Recognition and Location of Ugaritic Character Stylus Strokes from Clay Tablet Images

Thomas Anthoni

MSc Information Technology: Knowledge Based Systems

Department of Artificial Intelligence

University of Edinburgh

1994

Abstract

Ugaritic is claimed to be the world's first alphabetical character set, and was written on clay tablets in a cuneiform script. It comprises 30 characters each of which consists of several wedge-shaped primitives. So far, several thousand tablets have been excavated and studied and finally the wish arose to have some tool to read the tablets automatically.

The task of recognizing the characters in the images taken of the tablets can be split into two parts: the low-level stage to extract features from the images and the second step which uses the features to classify the characters.

This project is concerned with the first part, the image preprocessing. The aim is to locate the character primitives in the image. This is achieved by correlating models of the primitives with the image data.

Acknowledgements

I would first like to thank my supervisor, Dr Robert B. Fisher (DAI), for all his comments and guidance during the period of the project. I would also like to thank Dr Jeff B. Lloyd and Dr Nick Wyatt (both Faculty of Divinity) for their help and support concerning the aspects of the project not related to computers. Thanks are also due to Simon Perkins (DAI) and Andrew Fitzgibbon (DAI) for their advice.

Table of Contents

1. Introduction **1**

1.1 Aims of the Project 1

1.2 Motivation 1

1.3 Template Matching Using Correlation 2

1.4 Research Activities 2

1.5 Guide to this Thesis 3

2. Background **4**

2.1 History 4

2.2 The Clay Tablets 5

2.3 Languages in Ugarit and the Ugaritic Script 5

2.4 How the Data Was Obtained 7

2.4.1 TIFF and PGM Image Files 9

2.4.2 The Appearance of the Wedges 10

2.5 Examples of Other Optical Character Recognition Applications . . 11

2.5.1 Recognizing Hand-Written Zip Code Digits 11

2.5.2 Discriminating Similar Kanji Characters 12

2.5.3 Signature Verification 12

2.6 Initial Approaches 13

3. Overview of the Project **16**

3.1 Project Phases 16

3.2 Correlation 18

4. Preprocessing the Image Data 20
4.1 Models of Wedges 20
4.2 Estimation of Scale and Rotation 23
4.2.1 Rotation 25
4.2.2 Scaling 25
4.3 Background Elimination 27

5. Program Structure 30
5.1 The C and C++ Language 30
5.2 Data Structures 31
5.3 Overall Control Structure 34
5.3.1 Program Options 34
5.3.2 Control Flow 35

6. The Template Matching Techniques Employed 37
6.1 Correlation 37
6.2 Standard Deviation and Subtemplates 39
6.2.1 The Standard Deviation 40
6.2.2 Subtemplates 41
6.3 Final Filtering 44
6.3.1 Allowed Overlaps 44
6.3.2 Ambiguous Labels 46

7. Experiments and Results 49

8. Conclusions and Further Work 63
8.1 Summary of Main Results 63
8.2 Limitations and Extensions 63
8.3 Alternative Approaches 66

A. Program code 69

List of Figures

2–1 Clay tablet 6

2–2 Ugaritic script and its transcription 8

2–3 Bright and dark regions in the wedges 10

2–4 Enlarged view of original letter and letter after thresholding (black pixels correspond to binary 1 here) 14

2–5 Histogram of the image region 14

2–6 Binarized image region: left after Closing, right after Opening . . . 15

3–1 Stages of the project 17

3–2 How the correlation is calculated 19

4–1 The 9 templates used 22

4–2 Left: original wedge model, right: rotated by +33° 25

4–3 How the pixels of the input image map onto a pixel in the shrunk image 26

4–4 Wedge model on the left: original size, centre: shrunk to 70%, right: shrunk to 50% 26

4–5 Clay tablet image with framed background regions 29

5–1 How the wedge locations are marked in the output image 33

6–1 Two false matches (+) on the top left 40

6–2 Left: enlarged view of template, right: corresponding subtemplate . 41

6–3 Combinations of wedges of type 1 and 2 48

7–1 Image 169 processed with filtering 50

7–2 Image 169 processed without filtering 51

7–3 Image 718 processed with filtering 52

7–4 Image 718 processed without filtering 53

7–5 Image 658 processed with filtering 54

7–6 Image 658 processed without filtering 55

7–7 Image 663 processed with filtering 56

7–8 Image 663 processed without filtering 57

7–9 Program output The first column contains the template number (i.e. *temp_nr*), a number between 1 and 27. The second and third columns tell the x and y coordinates of the wedge's marked reference point in the image. The value in the fourth column is the correlation coefficient according to which the table is sorted. The last but one column is for the standard deviation and the last one for the subtemplate correlation coefficients. 62

List of Tables

6–1 Thresholds 43

7–1 Statistics for tablet 169 58

7–2 Statistics for tablet 718 59

7–3 Statistics for tablet 658 59

7–4 Statistics for tablet 663 60

8–1 Frequency of Ugaritic characters in a set of 26 texts 65

Chapter 1

Introduction

1.1 Aims of the Project

The principal goal of this project is to find the locations and types of character primitives (wedges) impressed into clay tablets. The program developed in the course of this project reads an image of a clay tablet and outputs a list of detected wedges together with their types and positions in the image.

1.2 Motivation

Character recognition systems are important tools in automating the processing of text and other visual information. In recent years more and more such software systems have been developed that incorporate neural nets and thus produce better results than conventional methods.

An optical character recognition system generally contains a preprocessor and a classifier. The preprocessor is to select a set of features and to remove noise and other meaningless visual variations from the input image.

Often, the preprocessor consists of a simple procedure to binarize images by thresholding them but may also contain routines to extract features such as moments, area, perimeter etc. from the binarized image. The problems encountered at the preprocessing stage are usually insignificant if, as it is often the case, the characters are written on a background of different colour.

1.3 Template Matching Using Correlation

The situation in the images for this project is different from the one above. The background of the tablets has basically the same colour as the characters and contains distracting texture such as watermarks, cracks and traces of erosion. Only the fact that the wedges are impressed into the clay can provide usable features.

When the tablets were photographed the light came at an angle thus making the characters cast shadows. So, the shape of the shadows is the principal feature in this recognition task. As the shadows do not stand out enough from the tablet background to make an image binarization feasible, another technique to detect the character primitives was used.

Models (or templates) of the primitives were correlated with the image data. High correlation values at a particular position are considered to indicate that a model has matched there with a wedge in the image. After that, the set of such matches has to be cleared of false and multiple matches so that finally a set of matches remains with each match indicating exactly one found wedge and its type.

1.4 Research Activities

During the time span of this project, the scope of the research covered:

- Background reading on vision techniques and neural networks
- Meetings with experts at the Faculty of Divinity to obtain information about the nature of the characters, exploitable constraints, and to obtain slides to be used as test images
- Choosing one test image to start with
- Trying a range of vision algorithms and operators to find features in the image that could discriminate the character primitives

- Finally, deciding on correlation of models with image data and writing a program for this purpose
- Extending the set of models, testing and adapting them until the vast majority of wedges in the image were detected
- Implementing a filter to reject false matches in tablet background areas
- Implementing a filter that removes multiple matches for a wedge and keeps only one match
- Testing the program on four images and drawing conclusions from the results

1.5 Guide to this Thesis

The remainder of the thesis is organized as follows:

Chapter 2 gives background information about the Ugaritic language, the characters to be dealt with, and the clay tablets.

An overview of the program and its stages is given in Chapter 3. It also presents the formulae needed to compute the correlation.

Chapter 4 describes how to obtain necessary parameters for the template matching by preprocessing the image, the models used, and also how the image background is detected. The way the program works is discussed in Chapter 5 containing the data structures, the control flow and the options it can be given when it is started.

Chapter 6 describes the features and the constraints employed to remove false matches. The experiments carried out are presented in Chapter 7, which also discusses the results obtained. Finally, suggestions for further extensions and conclusions can be found in Chapter 8.

Chapter 2

Background

This chapter gives a brief overview of some features of the Ugaritic language as well as of the characters to be recognized. For more information on this language and the culture, the reader is referred to [Curtis 85] and [Craigie 83].

Furthermore, the kind of data used for the recognition task is explained and some examples of related OCR solutions are given.

2.1 History

In 1928, some farmers found some stone slabs on a field near the Syrian coast. These slabs turned out to be part of a tomb dating from the thirteenth century BC and soon extensive excavations began. After digging in the area, the excavators zeroed in on a nearby hill known as Ras Shamra and it was there that they found the remains of an ancient city which had been destroyed by fire (apparently about 1200 BC by barbarians). Among the different objects encountered in the ruins were baked clay tablets containing cuneiform writing. Some of the tablets were even still in the piles they were stacked in when the place was destroyed. To date, tens of thousands of such tablets have been dug out and the excavations still continue. In 1994, over 300 new fragments were found.

2.2 The Clay Tablets

Besides papyrus, clay was a widely used writing material in the area along the Syrian coast. Wedge-shaped marks were impressed into the soft clay with a stylus and made permanent by baking the tablets hard afterwards. Figure 2–1 shows one such clay tablet. The tablet size varied enormously as did the size of the imprinted characters. The smallest tablets found are only 3×4 cm whereas the biggest ones measure up to 0.7×1.2 m. This applies similarly to the characters, the sizes of which range between approximately 1 mm and more than 1 cm. The direction of writing is in general from the left to the right with the text running from the top to the bottom. Many tablets are not flat but rather oval and the text is often written on them in several columns separated by two vertical lines. Often, the text continues on the tablet back with the rightmost column going on around the bottom edge and then **upwards** to the top of the rear side. All following columns are added at the left.

Many texts are quite short and occupy only one tablet, especially the ones with administrative or commercial content. Apart from texts concerning everyday matters, there were archives of mythological and religious texts, which turned out to be very useful for comparative studies with the Old Testament and for understanding the nature of Canaanite religion. Clay tablets with less important or only temporarily useful information would be re-used after the previous text had been scraped off. On some tablets, however, the former characters can still be seen in places.

2.3 Languages in Ugarit and the Ugaritic Script

Ugaritic itself is a Semitic language such as Biblical Hebrew. Although it was the local language it was by far not the only language in use at Ras Shamra in the mid-fourteenth century BC. Less than 3000 of the tablets found so far are written in Ugaritic and most of the others are in Akkadian (the lingua franca for administration at that time), Sumerian, Hurrian, Hittite and even in Egyptian.

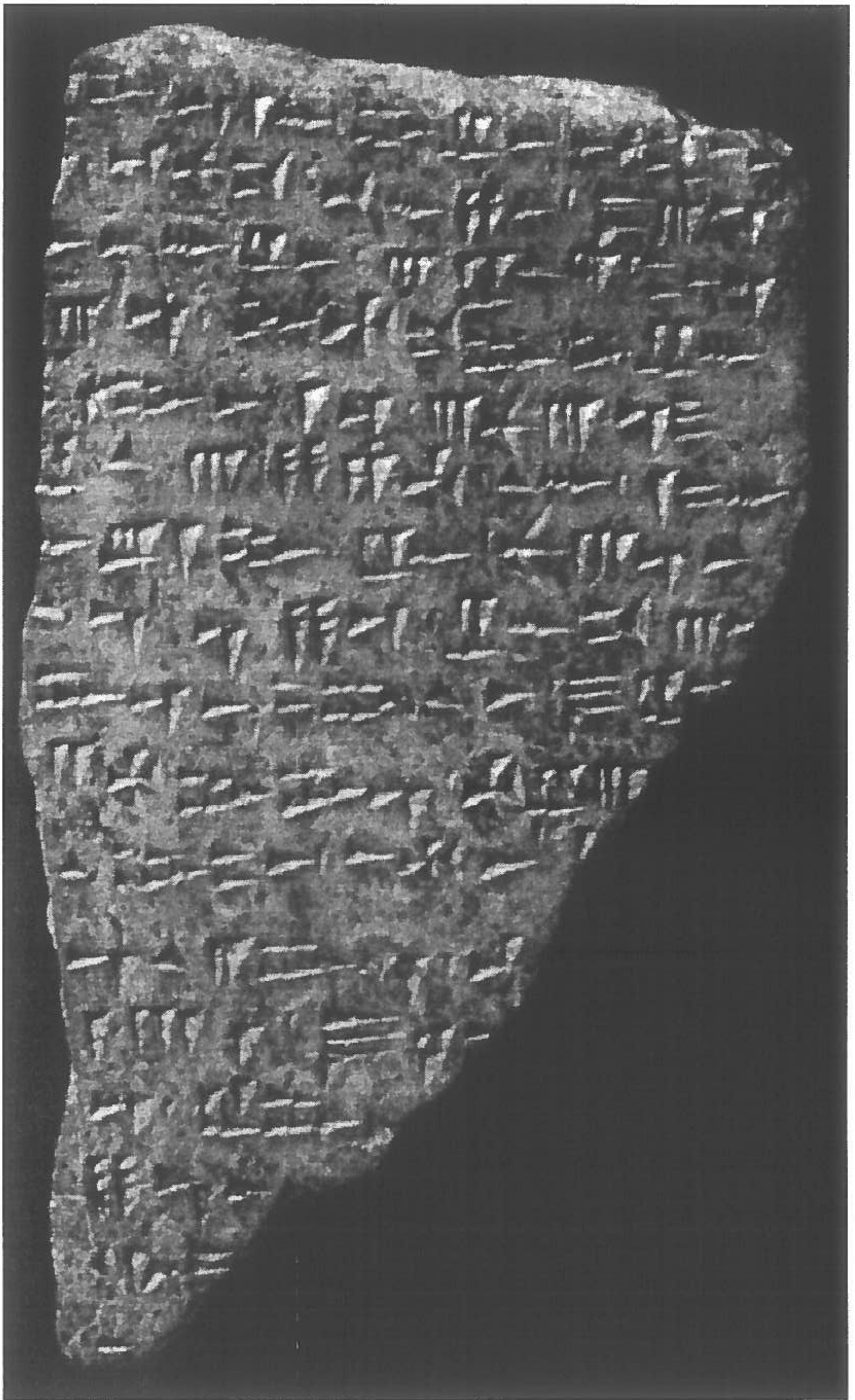


Figure 2–1: Clay tablet

The alphabetic cuneiform script used for Ugaritic texts as well as for some Akkadian and Hurrian ones seems to have developed from older types of syllabic cuneiform writing or, as some researchers believe, from Egyptian hieroglyphs and it claims to be the oldest alphabet, although this has yet to be proved. The standard alphabet consists of 30 letters, 27 of which are consonants. Depending on the scribe's writing style the strokes look more or less like wedges. Each letter is made up by up to 7 of these wedges which overlap one another in some letters. A small wedge is used to divide words, which are in general short as only the consonants and no vowels are written and because of the nature of Semitic languages in general. Figure 2–2 shows the Ugaritic alphabet and the corresponding letter transcriptions commonly used today.

The first and quite quick successes in deciphering the Ugaritic tablet texts were made in the early 1930s by applying cryptoanalytic techniques based amongst others on the character frequency and on the fact that only a few different characters appeared (therefore the assumption of an alphabetical script). Later, in the 1950s, after even some quadrilingual texts had been discovered, the previous decipherments could be verified with only very few corrections being necessary. It also became clear in the deciphering process that the name of the ancient city must have been Ugarit. This name is found both on some of the tablets and in other texts of that era which were discovered in the Near East and refer to a city called Ugarit somewhere in Syria or Palestine. Since then, the name Ugarit has gradually been replacing the Arab name Ras Shamra in publications.

2.4 How the Data Was Obtained

The clay tablets to be dealt with here were photographed in the museums in Damascus, Aleppo and Paris where they are stored and later the slides were scanned in order to obtain grey-level images. The scanner used was a Microtek Slide Scanner; the software was Adobe Photoshop 2.5. The highest possible resolution available from the scanner used is ca 1800 dots per inch so that the images occupy up to 2200 × 1600 pixels. (The surrounding background was cut off where possible.) The pixel depth is 8 bit which means that 256 grey levels can be represented in the image with the value 0 standing for black and 255 for white.

a
b
g
ḫ
d
h
w
z
ḥ
ṭ
y
k
š
l
m

ḏ
n
ẓ
s
c
p
ṣ
q
r
ṯ
ǵ
t
i
u
ś

Figure 2–2: Ugaritic script and its transcription

2.4.1 TIFF and PGM Image Files

The image files output by the scanner are in the TIFF format; a standard supported by many image processing software systems on a variety of hardware platforms. In order to have an easier access to the image data, the files are subsequently converted into another standard image format (which is, however, not available on the scanner), called Portable Graymap or PGM. This transformation can be done using the following pipeline of UNIX commands which are part of the operating system environment (SunOS 4.1):

```
tifftopnm < input_file | ppmtopgm | pnmnoraw > output_file
```

The generated output will be an ASCII PGM file that one can read and manipulate with any editor. Typically, it looks like this:

```
P2
# any comment
5 2
255
113  100  100   90  240
255  234   78  129   12
```

The first line contains the file format identification (or Magic number) which is P2 for PGM files. Any comments introduced by # are ignored. The following line holds the number of columns, in this example 5, and the number of rows, here 2, which the image has. The last line of the header stores the highest pixel value to appear in the image, 255 here. All that follows are the pixel grey-level values, line by line, listed in columns from top to bottom, and separated by one or more spaces. A PGM file must not be wider than 70 characters. As there need not be a special mark, such as a newline character, to indicate the end of an image line, the correct row and column numbers in the header are vital information to keep track when reading or writing an PGM file.

Both TIFF and PGM files as well as images in many other standards can be displayed by the *xv* program which apart from some image operations also allows

images to be saved as Postscript files. This is how the images in this dissertation were produced.

The reason why the PGM format was used for this project is that the data can be easily manipulated and it is possible to create PGM files as output from within a program without having to worry about any complicated file header or trailer information or image compression.

2.4.2 The Appearance of the Wedges

Some of the scanned photographs show a whole clay tablet whereas others only contain a smaller but more detailed part of a tablet. The pictures were usually taken in front of a uniform dark or bright background.

When photographing a tablet the light always came from a lamp or through the window located beyond the top left tablet corner which results in the upper and left regions of the wedge-shaped marks casting shadows and the bottom and right parts looking quite bright. This is due to the fact that the left part of a wedge was generally imprinted deeper by the stylus than the right part.

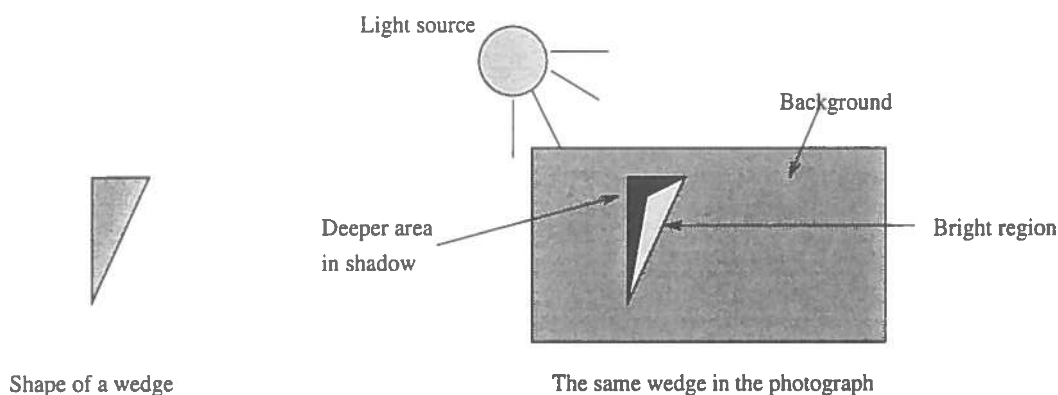


Figure 2–3: Bright and dark regions in the wedges

2.5 Examples of Other Optical Character Recognition Applications

In recent years much work has been done in the field of Optical Character Recognition and many working software or hardware solutions have resulted from this. A great deal of these systems involve neural networks which seem to produce better results in many cases than "classical" solutions. A few widely noted OCR applications and their main features shall be mentioned in this section.

All these systems have in common that the characters to be recognized are written on smooth material rather than being carved into it so that no shadow effects occur due to the different ways of illuminating the material. Often, the input into the neural net consists of binary data, i.e. the intensity values of the characters are usually significantly less than the background intensity values and so the two can be separated by applying a simple thresholding procedure, which is definitely not the case as for the clay tablet images. No other image processing operations are necessary in order to obtain the kind of data the OCR system requires. Most systems extract features from the binary image which are then fed into the neural net (as opposed to raw image pixel intensity values), although sometimes as is the case with Fukushima's Neocognitron [Fukushima & Miyake 82], this feature extraction step is part of the neural network.

2.5.1 Recognizing Hand-Written Zip Code Digits

One well known task is to classify hand-written digits used in US Zip codes on envelopes. A project was undertaken by Denker et al. and more information can be found in [Denker 89], [Denker 90], and [Denker *et al.* 89]. In the preprocessing stage the image has to be segmented into single digits. Errors can occur when digits overlap or touch one another. The window containing a digit has then to be scaled to a unit size of 20×32 pixels. Following this, the digits are thinned to one-pixel line width.

Using 49 different feature extractor templates each being 7×7 pixels large, the several types of strokes are found. Combining groups of the 49 outputs finally yields 18 feature maps each occupying 3×5 pixels.

The last step comprises the actual classification. Only three of the tested classifiers have been found applicable for this task: the k-Nearest-Neighbour classifier, a Parzen-Windows based classifier, and an adaptive network of 2 layers and 40 hidden units. In the tests with 10,000 digits taken from real envelopes all three classifiers performed similarly well. (The samples provided for the test had already been binarized and divided into single digits.) 14% of the samples were rejected by the system and 1% classified wrongly using the k-Nearest-Neighbour classifier. With no rejections allowed 6% misclassified samples were obtained.

2.5.2 Discriminating Similar Kanji Characters

Mori and Yokosawa used a 3-layered feedforward network trained with the backpropagation algorithm to discriminate similar hand-written Japanese Kanji characters. Their results are published in [Mori & Yokosawa 89]. The training and testing patterns are taken from a database with several hundred samples for each of the about 3,000 commonly used Kanji characters. Each sample is 64×63 dots large.

One of the features taken as input for the net is the length of the line segments that a character consists of, with the length measured in 4 different orientations. Using 100 samples of each character for training and another 100 for testing the net, 92% of the testing patterns were classified correctly.

In a later extension, many such backpropagation networks were each employed to deal with only part of the set of 3,000 characters and then all these subnets were combined in a modular way in a large-scale net. The training times diminish drastically as one subnet has to be trained only on a subset of all characters.

2.5.3 Signature Verification

Another application is the recognition and verification of signatures. For this task Mighell, Wilkinson and Goodman scanned hand-written signatures from cheques

and thresholded them to obtain binary images. More information on this project is in [Mighell 89]. These were then centred and normalized to fit into a 128 $\times$ 64 matrix. A feed-forward network trained with backpropagation uses the matrix as input. The net had only 1 hidden unit and 1 output unit in the simplest case investigated and it needed $128 \times 64 + 1$ weights (one for each pixel of the matrix). After training the net with 10 true and 10 forged signatures, it was tested on 70 true and 56 forged signatures. The result was that 1% of the signatures presented was rejected with another 4% forged signatures being accepted.

Other network setups were tested as well. For example, the matrix was divided into 32 regions of 16 $\times$ 16 pixels each so that the network needed 32 hidden units. The tests for this case resulted in 2% rejected signatures and 2% accepted forgeries.

2.6 Initial Approaches

As one can observe in both Figure 2–1 and Figure 2–3, the area covered by the wedge-shaped characters in the images is either darker or brighter than the background (i.e. the rest of the image). This led to the assumption that it is possible to binarize the image in two thresholding steps. A lower threshold has to be found to single out the dark regions (e.g. everything below the grey level 100) and an upper threshold to preserve the bright areas (e.g everything above 210). Then, both dark and bright regions could be merged and assigned the binary value 1 whereas the rest (e.g. everything between 100 and 210) would be set to 0. The 1-regions are supposed to represent the wedges which would then be classified using some appropriate pattern matching techniques.

However, it is not possible to find suitable thresholds, even when searching manually, as both dark and bright areas usually include subregions with higher or lower intensity values, respectively.

One letter cut out of an image can be seen on the left of Figure 2–4 as an original grey-level image and on the right after binarizing it by applying a lower threshold of 140 and an upper one of 222. These thresholds seem to be the best ones possible and were found after several trials using the corresponding histogram shown in Figure 2–5.

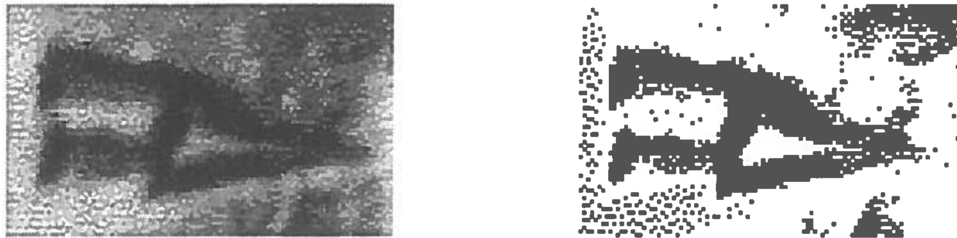


Figure 2–4: Enlarged view of original letter and letter after thresholding (black pixels correspond to binary 1 here)

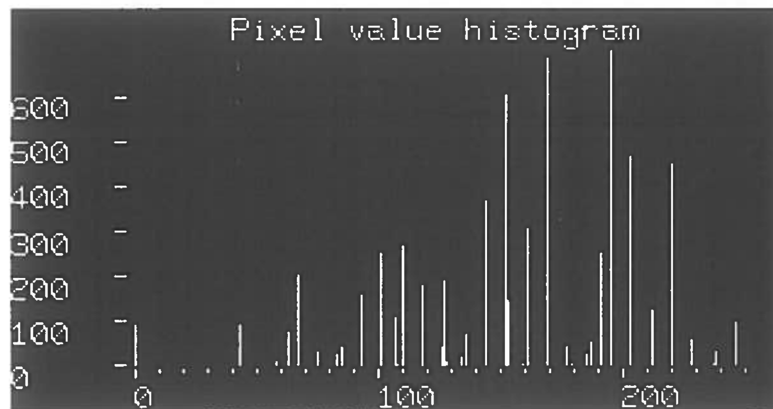


Figure 2–5: Histogram of the image region

In order to achieve decent binary images we tried to fill up the holes in the wedges and smooth their outlines by exploiting a combination of the dilation and erosion operators. The erosion operator changes every bright pixel which has a black one among its neighbours into a black pixel and the dilation operator, in its turn, does the reverse. It changes black pixels to bright ones.

The left image in Figure 2–6 was obtained by dilating and then eroding the binarized image of Figure 2–4 (an operation known as Closing) whereas for the right one the Opening operator was applied. Opening comprises first erosion and then dilation. Both results are, however, not satisfactory for any further processing so another approach had to be found and explored. The remainder of the thesis describes the new approach taken.

Figure 2–6: Binarized image region: left after Closing, right after Opening

Chapter 3

Overview of the Project

This chapter provides a general overview of the steps and their order taken to recognize the character strokes. The mathematical formulae used in the following chapters are explained as well.

3.1 Project Phases

The process used for locating the wedge strokes in this project is outlined in Figure 3–1. Two things should be observed here:

1. it is necessary to have (good) models of the wedges to be detected

2. the Template Matching technique applied is based on the **correlation** of the models with the data (as opposed to e.g. convolution or other operations)

An image which has been obtained in the way described in Chapter 2 is input into the program. The program's preprocessing stage tries to find the optimal scaling and rotation parameters for the 9 templates and subtemplates of the model base, which are small image files that have been linked into the program.

Having adapted the models the correlation of models and image data is computed and the correlation maxima are extracted (together with their positions). The final part of the program employs additional features (the standard deviation,

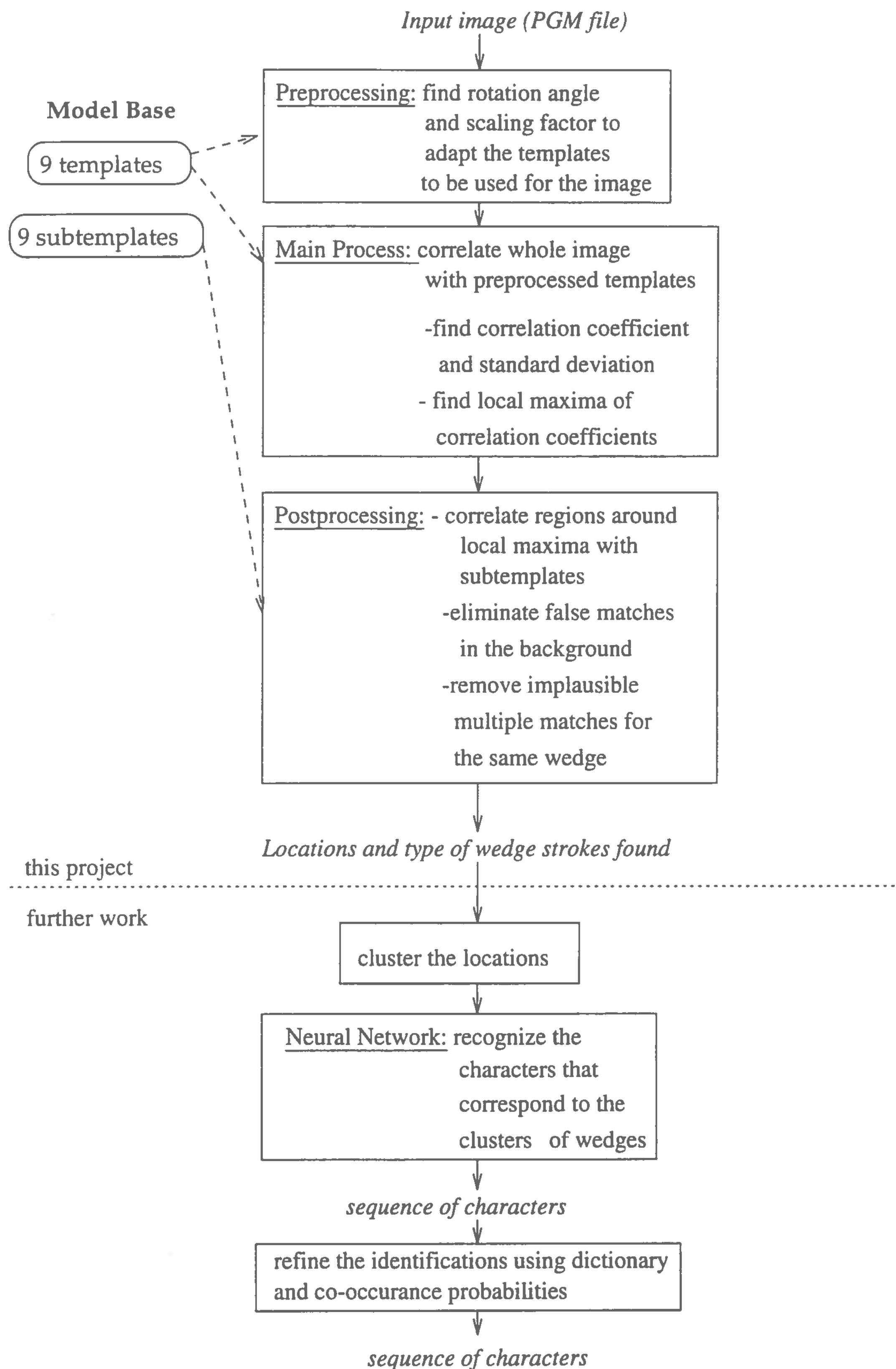


Figure 3–1: Stages of the project

the correlation with subtemplates) as well as constraints resulting from the character set itself to reject those matches of models with data that are false. (I.e. at those positions are no wedges.)

In a second stage, which is not part of this project, the set of wedge types and locations will be input into an algorithm that groups wedges located in some neighbourhood to clusters which should correspond to the wedges a character consists of. Such a cluster of wedges would then be classified by a neural net which in its turn outputs the character that corresponds best to the wedge cluster. Afterwards, the identification result may be refined by the use of statistical data such as the character frequency in Ugaritic texts and information about what pairings of characters are possible and with which probability.

3.2 Correlation

The main feature exploited to match a model wedge with a wedge in the image data is the correlation coefficient, which tells how well the model correlates with the data. The formula to obtain the correlation coefficient is the following:

$$Correlation = \rho = \frac{\sigma_{D,M}}{\sigma_D \cdot \sigma_M} \qquad -1 \leq \rho \leq +1 \tag{3.1}$$

with $\sigma_{D,M}$ being the covariance of data (index D) and model pixels (index M). σ_D and σ_M are the standard deviations of data and model pixels, respectively.

$$\sigma_{D,M} = \frac{1}{M} \cdot \frac{1}{D} \cdot \sum_j \sum_i (d_j - \bar{d}) \cdot (m_i - \bar{m}) \tag{3.2}$$

d_j and m_i are the data and model pixel intensity values, respectively. The mean of the model and data pixel values $\bar{m}$ and $\bar{d}$ are:

$$\bar{m} = \frac{\sum_i m_i}{M} \qquad \bar{d} = \frac{\sum_j d_j}{D} \tag{3.3}$$

The standard deviations are calculated like this:

$$\sigma_M = \frac{\sqrt{\sum_i (m_i - \bar{m})^2}}{M} \qquad \sigma_D = \frac{\sqrt{\sum_j (d_j - \bar{d})^2}}{D} \tag{3.4}$$

In the above formulae M is the numbers of model pixels considered and D the numbers of data pixels.

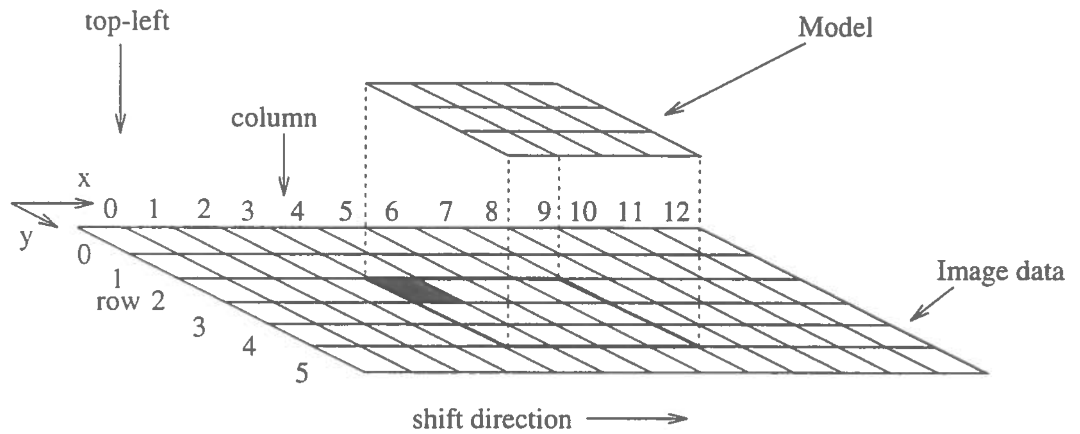


Figure 3–2: How the correlation is calculated

Both model and data pixel intensity values are stored in two-dimensional arrays with the model arrays smaller than the image data array. As one can see in Figure 3–2, the model array covers some window of the image data array and it is the pixels of this window which are used to calculate one correlation value. This value is then assigned to the position in the output array that maps onto the top-left corner of the model array (position (x=4, y=2) in Figure 3–2).

In order to calculate the correlation values for all pixels in the data array, the model array has to be shifted over the whole image data array by one position every time. The computed correlation coefficient has to be assigned again to the pixel position right under the model's top-left corner in some output array. In the program which carries out these calculations the model is shifted row by row over the image starting in its upper left corner (x=0, y=0) so that the first correlation value obtained is at position (0, 0), the second at (1, 0) etc.

When the correlation is computed it is not generally the case that all pixels of the model window are used. In this program only the pixels of a model which are inside a wedge are selected and used with its corresponding data image pixels for the calculation. Thus, the model window background does not affect the correlation coefficient, which is desired.

Chapter 4

Preprocessing the Image Data

This chapter deals with the steps to be taken in order to preprocess the image data. As it has already been mentioned in the introduction chapter, the goal of this first step towards recognizing the Ugaritic characters is to look for the primitives (i.e. the wedges) the characters consist of and find their locations in the image.

The technique employed to detect wedge locations is Template Matching based on correlating model data with image data. That is why it is necessary to define models that represent what is being looked for. Each kind of wedge will have a corresponding model (a template) to be correlated with the image data. The correlation function, which is dealt with in Chapter 3, yields values between -1 and +1. A very good match of model and image data will be indicated by a correlation coefficient close to +1 (or exactly +1 for a perfect match).

4.1 Models of Wedges

The wedges the characters are made up of look similar but they can have varying sizes and appear in different orientations. Taking into account only the wedge orientation, there are four basic types to be found in the alphabet:

1. a "horizontal" wedge
2. a "vertical" wedge
3. a wedge "pointing down to the right"
4. a wedge "pointing left"

Wedge type 3 only exists in the form of the single-wedge character *c* whereas the other three wedge types are either part of a more complex character or represent a character of their own such as the letters *g* and *t* . As one can see in Figure 2–2, the basic wedges comprise the characters as follows:

Wedge type 1 in *a, b, d, h, w, ḥ, ṭ, k, m, n, ẓ, p, q, r, ǵ, t, i, u*
Wedge type 2 in *b, g, ḫ, d, z, ḥ, ṭ, y, l, m, s, ṣ, ṯ, u, ś*
Wedge type 3 in *c*
Wedge type 4 in *ḥ, ṭ, ẓ, q, ṯ* .

The letters *š*, *ǵ*, *ḏ*, and *ś* are very rare in the test images available for this project. The *ś* character does not appear at all and only few, corrupted samples can be found for the other three. Thus, it should be possible to recognize the great majority of the characters occurring in the texts by using the four wedge shapes mentioned above.

However, confining the model set to merely four templates turned out to be not sufficient. The wedges in several characters overlap one another which results in a poor correlation between such a piece of image data and a one-wedge model if more than about half the wedge is occluded by others. This does not generally happen with pairs of wedges (e.g. the letters *a* or *z*) because the degree of overlap is not high enough to affect the correlation test. But, stacks of three wedges tend to contain two very occluded wedges and one fully visible one. This is the case especially for triples of horizontal wedges (character *n* and part of *d*) as well as for stacks of three vertical wedges (character *ḫ* and part of *y*). As both these triples are quite frequent, it was necessary to add more templates to the basic set in order to meet the difficulties in recognizing partly occluded wedges.

Furthermore, it did not prove sufficient to have only one template per basic wedge type. The shapes of horizontal and vertical wedges vary too much to represent them by only one model each. So, a couple of variations of vertical and horizontal wedges were also added to the template set to obtain finally the 9 models shown in figure 4–1:

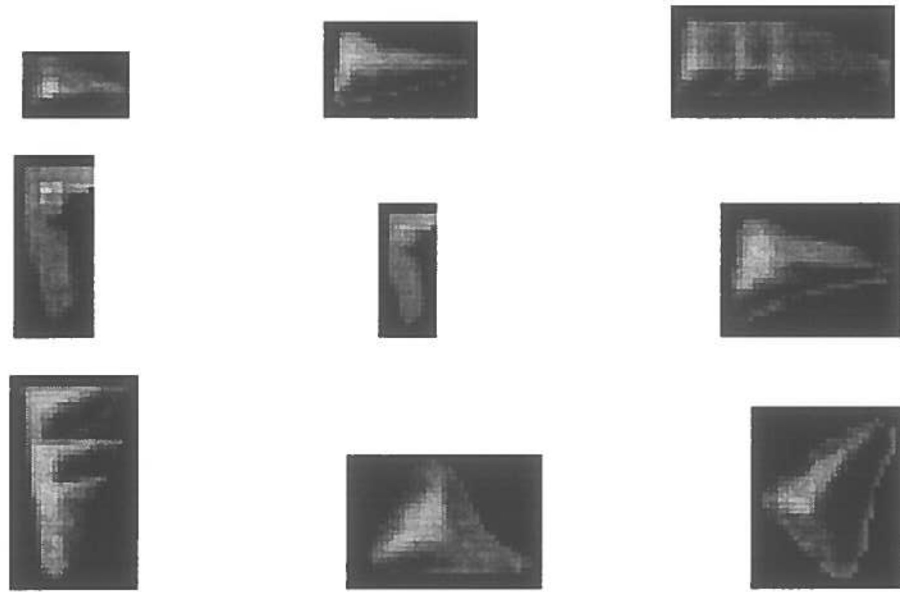


Figure 4–1: The 9 templates used

Each such model is cut out from a suitable image region and saved as a PGM file. Noise was then cleaned from the samples (i.e. bright and dark blobs were neutralized) and the background within the model windows was set to a unique pixel intensity value; +1 in this case. This is necessary to be able to discriminate wedge area and background in the course of the recognition process. The value +1 was chosen because no pixel within the wedge areas has this intensity value. All these adaptations were made by hand with an editor.

It was found in initial tests that neither completely manually designed templates nor unmodified image samples produce satisfactory results when used for the template matching later on. The slightly adapted models, however, are no longer tuned to only one particular instance of a wedge in an image but are general enough to correlate well with a number of instances of the corresponding wedge type.

After that, all models were converted into C source code for the actual program to be able to use them. This is done by means of a small program *pgm2c* , which takes the PGM files given as command line arguments (can be any number) and generates a C source file which is sent to the standard output. The command is for example used in the following way:

```
pgm2c temp1.pgm temp2.pgm temp3.pgm > models.cc
```

The C file (here *models.cc*) contains the definition of a vector *basic_temps*[] with as many elements as the *pgm2c* command has arguments. Each vector element is a structure *temp* containing a 100×100 double array t as well as two integer variables *cols* and *rows* to store the number of columns and rows a template has. The structure elements are initialized by the data read from the PGM files. The 2-D array is initialized with the pixel values starting at position (0,0) in the "top left array corner" and finishing at position (rows,cols). So, only the top left part of the array is occupied by the template pixel values. The remaining array elements are set to 0 by the compiler. The dimensions of the array correspond to those of the largest template in the set.

The C source file is compiled and linked to the other modules later on.

4.2 Estimation of Scale and Rotation

The wedges appear in different sizes in every image depending on their real size on the clay tablets as well as on the distance the pictures were taken from. Similarly, the wedges in the images need not be aligned in the same way as the templates are defined; i.e. a "horizontal" wedge may point somewhere to the top right rather than straight to the right because either the photograph was taken diagonally or the lines on the tablet run diagonally. However, it can be assumed that the orientation of the wedges in one particular image is constant as is the size of the characters within a certain range because a tablet was usually written by one and the same person who did not change his style while writing that tablet. On examining the images one will find that the sizes of wedges of one type vary by about $\pm 15\%$ in one image. This is why each wedge type was eventually represented

by 3 instances, the second and third of which being shrunk to 85% and 70% of the first one, respectively.

Therefore, it is possible to estimate how much the templates in the model set have to be shrunk in only a part of the image and use the optimal scaling factor found to shrink all models by that amount. The same applies to the orientation of the templates. They can be rotated by the angle found to be optimal.

It seems reasonable to run this rotation and scale estimation test in a central region of the image as it is very likely to be covered by characters. The dimensions of that rectangle depend on the dimensions of the whole image; an area of $\frac{1}{5}$ of the image height times $\frac{1}{5}$ of the image width is used.

The test itself employs the algorithm to calculate correlation coefficients, which is dealt with in Chapter 3. In order to distinguish in the program between the first pass, which this section is about, and the second one (the actual template matching over the whole image), a flag *shrink_rot_flag* is defined to indicate that. Its value being 1 means the first pass is being executed whereas 0 stands for the second pass.

The test is carried out as follows: the variable *SHRINKINGS* holds the number of shrinkings to be executed. *SHRINKINGS = 20* would mean that the template set will be scaled to sizes of 100%, 95%, 90%, ..., 5% of the original size. In each of the 20 steps rotations have to be considered as well. *ROT_STEP* tells by how many degrees the template is to be rotated each time. The number of rotations carried out at one "shrinking stage" is 2 × *ROTATIONS* + 1. E.g. *ROT_STEP = 3* and *ROTATIONS = 5* means the template is rotated in steps of 3°. 11 orientations are tried: 0°, +3°, -3°, +6°, -6°, ..., +15°, -15°. Sticking to these examples, each template of the model set would be tested on the central image region 20 × 11 times.

In order to decide which combination of shrinking factor and rotation angle is the best one to use, a double vector *shrink_results[SHRINKINGS * (2 * ROTATIONS + 1)]* is defined to hold a comparative value for each combination applied. This comparative value is the sum of the 27 correlation coefficient maxima, i.e. for each template processed during a particular rotation-scaling combination the corresponding maximum correlation value is looked for. As the central window

that the correlation is calculated in does not change, one only needs to pick the greatest element from the vector to find which rotation angle and scaling factor it stands for. Then, at the beginning of the second pass the templates are shrunk and rotated accordingly before they are used.

4.2.1 Rotation

The function *rotate()* (see Appendix A) was adapted for this program from a routine for HIPS image files written by D. Croft. The input array is rotated around its centre by the argument *angle* given in degrees. The rotated input image is finally written into the array *outarray*. Each pixel in the output array is assigned either the value of the input pixel that maps by the desired rotation onto the output pixel considered or, if the input pixel to be used would be outside the array boundaries, the value specified by the argument *bg_colour*. *Bg_colour* stands for the intensity value of the template background and is +1 as it was already explained in section 4.1.

As an input pixel does not usually map exactly onto one output pixel one could take the pixel intensity value and distribute it over the 2×2 output pixel square on which the input pixel maps taking into consideration the amounts of overlap . This, however, would blur the output image to some extent so that in this function pixel intensity values are transferred only as a whole from the input array to the position in the output array that is closest to the desired one.

Figure 4–2: Left: original wedge model, right: rotated by +33°

4.2.2 Scaling

The function *shrink()* (see Appendix A) is based on a program by Mike Landy (also for HIPS image files) and was changed for the use in this project. It takes

as arguments the number of rows and columns both of the input array and of the output array as well as pointers to the two arrays themselves.

If the input image is scaled down, several pixels or at least parts of several pixels will map onto one pixel in the output array. In Figure 4–3 one entire input pixel (15,9) and parts of the 8 neighbour pixels contribute to the value of the output pixel considered.

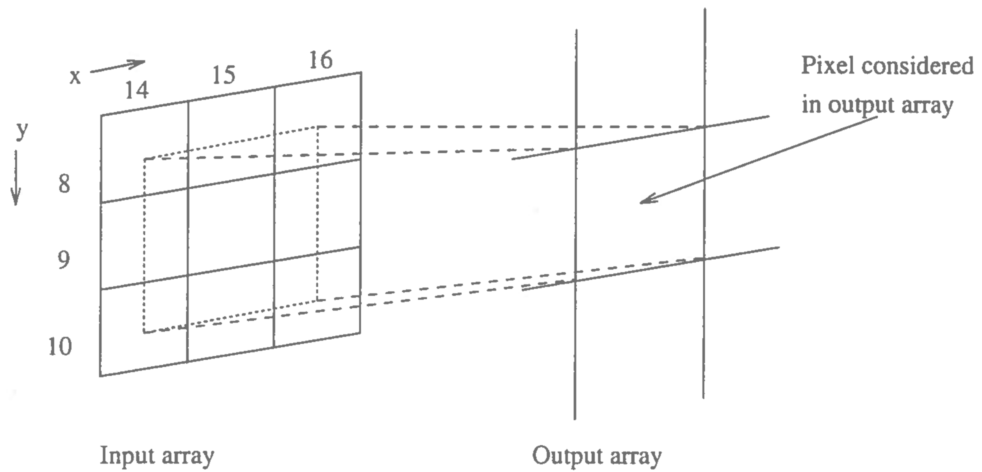


Figure 4–3: How the pixels of the input image map onto a pixel in the shrunk image

In order to find the value to be assigned to the output pixel the percentage of each input pixel mapping onto this output pixel has to be multiplied with the corresponding input pixel intensity value. The sum of these products is the value for the pixel in the output array.

Figure 4–4: Wedge model on the left: original size, centre: shrunk to 70%, right: shrunk to 50%

4.3 Background Elimination

Most of the images available are not completely covered by the clay tablet but contain as well some background. As there are obviously no wedges to be detected in the background, it would be desirable to exclude these areas from any further processing. Another reason to do this is that the time-consuming correlation routine could skip the background regions, which would result in a considerable program speed-up.

The background is characterized by its uniform colour, either bright or black. Thus, the test to discriminate background from tablet areas is based on the deviation of the pixel intensity values within a certain neighbourhood. Identifying background regions uses the mean pixel value of the current window and tests each pixel against the mean. In background regions a majority of pixels should be approximately the mean value (e.g. within ± 5), whereas in regions containing tablet portions a much smaller percentage of pixels should have the mean value. To allow for some noise two thresholds are introduced: *DEV_THRESHOLD* for the permitted difference of a background pixel value from the mean (e.g. 5) and *BG_THRESHOLD* (e.g. 0.02). The number of pixels which do not meet the first condition (i.e. they differ more than *DEV_THRESHOLD* from the mean) is counted and divided by the total number of pixels considered in the window. If this ratio is below the value *BG_THRESHOLD*, the test has been successful and that window region is declared to be background. Otherwise it is assumed to be part of the clay tablet and must be further dealt with.

The test is carried out during the second stage of the template matching (i.e. over the whole image) of only the very first template. Each time such a test succeeds a reference item to the new background patch has to be created and stored. As it is not known how many background patches exist in a particular image, a linked list is used to hold the information. Each list element of the structure type *bg_list_element* represents one background patch. The position of the rectangle's top left corner is written in the slots *x1* and *y1*; the bottom right corner coordinates into *x2* and *y2*.

Figure 4–5 shows again the clay tablet of Figure 2–1 this time with the background areas marked by white frames.

In order to save some memory and run-time, two of the rectangles are merged if they overlap or touch each other and have the same horizontal position (i.e. the same *x1* and *x2* coordinates). In practice, a new list item is not created if this rectangle would just be a vertical extension of an already existing one. Rather, the existing rectangle is pulled down to cover the new background patch as well. This is the reason why some of the frames in figure 4–5 are longer than others.

After the very first template has been dealt with and the list of background regions has been established alongside it is necessary for the processing of all further templates to check at each image position if that corresponding pixel is part of the background. For this purpose, every background list item beginning with the one at the top of the list has to be checked whether the pixel under consideration is situated within the rectangle represented by this list element. If such a background patch is found, a number of pixels can be skipped. As the image is processed row by row from the left to the right, one only needs to set the column counter (representing the x-value of the position considered) to the position right of the right edge of that background window (this value is stored in the *x2* slot of a list element) and continue testing all remaining list elements on the new position until the end of the list is reached. It should be mentioned that the order of the background patches in the list is the same as the order the image positions are processed (i.e. row by row going top down always starting on the left). That is why the above described procedure will work with no need of running through the list more than once.

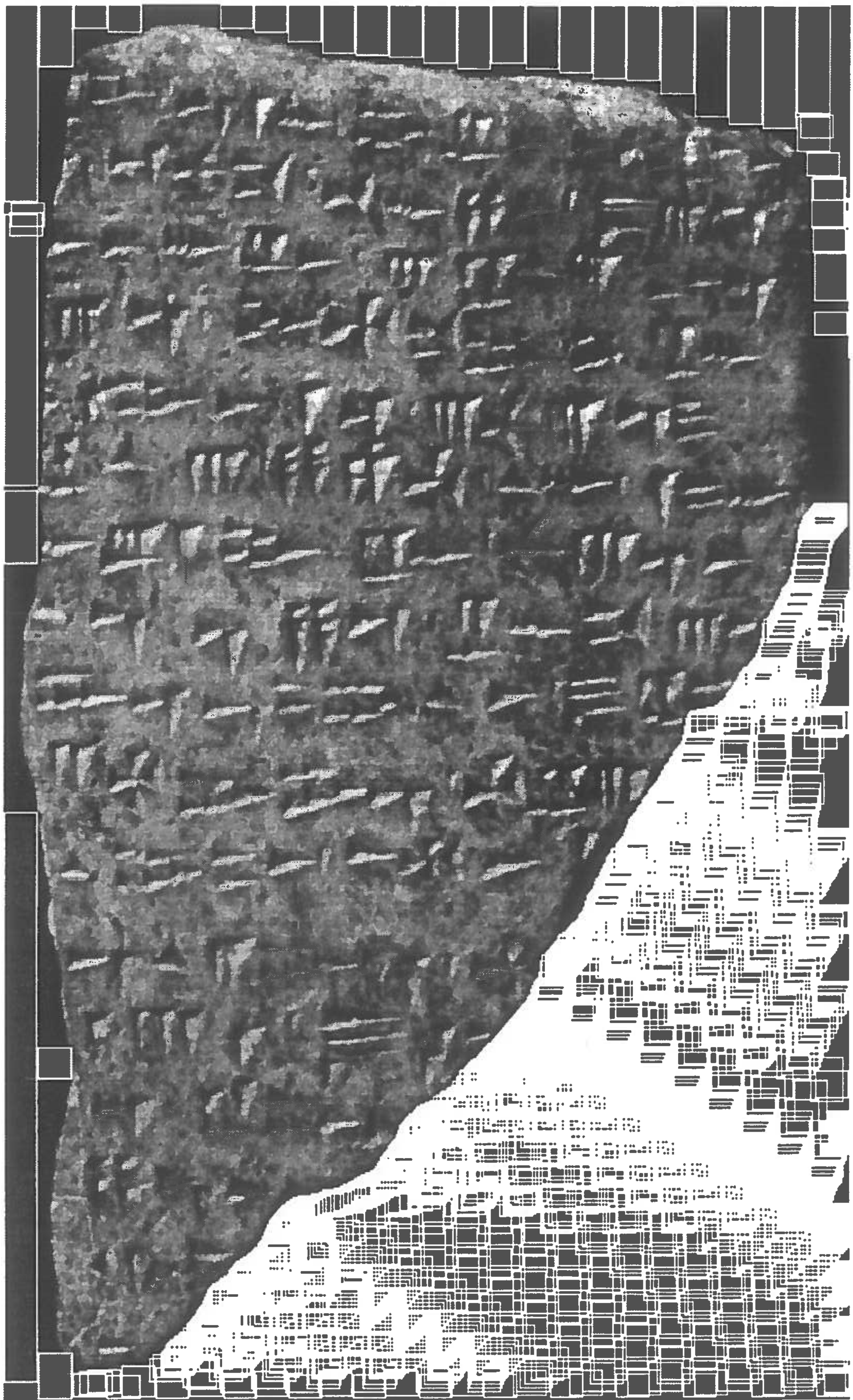


Figure 4–5: Clay tablet image with framed background regions

Chapter 5

Program Structure

The aim of this chapter is to provide a general overview of the actions and the order they are carried out by the program as well as the main data structures used. Details will be given about how to handle the program, which files it consists of, and what their dependencies are.

5.1 The C and C++ Language

The program is written in ANSI-C with the exception that one feature of the C++ language standard was exploited. Unlike in C, it is possible to define variables in C++ at any place in a function, which is of use for instance when the array to store the input image has to be allocated. As its dimensions are not known yet when the program is started, the array can not be defined until the image file's header has been read, which contains the dimensions. Thus, the array has to be allocated dynamically at run-time. One could do that in C, too, using the *malloc()* function of the standard library but it provides only a vector rather than a two-dimensional array so that the index of a pixel in the vector would have to be calculated from the x and y coordinates of that pixel. The source code, however, is easier to follow if a genuine 2-D array holds the data, especially if not only one pixel and perhaps its left and right neighbours are considered but, as it is the case in some places of this program, more pixel positions are involved in calculations at a time. That is the reason why the image data are stored in a dynamically allocated 2-D array.

The C++ compiler used in the course of this project was the GNU compiler *g++ 2.3.3*. Apart from the standard C libraries no other tools or libraries are necessary to compile and run the program. Furthermore, there are no features of Object Oriented Programming in this program though a C++ compiler is needed.

The main program occupies the file *template.cc*. All other functions called by it can be found in the module *funcs.cc*. The data structures as well as the global variables are in the header file *defs.h*. Two more files *defs.cc* and *subdefs.cc* belong to the project. The executable file *template* is built using *make*.

5.2 Data Structures

The main data structures used are one- and two-dimensional arrays, doubly linked lists, and structures.

The wedge models are each represented by an instance of the structure type *temp*.

```
struct temp {
            int rows;
            int cols;
            int wedge_width;
            int wedge_height;
            double t[TEMP_ROWS][TEMP_COLS];  /* both 100 */
        };
```

It contains the double array *t* to store the template pixel intensity values. Furthermore, it possesses two integer slots for the number of columns and rows of the corresponding template and another two integer slots for the width and the height of the wedges within the template window. (The function *filter()* will make use of the latter two.)

The *pgm2c* program (see Chapter 4.1) generates the file *defs.cc* which contains the definition of the vector *basic_temps*[]. This vector has 9 elements, each representing 1 wedge model.

As each of the 9 models has 3 instances of different sizes, another vector of the structure type *temp* is needed, which has 3 times as many elements as *basic_temps*[], i.e. 27. At the beginning of the program the contents of *basic_temps*[] are copied into every third vector element and the remaining 18 elements between are filled with the corresponding data of the 85% and the 70% versions of the original 9 templates. Thus, *templates*[*0*] equals *basic_temps*[*0*], *templates*[*3*] is equal to *basic_temps*[*1*], *templates*[*1*] is 85% of *basic_temps*[*0*], and *templates*[*2*] is 70% of *basic_temps*[*0*] etc. Then, during the preprocessing stage, both scaling and rotation are always applied to all 27 elements of the vector *templates*[].

A similar setup exists for the subtemplates, parts cut out from the original 9 templates. (Their purpose will be described in detail in Chapter 6.) There is a vector *basic_sub_temps*[] whose elements are again of the type *temp*. They are also copied into a 27 element vector *subtemplates*[] and rotation and shrinking are carried out for all 27 subtemplates.

The vector *basic_sub_temps*[] is defined in the file *subdefs.cc*, which is generated by the *pgm2subc* program in a similar fashion as is the file *defs.cc* by *pgm2c*.

The principal output of the program is written into a file and consists of a list of the wedge locations found as well as some additional data for each location. These data are stored in instances of the structure type *list_element* and are linked together to a list because the number of wedges to be detected is unknown beforehand:

```
struct list_element {
                        int temp_nr;
                        int x;
                        int y;
                        int upp_left_x;
                        int upp_left_y;
                        double correlation;
                        int dev;
                        double sub_corr;
                        struct list_element *prev;
                        struct list_element *succ;
                      };
```

The structure slot *temp_nr* is for the number of the template (between 1 and 27) which has matched the image data at that position. The coordinates x and y tell where the upper (left) corner of the found wedge is in the input image. For these reference points and how they are marked in the four basic wedge types see Figure 5–1.

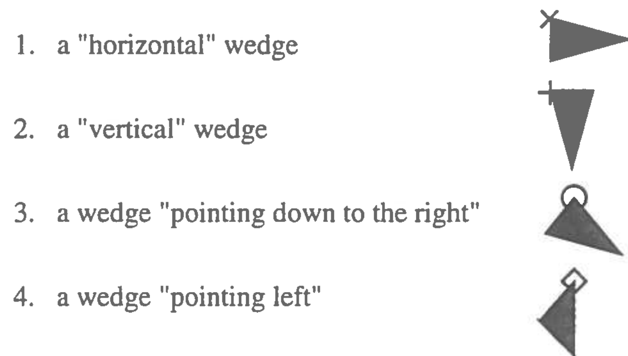


Figure 5–1: How the wedge locations are marked in the output image

In some cases the program also needs to know the coordinates of the upper left corner of the template window with respect to the image coordinates. For this purpose the integer slots *upp_left_x* and *upp_left_y* are provided. Their values are both by some numbers (i.e. pixels) smaller than x and y, by how much depends on the particular template. The remaining three slots hold the data computed during the template matching process: *correlation* stands for the correlation coefficient, *sub_corr* for the correlation coefficient obtained from using the subtemplates, and *dev* contains the standard deviation value. As the structure instances are to be chained to a list, it is of course necessary to have pointers to the previous and to the next list element: *prev* and *succ*.

After the program has been started the header of the input file is read and an unsigned char array *in_array* with the appropriate dimensions is allocated. It is possible to use the type unsigned char because there cannot be more than 256 grey values in an image. Moreover, during the experiments the program sometimes failed to allocate enough memory so that only such data types should be chosen here whose precision can be really exploited. A float array *out_array* of the same dimensions is defined afterwards for the correlation coefficients obtained in each of the 27 matching cycles. Similarly, another unsigned char array *dev* stores the

standard deviation values, which are rounded to integer numbers since the precise values are not required and to save memory.

5.3 Overall Control Structure

5.3.1 Program Options

The executable program *template* is used as a command which reads from the standard input and writes to the standard output. As a PGM file is expected to be read, one should use the program the following way:

```
template [options] < input.pgm > results
```

The output file (here *results*) contains the list of wedge locations mentioned before and can be used as input for subsequent programs that cluster the locations and feed them into a neural net.

The options available are:

-o: A PGM file *overlay.pgm* is generated and written into the current directory. It contains the input image with all wedge locations marked like in Figure 5–1. This option is helpful for analyzing the output.

-k: For each template employed, a file *correlationX.pgm* is output with X being the template number beginning with 0. They are image files of the same size as the input image but each pixel has the correlation coefficients corresponding to the particular coordinates. As the correlation coefficients are between -1 and +1, they must be scaled up to values between 0 and 255 in these images so that bright spots stand for high correlation. It should be mentioned that with this option being turned on 27 such files are generated, which can mean a huge amount of disk space will be occupied. However, it is at times a useful analysis tool as well.

-c: The array containing the correlation coefficients is blurred by convolving it with a 3×3 mask before the local maxima are looked for. The mask looks like this:

$\frac{1}{16}$	$\frac{2}{16}$	$\frac{1}{16}$
$\frac{2}{16}$	$\frac{4}{16}$	$\frac{2}{16}$
$\frac{1}{16}$	$\frac{2}{16}$	$\frac{1}{16}$

This step suppresses some of the local maxima if they are very close together (which would contradict the fact that wedge locations are usually separated by at least 5 pixels in the test images). The greatest local maxima in that region, however, survive this smoothing operation.

-f: The last filtering routine *filter()* is not applied so that no attempt is made to remove multiple responses indicating several detected wedges in a place where only one wedge can be.

-b: The areas found to be largely image background are shown framed like in Figure 4–5.

5.3.2 Control Flow

When the program is started, the flag *shrink_rot_flag* is set to 1 and the preprocessing steps discussed in the previous chapter are carried out. When the optimal shrinking factor and rotation angle have been found the actual template matching begins. (*Shrink_rot_flag* is 0 to indicate that.) The pseudocode on the next page shows the principal actions executed.

The function *sort_list()* (see Appendix A) takes the list of wedge locations as argument and sorts the items with respect to the correlation coefficients. The list element with the greatest correlation coefficient goes to the start of the list. A pointer to the first element of the sorted list is returned.

The local maxima are found by checking the 3×3 neighbourhood of each pixel (i.e. its 8 direct neighbours) if all of them have smaller values than the pixel considered. If so, this pixel is a local maximum.

```
/*** calculate initial correlation values ***/
for each template in templates[] do
   for each pixel in in_array[][] do
   {
      if pixel within background area
         skip positions horizontally
      else
      {
         if first template is being processed
            do background check
         calculate correlation coefficient and write it into out_array[][]
         calculate standard deviation and write it into dev_array[][]
      }
   }
   if blurring is required
      do blurring of out_array[][]
   find local maxima in out_array[][] and store them in list
   append list to glob_list

/*** refine match using subtemplates ***/
for each element in glob_list do

   calculate correlation coefficient using the subtemplates
/*** sort list of matches w.r.t correlation (see end of section) ***/
call sort_list(glob_list)

/*** with 3 features filter the set of matches (see Chapter 6) ***/
for each element in glob_list do
{
   if at least 1 of its 3 features is below a corresponding threshold
      remove this element from glob_list
}

/*** check for false and multiple matches and remove them (see Chap 6) ***/
call filter(glob_list)
for each element in glob_list do
   put out data related to this wedge location
```

Chapter 6

The Template Matching Techniques Employed

This chapter describes the three main features used for the Template Matching, their calculation, and how false matches are suppressed.

6.1 Correlation

The formula to compute the correlation coefficient has been explained in Chapter 3 already. However, its application in this program contains a further condition. As only the triangular areas of the wedges are to be correlated with the image data, the rest of the pixels in a model array have to be excluded from this calculation. For this purpose the background pixels in the model PGM files were set to +1 (as explained earlier) and after the shrinking and rotation operations the same pixels have the value -5; now stored in the array t of the elements of vector *templates*[]. So, only the pixels in the model whose values are greater than -5 are to be considered for the correlation.

Furthermore, the part of the formula involving the model data can be calculated separately in advance and stored in the arrays t of *templates*[] rather than the original intensity values. If in Equation (3.1) in Section 3.2:

$$m'_x = \frac{m_x - \bar{m}}{\sqrt{\sum_k (m_k - \bar{m})^2}} \tag{6.1}$$

the correlation ρ is:

$$\rho = \frac{\sum_x[(d_x - \bar{d}) \cdot m'_x]}{\sqrt{\sum_x (d_x - \bar{d})^2}} \tag{6.2}$$

Using this, the loops to compute the correlation coefficient look like this in C code:

```
for( i=rowstart ; i < rowend ; i++ ) /* go row by row */
  for( j=colstart ; j < colend ; j++ ) /* go line by line */
  {
/********** calculate data mean using the k-th template **********/
    mean = 0.0;
    count = 0;
    for( a=0 ; a < templates[k].rows ; a++ )
      for( b=0 ; b < templates[k].cols ; b++ )
      {
        if( templates[k].t[a][b] > -5.0 )
        {
          mean += (double)in_array[i+a][j+b];
          count++;
        }
      }
      if( count > 0 )
        mean /= count;
/************* calculate correlation and deviation ************/
      dev = 0.0;
      accum = 0.0;
      count = 0;
      for( a=0 ; a < templates[k].rows ; a++ )
        for( b=0 ; b < templates[k].cols ; b++ )
          if( templates[k].t[a][b] > -5.0 )
          {
            diff = (double)in_array[i+a][j+b] - mean;
            accum += diff * templates[k].t[a][b];
            dev += diff * diff;
            count++;
          }
          if( dev > 0.0 && count > 0 )
          {
            dev_array[i][j] = (unsigned char)(sqrt(dev/count) + 0.5);
            /* write correlation into out_array */
            out_array[i][j] = (float)(accum / sqrt( dev ));
          }
  }
```

Experiments with three images have shown that the correlation coefficients for correct matches of a model with a wedge in the data are greater than 0.4 so that this threshold is applied when the local maxima are collected from the *out_array*[][] (see Chapter 5.3.2). I.e. only correlation coefficient maxima greater than 0.4 are stored in *list*, which is appended to *glob_list*. More specific thresholds for each specific wedge type are applied together with others for the standard deviation and the subtemplate correlation when these other two features have been computed.

6.2 Standard Deviation and Subtemplates

When the above mentioned experiments were conducted it also became clear that the correlation itself is a good feature to detect the vast majority of wedges but should not be used without any other constraints. Apart from the correct responses, a great number of false matches with correlation values above 0.4 are output as well. They indicate either the wrong wedge type at a position where really a wedge is (quite often a horizontal wedge instead of a vertical one or vice versa) or, what is even worse, a wedge in some empty tablet area without any wedges. An example of the latter kind of false match is shown in Figure 6–1 where the two pluses in the top-left corner, which is obviously not covered by a vertical nor any other kind of wedge, indicate matches with wedge models of type 2.

Figure 6–1: Two false matches (+) on the top left

The correlation coefficients of the two false responses are 0.54 and 0.58, respectively which is why these matches are very unlikely to be eliminated by a simple thresholding operation. Therefore, additional features had to be sought which would - in some combination with the correlation coefficient - allow the suppression of false matches.

The analysis of three such problem cases showed that the texture in these greyish tablet background regions is of a similar structure as in areas where real wedges are. The patterns are just fainter but still clear enough to effect a good correlation with one of the models. One possible explanation for these faint patterns is that the tablets were re-used after the previous information had been scratched off. Traces of the former wedges might still be present and could cause the problem described. Another explanation is that the presence of random texture in the tablets may be responsible for the probability of random texture similar to wedges.

As the histograms for the data pixel intensity values within the regions where false matches with background happened were not significantly different from histograms of areas with actual wedges, all attempts to exploit features such as the distance between two peaks in the histogram etc. failed.

6.2.1 The Standard Deviation

Theoretically, the standard deviation in a window of image data with real wedges should be much greater than in a window of tablet background where most pixel intensity values do not differ very much from the mean. However, as for the analyzed problem cases, this was not always completely true in the images examined.

The formula to calculate the standard deviation is the following:

$$\sigma = \frac{\sqrt{\sum_i (d_i - \bar{d})^2}}{D} \tag{6.3}$$

It is computed at the same time as the correlation and (as can be observed in the piece of C code in Section 6.1) in the same loop. Again, only those data pixels are taken into consideration whose corresponding model pixels are part of the wedge rather than model background. (Their number is $D = M$ in Equation 6.3).

Wedges in the data with a high contrast (i.e. dark shadows on the top-left and bright areas on the bottom-right) typically have a standard deviation greater than 50 and may be up to 90. Tablet background areas usually have values smaller than 50 or 60; sometimes even below 20. When the deviation values of all matches (correct and false ones) in one tablet are looked at closer, the idea of a general threshold cannot be maintained any longer. As it is the case with the correlation coefficients, one has to introduce separate thresholds for each wedge type and some of the thresholds had to be set without being able to allow an adequate margin above and below them.

6.2.2 Subtemplates

The subtemplates were each cut out from the corresponding full template and each one represents some unique feature of the complete template. For example, the subtemplate corresponding to the stack of three vertical wedges (model in Figure 6–2 on the left) contains only the three tips (right picture in Figure 6–2).

Figure 6–2: Left: enlarged view of template, right: corresponding subtemplate

The purpose of these subtemplates is to correlate the image data of only that small rectangle in which a candidate wedge is located with the subtemplate belonging to the reported wedge type. The idea of this is to check for specific wedge features to further confirm the identification of a wedge.

For example: suppose the i-th item in *glob_list* has wedge type 4. The slot *temp_nr* may have the value 25 to indicate this (i.e. type 4 in its 100% scaled

version has been matched). The subtemplate to be selected for the test will be the one stored in *subtemplates[24]*. (24 because vector indexes start with 0.) It contains part of the full wedge stored in *templates[24]*. What is the rectangular area over which the subtemplate has to be shifted in order to obtain correlation coefficients? Its top-left corner coordinates are held in ($glob_list \rightarrow upp_left_x$, $glob_list \rightarrow upp_left_y$) and the horizontal and vertical offset to calculate the bottom-right corner will be taken from *templates[24].cols* and *templates[24].rows*. In other words: the image data region in which correlation coefficients are computed using a subtemplate has the same dimensions as the template whose match with the data is to be verified. The correlation values are written into *out_array[][]* the previous values of which (resulting from the main processing stage) are no longer needed since all interesting information is stored in *glob_list*. When all correlation coefficients for this subtemplate match are calculated, their maximum is sought and stored in the slot *sub_corr* of the list item considered. This list element (of *glob_list*) contains now all three necessary features (correlation coefficient, standard deviation and subtemplate correlation coefficient) to do the final test.

After each *glob_list* element has obtained its *sub_corr* value, it has to pass a test in which is checked if all three features are above their corresponding thresholds. If either the correlation coefficient or the subtemplate correlation coefficient or the standard deviation value does not meet this condition the list item is removed and destroyed; i.e. the match is declared to be false.

The thresholds applied are shown in Table 6–1 and have been found by conducting tests on three images.

template number	wedge type	correlation threshold	subtemplate correlation threshold	standard deviation threshold
1	1	0.55	0.3	39
2	1	0.55	0.3	37
3	1	0.55	0.3	30
4	1	0.49	0.25	28
5	1	0.42	0.2	25
6	1	0.4	0.2	25
7	1	0.65	0.5	30
8	1	0.55	0.4	30
9	1	0.5	0.4	30
10	2	0.46	0.25	28
11	2	0.46	0.25	28
12	2	0.46	0.25	28
13	2	0.45	0.22	25
14	2	0.45	0.22	25
15	2	0.45	0.22	25
16	1	0.49	0.4	27
17	1	0.44	0.3	26
18	1	0.4	0.3	25
19	2	0.6	0.35	40
20	2	0.6	0.35	40
21	2	0.6	0.35	40
22	3	0.65	0.45	35
23	3	0.55	0.35	33
24	3	0.55	0.35	33
25	4	0.72	0.43	30
26	4	0.65	0.43	30
27	4	0.65	0.43	30

Table 6–1: Thresholds

6.3 Final Filtering

After the matches in *glob_list* have been filtered by applying thresholds for the three features discussed before, a further step remains to be carried out. This last filter aims at removing all implausible matches from *glob_list* and would ideally result in exactly one match per wedge in the image. Implausible matches occur on the one hand because many characters comprise overlapping wedges the partly occluded wedges of which sometimes correlate equally well with several models. On the other hand, there are quite often several local correlation maxima of the same wedge type in the neighbourhood of a real wedge which have passed the previous two tests. The one with the highest correlation value is likely to best indicate the detected wedge.

In order to design a good filter, a number of constraints have to be found which define what plausible matches are.

6.3.1 Allowed Overlaps

One constraint is that in a close neighbourhood of one match no other match can be correct. One or both of the two must be false and therefore rejected. (Although in future projects one might allow a general wedge classification with different types reported at the same place in order to let the neural network disambiguate them by using information about the local configuration of its input wedge locations.) The decision which match to reject can be based on the correlation coefficient; the greater it is the more plausible is the match. Other constraints can be found exploiting the fact that the 30 characters of the alphabet comprise only a limited number of overlapping wedge combinations. As the type of the wedge that belongs to a particular match is available via the *temp_nr* of the list element in *glob_list*, one can look at pairs of matches in terms of the wedge types they indicate and check if the combination of these two wedge types occurs in the characters of the alphabet or not. Of course, such a pair of matches should be tested only if both matches are at positions close together and this is roughly what the function *filter()* does (see Appendix A):

filter() takes the *glob_list* as argument, finds implausible matches, removes the corresponding list elements and returns the (shorter) *glob_list*. The list is sorted according to the correlation coefficients. The element with the greatest one is at the beginning. The function works in two steps. First, each element X is paired with every remaining element Y which comes after X in the list. Thus, no combination is tried twice.

The first step checks if the two wedges represented by the list elements would overlap in the image. If that is not the case, no further test is required and the next pair can be checked. The two matches are plausible and remain (at least at the moment) in the list.

The overlap is found like this: It is tested for each non-background pixel of the first wedge model if by adding the x and y offset (i.e. the distance between the wedges' marked reference points) a non-background position in the other wedge model can be reached. If so, the two wedges overlap.

Should there be any overlap, a second test is carried out which employs constraints on wedge combinations. By default, the match with the smaller correlation coefficient is to be removed unless the following test succeeds. This test checks sequentially the set of allowed overlapping wedge configurations and is structured according to the overlapping pairings possible:

1. 2 overlapping wedges of type 1
2. 2 overlapping wedges of type 2
3. a type 1 wedge overlapping one of type 4
4. a type 4 wedge overlapping one of type 1
5. a type 1 wedge overlapping one of type 2
6. a type 2 wedge overlapping one of type 1
7. a type 2 wedge overlapping one of type 4
8. a type 4 wedge overlapping one of type 2

9. 2 overlapping stacks of type 2 wedges

10. a stack of type 2 wedges overlapping one wedge of type 2

11. a type 2 wedge overlapping a stack of type 2 wedges

12. a stack of type 1 wedges overlapping one wedge of type 2

13. a type 2 wedge overlapping a stack of type 1 wedges

These are the combinations where the overlap is permitted under certain circumstances. They were found by examining the Ugaritic alphabet. To decide whether overlap is permitted, thresholds had to be introduced that work on the base of the width and height of each wedge model. The structure *temp*, of which *templates*[] is an instance, has two slots: *wedge_width* and *wedge_height*. They are initialized at the beginning of the main program with values telling the dimensions (in terms of pixels) that the wedge occupies in the model window: e.g. in a 60×40 window a model wedge of type 1 might be 52 pixels long (*wedge_width*) and 30 pixels high (*wedge_height*). The conditions in *filter()* are based on these two pieces of information; i.e. percentages of them are used.

For example: Two overlapping wedges of type 1 are permissible if the distance between their reference points (where the $\times$ is marked in the output images) is more than 35% of the width of the first wedge. This is quite a small number but as the reference points in the characters *a* and *n* are rather close together, it cannot be increased to 40% or more.

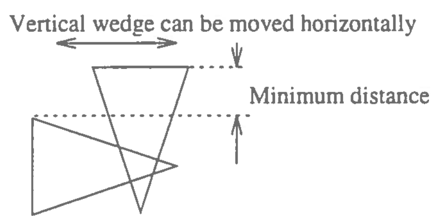
Particularly tricky combinations are all those involving a wedge of type 1 and one of type 2. There are several possibilities the wedges can be placed and also some placements that are not allowed: see Figure 6–3.

6.3.2 Ambiguous Labels

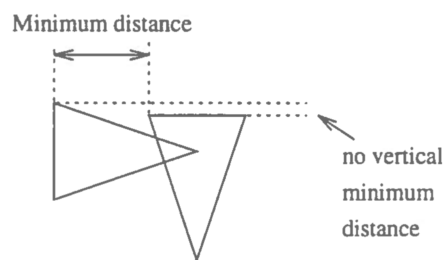
The bottom half of a horizontal wedge resembles the upper part of a vertical wedge so that at times a type 1 model matches a vertical wedge in the image better than a type 2 model. In this special case *filter()* removes either the type 1 wedge (although its correlation coefficient is greater) if the correlation coefficient of the

type 2 wedge is only up to 0.1 smaller than the one of wedge 1 and if additionally wedge 1 is located above wedge 2 (in terms of the y coordinates) or otherwise wedge type 2 will be removed.

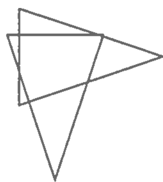
The percentage values used in the conditions as well as some of the conditions were derived empirically through a number of experiments.



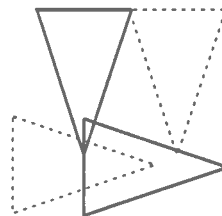
permitted combination



permitted combination



not permitted



permitted combination
(e.g. in letter "b")

Figure 6–3: Combinations of wedges of type 1 and 2

Chapter 7

Experiments and Results

After the parameter adjustments had been completed, the program was tested on four test images. The numbers used for them are the same as the ones they have in the collection they were taken from. Image 169 is the one shown in Chapter 2 already.

The problem encountered at the beginning was that the images 658 and 663 were too large to be processed with the memory available in the computer. So, all images were shrunk and their reduced versions were used by the program.

Each image was processed twice: once with the final *filter()* function enabled (i.e. without setting option -f) and, in order to obtain some reference to assess the performance of *filter()*, without the last filter being applied (option -f was set on the command line). Figures 7–1, 7–2, 7–3, 7–4, 7–5, 7–6, 7–7 and 7–8 show the four pairs of images produced as files *overlay.pgm* each time since the option -o was used throughout. The detected wedges are marked in the way described in Figure 5–1.

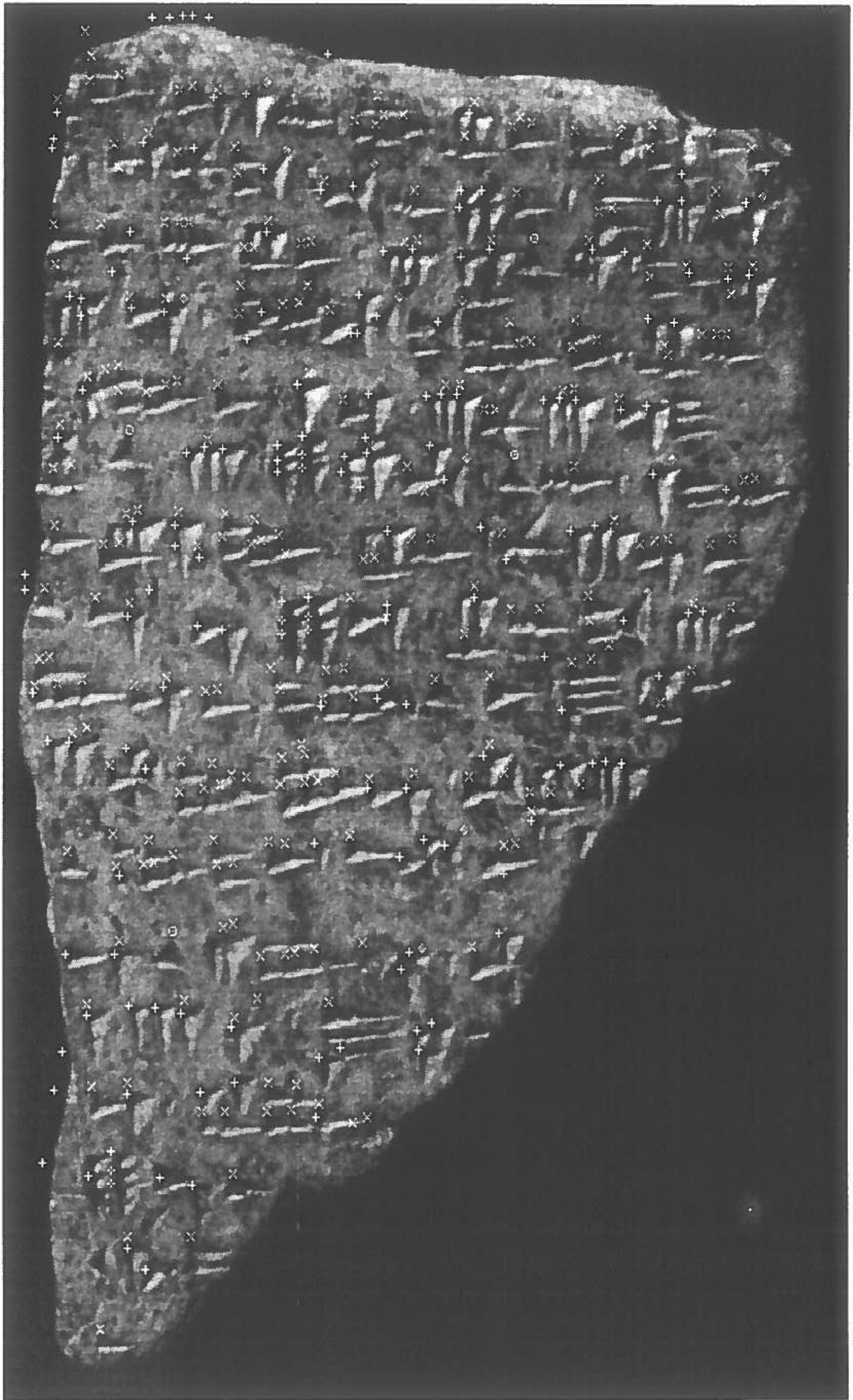


Figure 7–1: Image 169 processed with filtering

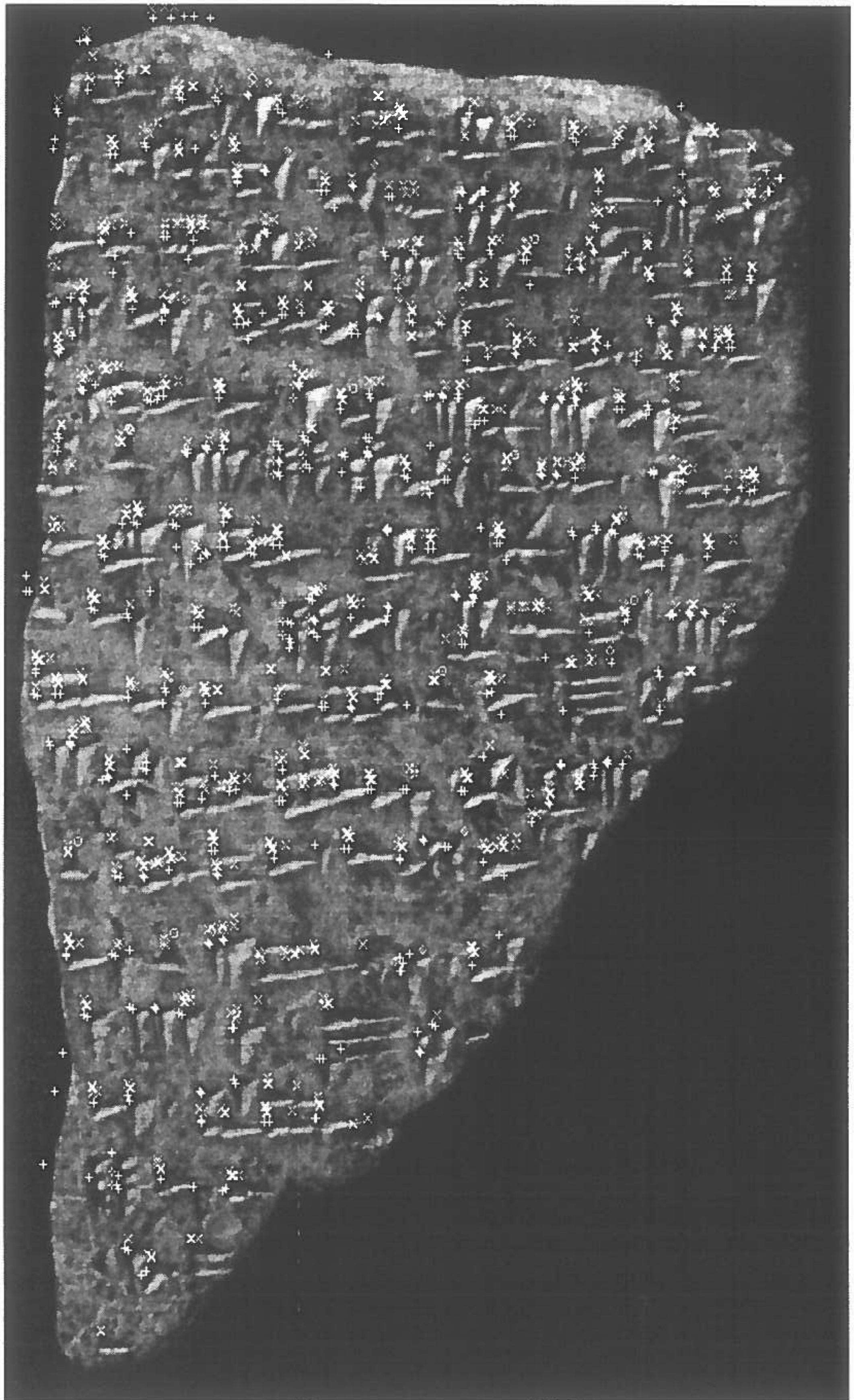


Figure 7–2: Image 169 processed without filtering

Figure 7–3: Image 718 processed with filtering

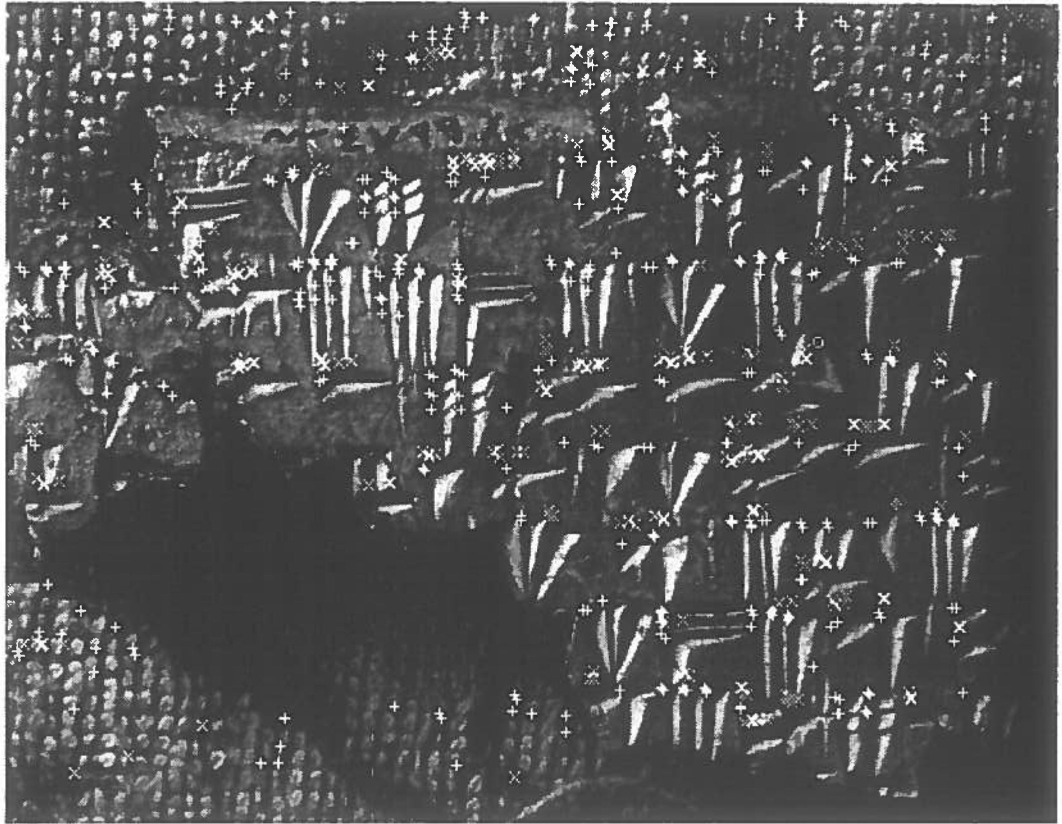


Figure 7–4: Image 718 processed without filtering

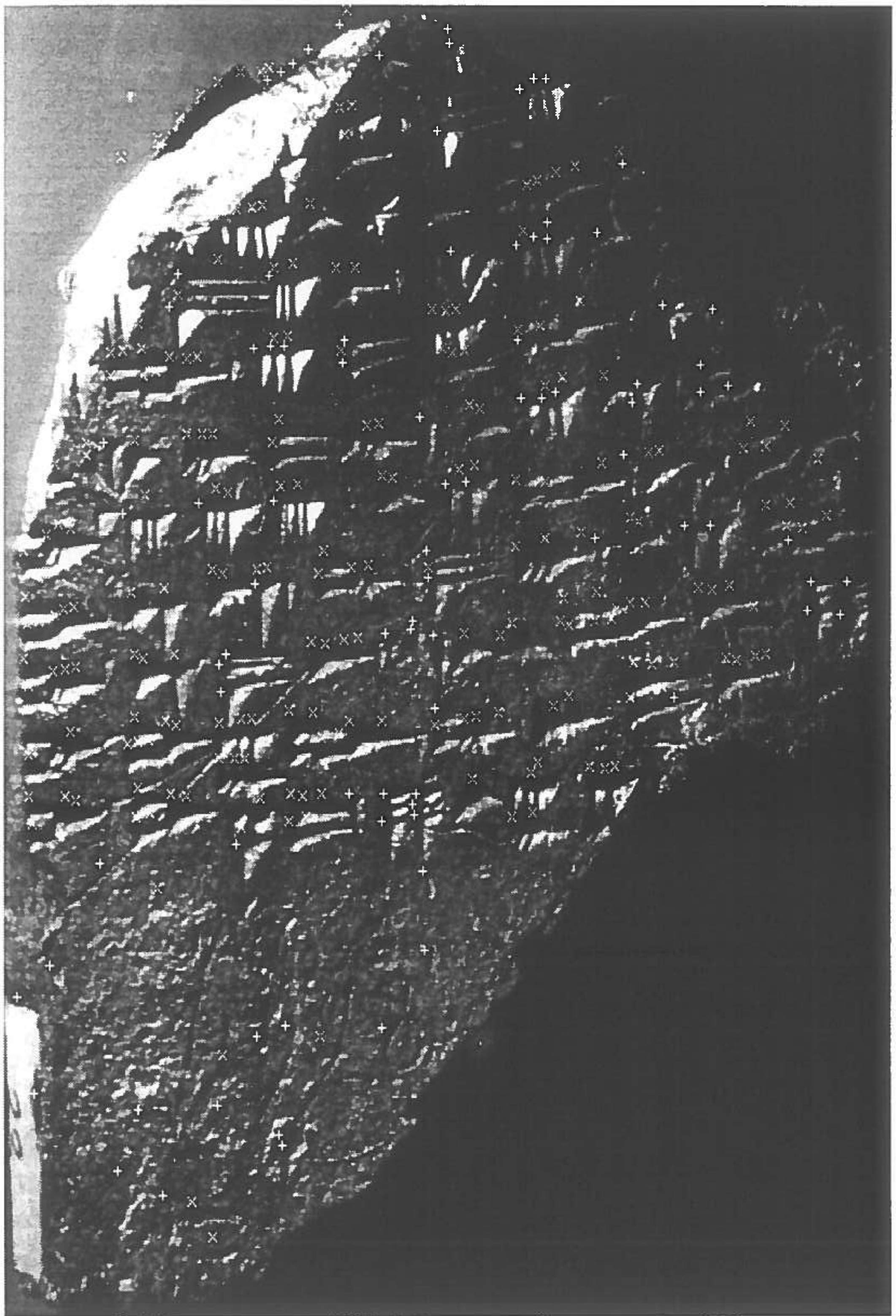


Figure 7–5: Image 658 processed with filtering

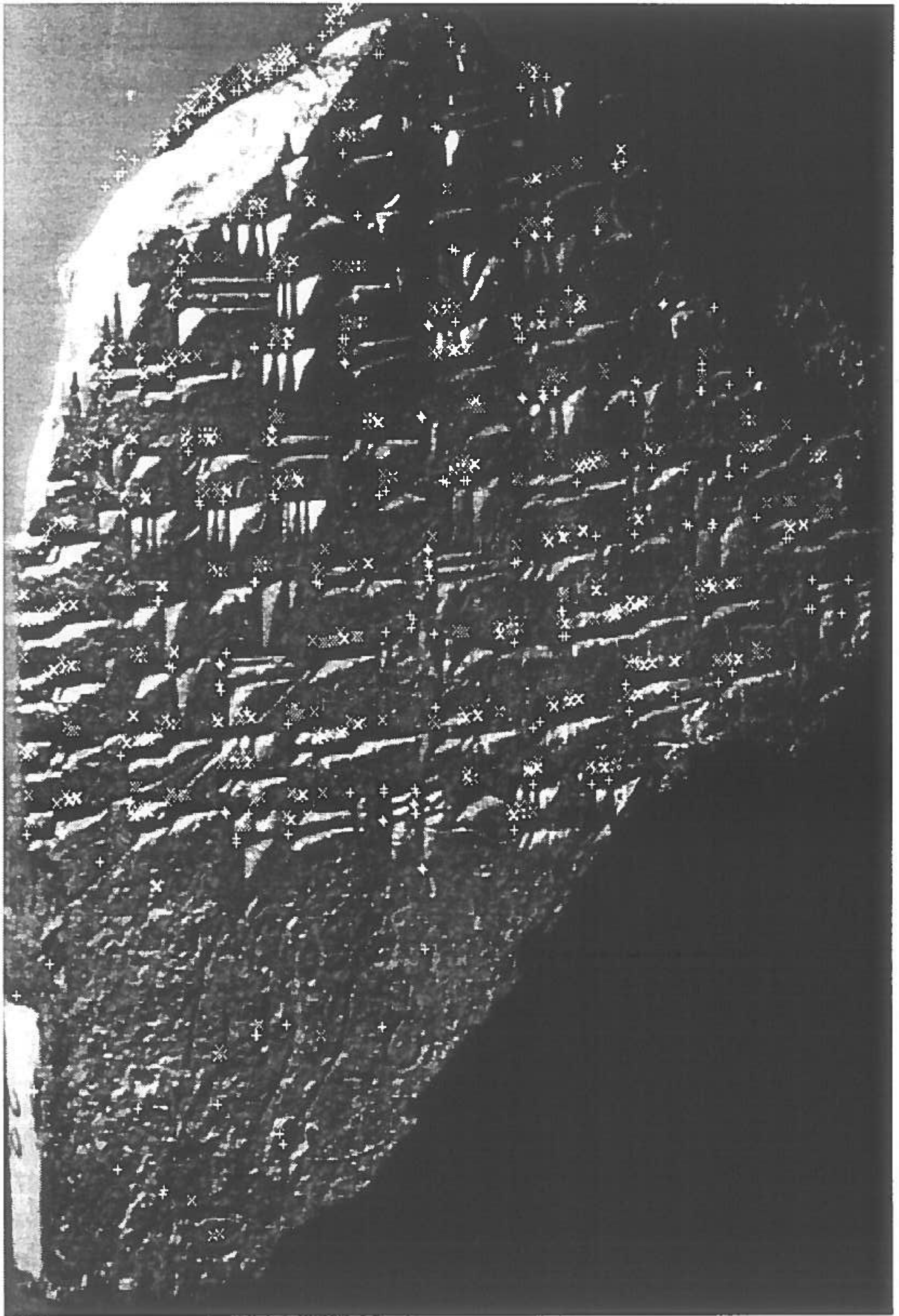


Figure 7–6: Image 658 processed without filtering

Figure 7–7: Image 663 processed with filtering

Figure 7–8: Image 663 processed without filtering

Figure 7–9 contains as an example the first part of the output produced when image 169 was processed.

The run-time for each image amounted to several hours as the correlation algorithm is quite time-consuming. It accounts for about 95% of the time spent. However, correlation is a process that can easily be parallelized and thus run many times faster on several processors at the same time.

The four tables 7–1, 7–2, 7–3, and 7–4 show the percentage of wedges detected. (It is only recorded for each wedge if it was found or not. If it was found, the distinction is made if the wedge type reported is correct as well.) The two percentages given under every table are: r_1 is the recognition rate considering only wedges detected with the correct type whereas r_2 is the ratio of all detected wedges and the total number of wedges in the image.

Table 7–1: Statistics for tablet 169

wedge type	number of wedge appearances	number of wedges found with correct type	number of wedges found with wrong type	number of wedges not detected	number of spurious wedges of this type
1	190	146	6	38	72
2	133	103	1	29	46
3	7	4	3	0	0
4	9	7	0	2	6
Σ	339	260	10	69	124

$r_1 = \frac{260}{339} \hat{=} 76.7\%$ $\qquad r_2 = \frac{260+10}{339} = \frac{270}{339} \hat{=} 79.6\%$

Table 7–2: Statistics for tablet 718

wedge type	number of wedge appearances	number of wedges found with correct type	number of wedges found with wrong type	number of wedges not detected	number of spurious wedges of this type
1	57	32	9	16	33
2	94	74	1	19	38
3	1	0	1	0	0
4	3	1	2	0	0
Σ	155	107	13	35	71

$r_1 = \frac{107}{155} \hat{=} 69\%$ $\qquad r_2 = \frac{107+13}{155} = \frac{120}{155} \hat{=} 77.4\%$

Table 7–3: Statistics for tablet 658

wedge type	number of wedge appearances	number of wedges found with correct type	number of wedges found with wrong type	number of wedges not detected	number of spurious wedges of this type
1	127	94	10	23	70
2	87	46	16	25	28
3	6	0	5	1	0
4	1	0	0	1	0
Σ	221	140	31	50	98

$r_1 = \frac{140}{221} \hat{=} 63.3\%$ $\qquad r_2 = \frac{140+31}{221} = \frac{171}{221} \hat{=} 77.4\%$

Table 7–4: Statistics for tablet 663

wedge type	number of wedge appearances	number of wedges found with correct type	number of wedges found with wrong type	number of wedges not detected	number of spurious wedges of this type
1	158	132	3	23	64
2	126	91	1	34	29
3	4	0	4	0	0
4	4	0	3	1	1
Σ	292	223	11	58	94

$r_1 = \frac{223}{292} \hat{=} 76.4\%$ $\qquad$ $r_2 = \frac{223+11}{292} = \frac{234}{292} \hat{=} 80\%$

The observations made are the following:

Wedges of type 3 (letter *c*) are often misclassified as type 1. This is the case especially in the images 658 and 663 where this wedge correlates better with a small model of type 1 rather than with a type 3 model.

The results became worse towards the tablet edges which is not surprising as the illumination conditions change there (especially when the tablet is not flat) and also because some wedges are not preserved completely.

The tablet background elimination method works quite reliable. Image 169 has some empty areas (i.e. without wedges) where, as it is desired, almost no (false) matches were found. The same applies to tablet 658 which contains much texture but no characters in the bottom third.

The image background elimination could of course not work in image 718. The texture there is due to a piece of cloth. Apart from some background pieces along the tablet edges all image background was identified in the experiments and excluded from any processing. The few regions where wedges were reported erroneously could not be found as the vast majority of pixels in a window must have uniform intensity values for this window to become background. If only one corner of the window juts out and encloses some background, this window will not become a background region so that later matches are possible which report

wedges slightly jutting out as well. As the windows are always slid over the image from the left to the right, this kind of matches happens on the left tablet edges.

As it was described in Chapter 6, there are problems in deciding between wedges of type 1 and 2. When comparing the "unfiltered" and the "filtered" image one can observe that a number of wedges which are not detected in the final filtered output were in fact found but then removed by the function *filter()*. It happens in places where both wedge 1 and 2 types match well a vertical or a horizontal wedge. These cases represent a great part of the misclassified wedges and also some of the missed ones. It has been explained in Chapter 6 that it is possible for a horizontal wedge match (type 1) with higher correlation than an overlapping vertical wedge (type 2) to be removed. If the remaining vertical wedge match is later removed by another match, whatever its type might be, none of the original matches is left to indicate the detection of the wedge in the image. This happens not too often. Most of the wedges missed are either located near or on the tablet edges or have shapes that do not fully correspond to those of the 9 models.

The recognition rates r_1 for correct wedge types detected and r_2 for wedges detected in general are not bad considering that the input data is considerably noisier and more corrupted than it is the case in OCR systems that claim recognition rates of over 95% for typed characters. Thus, the experiments showed that the correlation of models with the image data is a suitable detection method but more refinement in the last filter could produce a better result with less undetected and less misclassified wedges. This is essential when the set of wedge locations will be used as input into a clustering algorithm and subsequently for a neural net.

```
Shrunk down to 100.00%
Templates rotated by  0.00 degrees
25   173    77   0.914902   46   0.720615
22    77   224   0.903374   41   0.668516
 7   431   173   0.844164   43   0.636143
16   131   200   0.832359   47   0.658431
16   300   353   0.817611   50   0.631806
19   187   312   0.816182   52   0.655454
25   284   434   0.812887   37   0.485024
25   227    84   0.812290   39   0.618842
22   327   123   0.807741   40   0.540562
 1   399   143   0.805404   39   0.540754
22   104   487   0.801934   40   0.533403
 1   141    75   0.796116   39   0.559817
 1   143    89   0.791163   45   0.618283
 1    67    74   0.790555   45   0.599044
25   398   309   0.790144   52   0.561457
 1   318   178   0.788208   49   0.586535
19    66   602   0.786538   43   0.455018
 1   338   414   0.786363   41   0.455576
 7    98   115   0.786145   41   0.625205
22   315   237   0.786055   46   0.609941
10   105   274   0.784417   37   0.421658
10    51   531   0.783423   34   0.405940
```

Figure 7–9: Program output
The first column contains the template number (i.e. *temp_nr*), a number between 1 and 27. The second and third columns tell the x and y coordinates of the wedge's marked reference point in the image. The value in the fourth column is the correlation coefficient according to which the table is sorted. The last but one column is for the standard deviation and the last one for the subtemplate correlation coefficients.

Chapter 8

Conclusions and Further Work

8.1 Summary of Main Results

A set of 9 models of stylus strokes (wedges) has been created. Then, one approach to match these models with image data in order to locate the wedges in the image has been successfully implemented. This program, which correlates image data with models to extract features from the image, represents the first part of the Ugaritic character recognition task.

8.2 Limitations and Extensions

The 9 models of wedges are to represent the 4 shapes most frequently found in the Ugaritic texts. However, there are variations due to the writing style of the scribes. In order to make the program more general and flexible, more wedge samples would have to be found, adapted and added to the model set. To allow for easy extensions the model base was created as a set of (small) images which can be translated automatically into program code.

The routine *filter()* to remove false or multiple matches between model and data wedges does work. But, a more sophisticated algorithm that would work in several passes could perform better in those cases when correct matches are removed erroneously or when false matches fail to be removed. Limitations of time did not allow to extend this routine further since each adjustment made had to be verified by at least one test on an image.

When processing large images one encounters the problem that the computer memory is not sufficient. One solution would be to shrink the image (and the models, too, if necessary). Another possibility is to process only part of the image at a time (e.g. first the upper half and later the rest).

It should be mentioned here that the program was designed for the use with the 30 character alphabet. It is not possible to adapt it to another recognition task by simply exchanging the model set. The function *filter()* for example exploits constraints specific to the Ugaritic script.

The clustering algorithm to group the wedge positions as well as the neural net to classify the characters may both need information about the frequency of the 30 characters in the texts. For this purpose Jeff Lloyd collected information from 26 representative texts comprising 7973 characters. In Table 8–1 both the total number of character appearances in these texts and the corresponding frequency is given:

character	number of appearances	frequency in %
a	226	2.82
i	213	2.66
u	69	0.86
b	552	6.90
g	128	1.60
d	340	4.25
ḏ	26	0.32
h	179	2.24
w	215	2.69
z	32	0.40
ḥ	135	1.69
ḫ	115	1.44
ṭ	34	0.42
ẓ	19	0.24
y	331	4.14
k	335	4.19
l	755	9.44
m	679	8.49
n	516	6.45
s	41	0.51
ś	0	0
c	363	4.54
ǵ	60	0.75
p	290	3.62
ṣ	117	1.46
q	111	1.39
r	540	6.75
š	419	5.24
t	704	8.80
ṯ	256	3.20
uncertain characters	173	2.16

Table 8–1: Frequency of Ugaritic characters in a set of 26 texts

8.3 Alternative Approaches

An approach slightly different from the one in this project can be made if each tablet is photographed twice: once with the light coming from the left and the second time with the light from the right. Camera and tablet must be kept in the same fixed position. The intensity values of one image can be subtracted from the ones of the other image. On the assumption that the tablet and image background in both images has roughly the same intensity values these regions would cancel each other out and result in intensity values of approximately 0. The places that are covered by characters are supposed to yield values which are significantly different from 0. It should be possible to find a suitable threshold to produce a binary image that contains the shapes of the wedges. After some normalization procedures the image could be fed into a self-organizing neural net directly. Without having to compute correlation coefficients the preprocessing would be much faster.

Another approach is the use of range images rather than light intensity images in order to obtain binary images. This is possible since the characters are impressed into the clay. The problem is that the range finder to be used to scan the tablets would need to have quite a high resolution of about 0.1 mm.

Bibliography

[Craigie 83] P.C. Craigie. *Ugarit and The Old Testament.* Eerdmans, Grand Rapids, Michigan, 1983.

[Curtis 85] A. Curtis. *Ugarit (Ras Shamra).* Lutterworth Press, Cambridge, 1985.

[Denker 89] J. S. Denker. Neural network recognizer for handwritten zip code digits. In D. S. Touretzky, editor, *Advances In Neural Information Processing Systems 1*, pages 323–331. Morgan Kaufmann, 1989.

[Denker 90] J. S. Denker. Handwritten digit recognition with a back-propagation network. In D. S. Touretzky, editor, *Advances In Neural Information Processing Systems 2*, pages 396–404. Morgan Kaufmann, 1990.

[Denker *et al.* 89] J. S. Denker, R. E. Howard, and B. Boser. Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1(4), Winter 1989.

[Fukushima & Miyake 82] K. Fukushima and S. Miyake. Neocognitron: A new algorithm for pattern recognition tolerant of deformations and shifts in position. *Pattern Recognition*, 15, 1982.

[Mighell 89] D. A. Mighell. Backpropagation and its application to handwritten signature verification. In D. S. Touretzky, editor, *Advances In Neural Information Processing Systems 1*, pages 340–347. Morgan Kaufmann, 1989.

[Mori & Yokosawa 89] Y. Mori and K. Yokosawa. Neural networks that learn to discriminate similar kanji characters. In D. S. Touretzky, editor, *Advances In Neural Information Processing Systems 1*, pages 332–339. Morgan Kaufmann, 1989.

Appendix A

Program code

```
/*******************************************************************/
/***                    Header file defs.h                       ***/
/*******************************************************************/

#define NUMBER_OF_TEMPLATES 9
#define SHRINKINGS 1
#define ROTATIONS 0
#define SHR_FACT 0.15               //  i.e shrink by 15% each time
#define ROT_STEP 5                  //  i.e. rotate in 5 degree steps
#define DEV_THRESHOLD 5
#define BG_THRESHOLD 0.03           //
#define TEMP_ROWS 100
#define TEMP_COLS 100

struct temp {
               int rows;
               int cols;
               int wedge_width;
               int wedge_height;
               double t[TEMP_ROWS][TEMP_COLS];
            };

struct list_element {
                        int temp_nr;
                        int x;
                        int y;
                        int upp_left_x;
                        int upp_left_y;
                        double correlation;
                        int dev;
                        double sub_corr;
                        struct list_element *prev;
                        struct list_element *succ;
                      };
```

```
struct bg_list_element {
                         int x1;
                         int x2;
                         int y1;
                         int y2;
                         struct bg_list_element *prev;
                         struct bg_list_element *succ;
                       };

struct list_element *sort_list( struct list_element * );

int shrink( double inarray[][TEMP_COLS], int rows, int cols,
            double outarray[][TEMP_COLS], int outrows, int outcols );

int rotate( double angle, double inarray[][TEMP_COLS],
                         double outarray[][TEMP_COLS],
                         int cols, int rows, int bg_colour );

struct list_element *filter( struct list_element *start,
                             struct temp *templates );


extern struct temp basic_temps[NUMBER_OF_TEMPLATES];

extern struct temp basic_sub_temps[NUMBER_OF_TEMPLATES];

/*********************************************************************/
```

```
/*********************************************************************/
/***  file: funcs.cc                                             ***/
/*********************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "defs.h"

/***********************************************************************/
/***  function: shrink                                               ***/
/***  scales inarray down to outarray                                ***/
/***            based on code for HIPS by Mike Landy                 ***/
/***********************************************************************/

int shrink( double inarray[][TEMP_COLS], int rows, int cols,
            double outarray[][TEMP_COLS], int outrows, int outcols )
{
  int k, c, r, inxp, inyp;
  register double *ip3, *ip4;
  double xfactor, yfactor, xmag, ymag, xlo, xhi, ylo, yhi, sum, *ip,
         tmp, tmp2, x0tmp, x1tmp, y0tmp, y1tmp;

  xfactor = (double)outcols / cols;
  yfactor = (double)outrows / rows;
  xmag = 1.0 / xfactor;
  ymag = 1.0 / yfactor;

  int fxlo[outcols];
  int fxhi[outcols];
  int inxpix[outcols];
  double xlofract[outcols];
  double xhifract[outcols];
  double xmifract[outcols];

  for (c = 0; c < outcols; c++)
  {
    if (xfactor == 1.0)
    {
      /* important special case */
      fxlo[c] = c;
      inxpix[c] = -1;
    }
    else
    {
      xlo = (double)c * xmag;
      xhi = xlo + xmag;
      /* truncate */
      fxlo[c] = (int)xlo;
      fxhi[c] = (int)xhi;
      /* if on integer boundary, ignore */
      if ((double)fxhi[c] == xhi)
        fxhi[c]--;
```

```
      inxpix[c] = fxhi[c] - fxlo[c] - 1;
    }
    if (inxpix[c] == 0)
    {
      x0tmp = fxlo[c] + 1 - xlo;
      xlofract[c] = x0tmp / xmag;
      xhifract[c] = 1.0 - xlofract[c];
    }
    else
      if (inxpix[c] > 0)
      {
        x0tmp = fxlo[c] + 1 - xlo;
        x1tmp = xhi - fxhi[c];
        xlofract[c] = x0tmp / xmag;
        xhifract[c] = x1tmp / xmag;
        xmifract[c] = 1.0 - xlofract[c] - xhifract[c];
      }
  }

  int fylo[outrows];
  int fyhi[outrows];
  int inypix[outrows];
  double ylofract[outrows];
  double yhifract[outrows];
  double ymifract[outrows];

  for (r = 0; r < outrows; r++)
  {
    if (yfactor == 1.0)
    {
      /* important special case */
      fylo[r] = r;
      inypix[r] = -1;
    }
    else
    {
      ylo = r * ymag;
      yhi = ylo + ymag;
      /* truncate */
      fylo[r] = (int)ylo;
      fyhi[r] = (int)yhi;
      /* if on integer boundary, ignore */
      if ((double)fyhi[r] == yhi)
        fyhi[r]--;
      inypix[r] = fyhi[r] - fylo[r] - 1;
    }
    if (inypix[r] == 0)
    {
      y0tmp = fylo[r] + 1 - ylo;
      ylofract[r] = y0tmp / ymag;
      yhifract[r] = 1.0 - ylofract[r];
    }
```

```
  else
    if (inypix[r] > 0)
    {
      y0tmp = fylo[r] + 1 - ylo;
      y1tmp = yhi - fyhi[r];
      ylofract[r] = y0tmp / ymag;
      yhifract[r] = y1tmp / ymag;
      ymifract[r] = 1.0 - ylofract[r] - yhifract[r];
    }
}
for( r=0 ; r<outrows ; r++ )
{
  ip = &inarray[r][0]; // + fylo[r]*cols;
  for( c=0 ; c<outcols ; c++ )
  {
    inxp = inxpix[c];
    inyp = inypix[r];
    /* start first the lower row */
    sum = 0.0;
    ip4 = ip + fxlo[c];
    ip3 = ip4 + 1;
    if (inxp > 0)
    {
      /* add in the middle region */
      /* this contains whole pixels across */
      for (k = inxp; k > 0; k--)
        sum += *ip3++;
      sum *= xmifract[c]; /* weight */
      sum /= (double)inxp;
    }
    if (inxp >= 0)
    {
      /* add in partial regions */
      /* these are at the start and end */
      sum += (*ip4) * xlofract[c];
      sum += (*ip3) * xhifract[c];
    }
    else     /* inxp < 0 */
        /* everything within one pixel across */
      sum += *ip4;
    if (inyp < 0)
    {
      /* only one row to consider */
      /* everything fits in a y pixel */
      outarray[r][c] = sum;
      continue;
    }
    sum *= ylofract[r]; /* weight */
    ip4 += cols;
    tmp = sum;
       /* add the middle rows */
    if (inyp > 0)
```

```
{
  /* whole rows to add in */
  register int l;
  tmp2 = 0.0;
  for (l = inyp; l > 0; l--)
  {
    sum = 0.0;
    ip3 = ip4 + 1;
    if (inxp > 0)
    {
      for (k = inxp; k > 0; k--)
        sum += *ip3++;
      sum *= xmifract[c];
      sum /= (double)inxp;
    }
    if (inxp >= 0)
    {
       /* add in partial regions */
      sum += (*ip4) * xlofract[c];
      sum += (*ip3) * xhifract[c];
    }
    else
      sum += *ip4;
    ip4 += cols;
    tmp2 += sum;
  }
  tmp2 *= ymifract[r];
  tmp2 /= (double)inyp;
  tmp += tmp2;
}
  /* now the upper row */
sum = 0.0;
ip3 = ip4 + 1;
if (inxp > 0)
{
  /* add in the middle region */
  for (k = inxp; k > 0; k--)
    sum += *ip3++;
  sum *= xmifract[c]; /* weight */
  sum /= (double)inxp;
}
if (inxp >= 0)
{
  /* add in partial regions */
  sum += (*ip4) * xlofract[c];
  sum += (*ip3) * xhifract[c];
}
else     /* inxp < 0 */
  sum += *ip4;
sum *= yhifract[r];

/* combine and normalize */
```

```
      sum += tmp;
      outarray[r][c] = sum;
    }
  }
  return 0;
}
```

```
/*********************************************************************/
/***  function: rotate                                             ***/
/***  rotates inarray by angle degrees & returns outarray          ***/
/***  bg_colour: value for the pixels that do not have a           ***/
/***             corresponding ''input'' pixel                     ***/
/***             based on code for HIPS by D. Croft                ***/
/*********************************************************************/

int rotate( double angle, double inarray[][TEMP_COLS],
            double outarray[][TEMP_COLS],
            int cols, int rows, int bg_colour )
{
  int i, j, x2, y2, x_centre, y_centre;
  double sine, cosine, dist_x, dist_y, tmp_array[TEMP_ROWS][TEMP_COLS];

  x_centre = cols / 2;
  y_centre = rows / 2;
  sine = sin( angle * M_PI / 360.0 );
  cosine = cos( angle * M_PI / 360.0 );
  for( i=0 ; i<rows ; i++ )
  {
    dist_y = (double)(i - y_centre);
    for( j=0 ; j<cols ; j++ )
    {
      dist_x = (double)(j - x_centre);
      x2 = x_centre + (int)(cosine * dist_x - sine * dist_y);
      y2 = y_centre + (int)(cosine * dist_y + sine * dist_x);
      if((x2 < 0) || (y2 < 0) || (x2 >= cols) || (y2 >= rows))
        tmp_array[i][j] = bg_colour;
      else
        tmp_array[i][j] = inarray[y2][x2];
    }
  }
  for( i=0 ; i<rows ; i++ )
    for( j=0 ; j<cols ; j++ )
      outarray[i][j] = tmp_array[i][j];
  return 0;
}
```

```
/*********************************************************************/
/***  function: sort_list                                          ***/
/***  sorts a doubly linked list of structures list_element        ***/
/***  with respect to the slot correlation; the highest one        ***/
/***  comes first                                                  ***/
/*********************************************************************/

struct list_element *sort_list( struct list_element *list )
{
  double max;
  struct list_element *start, *aux_ptr, *ret;

  start = list;
  ret = aux_ptr = NULL;
  while( start != NULL )
  {
    max = 0.0;
    while( list != NULL )
    {
      if( list->correlation > max )
      {
        max = list->correlation;
        aux_ptr = list;
      }
      list = list->succ;
    }
    if( start != aux_ptr )
    {
      if( aux_ptr->prev != NULL )
        aux_ptr->prev->succ = aux_ptr->succ;
      if( aux_ptr->succ != NULL )
        aux_ptr->succ->prev = aux_ptr->prev;
      aux_ptr->prev = start->prev;
      if( start->prev == NULL )
        ret = aux_ptr;
      aux_ptr->succ = start;
      if( aux_ptr->prev != NULL )
        aux_ptr->prev->succ = aux_ptr;
      if( aux_ptr->succ != NULL )
        aux_ptr->succ->prev = aux_ptr;
      list = start;
    }
    else
    {
      if( start->prev == NULL )
        ret = aux_ptr;
      start = start->succ;
      list = start;
    }
  }
  return ret;
}
```

```
/**********************************************************************/
/***  function: filter                                             ***/
/***  checks all possible pairings of list-elements for           ***/
/***  overlap                                                      ***/
/***  if they overlap check if the configuration is permitted     ***/
/***  (i.e. possible in one of the Ugaritic characters)           ***/
/**********************************************************************/

struct list_element *filter( struct list_element *start,
                             struct temp *templates )
{
  int a, b, i, j, xdiff, ydiff, overlap, del, stop;
  struct list_element *s, *l, *ptr, *prev_list_elem;

  s = start;
  while( s != NULL )
  {
    l = s->succ;
    while( l != NULL )
    {
      // 1. check for overlap
      overlap = 0;
      xdiff = s->x - l->x;
      ydiff = s->y - l->y;
      for( i=0 ; i<templates[s->temp_nr-1].rows ; i++ )
        for( j=0 ; j<templates[s->temp_nr-1].cols ; j++ )
          if( templates[s->temp_nr-1].t[i][j] > -5.0 )
            for( a=0 ; a<templates[l->temp_nr-1].rows ; a++ )
              for( b=0 ; b<templates[l->temp_nr-1].cols ; b++ )
                if( overlap == 0 &&
                   templates[l->temp_nr-1].t[a][b] > -5.0 &&
                  ( s->x + j == l->x + b ) &&
                  ( s->y + i == l->y + a ))
                  overlap = 1;

      // 2. check distances if there is overlap
      if( overlap == 1 )
      {
        del = 1;          // by default
                // type 1 with type 1
        if(( s->temp_nr ==  1 || s->temp_nr ==  2 || s->temp_nr ==  3 ||
             s->temp_nr ==  4 || s->temp_nr ==  5 || s->temp_nr ==  6 ||
             s->temp_nr == 16 || s->temp_nr == 17 || s->temp_nr == 18 ) &&
           ( l->temp_nr ==  1 || l->temp_nr ==  2 || l->temp_nr ==  3 ||
             l->temp_nr ==  4 || l->temp_nr ==  5 || l->temp_nr ==  6 ||
             l->temp_nr == 16 || l->temp_nr == 17 || l->temp_nr == 18 ))
          if( sqrt((l->x - s->x) * (l->x - s->x) +
                   (l->y - s->y) * (l->y - s->y)) >
                   0.35 * templates[s->temp_nr].wedge_width )
            del = 0;
          else
```

```
        del = 1;
                       // Type 2 with type 2
    if(( s->temp_nr == 10 || s->temp_nr == 11 || s->temp_nr == 12 ||
         s->temp_nr == 13 || s->temp_nr == 14 || s->temp_nr == 15 ) &&
       ( l->temp_nr == 10 || l->temp_nr == 11 || l->temp_nr == 12 ||
         l->temp_nr == 13 || l->temp_nr == 14 || l->temp_nr == 15 ))
      if( sqrt((l->x - s->x) * (l->x - s->x) +
               (l->y - s->y) * (l->y - s->y)) >
                0.45 * templates[s->temp_nr].wedge_width )
        del = 0;
      else
        del = 1;
                       // Type 1 with type 4
    if(( s->temp_nr ==  1 || s->temp_nr ==  2 || s->temp_nr ==  3 ||
         s->temp_nr ==  4 || s->temp_nr ==  5 || s->temp_nr ==  6 ||
         s->temp_nr == 16 || s->temp_nr == 17 || s->temp_nr == 18 ) &&
       ( l->temp_nr == 25 || l->temp_nr == 26 || l->temp_nr == 27 ))
      if( l->x - s->x > 0.9 * templates[s->temp_nr].wedge_width )
        del = 0;
      else
        del = 1;
                       // Type 1 with type 2
    if(( s->temp_nr ==  1 || s->temp_nr ==  2 || s->temp_nr ==  3 ||
         s->temp_nr ==  4 || s->temp_nr ==  5 || s->temp_nr ==  6 ||
         s->temp_nr == 16 || s->temp_nr == 17 || s->temp_nr == 18 ) &&
       ( l->temp_nr == 10 || l->temp_nr == 11 || l->temp_nr == 12 ||
         l->temp_nr == 13 || l->temp_nr == 14 || l->temp_nr == 15 ))
      if( s->y - l->y > - 0.35 * templates[l->temp_nr].wedge_height &&
          s->y - l->y < 0.8 * templates[l->temp_nr].wedge_height )
{
        ptr = start;
        stop = 0;
        while( s->correlation - l->correlation <
               0.1 && ptr != NULL && stop != 1 )
  {
          if(( ptr->temp_nr == 10 || ptr->temp_nr == 11
                                  || ptr->temp_nr == 12 ||
               ptr->temp_nr == 13 || ptr->temp_nr == 14
                                  || ptr->temp_nr == 15 ) &&
               abs( ptr->x - l->x ) <
                     0.9 * templates[l->temp_nr].wedge_width &&
               abs( ptr->y - l->y ) <
                      0.2 * templates[l->temp_nr].wedge_height )
            stop = 1;
          ptr = ptr->succ;
        }
        if( stop == 1 && l->y - s->y > 0 && l->y - s->y <
                     0.4 * templates[l->temp_nr].wedge_height )
  {
          if( s->prev != NULL )
            s->prev->succ = s->succ;
          if( s->succ != NULL )
```

```
                s->succ->prev = s->prev;
              prev_list_elem = s;        }

              s = s->prev;
              free( prev_list_elem );
              del = 0;
              break;
            }
            else
      if( l->x - s->x > 0.6 * templates[s->temp_nr].wedge_width ||
                                                // be far enough on the right
  s->y - l->y > 0.3 * templates[l->temp_nr].wedge_height &&
                                                // or far enough above
                  l->x - s->x > - 0.4 * templates[s->temp_nr].wedge_width )
                del = 0;
              else
                del = 1;
          }
                           // Type 3*1 with type 2
        if(( s->temp_nr ==  7 || s->temp_nr ==  8 || s->temp_nr ==  9 ) &&
           ( l->temp_nr == 10 || l->temp_nr == 11 || l->temp_nr == 12 ||
             l->temp_nr == 13 || l->temp_nr == 14 || l->temp_nr == 15 ))
          if( s->y - l->y > 0.2 * templates[l->temp_nr].wedge_height )
            del = 0;
          else
            del = 1;
                           // Type 2 with type 1
        if(( s->temp_nr == 10 || s->temp_nr == 11 || s->temp_nr == 12 ||
             s->temp_nr == 13 || s->temp_nr == 14 || s->temp_nr == 15 ) &&
           ( l->temp_nr ==  1 || l->temp_nr ==  2 || l->temp_nr ==  3 ||
             l->temp_nr ==  4 || l->temp_nr ==  5 || l->temp_nr ==  6 ||
             l->temp_nr == 16 || l->temp_nr == 17 || l->temp_nr == 18 ))
          if( l->y - s->y > - 0.35 * templates[s->temp_nr].wedge_height &&
              l->y - s->y < 0.8 * templates[s->temp_nr].wedge_height )
            del = 0;
          else
            del = 1;
                            // Type 2 with type 4
        if(( s->temp_nr == 10 || s->temp_nr == 11 || s->temp_nr == 12 ||
             s->temp_nr == 13 || s->temp_nr == 14 || s->temp_nr == 15 ) &&
           ( l->temp_nr == 25 || l->temp_nr == 26 || l->temp_nr == 27 ))
          if( abs( s->y -l->y) < 0.3 * templates[s->temp_nr].wedge_height &&
              l->x - s->x >= templates[s->temp_nr].wedge_width )
            del = 0;
          else
            del = 1;
                            // Type 2 with type 3*2
        if(( s->temp_nr == 10 || s->temp_nr == 11 || s->temp_nr == 12 ||
             s->temp_nr == 13 || s->temp_nr == 14 || s->temp_nr == 15 ) &&
           ( l->temp_nr == 19 || l->temp_nr == 20 || l->temp_nr == 21 ))
          if( abs( s->x - l->x ) > 0.8 * templates[s->temp_nr].wedge_width )
            del = 0;
```

```
  else
    del = 1;
                          // Type 2 with type 3*1
if(( s->temp_nr == 10 || s->temp_nr == 11 || s->temp_nr == 12 ||
     s->temp_nr == 13 || s->temp_nr == 14 || s->temp_nr == 15 ) &&
   ( l->temp_nr ==  7 || l->temp_nr ==  8 || l->temp_nr ==  9 ))
  if( l->y - s->y > 0.2 * templates[s->temp_nr].wedge_height )
    del = 0;
  else
    del = 1;
                          // Type 3 with all others
if( s->temp_nr == 22 || s->temp_nr == 23 || s->temp_nr == 24 )
  del = 1;
                          // Type 4 with type 2
if(( s->temp_nr == 25 || s->temp_nr == 26 || s->temp_nr == 27 ) &&
   ( l->temp_nr == 10 || l->temp_nr == 11 || l->temp_nr == 12 ||
     l->temp_nr == 13 || l->temp_nr == 14 || l->temp_nr == 15 ))
  if( abs( s->y -l->y )< 0.17 * templates[s->temp_nr].wedge_height &&
      s->x - l->x > templates[s->temp_nr].wedge_width )
    del = 0;
  else
    del = 1;
                          // Type 4 with type 1
if(( s->temp_nr == 25 || s->temp_nr == 26 || s->temp_nr == 27 ) &&
   ( l->temp_nr ==  1 || l->temp_nr ==  2 || l->temp_nr ==  3 ||
     l->temp_nr ==  4 || l->temp_nr ==  5 || l->temp_nr ==  6 ||
     l->temp_nr == 16 || l->temp_nr == 17 || l->temp_nr == 18 ))
  if( s->x - l->x > templates[s->temp_nr].wedge_width )
    del = 0;
  else
    del = 1;
                          // Type 3*2 with type 3*2
if(( s->temp_nr == 19 || s->temp_nr == 20 || s->temp_nr == 21 ) &&
   ( l->temp_nr == 19 || l->temp_nr == 20 || l->temp_nr == 21 ))
  if( abs( s->y - l->y ) < 0.2 * templates[s->temp_nr].wedge_height )
    del = 0;
  else
    del = 1;
                          // Type 3*2 with type 2
if(( s->temp_nr == 19 || s->temp_nr == 20 || s->temp_nr == 21 ) &&
   ( l->temp_nr == 10 || l->temp_nr == 11 || l->temp_nr == 12 ||
     l->temp_nr == 13 || l->temp_nr == 14 || l->temp_nr == 15 ))
  if( abs( s->x - l->x ) > 0.8 * templates[s->temp_nr].wedge_width )
    del = 0;
  else
    del = 1;

if( del == 1 )
{
  l->prev->succ = l->succ;
  if( l->succ != NULL )
    l->succ->prev = l->prev;
```

```
            prev_list_elem = l;
            l = l->succ;
            free( prev_list_elem );
          }
          else
            l = l->succ;
        }
        else
          l = l->succ;
      }                         // end of while l
      s = s->succ;
    }                         // end of while s
    return start;
  }
```