6-28-1993

# Character recognition of optically blurred textual images using moment invariants

Adam Hanson

# CHARACTER RECOGNITION OF OPTICALLY BLURRED TEXTUAL IMAGES USING MOMENT INVARIANTS

Adam Hanson

B.S. Rochester Institute of Technology
(1991)

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in the Center for
Imaging Science in the College of
Imaging Arts and Sciences of the
Rochester Institute of Technology

June 1993

Signature of Author

_____

Adam Hanson

Accepted by

July 14, 1993

_____

Coordinator, M.S. Degree Program

Center for Imaging Science
Rochester Institute of Technology
Rochester, NY

## Certificate of Approval

---

M.S. DEGREE THESIS

---

The M.S. Degree Thesis of Adam Hanson has
been examined and approved by the thesis
committee as satisfactory for the thesis
requirement for the Master of Science degree

Dr. Roger Easton

Dr. Robert H. Johnston

Mr. Carl Salvaggio

June 28, 1993
Date

## Thesis Release Permission

Rochester Institute of Technology
College of Imaging Arts and Sciences

CHARACTER RECOGNITION OF OPTICALLY BLURRED
TEXTUAL IMAGES USING MOMENT INVARIANTS

I, Adam Hanson, grant permission to the Wallace Memorial Library of R.I.T. to reproduce my thesis in whole or in part. Any reproduction will not be for commercial use or profit.

Date: JUNE 28, 1993

Signature: Adam Hanson

# CHARACTER RECOGNITION OF OPTICALLY BLURRED TEXTUAL IMAGES USING MOMENT INVARIANTS

Adam Hanson

Submitted to the Center for Imaging Science
in partial fulfillment of the requirements
for the Master of Science Degree at the
Rochester Institute of Technology

## ABSTRACT

Statistical moment invariants were used to generate a feature space for classifying images of text characters. The feature vector of a given letter is invariant to changes in scale, position, rotation, and contrast in the image. Test character images were generated by simulated optical blurring. Images were classified by calculating the distance between the feature vector of a given test character and that of each reference character. The test character was identified as the reference character for which the distance between feature vectors is a minimum. Significantly blurred characters were classified correctly using this method.

## ACKNOWLEDGMENTS

I would like to thank Dr. Robert H. Johnston for giving me the opportunity to work on such a fascinating and exciting project as the Dead Sea Scrolls research that motivated this thesis, and for supporting me throughout my graduate career in more ways than I can count.

Thanks to Carl Salvaggio for all his patience and assistance with my programming woes and for his insight and friendship over the years.

Special thanks to God, family, and friends who have been with me from the start; providing inspiration, and helping me stay motivated.

TABLE OF CONTENTS

## TABLE OF FIGURES

# 1.0 - INTRODUCTION & SUMMARY

Computer classification of complex images has become more feasible in recent years due to improvements in computational hardware. Factors such as decreased computation time and the ability to handle large data arrays have contributed to the wider use of computer classification algorithms to classify images for such applications as land use mapping and for recognition of complex shapes. The purpose of this project is to create and test a classifier for recognition of text characters that have been degraded by optical defocus.

The motivation for this project stems from previous research conducted in enhancing degraded images of ancient, textual material as shown in figures 1a and 1b. If a classifier could be developed that could identify degraded text in these images, the analysis of an image would be easier and faster than the lengthy enhancement procedure. This thesis classifies a series of eight English alphabet characters that have undergone simulated optical blurring. The letters are the upper-case 'A', 'E', 'H', 'L', 'M', 'N', 'T', and 'V'. Each character was blurred with an allpass quadratic phase filter. This type of phase filter is often called a chirp filter because the real and imaginary parts of the phase are chirp functions, *i.e.* a sinusoid whose rate of oscillation increases linearly with the spatial coordinate. By adjusting the rate of change of frequency, the

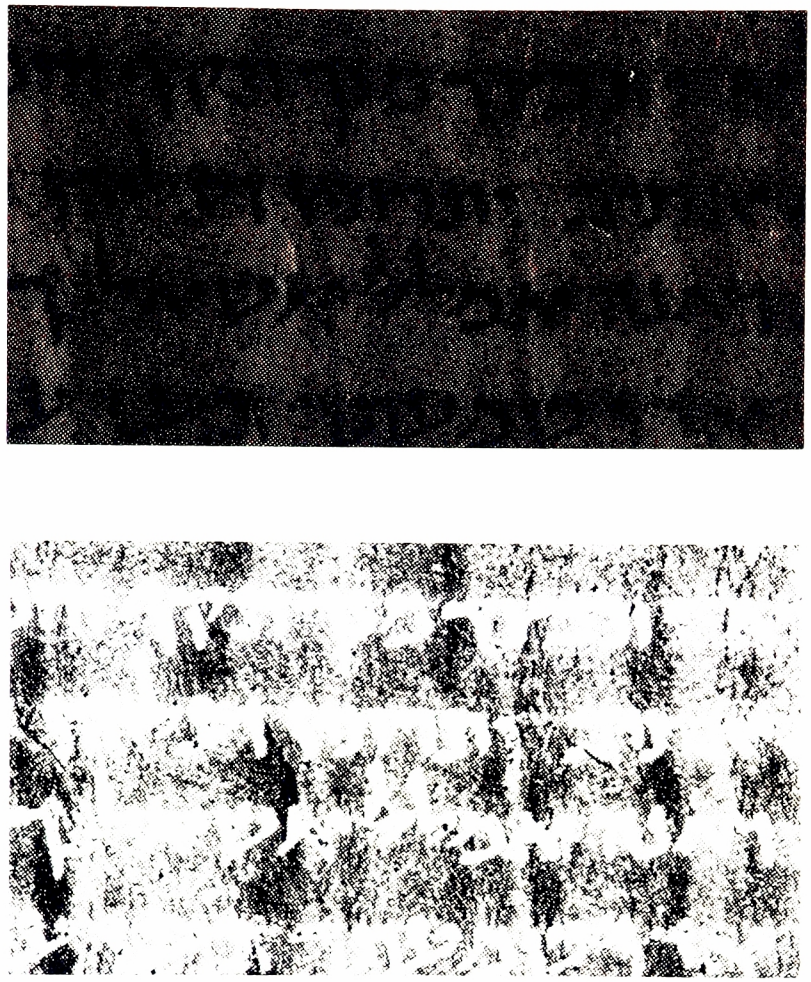


**Figure 1a, 1b. Image of a section from the Targum of Job in original and enhanced version.**

"quantity" of blur could be varied. After blurring by a specific amount, a letter is classified against the ensemble of characters to determine which reference character most closely matched the blurred test character. Successful classification would occur if the computer chose the unblurred test character from the reference set as the closest match. The amount of blur applied to each character would be increased until the classification was incorrect.

The classifier is based on the set of invariant image moments derived by Hu (1969). This set of moments was modified to create a six-dimensional feature vector that is invariant to scale, translation, rotation, and contrast (Maitra, 1979). Thus, each reference character and test image was converted into a six-element vector. A minimum-distance algorithm was used to calculate the Euclidean distance between the test character and each of the reference characters. The test character would be identified as the reference character whose vector most closely matched the vector of the test character.

The results from the classifications demonstrated that characters could be properly identified when subjected to large amounts of blur. The amount of blur after which a given letter was improperly classified was different for each character, but the overall results clearly demonstrate that the moment invariants provide a robust algorithm that can be used to classify degraded objects.

## 2.0 - BACKGROUND

A moment of an image is the sum of its gray values weighted by some power of the coordinate position of each gray value. Invariant moments of an image are a set of numbers that do not change as certain parameters of the image are altered, *e.g.* the position, size, and orientation. Some combinations of the moments can be used to construct a vector for the image which is invariant to changes in scale, translation, and rotation. Thus any differences between vectors of two images (*e.g.* a reference character image and a degraded test image) should depend only on shape, not orientation or location. An image of an alphanumeric character should yield the same vector regardless of the size or location of the character in the image.

When studying degraded images, the noise in the image may render the actual character unrecognizable. Therefore, the identity, proper orientation and size of the character may be unknown. However, when using moment invariants to classify the character, knowledge of the orientation and size is not necessary. Thus character recognition by moment invariants has an advantage over other forms of pattern recognition, such as matched filtering.

Image pattern recognition by moment invariants has been applied to several problems. The pioneering work was done by Ming-Kuei Hu (1962), who demonstrated that a set of moments can

be derived that are invariant to changes in position, scale, and rotation. Hu defined the two-dimensional moments of the $(p+q)^{th}$ order of a density distribution f(x,y) where x and y are spatial coordinates of the distribution. These noncentral moments are:

$$m_{pq} = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} x^p y^q f(x,y)dxdy \quad p,q = 0,1,2... \qquad (1)$$

f(x,y) is assumed to be a piecewise continuous function and therefore is bounded. It is also assumed that p(x,y) has finite support, (*i.e.* $p(x,y) = 0$ for $|x| > x_{max}$ and $|y| > y_{max}$). The central moments are defined from the noncentral moments as

$$\mu_{pq} = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty}(x-\bar{x})^p (y-\bar{y})^q p(x,y)dxdy \quad p,q = 0,1,2... \qquad (2)$$

where the values $\bar{x} = m_{10}/m_{00}$ and $\bar{y} = m_{01}/m_{00}$ define the position of the centroid of the distribution p(x,y). The central moments are invariant under translation; *i.e.* if p(x,y) is transferred to a new set of coordinates:

$$x' = x+\alpha$$

$$y' = y+\beta$$

where $\alpha$ and $\beta$ are constants, the central moments in the new coordinate system will not change.

Hu proved that a specific set of moments are invariant under changes in scale and coordinate rotation. These invariant properties stem from what Hu calls the Fundamental Theorem of Moment Invariants. This theorem states that if the algebraic form of order p has an algebraic invariant,

$$I(a'_{p0},\ldots,a'_{0p}) = \Delta^{\varpi} I(a_{p0},\ldots,a_{0p}),$$

then the moments of order p have the same invariant but with the additional factor $|J|$,

$$I(\mu'_{p0},\ldots,\mu'_{0p}) = |J|\Delta^{\varpi} I(\mu_{p0},\ldots,\mu_{0p})$$

where |J| is the absolute value of the Jacobian of the transform and $\Delta$ is the determinant of the transform. However, Hu states that under some linear transformations, $\Delta$ may not be limited to the determinant of the transformation.

A scale change is described by a transformation of coordinates,

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \alpha & 0 \\ 0 & \alpha \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

where $\alpha$ is a constant. This scale transformation assumes that the change in x and y are equal. Each coefficient of any algebraic form is an invariant,

$$a'_{pq} = \alpha^{p+q} a_{pq} \qquad (3)$$

where $\alpha$ is not the determinant in this case. For moment invariants,

$$\mu'_{pq} = \alpha^{p+q+2}\mu_{pq} \qquad (4)$$

By eliminating $\alpha$ between the zero$^{th}$-order relation, (*i.e.* p = q = 0)

$$\mu' = \alpha^2 \mu$$

and the remaining moments, the following absolute moment invariants result:

$$\frac{\mu'_{pq}}{(\mu')^{1+(p+q)/2}} = \frac{\mu_{pq}}{(\mu)^{1+(p+q)/2}}; \; p+q = 2,3... \qquad (5)$$

where $\mu'_{10} = \mu'_{01} = 0$.

Hu also derived the conditions for invariance under rotation. Given a rotation by $\theta$ about the origin, the output coordinates x', y' are defined by

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \qquad (6)$$

The Jacobian of the transformation is unity and therefore no scale change occurs under the rotation transformation.

$$J = \begin{vmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{vmatrix} = +1$$

By treating the moments as coefficients of an algebraic form

$$(\mu_{p0,} \ldots \mu_{0p})(u,v)^p \qquad (7)$$

under the following transformation,

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} u' \\ v' \end{bmatrix}$$

then the moment invariants can be derived by the following algebraic method. It should be noted that the location of the negative sign in the transform matrix depends on whether a forward or inverse coordinate transform is being calculated.

$$\begin{bmatrix} u' \\ v' \end{bmatrix} = \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} u \\ v \end{pmatrix} \Rightarrow \begin{bmatrix} u \\ v \end{bmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} u' \\ v' \end{pmatrix}$$

Both u,v and u',v' may be further transformed to obtain [U,V], and [U'V']

$$\begin{bmatrix} U \\ V \end{bmatrix} = \frac{1}{2} \begin{bmatrix} 1 & i \\ 1 & -i \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix}, \begin{bmatrix} U' \\ V' \end{bmatrix} = \frac{1}{2} \begin{bmatrix} 1 & i \\ 1 & -i \end{bmatrix} \begin{bmatrix} u' \\ v' \end{bmatrix} \qquad (8)$$

where $i = \sqrt{-1}$. The orthogonal transformation is converted into the two relations between complex quantities

$$U' = Ue^{-i\theta}, V' = Ve^{i\theta} \qquad (9)$$

By substituting (8) and (9) into (7), the following identities result:

$$(I_{p0}, \ldots, I_{0p})(U,V)^p \equiv (\mu_{p0}, \ldots, \mu_{0p})(u,v)^p$$
$$(I_{p0}, \ldots, I_{0p})(U,V)^p \equiv (\mu'_{p0}, \ldots, \mu'_{0p})(u',v')^p$$
$$(I_{p0}, \ldots, I_{0p})(U,V)^p \equiv (I'_{p0}, \ldots, I'_{0p})(Ue^{-i\theta}, Ve^{i\theta})^p$$

where $I_{p0},\ldots,I_{0p}$ and $I'_{p0},\ldots,I'_{0p}$ are the corresponding coefficients after the substitutions into the above equations. From the identity in U and V, the coefficients of the various monomials $U^{p-r}V^r$ must be identical. Therefore,

$$I'_{p0} = e^{ip\theta} I_{p0}; \qquad I'_{p-1,1} = e^{i(p-2)\theta} I_{p-1,1}; \ldots$$
$$I'_{1,p-1} = e^{-i(p-2)\theta} I_{1,p-1}; \qquad I'_{0p} = e^{-ip\theta} I_{0p}.$$

These are (p+1) linearly independent moment invariants under proper orthogonal transformations.

Finally, by combining the orthogonal invariants with the scale invariants of central moments, Hu states that pattern recognition can

be achieved independent of position, size, and orientation. He derives six second- and third-order absolute moment invariants:

$$\phi(1) = \mu_{20} + \mu_{02} \tag{10}$$

$$\phi(2) = (\mu_{20} - \mu_{02})^2 + 4\mu_{11}^2 \tag{11}$$

$$\phi(3) = (\mu_{30} - 3\mu_{12})^2 + (3\mu_{21} - \mu_{03})^2 \tag{12}$$

$$\phi(4) = (\mu_{30} + \mu_{12})^2 + (\mu_{21} + \mu_{03})^2 \tag{13}$$

$$\begin{aligned}\phi(5) = {} & (\mu_{30} - 3\mu_{12})(\mu_{30} + \mu_{12})[(\mu_{30} + \mu_{12})^2 - 3(\mu_{21} + \mu_{03})^2] + \\ & (3\mu_{21} - \mu_{03})(\mu_{21} + \mu_{03})[3(\mu_{30} + \mu_{12})^2 - (\mu_{21} + \mu_{03})^2]\end{aligned} \tag{14}$$

$$\begin{aligned}\phi(6) = {} & (\mu_{20} - \mu_{02})[(\mu_{30} + \mu_{12})^2 - (\mu_{21} + \mu_{03})^2] + \\ & 4\mu_{11}(\mu_{30} + \mu_{12})(\mu_{21} + \mu_{03})\end{aligned} \tag{15}$$

and one skew-orthogonal invariant useful in distinguishing mirror images:

$$\begin{aligned}\phi(7) = {} & (3\mu_{21} - \mu_{03})(\mu_{30} + \mu_{21})[(\mu_{30} + \mu_{21})^2 - 3(\mu_{21} + \mu_{03})^2] \\ & -(\mu_{30} - 3\mu_{12})(\mu_{21} + \mu_{03})[3(\mu_{30} + \mu_{12})^2 - (\mu_{21} + \mu_{30})^2]\end{aligned} \tag{16}$$

This set of seven moment invariants provides a seven-dimensional feature vector that can be employed for character recognition. Research resulting from this work is summarized below.

## 2.1 Optical Implementation of Moment Invariants

The work presented by Casasent and Psaltis (1979) demonstrates the usefulness of moment invariants in pattern recognition by providing an example of the invariant nature of the moment equations and a method for calculating the moment invariants optically. The seven absolute moment invariants are derived in a similar manner to Hu's method. Experiments were performed to demonstrate the validity of the invariance to translation, scale, and rotation. The normalized moment invariants were calculated from a two-dimensional rectangle function, and for translated, rotated, and scaled versions. The results were unchanged by the variation. See appendix A for the results presented in the article as well as diagrams of the various input functions used.

Casasent and Psaltis also demonstrated a method for calculating moment invariants optically. Their system is shown below:

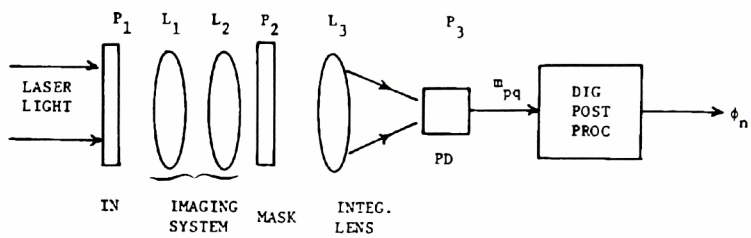



**Figure 2. Optical system to compute normal moments. (Casasent, Psaltis, 1979)**

The input function f(x,y) is placed at $P_1$, and is imaged onto a mask g(x,y) at $P_2$. Therefore the light leaving $P_2$ is f•g. The third lens forms the Fourier transform of the product f•g at $P_3$. By changing the mask at $P_2$ different noncentral moments ($m_{pq}$) can be computed. For g(x,y) = 1, the output was $m_{00}$; for g(x,y) = x or g(x,y) = y, the output at $P_3$ was $m_{10}$ or $m_{01}$ respectively. Thus Casasent and Psaltis demonstrated that all of the ordinary moments of a two-dimensional function could be generated with the proper mask g(x,y). A digital postprocessor was used to calculate the absolute normalized moment invariants from the ordinary moments. The results of this optical implementation of moment invariants proved very successful. The masks created on film were not perfect transmitters or absorbers, thus the $m_{00}$"/$m_{pq}$" notation represents the approximation to the actual theoretical moments calculated. As seen in the table in appendix A, the theoretical and experimental results are very close. This experiment clearly demonstrated that an object could be represented by a vector set of moment invariant equations independent of its size, location in the image, or angle.

## 2.2 A comparison of classifiers for character recognition using moment generated features

Cash and Hatamian (1987) demonstrated the use of moments as a feature vector in a classification process. Cash and Hatamian used a set of central moments to classify six different font styles of

the English alphabet. Three different classifiers were compared to determine which had the best performance with the least computation.

The feature vector used for classification was developed from Hu's original derivation. This vector had ten components, consisting of eight central moments and the two noncentral moments, $m_{01}$ and $m_{10}$. The central moments were used without normalization because they demonstrated similar recognition ability with less computation time. The two noncentral moments were added to the feature vector since they contain height and width information about the character, which may be useful for classification.

Three classifiers were used to measure the similarity between a reference library feature and an input feature: Euclidean distance, cross correlation, and Mahalanobis distance. Euclidean distance simply measures the vector distance between two points in the N-dimensional space:

$$D_E = \sqrt{\sum_{i=1}^{N} (F_{Li} - F_{Ii})^2} \qquad (17)$$

where $F_{Li}$ is the *i*th library feature and $F_{Ii}$ is the *i*th input feature. The library feature vector producing the smallest Euclidean distance when compared with the input feature (*i.e.* the least amount of error) is assigned to the input character. The cross-correlation classifier

takes a very different approach to classifying from the Euclidean model. The class of the library feature producing the largest output when correlated with the input feature vector is assigned to that input character. The normalized cross correlation between two vectors is calculated from:

$$R = \frac{\sum_{i=1}^{N} F_{Li} F_{Ii}}{\sqrt{\sum_{i=1}^{N} F_{Li}^2 \cdot \sum_{i=1}^{N} F_{Ii}^2}} \qquad (18)$$

The Mahalanobis distance is a weighted distance measure between the input feature vector and the mean of the library feature vectors for a particular class. The weight used is the reciprocal of the class feature variance. This distance is calculated from:

$$D_H = \sum_{i=1}^{N} \left( \frac{(\bar{F}_{Li} - F_{Ii})^2}{\sigma_{Li}^2} \right) \qquad (19)$$

The results of the different classifiers indicated that the cross-correlation measure yielded the best results, but at the greatest cost of computation time. It was shown that the ability to correctly classify a feature increased with the number of reference library features used to compare to the input. If twenty library sets were used, each classifier could correctly identify features over 95% of the time. Thus, this research has shown the ability for moment invariants to be used in character recognition.

## 2.3 Use of moment invariants to recognize Arabic text

The research presented by El-Dabi *et al.* (1989) demonstrates how moment invariants can be implemented for character recognition of a complex language such as Arabic. The objective was to create a recognition system that would classify cursive typewritten text. The recognition system calculates moment invariants for each character and uses those moments as a feature for classification. The difficulty with this particular language is that many of the characters are connected or overlap when written, thus it is difficult to recognize individual characters. To solve this problem, characters were separated after classification. On each line of text, portions of words were grouped into columns, and each column was treated as a separate character even though a column may consist of more than one character. The width of each column was as long as a word portion until gaps on both sides of the word portion occurred as in the figure below.

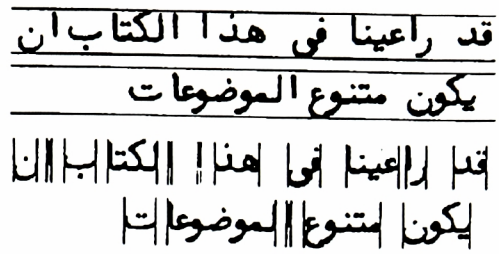



**Figure 3. Method of word segmentation.**
**(El-Dabi, Ramsis, Kamel, 1989)**

The feature space for a given column was calculated and compared to the reference feature space. If a classification could not be made, the next column is added to the previous one and reclassified. This process would be repeated until the end of the word portion was reached. Once a classification was successful, the characters could be separated and identified.

The entire character recognition process consisted of four stages: input, segmentation, recognition, and character separation. The input to this system consisted of a scanned typewritten document stored as an image file. In the segmentation stage, the lines of text were truncated into separate columns of word portions as described above. The moment invariants used to generate the feature space in the character recognition system are based on Hu's fundamental results. To determine the optimal number of moment invariants, multiple classifications were conducted with feature spaces of different sizes. The first classification was run with only two normalized invariant equations as the feature space. The

resulting recognition rate was very poor, as might be expected due to the small number of features. As the dimensionality of the feature space increased, the recognition rates improved. This shows that each added feature more uniquely defines a given character. Thus, a larger feature space increases the ability to separate and identify characters, but it requires more computation time than a smaller one.

The final feature space used for this application consisted of eleven moments, four of these were skew moments to recognize the orientation of the characters. A recognition rate of 94% could be attained. The use of moment invariants provided a solution to the problem of classifying connected characters while providing the flexibility needed to build the optimal feature space.

## 2.4 An application of moment invariants to neurocomputing for pattern recognition

The experiment described by Li, (1991) combines moment invariant signal processing and neurocomputing to create a pattern recognition system that does not require the extensive training necessary for most neural networks, and is less sensitive to noise than many pattern recognition systems. The 6-D feature vector described by Hu is implemented as the input to a Hopfield neural network. Once each feature space is calculated for a character image,

it is converted to a binary signal and applied to the network. Thus, each character will have a unique binary encoded signal.

A Hopfield network consists of a series of nodes with each receiving a binary digit (1 or 0) as input. The nodes are interconnected by a series of positive or negative weights (biases), and thus the output from a given node will be a function of its own input plus the inputs of neighboring nodes and connecting weights. If a series of iterations need be calculated, the output from one series (or layer) of nodes acts as the input to another layer where the calculations are repeated with the revised input. Once a network is trained to recognize a given character, it can be used to test binary signals corrupted by noise. This experiment was described in the article by using an image of the letter 'T'. The binary signal of the letter was used to train the network; the same signal altered by noise was then applied to the network. After performing several iterations on the input signal, the binary output from the 64-node network exactly matched the uncorrupted binary code for the letter 'T'. Thus it was demonstrated that moment invariants can be implemented as the input to a neural network that successfully recognizes characters corrupted by noise.

## 2.5 Expansion of moment invariants to include contrast invariance

Other invariants besides geometrical moment invariants can be implemented in image processing. The remainder of the review summarizes research that focuses on different types of moments used in image analysis. The work conducted by S. Maitra expanded Hu's derivation to create a new set of moments that was invariant to contrast and illumination as well as to shift, scale, and rotation (Maitra, 1979). Maitra confirms the invariance of Hu's moments to shift and rotation, and demonstrates that moments are invariant to scale with the following normalization:

$$\eta_{pq} = \frac{\mu_{pq}}{\mu_{00}^{\gamma}}$$

where $\gamma = (p+q)/2 + 1$ and $\mu_{pq}$ is the $(p+q)^{th}$ order central moment. However, the moments are not invariant to contrast changes. Maitra considers two images $f_1(x,y)$ and $f_2(x',y')$ with varying contrast. Under the previously developed invariants, the two images should remain unchanged under the following transformation:

$$f_1(x, y) = kf_2(x', y') \tag{20}$$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = c \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} a \\ b \end{bmatrix} \tag{21}$$

where $\theta$ is an arbitrary rotation, (a,b) the translation, k the change in contrast, and c an arbitrary scale change. Without normalization of Hu's seven equations (denoted here as $\phi(1)$-$\phi(7)$), Maitra states that under the above transformation, $\phi_1(1)$-$\phi_1(7)$ computed on $f_1(x,y)$ and $\phi_2(1)$-$\phi_2(7)$ computed on $f_2(x',y')$ are related as:

$$\phi_1(1) = \frac{k}{c^4}\phi_2(1) \tag{22}$$

$$\phi_1(2) = \frac{k^2}{c^8}\phi_2(2) \tag{23}$$

$$\phi_1(3) = \frac{k^2}{c^{10}}\phi_2(3) \tag{24}$$

$$\phi_1(4) = \frac{k^2}{c^{10}}\phi_2(4) \tag{25}$$

$$\phi_1(5) = \frac{k^4}{c^{20}}\phi_2(5) \tag{26}$$

$$\phi_1(6) = \frac{k^3}{c^{14}}\phi_2(6) \tag{27}$$

$$\phi_1(7) = \frac{k^4}{c^{20}}\phi_2(7) \tag{28}$$

Maitra notes that normalization of $\phi(1)$-$\phi(7)$ removes the variability due to scale changes, but does not account for the variability of the contrast *k*. Thus Maitra provides (without mathematical derivation) the following set of six moments that are invariant under the general

transforms described above, and are also invariant under changes in contrast (*i.e.* variability due to $k$ is removed):

$$\beta(1) = \frac{\sqrt{\phi(2)}}{\phi(1)} \tag{29}$$

$$\beta(2) = \frac{\phi(3) \bullet \mu_{00}}{\phi(2) \bullet \phi(1)} \tag{30}$$

$$\beta(3) = \frac{\phi(4)}{\phi(3)} \tag{31}$$

$$\beta(4) = \frac{\sqrt{\phi(5)}}{\phi(4)} \tag{32}$$

$$\beta(5) = \frac{\phi(6)}{\phi(4) \bullet \phi(1)} \tag{33}$$

$$\beta(6) = \frac{\phi(7)}{\phi(5)} \tag{34}$$

An original digitized image and a transformed copy of the same image was used for experimental testing. The transforms performed on the copy include changes in scale, shift, rotation, and contrast. The results demonstrated invariance to all transformations.

## 2.6 An example of calculating a moment invariant feature vector

Consider a binary image consisting of a pair of delta functions located about the center of a 64 x 64 image and separated by one pixel. The pixels corresponding to the delta functions are the only two pixels that have a non-zero value (digital count value = 255) and are located at x, y coordinates (30, 31) and (32, 31). The first step in calculating the invariant feature vector consists of using equation (1) to find the mean of x and y where $\bar{y} = 31$, and $\bar{y} = m_{01} / m_{00}$. For this example, $\bar{x} = 31$. These mean values are used in calculating the central moment invariants according to equation (2). Substituting the mean values, the central moments are: $\mu_{00} = 510$ and $\mu_{20} = 510$, the rest of the central moments are zero. The central moments are normalized by equation (5) which can be written as:

$$\eta_{pq} = \frac{\mu_{pq}}{\mu_{00}^{\gamma}}$$

where $\gamma = 1+(p+q)/2$. Each value of $\mu$ in equations 10-16 is normalized according to the above equation. The values resulting from equations 10-16, denoted as $\phi(1) - \phi(7)$, result in $\phi(1) = 0.001961$ and $\phi(2) = 3.84 \cdot 10^{-6}$. All remaining components in the feature vector are zero. Finally, the feature vector $\phi(1) - \phi(7)$ is transformed according to equations (29)-(34) to create a vector that

is invariant to all transformations: shift, scale, rotation, and contrast. The final vector has components of $\beta(1) = 1$ and all remaining components are zero.

If invariance exists, a different image of two delta functions will produce the same feature vector. The same calculations were performed on a second image of two delta functions having a value of 1, located at (2,0) and (2,4). This is a scaled, translated, and rotated version of the previous image, it also has a different brightness (contrast). In this case, $\mu_{00} = 2$ and all remaining central moments are zero. Before the final transformation, $\phi(1) = 2$ and $\phi(2) = 4$, all remaining components are zero. After the final transformation, $\beta(1) = 1$, and the other vector components are zero. Having obtained the same result as the previous case, the invariance properties of the feature vector are demonstrated.

## 3.0 - APPROACH

The biggest task involved with the experimental portion of this project consisted of writing the necessary computer code to generate the results. Two separate programs were written, one that created the blurred test images, and one that classified the test images. All code was written in C language using a UNIX workstation. The images were downloaded to a Macintosh and displayed using Adobe Photoshop software. Numerical distances between feature vectors were also sent to a Macintosh from the UNIX workstation and tabulated in Microsoft Excel™. The reference images were generated in Adobe Photoshop™ as were images used to test the invariance properties of the feature vector.

The first program computed several functions that allowed an input image (1 byte per pixel) to be digitally filtered and written out in double precision floating point format (8 bytes per pixel). This filtered image would then be classified. Roundoff error of the digital data became a serious problem. To preserve as much precision as possible, the filtered images were saved in double precision format instead of byte format. By preserving full floating point information, computation error due to roundoff from decimal to integer notation was minimized. The following flow chart shows the steps to create the output test image.

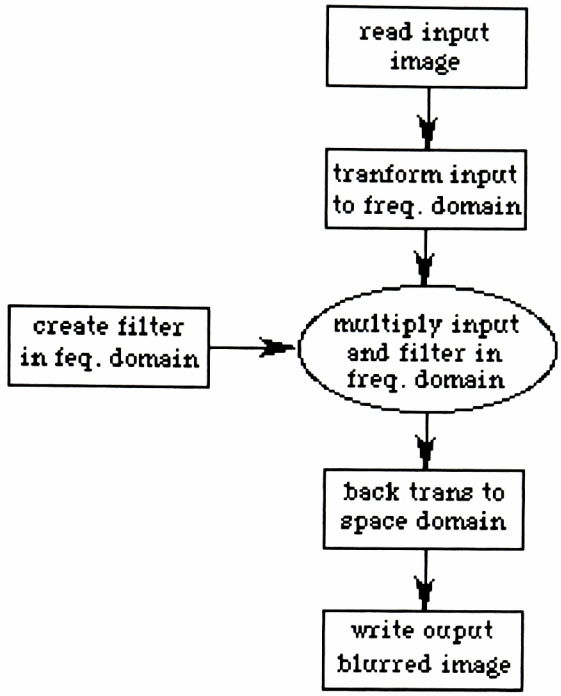



**Figure 4. Flow chart for program to create blurred test image.**

Each step in the flow chart represents a separate routine that was called by the main program. The quadratic-phase filter is implemented in the frequency domain, thus the Fourier-spectrum of the input image must be computed. The output spectrum is computed by multiplying the input image spectrum with the frequency-domain representation of blur function. The inverse Fourier transform of this output spectrum generates the optically blurred image which is stored using double precision.

## 3.1 Description of the quadratic-phase filter and frequency-domain filtering

Optical defocus in a coherent imaging system may be modeled as a convolution of the object with a filter which has quadratic-phase impulse response. The frequency-domain representation of the filter is easy to generate, therefore it does not have to be transformed before being applied to the input image. The filter has constant magnitude and quadratic phase. The general form for the frequency-domain two dimensional quadratic phase filter is:

$$H[\varepsilon, \eta] = e^{i\pi\alpha^2(\varepsilon^2+\eta^2)} \tag{35}$$

where $\varepsilon$ and $\eta$ are frequency coordinates corresponding to spatial coordinates x and y respectively, and $\alpha$ is the rate-of-change of phase, also called the chirp rate. If $\alpha$ is increased, the space-domain representation becomes wider and the frequency-domain filter becomes narrower (figure 14). The corresponding impulse response in the space domain has the form:

$$h[x, y] = \frac{1}{|\alpha|} e^{+i\frac{\pi}{4}} e^{-i\pi\frac{x^2+y^2}{\alpha^2}}$$

The space-domain representation of the transfer function (*i.e.* the point-spread function) is also a constant-magnitude quadratic-phase filter. However, the scaling theorem demonstrates that the point-

spread function is expanded for large $\alpha$, and contracted for small $\alpha$; this is opposite of the frequency-domain case. The units of $\alpha$ are length; in an optical system, the units are usually expressed in millimeters or microns, in the digital system, $\alpha$ is expressed in pixels.

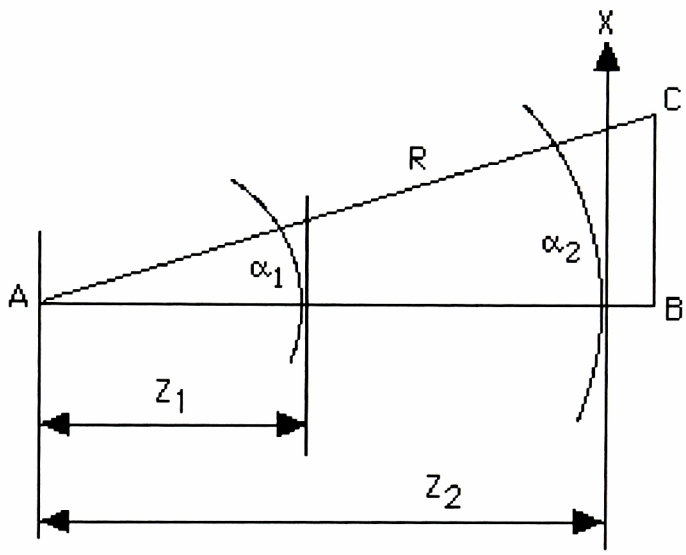



**Figure 5. Diagram depicting the quadratic-phase of a coherent optical system.**

A spherical wavefront traveling from point A to an image plane along B-C will be in phase at A. The wavefront expands as it travels, causing a phase change since all points on the wavefront are no longer in the same plane. The phase difference between the wavefront traveling along Z and the wavefront traveling along R is approximately a quadratic function of X due to the spherical nature of the wavefront. At a distance $Z_1$, the wavefront is out of phase and will have a certain phase rate, $\alpha_1$. At a distance $Z_2$, the wavefront

will have less phase difference as a function of X, thus $\alpha 2 < \alpha 1$. If the quadratic phase of the wavefront was examined as real and imaginary parts, the rate-of-change of the chirp function would be evident. The following chirp functions are examples of the space-domain and corresponding frequency-domain for the wavefront at $\alpha 1$ and $\alpha 2$.

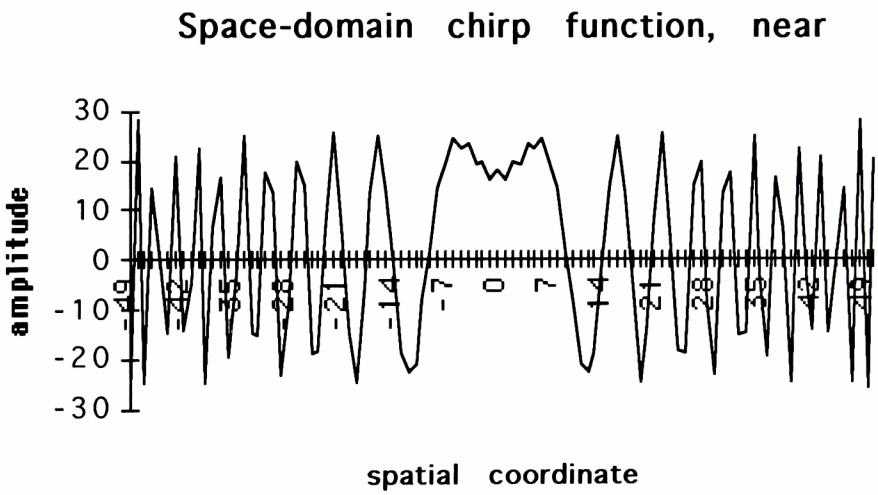



Figure 6a. Space-domain chirp function for $\alpha 1$.

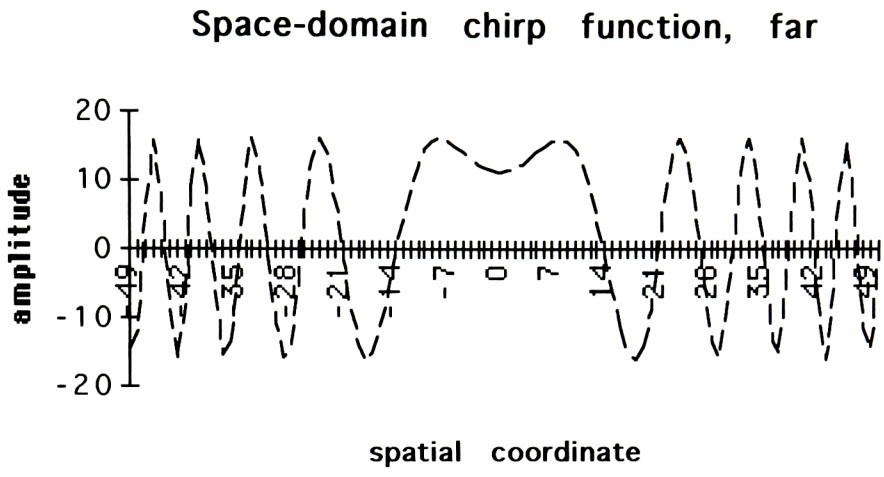



Figure 6b. Space-domain chirp function for $\alpha_2$.

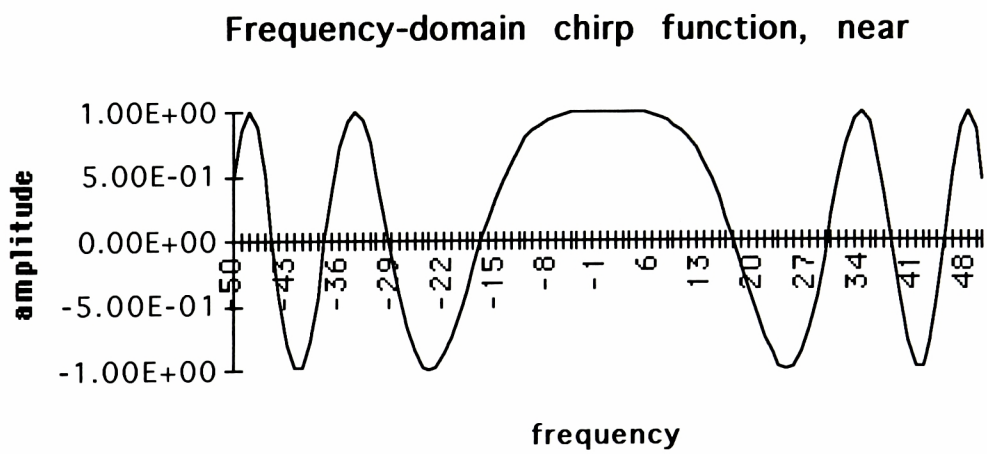



Figure 7a. Frequency-domain chirp function for $\alpha_1$.

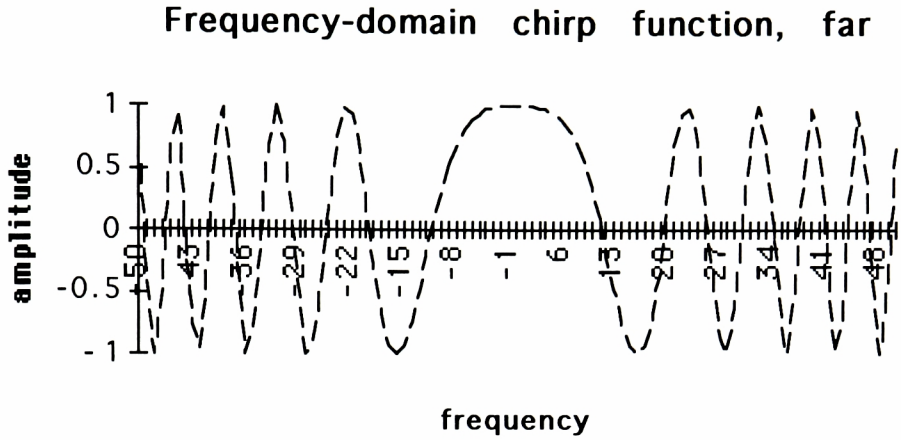



**Figure 7b. Frequency-domain chirp function for $\alpha_2$.**

These curves illustrate the scaling theorem already discussed. If $\alpha_1$ is large, the frequency-domain representation is narrow, as its width is proportional to $1/\alpha_1$. The amount of defocus is determined by $\alpha$, and worsens as $\alpha$ is increased. Thus, the amount of defocus observed at $Z_2$ is greater than at $Z_1$.

The Fourier transform of the input character and the phase filter are both complex functions, and the filtered spectrum is the complex product. The complex exponential filter equation can easily be rewritten as real and imaginary parts by recalling that $e^{i\theta} = \cos(\theta) + i\sin(\theta)$, where $i = \sqrt{-1}$. The real and imaginary parts of the frequency-domain representation phase filter have the form:

$$H_{real}[\varepsilon, \eta] = \cos(\pi\alpha^2(\varepsilon^2 + \eta^2)) \quad (36)$$

$$H_{imag}[\varepsilon, \eta] = \sin(\pi\alpha^2(\varepsilon^2 + \eta^2)) \quad (37)$$

The oscillation of the chirp increases quadratically with $\varepsilon$ and $\eta$. The rate at which the oscillation increases is determined by the chirp rate $\alpha$. The equation for the filtering in the frequency domain has the form:

$$(F_{real}[\varepsilon,\eta] + iF_{imag}[\varepsilon,\eta])(H_{real}[\varepsilon,\eta] + iH_{imag}[\varepsilon,\eta]) \qquad (38)$$

where $F_{real}[\varepsilon,\eta]$ and $F_{imag}[\varepsilon,\eta]$ represent the real and imaginary parts of the Fourier transform of the input image, respectively. This multiplication is performed for each frequency coordinate $\varepsilon$ and $\eta$.

In an optical system the filtered image is the squared magnitude, $|g(x,y)|^2$. This function was calculated by inverse-transforming the result from (38), then squaring the magnitude. The amount of blur applied to a character was increased by incrementing $\alpha$ in units of pixels from $\alpha = 0$ to a maximum of $\alpha = 9$. A series of test images for each character was generated in this manner to be tested by the classifier. Test images with increasing amounts of blur were created until the classifier failed to properly classify an image with a certain phase rate. This method of testing allowed a cutoff phase rate to be established; the cutoff phase rate corresponded to the amount of blur applied to a given test image that caused incorrect classification.

## 3.2 Description of the classifier

The classification process consisted of a comparison of a test character with the set of eight reference characters. The algorithm classified the test character as the most similar reference character. "Most similar" is defined as the closest pair in the 6-dimensional feature space. The reference and test images had to be converted to a representation that the computer could easily manipulate in statistical calculations. Equations 10-16 developed by Hu are a set of moment invariants that describe a seven-dimensional feature space. This feature space was initially implemented in this project to generate the feature vectors. However, this set of moments does not take into account variations in contrast. In other words, if a test character has different gray levels after blurring than the corresponding reference character, the classifier may improperly identify the test character based on the difference in gray level. Since it was desirable to have this variation in contrast removed, equations 29-34 developed by Maitra were used in place of Hu's equations. These equations describe a six-dimensional feature space that is invariant to changes in scale, translation, rotation, and contrast. It should be noted that since the computer is calculating a digital approximation of the continuous moments, there is some minor variability in the invariant moments. This variability is generally small enough to be negligible.

The following diagram shows how the classifier works once the test images are generated.

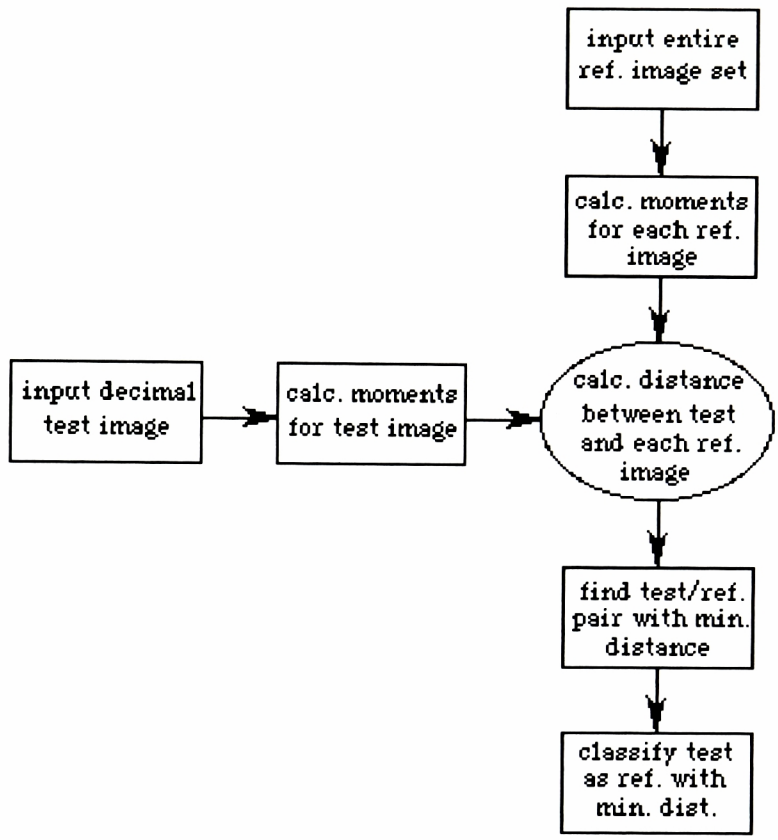



**Figure 8. Flow chart showing classification process.**

The six-dimensional feature vector is first calculated for each reference image. The feature space for the test image is then calculated. The results from these calculations consist of eight 6-D vectors, that are classified.

The classifier uses a simple Euclidean distance measure to determine the distance between the test character and reference

character. The calculation for squared distance, D, has the following form:

$$D = \sum_{i=1}^{6} (\beta(i)_{test} - \beta(i)_{ref})^2$$

where $\beta(i)_{test}$ is the $i^{th}$ component of the test vector and $\beta(i)_{ref}$ is the $i^{th}$ component of the reference vector. The absolute distance is found by calculating the square root of D; however, the smallest distance would remain smallest whether it was squared or absolute. Thus, for this experiment, the squared distance measure was used instead of the absolute distance. The main advantage of this procedure was increased computation speed.

Once the squared distances have been calculated, the program classifies the test character as the reference character that corresponds to the shortest distance. The distance measure can be thought of as a measure of similarity between two character images. From the above distance equation, one can see that if two characters were identical, they would have the same feature vectors and the distance between them would be zero. As two characters become less similar, the feature vectors would change, and the distance between them increases. Thus, a smaller distance between two feature vectors indicates greater similarity between the two corresponding images. The reference character that is most similar

to the test character will have the smallest distance measure, and is therefore selected as the classification for that test character.

This experiment originally proposed to test the classifier using a reference set of four characters: capital 'A', 'E', 'H', and 'V'. However, a larger set was used in the actual experiment including upper-case letters: 'A', 'E', 'H', 'L', 'M', 'N', 'T', and 'V'. This set of eight character set provided a larger data set that helped demonstrate the behavior of the classifier. Each character was independently tested against the entire reference set with increased blur in the test character for each classification. Classifications for a given character were recorded until the first failure occurred, at which point classification for that character ceased. To tabulate the data, the program saved files containing distance information for a given test character with a given blur factor. This data was then analyzed and compiled in a graphical format. The following section describes the results from this experiment.

# 4.0 - RESULTS

It has been claimed throughout this report that the moments used to generate the feature vectors of the images are invariant under a series of transformations. The importance of this invariance property is derived from the desire that the classifier only use the shape of a character in the decision-making process. Ideally, the letter 'A' would be recognized as the letter 'A' regardless of its location, orientation, size, or gray value in the image. To validate this claim, a series of tests were run on the image of a character to determine if the moment-generating function was in fact invariant to scale, translation, rotation, and contrast transformations. Four test images of the capital letter 'A' were generated using Adobe Photoshop™, each with different parameter changes. The translation test image was shifted by 10 pixels, the rotated image was rotated by 15 degrees, the scaled image was changed from 36 to 24 point. Each test image was then classified against all eight reference characters. The following pages contain the graphs of the invariance tests. The data for these graphs is tabulated in appendix B.

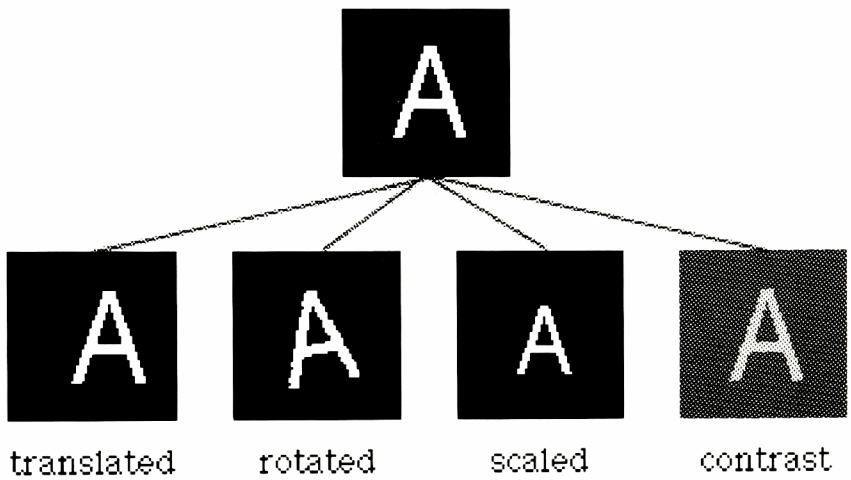



**Figure 9. Test images for invariance validation.**

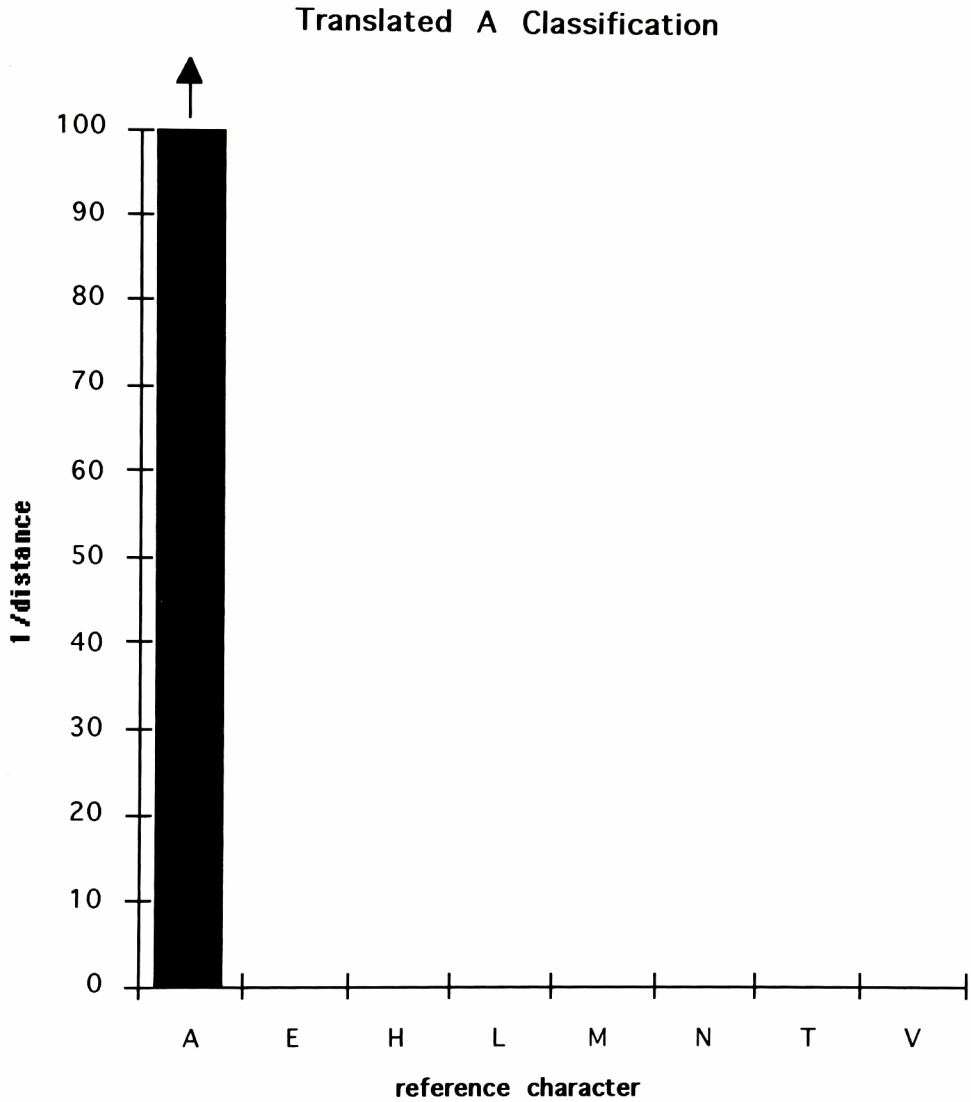



**Figure 10. Graph for translation invariance test.**

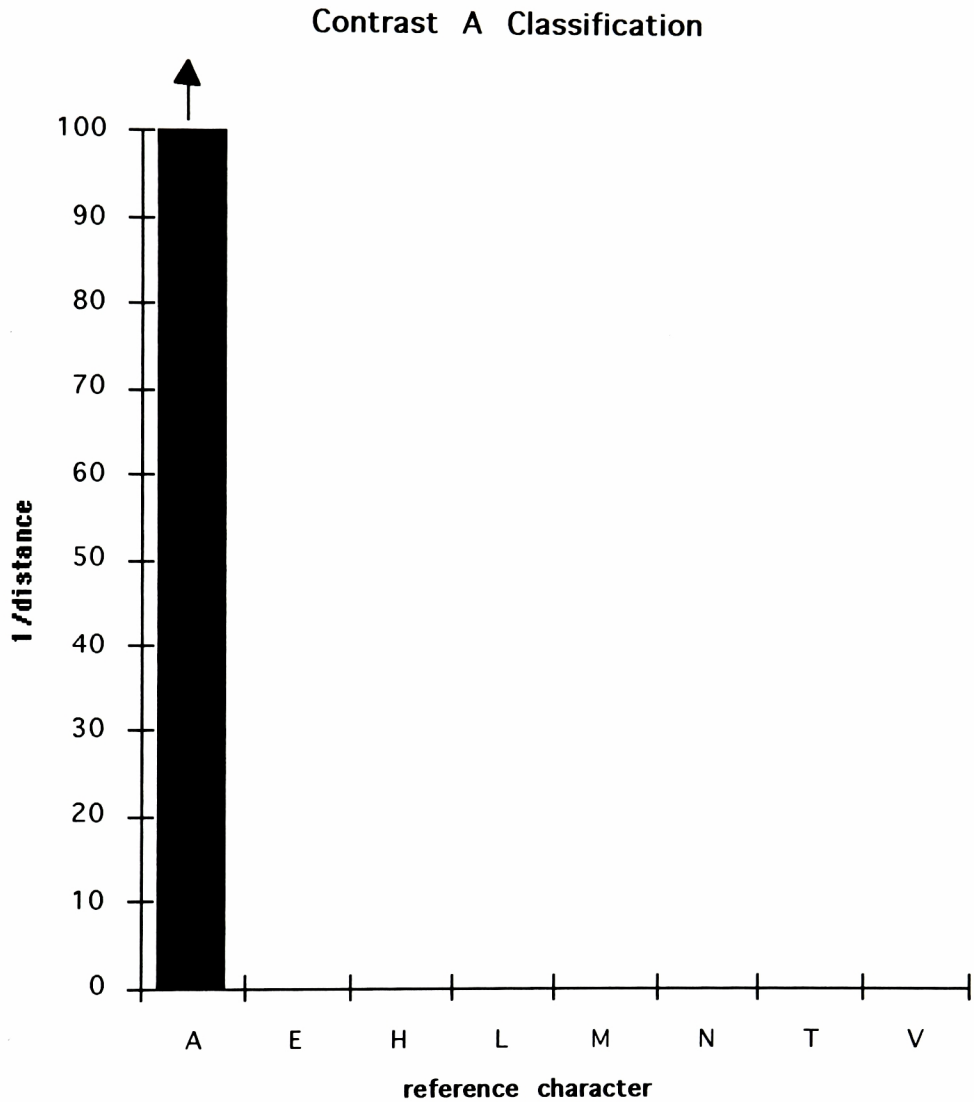



**Figure 11. Graph for contrast invariance test.**

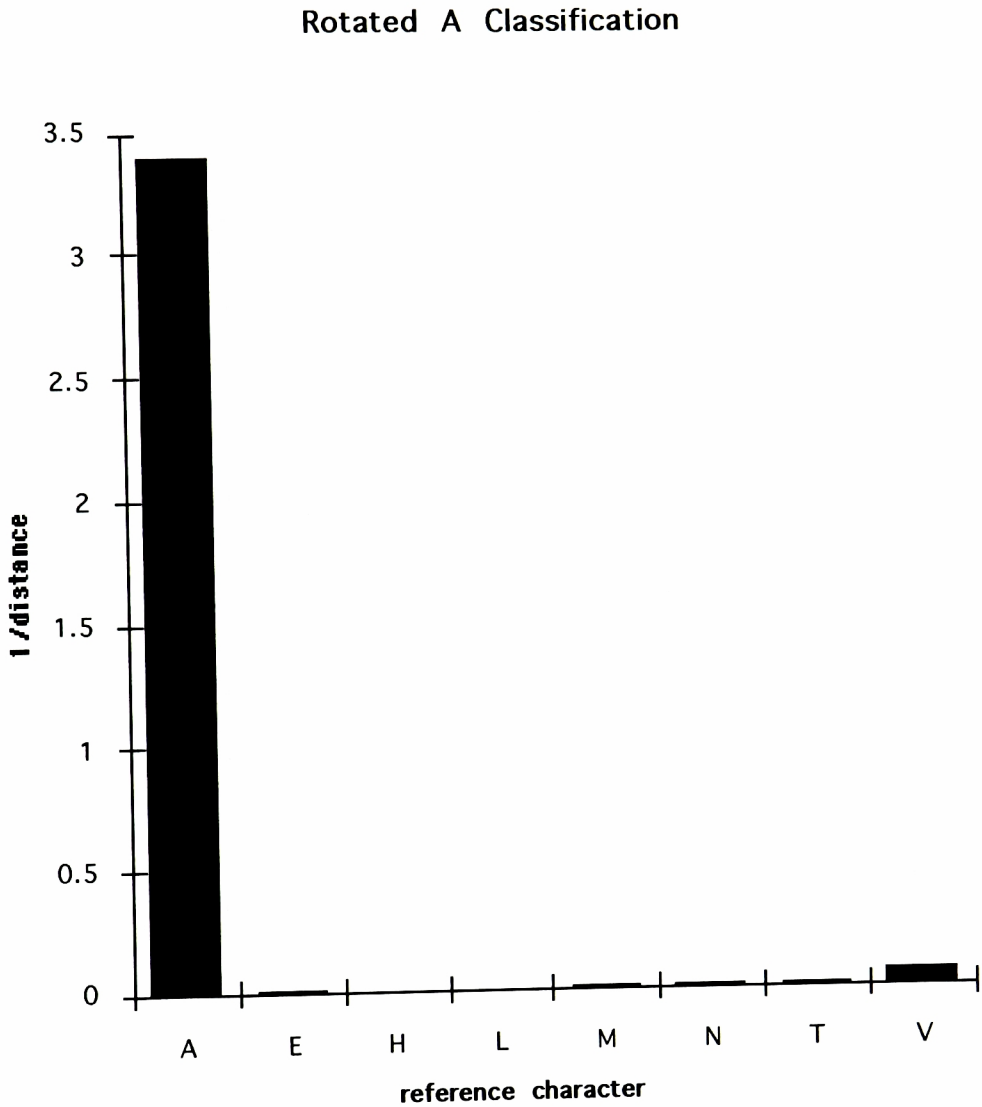



Figure 12. Graph for rotation invariance test.

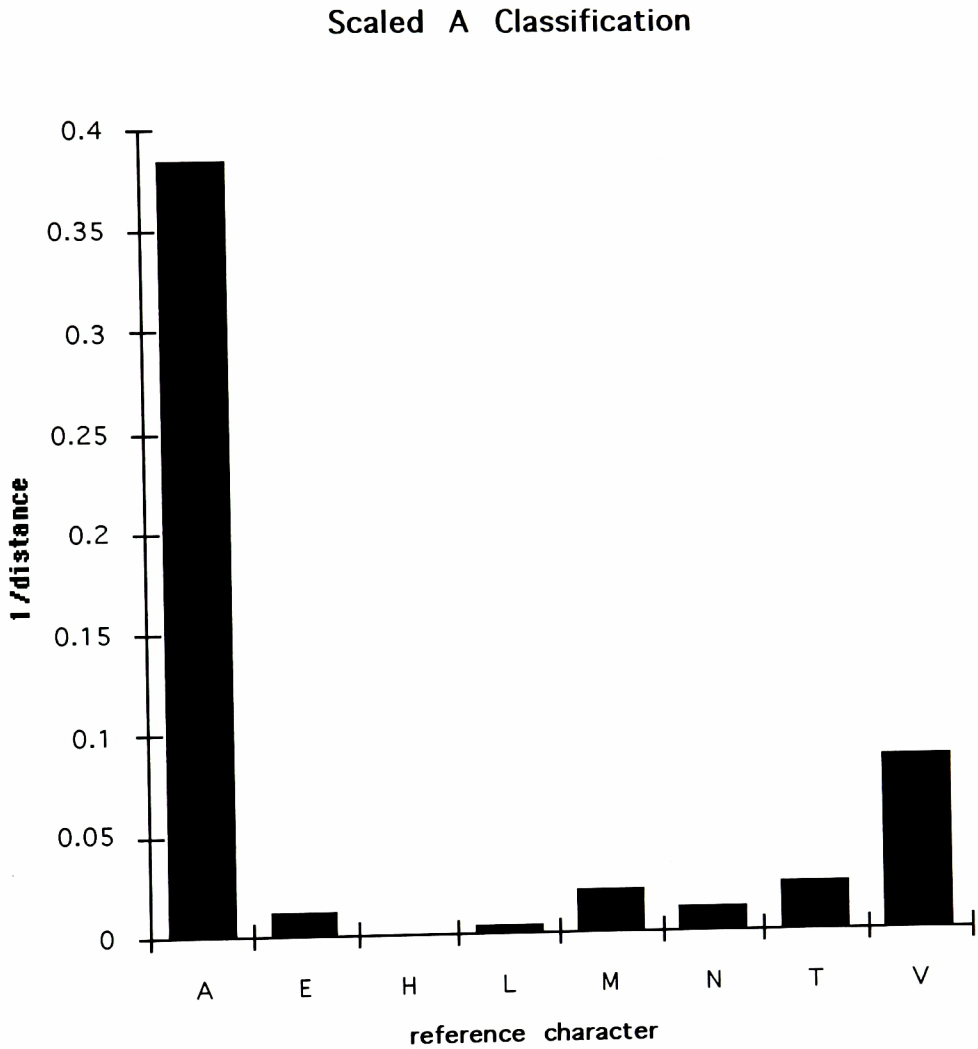



Figure 13. Graph for scale invariance test.

To make the graphs more visually intuitive, the inverse of the distance to each reference character was plotted. Thus, when interpreting the graphs, large bars indicate high similarity between test and reference characters. If the moments were truly invariant, then the feature vectors of each test character and the reference letter 'A' would be the same, the distance between them would be zero and the inverse of the distance would be infinite. This is observed for the translation and contrast transformation tests. The calculated distance between the test letter 'A' and reference 'A' is zero for both cases.

Though the tests for scale and rotation invariance may appear less convincing, there are two important facts to be noted. First, the calculation of the moments is a discrete approximation of the continuous moments, thus some error occurs due to the digital approximation. Secondly, the images have relatively low spatial resolution (64 X 64 pixels). The scaled or rotated character does not keep the exact same shape as the original due to resampling artifacts such as staircasing (figure 9) that occurs during the transformation; this also causes error to occur in calculation of the feature vector of the test image. This problem does not occur in the translation and contrast transformations since the pixels that form the character itself are not altered. Despite this error, the test character for the rotated and scaled cases is still easily matched to the correct reference character.

After having demonstrated invariance under the stated transformations, it is safe to conclude that each feature vector is a function of the character shape only (to within the digital resolution). The distance calculated between a test character and a given reference character depends on the similarity between the two characters. If a test character has been subjected to little degradation by the phase filter, the distance to the matching reference character will be small. As the test character is subjected to more blurring, its shape becomes more distorted and the distance to its corresponding reference character increases. At some point, when the test character is distorted enough, it will have a shorter distance measure to some other reference character than to the correct reference, and an incorrect classification results.

Figure 14 is an image sheet of the character images used in this experiment. The phase filters and degraded test characters are shown with the original images, the first test image below the line in each column corresponds to the cuttoff phase rate where the classifier failed for that character. All characters above the line were correctly identified. Figures 15-22 are and the graphical representations of the distance classification data. To plot the data in a visually intuitive manner, the distance data was transformed to the log of the inverse distance with an added bias, and plotted as "relative classification weights". The distance data was transformed in three steps: (1) the inverse of the distance data was found, (2) the

base 10 logarithm of the inverse data was calculated, and (3) a constant bias was added to the log data to make all data points positive. As before, large magnitudes of the relative classification weights indicate high similarity and small distance between test and reference characters. Smaller bars indicate low similarity, and a larger distance between test and reference characters. The data for the graphs can be found in appendix B.

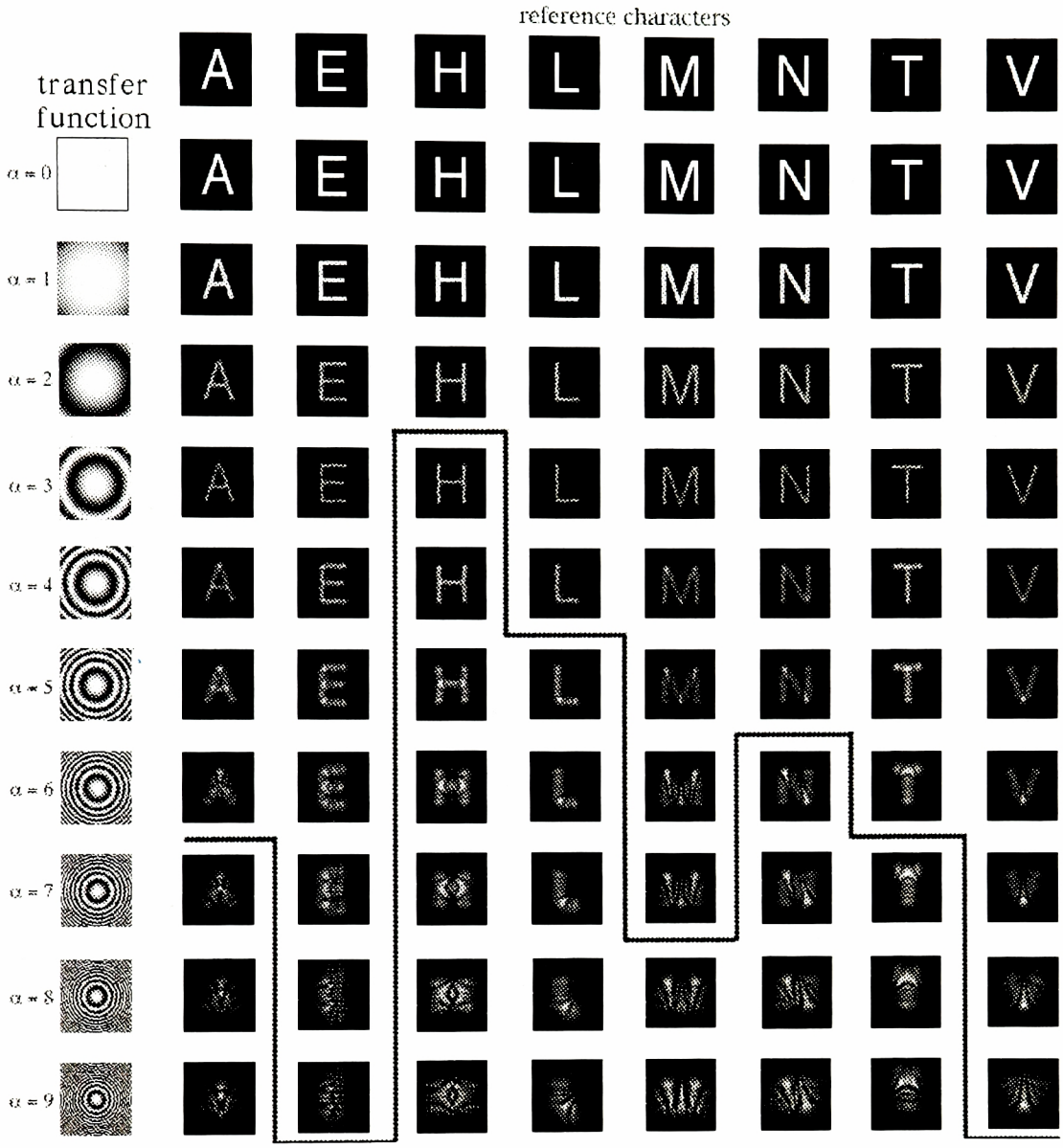



**Figure 14. Reference images, filters, and resulting test images from filtering**

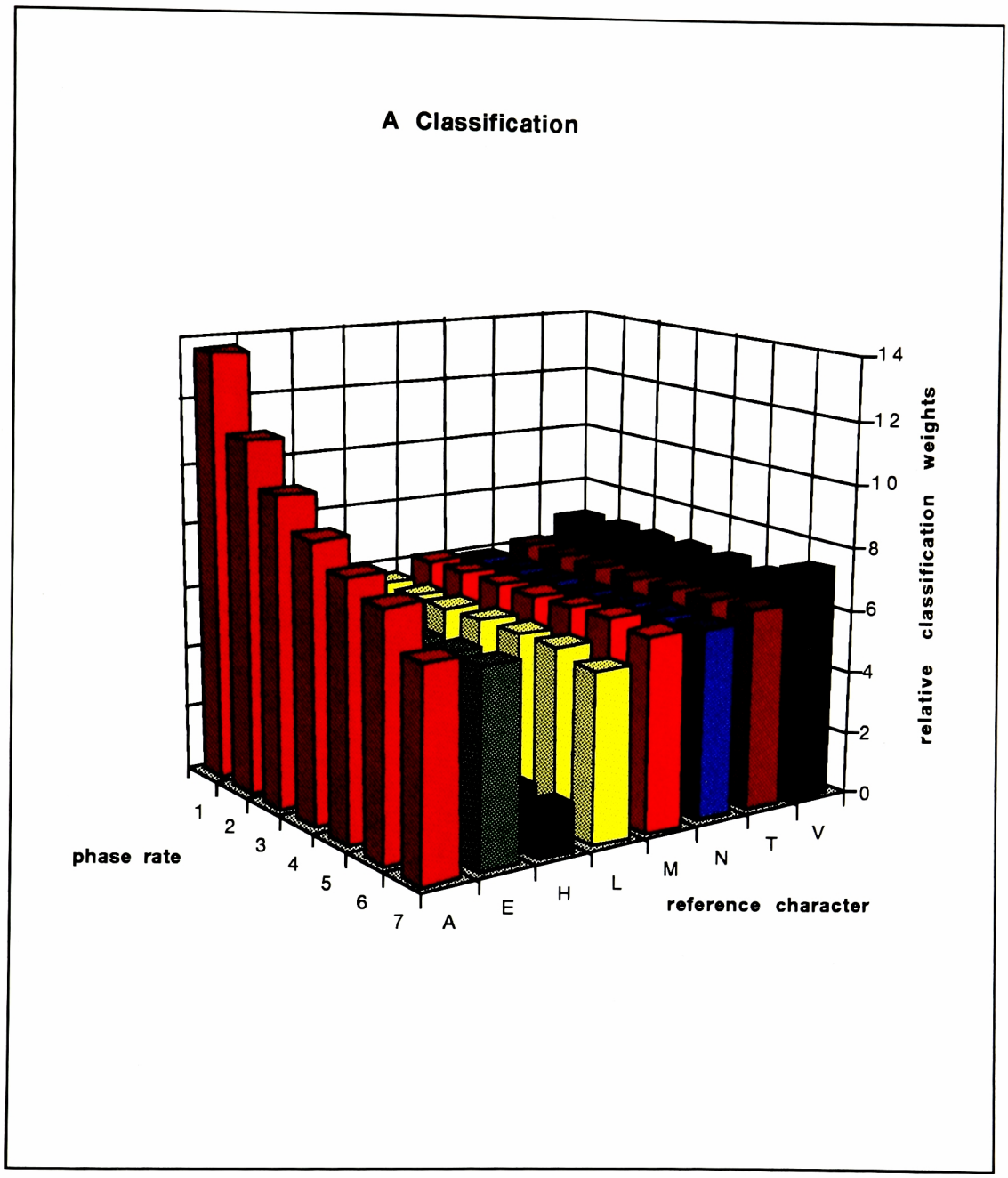



**Figure 15. Classification graph for the letter 'A'.**

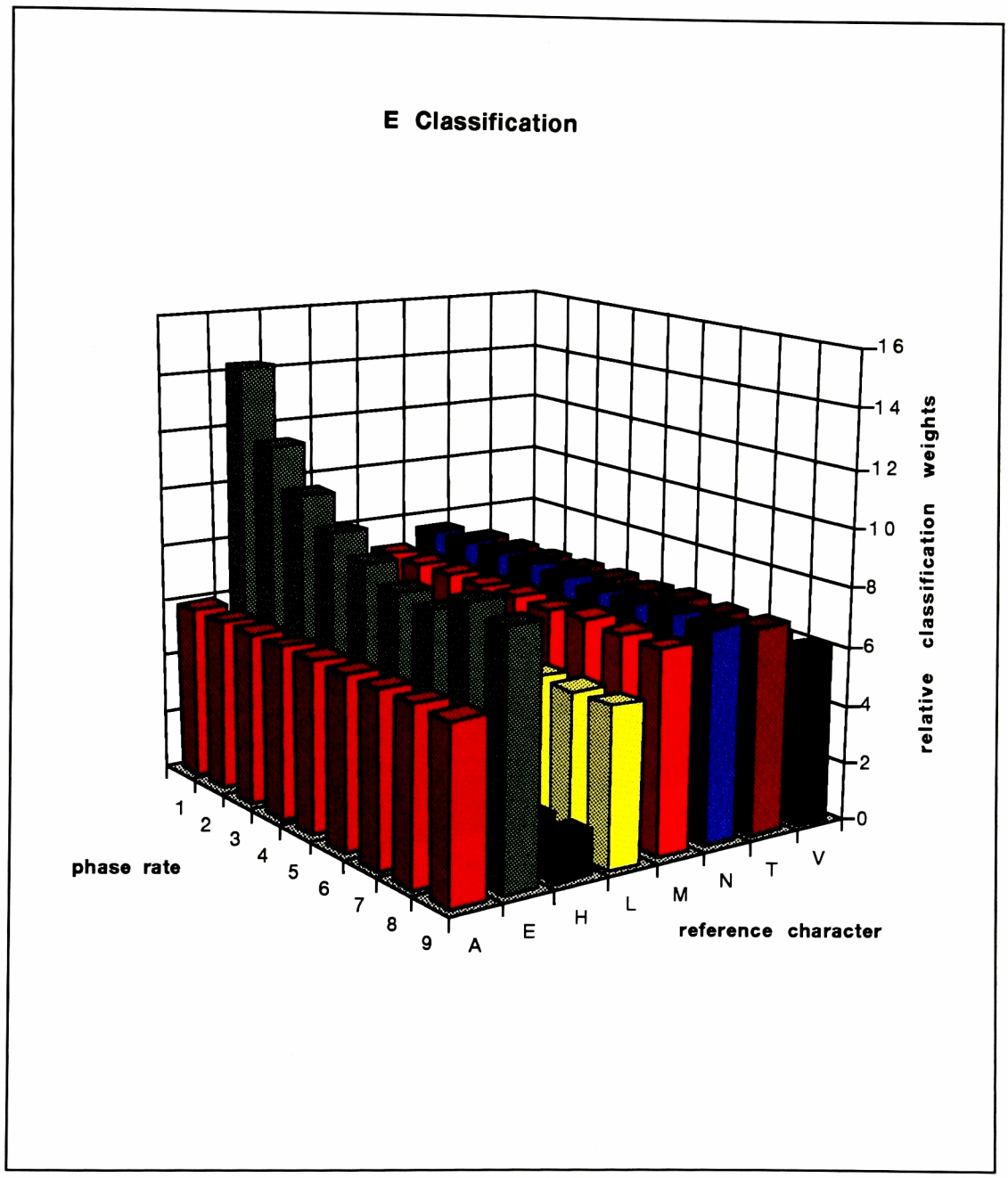



**Figure 16. Classification graph for the letter 'E'.**

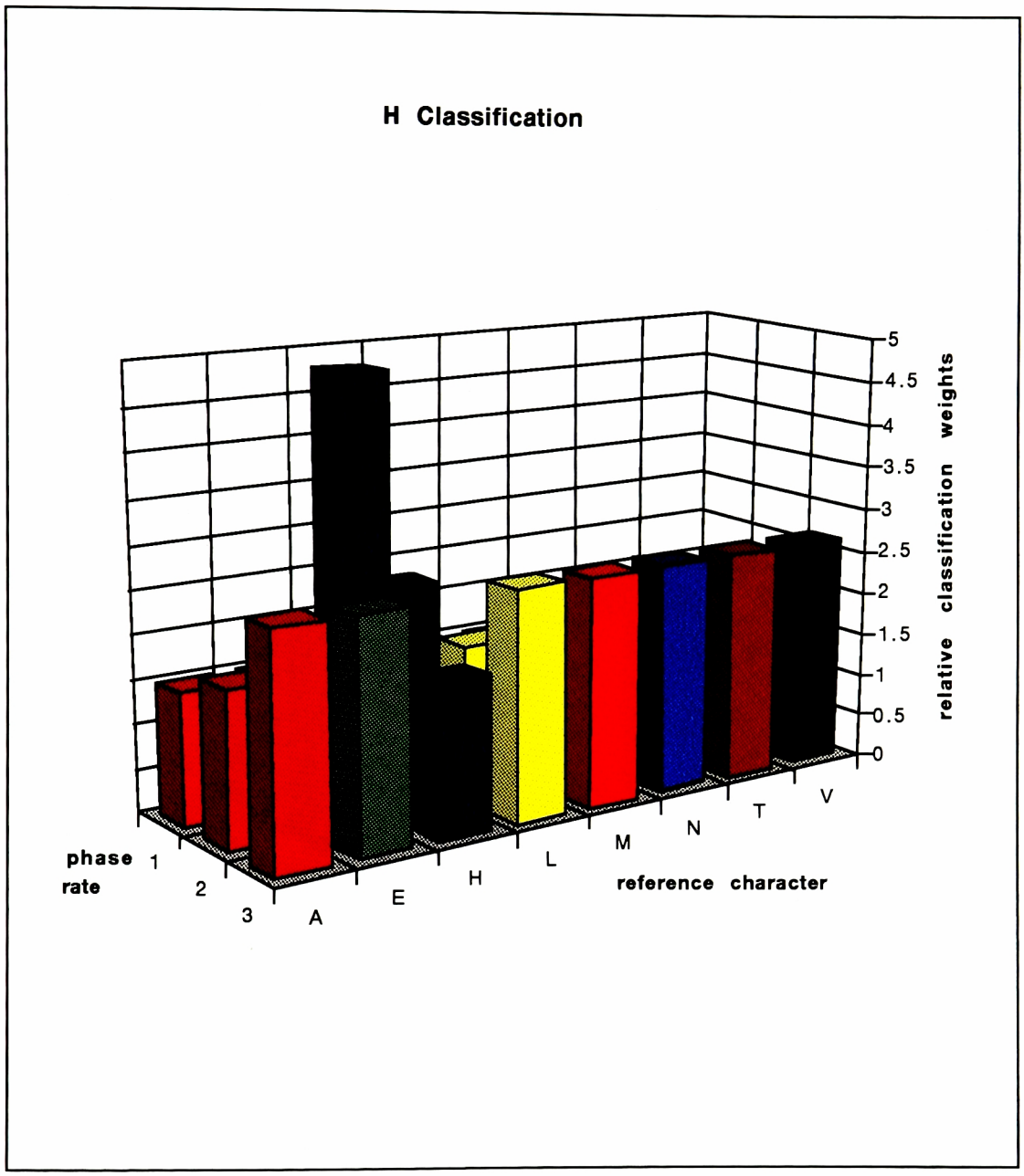



**Figure 17. Classification graph for the letter 'H'.**

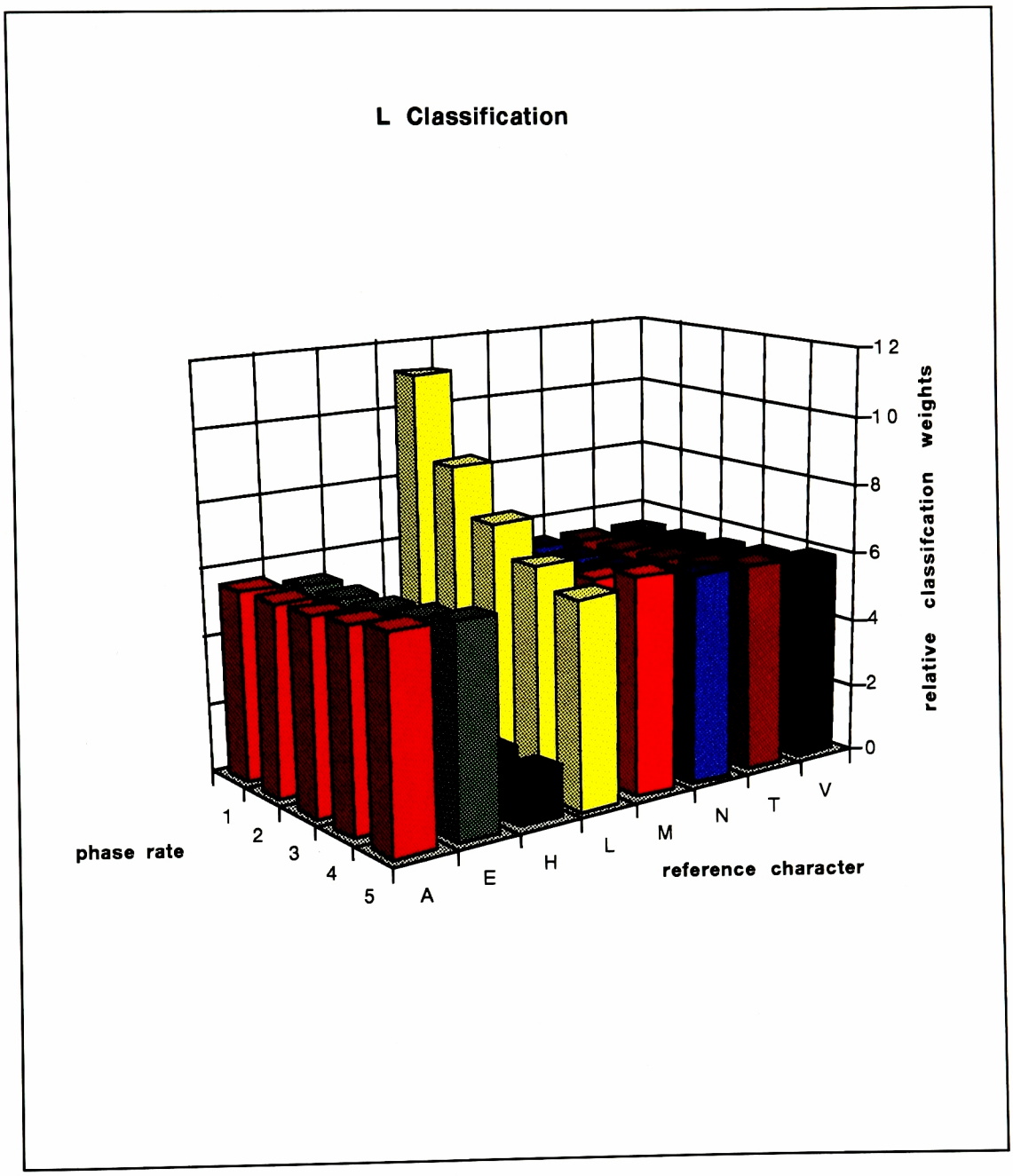



**Figure 18. Classification graph for the letter 'L'.**

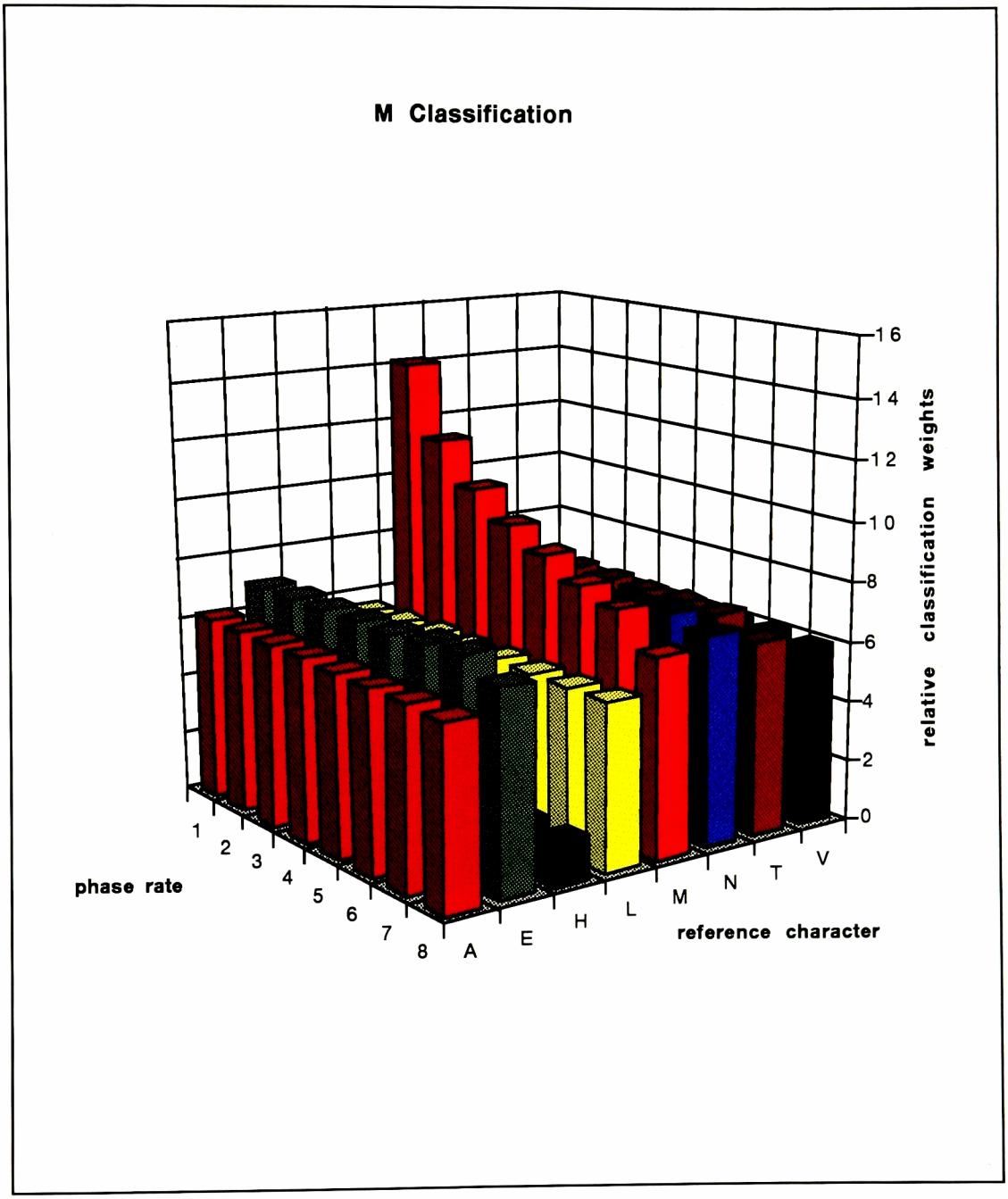



**Figure 19. Classification graph for the letter 'M'.**

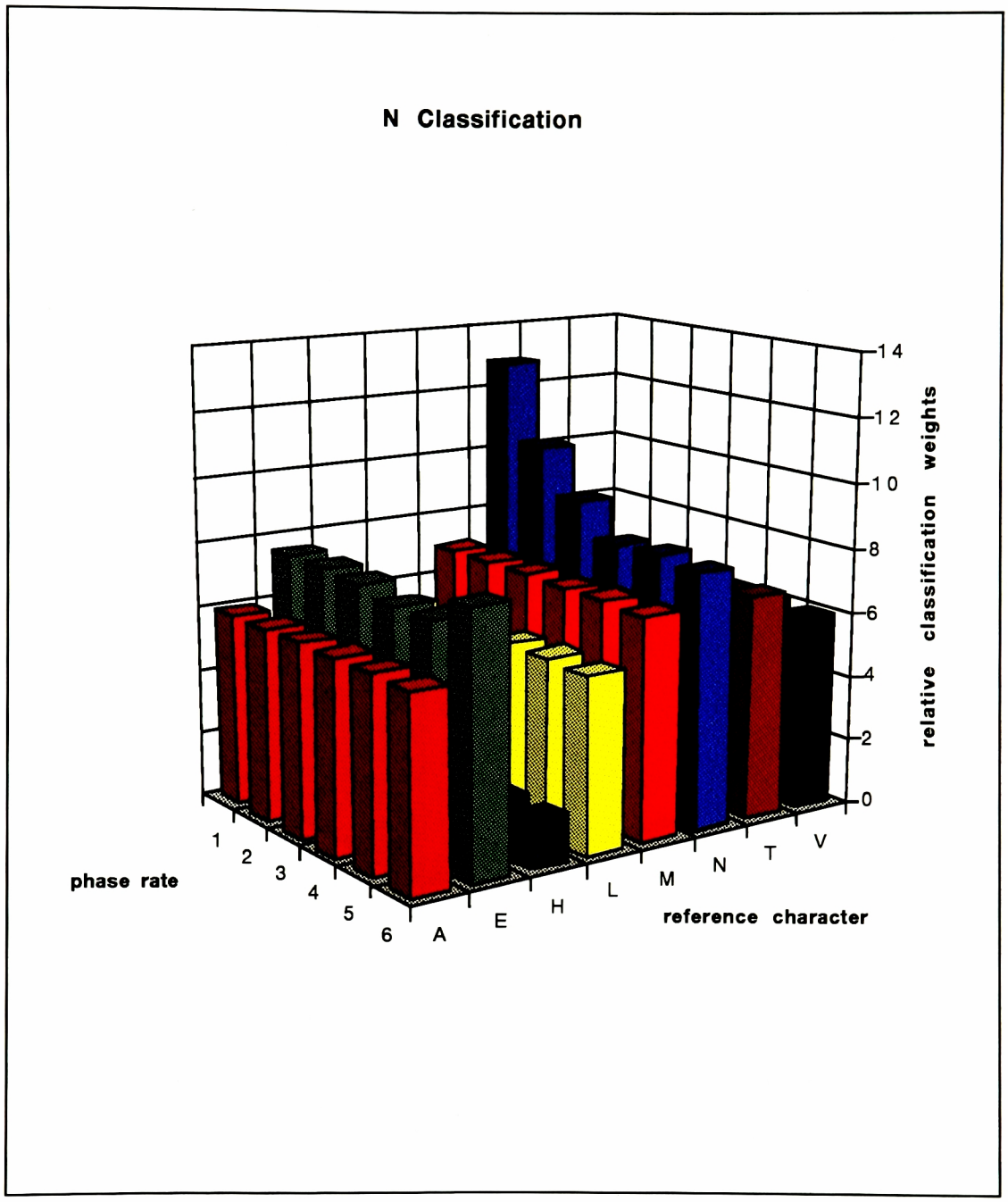



**Figure 20. Classification graph for the letter 'N'.**

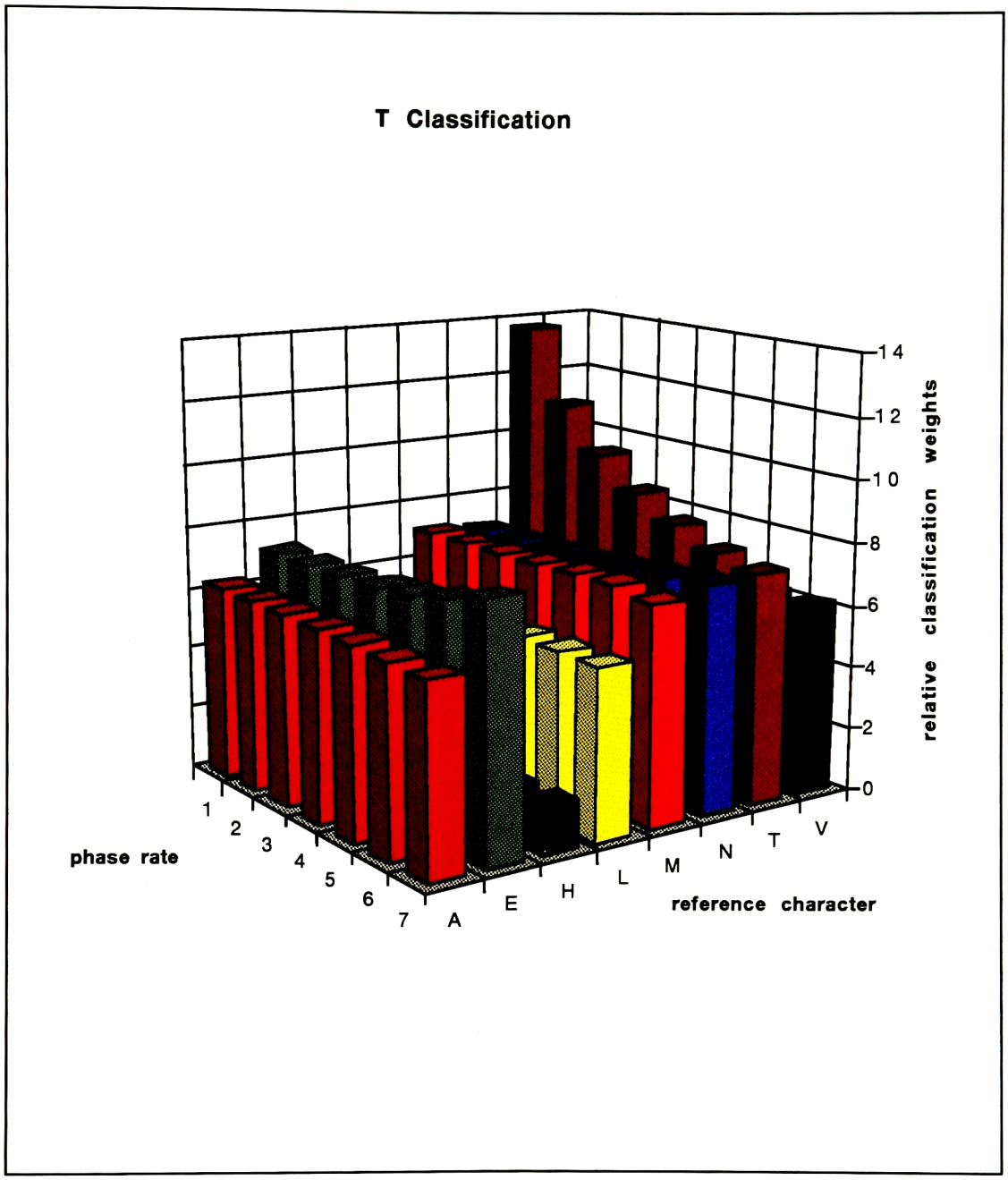



**Figure 21. Classification graph for the letter 'T'**

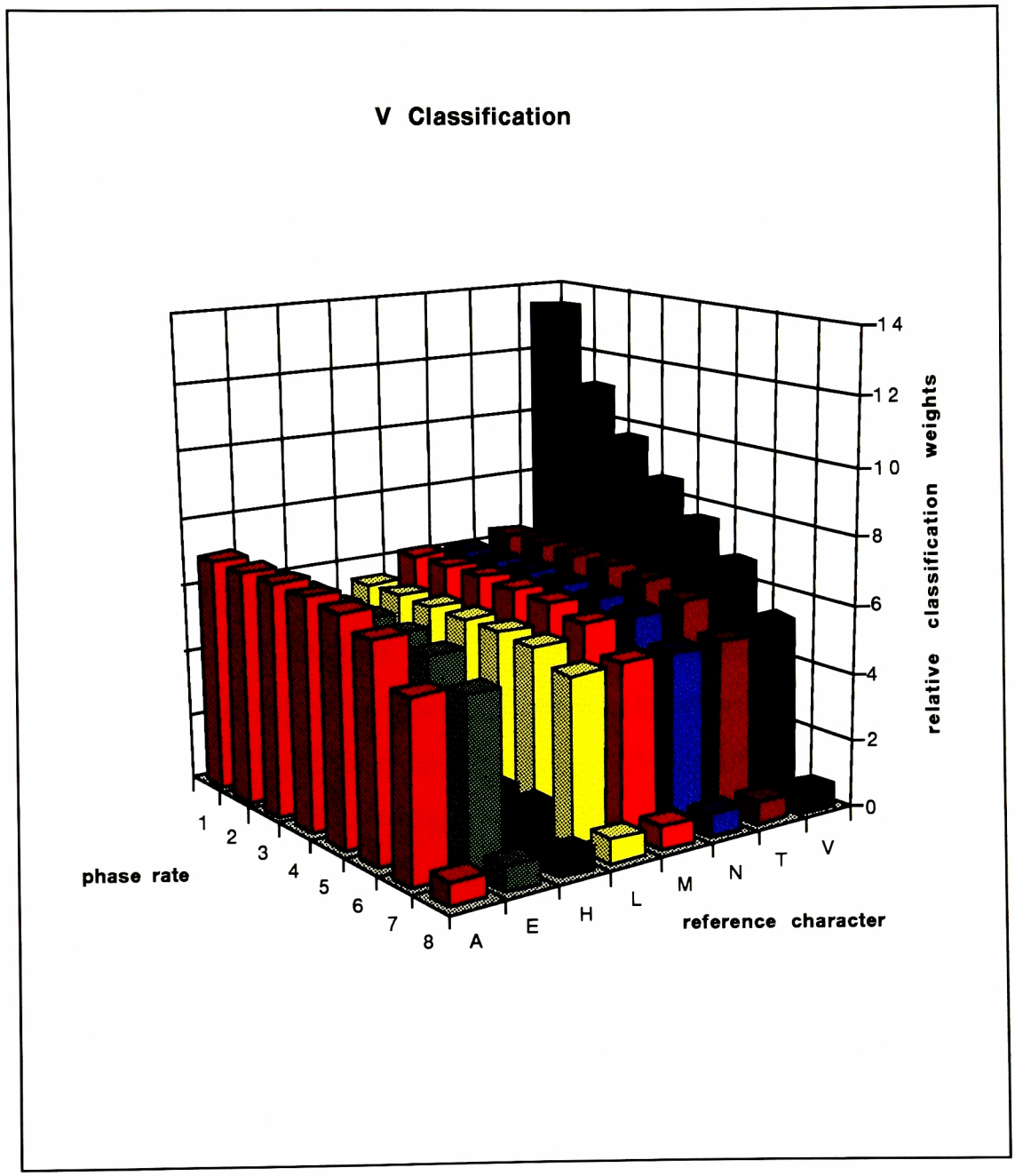



**Figure 22. Classification graph for the letter 'V'.**

By plotting the classification weights versus the phase rate, one can observe how the classifier behaves as the amount of blur is increased. The general performance of the classifier for all characters declines steadily as the blur is increased, and usually misclassifies when the phase rate of the quadratic filter reaches a value of six to eight units. A question to be answered is whether there is a functional relationship between the phase rate and distance. Before addressing this problem, an explanation for the results of each test character is provided in the following paragraphs.

The letter 'A' is one of the best behaved characters of the entire set; the performance of the classifier for this character is quite intuitive. The classifier has no difficulty recognizing the test character as the letter 'A' through several levels of degradation, and eventually mis-classifies as the letter 'V' when the phase rate reaches 7 pixels. One might expect a blurred letter 'A' to be mistaken for the letter 'V' since the two shapes are very similar. From figure 14, one can see that a phase rate of 7 pixels degrades the test character almost beyond visual recognition. This example clearly demonstrates the ability of the classifier to recognize degraded text.

The letter 'E' behaves similarly to the letter 'A'. However, one can see that this letter is correctly classified for more severe blurring than for the previous letter, and that the rise in relative weight values from blur level 7 to 8 seems to indicate an improved

recognition ability. This jump in weight value does not actually indicate an increase similarity between that test character and the reference character for that level of blur. Clearly, the test character with phase rate of eight pixels is less recognizable (*i.e.* is blurred to a greater extent) than the same character with a phase rate of 7 pixels. A valid question is why there appears to be improved performance when in fact the distance between test and reference characters should be increasing. This is an effect of the blurring of the phase filter. One can see from figure 14 that phase rates of 7, 8 and 9 pixels blur the character almost beyond visual recognition; the noise is dominant component of the image rather than the character. The effect of this much blur causes the feature vector of the test character to shift so that it is numerically closer to the feature vector of the reference character, resulting in an apparent improvement in recognizability of the test character. It would be incorrect to conclude that the performance of the classifier actually improved between the test character blurred with a filter having a phase rate of 7 pixels and that of 8 pixels.

One can also observe that the test character does not misclassify at the highest level of degradation. This test image is comprised completely of noise, yet is still closer to the correct reference character than to any other reference character. Thus, the letter 'E' was never incorrectly identified. Further levels of degradation were not continued since a phase rate of 9 pixels

represents a cutoff point where minimal useful information remained in the test image and it is not the objective here to determine the performance of the classifier in pure noise. It should be noted that the classifier's ability to recognize the letter 'E' under greater degradation than the letter 'A' does not necessarily indicate that the letter 'E' is more easily identified. The chirp rate at which a character is misclassified may differ if a larger reference set is used. For example, if the letter 'F' was added to the reference set, the letter 'E' may be misclassified with only a small amount of blur applied to it. Since the letter 'E' has unique shape characteristics, its feature vector remains significantly different than the rest of the character set.

The letter 'H' has the most peculiar behavior. It is misclassified after blurring with a filter having a chirp rate of 3 pixels. The explanation for this result is based on the shape of the letter; 'H' is the only character that is symmetric both horizontally and vertically. Because of this symmetry, the higher-order moments are approximately zero, and components of the feature vector calculated from these moments also nearly vanish. Therefore, the character is mostly described by the first- and second-order moments, and the feature vector is effectively reduced from a six-dimensional space to a two-dimensional space consisting of the mean and variance of the gray levels in the image. It is easy to see from figure 8 that a small amount of blur applied to the test image causes a large change in the

mean and variance. As these features change, the statistical distance between the test and reference characters quickly increases, resulting in the rapid misclassification of this letter.

The example of the letter 'H' illustrates a possible weakness in the classifier when attempting to recognize characters having horizontal and vertical symmetry. To compensate for this problem, the feature vector must include elements that check for circular symmetry, and create part of the feature space based on that property of the character. Though perhaps obvious, it should be noted that the shape of a given character depends on the font used, and the classifier will perform differently for different font types. Helvetica font was used in this experiment; this is a very simple font with equal-width lines and no serifs. However, other fonts may not have constant line width. In this case, symmetry properties may not be as significant due to the different line widths in the character.

The classifier demonstrates robust behavior when classifying the letter 'L'; this letter is easily recognized through several levels of degradation and is eventually incorrectly classified as the letter 'M'. Having been misclassified under the fifth level of degradation (filter with a phase rate of 5 pixels) suggests that perhaps this character is not as easily classified as some of the other letters. However, this letter has no distinguishing shape characteristics (lines coming to a point, cross-bars, etc.) found in other reference characters and the components of this letter are found in many other characters. One

can think of this letter as having a rather "generic" feature vector. Observing figure 18, one can see that when blurred in large amounts, the feature vector of the 'L' is very similar to all reference vectors, except the letter 'H' for reasons described earlier. The lack of distinguishing characteristics pertaining to this letter explain the mediocre performance of this letter.

Letters 'M' and 'N' both present few problems for the classifier. The letter 'M' is recognized two levels beyond the letter 'N', but this is understandable since the letter 'M' is a more complex shape than 'N' and therefore will tend to retain its uniqueness longer. The letter 'M' is misclassified as the letter 'N' at eight levels of degradation, thus one might expect the letter 'N' to be misclassified as the letter 'M'. However, one can observe from the graph that the letter 'N' is actually misclassified as the letter 'E'. Although this is perhaps not the intuitive prediction one would make, the misclassification is a result of the blur applied to the character. Though the filtering function is the same for all characters, the effect of the defocus blur will vary between each character. Thus it is not trivial to predict how a given letter will be misclassified; as stated earlier, misclassification of a letter also depends on the size of the reference letter set. However, it is useful to know how a given letter was misclassified. If a character was incorrectly identified as the letter 'F' for example, one might be able to use that information to logically

deduce what the correct identification was instead of guessing with no prior information.

Letters 'T' and 'V' are also both easily handled by the classifier in the presence of high levels of defocus blur. The letter 'T' misclassifies as the letter 'E' at level 7 and the 'V' is not misclassified after level 9. The performance of the letter 'V' is similar to the letter 'E' in that neither of these characters are incorrectly identified. Observing figure 22, at $\alpha = 8$ the distances between the test 'V' and reference characters is very small due to the high level of blur in the test character. Thus level 8 represents the cutoff phase rate for the letter 'V'; classification of levels higher than that would have little meaning since the test image would contain minimal character information. The following chart summarizes the classification performance of the eight characters.

| reference character | degradation level (phase rate) | misclassified as |
|---|---|---|
| A | 7 | V |
| E | 9 | N/A |
| H | 3 | N |
| L | 5 | M |
| M | 8 | N |
| N | 6 | E |
| T | 7 | E |
| V | 8 | N/A |

**Figure 23. Chart of classification summary.**

As stated at the beginning of this section, it is desirable to determine if there is a functional relationship between the phase rate of the quadratic phase filter and the statistical distance between test and reference characters for a particular letter resulting from that phase rate. If this relationship can be established, then it might be possible to predict classification performance as a function of $\alpha$. To investigate this question, the data was tested to find a power-law relationship between phase rate and distance. In other words, the variables would be functionally related as follows:

$$y = (m\alpha + b)^n \tag{39}$$

$$y^{\frac{1}{n}} = (m\alpha + b) \tag{39a}$$

where y is the distance between a given test character and its matching reference character, n is some exponential power that satisfies the equation, m is the slope of the linear relationship, $\alpha$ is the phase rate, and b is the intercept of the line with the y-axis. If n can be found that satisfies equation 39 to some statistical criterion, then the relationship between distance and phase rate is known.

Several values of n were tested in equation 39, these consisted of 2, 4, 5, 6, 7, and 8. The following graphs show the best linear fit that could be established between the two parameters. Since $\alpha$ is linear, the regression curves were plotted using equation 39a.

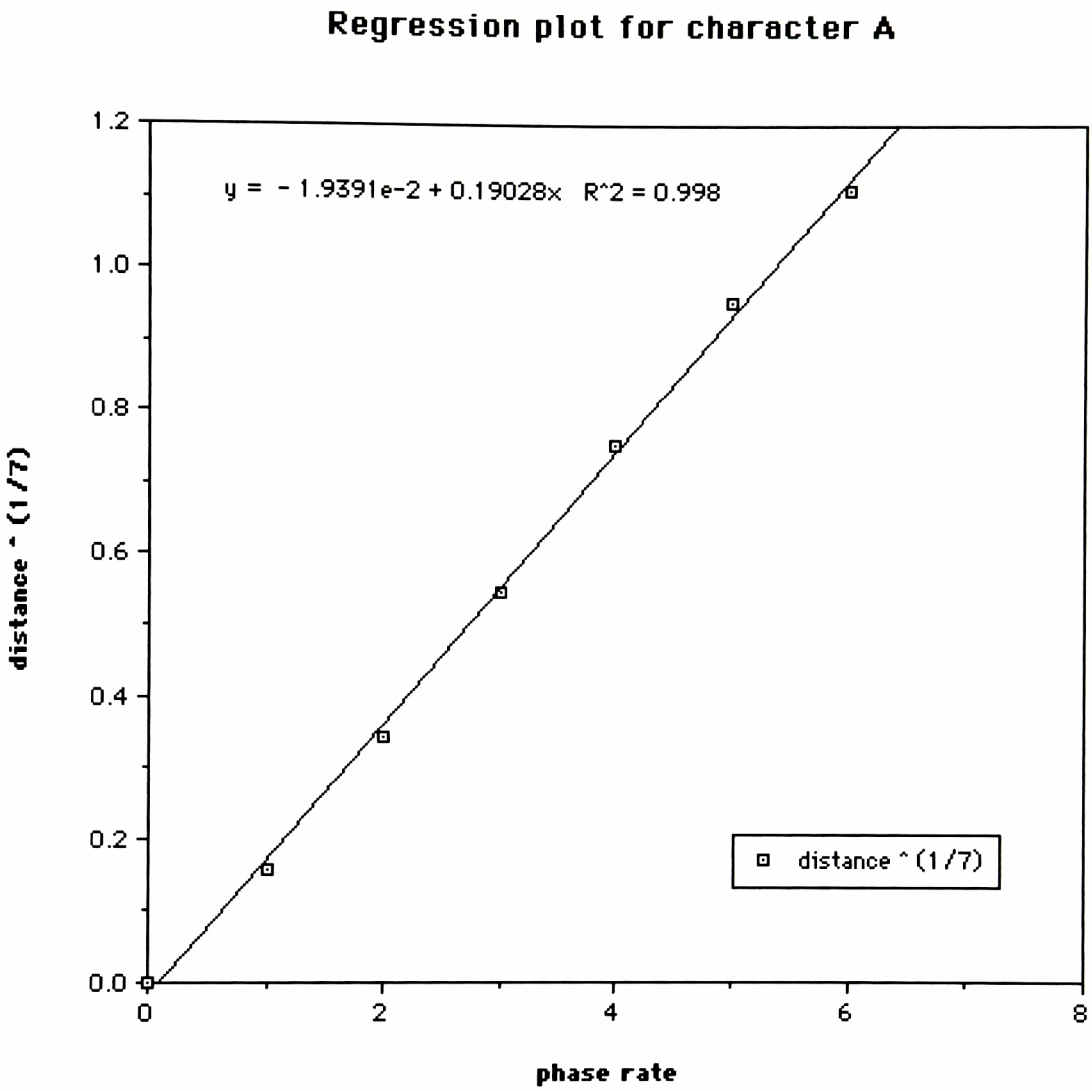



Figure 24. Regression curve for letter A.

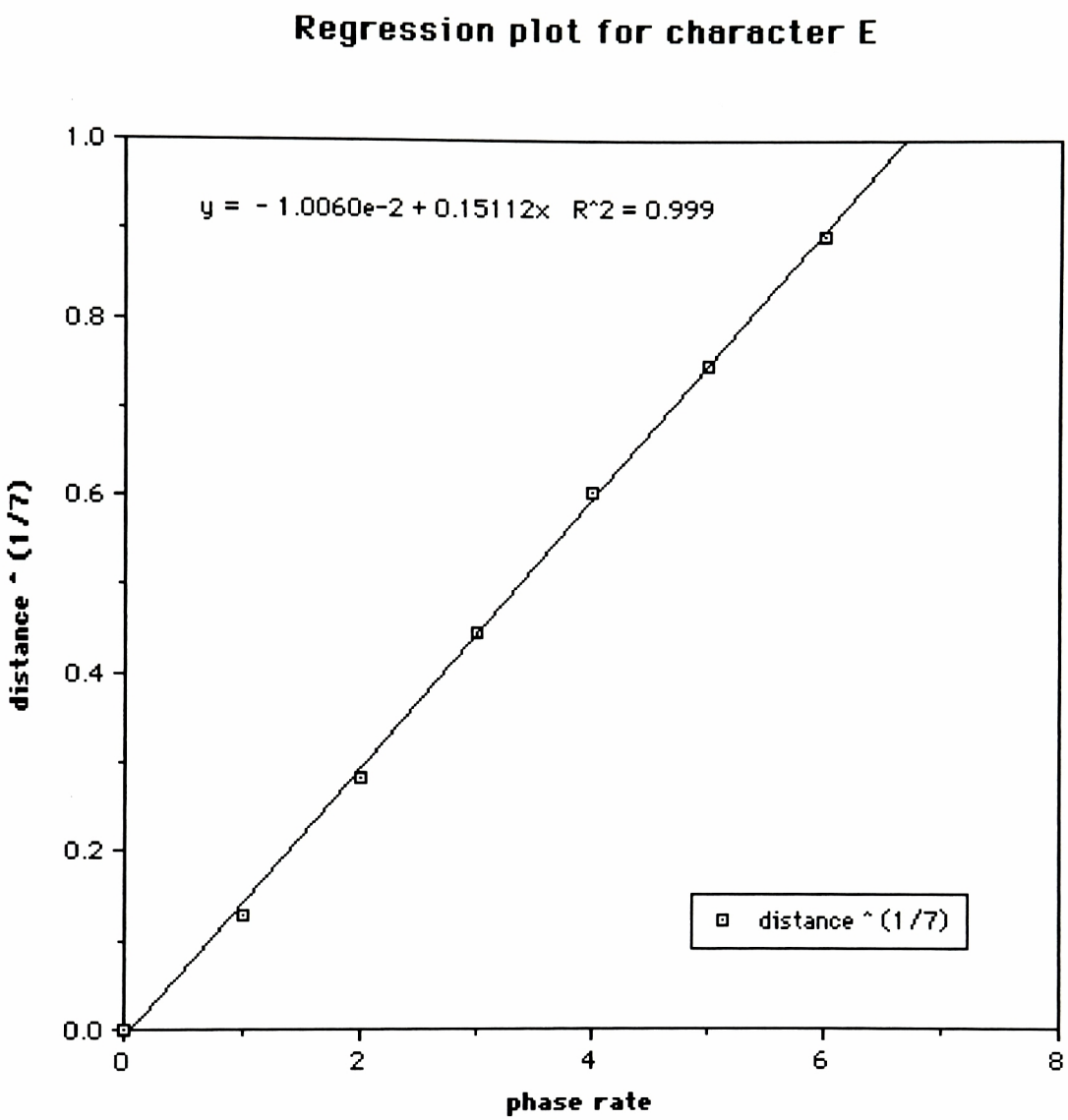



Figure 25. Regression curve for letter E.

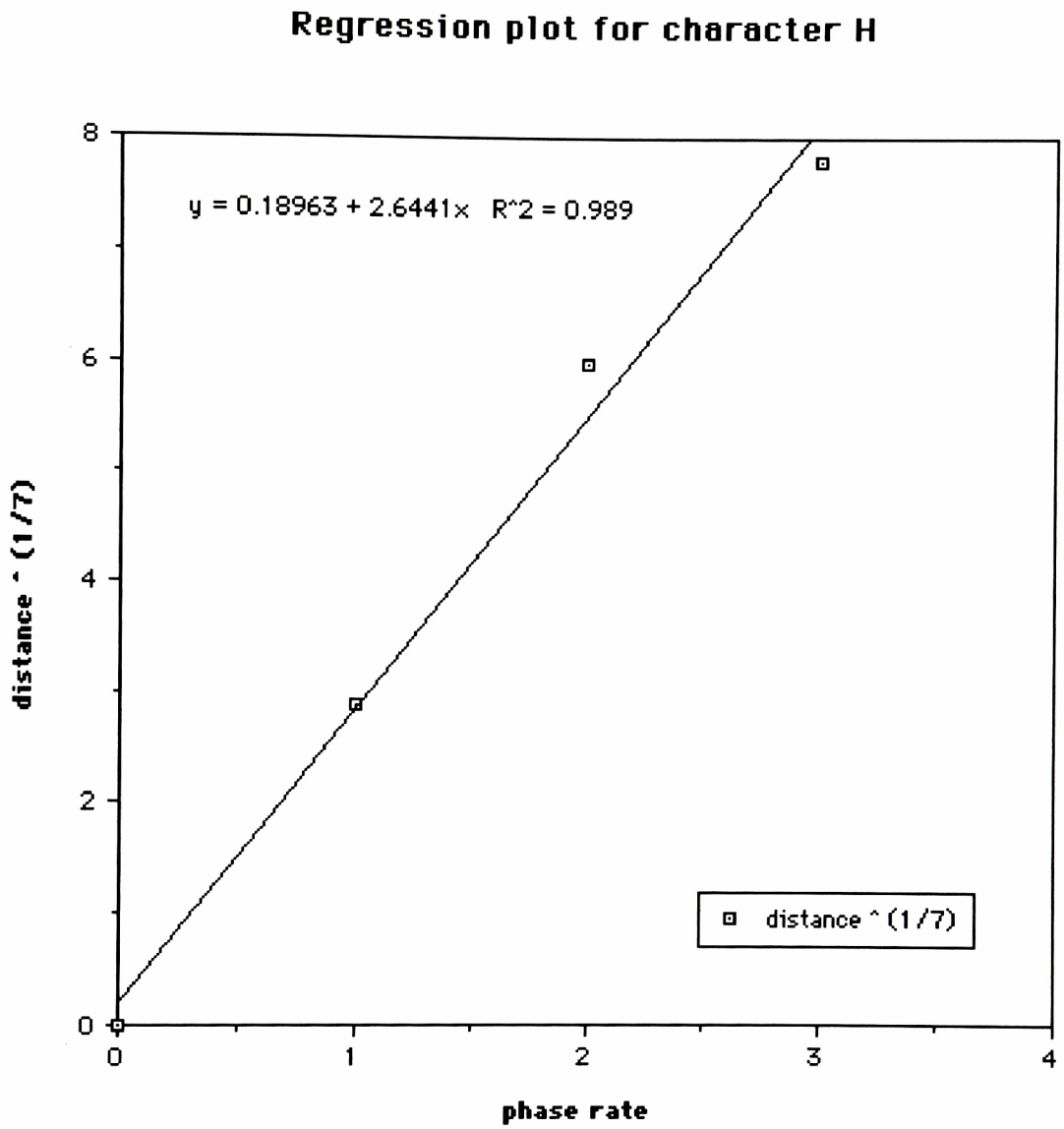



Figure 26. Regression curve for letter H.

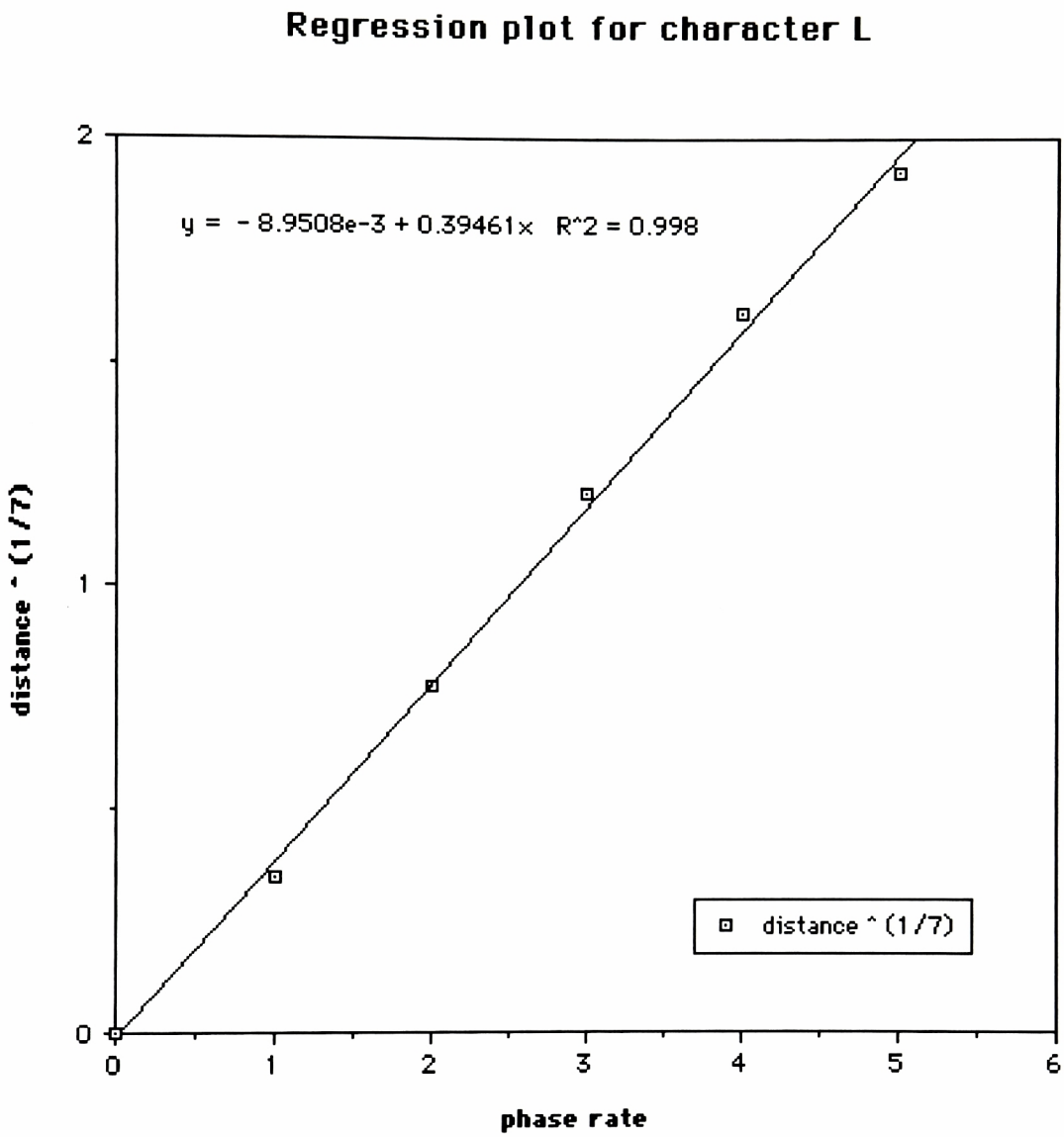



Figure 27. Regression curve for letter L.

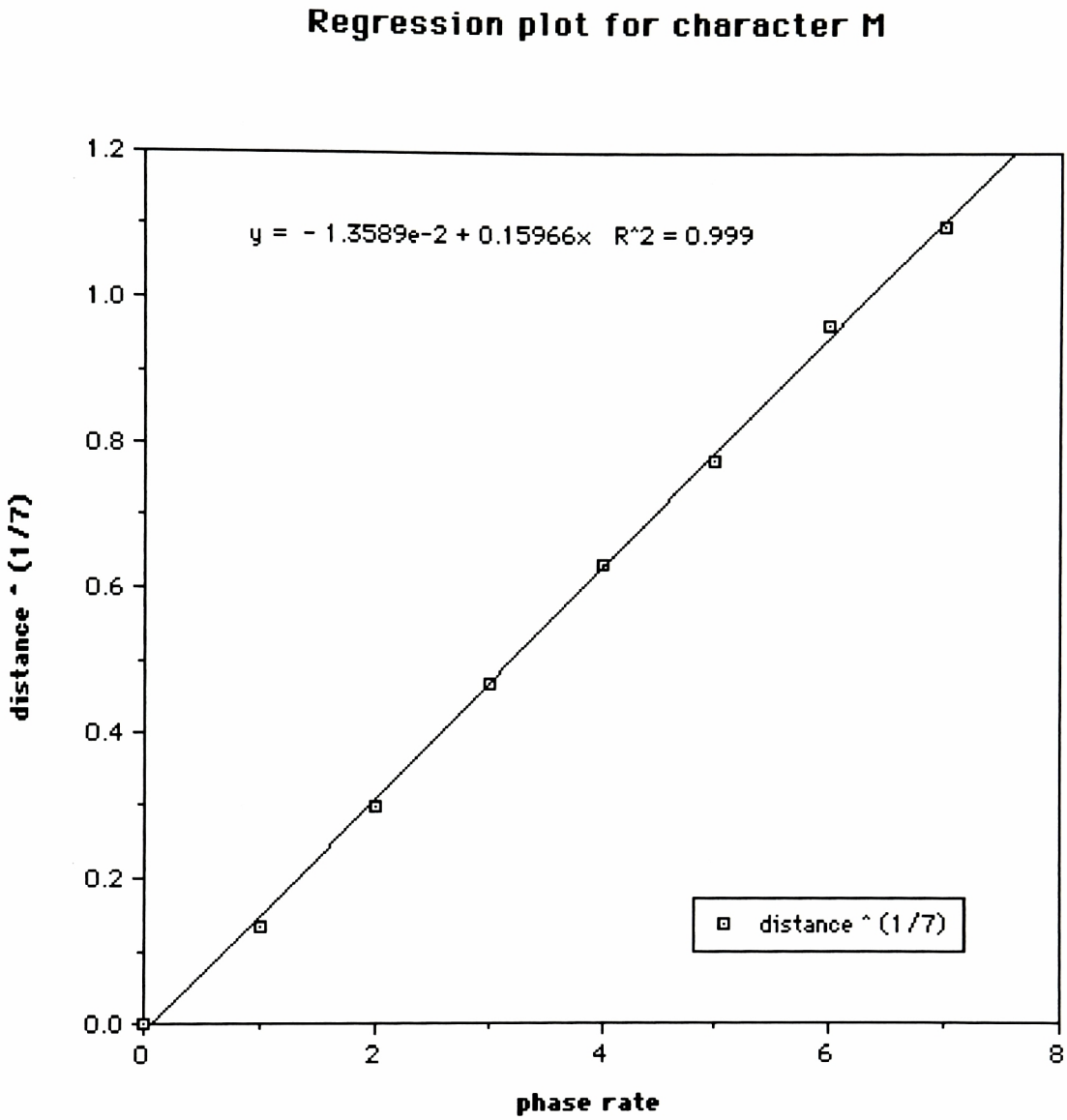



Figure 28. Regression curve for letter M.

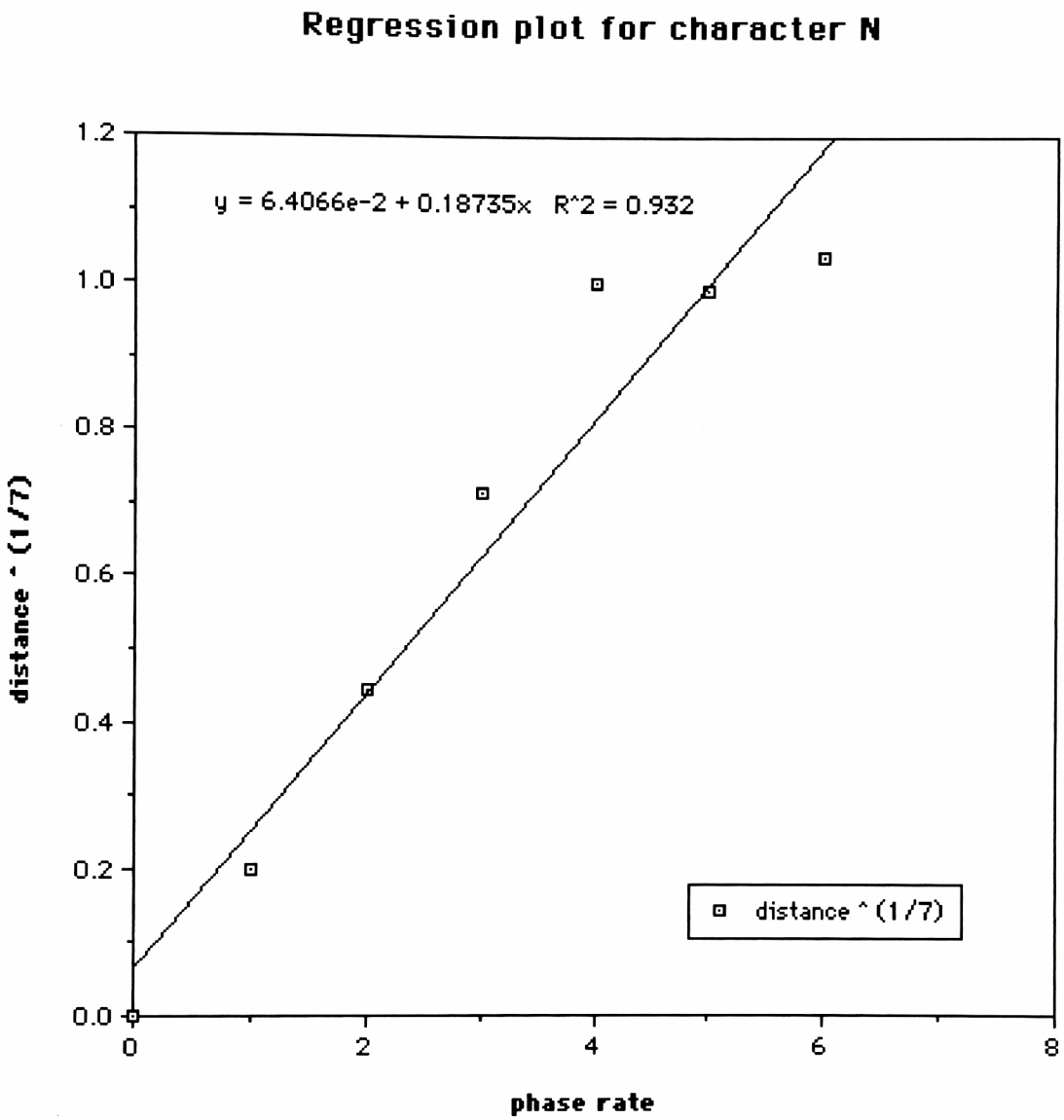



Figure 29. Regression curve for letter N.

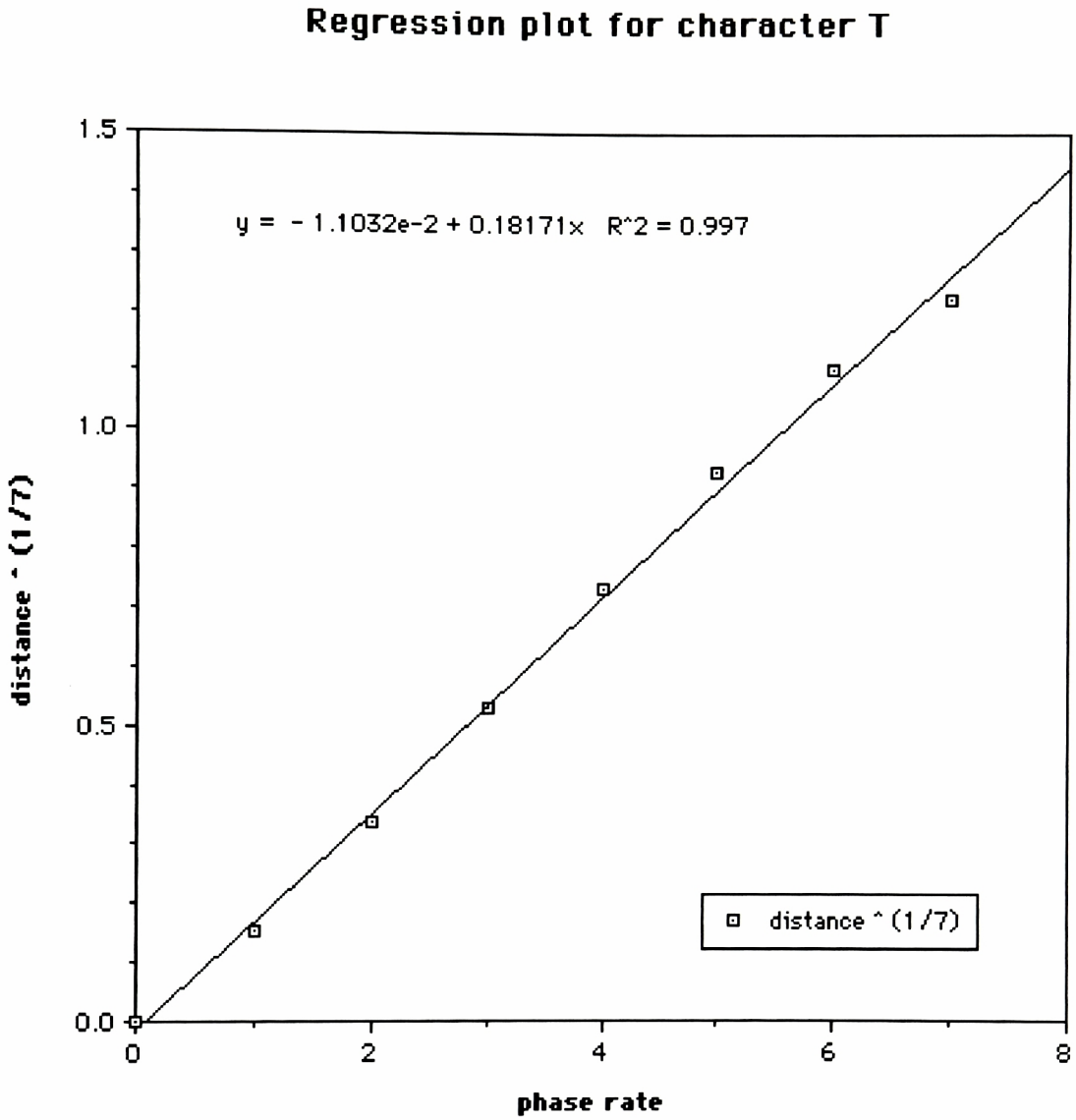



Figure 30. Regression curve for letter T.

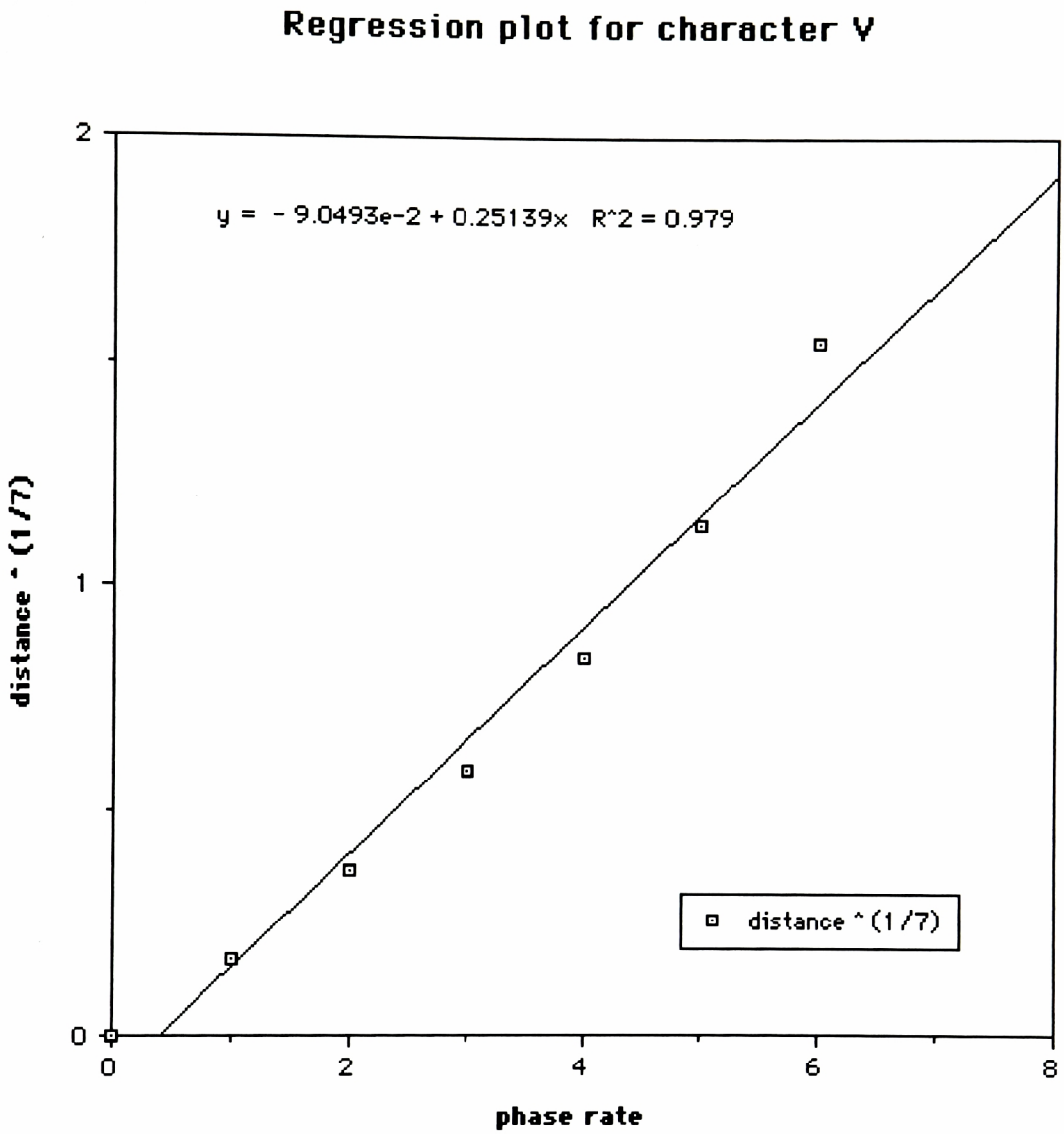



**Figure 31. Regression curve for letter V.**

These regression curves demonstrate a very high correlation between the phase rate of the blur filter and the 1/7 power of the

distance. Thus, a good experimental value for n in equation 39 is 7, and distance as a function of phase rate would be written as:

$$y = \left(m\alpha + b\right)^7 \tag{40}$$

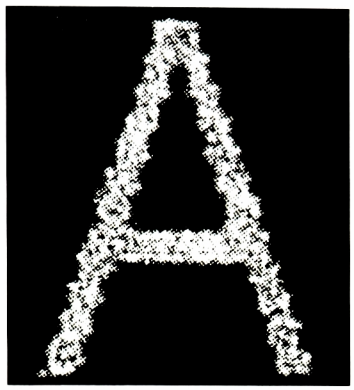
The distance between reference character a test character blurred with a filter having a phase rate of $\alpha = 0$ is zero, thus the regression line for every character should go through zero. However, the regression lines do not go through zero since all the data for a given characer does not fall on a straight line. If the regression lines were forced through zero, the line equations would be slightly different, a different value for n may also be observed.

The regression curve does not fit as well for letter 'H' because of the small data set. The regression curve for the letter 'N' is also not very convincing, due to the influence of defocus blur from the quadratic phase filter in the classification. Because of the apparent jump in classification performance from level 4 to level 5, the regression curve does not fit the data as well as most of the other characters. All other data sets have a very convincing correlation between the phase rate and distance according to equation 40. When n = 6 or 8, a high correlation was also present in the data, however n = 7 provided the best overall correlation between the two variables. Having experimentally determined this relationship, it is more difficult to determine why this particular relationship exists between the phase rate of the quadratic filter and the distance between a test

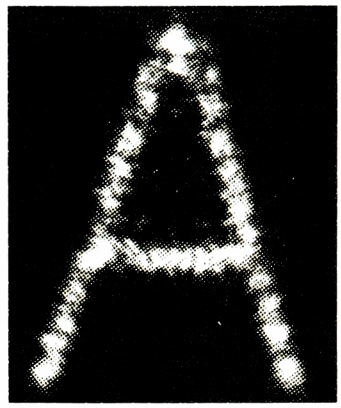
character and its matching reference. This question has been left as further research to be conducted.

## 4.1 Optical validation of digital simulation of blurred characters

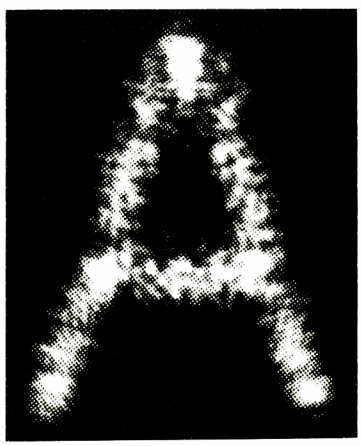
To validate a digital-filtering approximation of defocusing in a coherent optical system, a transparency of the uppercase 'A' was tested in an optical system consisting of a collimated laser source. The object was "projected" onto the film plane of a 35mm camera and photographed at several distances from the object to observe increasing defocus in the image. A translation of the image plane with respect to the object in the optical system was equivalent to changing the phase rate $\alpha$ of the filter in the digital case. The recorded images are shown in figure 32 A-D.



A



B



C

D

**Figures 32 A-D. Photographs captured in the image plane of the defocusing optical system.**

These images should be compared to the test images generated by digital filtering (figure 14). Note the similarities of the blurred images. This confirms that the digital quadratic phase filter accurately models defocus blur in an imaging system.

As a final step in the validation experiment, the value for $\alpha$ was experimentally calculated for the letters in figure 32 C and D. Figure 32 C closely resembles the image in figure 14 that was blurred with a filter having a phase rate $\alpha = 4$. Thus the experimental value for $\alpha$ should be close to 4 pixels, which corresponds to a distance measure of 1.53mm. The distance was measured from the object to image plane, Z = 3.6m. Point A represents the location where there has been a phase shift of $\pi$ radians in the light reaching the image plane.

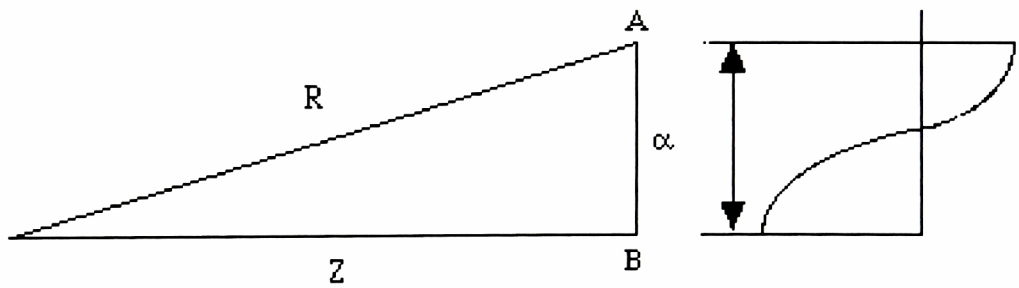



**Figure 33. Diagram for experimental calculation of $\alpha$.**

Thus there is a phase difference of $\pi$ radians between light reaching point B and light reaching point A (see figure 4). A phase shift of $\pi$ radians is created by delaying a sinusoid by one-half wavelength. Thus, light reaching point A had to travel one half wavelength longer

than light reaching point B, indicating that $R = Z + \lambda / 2$. The answer for $\alpha$ is determined simply from the Pythagorean theorem. The calculated for $\alpha' = 1.51$mm was $\alpha = 1.53$mm. For figure 32-D, the theoretical value for $\alpha' = 1.92$mm compared to an experimental value of $\alpha = 1.81$mm. Thus experimental values for $\alpha$ determined from the optical system coincide with values used for digital filtering.

# 5.0 - CONCLUSION

It was desirable to create a classifier that would classify a letter based solely on its shape. Each character was transformed into a six-dimensional feature vector calculated from a set of moment invariants. The vectors are invariant to translation, rotation, scaling, and contrast changes of the character. The results of this project demonstrate that the classifier can correctly identify most characters in the presence of high levels of defocus blur. The only character with a problem was the letter 'H'. Due to the symmetry of this character, the feature vector was almost completely described by the first two dimensions of the vector; these two dimensions are functions of the mean and variance of the image respectively. Since these two variables change rapidly with increased blur, the character was misclassified sooner than the others. Future research conducted will need to alter the feature space in order to account for the symmetry of some characters. Having tested the classifier with a pilot set of characters, future work needs to examine the performance when using the entire alphabet.

A functional relationship was formed that relates the phase rate of a filter and the distance measure between a test character and its matching reference character. It was discovered that distance between the vectors of the original and blurred character varies on the seventh power of the phase rate. Future research

should investigate this result and determine the reason for this relationship. A final area of further research will be to implement other language bases into the classifier. The elementary letters of any language could be used the same way the characters chosen for this project. Each image would consist of one character, the reference set would consist of individual images of all the letters of that particular alphabet. The eventual objective is to use the classifier to analyze images containing severely degraded text. As an example, these images may contain fragments of ancient scroll text which translators need to identify. Instead of attempting to enhance the image to make the text more readable, the classifier may predict the text correctly. A combination of these two procedures may also prove useful in the analysis of such images. Although this is a long-term objective, the results from this thesis indicate that such an objective could be achieved.

# APPENDIX A

The following results are taken from Casasent, Psaltis (1979) demonstrating the invariant properties of moments. The first table shows the compiled theoretical results of moments for the two-dimensional figure and its variations. The second figure and table shows the experimental results obtained from the optical generation of the moments.

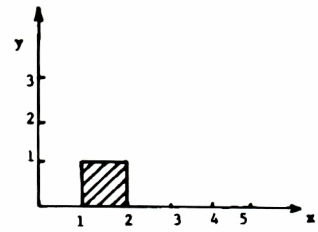



Fig.1. Original Input

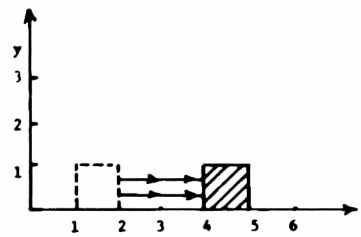



Fig.2. Translated Input

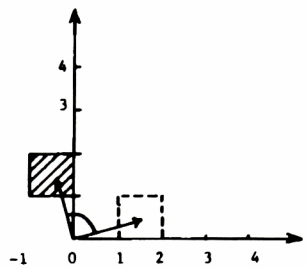



Fig.3. Rotated Input

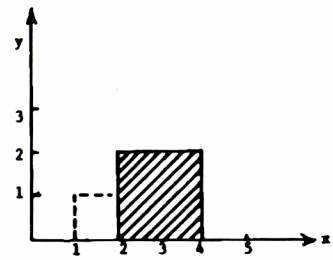



Fig.4. Scaled Input

**NUMERICAL CASE STUDY EXAMPLE**

Table 1

THEORETICAL CASE STUDY (SQUARE INPUT)

| n | (p,q) | ORIGINAL | | | TRANSLATED ($\phi_n$) | ROTATED ($\phi_n$) | SCALED ($\phi_n$) |
|---|---|---|---|---|---|---|---|
| | | $m_{pq}$ | $\mu_{pq}$ | $\phi_n$ | | | |
| 1 | 00 | 1 | 1 | 0.166 | 0.166 | 0.166 | 0.166 |
| 2 | 10 | 1.5 | 0 | 0 | 0 | 0 | 0 |
| 3 | 01 | 0.5 | 0 | 0 | 0 | 0 | 0 |
| 4 | 20 | 0.33 | 0.083 | 0 | 0 | 0 | 0 |
| 5 | 02 | 0.33 | 0.083 | 0 | 0 | 0 | 0 |
| 6 | 30 | 3.75 | 0 | 0 | 0 | 0 | 0 |
| 7 | 03 | 0.25 | 0 | 0 | 0 | 0 | 0 |
| | 11 | 0.75 | 0 | $m_{pq}$ = ordinary moments | | | |
| | 21 | 1.66 | 0 | $\mu_{pq}$ = central moments | | | |
| | 12 | 0.5 | 0 | $\phi_n$ = invariant moments | | | |

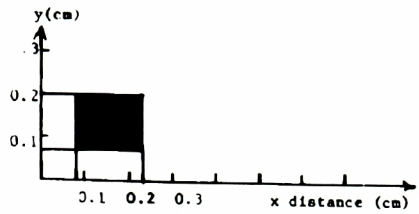



Fig.6. Input

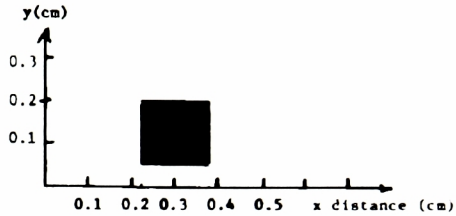



Fig.7. Translated input


Fig.8. Scaled input

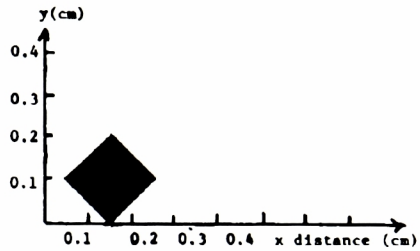



Fig.9. Rotated input

Table 2. Comparison of theoretical and experimental $\bar{m}_{pq}$ moment computations.

| NORMALIZED MOMENT | ORIGINAL | | TRANSLATED | | SCALED | | ROTATED | |
|---|---|---|---|---|---|---|---|---|
| | THEORY | EXPERIM | THEORY | EXPERIM | THEORY | EXPERIM | THEORY | EXPERIM |
| $m''_{00}/m''_{10}$ | 2.28 | 2.27 | 2.0 | 1.65 | 2.16 | 1.82 | 2.37 | 1.71 |
| $m''_{00}/m''_{20}$ | 5.03 | 5.2 | 4.32 | 4.64 | 4.4 | 5.16 | 5.62 | 5 |
| $m''_{00}/m''_{11}$ | 5.25 | 6.7 | 4.87 | 5.8 | 4.7 | 5.3 | 5.4 | 6 |
| $m''_{00}/m''_{30}$ | 11.16 | 11.7 | 8.9 | 9.5 | 9.8 | 12.8 | 11.8 | 11.8 |
| $m''_{00}/m''_{21}$ | 11.62 | 12.3 | 9.7 | 10.4 | 10.2 | 11.8 | 11.25 | 12.1 |

(Results from Casasent and Psaltis, 1979)

# APPENDIX B

The following pages contain data tables for the classification distance graphs, invariance demonstration graphs, and the regression graphs. For the classification distance tables (tables 5-12), there are three sections, showing the data as it was converted from distance values to the relative classification values used in the graphs. Each section of data has a label heading, demonstrating how that section was calculated. Tables are listed in the appendix in order that they appear in the thesis.

| reference character | 1/distance |
|---|---|
| A | 100 |
| E | 0.00865138 |
| H | 3.09E-07 |
| L | 0.00409296 |
| M | 0.0134013 |
| N | 0.00887351 |
| T | 0.0155473 |
| V | 0.0796327 |

**Table 1. Data for translation invariance graph.**

| reference character | 1/distance |
|---|---|
| A | 100 |
| E | 0.00865138 |
| H | 3.09E-07 |
| L | 0.00409296 |
| M | 0.0134013 |
| N | 0.00887351 |
| T | 0.0155473 |
| V | 0.0796327 |

**Table 2. Data for contrast invariance graph.**

| reference character | 1/distance |
|---|---|
| A | 3.39492 |
| E | 0.0082063 |
| H | 3.09E-07 |
| L | 0.00435787 |
| M | 0.0128603 |
| N | 0.00848253 |
| T | 0.0143271 |
| V | 0.0610407 |

**Table 3. Data for rotation invariance graph.**

| reference character | 1/distance |
|---|---|
| A | 0.38439 |
| E | 0.0119313 |
| H | 3.09E-07 |
| L | 0.00408157 |
| M | 0.0201603 |
| N | 0.0122413 |
| T | 0.0239943 |
| V | 0.0860049 |

**Table 4. Data for scale invariance graph.**

| phase rate | | | | 1/Distance | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | A | E | H | L | M | N | T | Y |
| 1 | 442323 | 0.00865355 | 3.09E-07 | 0.00409322 | 0.0134059 | 0.0088758 | 0.0155522 | 0.0796632 |
| 2 | 1736.95 | 0.00868609 | 3.09E-07 | 0.00409702 | 0.0134745 | 0.00891015 | 0.0156266 | 0.0796184 |
| 3 | 69.3306 | 0.00882734 | 3.09E-07 | 0.00411295 | 0.0137737 | 0.00905931 | 0.0159505 | 0.079508 |
| 4 | 7.37272 | 0.00920977 | 3.09E-07 | 0.00415152 | 0.014596 | 0.00946336 | 0.0168366 | 0.0788466 |
| 5 | 1.39962 | 0.0100265 | 3.09E-07 | 0.00421286 | 0.0164075 | 0.0103268 | 0.0187744 | 0.0762887 |
| 6 | 0.485206 | 0.0110783 | 3.09E-07 | 0.0043245 | 0.01904 | 0.0114635 | 0.0212475 | 0.0670945 |
| 7 | 0.0362197 | 0.0129558 | 3.09E-07 | 0.00243497 | 0.0147146 | 0.0119806 | 0.0275486 | 0.205435 |
| | | | | | | | | |
| | | | | Log( 1/Distance) | | | | |
| | A | E | H | L | M | N | T | Y |
| 1 | 5.64573952 | -2.0628057 | -6.5094418 | -2.3879349 | -1.872704 | -2.0517925 | -1.8082082 | -1.0989124 |
| 2 | 3.23978732 | -2.0611757 | -6.5094418 | -2.3875319 | -1.8704873 | -2.050115 | -1.8061355 | -1.0989866 |
| 3 | 1.84092496 | -2.0541701 | -6.5094418 | -2.3858466 | -1.8609494 | -2.0429049 | -1.7972257 | -1.0995892 |
| 4 | 0.86762774 | -2.0357512 | -6.5094404 | -2.3817929 | -1.8357661 | -2.0239546 | -1.7737456 | -1.103217 |
| 5 | 0.14601014 | -1.9988506 | -6.509439 | -2.3755423 | -1.7849576 | -1.9860342 | -1.7264339 | -1.1175398 |
| 6 | -0.3140738 | -1.9555269 | -6.5094376 | -2.3640641 | -1.7203331 | -1.9406828 | -1.6726922 | -1.1733131 |
| 7 | -1.4410552 | -1.8875358 | -6.5094376 | -2.2613506 | -1.8322515 | -1.9215214 | -1.5599005 | -0.6873256 |
| | | | | | | | | |
| | | | | Log( 1/Distance) + Bias, bias = 8 | | | | |
| | A | E | H | L | M | N | T | Y |
| 1 | 13.645739 | 5.93719431 | 1.49055821 | 5.61206509 | 6.12729598 | 5.94820751 | 6.19179183 | 6.90108762 |
| 2 | 11.239787 | 5.93882432 | 1.49055821 | 5.61246808 | 6.12951266 | 5.94988502 | 6.1938645 | 6.90101345 |
| 3 | 9.8409249 | 5.94582985 | 1.49055821 | 5.61415343 | 6.13905062 | 5.95709512 | 6.2027743 | 6.90041083 |
| 4 | 8.8676277 | 5.96424878 | 1.49055961 | 5.61820713 | 6.16423385 | 5.97604536 | 6.22625439 | 6.89678297 |
| 5 | 8.1460101 | 6.00114936 | 1.49056101 | 5.62457703 | 6.21504241 | 6.01396577 | 6.27356607 | 6.88246021 |
| 6 | 7.6859261 | 6.04447312 | 1.49056242 | 5.6359359 | 6.27966694 | 6.05931724 | 6.32730784 | 6.82668692 |
| 7 | 6.5589448 | 6.11246424 | 1.49056242 | 5.38649361 | 6.16774846 | 6.07847857 | 6.44009953 | 7.31267444 |

**Table 5. Classification distance data for the letter A.**

| phase rate | | | | 1/Distance | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | A | E | H | L | M | N | T | Y |
| 1 | 0.00865171 | 1.68E+06 | 3.10E-07 | 0.00260006 | 0.0946821 | 0.364632 | 0.116274 | 0.0104029 |
| 2 | 0.00865647 | 667471 | 3.10E-07 | 0.00260035 | 0.0949182 | 0.360998 | 0.116611 | 0.0104083 |
| 3 | 0.00867667 | 280.485 | 3.10E-07 | 0.00260154 | 0.0958976 | 0.34637 | 0.11802 | 0.0104313 |
| 4 | 0.00872794 | 33.7635 | 3.10E-07 | 0.00260431 | 0.0982274 | 0.314315 | 0.12146 | 0.0104894 |
| 5 | 0.00881883 | 7.82775 | 3.10E-07 | 0.00260644 | 0.101638 | 0.267867 | 0.127393 | 0.0105959 |
| 6 | 0.00870094 | 2.21292 | 3.10E-07 | 0.00253408 | 0.0895048 | 0.192446 | 0.12651 | 0.0105866 |
| 7 | 0.00931434 | 1.44584 | 3.09E-07 | 0.00265971 | 0.123959 | 0.204743 | 0.150921 | 0.0110784 |
| 8 | 0.0086836 | 7.16381 | 3.10E-07 | 0.00256914 | 0.0937454 | 0.255428 | 0.1223 | 0.010504 |
| 9 | 0.00858791 | 3.33806 | 3.10E-07 | 0.00253219 | 0.087721 | 0.214763 | 0.119821 | 0.0104451 |
| | | | | | | | | |
| | | | | Log(1/Distance) | | | | |
| | A | E | H | L | M | N | T | Y |
| 1 | -2.0628985 | 6.22503776 | -6.5091065 | -2.5850166 | -1.0237321 | -0.4381452 | -0.9345174 | -1.9828456 |
| 2 | -2.0626592 | 3.82443224 | -6.5091121 | -2.5849682 | -1.0226505 | -0.4424952 | -0.9332605 | -1.9826202 |
| 3 | -2.0616469 | 2.44790964 | -6.5091331 | -2.5847695 | -1.0181923 | -0.4604597 | -0.9280444 | -1.9816616 |
| 4 | -2.0590882 | 1.52844746 | -6.5091808 | -2.5843073 | -1.0077674 | -0.5026349 | -0.9155667 | -1.9792494 |
| 5 | -2.054589 | 0.89363695 | -6.5092552 | -2.5839523 | -0.9929439 | -0.5720808 | -0.8948544 | -1.9748621 |
| 6 | -2.0604338 | 0.34496571 | -6.5093323 | -2.5961797 | -1.0481537 | -0.7156911 | -0.8978751 | -1.9752435 |
| 7 | -2.0308479 | 0.16012024 | -6.5093758 | -2.5751657 | -0.9067219 | -0.6887909 | -0.8212503 | -1.955523 |
| 8 | -2.0613002 | 0.85514406 | -6.5092467 | -2.5902122 | -1.02805 | -0.5927315 | -0.9125735 | -1.9786453 |
| 9 | -2.0661125 | 0.52349414 | -6.5092916 | -2.5965037 | -1.0568964 | -0.6680405 | -0.9214671 | -1.9810874 |
| | | | | | | | | |
| | | | | Log(1/Distance) + Bias, bias = 8 | | | | |
| | A | E | H | L | M | N | T | Y |
| 1 | 5.93710145 | 14.2250378 | 1.49089352 | 5.41498337 | 6.97626788 | 7.56185478 | 7.06548261 | 6.01715442 |
| 2 | 5.93734083 | 11.8244324 | 1.49088791 | 5.41503181 | 6.97734949 | 7.5575048 | 7.06673952 | 6.0173798 |
| 3 | 5.93835308 | 10.4479096 | 1.49086688 | 5.41523051 | 6.98180774 | 7.5395402 | 7.07195561 | 6.01833844 |
| 4 | 5.94091175 | 9.52844746 | 1.49081919 | 5.41569268 | 6.99223265 | 7.49736511 | 7.08443328 | 6.02075065 |
| 5 | 5.94541097 | 8.89363695 | 1.49074484 | 5.41604773 | 7.00705611 | 7.4279192 | 7.10514557 | 6.02513785 |
| 6 | 5.93956617 | 8.34496571 | 1.49066767 | 5.40382032 | 6.95184633 | 7.28430889 | 7.10212486 | 6.02475565 |
| 7 | 5.96915209 | 8.16012024 | 1.49062417 | 5.42483429 | 7.09327806 | 7.31120906 | 7.17874967 | 6.04447704 |
| 8 | 5.93869981 | 8.85514406 | 1.49075326 | 5.40978777 | 6.97194997 | 7.4072685 | 7.08742646 | 6.02135471 |
| 9 | 5.93388748 | 8.52349414 | 1.49070836 | 5.40349629 | 6.94310357 | 7.33195946 | 7.07853294 | 6.01891126 |

**Table 6. Classification distance data for the letter E.**

| phase rate | | | | 1/Distance | | | | |
|---|---|---|---|---|---|---|---|---|
| | A | E | H | L | M | N | T | Y |
| 1 | 3.24E-07 | 3.24E-07 | 0.000060844 | 3.24E-07 | 3.24E-07 | 3.25E-07 | 3.24E-07 | 3.24E-07 |
| 2 | 6.08E-07 | 6.08E-07 | 3.77E-06 | 6.08E-07 | 6.08E-07 | 6.09E-07 | 6.08E-07 | 6.08E-07 |
| 3 | 4.35E-06 | 4.37E-06 | 5.75E-07 | 4.35E-06 | 4.36E-06 | 4.39E-06 | 4.36E-06 | 4.35E-06 |
| | | | | | | | | |
| | | | | | | | | |
| | | | | Log( 1/Dsitance) | | | | |
| | A | E | H | L | M | N | T | Y |
| 1 | -6.4896306 | -6.4892861 | -3.2157815 | -6.4895971 | -6.4895837 | -6.488717 | -6.4895529 | -6.4895864 |
| 2 | -6.2163508 | -6.2158715 | -5.4234122 | -6.2163222 | -6.2162793 | -6.2150911 | -6.2162364 | -6.2162893 |
| 3 | -5.3613021 | -5.3598834 | -6.2401313 | -5.3615217 | -5.3609869 | -5.3577908 | -5.3608723 | -5.3611176 |
| | | | | | | | | |
| | | | | | | | | |
| | | | | Log( 1/Distance) + Bias, bias = 8 | | | | |
| | A | E | H | L | M | N | T | Y |
| 1 | 1.51036938 | 1.51071387 | 4.784218 47 | 1.5104029 | 1.51041631 | 1.51128295 | 1.51044715 | 1.51041363 |
| 2 | 1.78364921 | 1.78412853 | 2.5765878 | 1.7836778 | 1.78372068 | 1.78490886 | 1.78376355 | 1.78371068 |
| 3 | 2.63869787 | 2.64011656 | 1.75986871 | 2.6384782 7 | 2.63901309 | 2.64220921 | 2.63912775 | 2.63888244 |

**Table 7. Classification distance data for the letter H.**

| phase rate | | | | 1/Distance | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | A | E | H | L | M | N | T | V | |
| 1 | 0.00410452 | 0.00260694 | 3.09E-07 | 1483.41 | 0.00362717 | 0.00285131 | 0.00284935 | 0.00284285 | |
| 2 | 0.00428011 | 0.00271189 | 3.09E-07 | 5.97552 | 0.0038 | 0.00297103 | 0.00296853 | 0.00294778 | |
| 3 | 0.0050863 | 0.00320241 | 3.09E-07 | 0.265008 | 0.0046316 | 0.00353411 | 0.00352839 | 0.00342345 | |
| 4 | 0.00752974 | 0.00485125 | 3.09E-07 | 0.0361383 | 0.00770687 | 0.00546446 | 0.00543714 | 0.00484289 | |
| 5 | 0.012263 | 0.0104767 | 3.09E-07 | 0.0102962 | 0.0213922 | 0.0123653 | 0.012094 | 0.00798934 | |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | Log(1/Distance) | | | | | |
| | A | E | H | L | M | N | T | V | |
| 1 | -2.3867376 | -2.583869 | -6.5094081 | 3.1712612 | -2.4404321 | -2.5449556 | -2.5452542 | -2.5462461 | |
| 2 | -2.3685451 | -2.5667279 | -6.5094067 | 0.7763757 | -2.4202164 | -2.527093 | -2.5274586 | -2.5305049 | |
| 3 | -2.293598 | -2.4945231 | -6.5093983 | -0.576741 | -2.334269 | -2.4517199 | -2.4524234 | -2.465536 | |
| 4 | -2.12322 | -2.3141463 | -6.5093828 | -1.4420323 | -2.113122 | -2.2624527 | -2.2646295 | -2.3148954 | |
| 5 | -1.9114033 | -1.9797755 | -6.5093632 | -1.987323 | -1.6697445 | -1.9077953 | -1.91743 | -2.0974891 | |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | Log(1/Distance) + Bias, bias = 8 | | | | | |
| | A | E | H | L | M | N | T | V | |
| 1 | 5.61326238 | 5.41613104 | 1.49059189 | 11.1712612 | 5.55956791 | 5.45504444 | 5.4547458 | 5.45375395 | |
| 2 | 5.63145493 | 5.43327207 | 1.49059329 | 8.7763757 | 5.5797836 | 5.47290704 | 5.4725414 | 5.46949507 | |
| 3 | 5.70640197 | 5.50547693 | 1.49060171 | 7.42325898 | 5.66573105 | 5.54828006 | 5.5475765 | 5.53446399 | |
| 4 | 5.87677998 | 5.68585366 | 1.49061715 | 6.55796772 | 5.88687803 | 5.73754725 | 5.7353705 | 5.6851046 | |
| 5 | 6.08859673 | 6.02022451 | 1.4906368 | 6.01267697 | 6.33025545 | 6.09220466 | 6.0825699 | 5.9025109 | |

**Table 8. Classification distance data for the letter L.**

| phase rate | | | | 1/Distance | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | A | E | H | L | M | N | T | V |
| 1 | 0.0133988 | 0.0946761 | 3.09E-07 | 0.00361589 | 1.24E+06 | 0.101889 | 0.112635 | 0.0131742 |
| 2 | 0.0133609 | 0.0948219 | 3.09E-07 | 0.00361582 | 494414 | 0.102092 | 0.111954 | 0.0131341 |
| 3 | 0.0132056 | 0.0954139 | 3.09E-07 | 0.00361536 | 208.385 | 0.102914 | 0.109201 | 0.0129703 |
| 4 | 0.0128465 | 0.0968877 | 3.09E-07 | 0.00361213 | 25.4189 | 0.104886 | 0.103215 | 0.0125988 |
| 5 | 0.0122249 | 0.103763 | 3.09E-07 | 0.00357514 | 5.85546 | 0.112228 | 0.0973167 | 0.0120145 |
| 6 | 0.0112434 | 0.139539 | 3.09E-07 | 0.00343947 | 1.33137 | 0.155953 | 0.0982922 | 0.011427 |
| 7 | 0.0106432 | 0.163996 | 3.10E-07 | 0.00337319 | 0.52181 | 0.217715 | 0.092604 | 0.0110769 |
| 8 | 0.00853432 | 0.0444666 | 3.11E-07 | 0.0031999 | 0.042277 | 0.0969703 | 0.030889 | 0.00888023 |
| | | | | | | | | |
| | | | | Log(1/Distance) | | | | |
| | A | E | H | L | M | N | T | V |
| 1 | -1.8729341 | -1.0237596 | -6.5093969 | -2.4417848 | 6.09447812 | -0.9918727 | -0.9483266 | -1.8802757 |
| 2 | -1.8741643 | -1.0230913 | -6.5093969 | -2.4417932 | 3.69409076 | -0.9910083 | -0.9509604 | -1.8815997 |
| 3 | -1.8792419 | -1.0203884 | -6.5093969 | -2.4418485 | 2.31886645 | -0.9875255 | -0.9617734 | -1.88705 |
| 4 | -1.8912152 | -1.0137314 | -6.5093969 | -2.4422366 | 1.40515675 | -0.9792825 | -0.9862572 | -1.8996708 |
| 5 | -1.9127547 | -0.9839575 | -6.5093941 | -2.4467069 | 0.76756102 | -0.9498988 | -1.0118126 | -1.9191932 |
| 6 | -1.9491023 | -0.8553044 | -6.5093478 | -2.4635085 | 0.12429877 | -0.8070063 | -1.0074809 | -1.9420678 |
| 7 | -1.9729278 | -0.7851667 | -6.5092818 | -2.4719592 | -0.2824876 | -0.6621116 | -1.0333703 | -1.9555818 |
| 8 | -2.0688311 | -1.3519661 | -6.5071977 | -2.4948636 | -1.3738958 | -1.0133613 | -1.5101962 | -2.0515758 |
| | | | | | | | | |
| | | | | Log (1/Distance) + Bias, bias = 8 | | | | |
| | A | E | H | L | M | N | T | V |
| 1 | 6.1270659 | 6.9762404036 | 1.490603312 | 5.55821521 | 14.0944781 | 7.00812273 | 7.05167336 | 6.11972425 |
| 2 | 6.1258357 | 6.9769086 | 1.490603312 | 5.5582068 | 11.6940908 | 7.00899171 | 7.04903962 | 6.11840032 |
| 3 | 6.1207581 | 6.9796116 | 1.490603312 | 5.55815155 | 10.3188665 | 7.01247446 | 7.03822662 | 6.11295002 |
| 4 | 6.1087848 | 6.9862686 | 1.490603312 | 5.55776337 | 9.40515675 | 7.02071752 | 7.01374282 | 6.10032918 |
| 5 | 6.0872453 | 7.0160425 | 1.490606592 | 5.55329305 | 8.76756102 | 7.05010122 | 6.98818737 | 6.0808068 |
| 6 | 6.0508977 | 7.1446956 | 1.490652223 | 5.53649153 | 8.12429877 | 7.19299373 | 6.99251906 | 6.05793223 |
| 7 | 6.0270722 | 7.2148332 | 1.49071818 | 5.5280408 | 7.71751247 | 7.33788835 | 6.96662975 | 6.04441824 |
| 8 | 5.9311689 | 6.6480339 | 1.49280228 | 5.50513641 | 6.62610416 | 6.98663874 | 6.48980385 | 5.94842421 |

**Table 9. Classification distance data for the letter M.**

| phase rate | 1/Distance | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | A | E | H | L | M | N | T | Y | |
| 1 | 0.00887322 | 0.363559 | 3.10E-07 | 0.00284366 | 0.101785 | 77494.5 | 0.0801648 | 0.0101704 | |
| 2 | 0.00886934 | 0.344182 | 3.10E-07 | 0.00284688 | 0.100434 | 296.694 | 0.0789872 | 0.0101564 | |
| 3 | 0.00886233 | 0.270011 | 3.10E-07 | 0.00286669 | 0.094913 | 10.731 | 0.0737455 | 0.0100949 | |
| 4 | 0.00894162 | 0.148595 | 3.10E-07 | 0.0029712 | 0.0839635 | 0.995178 | 0.0605991 | 0.00994482 | |
| 5 | 0.00900543 | 0.155365 | 3.10E-07 | 0.00297961 | 0.0880148 | 1.08094 | 0.0623955 | 0.00999803 | |
| 6 | 0.00817272 | 0.94098 | 3.10E-07 | 0.00255613 | 0.073433 | 0.775369 | 0.0830417 | 0.00998286 | |
| | | | | | | | | | |
| | | | | Log( 1/Distance) | | | | | |
| | A | E | H | L | M | N | T | Y | |
| 1 | -2.0519188 | -0.4394251 | -6.5085487 | -2.5461223 | -0.9923162 | 4.88927088 | -1.0960163 | -1.992662 | |
| 2 | -2.0521087 | -0.4632118 | -6.5085234 | -2.5456308 | -0.9981192 | 2.47230876 | -1.1024433 | -1.9932602 | |
| 3 | -2.0524521 | -0.5686185 | -6.5084114 | -2.5426193 | -1.0226743 | 1.03064019 | -1.1322645 | -1.995898 | |
| 4 | -2.0485838 | -0.8279958 | -6.5081343 | -2.5270681 | -1.0759095 | -0.0020992 | -1.2175338 | -2.0024031 | |
| 5 | -2.045495 | -0.8086468 | -6.5081636 | -2.5258406 | -1.0554443 | 0.03380159 | -1.2048467 | -2.0000856 | |
| 6 | -2.0876334 | -0.0264196 | -6.5086691 | -2.5924171 | -1.1341087 | -0.1104916 | -1.0807038 | -2.0075083 | |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | Log( 1/Distance) + Bias, bias = 8 | | | | | |
| | A | E | H | L | M | N | T | Y | |
| 1 | 5.94808125 | 7.5605749 | 1.49145135 | 5.45387767 | 7.00768378 | 12.8892709 | 6.90398371 | 6.00733803 | |
| 2 | 5.9478913 | 7.53678815 | 1.49147656 | 5.45436916 | 7.00188076 | 10.4723088 | 6.89755672 | 6.0067398 | |
| 3 | 5.94754792 | 7.43138146 | 1.49158859 | 5.45738073 | 6.9773257 | 9.03064019 | 6.86773552 | 6.0041202 | |
| 4 | 5.95141621 | 7.1720042 | 1.49186574 | 5.47293189 | 6.92409053 | 7.99790077 | 6.78246617 | 5.99759693 | |
| 5 | 5.95450445 | 7.19135319 | 1.49183636 | 5.47415942 | 6.94455571 | 8.03380159 | 6.79515327 | 5.99991444 | |
| 6 | 5.91236662 | 7.97358039 | 1.49133087 | 5.40758294 | 6.86589127 | 7.88950843 | 6.91929623 | 5.99249166 | |

**Table 10. Classification distance data for the Letter N.**

| phase rate | | | | 1/Distance | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | A | E | H | L | M | N | T | Y |
| 1 | 0.0155431 | 0.116353 | 3.09E-07 | 0.00284154 | 0.112747 | 0.0802938 | 535366 | 0.0206525 |
| 2 | 0.0154811 | 0.117892 | 3.09E-07 | 0.00284167 | 0.113745 | 0.0810651 | 2101.89 | 0.0205357 |
| 3 | 0.0152189 | 0.124819 | 3.09E-07 | 0.00284213 | 0.118074 | 0.0844706 | 83.8281 | 0.0200466 |
| 4 | 0.0145624 | 0.14572 | 3.09E-07 | 0.00284232 | 0.129655 | 0.0941466 | 8.89827 | 0.0188576 |
| 5 | 0.0133814 | 0.203071 | 3.09E-07 | 0.00283821 | 0.151999 | 0.116826 | 1.68609 | 0.0168391 |
| 6 | 0.0117415 | 0.383134 | 3.09E-07 | 0.00280136 | 0.165796 | 0.162825 | 0.502987 | 0.0143716 |
| 7 | 0.0101518 | 1.03152 | 3.10E-07 | 0.00262789 | 0.106712 | 0.220626 | 0.250935 | 0.0127316 |
| | | | | | | | | |
| | | | | | | | | |
| | | | | Log(1/Distance) | | | | |
| | A | E | H | L | M | N | T | Y |
| 1 | -1.8084624 | -0.9342224 | -6.509366 | -2.546462 | -0.947895 | -1.095318 | 5.72865079 | -1.6850274 |
| 2 | -1.8101982 | -0.9285157 | -6.509366 | -2.546426 | -0.9440677 | -1.0911661 | 3.32260998 | -1.6874905 |
| 3 | -1.8176167 | -0.9037193 | -6.509366 | -2.5463561 | -0.9278457 | -1.0732944 | 1.92338962 | -1.6979593 |
| 4 | -1.836767 | -0.8364808 | -6.5093646 | -2.546327 | -0.8872107 | -1.0261954 | 0.94930558 | -1.7245136 |
| 5 | -1.8734984 | -0.6923521 | -6.5093604 | -2.5469555 | -0.8181593 | -0.9324605 | 0.22688075 | -1.7736811 |
| 6 | -1.9302764 | -0.4166493 | -6.5093478 | -2.5526311 | -0.780426 | -0.7882789 | -0.2984432 | -1.8424949 |
| 7 | -1.9934569 | 0.01347765 | -6.5092159 | -2.5803928 | -0.9717867 | -0.6563433 | -0.6004388 | -1.895117 |
| | | | | | | | | |
| | | | | Log(1/Distance) + Bias, bias = 8 | | | | |
| | A | E | H | L | M | N | T | Y |
| 1 | 6.19153764 | 7.06577759 | 1.49063399 | 5.45355377 | 7.05210499 | 6.90468201 | 13.7286508 | 6.31497263 |
| 2 | 6.18980182 | 7.07148434 | 1.49063399 | 5.45357364 | 7.05593232 | 6.90883392 | 11.32261 | 6.31250951 |
| 3 | 6.18238326 | 7.09628807 | 1.49063399 | 5.45364394 | 7.07215428 | 6.92670558 | 9.92338962 | 6.30204072 |
| 4 | 6.16323296 | 7.16351916 | 1.49063539 | 5.45367297 | 7.11278927 | 6.97380464 | 8.94930558 | 6.27548642 |
| 5 | 6.12650155 | 7.30764791 | 1.4906396 | 5.45304453 | 7.18184073 | 7.06753951 | 8.22688075 | 6.22631888 |
| 6 | 6.06972358 | 7.58335069 | 1.49065223 | 5.44736892 | 7.21957405 | 7.21172109 | 7.70155676 | 6.15750512 |
| 7 | 6.00654305 | 8.01347765 | 1.49078412 | 5.41960718 | 7.02821326 | 7.34365669 | 7.39956124 | 6.10488299 |

**Table 11. Classification distance data for the letter T.**

| phase rate | | | | 1/Distance | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | A | E | H | L | M | N | T | V |
| 1 | 0.07960099 | 0.0103988 | 3.09E-07 | 0.00283564 | 0.0131712 | 0.0101676 | 0.0206498 | 278144 |
| 2 | 0.07926627 | 0.0103425 | 3.09E-07 | 0.00283132 | 0.0130854 | 0.0101129 | 0.0204905 | 1072.71 |
| 3 | 0.07763332 | 0.0100951 | 3.09E-07 | 0.00281204 | 0.0127107 | 0.00987235 | 0.0197973 | 39.5911 |
| 4 | 0.07220052 | 0.00940398 | 3.09E-07 | 0.00275513 | 0.011683 | 0.00920018 | 0.0179163 | 3.406334 |
| 5 | 0.05629248 | 0.00786131 | 3.09E-07 | 0.00260727 | 0.00948442 | 0.00769948 | 0.0140084 | 0.41117 |
| 6 | 0.02443641 | 0.0049593 | 3.09E-07 | 0.00220235 | 0.00567014 | 0.00487341 | 0.00769689 | 0.0478512 |
| 7 | 0.00171384 | 0.00093083 | 3.09E-07 | 0.00079002 | 0.00097891 | 0.00092406 | 0.00110884 | 0.00184869 |
| 8 | 4.87E-08 | 4.85E-08 | 4.19E-08 | 4.85E-08 | 4.85E-08 | 4.85E-08 | 4.86E-08 | 4.87E-08 |
| 9 | 0.0300279 | 0.0115858 | 3.10E-07 | 0.00754456 | 0.0242525 | 0.0134524 | 0.0168609 | 0.0157206 |
| | | | | | | | | |
| | | | | Log( 1/Distance) | | | | |
| | A | E | H | L | M | N | T | V |
| 1 | -1.0990329 | -1.9830168 | -6.5093997 | -2.5473489 | -1.8803747 | -1.9927815 | -1.6850842 | 5.4442697 |
| 2 | -1.1009311 | -1.9853745 | -6.5093997 | -2.548011 | -1.883213 | -1.9951243 | -1.6884474 | 3.0304823 |
| 3 | -1.1099525 | -1.9958894 | -6.5093997 | -2.5509785 | -1.8958305 | -2.0055795 | -1.7033394 | 1.5975975 |
| 4 | -1.1141315 | -2.0266883 | -6.5093997 | -2.5598579 | -1.9324456 | -2.0362037 | -1.7467517 | 0.5322879 |
| 5 | -1.2495317 | -2.1045051 | -6.5093997 | -2.583814 | -2.0229892 | -2.1135386 | -1.8536115 | -0.385978 |
| 6 | -1.6119628 | -2.3045796 | -6.5094053 | -2.6571137 | -2.2464062 | -2.312167 | -2.1136847 | -1.320107 |
| 7 | -2.7660297 | -3.0311292 | -6.5095246 | -3.1023597 | -3.009259 | -3.0342998 | -2.9551311 | -2.733135 |
| 8 | -7.3124924 | -7.3141463 | -7.3774099 | -7.3139647 | -7.3140246 | -7.3141759 | -7.3136355 | -7.312356 |
| 9 | -1.522475 | -1.936074 | -6.5092944 | -2.1223661 | -1.6152435 | -1.8712002 | -1.7731192 | -1.803531 |
| | | | | | | | | |
| | | | | Log( 1/Distance) + Bias, bias = 8 | | | | |
| | A | E | H | L | M | N | T | V |
| 1 | 6.90096708 | 6.01698323 | 1.49060031 | 5.45265109 | 6.11962534 | 6.00721845 | 6.31491585 | 13.4442697 |
| 2 | 6.89906886 | 6.01462553 | 1.49060031 | 5.45198896 | 6.116787 | 6.00487571 | 6.31155256 | 11.0304823 |
| 3 | 6.89004749 | 6.00411063 | 1.49060031 | 5.44902149 | 6.10416947 | 5.99442054 | 6.29660596 | 9.59759757 |
| 4 | 6.85856848 | 5.97331177 | 1.49060031 | 5.4401421 | 6.06755438 | 5.96379632 | 6.25324833 | 8.53228799 |
| 5 | 6.75046828 | 5.89549492 | 1.49060031 | 5.41618601 | 5.97701078 | 5.88646114 | 6.14638853 | 7.61402142 |
| 6 | 6.38803723 | 5.69542038 | 1.49059947 | 5.34288634 | 5.75359378 | 5.68783295 | 5.88631528 | 6.67989283 |
| 7 | 5.23397027 | 4.96887084 | 1.49047539 | 4.89764028 | 4.99074099 | 4.96570017 | 5.04486888 | 5.26686409 |
| 8 | 0.68750756 | 0.68585366 | 0.62259011 | 0.68603535 | 0.68597539 | 0.68582411 | 0.68636453 | 0.68764309 |

**Table 12. Classification distance data for the letter V.**

| phase rate | distance^(1/7) |
|---|---|
| 0.00000 | 0.00000 |
| 1.00000 | 0.15612 |
| 2.00000 | 0.34449 |
| 3.00000 | 0.54577 |
| 4.00000 | 0.75172 |
| 5.00000 | 0.95311 |
| 6.00000 | 1.10884 |

**Table 13. Data for A regression graph.**

| phase rate | distance^(1/7) |
|---|---|
| 0.00000 | 0.00000 |
| 1.00000 | 0.12902 |
| 2.00000 | 0.28422 |
| 3.00000 | 0.44699 |
| 4.00000 | 0.60485 |
| 5.00000 | 0.74531 |
| 6.00000 | 0.89273 |

**Table 14. Data for E regression graph.**

| phase rate | distance^(1/7) |
|---|---|
| 0.00000 | 0.00000 |
| 1.00000 | 2.88003 |
| 2.00000 | 5.95398 |
| 3.00000 | 7.78888 |

**Table 15. Data for H regression graph.**

| phase rate | distance^(1/7) |
|---|---|
| 0.00000 | 0.00000 |
| 1.00000 | 0.35234 |
| 2.00000 | 0.77462 |
| 3.00000 | 1.20890 |
| 4.00000 | 1.60696 |
| 5.00000 | 1.92266 |

**Table 16. Data for L regression graph**

| phase rate | distance^(1/7) |
|---|---|
| 0.00000 | 0.00000 |
| 1.00000 | 0.13474 |
| 2.00000 | 0.29667 |
| 3.00000 | 0.46637 |
| 4.00000 | 0.62989 |
| 5.00000 | 0.77687 |
| 6.00000 | 0.95994 |
| 7.00000 | 1.09738 |

**Table 17. Data for M regression graph.**

| phase rate | distance^(1/7) |
|---|---|
| 0.00000 | 0.00000 |
| 1.00000 | 0.20023 |
| 2.00000 | 0.44342 |
| 3.00000 | 0.71247 |
| 4.00000 | 1.00069 |
| 5.00000 | 0.98894 |
| 6.00000 | 1.03701 |

**Table 18. Data for N regression graph.**

| phase rate | distance^(1/7) |
|---|---|
| 0.00000 | 0.00000 |
| 1.00000 | 0.15192 |
| 2.00000 | 0.33523 |
| 3.00000 | 0.53117 |
| 4.00000 | 0.73179 |
| 5.00000 | 0.92809 |
| 6.00000 | 1.10315 |
| 7.00000 | 1.21836 |

**Table 19. Data for T regression graph.**

| phase rate | distance^(1/7) |
|---|---|
| 0.00000 | 0.00000 |
| 1.00000 | 0.16682 |
| 2.00000 | 0.36904 |
| 3.00000 | 0.59125 |
| 4.00000 | 0.83938 |
| 5.00000 | 1.13538 |
| 6.00000 | 1.54378 |

**Table 20. Data for V regression graph.**

# APPENDIX C

The following pages contain the computer programs created for this experiment. The first few pages contain code for the image header file and routines that read and write image files. The remaining code is divided into two sections; functions belonging to the program moment_main, and functions belonging to the program classifier. The moment_main program allowed the user to generate the character test images with a specific amount of degradation applied to it. The program classifier was then implemented to classify the test characters. Each main program is listed, followed by its supporting functions. The last few pages contain the makefiles for the two main programs. These files are written to compile a main program with its supporting functions in the UNIX environment.

```
/* IMAGE.h header file */

#ifndef __IMAGE_LOADED
#define __IMAGE_LOADED       1

/*  IMAGE - V1.0 - Prototypes for image processing functions     */

#include <stdio.h>
#include <math.h>

#define MX_BANDS     256

#define TRUE  1
#define FALSE 0

#define _abs( x ) (x > 0) ? (x) : (-x)
#define _min( a, b ) (a <= b) ? (a) : (b)
#define _max( a, b ) (a >= b) ? (a) : (b)

struct PICTURE_OPTIONS {
   FILE               *file_pointer;
   short              element_size;
   short              number_of_bands;
   long               number_of_columns;
   long               number_of_rows;
   long               header_length;
};

int getpixel( struct PICTURE_OPTIONS picops, long row, long col,
              unsigned char band_mask[], long number_of_elements,
              unsigned char *data_location );

int putpixel( struct PICTURE_OPTIONS picops, long row, long col,
              unsigned char band_mask[], long number_of_elements,
              unsigned char *data_location );

int open_raw_file( long number_of_rows, long number_of_columns,
                   long number_of_bands, long element_size,
                   const char *filename, const char *mode, long create_value,
                   struct PICTURE_OPTIONS *picops );

void switch_longword( );

void switch_word( );


#endif                                  /* __IMAGE_LOADED */
```

```
#include "image.h"

/*********************************************************************

function:       open_raw_file

description:    this routine will open a raw image file with user
                specifieddimensions for read or write mode.  the read mode implies that
                the file already exists and the picture options data structure is to be
                filled.  the write mode implies that the file is to be created.  the
                write mode will create a file of the size specified in the user supplied
                data when this routine is called and fill it with the specified
                initialization value.

variables:      number_of_rows (longword, value)
                indicates the number of rows in the raw image data
                number_of_columns (longword, value)
                indicates the number of columns in the raw image data

                number_of_bands (longword, value)
                indicates the number of bands in the raw image data
                and assumes the bands are interleaved by line

                element_size (longword, value)
                indicate the number of byte per pixel (valid values are either1 or 2)

                filename (array of char, address)
                the address of the character string in the calling program
                where the name of the file to be opened is specified

                mode (char, address)
                the open mode of the file, can be "w" (119) for a write or
                can be "r" (114) for a read (a value of 114 is assumed
                if no value or an erroneous value is past)

                create_value (longword, value)
                the create_value specifies the value which should
                initialize a file when it is created in the "w" mode, a -1
                will cause no initialization to occur and a 0 is the default

                picops (struct PICTURE_OPTIONS, address)
                the picture options structure which will be filled with image specific
                information such as number of bands, number of rows and columns, element
                size, header length, and the file pointer

return value:   1 if success, NULL if not

author:         carl salvaggio
date:           12/18/91
modifications:

*********************************************************************/

int open_raw_file( long number_of_rows, long number_of_columns,
                   long number_of_bands, long element_size,
                   const char *filename, const char *mode, long
               create_value, struct PICTURE_OPTIONS *picops )
```

```
{
   FILE *fp;
   int status;
   int rows;
   int element;
   int elements_per_row;
   unsigned char *buffer;

/************************************************************************
   open a new file if mode is "w" (119) and initialize as specified
************************************************************************/
   if ( *mode == 119 ) {
      fp = fopen( filename, "wb" );
      if ( fp == NULL ) {
         return( 0 );
      }
      picops->file_pointer = fp;

      if ( create_value >= 0 ) {
         if ( element_size == 2 ) {
            elements_per_row = number_of_columns *
                               number_of_bands *
                               element_size;
         }
         else {
            elements_per_row = number_of_columns *
                               number_of_bands;
         }
         buffer = (unsigned char *) calloc( (size_t) elements_per_row,
                                            (size_t) sizeof( unsigned char ) );

         if ( create_value != 0 ) {
            for ( element=0; element<elements_per_row; element++ ) {
               *(buffer+element) = create_value;
            }
         }

         for ( rows=1; rows<=number_of_rows; rows++ ) {
            status = fwrite( buffer,
                             (size_t) 1,
                             (size_t) elements_per_row,
                             (picops->file_pointer) );
            if ( status == 0 ) {
               return( 0 );
            }
         }
      }
   }

/************************************************************************
   open an existing file if mode is "r" (114) and fill the picture options
************************************************************************/
   else {
      fp = fopen( filename, "rb" );
      if ( fp == NULL ) {
         return( 0 );
      }
```

```
      picops->file_pointer = fp;
   }

/************************************************************************
   transfer the relevant header data to the picops structure
************************************************************************/
   (picops->number_of_bands)   = number_of_bands;
   (picops->number_of_columns) = number_of_columns;
   (picops->number_of_rows)    = number_of_rows;
   (picops->element_size)      = element_size;
   (picops->header_length)     = 0;

   return( 1 );

}
```

```
#include "image.h"

/*********************************************************************

function:       getpixel

description:    this routine will return a stream of data from an n-band image of
                arbitrary size.  the starting point of the stream is denoted by the row
                and col number.  the convention for this indexing scheme is that the
                upper left hand corner of the image has coordinate (0,0) and the
                row corresponds to the y-axis of the cartesian coordinate
                system while the col refers to the x-axis.

variables:      picops (struct PICTURE_OPTIONS, value)
                the picture options structure which contains image
                specific information such as number of bands, number
                of columns, element size, header length, and the file pointer

                row (longword, value)
                the y-axis coordinate of the starting point for the
                data stream that is to be returned to the calling
                routine (starts at 0)

                col (longword, value)
                the x-axis coordinate of the starting point for the
                data stream that is to be returned to the calling
                routine (starts at 0)

                band_mask[] (array of unsigned char, address)
                an array of flags (1 is on, 0 is off) indicating which
                bands are to be grabbed from the open file, array
                element 0 corresponds to band 1, element 1 to band 2,
                and so on to element (MX_BANDS-1) to band MX_BANDS

                number_of_elements (longword, value )
                the number of data elements to be grabbed across the row
                starting at the element specified by row and col
                data_location (address of data structure in calling
                routine) the address of the memory location to place the
                returned data stream at

return value:   number of bytes grabbed if success, NULL if read is past end-of-file

author:         carl salvaggio
date:           1/28/91
modifications:          1/29/91 - to work with picture options structure
                        3/14/91 - to correct single band grab bug

*********************************************************************/

int getpixel( struct PICTURE_OPTIONS picops, long row, long col,
              unsigned char band_mask[], long number_of_elements,
              unsigned char *data_location )
{
   int                  status;
   long                 offset;
   int                  number_of_bytes;
```

```
   int                  band, number_of_bands_on;
   long                 element;
   unsigned char        *data_location_ptr;

   data_location_ptr = data_location;
   number_of_bytes = number_of_elements * (picops.element_size);

#ifndef VMS
   number_of_bands_on = 0;
   for ( band=1; band<=(picops.number_of_bands); band++ ) {
      if ( band_mask[band-1] != 0 ) {
         number_of_bands_on++;
      }
   }
#endif

   for ( band=1; band<=(picops.number_of_bands); band++ ) {
      if ( band_mask[band-1] != 0 ) {
         offset = ( row * (picops.number_of_columns) * (picops.number_of_bands) +
                  (picops.number_of_columns) * (band-1) + col ) *
                  (picops.element_size) + (picops.header_length);
         status = fseek( (picops.file_pointer), offset, 0 );
         if ( status != 0 ) {
            return( 0 );
         }
         status = fread( data_location, (size_t) 1, (size_t)
                     number_of_bytes, (picops.file_pointer) );
         if ( status != number_of_bytes ) {
            return( 0 );
         }
         data_location = data_location + number_of_bytes;
      }
   }

#ifndef VMS
   if ( picops.element_size == 2 ) {
      data_location = data_location_ptr;
      for ( element=0;
            element<(number_of_elements*number_of_bands_on);  element++ ) {
         switch_word( data_location );
         data_location = data_location + 2;
      }
   }
#endif

   return( status );
}
```

```
#include "image.h"

/*********************************************************************

function:       putpixel

description:    this routine will place a stream of data to an n-band image of arbitrary
                size.  the starting point of the stream is denoted by the row and col
                number.  the convention for this indexing scheme is that the upper left
                hand corner of the image has coordinate (0,0) and the row
                corresponds to the y-axis of the cartesian coordinate system while the
                col refers to the x-axis.

variables:      picops (struct PICTURE_OPTIONS, value)
                the picture options structure which contains image
                specific information such as number of bands, number
                of columns, element size, header length, and the file pointer

                row (longword, value)
                the y-axis coordinate of the starting point for the
                data stream that is to be placed from the calling
                routine (starts at 0)

                col (longword, value)
                the x-axis coordinate of the starting point for the
                data stream that is to be placed from the calling
                routine (starts at 0)

                band_mask[] (array of unsigned char, address)
                an array of flags (1 is on, 0 is off) indicating which
                bands are to be placed to the open file, array
                element 0 corresponds to band 1, element 1 to band 2,
                and so on to element (MX_BANDS-1) to band MX_BANDS

                number_of_elements (longword, value )
                the number of data elements to be placed across the row
                starting at the element specified by row and col

                data_location (address of data structure in calling
                routine) the address of the memory location to grab the data stream from
                in the calling routine

return value:   number of bytes written if success, NULL if the write encounters a
                problem

author:         carl salvaggio
date:           1/28/91
modifications:          1/29/91 - to work with picture options structure

*********************************************************************/

int putpixel( struct PICTURE_OPTIONS picops, long row, long col,
              unsigned char band_mask[], long number_of_elements,
              unsigned char *data_location )
{
   int                  status;
   long                 offset;
```

```
   int                  number_of_bytes;
   int                  band, number_of_bands_on;
   long                 element;
   unsigned char        *data_location_ptr;

#ifndef VMS
   number_of_bands_on = 0;
   for ( band=1; band<=(picops.number_of_bands); band++ ) {
      if ( band_mask[band-1] != 0 ) {
         number_of_bands_on++;
      }
   }
#endif

#ifndef VMS
   if ( picops.element_size == 2 ) {
      data_location_ptr = data_location;
      for ( element=0;
            element<(number_of_elements*number_of_bands_on);  element++ ) {
         switch_word( data_location );
         data_location = data_location + 2;
      }
      data_location = data_location_ptr;
   }
#endif

   number_of_bytes = number_of_elements * (picops.element_size);

   for ( band=1; band<=(picops.number_of_bands); band++ ) {
      if ( band_mask[band-1] != 0 ) {
         offset = ( row * (picops.number_of_columns) * (picops.number_of_bands) +
                  (picops.number_of_columns) * (band-1) + col ) *
                  (picops.element_size) + (picops.header_length);
         status = fseek( (picops.file_pointer), offset, 0 );
         if ( status == EOF ) {
            return( 0 );
         }
         status = fwrite( data_location, 1, number_of_bytes, (picops.file_pointer) );
         if ( status == 0 ) {
            return( 0 );
         }
         data_location = data_location + number_of_bytes;
     }
   }

   return( status );
}
```

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

routine:        switch_longword

description:    this routine will perform a byte reversal on a longword to allow input
                data on a PC and most UNIX systems to be interpreted properly (byte
                reversal occurs as ABCD --> DCBA

variables:      longword (address of a longword)
                the address of the longword which is to undergo byte
                reversal

return value:   none

author:         michael heath
date:           1/28/91

*************************************************************************/

void switch_longword( char *longword )
{
   char temp;

   temp = *(longword+3);
   *(longword+3) = *longword;
   *longword = temp;
   temp = *(longword+2);
   *(longword+2) = *(longword+1);
   *(longword+1) = temp;
}
```

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

routine:        switch_word

description:    this routine will perform a byte reversal on a word
                to allow input data on a PC and most UNIX systems to be
                interpreted properly (byte reversal occurs as AB --> BA

variables:      word (address of a word)
                the address of the word which is to undergo byte
                reversal

return value:   none

author:         michael heath
date:           1/28/91

*************************************************************************/

void switch_word( char *word )
{
   char temp;

   temp = *(word + 1);
   *(word + 1) = *word;
   *word = temp;
}
```

```
/*************************************************************************
   main program: moment_main.c

   description: This program acts as the controller for all functions
                that it calls.  User interface is achieved through a
                main menu environment where the user can select the
                desired task to be performed.  Any selection from the
                main menu results in a function call that will
                perform the desired task, once that task has been
                completed, the user is returned to the main menu to
                select another option.  All data arrays are passed
                between functions through this main program.

   return value: none

   author: Adam Hanson
           Center for Imaging Science, RIT

   date: 8/19/92
*************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define SIZE 64

void load_image(double temp_re[SIZE][SIZE], double temp_im[SIZE][SIZE]);
void save_parts(int int_re[SIZE][SIZE], int int_im[SIZE][SIZE]);
void save_mag(int int_mag[SIZE][SIZE]);
void scale_parts(double mult_re[SIZE][SIZE], double mult_im[SIZE][SIZE],
                 int int_re[SIZE][SIZE], int int_im[SIZE][SIZE]);
void scale_mag(double mult_re[SIZE][SIZE], double mult_im[SIZE][SIZE],
               int int_mag[SIZE][SIZE]);
void transform(double temp_re[SIZE][SIZE], double temp_im[SIZE][SIZE],
               long sign, long size);
void filter(double obj_re[SIZE][SIZE], double obj_im[SIZE][SIZE],
            double ftr_re[SIZE][SIZE], double ftr_im[SIZE][SIZE],
            double mult_re[SIZE][SIZE], double mult_im[SIZE][SIZE]);
void phase_maker(double ftr_re[SIZE][SIZE], double ftr_im[SIZE][SIZE]);

main()
{
   double obj_re[SIZE][SIZE], obj_im[SIZE][SIZE];
   double ftr_re[SIZE][SIZE], ftr_im[SIZE][SIZE];
   double mult_re[SIZE][SIZE], mult_im[SIZE][SIZE];
   int int_re[SIZE][SIZE], int_im[SIZE][SIZE], int_mag[SIZE][SIZE];
   int option, choice, ans, flag, size, sign;

/*** prompt user for desired operation from main menu ***/
   size = SIZE;
   flag = 1;
   while (flag == 1){
      printf("\n\n\t\t(1) Load object image");
      printf("\n\t\t(2) Load impulse response");
      printf("\n\t\t(3) Save image");
      printf("\n\t\t(4) Scale image");
```

```
        printf("\n\t\t(5) Transform image");
        printf("\n\t\t(6) Filter object and impulse response ");
        printf("\n\t\t(7) Create chirp impulse response ");
        printf("\n\t\t(8) Quit");
        printf("\n\n\t\tEnter option...");
        scanf("%d", &option);
        if(option == 1) load_image(obj_re, obj_im);
        if(option == 2) load_image(ftr_re, ftr_im);
        if(option == 3){
           printf("\nSave (1) real/imaginary parts or (2) magnitude: ");
           scanf("%d", &choice);
           if(choice == 1) save_parts(int_re, int_im);
           if(choice == 2) save_mag(int_mag);
        }
        if(option == 4){
           printf("\n\t\tScale:");
           printf("\n\t\t\t(1) object");
           printf("\n\t\t\t(2) impulse response");
           printf("\n\t\t\t(3) filtered object");
           printf("\n\n\t\t\tEnter option...");
           scanf("%d", &option);
           printf("\nScale (1) real/imaginary parts or (2) magnitude: ");
           scanf("%d", &choice);
           if(option==1 && choice==1) scale_parts(obj_re, obj_im,
                                          int_re, int_im);
           if(option==2 && choice==1) scale_parts(ftr_re, ftr_im,
                                          int_re, int_im);
           if(option==3 && choice==1) scale_parts(mult_re, mult_im,
                                          int_re, int_im);
           if(option==1 && choice==2) scale_mag(obj_re, obj_im, int_mag);
           if(option==2 && choice==2) scale_mag(ftr_re, ftr_im, int_mag);
           if(option==3 && choice==2) scale_mag(mult_re, mult_im,
                                                        int_mag);
        }
        if(option == 5){
           printf("\n\t\tTransform:");
           printf("\n\t\t\t(1) object");
           printf("\n\t\t\t(2) impulse response");
           printf("\n\t\t\t(3) filtered object");
           printf("\n\n\t\t\tEnter option...");
           scanf("%d", &choice);
           printf("\nChoose transform type; (-1) forward, (1) inverse: ");
           scanf("%d", &sign);
           if(choice == 1) transform(obj_re, obj_im, sign, size);
           if(choice == 2) transform(ftr_re, ftr_im, sign, size);
           if(choice == 3) transform(mult_re, mult_im, sign, size);
        }
        if(option == 6) filter(obj_re, obj_im, ftr_re, ftr_im,
                               mult_re, mult_im);
        if(option == 7) phase_maker(ftr_re, ftr_im);
        if(option == 8) flag = 0;
     }
}
```

```
/*****************************************************************************
   function: load_image.c

   description: This function opens an input image file in a raw format
                and places the digitial counts of the image into
                temp_re.  Since the input image is completely real with
                no imaginary part, the temp_im array is filled with
                zeros.

   return value: none

   author: Adam Hanson
             Center for Imaging Science, RIT

   date: 8/19/92
*****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "image.h"

#define SIZE 64

void load_image(double temp_re[SIZE][SIZE], double temp_im[SIZE][SIZE])
{
   struct PICTURE_OPTIONS picops_in;
   long number_of_rows, number_of_columns, number_of_bands;
   long element_size, band;
   unsigned char band_mask[MX_BANDS], *DC;
   char input_filename[80];
   int status, row,col;

   for(band = 0; band < MX_BANDS; band++){
      band_mask[band] = 0;
   }
   band_mask[0] = 1;
   number_of_bands = 1;
   element_size = 1;
   number_of_rows = SIZE;
   number_of_columns = SIZE;

/*** prompt user for filename ***/
   printf("\nEnter the input filename: ");
   scanf("%s", input_filename);

/*** open raw image file to read ***/
   status = open_raw_file(number_of_rows, number_of_columns,
                          number_of_bands, element_size,
                          input_filename, "r", -1, &picops_in);

   DC = (unsigned char *)    calloc(sizeof(picops_in.number_of_columns),1);

/*** place digital count values into temp_re and temp_im arrays ***/
   for(row = 0; row < picops_in.number_of_rows; row++){
      status = getpixel(picops_in, row, 0, band_mask,
                        picops_in.number_of_columns, DC);
```

```
      for(col = 0; col < picops_in.number_of_columns; col++){
         temp_re[row][col] = (double) *(DC+col);
         temp_im[row][col] = 0.0;
      }
   }
}
```

```
/*************************************************************************
   function: fast fourier transform

   description: This function calculates the fourier transform of the
                real and imaginary parts of an image.  The input
             parameters are the real and imaginary arrays, received
             as pointers, the sign determines whether a forward or
             inverse transform will be performed, the size is the
             size of the image (the image must be square).

             This function was not written by th author of this
             thesis, but was obtained from Michael Heath.
*************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define TWOPI 6.283185307
#define PI 3.141592654
#define SWAP(a,b) tempr = (a); (a)=(b); (b) = tempr

void transform(double *r_im, double *i_im, long sign, long size);
void four1(double data[], int nn, int isign);

void transform(double *r_im, double *i_im, long sign, long size)
{
   double *rt_im, *it_im, *data;
   double freq, sumr, sumi;
   int row, col, x_pos, y_pos, freq_ref, x, y, x_val, y_val;
   rt_im = (double *) calloc((size_t)size, (size_t)sizeof(double));
   it_im = (double *) calloc((size_t)size, (size_t)sizeof(double));
   data = (double *) calloc((size_t)(2*size), (size_t)sizeof(double));
   for(col=0;col<size;col++){
      for(x_pos=0;x_pos<size;x_pos++){
         *(rt_im + x_pos) = *(r_im + (size * col) + x_pos);
         *(it_im + x_pos) = *(i_im + (size * col) + x_pos);
      }
      x_val = size/2.0;
      for(x_pos=0;x_pos<(2*size);x_pos++){
          *(data + x_pos) = *(rt_im + x_val);
          x_pos++;
          *(data + x_pos) = *(it_im + x_val);
          x_val++;
          if (x_val >= size) x_val = x_val - size;
      }
      four1(data-1, size, sign);
      x_val = size/2.0;
      for(x_pos=0;x_pos<(2*size);x_pos++){
          *(r_im + (col * size) + x_val) = *(data + x_pos);
          x_pos++;
          *(i_im + (col * size) + x_val) = *(data + x_pos);
          x_val++;
          if (x_val >= size){x_val = x_val - size;
          }
      }
   }
```

```
   for(row=0;row<=size-1;row++){
      for(y_pos=0;y_pos<size;y_pos++){
               *(rt_im + y_pos) = *(r_im + (size * y_pos) + row);
               *(it_im + y_pos) = *(i_im + (size * y_pos) + row);
      }
      y_val = size/2.0;
      for(y_pos=0;y_pos<(2*size);y_pos++){
          *(data + y_pos) = *(rt_im + y_val);
          y_pos++;
          *(data + y_pos) = *(it_im + y_val);
          y_val++;
          if (y_val >= size){y_val = y_val - size;
          }
      }
      four1(data-1, size, sign);
      y_val = size/2.0;
      for(y_pos=0;y_pos<(2*size);y_pos++){
          *(r_im + (size * y_val) + row) = *(data + y_pos);
          y_pos++;
          *(i_im + (size * y_val) + row) = *(data + y_pos);
          y_val++;
          if (y_val >= size){y_val = y_val - size;
          }
      }
   }
   for(y=0;y<size;y++){
      for(x=0;x<size;x++){
          *(r_im + (size * y) + x) = (*(r_im + (size * y) + x))/
                                                  (double)(size*size);
          *(i_im + (size * y) + x) = (*(i_im + (size * y) + x))/
                                                  (double)(size*size);
      }
   }
   cfree(rt_im);
   cfree(it_im);
   cfree(data);
}

void four1(double data[], int nn, int isign)
{
   int n, mmax, m, j, istep, i;
   double wtemp, wr, wpr, wpi, wi, theta;
   double tempr, tempi;
   n=nn << 1;
   j=1;
   for (i=1;i<=n;i+=2){
      if(j > i){
         SWAP(data[j], data[i]);
         SWAP(data[j+1], data[i+1]);
      }
      m=n >> 1;
      while(m >= 2 && j > m){
         j -= m ;
         m >>= 1;
      }
      j+= m;
   }
```

```
    mmax = 2;
    while(n > mmax){
        istep = 2*mmax;
        theta = 6.28318530717959/(isign*mmax);
        wtemp = sin(0.5*theta);
        wpr = -2.0*wtemp*wtemp;
        wpi = sin(theta);
        wr = 1.0;
        wi = 0.0;
        for(m=1;m<mmax;m+=2){
            for(i=m;i<=n;i+=istep){
                j = i + mmax;
                tempr = wr * data[j] - wi * data[j+1];
                tempi = wr * data[j+1] + wi * data[j];
                data[j] = data[i] - tempr;
                data[j+1] = data[i+1] - tempi;
                data[i]+=tempr;
                data[i+1]+=tempi;
            }
            wr = (wtemp = wr) * wpr - wi * wpi + wr;
            wi = wi * wpr + wtemp * wpi + wi;
        }
        mmax = istep;
    }
}
```

```
/*****************************************************************************
   function: phase_maker.c

   description: When called from the main program. this function will
                generate a 2-D quadratic phase filter.  The phase rate
                is controlled by user input.  The input paramters are
                the real and imaginary parts of the transfer function.
                This function fills these arrays with the real and
                imaginary parts of the phase filter, which is then
                passed back to the main program.

   return value: none

   author: Adam Hanson
           Center for Imaging Science, RIT

   date: 8/19/92
*****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define SIZE 64
#define PI 3.1415926

void phase_maker(double ftr_re[SIZE][SIZE], double ftr_im[SIZE][SIZE])
{
   double  alpha, alpha_sq, rho_sq, size;
   int row, col, ans, row_shift, col_shift;

/*** prompt user for phase rate ***/
   printf("\nEnter a value for the rate of phase change: ");
   scanf("%lf", &alpha);
   alpha_sq = alpha*alpha;

/*** generate real and imaginary phase arrays ***/
   size = SIZE;
   for(row = 0; row < SIZE; row++){
      for(col = 0; col < SIZE; col++){
         row_shift = row - SIZE/2;
         col_shift = col - SIZE/2;
         rho_sq = (row_shift*row_shift +
                   col_shift*col_shift)/(size*size);
         ftr_re[row][col] = cos(PI*alpha_sq*rho_sq);
         ftr_im[row][col] = sin(PI*alpha_sq*rho_sq);
      }
   }
}
```

```
/*****************************************************************************
   function: filter.c

   description:  This function performs a complex multiplication on an
                 input function and a frequency filter.  This filtering
                 multiplication process takes place in the frequency
                 domain and is of the form (a + ib)(c + id).  The input
                 parameters are the real and imaginary arrays of the
                 object (after transforming) and transfer function.
                 The resulting arrays (mult_re, mult_im) are passed
                 back to the main program.

   return value: none

   author: Adam Hanson
             Center for Imaging Science, RIT

   date: 8/19/92
*****************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define SIZE 64

void filter(double obj_re[SIZE][SIZE], double obj_im[SIZE][SIZE],
            double ftr_re[SIZE][SIZE], double ftr_im[SIZE][SIZE],
             double mult_re[SIZE][SIZE], double mult_im[SIZE][SIZE])
{
   int sign, size, row, col, choice;

   for(row = 0; row < SIZE; row++){
      for(col = 0; col < SIZE; col++){
         mult_re[row][col] = (obj_re[row][col]*ftr_re[row][col]) -
                             (obj_im[row][col]*ftr_im[row][col]);
         mult_im[row][col] = (obj_re[row][col]*ftr_im[row][col]) +
                             (obj_im[row][col]*ftr_re[row][col]);
      }
   }
}
```

```
/*****************************************************************************
   function: scale_parts.c

   description: This function scales real/imaginary image data to
                values between 0-255.  The input paramters temp_re
                and temp_im contain object, filter, or filtered object
                data.  After scaling, the integer data is stored in
                int_re and int_im; these arrays are passed back to
                the main program.

   return value: none

   author: Adam Hanson
           Center for Imaging Science, RIT

   date: 8/17/92
*****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define SIZE 64

void scale_parts(double temp_re[SIZE][SIZE], double temp_im[SIZE][SIZE],
                 int int_re[SIZE][SIZE], int int_im[SIZE][SIZE])
{
   char re_out[80], im_out[80];
   double scale_real[SIZE][SIZE], scale_imag[SIZE][SIZE];
   double double_re[SIZE][SIZE], double_im[SIZE][SIZE];
   double rmin, rmax, imin, imax, a, b, rrange, irange;
   int row, col, status;


/*** scale arrays by N ***/

   for(row = 0; row < SIZE; row++){
      for(col = 0; col < SIZE; col++){
         scale_real[row][col] = SIZE*temp_re[row][col];
         scale_imag[row][col] = SIZE*temp_im[row][col];
      }
   }


   rmin = HUGE_VAL;
   rmax = -HUGE_VAL;
   imin = HUGE_VAL;
   imax = -HUGE_VAL;
   for(row = 0; row < SIZE; row++){
      for(col = 0; col < SIZE; col++){
         if(scale_real[row][col] < rmin){
            rmin = scale_real[row][col];
         }else{
            if(scale_imag[row][col] < imin){
               imin = scale_imag[row][col];
            }
```

```
            }
            if(scale_real[row][col] > rmax){
               rmax = scale_real[row][col];
            }else{
               if(scale_imag[row][col] > imax){
                  imax = scale_imag[row][col];
               }
            }
         }
      }
      printf("\nrmax = %f", rmax);
      printf("\nrmin = %f\n", imin);
      printf("\nimax = %f", imax);
      printf("\nimin = %f", imin);

/*** scale arrays from 0-255 ***/

      rrange = rmax - rmin;
      irange = imax - imin;
      for(row = 0; row < SIZE; row++){
         for(col = 0; col < SIZE; col++){
            double_re[row][col] = ((scale_real[row][col] -
                                    rmin)*255)/rrange;
            double_im[row][col] = ((scale_imag[row][col] -
                          imin)*255)/irange;
            int_re[row][col] = (int) (double_re[row][col] + 0.5);
            int_im[row][col] = (int) (double_im[row][col] + 0.5);
         }
      }
}
```

```
/*****************************************************************************
   function: scale_mag.c

   description: This function scales decimal magnitude data to
                grey values ranging from 0-255.  The scaled data is
                then passed back to the main program.  The function
                receives real/imaginary data arrays, calculates the
                magnitude as well as the power.  The power data is then
                stored in rounded integer format and passed back to
                main as int_mag.  The decimal magnitude data can be
                stored by calling the writefilter function.

   return value: none

   author: Adam Hanson
             Center for Imaging Science, RIT

   date: 8/17/92
*****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define SIZE 64

void writefilter(double double_power[SIZE][SIZE]);

void scale_mag(double temp_re[SIZE][SIZE], double temp_im[SIZE][SIZE],
               int int_mag[SIZE][SIZE])
{
   char output_filename[80], filterdata[80];
   double mag[SIZE][SIZE], log_mag[SIZE][SIZE], sq_mag[SIZE][SIZE];
   double scale_real[SIZE][SIZE], scale_imag[SIZE][SIZE];
   double double_power[SIZE][SIZE], min, max, a, b, range;
   int row, col, status, choice;


/*** scale arrays by N squared ***/

   for(row = 0; row < SIZE; row++){
      for(col = 0; col < SIZE; col++){
         scale_real[row][col] = SIZE*SIZE*temp_re[row][col];
         scale_imag[row][col] = SIZE*SIZE*temp_im[row][col];
      }
   }

/*** calculate magnitude and power ***/

   for(row = 0; row < SIZE; row++){
      for(col = 0; col < SIZE; col++){
         a = scale_real[row][col]*scale_real[row][col];
         b = scale_imag[row][col]*scale_imag[row][col];
         mag[row][col] = sqrt(a+b);
         log_mag[row][col] = log(mag[row][col]);
         sq_mag[row][col] = a + b;
      }
```

```
   }

/*** find min and max for scaling ***/

   min = HUGE_VAL;
   max = -HUGE_VAL;
   for(row = 0; row < SIZE; row++){
      for(col = 0; col < SIZE; col++){
         if(sq_mag[row][col] < min){
            min = sq_mag[row][col];
         }
         if(sq_mag[row][col] > max){
            max = sq_mag[row][col];
         }
      }
   }
   printf("\nmax value = %f", max);
   printf("\nmin value = %f\n", min);

/*** scale resulting values from 0-255 ***/

   range = max - min;
   for(row = 0; row < SIZE; row++){
      for(col = 0; col < SIZE; col++){
         double_power[row][col] = ((sq_mag[row][col] -
                              min)*255.0)/range;
         int_mag[row][col] = (int) (double_power[row][col] + 0.5);
      }
   }
   printf("\nOutput power float file(1/0): ");
   scanf("%d", &choice);
   if(choice==1) writefilter(double_power);
}
```

```
/*****************************************************************************
   function: save_mag.c

   description: This function writes out integer magnitude data to
                disk.  The integer magnitude data is passed into the
                function from the main program.  The magnitude data
                could be object, filter, or filtered object data.

   return value: none

   author: Adam Hanson
           Center for Imaging Science, RIT

   date: 8/17/92
*****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "image.h"

#define SIZE 64

FILE *fpout;

void save_mag(int temp[SIZE][SIZE])
{
   struct PICTURE_OPTIONS picops_out;
   long number_of_rows, number_of_columns, number_of_bands;
   long element_size, band;
   unsigned char band_mask[MX_BANDS], *DC;
   char output_filename[80], filterdata[80];
   int row, col, status;

   for(band = 0; band < MX_BANDS; band++){
      band_mask[band] = 0;
   }

   band_mask[0] = 1;
   number_of_bands = 1;
   element_size = 1;
   number_of_rows = SIZE;
   number_of_columns = SIZE;

/*** store output magnitude image ***/

   printf("\nEnter the output filename: ");
   scanf("%s", output_filename);

   status = open_raw_file(number_of_rows, number_of_columns,
                          number_of_bands, element_size,
                          output_filename, "w", -1, &picops_out);

   DC = (unsigned char *) calloc( (size_t) SIZE*SIZE,
                                  (size_t) sizeof( unsigned char ) );

   for(row = 0; row < picops_out.number_of_rows; row++){
```

```
      for(col = 0; col < picops_out.number_of_columns; col++){
         *(DC+col) = (unsigned char) temp[row][col];
      }
      status = putpixel(picops_out, row, 0, band_mask,
                        picops_out.number_of_columns, DC);
   }

   cfree( DC );

   status = (fclose(picops_out.file_pointer));
}
```

```
/*****************************************************************************
   function: save_parts.c

   description: This function is similar to the save_mag function
                except this function saves integer data in real/
                imaginary format.  The input parameters temp_re
                and temp_im could contain object, filter or
                filtered object data.

   return value: none

   author: Adam Hanson
           Center for Imaging Science, RIT

   date: 8/17/92
*****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "image.h"

#define SIZE 64

void save_parts(int temp_re[SIZE][SIZE], int temp_im[SIZE][SIZE])
{
   struct PICTURE_OPTIONS picops_outr, picops_outi;
   long number_of_rows, number_of_columns, number_of_bands;
   long element_size, band;
   unsigned char band_mask[MX_BANDS], *DCR, *DCI;
   char re_out[80], im_out[80];
   int row, col, status;

   for(band = 0; band < MX_BANDS; band++){
      band_mask[band] = 0;
   }

   band_mask[0] = 1;
   number_of_bands = 1;
   element_size = 1;
   number_of_rows = SIZE;
   number_of_columns = SIZE;

/*** store real and imaginary output image files ***/

   printf("\n\nEnter the real output filename: ");
   scanf("%s", re_out);
   printf("Enter the imaginary output filename: ");
   scanf("%s", im_out);

   status = open_raw_file(number_of_rows, number_of_columns,
                          number_of_bands, element_size,
                          re_out, "w", -1, &picops_outr);
   status = open_raw_file(number_of_rows, number_of_columns,
                          number_of_bands, element_size,
                          im_out, "w", -1, &picops_outi);
```

```
   DCR = (unsigned char *) calloc( (size_t) SIZE*SIZE,
                                   (size_t) sizeof( unsigned char ) );
   DCI = (unsigned char *) calloc( (size_t) SIZE*SIZE,
                                   (size_t) sizeof( unsigned char ) );

   for(row = 0; row < picops_outr.number_of_rows; row++){
      for(col = 0; col < picops_outr.number_of_columns; col++){
         *(DCR+col) = temp_re[row][col];
         *(DCI+col) = temp_im[row][col];
      }
      status = putpixel(picops_outr, row, 0, band_mask,
                        picops_outr.number_of_columns, DCR);
      status = putpixel(picops_outi, row, 0, band_mask,
                          picops_outi.number_of_columns, DCI);
   }

   cfree( DCR );
   cfree( DCI );
   status = (fclose(picops_outr.file_pointer));
   status = (fclose(picops_outi.file_pointer));
}
```

```
/*****************************************************************************
   function: writefilter.c

   description: This function writes out the filter magnitude data in
                floating point format.  This is the data used for the
                multiplication in the filtering process.  Using float
                data as opposed to integer data preserves acuracy.

   author: Adam Hanson
           Center for Imaging Science, RIT

   return value: none

   date: 8/17/92
*****************************************************************************/

#include <stdio.h>

#define SIZE 64

FILE *fpout;

void writefilter(double double_mag[SIZE][SIZE])
{
   char filename[80];

   printf("\nEnter a filename for the output datafile: ");
   scanf("%s", filename);

   fpout = fopen(filename, "w");
   fwrite(double_mag, sizeof(double), SIZE*SIZE, fpout);
   fclose(fpout);
}
```

```
/*****************************************************************************
   main program: classifier.c

   description: This is the main program for the classifier. It calls
                the moment generating function for the reference and
                test characters, then calls the minimum distance
                function.  The reference files are opened from a list
                of reference file names called "ref_files.dat".  The
                number of reference moment vectors to be generated is
                controlled by user input and corresponds to the number
                of reference character names listed in ref_files.dat.
                More names can be added or subtracted to adjust the
                size of the reference set.

   return value: none

   authors: Adam Hanson
            Carl Salvaggio
            Center for Imaging Science, RIT

   date: 8/21/92
*****************************************************************************/


#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>

FILE *filenames;

#define NUMBER_OF_MOMENTS 6

main()
{
   char ref_filename[80], test_filename[80];
   double *test_moments;
   double *ref_moments;
   long number_of_rows, number_of_columns;
   int class;
   int number_of_references;
   int row;
   int choice = 1;

/*** input the number of reference characters in the set ***/

   printf( "\nHow many reference vectors to form? " );
   scanf( "%d", &number_of_references );
   printf("Enter the image size (N,N): ");
   scanf("%ld, %ld", &number_of_rows, &number_of_columns);

/*** allocate memory for reference and test vectors ***/

   ref_moments = (double *) calloc(
                              (size_t)
(number_of_references*NUMBER_OF_MOMENTS),
                              (size_t) sizeof( double ) );
```

```
   test_moments = (double *) calloc( (size_t) (NUMBER_OF_MOMENTS),
                                     (size_t) sizeof( double ) );

/*** open file containing reference character filenames ***/
   filenames = fopen("ref_files.dat", "r");

/*** call moment generator to calculate feature space of reference
       characters ***/

   for ( row=0; row<number_of_references; row++ ) {
      fscanf(filenames, "%s", &ref_filename);
      printf( "\nFor reference (%d): %s\n", row+1, ref_filename );
      ref_class( ref_moments+row*NUMBER_OF_MOMENTS,
                 number_of_rows, number_of_columns,ref_filename);
   }
   fclose(filenames);

/*** open file containing test character data ***/
   printf( "\n\nEnter the filename of the test data: " );
   scanf("%s", &test_filename);

/*** call moment generator to calculate feature space of test character ***/

   test_class( test_moments, number_of_rows,
               number_of_columns, test_filename);

/*** call classifier to calculate distances between test and reference
     characters and determine which reference the test character is closest to ***/

   min_dist_mean( ref_moments, test_moments,
       number_of_references,
                  NUMBER_OF_MOMENTS, &class );

   printf("\n");
   printf("The test character most closely matches ");
   printf("reference character: %d\n", class);
}
```

```
/*****************************************************************************
   function: ref_class.c

   description: This function calculates the moment invariant
                feature vector space for the reference
                charcater images.  The feature vectors are calculated
                from the images that are in integer format.  The
                function first calculates the normalized central
                moments, then calculates the feature vector from those
                normalized central moments.

   return value: none

   authors: Adam Hanson
            Carl Salvaggio
            Center for Imaging Science, RIT

   date: 8/25/92
*****************************************************************************/

#include <image.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>

ref_class( double *moments, long number_of_rows,
           long number_of_columns, char filename[80] )
{
   struct PICTURE_OPTIONS picops_in;
   long number_of_bands, element_size, band;
   long row, col, p, q;
   double phi[7];
   double moment, moment_array[2][2], term_a, term_b,image_size,
   double xbar, ybar, gamma, central_moment[4][4], ncm[4][4];
   unsigned char band_mask[MX_BANDS], *DC;
   char input_filename[80];
   int status;

   for( band = 0; band < MX_BANDS; band++){
      band_mask[band] = 0;
   }
   band_mask[0] = 1;
   number_of_bands = 1;
   element_size = 1;
   image_size = number_of_rows*number_of_columns;

/*** read in image file ***/

   status = open_raw_file(number_of_rows, number_of_columns, number_of_bands,
            element_size, filename, "r", -1, &picops_in);

   DC = (unsigned char *) calloc(sizeof(picops_in.number_of_columns), 1 );

   for (p = 0; p < 2; p++){
      for (q = 0; q < 2; q++){
         moment_array[p][q] = 0.0;
      }
```

```
   }
   printf("\n");

/*** this section calculates the non central moments from m(0,0) to m(1,1) ***/

   for (p = 0; p < 2; p++){
      for (q = 0; q < 2; q++){
         moment = 0.0;
         for (col = 0; col < picops_in.number_of_columns; col++){
            status = getpixel( picops_in, col, 0, band_mask,
                                    picops_in.number_of_rows, DC);
            for (row = 0; row < picops_in.number_of_rows; row++){
               if (p == 0){
                  term_a = 1.0;
               }else{
                  if (row == 0){
                     term_a = 0.0;
                  }else{
                     term_a = pow( (double) row, (double) p );
                  }
               }
               if (q == 0){
                  term_b = 1.0;
               }else{
                  if (col == 0){
                     term_b = 0.0;
                  }else{
                     term_b = pow( (double) col, (double) q );
                  }
               }
               moment = moment + term_a*term_b*(*(DC+row));
            }
         }
         moment_array[p][q] = moment/image_size;
      }
   }

   for (p = 0; p < 2; p++){
      for (q = 0; q < 2; q++){
         printf("moment (%ld %ld) = %lf\n", p,q,moment_array[p][q]);
      }
   }

   xbar = moment_array[1][0]/moment_array[0][0];
   ybar = moment_array[0][1]/moment_array[0][0];
   printf("\n");
   printf("xbar = %lf\t ybar = %lf\n\n", xbar, ybar);

/*** this section calculates the necessary central moments to be used
     in determining the vector spaces ***/

   printf("Calculating normalized central moments...\n");
   for (p = 0; p < 4; p++){
      for (q = 0; q < 4; q++){
         moment = 0.0;
         term_a = 0;
         term_b = 0;
```

```
            for (col = 0; col < picops_in.number_of_columns; col++){
               status = getpixel( picops_in, col, 0, band_mask,
                                  picops_in.number_of_rows, DC);
               for (row = 0; row < picops_in.number_of_rows; row++){
                 if (p == 0){
                    term_a = 1.0;
                 }else{
                    if (row - xbar == 0){
                       term_a = 0.0;
                    }else{
                       term_a = pow( (double) (row - xbar), (double) p );
                    }
                 }
                 if (q == 0){
                   term_b = 1.0;
                 }else{
                    if (col - ybar == 0){
                       term_b = 0.0;
                    }else{
                       term_b = pow( (double) (col - ybar), (double) q );
                    }
                 }
                 moment = moment + term_a*term_b*(*(DC + row));
               }
            }
            central_moment[p][q] = moment;
         }
      }

      for (p = 0; p < 4; p++){
         for (q = 0; q < 4; q++){
            if (p+q >= 2){
               gamma = ((p+q)/2.0) + 1;
            }else{
               gamma = 1;
            }
            ncm[p][q] = central_moment[p][q]/
                              pow( central_moment[0][0], gamma );
            printf("Central moment (%ld %ld) = %g\n", p,q,
                    ncm[p][q]);

         }
      }

   /*** vector space calculation ***/

      printf("\n");
      phi[0] = ncm[2][0] + ncm[0][2];

      phi[1] = pow((ncm[2][0] - ncm[0][2]), 2.0) + 4*pow(ncm[1][1], 2.0);

      phi[2] = pow((ncm[3][0] - 3*ncm[1][2]), 2.0) +
               pow((3*ncm[2][1] - ncm[0][3]), 2.0);


      phi[3] = pow((ncm[3][0] + ncm[1][2]), 2.0) +
               pow((ncm[2][1] + ncm[0][3]), 2.0);
```

```
    phi[4] = (ncm[3][0] - 3*ncm[1][2])*(ncm[3][0] + ncm[1][2]) *
           (pow((ncm[3][0] + ncm[1][2]), 2.0) -
            3*pow((ncm[2][1] + ncm[0][3]), 2.0)) +
           (3*ncm[2][1] - ncm[0][3])*(ncm[2][1] + ncm[0][3]) *
           (3*pow((ncm[3][0] + ncm[1][2]), 2.0) -
            pow((ncm[2][1] + ncm[0][3]), 2.0));

    phi[5] = (ncm[2][0] - ncm[0][2]) *
           (pow((ncm[3][0] + ncm[1][2]), 2.0) -
            pow((ncm[2][1] + ncm[0][3]), 2.0)) +
           4*ncm[1][1]*(ncm[3][0] + ncm[1][2])*(ncm[2][1] + ncm[0][3]);

    phi[6] = (3*ncm[2][1] - ncm[0][3])*(ncm[3][0] + ncm[2][1]) *
           (pow((ncm[3][0] + ncm[1][2]), 2.0) -
            3*pow((ncm[2][1] + ncm[0][3]), 2.0)) -
           (ncm[3][0] - 3*ncm[1][2])*(ncm[2][1] + ncm[0][3]) *
           (3*pow((ncm[3][0] + ncm[1][2]), 2.0) -
              pow((ncm[2][1] + ncm[0][3]), 2.0));

    for(row=0; row < 7; row++){
       if(phi[row] < 0.0){
          phi[row] = phi[row]*(-1.0);
       }
    }

 /*** adjust for contrast invariance ***/

    *(moments + 0) = sqrt(phi[1])/phi[0];

    *(moments + 1) = (phi[2]*ncm[0][0])/(phi[1]*phi[0]);

    *(moments + 2) = phi[3]/phi[2];

    *(moments + 3) = sqrt(phi[4])/phi[3];

    *(moments + 4) = phi[5]/(phi[3]*phi[0]);

    *(moments + 5) = phi[6]/phi[4];

    for(row = 0; row < 6; row++){
       printf("\nReference vector (%d) = %g", (row+1), *(moments+row));
    }

    status = (fclose(picops_in.file_pointer));
    if( status == 0){
       return;
    }else{
       printf("Error closing file.");
       exit(0);
    }
 }
```

```
/*****************************************************************************
   function: test_class.c

   description: This function is exactly the same as the function
                ref_class.c execpt that this function calculates
                the feature vector for the test character which
                is in floating point format.  The moments are
                calculated from a data file instead of an image file.

   return value: none
   authors: Adam Hanson
            Carl Salvaggio
            Center for Imaging Science, RIT

   date: 8/25/92
*****************************************************************************/

#include <image.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>

#define SIZE 64

double readfilter(double filter[SIZE][SIZE], char filename[80]);

test_class( double *moments, long number_of_rows,
            long number_of_columns, char filename[80] )
{
   struct PICTURE_OPTIONS picops_in;
   long row, col, p, q;
   double phi[7];
   double moment, moment_array[2][2], term_a, term_b,image_size;
   double xbar, ybar, gamma, central_moment[4][4], ncm[4][4];
   double filter[SIZE][SIZE];
   char input_filename[80];
   int status;

   image_size = number_of_rows*number_of_columns;

/*** read in image file ***/

   readfilter(filter, filename);

   for (p = 0; p < 2; p++){
      for (q = 0; q < 2; q++){
         moment_array[p][q] = 0.0;
      }
   }
   printf("\n");

/*** this section calculates the non central moments from m(0,0) to m(1,1) ***/

   for (p = 0; p < 2; p++){
      for (q = 0; q < 2; q++){
         moment = 0.0;
```

```
            for (col = 0; col < number_of_columns; col++){
               for (row = 0; row < number_of_rows; row++){
                  if (p == 0){
                     term_a = 1.0;
                  }else{
                     if (row == 0){
                        term_a = 0.0;
                     }else{
                        term_a = pow( (double) row, (double) p );
                     }
                  }
                  if (q == 0){
                     term_b = 1.0;
                  }else{
                     if (col == 0){
                        term_b = 0.0;
                     }else{
                        term_b = pow( (double) col, (double) q );
                     }
                  }
                  moment = moment + term_a*term_b*filter[col][row];
               }
            }
            moment_array[p][q] = moment/image_size;
         }
      }

      for (p = 0; p < 2; p++){
         for (q = 0; q < 2; q++){
            printf("moment (%ld %ld) = %lf\n", p,q,moment_array[p][q]);
         }
      }

      xbar = moment_array[1][0]/moment_array[0][0];
      ybar = moment_array[0][1]/moment_array[0][0];
      printf("\n");
      printf("xbar = %lf\t ybar = %lf\n\n", xbar, ybar);

   /*** this section calculates the necessary central moments to be used
        in determining the vector spaces ***/

      printf("Calculating normalized central moments...\n");
      for (p = 0; p < 4; p++){
         for (q = 0; q < 4; q++){
            moment = 0.0;
            term_a = 0;
            term_b = 0;
            for (col = 0; col < number_of_columns; col++){
               for (row = 0; row < number_of_rows; row++){
                  if (p == 0){
                     term_a = 1.0;
                  }else{
                     if (row - xbar == 0){
                        term_a = 0.0;
                     }else{
                        term_a = pow( (double) (row - xbar), (double) p );
                     }
```

```
                }
                if (q == 0){
                  term_b = 1.0;
                }else{
                   if (col - ybar == 0){
                      term_b = 0.0;
                   }else{
                      term_b = pow( (double) (col - ybar), (double) q );
                   }
                }
                moment = moment + term_a*term_b*filter[col][row];
             }
          }
          central_moment[p][q] = moment;
       }
    }

    for (p = 0; p < 4; p++){
       for (q = 0; q < 4; q++){
          if (p+q >= 2){
             gamma = ((p+q)/2.0) + 1;
          }else{
             gamma = 1;
          }
          ncm[p][q] = central_moment[p][q]/pow( central_moment[0][0], gamma );
          printf("Central moment (%ld %ld) = %g\n", p,q,
                  ncm[p][q]);
        }
    }

/*** vector space calculation ***/

    printf("\n");
    phi[0] = ncm[2][0] + ncm[0][2];

    phi[1] = pow((ncm[2][0] - ncm[0][2]), 2.0) + 4*pow(ncm[1][1], 2.0);

    phi[2] = pow((ncm[3][0] - 3*ncm[1][2]), 2.0) +
             pow((3*ncm[2][1] - ncm[0][3]), 2.0);


    phi[3] = pow((ncm[3][0] + ncm[1][2]), 2.0) +
             pow((ncm[2][1] + ncm[0][3]), 2.0);

    phi[4] = (ncm[3][0] - 3*ncm[1][2])*(ncm[3][0] + ncm[1][2]) *
            (pow((ncm[3][0] + ncm[1][2]), 2.0) -
             3*pow((ncm[2][1] + ncm[0][3]), 2.0)) +
            (3*ncm[2][1] - ncm[0][3])*(ncm[2][1] + ncm[0][3]) *
            (3*pow((ncm[3][0] + ncm[1][2]), 2.0) -
             pow((ncm[2][1] + ncm[0][3]), 2.0));

    phi[5] = (ncm[2][0] - ncm[0][2]) *
            (pow((ncm[3][0] + ncm[1][2]), 2.0) -
             pow((ncm[2][1] + ncm[0][3]), 2.0)) +
            4*ncm[1][1]*(ncm[3][0] + ncm[1][2])*(ncm[2][1] + ncm[0][3]);

    phi[6] = (3*ncm[2][1] - ncm[0][3])*(ncm[3][0] + ncm[2][1]) *
```

```
            (pow((ncm[3][0] + ncm[1][2]), 2.0) -
             3*pow((ncm[2][1] + ncm[0][3]), 2.0)) -
            (ncm[3][0] - 3*ncm[1][2])*(ncm[2][1] + ncm[0][3]) *
            (3*pow((ncm[3][0] + ncm[1][2]), 2.0) -
               pow((ncm[2][1] + ncm[0][3]), 2.0));

   for(row=0; row < 7; row++){
      if(phi[row] < 0.0){
         phi[row] = phi[row]*(-1.0);
      }
   }

/*** adjust for contrast invariance ***/

   *(moments + 0) = sqrt(phi[1])/phi[0];

   *(moments + 1) = (phi[2]*ncm[0][0])/(phi[1]*phi[0]);

   *(moments + 2) = phi[3]/phi[2];

   *(moments + 3) = sqrt(phi[4])/phi[3];

   *(moments + 4) = phi[5]/(phi[3]*phi[0]);

   *(moments + 5) = phi[6]/phi[4];

   for(row = 0; row < 6; row++){
      printf("\nTest vector (%d) = %g", (row+1), *(moments+row));
   }

}
```

```
/*****************************************************************************
   function: readfilter.c

   description: This function opens and reads the filter data stored
                by writefilter.c  The input parameters are the filter
                array and the filename of the filter array.

   return value: none

   author: Adam Hanson
             Center for Imaging Science, RIT

   date: 8/19/92
*****************************************************************************/

#include <stdio.h>

#define SIZE 64

FILE *fpin;

double readfilter(double filter[64][64], char filename[80])
{
   fpin = fopen(filename, "r");
   fread(filter, sizeof(double), SIZE*SIZE, fpin);
   fclose(fpin);
}
```

```
/*****************************************************************************
   function min_dist_mean.c

   description: This function calculates the statistical distance
                between the test character and each reference character.
                The function then selects the reference vector with the
                shortest distance to the test character as the proper
                match for the test character.  The input parameters are
                the reference and test vectors, the number of references
                and number of moments, and the chosen class (chosen
                character that test is classified as).

   authors: Adam Hanson
            Carl Salvaggio
            Center for Imaging Science, RIT

   retrun value: none

   date: 8/20/92
*****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <malloc.h>

FILE *fpout;

min_dist_mean( double *ref_classes, double *test_class,
               int number_of_references, int number_of_moments,
             int *class )
{
   double *distance, min_distance;
   int class_number, element_number;
   char datafile[30];

/*** allocate memory for distance values for each reference
       character ***/
    distance = (double *) calloc( (size_t) number_of_references,
                                  (size_t) sizeof( double ) );

/*** calculate distance between test character and each
       reference ***/
   for (class_number = 0; class_number < number_of_references; class_number++){
      for (element_number=0; element_number < number_of_moments; element_number++ ) {
         *(distance+class_number) = *(distance+class_number) +
           pow((*(ref_classes + class_number*number_of_moments +
                  element_number) - *(test_class + element_number)),2.0);
      }
   }

/*** find minimum distance out of all distances calculated ***/
   min_distance = HUGE_VAL;
   printf("\n\nEnter a filename for the distance data: ");
   scanf("%s", datafile);
   printf("\n");
   fpout = fopen(datafile, "w");
```

```
   for (class_number = 0; class_number < number_of_references;   class_number++){
      if ( *(distance+class_number) < min_distance ) {
         min_distance = *(distance+class_number);
         *class = (class_number+1);
      }

/*** classifiy test character as reference character that it is closest to ***/
      printf("class = %d distance = %g\n",(class_number+1) ,
              *(distance+class_number));
      fprintf(fpout, "%d\t%g\n", (class_number+1),
             1.0/ *(distance+class_number));
   }
   fclose(fpout);

   return;
}
```

```
 /* makefile for moment_main */

PACKAGE =       moment_main


LIBPATH =       $(HOME)/lib/$(HOSTARCH)
BINPATH =       $(HOME)/bin/$(HOSTARCH)
OBJPATH =               obj/$(HOSTARCH)
INCPATH =       $(HOME)/include

CC =            gcc
CFLAGS =        -c -g -I$(INCPATH)
CLOPTS =        -L$(LIBPATH)

ARCHIVE =       ar
AROPTS =        cr


OBJS =          $(OBJPATH)/getpixel.o \
                $(OBJPATH)/open_raw_file.o \
                $(OBJPATH)/putpixel.o \
                $(OBJPATH)/switch_longword.o \
                $(OBJPATH)/switch_word.o \
                $(OBJPATH)/fft.o \
                $(OBJPATH)/scale_parts.o \
                $(OBJPATH)/scale_mag.o \
                $(OBJPATH)/save_parts.o \
                $(OBJPATH)/save_mag.o \
                $(OBJPATH)/writefilter.o \
                $(OBJPATH)/load_image.o \
                $(OBJPATH)/phase_maker.o \
                $(OBJPATH)/filter.o

all:            moment_main


$(OBJPATH)/getpixel.o: getpixel.c
        $(CC) $(CFLAGS) -o $(OBJPATH)/getpixel.o getpixel.c

$(OBJPATH)/open_raw_file.o: open_raw_file.c
        $(CC) $(CFLAGS) -o $(OBJPATH)/open_raw_file.o open_raw_file.c

$(OBJPATH)/putpixel.o: putpixel.c
        $(CC) $(CFLAGS) -o $(OBJPATH)/putpixel.o putpixel.c

$(OBJPATH)/switch_longword.o: switch_longword.c
        $(CC) $(CFLAGS) -o $(OBJPATH)/switch_longword.o switch_longword.c

$(OBJPATH)/switch_word.o: switch_word.c
        $(CC) $(CFLAGS) -o $(OBJPATH)/switch_word.o switch_word.c

$(OBJPATH)/fft.o: fft.c
        $(CC) $(CFLAGS) -o $(OBJPATH)/fft.o fft.c

$(OBJPATH)/scale_parts.o: scale_parts.c
        $(CC) $(CFLAGS) -o $(OBJPATH)/scale_parts.o scale_parts.c
```

```
$(OBJPATH)/scale_mag.o: scale_mag.c
	$(CC) $(CFLAGS) -o $(OBJPATH)/scale_mag.o scale_mag.c

$(OBJPATH)/save_parts.o: save_parts.c
	$(CC) $(CFLAGS) -o $(OBJPATH)/save_parts.o save_parts.c

$(OBJPATH)/save_mag.o: save_mag.c
	$(CC) $(CFLAGS) -o $(OBJPATH)/save_mag.o save_mag.c

$(OBJPATH)/writefilter.o: writefilter.c
	$(CC) $(CFLAGS) -o $(OBJPATH)/writefilter.o writefilter.c

$(OBJPATH)/load_image.o: load_image.c
	$(CC) $(CFLAGS) -o $(OBJPATH)/load_image.o load_image.c

$(OBJPATH)/phase_maker.o: phase_maker.c
	$(CC) $(CFLAGS) -o $(OBJPATH)/phase_maker.o phase_maker.c

$(OBJPATH)/filter.o: filter.c
	$(CC) $(CFLAGS) -o $(OBJPATH)/filter.o filter.c


################################
# Control Routines and Libraries #
################################


$(OBJPATH)/moment_main.o: moment_main.c
	$(CC) $(CFLAGS) -o $(OBJPATH)/moment_main.o moment_main.c

$(LIBPATH)/libmoment_main.a: $(OBJS)
	$(ARCHIVE) $(AROPTS) $(LIBPATH)/libmoment_main.a $(OBJS)
	ranlib $(LIBPATH)/libmoment_main.a

moment_main: $(LIBPATH)/libmoment_main.a $(OBJPATH)/moment_main.o
	$(CC) $(OBJPATH)/moment_main.o $(CLOPTS) -lmoment_main -lm -o moment_main


###############
# Maintenance #
###############

clean:
	rm -f $(OBJS) *.o a.out $(LIBPATH)/libmoment_main.a moment_main

update:
	mv -f moment_main $(BINPATH)/ moment_main

architecture:
	@ printenv HOSTARCH
```

```
/* classifier makefile */

PACKAGE =       classifier


LIBPATH =       $(HOME)/lib/$(HOSTARCH)
BINPATH =       $(HOME)/bin/$(HOSTARCH)
OBJPATH =               obj/$(HOSTARCH)
INCPATH =       $(HOME)/include

CC =            gcc
CFLAGS =        -c -g -I$(INCPATH)
CLOPTS =        -L$(LIBPATH)

ARCHIVE =       ar
AROPTS =        cr


OBJS =          $(OBJPATH)/getpixel.o \
                $(OBJPATH)/mdm.o \
                $(OBJPATH)/open_raw_file.o \
                $(OBJPATH)/putpixel.o \
                $(OBJPATH)/switch_longword.o \
                $(OBJPATH)/switch_word.o \
                $(OBJPATH)/ref_class.o \
                $(OBJPATH)/test_class.o \
                $(OBJPATH)/readfilter.o


all:            classifier


$(OBJPATH)/getpixel.o: getpixel.c
        $(CC) $(CFLAGS) -o $(OBJPATH)/getpixel.o getpixel.c

$(OBJPATH)/mdm.o: mdm.c
        $(CC) $(CFLAGS) -o $(OBJPATH)/mdm.o mdm.c

$(OBJPATH)/moment_gen.o: moment_gen.c
        $(CC) $(CFLAGS) -o $(OBJPATH)/moment_gen.o moment_gen.c

$(OBJPATH)/open_raw_file.o: open_raw_file.c
        $(CC) $(CFLAGS) -o $(OBJPATH)/open_raw_file.o open_raw_file.c

$(OBJPATH)/putpixel.o: putpixel.c
        $(CC) $(CFLAGS) -o $(OBJPATH)/putpixel.o putpixel.c

$(OBJPATH)/switch_longword.o: switch_longword.c
        $(CC) $(CFLAGS) -o $(OBJPATH)/switch_longword.o switch_longword.c

$(OBJPATH)/switch_word.o: switch_word.c
        $(CC) $(CFLAGS) -o $(OBJPATH)/switch_word.o switch_word.c

$(OBJPATH)/ref_class.o: ref_class.c
        $(CC) $(CFLAGS) -o $(OBJPATH)/ref_class.o ref_class.c

$(OBJPATH)/test_class.o: test_class.c
```

```
	$(CC) $(CFLAGS) -o $(OBJPATH)/test_class.o test_class.c

$(OBJPATH)/readfilter.o: readfilter.c
	$(CC) $(CFLAGS) -o $(OBJPATH)/readfilter.o readfilter.c


################################
# Control Routines and Libraries #
################################

$(OBJPATH)/classifier.o: classifier.c
	$(CC) $(CFLAGS) -o $(OBJPATH)/classifier.o classifier.c


$(LIBPATH)/libclassifier.a: $(OBJS)
	$(ARCHIVE) $(AROPTS) $(LIBPATH)/libclassifier.a $(OBJS)
	ranlib $(LIBPATH)/libclassifier.a

classifier: $(LIBPATH)/libclassifier.a $(OBJPATH)/classifier.o
	$(CC) $(OBJPATH)/classifier.o $(CLOPTS) -lclassifier -lm -lmalloc -o classifier


###############
# Maintenance #
###############

clean:
	rm -f $(OBJS) *.o a.out $(LIBPATH)/libclassifier.a classifier

update:
	mv -f classifier $(BINPATH)/classifier

architecture:
	@ printenv HOSTARCH
```

## APPENDIX D

The following is an example of the normalized central moments calculated for the upper-case 'A' (figure 14), and the feature vector calculated from the moments.

Central moment (0 0) = 1
Central moment (0 1) = 2.39229e-14
Central moment (0 2) = 0.00104754
Central moment (0 3) = -8.1313e-06
Central moment (1 0) = -4.47264e-16
Central moment (1 1) = -6.86608e-06
Central moment (1 2) = 6.81853e-09
Central moment (1 3) = -1.62363e-08
Central moment (2 0) = 0.00060626
Central moment (2 1) = 1.50311e-05
Central moment (2 2) = 7.3128e-07
Central moment (2 3) = 2.86427e-08
Central moment (3 0) = -2.9026e-07
Central moment (3 1) = -1.72279e-08
Central moment (3 2) = -7.19674e-10
Central moment (3 3) = -3.87885e-11

Reference vector (1) = 0.266957
Reference vector (2) = 8.78835
Reference vector (3) = 0.016833
Reference vector (4) = 2.76669
Reference vector (5) = 0.26661
Reference vector (6) = 6.43944

## REFERENCES


1. Belkasim, S.O., M. Shridhar, M. Ahmad, "Shape - Contour Recognition Using Moment Invariants, Proceedings of the IEEE, (1990).

2. Casasent, David, Demetri Psaltis, "Optical Pattern Recognition Using Normalized Invariant Moments", Optical Pattern Recognition, SPIE Vol. 201, Bellinghham, Washington, (1979).

3. Cash, Glenn L., Mehdi Hatamian, "Optical Character Recognition By the Method of Moments", Computer Vision, Graphics, and Image Processing, Vol. 39, (1987).

4. El-Dabi, Sherif Sami, Refat Ramsis, Aladin Kamel, "Arabic Character Recognition System: A Statistical Approach For Recognizing Cursive Typewritten Text", Pattern Recognition, Vol. 23, No. 5, Great Britain, (1990).

5. Hu, Ming-Kuei, "Visual Pattern Recognition By Moment Invariants", IRE Transactions Information Theory, Vol. IT-8, (1962).

6. Li, Y., "Applications of Moment Invariants to Neurocomputing for Pattern Recognition", Electronic Letters, Vol. 27, No. 7, (1991).

7. Maitra, Sidhartha, "Moment Invariants", Proceedings of the IEEE, Vol. 67, No. 4, (1979).

8. Zinzindohoue, P., "Spatial Moment Invariants and Pattern Recognition", Optik, Vol. 87, No. 2, (1991).