (Boundary Representation) information about faces, edges, and vertices, and its member functions (methods) for building the model, producing B-rep information, accessing the data and so on. Each feature model is an instance of its corresponding class. This representation can take all advantages from object-oriented designs, e.g., data encapsulation and code reuse through the inheritance mechanism [Pohl I., 1992]. Taking a box feature as an example, its corresponding class can be defined as follows.

```cpp
class CCube:public CPrimitive
{

protected:
    double length, width, height; //dimensional parameters
    double tx, ty, tz;             //location parameters
    double ax, ay, az;            //directional parameters
    int ref[][3];                 //index of refereed previous feature and its faces.
    int norm[5];                  //normal of faces
    double xv[5][5],yv[5][5],xv2[5][5],yv2[5][5],norm_dis[5][3];

public:
    void Draw2D();
    void Draw_frame3D();
    void DrawShade3D();

```
void generating_faces();
void get_data_of_faces();

This class named CCube is derived from an existing public class named Cprimitives. In the data field, we declare modelling data and derived data as protected type. The derived data include index of a referred object and its corresponding faces, as well as the box’s B-Rep information: directions of faces, normal distances of faces to the origin, and edges information. In the function field, besides the class’s construction and destruction functions, we declare model operating functions to display the feature in 2D geometry, 3D wireframe model, and 3D shaded model, and functions to produce B-Rep information and access their encapsulated data.

5.4 Inference Engine in 3D

5.4.1 General Interpretation Assumptions

In order to recognise 3D features, the system makes following assumptions:

- Two dimensional input is an isometric drawing. The origin of the isometric projection co-ordinate system is the same as the origin of the display window (lower-left corner of the display window). The system selects isometric drawing for two reasons. First one is that parallel lines on the objects appear as parallel lines in the drawing, and the second one is that edges parallel to the principle axes are drawn with lengths proportional to the actual dimensions of the objects (about 0.8165 of the actual dimensions);
- The projection co-ordinate system has the same scale as the display system (default value is 1);
- Dimension unit is a screen pixel;
- The projection co-ordinate system has the same directions as the feature definition system. The Y direction is upright, the positive Z direction is pointing out of the screen.
5.4.2 Inferring Feature

Our 3D inference engine expresses interpretation knowledge in knowledge rules, and integrates them into a programme by *IF-THEN* structures. The system examines combinations of 2D sketched elements and topology information to infer a 3D feature. Different features have different inference rules. A general extrusion object features a closed profile and an extrusion edge. The closed profile may consist of only one ellipse, or two pairs of parallel lines, or several line segments combined with arcs. So, the system first examines whether a closed profile exists, then finds the direction of the extrusion. Afterwards, the system determines where the closed profile comes from (reference plane) by checking if its centroid is within a projection area of a boundary plane of previous objects. Finally, the system will determine modelling parameters for a specific feature, based on known information about the reference plane and the extrusion direction, and will produce the feature in 3D. The 2D topology information is used here to find a closed profile and to determine which line is an extrusion line. For example, if the closed profile (representing a face) is an ellipse, the feature may be a cylindrical shift, or hole following the condition that extrusion edge is a straight line. Some inference rules are given in pseudo-code below:

Rules for a box feature:

*IF*

- the feature is composed of a closed profile and one extrusion line *AND*
- the closed profile is composed of 4 lines (two pairs of parallel lines) *AND*
- the extrusion direction has been determined (by the extrusion line) *AND*
- the reference plane has been found (default reference is *XOZ, XOY, or YOZ* planes corresponding to different extrusion directions)*

*THEN*
a box feature is defined.

Rules for a swept surface could be:

\[ IF \]
- the feature is composed of two curves \textit{AND}
- the two curves are end-connected \textit{AND}
- the two reference planes for the two curves belong to one box feature.

\[ THEN \]

a swept feature is found.

Once a feature's type is determined, the system will further infer its modelling parameters. As a result, the feature's parameters and its derived data will be stored in an Object-Oriented database by its class construction function. The details of the above processes are given below.

5.4.3 Finding Closed Profiles and Extrusion Edges

If a feature is not a free-form surface, it may be an extrusion object. For general extrusion objects, it is very important to find a closed profile and an extrusion edge. The system first determines if one element from the current collection is an extrusion edge by checking existence of one free end. If an element has no free ends, it means that it is connected with others at both end points. If an element has two free ends, it means that it is totally separated. An element will become an extrusion edge only if it has one free end and one fixed end. If there is no extrusion edge in the current collection, the system will wait for newcomers until it can find one extrusion edge.

For finding a closed profile, the system searches and checks connection relationships from the determined extrusion edge. If the extrusion edge has only one type-1 connection, e.g., line 1 and line 2 in Figure 5.4 (a), the current collection can't have a closed profile. The search, therefore, will be stopped. If it has one type-2 or type-3 connection with elliptical element, the closed profile is
found. For example, the ellipse from Figure 5.4 (b) will become a closed profile. If the extrusion edge has two or more type-1 connections, the system searches the connections from two sides as shown in Figure 5.4 (c); if they can meet together, all searched elements will form a closed profile.

(a) only one type-1 relation  (b) one type-2 or 3  (c) two or more connections

(c) single chamfer  (d) single fillet  (e) single cut  (f) combined chamfer

Figure 5.4 Finding closed profile and extrusion edges

For chamfer or fillet features as illustrated in Figures 5.4 (c) to (f), the extrusion edge has one type-1 connection and one type-2 connection, the searching procedure for a closed profile is similar to the procedure for Figure 5.4 (c), but taking the shorter path.
5.4.4 Determining Extrusion Directions

Figure 5.5 Determining directions of extrusions

Information about the directions of extrusions can be extracted from different elements, according to different features. For cylindrical features, the direction of extrusion can be inferred from the directions of ellipses (closed profiles). In most cases, the extrusion direction is along one of the three co-ordinate axes. From Figure 5.5 (a), it can be seen that if the direction of the ellipse is horizontal, the extrusion direction is along OY axis (vertical). In the same way, if the direction of the ellipse is $60^\circ$ (the ellipse on XOY plane) or $120^\circ$ (the ellipse on YOZ plane), the direction of extrusion should be along OZ axis, or OX axis correspondingly. If the direction of the ellipse is not along the above three axis, the system needs to further detect a reference plane on which the ellipse is drawn. If the reference plane is found, the normal direction of it is regarded as the extrusion direction. If not, the system will change the direction of the ellipse to its nearest special direction, then the extrusion direction is determined subsequently, since it is believed that general cylindrical feature is hard to draw.
For box features, directions of extrusion edges can be taken as extrusion directions (see Figure 5.5 (b)). Normally, the directions are along one of the three axes directions. As shown in Figure 5.5 (b), the extrusion direction will keep vertical although the box has the $\emptyset$ angle rotation about OY axis. If the directions are far from the special directions, the system has to find a reference plane on which the corresponding closed profile is drawn. If the reference plane exists, its normal direction is taken as the extrusion direction. If not, in a similar way to the cylindrical features, the system changes the directions of extrusion edges to their nearest special ones. The reason for this is that general box features (e.g. after two arbitrary rotation transformation) are believed to be too hard to be drawn.

5.4.5 Looking for Reference Planes

After finding closed profiles, extrusion edges and directions, the system has to have information about features' sizes and their positions (transformation information). Therefore, it needs to find a reference plane from previous 3D objects, for the current feature under interpretation. If the reference plane exists, the 3D transformation information can be extracted.

To determine a reference plane, the system first computes the centroid of a closed profile. If the inferring feature is a cylindrical object, its centroid is the centre of the ellipse (closed profile). If the feature is a non-cylindrical object, the centroid can be received by

$$x_{cf} = \frac{\sum_{i=1}^{n} x_i}{n} , \quad y_{cf} = \frac{\sum_{i=1}^{n} y_i}{n} .$$

Here, $n$ is the number of elements involved in the closed profile, $x_i$ and $y_i$ are a pair of coordinates of the end points for each element.

After obtaining the centroid position, the system continues to conduct a containment test between the centroid point and a polygon (or ellipse), which belongs to a visual face of a previous object. To retrieve information (B-Rep) about a visual face, the system simply calls a member function: *get_data_of_faces*, as defined in the previous section. If the visual face is an ellipse section,
obviously, this containment test can be easily done, by checking the intersection between the ellipse and a line from the centre of the ellipse to the centroid point. If the intersection point is between two end points of the line, the centroid is out of the ellipse.

If the visual face is represented by a polygon, an angle method is adapted for a general polygon: convex or concave, illustrated in Figure 5.6. The system first computes the sum of the angles subtended by each of the oriented edges as seen from the centroid point C. If the sum is zero, the point is outside the polygon. If the sum is $2\pi$ or $-2\pi$, the point is inside. The minus sign reflects whether the vertices of the polygon are ordered in a clockwise direction instead of counterclockwise.

![Figure 5.6 Angle method of containment test](image)

Actually, for a convex polygon, e.g., box features, the system can simply substitute the $x_{cd}$ and $y_{cd}$ co-ordinates of the centroid into line equation of each oriented edge. If all substitutions result in the same sign, the centroid is on the same side of each edge and is therefore surrounded.

From the above containment test, the reference plane can be determined if the centroid point is within the plane. If not, we simply take one of the default co-ordinate system planes (XOY, YOZ and XOZ) as a reference plane, in accordance with the extrusion direction. The reason for this is that if one object's centroid is out of its reference plane (imaged as a supporting plane), the combined objects will lose aesthetic balance.
5.4.6 Discussion On Computation of Rotation Angles

For the computation of rotational angles, we currently assume that features have only one rotation about one of the co-ordinate axes. The corresponding equation to be solved can be one of the equations: (B.9), (B.11), or (B.13) in the appendix B. The reasons for this are that if we give features two rotational degrees of freedom, the interpretation of the features will become difficult, and actually, illustrations will become hard to draw. For example, if the current drawings are referenced to the \( n \)-th reference plane, the corresponding transformations can be described as:

For the \( n \)-th reference plane

\[
\Gamma_n = [T^n][R_1^n][R_2^n][\Gamma_{n-1}]
\]

Here, the superscripts \( 1, 2, ..., n \) are used to denote the sequence order of reference planes.

Matrices \([R_1^n]\) and \([R_2^n]\) represent the \( i \)-th reference plane’s rotational matrixes about its local co-ordinate system which relates to the \((i-1)\)-th reference plane. \([T^n]\) represents the corresponding translational transformation matrix, and \([B]\) represents one of the base planes. It is clear from (B.14) that it is hard to draw 2D primitives on the \( n \)-th reference plane under the isometric projection \([T_c]\) (see Eq. B.1). On the other hand, if we do have some drawings on the reference, it is very difficult to find every transformational matrix, due to uncertain local co-ordinate system definition.

Actually, only after one rotation, it is still not easy to draw primitives properly, although the system can recognise the rotational angle. For example, five lines are drawn to represent a box feature as shown in Figure 5.7 (a). The system is tested to recognise the box feature and the rotation
angle about OY axis. Results are shown in Figure 5.7 (b) (a wireframe model) and Figure 5.7 (c) (a shaded model). It can be seen from Figure 5.7 (b) that the rotational angle is computed correctly, since the lines along ISO_Z before and after the system recognition are kept roughly parallel. However, the lines along ISO_X do not seem properly interpreted. This is because the closed profile in this example is not a perfect parallelogram. After the rotation about OY axis, the lines are in their general directions. The system can’t give any correction. This closed profile (not a parallelogram) also cause problems of computing the feature’s size. On the other hand, the angle computation is based on one of the edges of the closed profile. Selecting different edges will result in different angles. Therefore, detecting a rotational angle in our system needs additional work to be included. Our system currently does not perform the rotational angle computation.

Figure 5.7 (a) Initial inputs

Figure 5.7 (b) A wireframe model
In essence, locating a feature to a desired position may be done effectively in further 3D manipulation processes or packages. For example, one would like to draw something on a specific reference plane with rotations, one could transfer a model first to make the reference plane parallel to one of the base planes. Afterwards, one can draw a special positional feature and rotate it. These manipulation functions need to be further developed.

5.4.7 Computing Modelling Parameters

After obtaining a reference plane, we can compute modelling parameters: size parameters and transformational parameters, based on our internal representations of features. Parameters for different features can be obtained in different ways.

Parameters For Box Features

As discussed in the last section, we currently assume that box features are always parallel to the system’s co-ordinate planes (base planes: $XOY$, $YOZ$, and $ZOX$ planes). This will make on-line sketching easier.

On the basis of the assumption, the parameter computation processes can be divided into the following steps, illustrated by Figure 5.8.
- Step 1: finding size parameters: From the extrusion edge, find two edges of the corresponding closed profile linked to it, and then project these edges to their corresponding isometric. The projection lengths of the extrusion edge and other two edges are height, length, and width parameters of the box feature, respectively. As noted, the projection lengths have been foreshortened to 0.8165 times of its true dimensions. Figure 5.8 (a) shows a vertically up extrusion with a reference plane \( Y=H \), whereas Figure 5.8 (b) illustrates a vertically down depression with a reference plane \( Y=M \).

- Step 2: sorting out positions: On the basis of the reference plane, the system first computes the 3D co-ordinates of the fixed joint of the extrusion edge. Then it obtains the centre position of the box in 3D by taking its size parameters into account, as shown in Figures 5.8 (c) and (d).

- Step 3: determining transformational parameters: The system finds the transformational parameters in accordance with the internal feature representation model. For a box feature, the co-ordinate components of its centre position are its shipping transformational parameters along \( X, Y, \) and \( Z \) axes. For a special case, when extrusion direction is vertically down and the reference plane is on \( Y=0 \), the system changes the shipping transformational parameters to make sure that
the bottom face of the box is on the $Y=0$ base plane, since this special case is a normal default situation, in which users normally intend to put a box on the ground, although the reference plane is a base plane.

- Step 4: displaying a model: As a result of the above processes, the system has obtained all the modelling parameters for a box feature. Then the system just shows the feature in 3D, and calls the feature's member function to generate its face information both in 2D and in 3D for further use.

**Parameters For General Cylinder Features**

![Diagram of a cylinder feature with parameters](image)

**Figure 5.9** Computing modelling parameters for a cylinder feature

For general cylinder features, the processes of computing modelling parameters are similar to those for box features. Here, we just give an example as shown in Figure 5.9. In this example, an ellipse (solid) and a line (solid) are received from 2D input for a cylinder feature. From the ellipse,
the extrusion direction can be determined along OX direction. Let us say that the reference plane for the ellipse is found with X=K. Now, the system starts to find size information about the feature’s height, base radius, and top radius. To obtain the height of the cylinder, the system projects the line onto the ISOX axis. The projection length is the height. In the same way, the length L1 can be received. For the base radius (on the left side) and top radius (on the right side), the system uses proportional computations, based on the known ellipse. Second, the system determines the centre position of the ellipse in 3D, with reference to the plane X=K. Then, it finds the rotational parameter η = -90° about OY axis according to the extrusion direction, and the shipping parameters according to the centre position of the ellipse and the length L1. Finally, the system generates the visual end face’s information and displays the feature in 3D modelling space.

5.5 Menu-Based Interactive Input of 3D Projections

Similarly to the 2D input, users can use icon menus to input 3D projections. In this way, they can mix freehand sketching and interactive 3D projection input, to quickly specify 3D primitives. For example, the users can quickly specify a 3D box by drawing a diagonal line for a top face and
subsequently dragging the face vertically, to produce height information. From the menus, the users can input vertical cylinders, semi-cones, or cones. After an input of a 3D projection through the system menu, the system first looks for a reference plane for it. If the reference plane exists, the system will compute positional parameters and transformational parameters for it as for a normal input feature. If there is no any reference plane available, the system simply put the bottom face of the primitive on the XOZ base plane. This input method can speed up the modelling processes very well. The system currently do not support the input of sphere or torus.

On the other hand, the system provides assistant grid lines in accordance with the isometric projection. These grid lines can help users to sketch in 3D.

Figure 5.10 shows an example of directly and isometrically entering of 3D projections. Figure 5.11 gives a clear view from the front of the scene, which illustrates that all input primitives with $y=0$ reference planes are adjusted to intentional positions. The others with $y>0$ reference planes are located properly.

![Figure 5.11. Front view of inputted 3D primitives](image)

5.6 Variational Geometry in 3D

The Fig. 5.12 is employed to illustrate variational geometry in my system.
Figure 5.12 (a) Initial sketch

Figure 5.12 (b) After solving constraints

Figure 5.12 (c) Interpretation
Figure 5.12 (d) Changing 2D geometry

Figure 5.12 (e) The corresponding interpretation

Figure 5.12 (f) Sketching on a wireframe model
Figure 5.12 (g) Newly recognised feature

Figure 5.12 (h) A Shaded model after changing in 3D

Figure 5.12 (i) A wireframe model
In my system, once a feature is inferred and built up, the user can change it by modifying its 2D input, with our constraint solver. In this way, the user can quickly input 2D sketches. We demonstrate this process with Fig. 5.12(a) to Fig. 5.12(i). The initial sketches are as shown in Fig. 5.12 (a). The system will automatically recognise and tidy up 2D sketches (Fig. 5.12(b)) based on our constraint solver, and will produce a 3D model (Fig. 5.12 (c)). The model can be evaluated by the user. If results are unsatisfactory, the user can simply click on a menu to let the system delete the feature model just built, and can return back to 2D space. The initial 2D sketches can be adjusted to their desired positions (Fig. 5.12(d)), based on the constraint solver. Consequently, a variational geometry in 3D (Fig. 5.12(e)) can be received. Figure 5.12(f) shows an example of sketching directly on the previous 3D model, for a cylinder feature. The corresponding result is given in Fig. 5.12(g). After further modification, the cylinder feature is changed to a semi-cone feature (a shaded model in Fig. 5.12(h) and a wireframe model in Fig. 5.12(i)).

It is clear that the system not only supports feature interpretation, but also generates variational geometry.

5.7 Examples and Discussion

I used this prototype system to deal with a conceptual design geometric model of a lathe. This geometric model consists of 10 parts: two base parts, a headstock, a spindle, a gear box, a lead screw, a feed shaft, a carriage, a tailstock and a cross slide. The spindle is expressed as a cylindrical feature. The feed shaft is modelled as a revolution object. Most of the parts are represented as box features. Figure 13 gives a wireframe model of the lathe (with some 2D input geometry). The shaded model is shown in Fig. 14. When working on this model, we drew some
features on the previous wireframe model, and some features on the shaded model, because it was easier to draw on the 3D faces.

Figure 13. A wireframe model of a lathe

Figure 14. A shaded model of a lathe
Figure 15. A wireframe model of a scene

Figure 16. A shaded model of a scene

Figure 15 describes a model of a scene including a desk and a small bench. Figure 16 shows its shaded model. The left feet of the desk is a semi-cone, formed from a revolution feature. On the desk, there is a vertical shelf (a thin box), which holds a horizontal lamp tube (cylinder). A small clock (cone feature) is situated on the shelf. In front of the desk, there is a small bench, which is modelled by three box features.

The system combines interactive input of 2D and 3D primitives, with sketched input recognition for building 3D geometry. It also support variational geometry based on constraint solver. The
system gives users greater freedom to quickly specify 2D and 3D geometry, than those with sketched input only [Hwang 1994]. This mixed automatic feature interpretation and interactive design environment can encourage designers with poor sketching skill to use it for creative design tasks. In principle, the system has a potential capability of supporting 3D surface design. It can model scenes, which are difficult for the labelling scheme and optimisation-based methods, although it requires corresponding recognition knowledge for the different features.
Chapter 6 Case Studies and Conclusions

6.1 System Interface

This system is implemented on Windows’95, using Visual C++ and Open GL. A part of the system has also been developed on SGI workstation and UNIX platform, in C++, Motif and Open GL. Figure 6.1 shows the interface of SGI-based system, which runs segmentation and curve fitting. This partial system demonstrates the possibility of transferring a PC-based on-line sketching system to a UNIX platform.

![Figure 6.1 Interface of partial system on SGI](image)

Currently, the prototype system is implemented on PC P5/120 machine. The development environment is based on Visual C++ 6.0, and the OpenGL graphics library. The interface is shown in Figure 6.2. On the interface, the system menus are used to input some parameters such as sketching skills, and to perform some document functions. A main part of the interface is a toolbar icon menu. The 2D icon menus are optionally used for directly entering 2D primitives. They can
also be used for correcting a mis-interpretated curve segmentation and fitting, by following an indicator menu. The 3D icon menu is optionally employed to directly input 3D projections. The other menus are used to manage 2D and 3D geometry. All primitives in the Figure 6.2 are directly inputted from the toolbar menus.

Figure 6.2 Interface of the system on PC

6.2 Case Studies On Curve Segmentation and Fitting

In order to test the system’ segmentation and curve fitting abilities, four cases are designed to be studied. In the first case, each investigator is required to draw 10 strokes. Each stroke consists of at least two primitives. If any stroke is fitted with only one primitive, this means that the system has failed to find the intended segmentation points. In the second case, each participant is asked to draw 10 straight lines, because the lines are the most common elements. If any sketch is fitted with different primitive, or is broken into several line segments, this implies that the system has classified curves or has detected segmentation points incorrectly. In the third case, each tester is required to sketch 20 elliptical primitives including circles, arcs, elliptical arcs and ellipses. If any element is replaced by another type of primitives, this means that the system has segmented and has
fitted the curves improperly. In the fourth case, ten free-form curves are required to be drawn. If any sketch is represented by another primitive, it means that the system does not detect inflection points properly.

Three volunteers were invited to conduct the case studies. Figure 6.3 (a), (b) and (c) show results of Case-1. Results for Case-2, are given in figure 6.4. Figures 6.5 (a), (b) and (c) illustrate the results of Case-3. Results for Case four are shown in Figures 6.6 (a), (b) and (c).

In Figure 6.3 (a), one sketch at the lower-right corner is segmented incorrectly, resulting in three line fittings. Actually, the tester has tried to draw a “v” like shape. The system detects one extra segmentation point, which leads to relatively greater linearity for each sub-segment. Others are correctly segmented and fitted. In Figure 6.3 (b), and Figure 6.3 (c), all sketches were segmented correctly. For this case study, in total, only one of thirty sketches was dealt with wrongly.

In studies of Case-2, all of the three participants were satisfied with the results. The sketches of 10 straight lines for a tester are fitted perfectly, as shown in Figure 6.4.

In Figure 6.5 (a), three sketches are fitted with straight lines improperly for Case-3 (in this case study, non-LS lines are produced to show the end or broken points). The first one (No. 1) is segmented into two lines, because the intended arc is quite big and the sketching speed is very high. During the sketching, the tester changed the drawing speed in the middle, which suggests a segmentation point. The second misinterpreted sketch (No. 2) is a similar case. The third one (No. 3) is relatively short arc. However, the drawing speed is high too, which implies that the user pays less attention to it. Therefore, its corresponding linearity is increasing, finally resulting in a line fitting. In Figure 6.5 (b), eight sketches are treated incorrectly, since extra segmentation points are detected. The reason could be that these sketches for elliptical elements are very thin and that drawing speed is high. In Figure 6.5 (c), only two sketches are fitted with B-splines instead of ellipses, since some inflection points are found. From this case study, it can be seen that if drawing
is very fast, extra segmentation points could be found. A very thin ellipse should be drawn slowly, otherwise, extra segmentation points are likely to be detected. In total, 13 of 60 sketches in this case study are dissatisfying.

For Case-4, there are four sketches fitted with lines, since the curves change their direction quite sharply (Figure 6.6 (a)). In Figure 6.6 (b), three sketches are wrongly fitted with several lines, since they are drawn very fast. In Figure 6.6 (c), only one sketch is treated improperly. In total, 8 out of 30 sketches are segmented incorrectly.

From the above four case studies it can be concluded that if the drawing speed is very high, it will affect both the linearity computation and the support region length computation, resulting in finding extra segmentation points. Second, if the users change their draw directions sharply to either produce very thin ellipses or sharp-peak curves, extra segmentation is likely to happen. In total from all the above studies, 22 out of 150 sketches were dealt with incorrectly. In other words, most of the cases (about 85 %) were treated properly. On the other hand, all the tested sketches are not required to be connected to form any meaningful graphics; therefore, the testers drew the sketches with artificially free speeds and sizes. Otherwise, in real model sketching, their drawing speed would be reduced, and better results are likely to be expected.
Figure 6.3 (a) Results for Case-1 from the first tester

Figure 6.3 (b) Results for Case-1 from the second tester
Figure 6.3 (c) Results for Case-1 from the third tester

Figure 6.4 Results for Case-2 from one tester
Figure 6.5 (a) Results for Case-3 from the first tester

Figure 6.5 (b) Results for Case-3 from the second tester
Figure 6.5 (c) Results for Case-3 from the third tester

Figure 6.6 (a) Results for Case-4 from the tester
Figure 6.6 (b) Results for Case-4 from the second tester

Figure 6.6 (c) Results for Case-4 from the third tester
6.3 Case Studies On 3D Geometry

Case Studies on 3D geometry are designed to investigate the effectiveness of the prototype system, by comparing the time for modelling similar objects between the system and commercial packages. Two cases were studied. The first case study was conducted for a conceptual model of a camera. The first tester built the model shown in Figure 6.7 (a),(b) and (c), by combining on-line sketching with directly 3D projection and 2D primitive input. In the model, box features were used for building a body frame, a flash light unit, and a base part of a lens unit. Several buttons were made by cylinder features. The lens and zoom unit including a UV filter were given by cylinder features, too. The testing results from the second and third testers are shown in Figures 6.8 (a) to (d). The average building time was about 9 minutes on the prototype system. During the working process, the testers can view the model as an isometric wireframe model as in Figure 6.7 (a), a shaded model as in Figure 6.7 (b), or an arbitrary wireframe model as in Figure 6.7 (c). A similar camera model (see the insert page 1) is built on HP workstations by I-DEAS commercial packages. The package is a well-known constraint-based solid modeller. During the designing process, users were required to select menu quite often, answer questions, or enter design parameters for both 3D primitives and their constraints. The building time on I-DEAS for intermediate-experienced users was about 25 minutes.

Figure 6.7 (a) An isometric wireframe model of a camera
Figure 6.7 (b) A shaded model of a camera

Figure 6.7 (c) An arbitrary wireframe model of a camera

Figure 6.7 A camera model

Figure 6.8 (a) Results for Case-1 from the second tester
Figure 6.8 (b) Results for Case-1 from the second tester

Figure 6.8 (c) Results for Case-1 from the third tester

Figure 6.8 (d) Results for Case-1 from the third tester

Figure 6.8 Results for Case-1
The second case study was development of a conceptual model for a testing device. The model produced with my system from the second tester is shown in Figure 6.9. An isometric wireframe model in Figure 6.9 (a) shows that main-frame components were built by box features. The test pieces and other features were given by cylinder features. The corresponding shaded model and an arbitrary wireframe model are shown in Figures 6.9 (b) and (c), respectively. Results from the other testers are given in Figures 6.10 (a) to (d). The modelling time on my system was about 9 minutes. A similar model of the testing device was performed, using the I-DEAS package. The modelling time was about 20 minutes. The final resulting model is given on the insert page-2.

As noted, for both models, the camera and the test device. Of course, all I-DEAS models were correctly dimensioned. But for conceptual designs, we don’t need such precise dimensioning.

Figure 6.9 (a) An isometric wireframe model of a test device
Figure 6.9 (b) A shaded model of a test device

Figure 6.9 (c) An arbitrary wireframe model of a test device

Figure 6.9 A model for a test device
Figure 6.10 (a) Wireframe model for Case-2 from the first tester

Figure 6.10 (b) Shaded model for Case-2 from the first tester

Figure 6.10 (c) Wireframe model for Case-2 from the third tester

Figure 6.10 (d) Shaded model for Case-2 from the third tester
Figure 6.10 Results for Case-2

For the case studies, our system performed better than the I-DEAS CAD package in terms of modelling time for inexperienced users, because the users had to be interrupted quite often to select and answer menus within I-DEAS packages. By contrast, our system does not ask the users to select and respond to too many menus.

6.4 Modification of 3D Objects

Manipulating 3D features during modelling processes is highly demanded for quickly trying different solutions for a conceptual design. Currently, some manipulation functions, such as undo have been implemented, although full manipulation functions are not developed. In Chapter 5, a method of adjusting a 3D feature by the constraint solver has been introduced. Here, other methods by undoing operation in 3D and 2D are illustrated. Processes of modifying a 3D feature are give by an example of modelling a office scene. The example starts with an initial model shown in Figure 6.11 (a), which contains a computer and a desk. A box feature is then added on the front face of the computer to indicate positions and sizes of CD drivers, shown in Figure 6.11 (b). The box feature looks too big, so it has to be modified. To do so, the user is required to select undo in 3D menu, and consequently the system erases the feature and returns to 2D sketches (Figure 6.11 (c)). Afterwards, the user erases the extrusion line of the feature by undoing operation in 2D, and redraw a new extrusion line. The system will automatically recognise the box feature again. The user continues with adding two cylinder features on the front face. The result is shown in Figure 6.11 (d). In the next step, the user sketches a cylinder feature on the left side of the desk as a pen box (Figure 6.11 (e)), but the feature’s position and shape are unsatisfactory. Thus, a total change of the feature is required. The user undoes the 3D feature and all sketches for this feature, and then sketches a new feature instead, as shown in Figure 6.11 (f). Finally, the user draws a keyboard, a mouse on a mouse
pad, and a chair, forming a scene. Its shaded model and a wireframe model are given in Figure 6.11 (g) and (h), respectively. Note that a wire line represented by a B-spline curve is added to the Figure 6.11 (g) and (h) to make the mouse connection to the computer in the scene.

From this example, it can be seen that the system can support 3D modification in multiple ways. This is very useful for conceptual design.
Figure 6.11 (f)

Figure 6.11 (g)

Figure 6.11 (h)
6.5 Conclusion

This research work has investigated some new techniques for applying on-line sketching into 2D and 3D conceptual design geometry through a whole development process from data collection, concrete curve segmentation and fitting, 2D geometric constraint extraction and solver, to 3D feature recognition and modelling.

For the on-line discrete curve segmentation, our system not only exploits static information, such as the position of a sketched point, but also dynamic information, such as drawing speed and acceleration, in which a user’s drawing intention can be inferred. It is clear from the case studies in this chapter and the examples given in Chapter 2 that in most cases the system can properly segment on-line sketches drawn with different speed, skill, and scales. The segmentation approach investigated is based on fuzzy knowledge, and was shown to be suitable for on-line application. In comparison with related algorithms (discussed in Chapter 2), this approach has the following advantages:

1. For computing concrete curvatures, a key parameter: support region length can be determined automatically, in accordance with drawing speed. This means that the curvature changes smoothly and enables the system to deal with noisy data;

2. An adaptive evaluation measurement (threshold: $\beta_0 + \beta_s + \beta_l$, and speed constraint 9) for each point is produced to check if the point is a real corner or not, with consideration of the drawing speed, acceleration, and linearity. Thus, the system has good potential for capturing real segmentation points and users’ intention;

3. This approach does not involve heavy computation as in the edge approximation methods. It is suitable for real time application.
(4) In addition, the ability to interactively correct unintentional segmentation makes the system very practical.

Following segmentation, a parallel curve classification and identification method was studied (Chapter 3). This parallel method employs some fuzzy heuristic knowledge in terms of linearity and existence of inflection points, or changes of convexity, in order to quickly classify straight lines and free-form curves. It can save significant computing time, in comparison with related methods. This property is very important for on-line application. Furthermore, an adaptive threshold of linearity is applied into classification of straight lines with respect to the drawing speed. This adds a useful feature of capturing user intention. In a similar way to the segmentation process, the method can accept interactive corrections from the users for improper classification, which makes the system flexible. From the case studies included in this chapter and the examples in Chapter 3, can be concluded that the proposed method and the developed system produce proper classification and identification for a variety of 2D shapes including straight lines, circles, arcs, ellipse, elliptical arcs, and free-form curves, even in over-drawn status.

A geometric constraint inference engine and a constraint solver are researched (Chapter 4) to capture the designer' intention, to infer geometric constraints, and to obtain a possible solution. From the examples given in Chapter 4, some remarks can be made: The inference engine can infer geometric constraints automatically, simply, and adaptively. It does not require users to specify dimensional constraints as in most commercial CAD systems. These constraints are interpreted by the solver, based on the degrees of freedom analysis and conceptual design domain knowledge. The inference engine and the solver can deal with elliptical primitives and free-form curves, comparing to most of the similar systems. The solver treats 2D primitives as semi-rigid-objects with a dimensional degree of freedom. It can produce configuration steps for various primitives without
solving simultaneous non-linear equations iteratively. The solver can not only give variational geometry in 2D and 3D, based on generic geometry and its constraints, but also can avoid occurrences of unrealisable objects under the same topology (or constraints) by fully exploiting default constraints. In comparison with similar systems Easel [Jenkin 1992] and Beautifier [Pavlidis 1985], my system makes use of constraints not only as a design aid in the creation of a part, but also uses geometric constraints for geometry modification.

During the last phase, feature interpretation and manipulation techniques have been investigated, using configuration analysis of 2D geometry and constraints. A feature inference engine is developed, currently based on general extrusion objects. Each extrusion object features a closed profile and an extrusion edge. The inference engine examines 2D geometry and constraints to find a closed profile and an extrusion in accordance with 2D connection analysis, and further matches the feature with a predefined one by recognition rules. Afterwards, the engine searches a reference plane for the inferred feature by conducting a containment test with previous features, and determines the feature’s dimensional and transformational parameters, based on reverse computing of isometric projections. Finally, the 3D feature is recognised and displayed. After the feature is generated, the system allows the users to modify it by either variation in 3D or redrawing the primitives. From the case studies in this chapter and the examples in Chapter 5, can be concluded that the feature inference engine can work properly and effectively.

From a perspective view, the given examples and case studies show that the fuzzy knowledge based system can interpret users’ intention on 2D and 3D geometry satisfactorily. This prototype system can combine interactive input of 2D and 3D projections, with sketched input recognition for building 2D and 3D geometry effectively. It also supports variational geometry based on the constraint solver. The system gives the users greater freedom to quickly specify 2D and 3D geometry, than system with ‘sketched input only’ [Hwang 1994]. This mixed automatic feature
interpretation and interactive design environment can encourage designers with poor sketching skills to use it for creative design tasks. In principle, the system has a potential capability of supporting 3D surface design. It can model scenes, which are difficult for labelling scheme and optimisation-based methods.

As for drawbacks, the system requires more pre-coded construction rules, if the number of constraint types is to be increased. It also asks for more 3D feature inference rules, if adding more feature types. Moreover, it suffers from the problem of users attention shifting to another feature before entering enough information for a feature interpretation.

As up-to-date outputs of this research, five papers have been published. They are included in the reference [Wright 1998], [Qin et al 1999A], [Qin et al 1999B], [Qin et al 1999C], [Qin 2000]. Four manuscripts submitted to journals are under review.

6.6 Suggestions for Future Work

According to my up-to-date experiences and knowledge, I would like to suggest further work as follows:

(1) investigation of touch-screen based 2D input techniques, since current 2D input devices such as mouse and tablet with a stylus are inconvenient for sketching in terms of interaction style and production of noisy data. Possible solution may accept real 'pencil' sketching on the screen, or even accept users finger sketching.

(2) enhancing the 2D constraint solver by increasing the number of constraint types.

(3) improving functions of the 3D feature recognition engine, by adding more rules and more features.
(4) studying new techniques to effectively manipulate this 2D and 3D mixed design space, and to
make communication between 2D and 3D easier. For example, after the system infers a pre-
defined feature and puts it in the 3D space, the user might rotate the models just built to a proper
direction to make further sketching easier on a intended plane, the system should remember the
rotation parameters and apply them in the next recognition action.

(5) providing an interface with commercial CAD packages or Web applications. This interface may
be based on Virtual Reality Modelling Language (VRML), since VRML models can be accepted
by most CAD packages and web browsers.
References


Appendia A Software Structure

A-1 File Structure

In MFC, a software program is organised into a project, which includes Source Files, Headers Files, Resources Files, ReadMe File and External Dependencies. My project is call ‘rgm’. Its structure is shown in Figure A-1.

Fig. A-1 Project structure
A 2 Class Structure

The developed prototype system is designed and implemented on Object-Oriented methods. The class structure is given in Figure A-2.

Figure A-2. Class structure
A.3 Resource Structure

The system interface is supported by means of interface resources. The structure of the interface resources is illustrated in Figure A-3.

Figure A-3. Interface Resources
Appendix B General Reverse Computing

For computing a feature’s modelling parameters, some general reverse computation techniques are considered for obtaining 3D information from the corresponding 2D projection.

B-1 Computing a 3D Point From its Isometric Projection

![Diagram](image)

Figure B-1 A viewing system

We first analyse the transformation processes from a 3D model to its 2D projection, and then consider the corresponding reverse problems. To obtain views of a model, we can apply transformations on a model to a viewing system shown in Figure B-1. The viewing co-ordinate system $O_v-X_vY_vZ_v$ is linked with a display screen, with the horizontal axis $X_v$ pointing to the right and the vertical axis $Y_v$ pointing upward. The axis $Z_v$ defines the viewing direction. The model system is initially made coincident with the viewing system. In order to obtain an isometric projection and keep the $Y$ axis vertical, the model can be rotated about the axis $Y_v$ by an angle $\theta=45^\circ$ followed by a rotation about the axis $X_v$ with $\phi=35.26^\circ$. This transformation process [Dewey 1988] can be presented as
\[ P_v = [T_x][T_y] P = [T_z] P =\]
\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & \cos \varphi & -\sin \varphi & 0 \\
0 & \sin \varphi & \cos \varphi & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
\cos \theta & 0 & \sin \theta & 0 \\
0 & 1 & 0 & 0 \\
-\sin \theta & 0 & \cos \theta & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
x \\
y \\
z \\
1
\end{bmatrix} =
\begin{bmatrix}
x \\
y \\
z \\
1
\end{bmatrix}
\]

(B.1)

The unit vector along the \( I \) axis \([P]=[0\ 1\ 0\ 1]\) is transferred. The result vector is \([P_v]=[0\ 0.8165\ 0.5774\ 1]\). The length of this unit vector, displayed on the screen, is 0.8165. Thus, the unit vector is foreshortened to approximately 82% of its original length.

In reversing problem, a screen projection \([P_v]\) with only \( x_v \) and \( y_v \) components is known. The third row of \([T_z]\) must be zero. From the (B.1), the following equations can be received:

\[ x + z = \sqrt{2} x_v , \]
\[ x + 2y - z = (\sqrt{2} / \sin \varphi) y_v . \]

This equation can be approximately simplified as
x + z = \sqrt{2}x_y,
\quad x + 2y - z = 2.45y_y. \quad (B.2)

Since the number of unknown variables in (B.2) is greater than the number of equations, additional information is needed. It can be collected from a reference plane equation. In general, the plane equation can be addressed as

\[ Ax + By + Cz - D = 0. \quad (B.3) \]

Here, coefficients A, B, and C are components of the normal vector of the plane.

Therefore, if knowing a screen projection \((x_v, y_v)\) and its corresponding 3D reference plane, we can combine Eq. B.2 with Eq. B.3 to determine a 3D point \((x, y, z)\) on the reference plane with the corresponding 2D projection. Thus, solving the following simultaneous linear equations can give a result of a reversing computation.

\[ x + y = 0.7071x_y + 1.225y_y, \]
\[ x + z = 1.414x_y, \]
\[ Ax + By + Cz = D \quad (B.4) \]

For example, if a screen point \((2\sqrt{2}, 0.8165)\) is given with a known reference plane: \(z=3\), a 3D point can be obtained from its reversing computation as \((1, 2, 3)\).

As noted, the reference planes generally can be any constraint surfaces, e.g., general quadratic surfaces. But, solving (B.2) combined with any one general constraint surface equation is not as simple as solving (B.4).

Actually, when reference planes are parallel to the base planes: \(XOY, YOZ\), and \(ZOX\), the reversing computation can be conducted from the vector equations related to a point \((x_v, y_v)\) in the display co-ordinate system and a point \((iso_x, iso_y, iso_z)\) in the isometric projection co-ordinate system.
Processes of determining a point \((iso_x, iso_y, iso_z)\) from a given point \((x_v, y_v)\) with different reference planes are as follows:

1. Reference planes paralleled to \(XOZ\) base plane with an equation \(iso_y = dy\).

\[
ISO_Y(Y) \quad P_v(x_v, y_v) \quad W \\
-ISO_Z \quad R \quad O \quad ISO_X
\]

**Figure B.2 Computing ISO_X, and ISO_Z**

From the reference plane shown in Figure B.2, we can receive a vector equation

\[
R + Q + W = P_v
\]

From this vector equation, we obtain

\[
q = y_v - dy + x_v / 1.732, \\
w = 2 (y_v - dy) - q. \tag{B.5}
\]

Actually, \(q\) is the length of vector \(Q\), representing the \(iso_x\) component of a given point \((x, y)\).

Correspondingly, \(w\) stands for its \(iso_z\) components;

2. Reference planes paralleled to \(XOY\) base plane with an equation \(iso_z = dz\).
From Figure B.3, we can obtain a vector equation \( W + Q + R = P \), and the equation

\[
q = \frac{x}{0.866} - dz; \\
r = y - (q - dz)/2.
\]

Here, \( q \) and \( r \) represent \( iso_x \), and \( iso_y \) co-ordinates of the given point \((x, y)\), respectively;

(3) Reference planes paralleled to \( XOZ \) base plane with an equation \( iso_x = dx \)
From Figure 8.4, a vector equation \( Q+W+R=P_v \) can be received, and the equation
\[
\begin{align*}
\mathbf{w} &= dx - x_v / 0.866, \\
r &= y_v - (dx + w) / 2.
\end{align*}
\] (B.7)

Here, \( w \) is for \( \text{iso}_z \), and \( r \) is for \( \text{iso}_y \) co-ordinates.

**B-2 Computing a Rotational Angle about OY Axis**

Although in most cases 3D primitives are located in their standard positions, in which they are parallel to the base planes: \( XOY, YOZ, \) and \( ZOX \). But, in some cases, they are positioned parallel to only one of the three base planes, which means that they are already rotated about one of the three axes: \( OX, OY, \) and \( OZ \). If the primitives after a rotation are isometrically projected onto a screen, we need to find the rotation parameter for them. For example, a box may be projected onto the screen just after a rotation about \( OY \) axis. Its extrusion edge is still parallel to \( OY \) axis, as shown in Figure 5.5 (b). Here, we first consider a rotation about \( OY \) axis.

To find the rotation angle about \( OY \) axis, we investigate the transformation processes of a unit vector \( \mathbf{P} = [1 \ 0 \ 0 \ 1] \) along \( OX \) axis.

1) Rotating about \( OY \) axis by \( \xi \) angle

The transformation can be addressed as
\[
\mathbf{P} = [T\_y] \mathbf{P} = \begin{bmatrix}
\cos \xi & 0 & \sin \xi & 0 \\
0 & 1 & 0 & 0 \\
-\sin \xi & 0 & \cos \xi & 0 \\
0 & 0 & 0 & 1
\end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix}
\cos \xi \\
0 \\
-\sin \xi \\
1
\end{bmatrix}
\]

2) Projecting the vector \( \mathbf{P'} \) isometrically
From (B.2), we can receive
\[
\cos \xi + \sin \xi = \frac{1.732 y_{vr}}{x_{vr}}
\]  
(B.8)

For reverse computation, the ratio of \( y_{vr} \) and \( x_{vr} \) of the projected unit vector along \( OX \) axis is known. Thus, the angle \( \xi \) can be obtained from the following equations:

\[
tg \xi = \frac{m - 1}{m + 1},
\]
\[
m = 1.732 \frac{y_{vr}}{x_{vr}}.
\]  
(B.9)

B-3 Computing a Rotational Angle about OX Axis

To find the rotation angle about \( OX \) axis, we investigate the transformation processes of a unit vector \( \hat{Q} [0 \ 0 \ 1] \) along \( OZ \) axis in a similar way.

(1) Rotating \( \psi \) angle about \( OX \) axis

The transformation can be expressed as
\[
\hat{Q}' = [T_{\psi}] \hat{Q} = \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & \cos \psi & -\sin \psi & 0 \\
0 & \sin \psi & \cos \psi & 0 \\
0 & 0 & 0 & 1
\end{bmatrix} \begin{bmatrix}
0 \\
-\sin \psi \\
cos \psi \\
1
\end{bmatrix}
\]

(2) Projecting the vector \( \hat{Q}' \) isometrically

From (B.2), we can obtain
\[
-(2 \sin \psi + \cos \psi) = \frac{1.732 y_{vr}}{x_{vr}}
\]  
(B.10)
For the reverse computation, the ratio of $y_{vz}$ and $x_{vz}$ of the projected unit vector along $OZ$ axis is known. Thus, the angle $\psi$ can be obtained from the following equations:

\[
tan \psi = \frac{n + 1}{2},
\]

\[
n = 1.732 \frac{y_{vz}}{x_{vz}}.
\]

**B-4 Computing a Rotational Angle about OZ Axis**

In order to find the rotation angle about $OZ$ axis, we investigate the transformation processes of a unit vector $R [0 \ 1 \ 0 \ 1]$ along $OY$ axis.

1. Rotating by $\zeta$ angle about $OZ$ axis

The transformation can be regarded as

\[
R' = [T_\zeta] R = \begin{bmatrix}
\cos \zeta & -\sin \zeta & 0 & 0 \\
\sin \zeta & \cos \zeta & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix} \begin{bmatrix}
0 \\
1 \\
0 \\
1
\end{bmatrix} = \begin{bmatrix}
-\sin \zeta \\
\cos \zeta \\
0 \\
1
\end{bmatrix}
\]

2. Projecting the vector $R'$ isometrically

From (B.2), we can obtain

\[
-\frac{\sin \zeta + 2 \cos \zeta}{-\sin \zeta} = 1.732 \frac{y_{wy}}{x_{wy}}
\]

For reverse computation, the ratio of $y_{wy}$ and $x_{wy}$ of the projected unit vector along $OY$ axis is known. Thus, the angle $\zeta$ can be obtained from the following equations:

\[
ctg \zeta = \frac{1 - k}{2},
\]

\[
k = 1.732 \frac{y_{wy}}{x_{wy}}.
\]