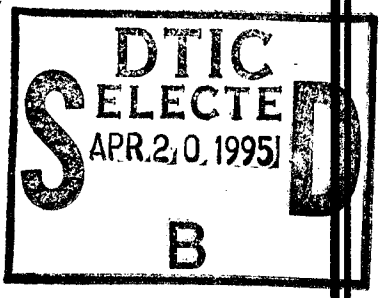
AL/HR-TR-1994-0106

IMAGE REPRESENTATION USING FAST ALGORITHMS BASED ON THE ZAK TRANSFORM

George A. Geri
University of Dayton Research Institute
300 College Park Avenue
Dayton, Ohio 45469



Izidor C. Gertner
Department of Computer Science
City College of CUNY
Convent Avenue & 138th Street
New York, New York 10031

HUMAN RESOURCES DIRECTORATE
AIRCREW TRAINING RESEARCH DIVISION
6001 S. Power Road, Bldg 558
Mesa, AZ 85206-0904

December 1994

Final Technical Paper for the Period October 1990 - March 1994

Approved for public release; distribution is unlimited.

AIR FORCE MATERIEL COMMAND
BROOKS AIR FORCE BASE, TEXAS

ARMSTRONG LABORATORY

NOTICES

When Government drawings, specifications, or other data are used for any purpose other than in connection with a definitely Government-related procurement, the United States Government incurs no responsibility or any obligation whatsoever. The fact that the Government may have formulated or in any way supplied the said drawings, specifications, or other data, is not to be regarded by implication, or otherwise in any manner construed, as licensing the holder, or any other person or corporation; or as conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

The Office of Public Affairs has reviewed this report, and it is releasable to the National Technical Information Service, where it will be available to the general public, including foreign nationals.

This report has been reviewed and is approved for publication.

Elizabeth L. Martin
ELIZABETH L. MARTIN
Project Scientist

Dee H. Andrews
DEE H. ANDREWS
Technical Director

LYNN A. CARROLL, Colonel, USAF
Chief, Aircrew Training Research Division

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1994	Final October 1990 - March 1994

4. TITLE AND SUBTITLE
Image Representation Using Fast Algorithms Based on the Zak Transform

5. FUNDING NUMBERS
C - F33615-90-C-0005
PE - 62205F
PR - 1123
TA - 03
WU - 85

6. AUTHOR(S)
George A. Geri
Izidor C. Gertner

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Armstrong Laboratory (AFMC)
Human Resources Directorate
Aircrew Training Research Division
6001 S. Power Road, Bldg 558
Mesa, AZ 85206-0904

8. PERFORMING ORGANIZATION REPORT NUMBER
AL/HR-TR-1994-0106

9. SPONSORING/MONITORING AGENCY NAMES(S) AND ADDRESS(ES)

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
Armstrong Laboratory Technical Monitor- Dr Elizabeth L. Martin, (602) 988-6561

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

Flight simulator imagery is often made up of natural scenes whose charateristics are not constant across the image. This property suggests that such imagery can be most efficiently represented by spectral techniques that use spatially localized basis functions. This report describes techniques for decomposing full gray-scale images into a joint position/spatial-frequency domain using bases derived from various window functions. The first set of window functions consists of the hermite functions which are related to gaussian derivatives. The second set is based on a new window function that is obtained from a weighted ZAK transform and that provides good localization properties and stable computation. The third set is based on a localized cosine function and allows images to be decomposed using real numbers only. All of the techniques described provide a framework for filtering images in a position-varying manner. For all the basis functions descibed here, image generation from combined position and spatial-frequency information involves a computationally intensive four-dimensional summation. By an application of the Zak Transform, we are able to replace this summation with a fast Fourier transform, this significantly reducing the complexity of the computation.

14. SUBJECT TERMS
Efficient Representation; Fast Algorithms; Fast Fourier Transform; FFT; Flight Simulators; Hermite Functions; Imagery; Images; Localized Bases; Zak Transform

15. NUMBER OF PAGES
125

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

CONTENTS

List of Figures

PREFACE

The research reported here was conducted in support of the Armstrong Laboratory, Human Resources Directorate, Aircrew Training Research Division (AL/HRA) under Work Unit 1123-03-85, Flying Training Research Support.

This research was supported by Air Force Contract F33615-90-0005 to the University of Dayton Research Institute. The laboratory contract monitor was Mrs. Patricia Spears.

The authors thank Colonel Lynn Carroll for supporting the Image Generator Project under which portions of the research reported here were performed. We also acknowledge the Laboratory Director's Funds which supported the early stages of this research. Finally, we thank Drs. Elizabeth Martin and Byron Pierce for their support of this project.

IMAGE REPRESENTATION USING FAST ALGORITHMS BASED ON THE ZAK TRANSFORM

I. GENERAL INTRODUCTION

Visual images can, in many contexts, be most efficiently represented by decomposing them into spectral functions that can then be added together to reconstruct the image (cf., Geri, Zeevi & Porat, 1990). Further, the human visual system has certain properties which suggest the most appropriate characteristics for the spectral functions used to represent visual imagery. As noted above, the human visual system is spatially inhomogeneous, and so the most appropriate basis functions are those that are *localized* (i.e., that have finite extent). Also, the visual mechanisms presumed to underlie form discrimination are *symmetrical*, and so it may be useful for the basis functions, used to represent visual imagery, to be symmetrical. Finally, it is most appropriate to represent images using *orthogonal* basis functions which allow representation with minimal redundancy and with the fewest number of coefficients.

The foregoing suggests that the ideal set of basis functions, for representing visual images, should be localized, symmetrical, and orthogonal. The problem is that a basis with finite support (i.e., one that is spatially localized) cannot be both orthogonal and symmetrical. Therefore, as a practical matter, the chosen basis must represent a compromise among the properties of localization, symmetry, and orthogonality. One example of a popular basis is the Gabor functions (cf. Porat & Zeevi, 1988), which, although not orthogonal, provide good combined localization in position and spatial-frequency. Another well-known basis is the wavelets devised by Daubechies (1988), which are localized and orthogonal but are not symmetrical. In the present experiment, we describe three bases derived from window functions that have various desirable properties in the context of image representation. We begin in Part II by attempting an intuitive description of the Zak transform (ZT) based on an analogy with the more familiar Fourier transform.

In Part III, we present a mathematical technique for analyzing images based on two-dimensional Hermite functions that are translated in both space and spatial frequency. Although the translated functions are not orthogonal, they do constitute a frame and hence can be

used for image expansion. The technique has the practical advantage that fast algorithms based on the ZT can be used to compute expansion coefficients. We describe properties of the ZT that are relevant to image representation, and which allow us to use the ZT to both efficiently compute expansion coefficients and to reconstruct images from them. Finally, we use a Hermite function frame to decompose and reconstruct a texture image.

In Part IV, an image representation technique is described, which uses a window function that provides good localization in both space and spatial frequency. The window function is obtained by weighting the ZT of a gaussian. The weighting procedure eliminates the zero in the ZT thus allowing efficient and stable computation of expansion coefficients with respect to the derived window function. Since the window function is related to Gabor functions and, in addition, resembles a visual receptive field, it may also be useful in visual representation and modeling.

Finally, in Part V, we describe an application of the discrete cosine transform (DCT) to the representation of images in the combined position/spatial-frequency domain. The major computational tool for calculating the coefficients is what we call the discrete Zak-Cosine transform (DZCT). The technique is mathematically complete in that the original image can be reconstructed exactly.

II. INTRODUCTION TO THE ZAK TRANSFORM

Fourier transform techniques are inherently limited in that their associated basis functions are theoretically infinite in extent and hence cannot adequately represent the localized features typical of most real-world images. This limitation can be addressed by combined position/spatial-frequency representations based on the Wigner distribution, ambiguity functions, Gabor functions, etc. (cf., Jacobson & Wechsler, 1988). Although the Zak Transform (ZT) was originally developed in the context of quantum mechanics (Zak, 1967), it has been used extensively in other areas of both pure (Janssen, 1982) and applied (Auslander, Gertner & Tolimieri, 1991; Bergmans & Janssen, 1987) research. Further, as noted by Zeevi and Gertner (1992), the ZT can be used in image representation to map a two-dimensional signal (such as an

image) into a four-dimensional space which can be interpreted as consisting of two spatial dimensions and two spatial-frequency dimensions. It will now be shown that the ZT is, in a sense, a partial or intermediate result of the Fast Fourier Transform (FFT), and that it provides an effective joint position/spatial-frequency representation. The following discussion of the Zak transform considers one-dimensional signals only and is meant to be elementary an intuitive. A more detailed description of the ZT, its properties, and its application to two-dimensional signals (images) will be presented in Part III.

Consider first an eight-element, one-dimensional vector of data as shown at the top of Figure 1. An FFT can be performed on this vector as follows: 1) Decompose the vector into smaller vectors and rearrange them as shown in the second and third panels of Figure 1. This maps the eight-element vector into a two-dimensional 4x2-element array. 2) Compute an FFT, of length two, on each of the four columns. 3) Multiply by an appropriate phase factor. 4) Compute an FFT, of length four, on each of the two rows. 5) Reindex back to a one-dimensional array that now represents frequency.

Now consider the discrete ZT as applied to a 16-element vector. The individual data points, which are at a relatively fine scale, are first partitioned into a coarser scale, as shown at the top of Figure 2. The samples corresponding to each index on the coarser scale are then mapped into rows forming a 4x4 array as shown in the second panel of Figure 2. Each cell in the resultant array is thus represented by two numbers--one corresponding to the coarse scale and one corresponding to the fine scale. Next, an FFT is performed on the first row of the array that represents the set of first samples, on the fine scale, within each coarse block. The results (i.e., the Fourier coefficients) are placed in the corresponding row of a second array, which therefore contains the FT of the set of first-fine samples within each coarse block. This process is continued for each row in the original array, so that each row in the second array contains the coefficients associated with successive sets of fine samples. As a result of this procedure, we now have a two-dimensional representation of the original one-dimensional signal. We have lost some resolution in one dimension (the original data dimension, e.g., time or position), but we have added resolution in another (the transform or frequency) dimension.

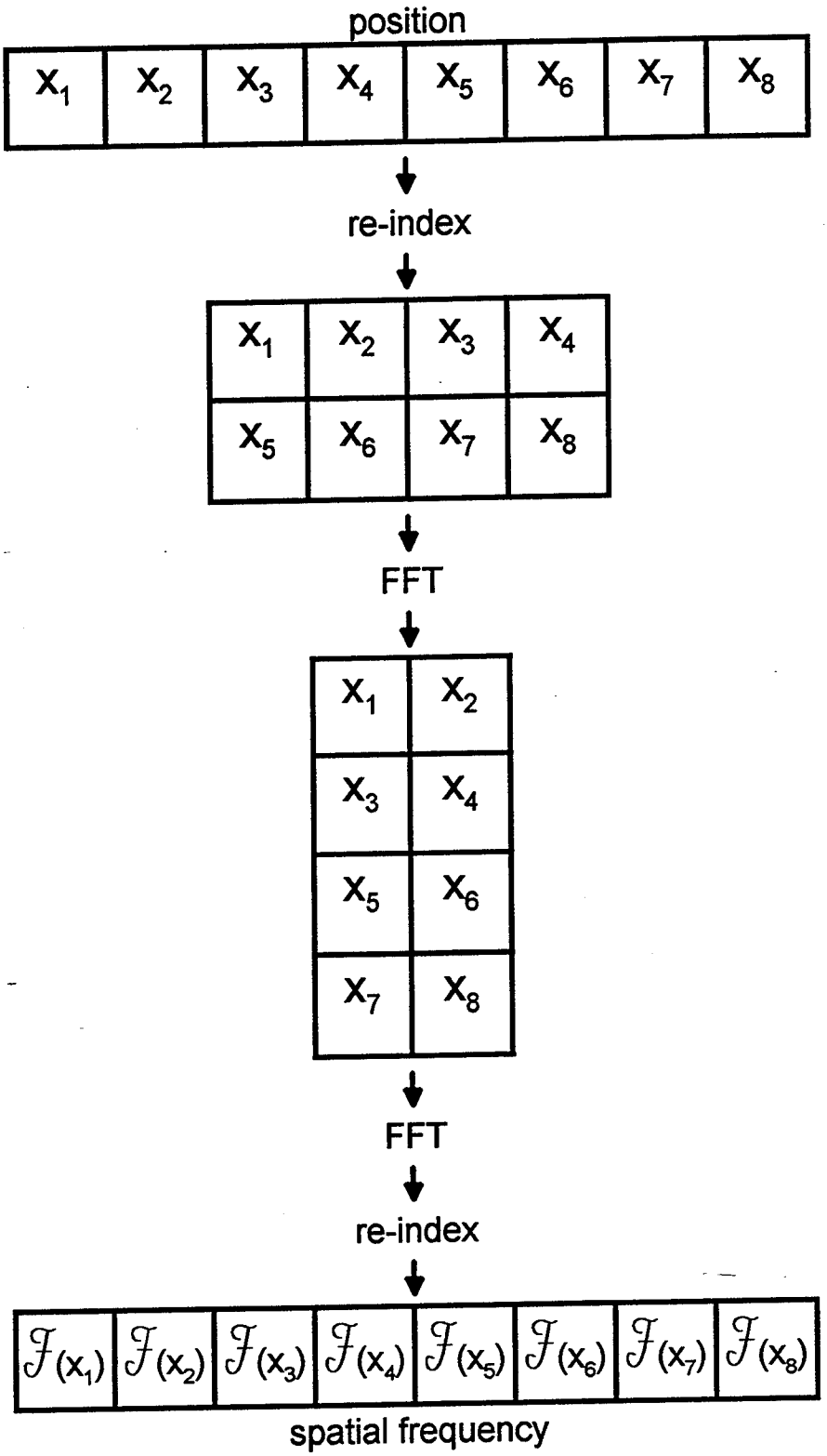


Figure 1.
Steps in Performing a 1-D Fast Fourier Transform

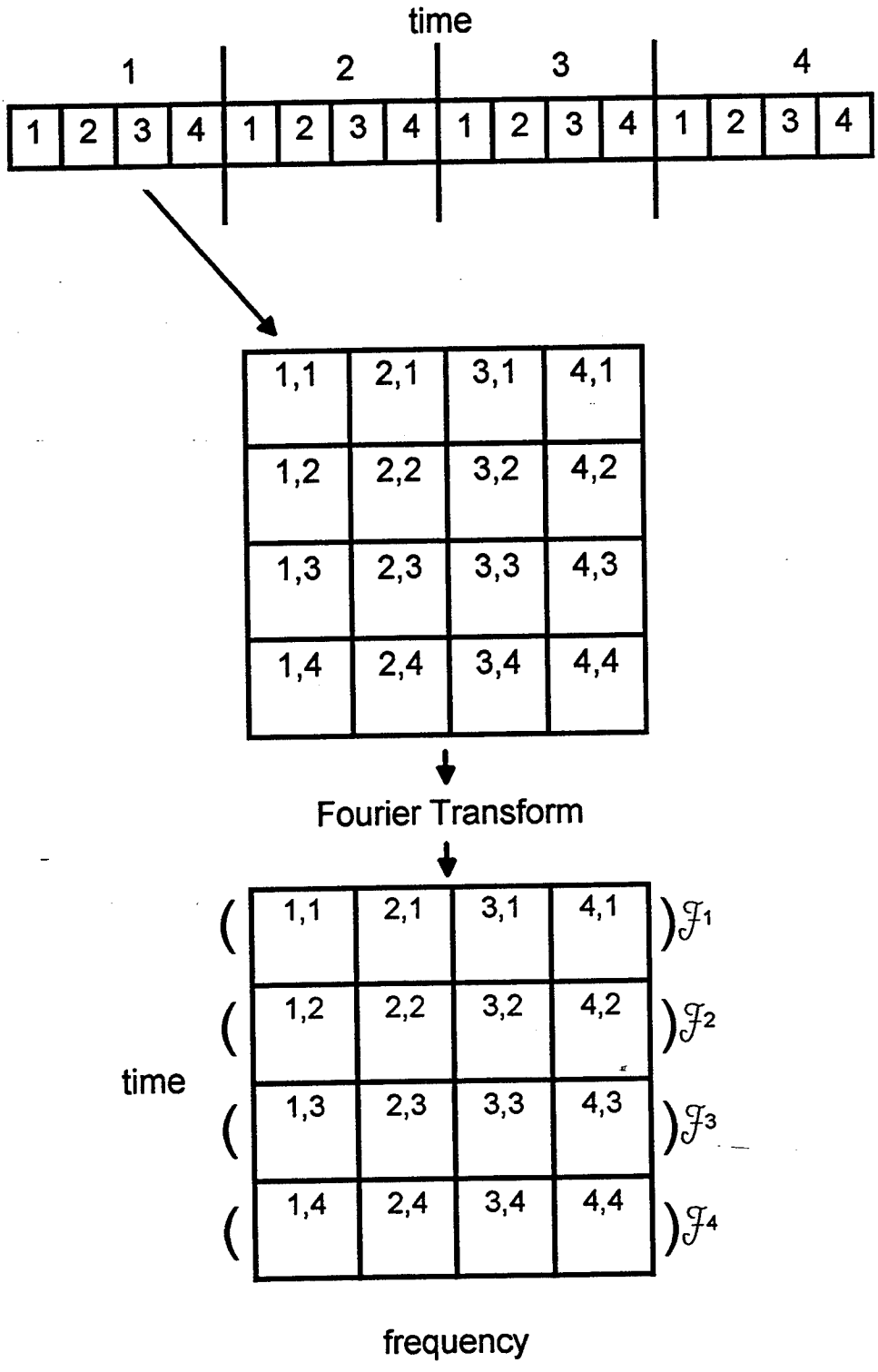


Figure 2

Steps in Performing a 1-D Zak Transform

By analogy with Figure 1, Figure 2c shows that *the ZT is an intermediate step in an ordinary FFT.* This intermediate procedure has several interesting and useful properties. First, the ZT is quasi-periodic (i.e., its absolute value is periodic in both of the dimensions described above), so that the ZT at a higher frequency can be obtained by simply extending the ZT at a lower frequency. Second, the ZT of a shifted (in either or both dimensions) gaussian is equal to the ZT of an unshifted gaussian times a phase factor. It is also a well-known *FT* property that a shift in time, for instance, is equivalent to multiplying by a phase factor in frequency. However, this property is even more powerful in the context of the ZT since it applies to both (all) dimensions. As a result, and as is more fully described in the Introduction to Part IV, rather than taking the ZT of gaussians at all positions and frequencies, it is only necessary to calculate the ZT of a single gaussian and multiply it by the appropriate (complex-exponential) phase factors.

The ZT properties just described, as well as others, will be discussed in more detail in the next section, where a technique is described for calculating Gabor-like expansion coefficients using the ZT of window functions related to gaussian derivatives.

III. IMAGE REPRESENTATION USING HERMITE FUNCTIONS

Introduction

Most features of interest in natural and man-made images are spatially localized. It is, therefore, not surprising that the mammalian visual system has evolved properties to deal with localized features. One such property is the visual receptive field (VRF), the most salient characteristics of which are its limited spatial extent and the form of its sensitivity profile. Early models of the visual system suggested that the VRF was analogous to a Fourier analyzer (Braddick, Campbell & Atkinson, 1978). This analogy was known even then to be inadequate since Fourier analysis is global in nature and thus cannot adequately model the visual response to localized spatial features. One of the more popular models suggested for addressing this inadequacy involved the introduction of Gabor functions (Daugman, 1980; Marcelja, 1980) to replace the sinusoids used in Fourier analysis. Gabor functions (i.e., gaussian-weighted sinusoids) are spatially localized, and were initially popular because they were known to provide the highest conjoint resolution in the spatial and spatial-frequency domains. Two-dimensional Gabor

functions have also been found to resemble the VRF of many cortical cells (Daugman, 1988; Jones & Palmer, 1987).

Many functions, in addition to Gabor functions, have been proposed to describe VRF profiles. These include differences of gaussians (Kulikowski, Marcelja & Bishop, 1982; Rodieck & Stone, 1965), Laplacians of gaussians (Marr & Hildreth, 1980), and gaussian derivatives (Stork & Wilson, 1990; Young, 1987). It is no coincidence that all of these functions involve gaussians in some form. As noted by Marr (1982) and by Koenderink (1984), efficient feature extraction requires that images be analyzed at multiple spatial scales. Thus, analysis at a particular scale involves the use of blurring to remove details at higher levels. The gaussian is uniquely suited to this role in that it is smooth and localized, and hence least likely to introduce either spatial or spectral artifacts.

Koenderink and van Doorn (1990) have suggested a taxonomy of mathematically allowable VRF profiles based on sets of functions consistent with known properties of the visual system such as size invariance, absence of spurious resolution, and effectively continuous spatial sampling. They note that the functions which describe the VRFs that possess these properties are solutions of a particular differential equation, and that one family of solutions is the weighted Hermite polynomials (Yang, 1992). Further, Zucker and Hummel (1986) have suggested that the spatial interpolation functions required, for instance, to explain visual hyperacuity should have spatial support that is less than that provided by more traditional sinc-functions. They recommended the use of Hermite polynomials, which are more local than sinc-functions, and which are related to gaussian derivatives and hence are appropriate for characterizing gaussian VRFs. Finally, the use of Hermite polynomials is also supported by the curve-fitting analysis performed by Young (1987), who tested several candidate functions for representing the VRFs derived from the visual responses of various mammalian species. Although many of the functions gave acceptable fits to certain of the data, he concluded that the gaussian derivative (also a type of weighted Hermite polynomial) provided the best overall fit.

In the present paper we describe a technique for decomposing images in a combined position/spatial-frequency domain, using a frame derived from Hermite functions (i.e., Hermite polynomials multiplied by gaussians). A frame is a generalization of a basis, which allows

expansion using a broader class of functions (cf., Daubechies, 1992). Unlike Martens (1990), who decomposed images with respect to an orthonormal family of Hermite functions (i.e., a basis), we generate a Gabor-like frame using any one of the Hermite functions as a window. This frame is generated by shifting the chosen Hermite function to all possible discrete image positions. At each position a Gabor-like expansion is performed using spatial frequencies matched to the image. We also describe a fast algorithm, based on the Zak transform, for computing expansion coefficients relative to the proposed frame.

Image Representation in the Position/Spatial-Frequency Domain

The human visual system is spatially inhomogeneous and thus visual information is most efficiently represented in both position and spatial frequency. Many methods have been proposed for representing images in a joint position/spatial-frequency domain (Jacobson & Wechsler, 1988). All of these methods are based on approximations to the Wigner distribution, wherein finite images are assumed to be two-dimensional, continuous, integrable functions, and the joint representation is computed by using integration with infinite bounds. Since integrals with infinite bounds are not always computable, various approximations have been introduced (Jacobson & Wechsler, 1988). In order to avoid these approximations, we propose a decomposition into a finite, position/2-D spatial frequency domain as follows:

We represent the image, $I(n_x, n_y)$, in a realistic (i.e., finite) form as an array of points where n_x and n_y are the row and column indices, respectively, of the image pixels. Let $h(n_x,n_y)$ be a basic generating function (i.e., any one of the functions shown in Figure 3). We then generate a frame (Daubechies, 1990; 1992) by the following procedure:

1. Let the image size be $N \times N$ pixels. Assume that N is a composite number (for all practical applications N is a power of two) such that $N = M \times L$. This is equivalent to saying that the image is represented by $L \times L$ blocks each of size $M \times M$.

2. Generate a finite set of functions, $h_{k,l}(n_x,n_y)$, by first centering $h(n_x,n_y)$ at the image points $(n_x\text{-}kM,\ n_y\text{-}lM)$, where $k,l = 0,1,...,L\text{-}1$. These are the positions where the expansion coefficients will be computed.

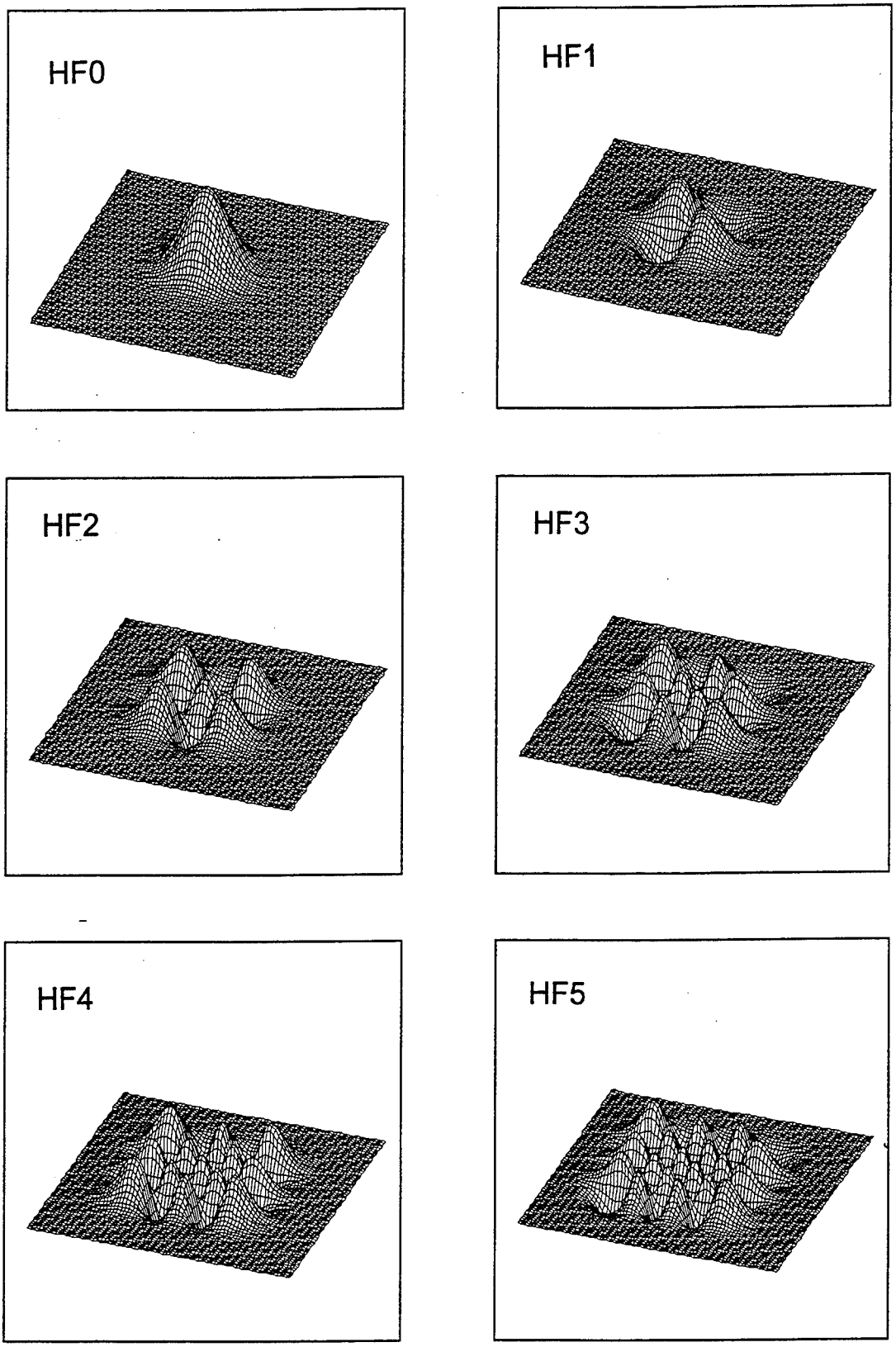


Figure 3
Hermite Functions of Order Zero Through Five

3. At each position, generate a set of functions, $h_{k,l,m,n}(n_x,n_y)$, at various 2-D spatial frequencies, as follows:

$$h_{k,l,m,n}(n_x,n_y) = h_{k,l}(n_x,n_y)\ \exp\left[-2\pi i\left(\frac{n_x m}{N}+\frac{n_y n}{N}\right)\right] \quad , \tag{1}$$

where $m,n=0,1,...M-1$. This corresponds to M, 2-D spatial frequencies over the range 0 to $2\pi\frac{m}{M}$. The set of functions given in Equation 1 constitutes a frame according to Theorem 1 from Janssen (in press), and thus the following decomposition is legitimate.

4. Compute the coefficients $c_{k,l,m,n}$ by decomposing the image relative to the set of functions $h_{k,l,m,n}(n_x,n_y)$ as an inner product as follows:

$$c_{k,l,m,n} = \ < I(n_x,n_y)\ ,\ h_{k,l,m,n}(n_x,n_y) > \tag{2}$$

such that

$$I(n_x,n_y) = \sum_{k,l,m,n}^{L,M} c_{k,l,m,n}\ h_{k,l,m,n}(n_x,n_y) \quad , \tag{3}$$

where

$$\sum_{k,l,m,n}^{L,M} \equiv \sum_{k=0}^{L-1}\sum_{l=0}^{L-1}\sum_{m=0}^{M-1}\sum_{n=0}^{M-1} \quad .$$

As a result of this decomposition, we now have a finite, discrete, 2-D spectrum at each location (k,l). The discrete 2-D spectrum can be converted to the more conventional representation in spatial frequency and orientation through a cartesian-to-polar coordinate transformation.

The direct computational procedure to synthesize the image from the expansion coefficients, $c_{k,l,m,n}$, requires that a 4-D summation be performed. This is a computationally intensive procedure. In the next section we will describe our implementation of a fast algorithm that has been proposed (cf., Zeevi & Gertner, 1992) for computing expansion coefficients and for

reconstructing an image from those coefficients. The major computational tool is the finite Zak transform.

The Finite Zak Transform of Images

The 4-D summation in Equation 3 is computationally intensive because it cannot in general be facilitated by an FFT-like algorithm. It has been shown that a fast algorithm can be developed using the Zak transform (Zeevi & Gertner, 1992). The Zak transform (ZT) of an image represents an intermediate result of the 2-D fast Fourier transform, and thus provides a combined position/spatial-frequency representation.

For an image of size $N \times N$, and $N = L \cdot M$, the finite Zak transform is computed as follows:

$$(Z_L I)(i,j;\rho,\sigma) = \sum_{r=0}^{L-1} \sum_{p=0}^{L-1} I(i+Mr, j+Mp)\ \exp\left[2\pi i\left(\frac{\rho r}{L} + \frac{\sigma p}{L}\right)\right], \tag{4}$$

where i and j are position variables and ρ and σ are frequency variables.

There are four properties of the ZT which are relevant to the development of fast algorithms for image representation. First, it follows directly from Equation 4 that the ZT is *periodic* in each pair of variables, *(i,j)* and *(ρ,σ)*. Moreover, the ZT satisfies the additional periodicity relations:

$$(Z_L)(i+M, j+M;\rho,\sigma) = \exp\left[-2\pi i\left(\frac{\rho}{L} + \frac{\sigma}{L}\right)\right] \times (Z_L I)(i,j;\rho,\sigma), \tag{5}$$

and

$$(Z_L I)(i,j;\rho+L,\sigma+L) = (Z_L I)(i,j;\rho,\sigma). \tag{6}$$

The significance of this property is that any computation that involves the ZT need be performed over the fundamental period only, since values outside that period can be determined by the periodicity relations. For example, the original image $I(n_x, n_y)$ can be recovered on the square of size $M \times M$ from its ZT by:

$$I(i,j) = L^{-2} \sum_{\rho=0}^{L-1} \sum_{\sigma=0}^{L-1} (Z_L I)(i,j;\rho,\sigma). \tag{7}$$

The remainder of the image can then be recovered by using the periodicity relations given in Equations 5 and 6.

A second important property of the ZT is that it is *energy preserving*. That is, the energy of the ZT of an image is equal to the energy of the image itself (Zak, 1967; Zeevi & Gertner 1992). More generally, the inner product of the ZTs of two images is equal to the inner product of the images themselves, a property which often results in a computational saving when the inner product is used to compute expansion coefficients.

The third property is that *the ZT of an image is equal to the ZT of the FT of the image times a phase factor*:

$$\exp\left[-2\pi i\left(\frac{\rho i}{N} + \frac{\sigma j}{N}\right)\right](Z_L I)(i,j;-\rho,-\sigma) = M^{-2}(Z_M \hat{I})(\rho,\sigma;i,j), \tag{8}$$

where $\hat{I}$ is the $N \times N$-point Fourier transform of I, and $\exp\left[-2\pi i\left(\frac{\rho i}{N} + \frac{\sigma j}{N}\right)\right]$ is the phase factor. This property is a consequence of the fact that multiplying a complex function (such as the ZT) by a complex exponential (the phase factor) is equivalent to a rotation in the complex plane. From Equation 8 it follows that after rotation we get the ZT of the FT of the image. This property confirms that the ZT contains position and spatial frequency information about the image.

The fourth property of the ZT that is useful in image representation is that the ZT of a function [such as the basis function $h(n_x,n_y)$] located at a particular point in the position and spatial frequency domains, is equal to the ZT of the same function located at the origin in both domains times an appropriate phase factor. Formally this can be written as:

$$Z_L h_{k,l,m,n} = Z_L h_{0,0,0,0} \exp\left[-2\pi i\left(\frac{m\rho + n\sigma}{L} + \frac{ki + lj}{M}\right)\right] . \tag{9}$$

This property will be used below to develop a fast algorithm for decomposing images using Hermite functions. We will first, however, describe the specific Hermite functions to be used for this purpose.

Hermite Functions

We propose to represent images using a basis derived from the Hermite polynomials, $H_n(x)$, which can be defined as:

$$H_n(x) = (-1)^n \exp(x^2) \frac{d^n}{dx^n}[\exp(-x^2)] \ , \tag{10}$$

where $n = 0,1,2,....$ An orthonormal basis, which we will refer to as Hermite functions, $HF_n(x)$, is related to $H_n(x)$ as follows (Lebedev, 1972):

$$HF_n(x) = (2^n n! \sqrt{\pi})^{-1/2} \exp\left(\frac{-x^2}{2}\right) H_n(x), \qquad n = 0, 1, 2, ... \tag{11}$$

Gabor (1946) has shown that $HF_0(x)$ provides a local minimum of the joint uncertainty product, $\Delta x \cdot \Delta\omega$, where x and ω are position and spatial frequency variables, respectively. For other orders, $HF_n(x)$ can provide either maxima or minima (c.f., Klein & Beutter, 1992). We will use $HF_n(x,y) = HF_n(x) \cdot HF_n(y)$ as a basis for image decomposition in the position/spatial-frequency domain. Shown in Figure 3 are examples of the two-dimensional window function, $HF_n(x,y)$, corresponding to $n = 0,1,2,3,4,5$.

Fast Algorithms for Image Expansion

One advantage of the Hermite function approach is that it allows the fast computational procedure developed by Zeevi and Gertner (1992) to be used for image synthesis and analysis. We will now summarize the rationale of that procedure.

As a first step toward developing a fast computational procedure for calculating expansion coefficients and synthesizing an image from them, we will compute the ZT by representing both

the image, I, and the basis functions, h, as four dimensional arrays whose indices extend over dimensions smaller than that of the original image. Thus, for image size N (where $N = M \times L$), and $r,s = 0,1,...,L-1$, and $i,j = 0,1,...,M-1$, denote $I(i,r;j,s) = f(i+rM, j+sM)$, and $h_{k,l,m,n}(i,r;j,s) = h_{k,l,m,n}(i+rM, j+sL)$. We then represent the image as

$$I(i,r;j,s) = \sum_{k,l,m,n}^{LM} c_{k,l,m,n}\, h_{k,l,m,n}(i,r;j,s)\,. \tag{12}$$

The coefficients, c, are now the expansion coefficients in the position/spatial-frequency domain relative to the base function h. The second step in developing the fast computational procedure is based on first taking the ZT of both sides of Equation 12, resulting in:

$$Z_L I(i,j;\rho,\sigma) = \sum_{k,l,m,n}^{LM} c_{k,l,m,n}\, Z_L h_{k,l,m,n}(i,r;j,s) \quad . \tag{13}$$

In this form, computation of the coefficients would require a 4-D summation. However, property four of the ZT, described above, allows us to simplify the computation by substituting Equation 9 into Equation 13, and removing the ZT of h from the summation since it is no longer dependent on the indices of summation. The result

$$Z_L I(i,j;\rho,\sigma) = Z_L h_{0,0,0,0} \sum_{k,l,m,n}^{LM} c_{k,l,m,n}\, \exp\left[-2\pi i \frac{m\rho + n\sigma}{L} + \frac{ki + lj}{M}\right] \tag{14}$$

is the one sought in that it is in a form to which FFT algorithms can be applied.

The interpretation of Equation 14 is that the ZT of an image is equal to the ZT of the base function, located at the origin, times the Fourier transform of the expansion coefficients. Once the ZT of the base function, h, is computed, it can be stored and used on any image. It should be noted that, since the ratio of the ZT of the image and the ZT of the base function is required, care must be taken if the ZT of h has a zero. Techniques for dealing with this situation have been developed (Assaleh, Zeevi, & Gertner, 1991). In the context of the present technique, image

synthesis is performed by taking the Fourier transform of the coefficients, multiplying by the ZT of the base function, and then taking the inverse ZT of the result.

Image Expansion Using Hermite Functions

An example of a simple texture image decomposed using Hermite functions (HFs), and the Zak transform technique just described, is shown in Figure 4. The original 256 x 256 image is shown on the right, and the HFs used for the decomposition were of order 2, 3, and 4. A program for performing this Zak-Hermite decomposition is presented in Appendix 1. The reconstructions shown in Figure 4 were obtained from only 3,000 coefficients selected from the total of 65,536. As can be seen from the figure, the even-order HFs produced good reconstructions whereas the odd order HFs did not adequately recover the original texture. This result might be related to the observation of Klein and Beutter (1992) that certain HFs minimize the joint position/spatial-frequency uncertainty whereas others maximize it. It is well known that HFs are eigenfunctions of the Fourier transform, which means that the Fourier transform of a HF has the same form as the HF itself. The difference in the efficacy of the odd and even HFs in representing the image of Figure 4 may also be related to the fact that the eigenvalues corresponding to even-order HFs are real (either -1 or +1), while those corresponding to odd-order HFs are pure imaginary (either -i or +i). HFs with imaginary eigenvalues may not be good basis functions for image analysis and synthesis since their Fourier transform is rotated in the position domain by 90 degrees (this rotation is equivalent to multiplication by +i or -i) relative to the HF. As a result, the expansion coefficients are computed with respect to functions that are mismatched in the position and spatial frequency domains and which therefore may introduce noise in the calculation of the coefficients.

Figure 4
Image Decomposed Using Zak-Hermite Technique

IV. IMAGE REPRESENTATION USING A WEIGHTED ZAK TRANSFORM

Introduction

Gabor elementary functions (GEFs) have been used increasingly in image representation and synthesis (Daugman, 1988; Ebrahimi & Kunt 1991; Porat & Zeevi, 1988). Since GEFs do not form an orthogonal basis, a biorthogonal auxiliary function is required to determine the expansion coefficients (Bastiaans, 1981; Porat & Zeevi, 1988). Several approaches have been taken to solving the problem of calculating the required coefficients. For instance, Daugman (1988) and Ebrahimi and Kunt (1991) have suggested iterative techniques for accomplishing this.

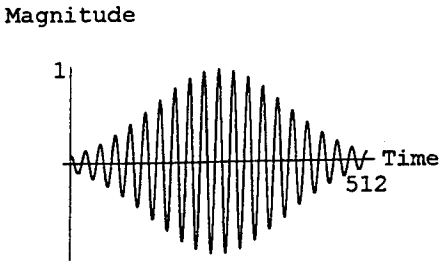
Zak (1967) showed that a signal could be transformed such that when the result is rotated by 90 degrees, the inverse transformation produces the Fourier transform (FT) of the original signal (see Figure 5). The Zak transform (ZT) has since been used to efficiently compute Gabor coefficients of images and to reconstruct images back from those coefficients (Zeevi & Gertner, 1992). Zeevi and Gertner (1992) took a more direct approach to the calculation of Gabor coefficients by developing a computationally efficient (4-D FFT) algorithm based on the Zak transform. However, it is well-known (Janssen, 1982) that if the Zak transform of a finite-energy function is continuous then it has a zero. Although Zeevi and Gertner's approach effectively avoided the "zero" of the Zak transform, it did not eliminate it, and so the method remained sensitive to noise. The problem can be illustrated as follows.

The Gabor representation of a one-dimensional signal, f(x) can be written as:

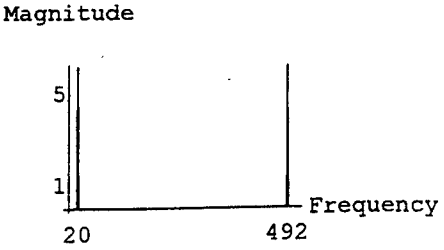
$$f(x) = \sum_m \sum_n a_{mn} \cdot g_{mn}(x) \quad , \tag{15}$$

where $g_{mn}(x)$ is a set of Gabor functions, and a_{mn} are the associated Gabor coefficients. By taking the ZT of both sides of this equation and using the property (Zak, 1967) that the ZT of a shifted (in both position and spatial frequency) gaussian is equal to the ZT of an unshifted gaussian multiplied by a phase factor, we get:

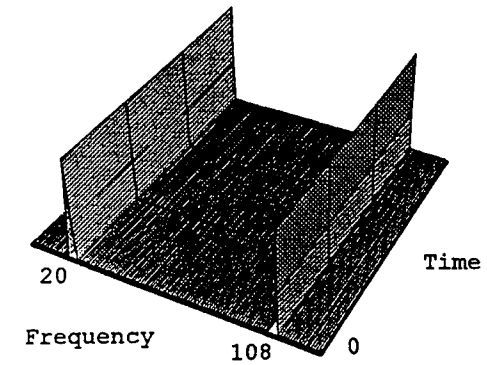
A) The Original Waveform.



B) The Fourier Transform.



C) The Zak Transform of the Original Waveform.



D) The Zak Transform of the Fourier Transform of the Original Waveform

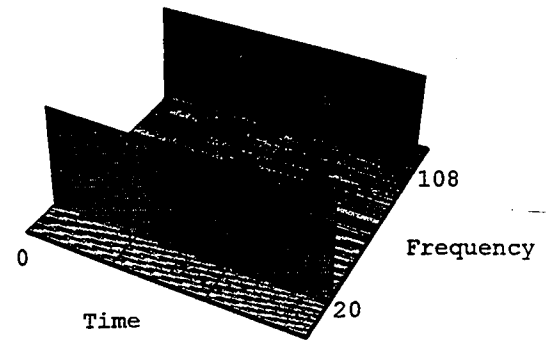


Figure 5
A Signal and Both Its Zak and Fourier Transforms

$$(Zf)(r,s) = (Zg_{00})(r,s) \sum_m \sum_n a_{mn} \cdot phase\ factor, \qquad (16)$$

where $g_{00}(x)$ is a gaussian function (i.e., a Gabor function of spatial frequency zero). This equation indicates that the Gabor coefficients can now be computed by taking the *inverse FT* of the ratio $Zf(x)/Zg_{00}(x)$. However, since $g_{00}(x)$ has a gaussian window, its Zak transform (see Figure 6a) has a zero which renders the computational ratio undefined and the computation of the coefficients unstable (i.e. noisy). The instability can result in an unacceptable time-frequency representation of the signal.

Previous attempts to address the zero problem have resulted in only approximate solutions. For instance, given a grid with a point at zero, another grid is chosen which is shifted by 1/2 in order to avoid sampling at zero. The problem with this approach is that while taking a large number of sampling points is desirable (in order to get as many coefficients as possible), it also results in sampling points close to zero, thus causing instability in the form of approximation errors near the (unsampled) zero point.

We have applied the mathematical concepts described by Daubechies (1990) and by Auslander, Gertner, and Tolimieri (1991) to develop a stable technique for calculating Gabor-like expansion coefficients. The technique allows a signal to be reconstructed from two sets of coefficients representing the even- and odd-indices, respectively, along one coordinate, and it avoids the zero problem discussed above. Further, the window function forming the basis used in the present decomposition procedure resembles a visual receptive field. Therefore, the expansion coefficients obtained with this technique may be relatable to cortical processes and, given their computational advantages, may represent an improvement over the weighting function outputs previously used in visual system models (Daugman, 1980, 1985; Fogel & Sagi, 1989; Malik & Perona, 1990; Stork & Wilson, 1990; Turner, 1986; Watson, 1983).

Approach

We first describe a formal procedure (Zeevi & Gertner, 1992) for computing stable expansion coefficients using the Zak transform. Denote a one-dimensional gaussian window function by:

$$g(x) = \exp(-\pi x^2) \quad . \tag{17}$$

The Zak transform (ZT) of any function, f, is a doubly-periodic function in two variables and is defined as:

$$(Zf)(r,s) = T^{-\frac{1}{2}} \sum_{l=-\infty}^{\infty} \exp(2\pi irl) \cdot f(T(s+l)) \quad , \tag{18}$$

where r and s are spatial frequency and position variables, respectively (Zak, 1967). Since the ZT is doubly-periodic with a fundamental period corresponding to the unit square, r and s each take on values in the interval 0 to 1.

Define a set of Gabor functions, $g_{mn}(x)$, which correspond to shifts in position (nq_0) and spatial frequency (mp_0):

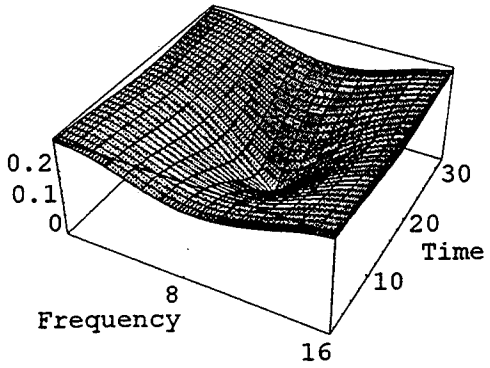
$$g_{mn}(x) \stackrel{\Delta}{=} \exp(imp_0 x) \cdot g(x - nq_0) \quad , \tag{19}$$

where $q_0=T/2$, $p_0=2/T$, and $g(x-nq_0)$ are window functions shifted in position. Next, compute the ZT of the set of (shifted) Gabor functions as:

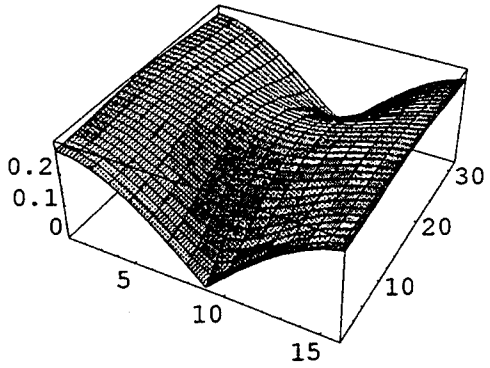
$$(Zg_{mn})(r,s) = T^{-\frac{1}{2}} \sum_{l=-\infty}^{\infty} \exp(2\pi irl) \cdot g_{mn}(T(s+l)) \quad . \tag{20}$$

By Equation 19:

A) Zak transform of Gaussian.



B) Zak transform of shifted Gaussian.



C) The weighting function.

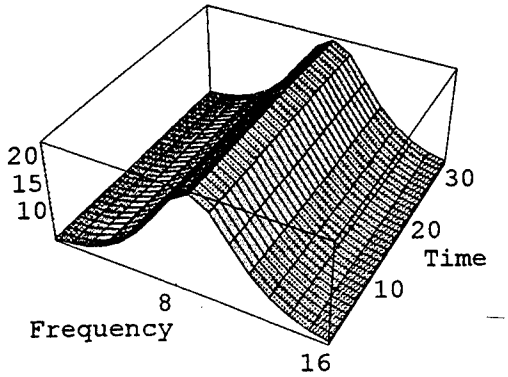


Figure 6
The Zak Transform of Two Gaussians and the Resultant Weighting Function

$$g_{mn}(T(s+l)) = \exp(2\pi i m(s+l)) \cdot g(T(s+l-\frac{n}{2})) \quad , \tag{21}$$

and substituting Equation 21 into Equation 20 gives:

$$(Zg_{mn})(r,s) = T^{-\frac{1}{2}} \sum_{l=-\infty}^{\infty} \exp(2\pi i r l) \cdot \exp(2\pi i m(s+l)) \cdot g(T(s+l-\frac{n}{2})) \quad . \tag{22}$$

As a first step toward obtaining an improved set of basis functions that avoids the zero-problem described earlier, we take the ZT of the set of shifted Gabor basis functions for two image subgrids. One subgrid corresponds to the even points:

$$(Zg_{m,2n_1})(r,s) = \exp(2\pi i n_1 r) \cdot \exp(2\pi i m s) \cdot (Zg)(r,s) \quad , \tag{23}$$

where $(Zg)(r,s)$ is the ZT of the unshifted, zero-frequency Gabor function, and the other subgrid corresponds to the odd points:

$$(Zg_{m,2n_1+1})(r,s) = \exp(2\pi i n_1 r) \cdot \exp(2\pi i m s) \cdot (Zg)(r,s-\tfrac{1}{2}) \quad , \tag{24}$$

where $(Zg)(r,s-\frac{1}{2})$ is the ZT of the shifted, zero-frequency Gabor function whose zero is now on the boundary of the unit square (see Figure 6b).

It can be shown (Daubechies, 1990) that the ZT of an image, f, can be written as:

$$\begin{aligned}(Zf)(r,s) = {} & (Zg)(r,s) \sum_m \sum_{n_1} a_{m,2n_1} \cdot \exp(2\pi i n_1 r) \cdot \exp(2\pi i m s) \\ & + (Zg)(r,s-\tfrac{1}{2}) \sum_m \sum_{n_1} a_{m,2n_1+1} \cdot \exp(2\pi i n_1 r) \cdot \exp(2\pi i m s) \, ,\end{aligned} \tag{25}$$

where the odd and even coefficients can be obtained by computing the inner product of the image with the function $g_{mn}(r,s)$ whose ZT is defined as:

$$Z\tilde{g}(r,s) = \frac{Zg(r,s)}{|Zg(r,s)|^2 + \left|Zg(r,s-\frac{1}{2})\right|^2} \quad . \tag{26}$$

where $|g|$ denotes the magnitude of the complex-valued function g. Using the unitarity (energy preserving) property of the ZT (Auslander *et al.*, 1991):

$$a_{mn} = \langle f, \tilde{g}_{mn} \rangle = \langle Zf, Z\tilde{g}_{mn} \rangle \quad . \tag{27}$$

Thus, the even coefficients for Equation 25 can be computed as:

$$a_{m,2n_1} = \int_0^1 \int_0^1 dr\, ds \exp(-2\pi i n_1 r) \cdot \exp(-2\pi i m s) \cdot Zf(r,s) \cdot \overline{Z\tilde{g}(r,s)} \quad , \tag{28}$$

where $\overline{Z\tilde{g}(r,s)}$ denotes the complex conjugate of the function $Z\tilde{g}(r,s)$. It can be seen from Equation 28 that the even coefficients are the FT of $Zf \cdot \overline{Z\tilde{g}}$. The odd coefficients can be computed similarly as the following FT:

$$a_{m,2n_1+1} = \int_0^1 \int_0^1 dr\, ds \exp(-2\pi i n_1 r) \cdot \exp(-2\pi i m s) \cdot Zf(r,s) \cdot \overline{Z\tilde{g}(r,s-\tfrac{1}{2})} \quad . \tag{29}$$

Once the even and odd coefficients are computed using Equations 28 and 29, the original signal can be reconstructed using Equation 25.

Discussion

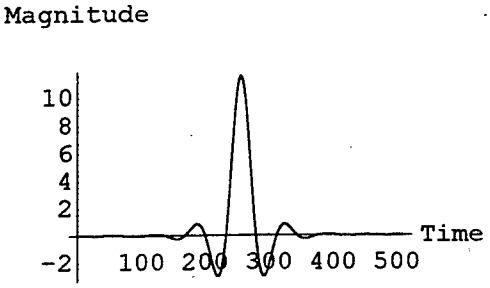
Gabor functions are used in image representation (Ebrahimi & Kunt, 1991; Manjunath & Chellappa, 1993; Porat & Zeevi, 1988) and visual system modeling (Daugman, 1980, 1985; Fogel & Sagi, 1989; Malik & Perona, 1990; Stork & Wilson, 1990; Turner, 1986; Watson, 1983) because they are well-localized in both position and spatial frequency, and they resemble certain

visual receptive fields. The major disadvantage of using Gabor functions is that they are not orthogonal and so do not provide the most efficient representation. Techniques have been developed (Bastiaans, 1981; Porat & Zeevi, 1988) for obtaining Gabor coefficients using biorthogonal functions, but these techniques are computationally intensive. A fast algorithm for computing Gabor coefficients has recently been developed (Gertner & Zeevi, 1990) based on the ZT. However, one property (Janssen, 1982) of the ZT is that when it is applied to a finite-energy, continuous function, the result has a zero in the unit square. This zero leads to instabilities in the calculation of expansion coefficients (see Introduction).

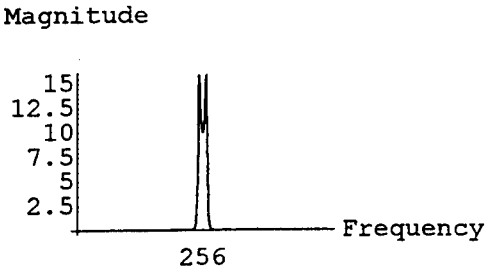
Shown in Figure 7a is a window function, $\tilde{g}(r)$, obtained by taking the inverse ZT of the function $Z\tilde{g}(r,s)$. The magnitude of the complex-valued function $Z\tilde{g}(r,s)$ is shown in Figure 7c. The function, $Z\tilde{g}(r,s)$, does not have a zero as is evident from the right side of Equation 26--the functions forming the denominator each have only one zero and each is in a different place. A graphical representation of the denominator of Equation 26 is shown in Figure 6c. The function, $Z\tilde{g}(r,s)$, is obtained when the denominator of Equation 26 is multiplied by $Zg(r,s)$. The denominator in Equation 26 is, in effect, a weighting function to $Zg(r,s)$, which removes the zero from the latter and which produces a symmetrical biphasic window function. The expansion coefficients of Equations 28 and 29 can, therefore, be computed and the computation is numerically stable. Further, in order to compute the expansion coefficients of a two-dimensional signal, it is necessary to perform a four-dimensional FT. Equations 28 and 29 are of a form to which an FFT can be applied, thus significantly reducing the computational effort required to obtain the expansion coefficients.

An ideal filter is one which uniformly passes some frequencies and completely rejects all others (Oppenheim & Willsky, 1983). The time-domain representation of an ideal filter is a sinc-function that typically displays significant side-lobes or ringing. As shown in Figure 7b, the FT of the window function, $g(r)$, approximates an ideal filter in that it has a relatively well-defined pass-band. The price paid for this property, however, is some oscillatory behavior in the time domain (see Figure 7a). Examples of a decomposition, with respect to $Z\tilde{g}(r,s)$, of both a spatially narrow and a spatially wide one-dimensional (time) signal, are shown in Figures 8 and 9, respectively. As is evident from these figures, the new window function can be used to obtain an

A) The new waveform.



B) The Fourier transform of new waveform.



C) The Zak transform of new waveform.

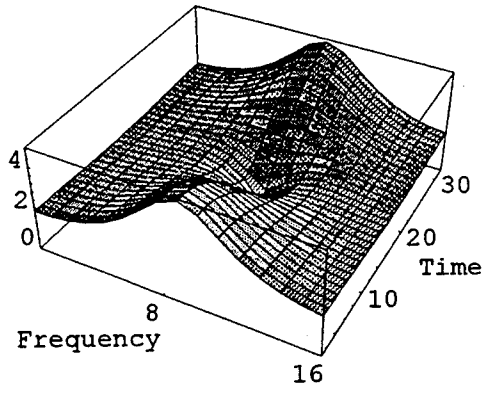
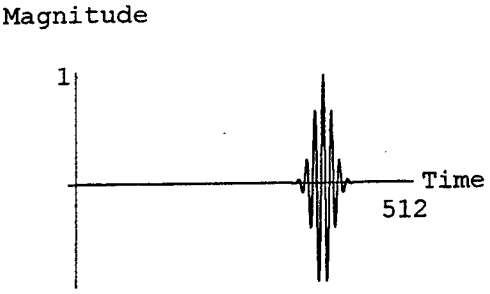
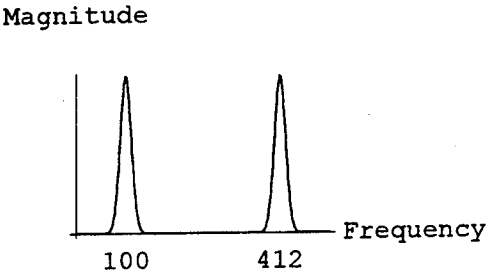


Figure 7
The New Waveform and Both Its Zak and Fourier Transforms

A) Input signal.



B) Fourier transform of input signal.



C) Expansion coefficients for new waveform.

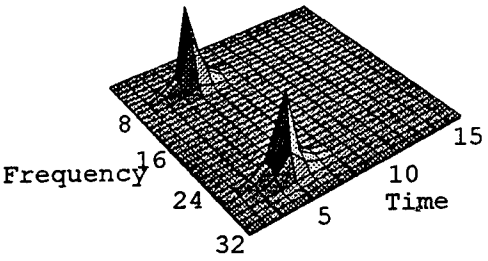
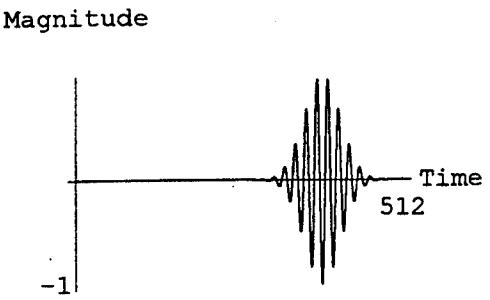
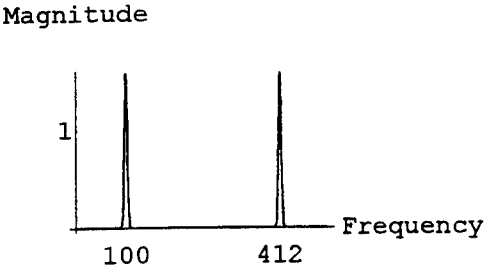


Figure 8
Decomposition of a Narrow Signal Using the New Waveform

A) Input Signal.



B) Fourier transform of input signal.



C) Expansion coefficients for new waveform.

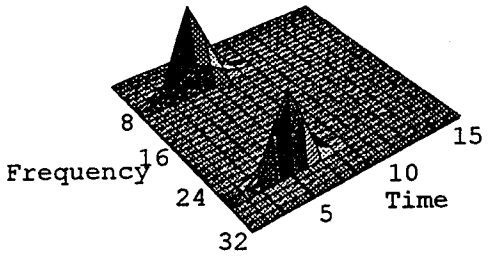


Figure 9
Decomposition of a Wide Signal Using the New Waveform

appropriate joint time/frequency representation that is consistent with the scaling property of the FT.

It should be noted that the basis functions generated using the new waveform, $g(r,s)$, as a window function are all of the same size. Although it may be more biologically relevant to use a pyramidally-scaled Gaborian basis (Daugman, 1988; Porat & Zeevi, 1988), computationally fast and stable techniques for performing such a decomposition have not to our knowledge been developed. While it is generally accepted that bases used in visual research should be spatially localized (Koenderink & van Doorn, 1990; Stork & Wilson, 1990), there are few theoretical criteria for choosing among the alternatives. One advantage, in this regard, of the window function derived here, is that it resembles a visual receptive field. Further, this window function can be easily scaled to give a basis suitable for multiresolution analysis, and it provides a mathematically rigorous, stable algorithm for computing expansion coefficients.

V. IMAGE REPRESENTATION USING A LOCALIZED COSINE TRANSFORM

Introduction

Like the Discrete Fourier Transform (DFT), the Discrete Cosine Transform (DCT) gives a spectral representation of an image. The global (spatial-frequency only) character of both the DFT and the DCT is not suitable for representing images whose spatial detail is not distributed homogeneously. This is the case for most natural images and thus there has recently been great interest in techniques that provide a combined representation in both position and spatial frequency (Jacobson & Wechsler, 1988; Zeevi & Gertner, 1992). One such combined technique is based on Gabor functions and represents images using complex-valued, Fourier-based transforms.

The purpose of the present study is to investigate the applicability of the DCT to the position/spatial frequency representation of images. For this purpose, a real, Discrete Zak-Cosine Transform (DZCT) has been developed and certain of its properties have been investigated. The DZCT provides a combined position/spatial frequency representation using only real variables (i.e., the local spectral characteristics of an image can be represented as an array of real numbers)

and hence is more efficient than Fourier-based approaches. The DZCT also provides a new perspective for localized image compression, in that it provides a local DCT whose spectral properties may be varied at each location in the image in order to match image characteristics. In addition, the DZCT has an inverse transform that reproduces the original signal, and thus can be used for image filtering and compression in coefficient space. Finally, since the DZCT is position sensitive, we can design filters that match the local properties of the image. The major benefit is that all computations are real, thus saving both memory and computational time.

The Discrete Zak-Cosine Transform

One form (Rao & Yip, 1990) of the DCT is:

$$F_k = \sum_{j=0}^{N-1} f_j \cdot \cos \frac{\pi k(2j+1)}{2N} \tag{30}$$

with inverse

$$f_j = \frac{1}{2} + \frac{2}{N} \sum_{k=1}^{N-1} F_k \cdot \cos \frac{\pi k(2j+1)}{2N} \quad . \tag{31}$$

This form is obtained by extending the given data from $j = 0, \ldots, N-1$, to $j = N, \ldots, 2N-1$ symmetrically around the point N-1/2 (Press, Teukolsky, Vetterling, & Flannery, 1993). This extension produces a sequence that is even about j = -1/2.

It has been shown (Zeevi & Gertner, 1992) that the Zak transform (ZT) can be used to adequately represent images in the combined position/spatial-frequency domain. Analogous to the manner in which Zeevi and Gertner computed the ZT using the DFT, we will define the ZT using the DCT as follows:

$$(Z_c f)(r,s) = \sum_{l=0}^{N-1} f(s+N \cdot l) \cdot \cos \frac{\pi l(2r+1)}{2N} \quad . \tag{32}$$

where r and s are spatial frequency and position variables, respectively. Thus, Equation 32 is the definition of the one-dimensional DZCT of the signal f.

Properties Of The Discrete Zak-Cosine Transform

The DZCT is a real transform that has many useful properties. For instance, while the DZCT is defined on the unit square only, it can be extended beyond the unit square using the following, easily verified, periodicity properties:

$$Z_c f(r+1,s) = Z_c f(r,s) \quad , \tag{33}$$

and

$$Z_c f(r,s+1) = \cos\frac{\pi(2r+1)}{2N} \cdot Z_c f(r,s) \quad . \tag{34}$$

Equations 33 and 34 indicate that the DZCT is periodic with period one in the r-variable, and is quasi-periodic with period one, in the s-variable.

In order to develop a combined position/spatial-frequency representation using a cosine basis, we will follow the development described in Zeevi and Gertner (1992). We will show below that the DCT can be used in place of the DFT in order to obtain the combined representation. To this end, we begin by defining a basis as:

$$g_{mn}(s) = \cos\frac{\pi s(2mp_0+1)}{2N} \cdot g(s-nq_0) \quad , \tag{35}$$

where g is a window function, nq_0 is a positional shift of the window, and mp_0 is the spatial frequency. We can then compute the DZCT of $g_{mn}(s)$ as:

$$(Z_c g_{mn})(r,s) = \sum_{l=0}^{N-1} \cos\frac{\pi l(2r+1)}{2N} \cdot g_{mn}(s+N\cdot l) \quad . \tag{36}$$

Performing some relevant algebraic manipulations (Zeevi & Gertner, 1992) on Equation 36 gives:

$$(Z_c g_{mn})(r,s) = (Z_c g)(r,s) \cdot \cos\frac{\pi m p_0 (2r+1)}{2N} \cdot \cos\frac{\pi n q_0 (2s+1)}{2N} \quad . \tag{37}$$

Thus, we can expand (Gertner & Geri, in press; Zeevi & Gertner, 1992) the signal f with respect to the basis shown as Equation 35 as:

$$f(s) = \sum_{mn} a_{mn} \cdot g_{mn}(s) \quad . \tag{38}$$

Then, by taking the DZCT of both sides and using the property shown as Equation 37, we get:

$$(Z_c f)(r,s) = \sum_{mn} a_{mn} \cdot (Z_c g_{mn})(r,s)$$

$$= (Z_c g)(r,s) \sum_{mn} a_{mn} \cdot \cos\frac{\pi m p_0 (2r+1)}{2N} \cdot \cos\frac{\pi n q_0 (2s+1)}{2N} \quad . \tag{39}$$

The coefficients a_{mn}, which are analogous to Gabor coefficients (Gabor, 1946), are computed by taking the inverse 2-D DCT of the ratio Z_f / Z_g.

Discussion

The DCT has been used in a wide range of digital signal processing applications including filtering, coding, compression, and classification (Rao & Yip, 1990). For instance, it has become a standard tool in the compression of both still and moving images (see, Pennebaker & Mitchell, 1993, and Appendix 2), and special hardware has been developed to implement it (Rao & Yip, 1990).

Unlike the DFT, the DCT involves only real numbers and thus requires less memory and less computational time to implement. Other techniques that have been suggested for

reconstructing images from real coefficients only (Oppenheim & Lim, 1981; Behar, Porat & Zeevi, 1992), do so using partial information. By contrast, the technique described in this section allows error-free image reconstruction.

VI. REFERENCES

Assaleh, K., Zeevi, Y.Y. & Gertner, I. (1991). On the realization of the Zak-Gabor representation of images, *SPIE Proc. Visual Comm. Image Process.* Boston.

Auslander, L., Gertner, I.C. & Tolimieri, R. (1991). The discrete Zak transform application to time-frequency analysis and synthesis of nonstationary signals. *I.E.E..E Trans. Signal Proc.* **39**, 825-835.

Bastiaans, M.J. (1981). A sampling theorem for the complex spectrogram, and Gabor's expansion of a signal in Gaussian elementary signals. *Optical Eng.* **20**, 594-598.

Behar, J., Porat, M. & Zeevi, Y.Y. (1992). Image reconstruction from localized phase. *IEEE Trans. Signal Process.* **40**, 736-743.

Bergmans, J.W.M. & Janssen, A.J.E.M. (1987). Robust data equalization, fractional tap spacing and the Zak transform. *Philips J. Res.*, **42**, 351-398.

Braddick, O., Campbell, F.W. & Atkinson, J. (1978). Channels in vision: Basic aspects. In *Handbook of sensory physiology*, vol 8. (pp 3-38). Heidelberg: Springer-Verlag.

Daubechies, I. (1988). Orthonormal bases of compactly supported wavelets. *Comm. Pure Appl. Math.*, **41**, 909-996.

Daubechies, I. (1990). The wavelet transform, time-frequency localization and signal analysis. *IEEE Trans. Inform. Theory*, **36**, 961-1005.

Daubechies, I. (1992). *Ten Lectures on Wavelets.* SIAM, Philadelphia, pp. 53-105

Daugman, J.G. (1980). Two-dimensional spectral analysis of cortical receptive field profiles. *Vision Res.*, **20**, 847-856.

Daugman, J.G. (1985). Uncertainty relation for resolution in space, spatial frequency, and orientation optimized by two-dimensional visual cortical filters. *J. Opt. Soc. Am. A*, **2**, 1160-1169.

Daugman, J.G. (1988). Complete discrete 2-D Gabor transforms by neural networks for image analysis and compression. *IEEE Trans. Acoust. Speech and Signal Process.* 36, 1169-1179.

Ebrahimi, T. & Kunt, M. (1991). Image compression by Gabor expansion. *Optical Eng.* **30**, 837-880.

Fogel, I. & Sagi, D. (1989). Gabor filters as texture discriminator. *Biol. Cybern.* **61**, 103-113.

Gabor, D. (1946). Theory of communication. *J. I.E.E.* **93**, 429-459.

Geri, G.A., Zeevi, Y.Y. & Porat, M. (1990). Efficient image generation using localized frequency components matched to human vision. (*AFHRL-TR-90-25, AD A224 903*). Williams Air Force Base AZ: Operations Training Division, Air Force Human Resources Laboratory.

Gertner, I.C. & Geri, G.A. (in press). Image representation using Hermite functions. *Biol. Cybern.*

Gertner, I. & Zeevi, Y.Y. (1990). On the Zak-Gabor representation of images. *SPIE Proc. Vis. Comm. Image Process.* **1360**, 1738-1748.

Jacobson, L.D. & Wechsler, H. (1988). Joint spatial/spatial-frequency representation. *Signal Processing,* **14**, 37-68.

Janssen, A.J.E.M. (1982). Bargmann transform, Zak transform, and coherent states. *J. Math. Phys.*, **23**, 720-731.

Janssen, A.J.E.M. (in press). Signal analytic proofs of two basic results on lattice expansions. *Applied Computational Harmonic Analysis.*

Jones, J.P. & Palmer, L.A. (1987). An evaluation of the two-dimensional Gabor filter model of simple receptive fields in cat striate cortex. *J. Neurophysiol.* **58**, 1233-1258.

Klein, S.A. & Beutter, B. (1992). Minimizing and maximizing the joint space-spatial frequency uncertainty of Gabor-like functions. *J. Opt. Soc. Am. A*, **9**, 337-340.

Koenderink, J.J. (1984). The structure of images. *Biol. Cybern.* **50**, 363-370.

Koenderink, J.J. & van Doorn, A.J. (1990). Receptive field families. *Biol. Cybern.* **63**, 291-297.

Kulikowski, J.J., Marcelja, S. & Bishop, P.O. (1982). Theory of spatial position and spatial frequency relations in the receptive fields of simple cells in the visual cortex. *Biol. Cybern.* **43**, 187-198.

Lebedev, N.N. (1972). *Special Functions and Their Applications*. New York: Dover Publications.

Malik, J. & Perona, P. (1990). Preattentive texture discrimination with early vision mechanisms. *J.Opt. Soc. Am. A*, 7, 923-932.

Manjunath, B.S. & Chellappa, R. (1993). A unified approach to boundary perception: Edges, textures, and illusory contours. *I.E.E.E. Trans. Neural Networks*, **4**, 96-108.

Marcelja, S. (1980). Mathematical description of the responses of simple cortical cells. *J. Opt. Soc. Am.* **70**, 1297-1300.

Marr, D. (1982). *Vision*. New York: W.H.Freeman.

Marr, D. & Hildreth, E. (1980). Theory of edge detection. *Proc. Roy. Soc. Lond. B,* **207**, 187-217.

Martens, J.-B. (1990). The Hermite transform--Theory. *I.E.E.E. Trans. Acoust. Speech and Signal Process.* **38**, 1595-1606.

Oppenheim, A.V. & Lim, J.S. (1981). The importance of phase in signals. *Proc. IEEE*, **69**, 529-541.

Oppenheim, A.V. & Willsky, A.S. (1983). *Signals and Systems* (p. 401). Englewood Cliffs: Prentice-Hall.

Pennebaker, W.B. & Mitchell, J.L. (1993). *JPEG Still Image Compression Standard.* New York: Van Nostrand Reinhold.

Porat, M. & Zeevi, Y.Y. (1988). The generalized Gabor scheme of image representation. *I.E.E.E. Trans. Pattern Anal. Machine Intell.* **10**, 452-468.

Press, W.H., Teukolsky, S.A., Vetterling, W.T. & Flannery, B.P. (1993). *Numerical Recipes in C,* (2nd ed.), Cambridge: Cambridge University Press.

Rao, K.R. & Yip, P. (1990). *Discrete Cosine Transform*. San Diego: Academic Press.

Rodieck, R.W. & Stone, J.S. (1965). Analysis of receptive fields of cat retinal ganglion cells. *J. Neurophysiol.* **28**, 965-980.

Stork, D.G. & Wilson, H.R. (1990). Do Gabor functions provide appropriate descriptions of visual cortical receptive fields? *J. Opt. Soc. Am. A* **7**, 1362-1373.

Turner, M.R. (1986). Texture discrimination by Gabor functions. *Biol. Cybern.* 55, 71-82.

Watson, A.B. (1983). Detection and recognition of simple spatial forms, *Physical and Biological Processing of Images*, (pp. 100-113). In O.J. Braddick and A.C. Sleigh (Eds.), New York: Springer-Verlag.

Yang, J. (1992). Do Gabor functions provide appropriate descriptions of visual cortical receptive fields?: comment. *J. Opt. Soc. Am. A*, **9**, 334-336.

Young, R.A. (1987). The Gaussian derivative model for spatial vision: I. Retinal mechanisms. *Spatial Vision*, **2**, 273-293.

Zak, J. (1967). Finite translations in solid-state physics. *Phys. Rev. Lett.* **19**, 1385-1397.

Zeevi, Y.Y. & Gertner, I. (1992). The finite Zak transform: An efficient tool for image representation and analysis. *J. Visual Comm. & Image Represent.* **3**, 13-23.

Zucker, S.W. & Hummel, R.A. (1986). Receptive fields and the representation of visual information. *Human Neurobiol.* **5**, 121-128.

VII. APPENDICES

Appendix 1:

Source code for the Zak-Hermite decomposition program, which consists of the following modules:

1) *image.c* -- the main module which calls other modules, defines menus and associated display windows, and defines all constants for display colors, the keyboard, window memory management, and the cursor,
2) *herm.c* -- module that performs the Zak transform using a Hermite-function window. The routine *fourn.c* (Press *et al.*, 1993) is used to perform the FFT.
3) *disp_ima.c* -- Module that displays a bitmapped, binary image in low resolution (320 x 200 x 8 bits). Image may be clipped depending on its size.
4) *cmp_tima.c* -- Module that produces graphics display using Trident graphics display card (or any compatible). Four graphics windows are placed on the screen. Total display resolution is 640 x 480 x 8 bits, so four 256 x 256 images can be displayed.
5) *5x8.inc* -- Module that defines a bitmap for each alphanumeric character needed to produce SuperVGA-resolution graphics using the Trident card. This module is required because the NDP compiler does not support the Trident card in SuperVGA resolution.
6) *prnt_ima.c* -- Module used to output a 256 x 256 image to a HP Laserjet (or compatible) printer.
7) *util.c* -- Module library for defining operations with complex numbers, and memory allocation for image and coefficient arrays.

```
/*****************************************************************/
/***********  Hermite/Gabor/Wavelets Transformation ************/
/*****************************************************************/
/***       University of Dayton Research Institute          ***/
/*****************************************************************/
/***  Developed by  : Alex Firdman                           ***/
/***  Developed on  : Oct. 11, 1993                          ***/
/***  Compiler used : NDP-C Ver. 4.2.1                       ***/
/***  Libraries     : Standard, GREX                         ***/
/*****************************************************************/

#define BLACK         0
#define BLUE          1
#define GREEN         2
#define CYAN          3
#define RED           4
#define MAGENTA       5
#define BROWN         6
#define LIGHTGRAY     7    /* Color Definition */
#define DARKGRAY      8
#define LIGHTBLUE     9
#define LIGHTGREEN   10
#define LIGHTCYAN    11
#define LIGHTRED     12
#define LIGHTMAGENTA 13
#define YELLOW       14
#define WHITE        15

#define ARRUP   -72     /*                              */
#define ARRDN   -80     /*   Keaboard KEYS definition   */
#define ARRRGHT -77     /*                              */
#define ARRLFT  -75     /*   User interface using       */
#define ENTER    13     /*   Arrows  UP and DN          */
#define PGDN    -81     /*                              */
#define PGUP    -73     /*                              */
#define F1      -59     /*   F1 - used for HELP         */

#define EXIT    4       /*   Exit on 'Exit to DOS'      */

#define TRUE    1
#define FALSE   0

#define HOR_SIZE 80
#define VER_SIZE 25
#define VID_BUFF 4000   /* Screen Video BUFFER */

#define VIDEO 0x010     /* Interrupt Call for Video functions */

#define OFF      0      /* Turns the cursor OFF */
#define ON_NORM  2      /* Reset Cursor to normal size */
#define ON_BLOCK 1      /* Create a Block Cursor shape */

#define NULL 0

#include <ctype.h>
```

FILE=image.c Thu Apr 21 11:58:44 1994 PAGE=1

```
  56: #include <stdio.h>
  58: #include <stdlib.h>
  59: #include <os.h>
  60: #include <conio.h>
  61: #include <string.h>
  62: #include <grex.h>
  63: #include <math.h>
  64:
  65: /**  Functions  Prototypes  **/
  66:
  67: char *StrIn(int,int,int,char *);
  68: int ch2int(void);
  69: char *int2ch(int);
  70: char Wrt_Ch(int,int,int);
  71:
  72: void Wrt_Str(int,int,char *,int,int);
  73: void cursoronof(int);
  74: int menu_bar_v(int,int,int,int,int,int,int,char *[20]);
  75: void put_window(int,int,int,int,int,int,char*,int,int);
  76: short *save_scr(int, int, int, int);
  77: void rest_scr(int, int, short*);
  78: void error_message(int,int,int,int,char*);
  79:
  80: void strext(char[],char[],char[]);
  81:
  82: void Help(void);
  83: void Hermite(void);
  84: void Wavelet(void);
  85: void Utility(void);
  86: int disp_image(char *, int);
  87: int comp_image(char *, char *, int, char *);
  88:
  89: void Exit(void);
  90:
  91: FILE *fpout, *fpin;
  92:
  93: main(void)
  94: {
  95: int Color;
  96: int V_ret;        /*  Item chosen from Main Menu       */
  97: int i;
  98: short *main;
  99:
 100: char *item[4];   /*  Array containing Menu Entries for Ver. Menus */
 101:
 102: cursoronof(OFF);  /* Turn the Cursor OFF */
 103:
 104:     item[0] = "  Hermite Function Decomposition   ";
 105:     item[1] = "  Orthogonal Wavelet Decomposition ";
 106:     item[2] = "  Utility Menu                     ";
 107:     item[3] = "  Exit to DOS                      ";
 108:
 109: do {
 110:
 111: _setbkcolor(LIGHTGRAY);
```

```
112: _settextwindow(1,1,80,25);
114: clear_text();
115:
116: Wrt_Str(1,1,"                          * Armstrong Laboratories *                          ",WHITE,BLUE);
117: Wrt_Str(9,7,"        I M A G E   A N A L Y S I S   P R O G R A M         ",WHITE,BLUE);
118: Wrt_Str(1,23,"                    * University of Dayton Research Institute *                ",WHITE,BLUE);
119:
120:     put_window(19,10,40,9,0x1f,3," MAIN MENU ",BLUE,WHITE);
121:
122:     V_ret = menu_bar_v(21,12,4,WHITE,RED,BLUE,YELLOW,item);
123:
124:     if (V_ret == 1)
125:     {
126:         Hermite();
127:     }
128:     else if (V_ret == 2)
129:     {
130:         Wavelet();
131:     }
132:     else if (V_ret == 3)
133:     {
134:         Utility();
135:     }
136:     else /* (V_ret == 4) exit the program */
137:     {
138:         Exit();
139:     }
140:   } while (V_ret != 4);
141: }
142:
143: int ch2int(void)
144: {
145: char CH;
146: int DIGIT[6];
147: int INT;
148: int counter=0;
149:
150:     do {
151:          counter++;
152:          CH = getche();
153:          DIGIT[counter] = (int) CH - 48;
154:
155:          if (CH == ENTER)
156:                counter = 0;
157:
158:          switch(counter)
159:          {
160:         case 1 :
161:             INT = DIGIT[counter];
162:             break;
163:         case 2 :
164:             INT = DIGIT[counter-1]*10 + DIGIT[counter];
165:             break;
166:         case 3 :
167:             INT = DIGIT[counter-2]*100 + DIGIT[counter-1]*10 +
```

```
168:                    DIGIT[counter];
170:            break;
171:        case 4 :
172:            INT = DIGIT[counter-3]*1000 + DIGIT[counter-2]*100 +
173:                  DIGIT[counter-1]*10 + DIGIT[counter];
174:            break;
175:        case 5 :
176:            INT = DIGIT[counter-4]*10000 + DIGIT[counter-3]*1000 +
177:                  DIGIT[counter-2]*100 + DIGIT[counter-1]*10 +
178:                  DIGIT[counter];
179:            CH = ENTER;
180:            break;
181:
182:         }
183:       } while(CH != ENTER);
184:
185: return INT;
186: }
187:
188: int IntIn(Fcol,Frow,Color)
189: int Fcol;
190: int Frow;
191: int Color;
192: {
193: int DATA;
194:
195: cursoronof(ON_BLOCK);
196:
197:     _settextcolor(Color);
198:     _setbkcolor(LIGHTGRAY);
199:     locate(Fcol,Frow);
200:     DATA = ch2int();
201:
202: cursoronof(OFF);
203:
204: return DATA;
205: }
206:
207: char *StrIn(Fcol,Frow,Color,ext)
208: int Fcol;
209: int Frow;
210: int Color;
211: char *ext;
212: {
213: char name[20]="\0";
214: char *STR=NULL;
215:
216:     cursoronof(ON_BLOCK);
217:     _settextcolor(Color);
218:     locate(Fcol,Frow);
219:
220:     gets(name);
221:
222:     STR = strcat(name,ext);
223:
```

```
224: cursoronof(OFF);
226: return STR;
227: }
228:
229: char Wrt_Ch(locx,locy,Tcolor)
230: int locx;
231: int locy;
232: int Tcolor;
233: {
234: char CH = '\0';
235:
236: cursoronof(ON_BLOCK);
237:
238:  locate(locx,locy);
239:  _settextcolor(Tcolor);
240:
241:  write_one_char(CH=getch());
242:  pauseb();
243:
244: cursoronof(OFF);
245:
246: return CH;
247: }
248:
249: void Wrt_Str(Fcol,Frow,message,color,bkgrd)
250: int Fcol;
251: int Frow;
252: char *message;
253: int color;
254: int bkgrd;
255: {
256: cursoronof(ON_BLOCK);
257:
258:  _setbkcolor(bkgrd);
259:  _settextcolor(color);
260:
261:  locate(Fcol,Frow);
262:  write_string(message);
263:
264: cursoronof(OFF);
265: }
266:
267: int menu_bar_v(Fcol,Frow,NumEntr,Color,Bkgrd,Bar_bgd,Bar_col,item)
268: int Fcol;
269: int Frow;
270: int NumEntr;
271: int Color;
272: int Bkgrd;
273: int Bar_bgd;
274: int Bar_col;
275: char *item[20];
276: {
277:
278: int Entry;
279: int c;
```

```
280: int y;
282: int i;
283: int FcolOld;
284:
285: for(i=0;i<NumEntr;i++)
286: {
287:     Wrt_Str(Fcol,Frow+i,item[i],Color,Bar_bgd);
288: }
289:
290: Wrt_Str(Fcol,Frow,item[0],Bar_col,Bkgrd);
291:
292: do
293:   {
294:     FcolOld = Fcol;
295:     get_cursor(&Fcol,&y);
296:     Fcol = FcolOld;
297:     c = pauseb();
298:
299:     switch (c)
300:        {
301:        case ARRUP :
302:            if (y == Frow)
303:            {
304:                Wrt_Str(Fcol,y,item[y-Frow],Color,Bar_bgd);
305:                Wrt_Str(Fcol,y+NumEntr-1,item[y-Frow+NumEntr-1],Bar_col,Bkgrd);
306:            }
307:
308:            if ((y <= Frow+NumEntr-1) && (y > Frow))
309:            {
310:                Wrt_Str(Fcol,y,item[y-Frow],Color,Bar_bgd);
311:                Wrt_Str(Fcol,y-1,item[y-Frow-1],Bar_col,Bkgrd);
312:            }
313:            break;
314:
315:        case ARRDN :
316:            if (y == Frow+NumEntr-1)
317:            {
318:                Wrt_Str(Fcol,y,item[y-Frow],Color,Bar_bgd);
319:                Wrt_Str(Fcol,y-NumEntr+1,item[y-Frow-NumEntr+1],Bar_col,Bkgrd);
320:            }
321:
322:            if ((y < Frow+NumEntr-1) && (y >= Frow))
323:            {
324:                Wrt_Str(Fcol,y,item[y-Frow],Color,Bar_bgd);
325:                Wrt_Str(Fcol,y+1,item[y-Frow+1],Bar_col,Bkgrd);
326:            }
327:            break;
328:
329:        case ENTER :
330:            Entry = y;
331:            Wrt_Str(Fcol,y,item[y-Frow],Color,Bar_col);
332:            break;
333:        }
334:
335:  }  while (c != ENTER);
```

```
336:
338: return (Entry-Frow+1);
339: }
340:
341: /* Controls Cursor type */
342: void cursoronof(togle)
343: int togle;
344: {
345:  union REGS16 reg;
346:
347: if (togle == OFF)   /* No Cursor */
348: {
349:  reg.h.ah = 1;
350:  reg.h.ch = -1;
351:  reg.h.cl = -1;
352:
353:  int86(0X10, &reg, &reg);
354: }
355: else if (togle == ON_BLOCK)  /* Block Cursor */
356:  {
357:     reg.h.ah = 1;
358:     reg.h.ch = 0x01;
359:     reg.h.cl = 0x0c;
360:
361:     int86(0x10, &reg, &reg);
362:  }
363:  else                /* Normal Cursor */
364:    {
365:     reg.h.ah = 1;
366:     reg.h.ch = 0x0b;
367:     reg.h.cl = 0x0c;
368:
369:     int86(0x10, &reg, &reg);
370:    }
371:
372: }
373:
374: void Exit(void)
375: {
376:     _setbkcolor(BLACK);
377:     _settextcolor(LIGHTGRAY);
378:
379:     clear_text();
380:     Wrt_Str(20,12,"** Program terminated by User's Request **",LIGHTGRAY,BLACK);
381:     cursoronof(ON_NORM);
382: exit(0);
383: }
384:
385: void Help()
386: {
387: }
388:
389: void Wavelet()
390: {
391: char infile[20]   = "\0";
```

```
392: char outfile1[20] = "\0";
394: char outfile2[20] = "\0";
395: int n, i;
396: int level;
397: char *itemW[5];
398: int M1, N1;
399: unsigned int found=1, attribute=_A_NORMAL;
400: struct find_t fileinfo;
401: char *ext=".img";
402:
403:     itemW[0] = " Enter Image Input File Name         ";
404:     itemW[1] = " Enter Decomposed Image File Name    ";
405:     itemW[2] = " Enter Reconstructed Image File Name ";
406:     itemW[3] = " Enter Image Size (64, 256 etc.)     ";
407:     itemW[4] = " Enter Level of Decompositions       ";
408:
409:     put_window(6,5,68,11,0x1f,3," Wavelet Transformations ",BLUE,YELLOW);
410:
411:     Wrt_Str(8,6,itemW[0],WHITE,LIGHTGRAY);
412:     Wrt_Str(60,6,"        .IMG",WHITE,LIGHTGRAY);
413:
414:     Wrt_Str(8,8,itemW[1],WHITE,LIGHTGRAY);
415:     Wrt_Str(60,8,"        .IMG",WHITE,LIGHTGRAY);
416:
417:     Wrt_Str(8,10,itemW[2],WHITE,LIGHTGRAY);
418:     Wrt_Str(60,10,"        .IMG",WHITE,LIGHTGRAY);
419:
420:     Wrt_Str(8,12,itemW[3],WHITE,LIGHTGRAY);
421:     Wrt_Str(60,12,"    ",WHITE,LIGHTGRAY);
422:
423:     Wrt_Str(8,14,itemW[4],WHITE,LIGHTGRAY);
424:     Wrt_Str(60,14,"    ",WHITE,LIGHTGRAY);
425:
426:
427:     strcpy(infile,StrIn(60,6,BLUE,ext));
428:
429:     found = _dos_findfirst((char *)infile,attribute,&fileinfo);
430:
431:         if ((strlen(infile) == NULL) || (found != 0))
432:         {
433:            cursoronof(OFF);
434:            error_message(30,12,30,5," Image File not found ...");
435:            return;
436:         }
437:
438:     strcpy(outfile1,StrIn(60,8,BLUE,".img"));
439:     strcpy(outfile2,StrIn(60,10,BLUE,".img"));
440:
441:     n = IntIn(60,12,BLUE);
442:
443:     M1 = (int) (sqrt((double) (n)));
444:     N1 = (int) n/M1;
445:
446:         if ((n < 0) || (M1 != N1) || (M1*N1 != n))
447:         {
```

```
448:            cursoronof(OFF);
450:            error_message(20,12,50,5,"Image Size is not accepted type..");
451:            return;
452:          }
453:
454:      level = IntIn(60,14,BLUE);
455:
456:      wavelts(infile, outfile1, outfile2, n, level);
457:
458: cursoronof(OFF);
459: }
460:
461: void Utility()
462: {
463: int menu_item=0;
464:
465: char *item[5] ;        /*      Array containing Menu Entries   */
466: char *item1[2];
467: char *item2[4];
468: char *pr_item[3];
469:
470: char f_size[13];
471:
472: char  im_name[20]  = "\0";
473: char  im_name1[20] = "\0";
474: char  im_name2[20] = "\0";
475: char  im_name3[20] = "\0";
476:
477: int im_size=0;
478: char im_type=' ';
479: int not_print = FALSE;
480: int prnt_result;
481:
482: unsigned int found=1, attribute=_A_NORMAL;
483: struct find_t fileinfo;
484:
485: div_t mdiv;
486:
487: int M1, N1;
488: int i=0;
489: int m=0;
490:
491: int done;
492: int right;
493:
494: int exitit;
495:
496: char *title = " Utilities for Image Examination ";
497: char *dir = "*.img";
498:
499:      exitit = FALSE;
500:      put_window(10,12,40,9,0x1f,3,title,BLUE,YELLOW);
501:
502:      item[0] = "  Display Directory of Image Files";
503:      item[1] = "  Display Image on Screen         ";
```

```
504:     item[2] = "  Print Image on Laser Printer    ";
506:     item[3] = "  Compare Two Images              ";
507:     item[4] = "  Exit to Main Menu               ";
508:
509:     menu_item = menu_bar_v(12,14,5,WHITE,LIGHTGRAY,BLUE,YELLOW,item);
510:
511:     if (menu_item == 1)   /* Display Image File Directory */
512:     {
513:         put_window(2,3,75,19,0x1f,3," Image Files Available ",BLUE,YELLOW);
514:
515:         done = _dos_findfirst(dir,attribute,&fileinfo);
516:         Wrt_Str(4,4,fileinfo.name,WHITE,BLUE);
517:         ltoa(fileinfo.size,f_size,10);
518:         Wrt_Str(18,4,f_size,WHITE,BLUE);
519:
520:         while((done == 0) || (i > 51))
521:         {
522:             i++;
523:
524:             mdiv = div(i,17);
525:
526:             if (mdiv.rem == 0)
527:             {
528:                i=0;
529:                m++;
530:             }
531:
532:             if (i < 51)
533:             {
534:                 Wrt_Str(m*21+4,i+4,fileinfo.name,WHITE,BLUE);
535:                 ltoa(fileinfo.size,f_size,10);
536:                 Wrt_Str(m*21+18,i+4,f_size,WHITE,BLUE);
537:             }
538:             done = _dos_findnext(&fileinfo);
539:         }
540:
541:         while(getche() != ENTER);
542:     }
543:
544:     if (menu_item == 2)   /* Display Image in 256x256 Maximum */
545:     {
546:         item1[0] = "  Enter Image File Name           ";
547:         item1[1] = "  Enter Image Size                ";
548:
549:         put_window(12,14,60,8,0x1f,3," Displaying Image ",BLUE,YELLOW);
550:
551:      do {
552:         right = TRUE;
553:
554:         Wrt_Str(14,16,item1[0],WHITE,LIGHTGRAY);
555:         Wrt_Str(54,16,"        .IMG",WHITE,LIGHTGRAY);
556:
557:         Wrt_Str(14,18,item1[1],WHITE,LIGHTGRAY);
558:         Wrt_Str(54,18,"     ",WHITE,LIGHTGRAY);
559:
```

```
560:         strcpy(im_name,StrIn(54,16,BLUE,".img"));
562:         found = _dos_findfirst(im_name,attribute,&fileinfo);
563:
564:         if ((strlen(im_name) == NULL) || (found != 0))
565:         {
566:            cursoronof(OFF);
567:            error_message(30,12,30,5," Image File not found ...");
568:            return;
569:         }
570:
571:         im_size = IntIn(54,18,BLUE);
572:
573:         M1 = (int) (sqrt((double) (im_size)));
574:         N1 = (int) im_size/M1;
575:
576:         if ((im_size < 0) || (M1 != N1) || (M1*N1 != im_size))
577:         {
578:            right = FALSE;
579:            cursoronof(OFF);
580:            error_message(20,12,50,5,"Image Size is not accepted type..");
581:            return;
582:         }
583:         right = TRUE;
584:      } while ( right != TRUE );
585:
586:         disp_image(im_name,im_size);
587:
588:         cursoronof(OFF);
589:      }
590:      else if (menu_item == 3)  /* Print Image on Laser Jet */
591:      {
592:         pr_item[0] = "  Enter Image File Name          ";
593:         pr_item[1] = "  Enter Image Size               ";
594:
595:         put_window(12,14,60,8,0x1f,3," Printing Image on Laser Printer ",BLUE,YELLOW);
596:
597:       do {
598:         right = TRUE;
599:
600:         Wrt_Str(14,16,pr_item[0],WHITE,LIGHTGRAY);
601:         Wrt_Str(54,16,"       .IMG",WHITE,LIGHTGRAY);
602:
603:         Wrt_Str(14,18,pr_item[1],WHITE,LIGHTGRAY);
604:         Wrt_Str(54,18,"    ",WHITE,LIGHTGRAY);
605:
606:         strcpy(im_name,StrIn(54,16,BLUE,".img"));
607:         found = _dos_findfirst(im_name,attribute,&fileinfo);
608:
609:         if ((strlen(im_name) == NULL) || (found != 0))
610:         {
611:            cursoronof(OFF);
612:            error_message(30,12,30,5," Image File not found ...");
613:            right = FALSE;
614:            return;
615:         }
```

```
616:
618:          im_size = IntIn(54,18,BLUE);
619:
620:          M1 = (int) (sqrt((double) (im_size)));
621:          N1 = (int) im_size/M1;
622:
623:          if ((im_size < 0) || (M1 != N1) || (M1*N1 != im_size) ||
624:              (im_size > 256))
625:          {
626:             cursoronof(OFF);
627:             error_message(20,12,50,5,"Image Size is not accepted type..");
628:             not_print = TRUE;
629:             right = FALSE;
630:             return;
631:          }
632:
633:          prnt_result = _bios_printer(2, 0, 0);
634:
635:          if(prnt_result == 16)
636:          {
637:             cursoronof(OFF);
638:             error_message(20,12,35,5,"Check PRINTER, it is not ON..");
639:             not_print = TRUE;
640:             right = FALSE;
641:             return;
642:          }
643:          else
644:             not_print = FALSE;
645:
646:          if (!not_print)
647:             prnt_ima(im_name,im_size);
648:
649:        right = TRUE;
650:
651:        } while ( right != TRUE );
652:          cursoronof(OFF);
653:      }
654:      else if (menu_item == 4)  /* Comparing Two Images */
655:      {
656:
657:          put_window(14,10,60,12,0x1f,3," Comparing Two Images ",BLUE,YELLOW);
658:
659:          item2[0] = "  Enter 1st Image File Name       ";
660:          item2[1] = "  Enter 2nd Image File Name       ";
661:          item2[2] = "  Enter Image Size                ";
662:          item2[3] = "  Enter Coefficient File Name     ";
663:
664:       do {
665:          right = TRUE;
666:
667:          Wrt_Str(16,12,item2[0],WHITE,LIGHTGRAY);
668:          Wrt_Str(56,12,"        .IMG",WHITE,LIGHTGRAY);
669:
670:          Wrt_Str(16,14,item2[1],WHITE,LIGHTGRAY);
671:          Wrt_Str(56,14,"        .IMG",WHITE,LIGHTGRAY);
```

```
672:
674:          Wrt_Str(16,16,item2[2],WHITE,LIGHTGRAY);
675:          Wrt_Str(56,16,"     ",WHITE,LIGHTGRAY);
676:
677:          Wrt_Str(16,18,item2[3],WHITE,LIGHTGRAY);
678:          Wrt_Str(56,18,"        .CFI",WHITE,LIGHTGRAY);
679:
680:          strcpy(im_name1,StrIn(56,12,BLUE,".img"));
681:          found = _dos_findfirst(im_name1,attribute,&fileinfo);
682:          if ((strlen(im_name1) == NULL) || (found != 0))
683:          {
684:             cursoronof(OFF);
685:             error_message(30,12,30,5," Image File not found ...");
686:             right = FALSE;
687:             return;
688:          }
689:
690:          strcpy(im_name2,StrIn(56,14,BLUE,".img"));
691:          found = _dos_findfirst(im_name2,attribute,&fileinfo);
692:          if ((strlen(im_name2) == NULL) || (found != 0))
693:          {
694:             cursoronof(OFF);
695:             error_message(30,14,30,5," Image File not found ...");
696:             right = FALSE;
697:             return;
698:          }
699:
700:          im_size = IntIn(56,16,BLUE);
701:
702:          M1 = (int) (sqrt((double) (im_size)));
703:          N1 = (int) im_size/M1;
704:
705:          if ((im_size < 0) || (M1 != N1) || (M1*N1 != im_size))
706:          {
707:             cursoronof(OFF);
708:             error_message(20,12,50,5,"Image Size is not accepted type..");
709:             right = FALSE;
710:             return;
711:          }
712:
713:          strcpy(im_name3,StrIn(56,18,BLUE,".cfi"));
714:
715:          comp_image(im_name1,im_name2,im_size,im_name3);
716:
717:          right = TRUE;
718:
719:        } while ( right != TRUE );
720:          cursoronof(OFF);
721:      }
722:      else  /* (menu_item == 5)   exit */
723:        {
724:          exitit = TRUE;
725:          cursoronof(OFF);
726:        }
727: return;
```

```
728: }
730:
731: void Hermite(void)
732: {
733: int Item_Num;
734: int i;
735: int num;
736: char ch;
737: int ImSize;
738: int HermDeg;
739: float GabWin;
740: int Freqm,Freqn;
741: char ImNameIn[20]  = "\0";
742: char ImNameOut[20] = "\0";
743: char CoeffName[20] = "\0";
744: char CoeffImg[20]  = "\0";
745: int count=0;
746: int Process=FALSE;
747: int errsum=0;
748: int errors=FALSE;
749:
750: int M1, N1;
751: unsigned int found, attribute=_A_NORMAL;
752: struct find_t fileinfo;
753:
754: char *item[8]; /*      Array containing Menu Entries   */
755:
756:     put_window(1,3,78,17,MAGENTA,3,"Hermite Function Decomposition",BLUE,WHITE);
757:
758:     item[0] = " Image Size to Process [64,256..] ";
759:     item[1] = " Input Image File Name (w/o EXT)  ";
760:     item[2] = " Output Image File Name (w/o EXT) ";
761:     item[3] = " Degree of Hermite Polynomial     ";
762:     item[4] = " Width Factor for Hermite Window  ";
763:     item[5] = " Frequency components considered  ";
764:     item[6] = " Save Coeffts. in file (w/o EXT)  ";
765:
766: do{
767:     errsum = 0;
768:     errors = FALSE;
769:     count++;
770:
771:     Wrt_Str(4,5,item[0],WHITE,LIGHTGRAY);
772:     Wrt_Str(44,5,"     ",WHITE,LIGHTGRAY);
773:
774:     Wrt_Str(4,7,item[1],WHITE,LIGHTGRAY);
775:     Wrt_Str(44,7,"       .IMG",WHITE,LIGHTGRAY);
776:
777:     Wrt_Str(4,9,item[2],WHITE,LIGHTGRAY);
778:     Wrt_Str(44,9,"       .IMG",WHITE,LIGHTGRAY);
779:
780:     Wrt_Str(4,11,item[3],WHITE,LIGHTGRAY);
781:     Wrt_Str(44,11,"    ",WHITE,LIGHTGRAY);
782:
783:     Wrt_Str(4,13,item[4],WHITE,LIGHTGRAY);
```

```
784:      Wrt_Str(44,13,"    ",WHITE,LIGHTGRAY);
786:
787:      Wrt_Str(4,15,item[5],WHITE,LIGHTGRAY);
788:      Wrt_Str(44,15,"   ",WHITE,LIGHTGRAY);
789:      Wrt_Str(48,15,"   ",WHITE,LIGHTGRAY);
790:
791:      Wrt_Str(4,17,item[6],WHITE,LIGHTGRAY);
792:      Wrt_Str(44,17,"       .CFT",WHITE,LIGHTGRAY);
793:
794:      Wrt_Str(2,20,"          Input the Analysed Image Side in Pixels (64, 256, 1024)          ",WHITE,RED);
795:      ImSize = IntIn(44,5,BLUE);
796:
797:      M1 = (int) (sqrt((double) (ImSize)));
798:      N1 = (int) ImSize/M1;
799:
800:      if ((ImSize < 0) || (M1 != N1) || (M1*N1 != ImSize))
801:      {
802:            cursoronof(OFF);
803:            error_message(20,12,50,5,"Image Size Cannot be Used in Decomposition...");
804:            errsum = 1;
805:            break;
806:      }
807:      Wrt_Str(2,20,"          Input the Analysed Image File Name (Omit extension .IMG)          ",WHITE,RED);
808:      strcpy(ImNameIn,StrIn(44,7,BLUE,".img"));
809:      found = _dos_findfirst(ImNameIn,attribute,&fileinfo);
810:      if ((strlen(ImNameIn) == NULL) || (found != 0))
811:      {
812:            cursoronof(OFF);
813:            error_message(30,12,30,5," Image File not found ...");
814:            errsum = 1;
815:            break;
816:      }
817:      Wrt_Str(2,20,"          Input the Processed Image File Name (Omit extension .IMG)         ",WHITE,RED);
818:      strcpy(ImNameOut,StrIn(44,9,BLUE,".img"));
819:      if (strlen(ImNameOut) == NULL)
820:      {
821:            /* Error Message comes here */
822:      }
823:      Wrt_Str(2,20,"             Input Hermite Polynomial Degree in Range - 0 to 5              ",WHITE,RED);
824:      HermDeg = IntIn(44,11,BLUE);
825:      if ((HermDeg < 0) || (HermDeg > 5))
826:      {
827:            cursoronof(OFF);
828:            error_message(25,12,50,5," Hermite Polynomial Degree is out of Bounds.. ");
829:            errsum = 1;
830:            break;
831:      }
832:      Wrt_Str(2,20," Input Width Factor - Width of the Gaussian Window at Half Amplitude (1-30) ",WHITE,RED);
833:      GabWin = (float) IntIn(44,13,BLUE);
834:      if ((GabWin < 1.) || (GabWin > 50.))
835:      {
836:            cursoronof(OFF);
837:            error_message(20,12,45,5," Gabor Window Width is out of Bounds...");
838:            errsum = 1;
839:            break;
```

```
840:     }
842:     Wrt_Str(2,20," Input 2 Frequency Components to be Considered (7 7), separated by ENTER    ",WHITE,RED);
843:     Freqm = IntIn(44,15,BLUE);
844:     Freqn = IntIn(48,15,BLUE);
845:     if (((Freqm < 0) || (Freqm > M1)) &&
846:         ((Freqn < 0) || (Freqn > M1)))
847:     {
848:           cursoronof(OFF);
849:           error_message(20,12,50,5," Threshold Frequencies are out of Bounds...");
850:           errsum = 1;
851:           break;
852:     }
853:     Wrt_Str(2,20,"       Save Calculated Coefficients into File for Image Reprocessing       ",WHITE,RED);
854:     strcpy(CoeffName,StrIn(44,17,BLUE,".cft"));
855:
856:     strext(CoeffName,"cfi",CoeffImg);
857:
858:     if (strlen(CoeffName) == NULL)
859:     {
860:           /* Error Message comes here */
861:     }
862:     cursoronof(OFF);
863:
864:     if (errsum == 0)
865:     {
866:         Wrt_Str(1,20,"                                                                                ",WHITE,LIGHTGRAY);
867:         Wrt_Str(18,20," The Input is Correct, Want to Proceed? (y/n)  ",WHITE,RED);
868:         ch = getch();
869:         errors = FALSE;
870:     }
871:     else
872:         {
873:         Wrt_Str(1,20,"                                                                                ",WHITE,LIGHTGRAY);
874:         Wrt_Str(20,20," Error(s) were found in Input, Try again  ",WHITE,RED);
875:         errors = TRUE;
876:         }
877:
878:     if (((ch == 'y') || (ch == 'Y')) && (errors == FALSE))
879:     {
880:           Process = TRUE;
881:           Wrt_Str(1,24,"                                                                                ",WHITE,LIGHTGRAY);
882:           Wrt_Str(18,20,"                                                  ",WHITE,LIGHTGRAY);
883:
884:           herm(ImSize,
885:                ImNameIn,
886:                ImNameOut,
887:                HermDeg,
888:                GabWin,
889:                Freqm,
890:                Freqn,
891:                M1,
892:                N1,
893:                CoeffName,
894:                CoeffImg);
895:     }
```

```
896:     else
898:       {
899:            Wrt_Str(1,24,"                                                                    ",WHITE,LIGHTGRAY);
900:       }
901:     } while((ch != 'y') && (ch != 'Y'));
902: }
903:
904: void put_window(x1,y1,w,h,attr,type,title,titlebk,titlecol)
905: int x1,y1,w,h,attr,type;
906: int titlebk,titlecol;
907: char *title;
908: {
909: short *winptr;
910:
911:   winptr = create_text_window(w,h,attr);
912:   box_text_window(winptr, type, attr);
913:   restore_text_window(x1, y1, winptr);
914:   locate(x1+(w/2-strlen(title)/2),y1);
915:   _setbkcolor(titlebk);
916:   _settextcolor(titlecol);
917:   write_string(title);
918:   free_text_window(winptr);
919: }
920:
921: short *save_scr(x1,y1,w,h)
922: int x1,y1,w,h;
923: {
924: short *original;
925:
926:     original = create_text_window(w,h,get_active_attribute());
927:     save_text_window(x1,y1,x1+w-1,y1+h-1,original);
928:
929:  return original;
930: }
931:
932: void rest_scr(x1,y1,original)
933: int x1,y1;
934: short *original;
935: {
936:     restore_text_window(x1,y1,original);
937: }
938:
939: void error_message(x1,y1,w,h,message)
940: int x1,y1,w,h;
941: char *message;
942: {
943: int type=5;
944: short *s_screen;
945: int k;
946:
947:    s_screen = save_scr(x1,y1,w,h);
948:    put_window(x1,y1,w,h,RED,type," Error Message ",LIGHTGRAY,BLUE);
949:    Wrt_Str(x1+2,y1+2,message,WHITE,LIGHTGRAY);
950:    while ((k = pauseb()) != ENTER);
951:    rest_scr(x1,y1,s_screen);
```

```
952: }
954:
955: char *int2ch(sum)
956: int sum;
957: {
958: int i;
959: int dig[7];
960: char *str="\0";
961:
962:  dig[0] = (int) sum/100000;
963:  dig[1] = (int) (sum - dig[0]*100000)/10000;
964:  dig[2] = (int) (sum - dig[0]*100000 - dig[1]*10000)/1000;
965:  dig[3] = (int) (sum - dig[0]*100000 - dig[1]*10000 - dig[2]*1000)/100;
966:  dig[4] = (int) (sum - dig[0]*100000 - dig[1]*10000 - dig[2]*1000 - dig[3]*100)/10;
967:  dig[5] = (int) (sum - dig[0]*100000 - dig[1]*10000 - dig[2]*1000 - dig[3]*100 - dig[4]*10);
968:
969:  for (i=0;i<6;i++)
970:     str[i] = (char) (dig[i] + 48);
971:
972: str[7] = '\0';
973:
974: return str;
975: }
976:
977: void strext(st1,st2,st3)
978: char st1[13];
979: char st2[3];
980: char st3[13];
981: {
982: int len;
983: int i,j=0;
984: int dot=FALSE;
985:
986:     len = strlen(st1);
987:
988:         strcpy(st3,st1);
989:
990:         for (i=0;i<len-1;i++)
991:         {
992:             if (st3[i] == '.')
993:                dot = TRUE;
994:
995:             if (dot)
996:                  st3[i+1] = st2[j++];
997:         }
998: }
```

```
 1: /*
 2:  ***************************************************************************
 3:  * RCS identification
 4:  ***************************************************************************
 5:  * $Author: thibaud $
 6:  * $Date: 1991/11/15 15:34:57 $
 7:  * $Locker: thibaud $
 8:  * $Revision: 1.3 $
 9:  * $Source: /isis/u2/thibaud/usr/gertner/2D/zak/RCS/zakgab.c,v $
10:  * $State: Exp $
11:  * $Log: zakgab.c,v $
12:  * Revision 1.3  1991/11/15  15:34:57  thibaud
13:  *
14:  * Revised by       : Alex Firdman, UDRI
15:  * Date of Revision : Oct. 11, 1993
16:  * Compiler Used    : NDP-C, Ver. 4.2.1
17:  *
18:  * move cabs to complex_abs (for the Vistra)
19:  *
20:  * Revision 1.2  1991/11/05  22:47:00  thibaud
21:  * cb_filter and RCS header
22:  *
23:  ***************************************************************************
24:  * End of RCS identification
25:  ***************************************************************************
26:  */
27:
28: #include <stdio.h>
29: #include <stdlib.h>
30: #include <os.h>
31: #include <math.h>
32: #include <grex.h>
33:
34: /************** Constants Definition ***************/
35:
36: #define BLACK         0
37: #define BLUE          1
38: #define GREEN         2
39: #define CYAN          3
40: #define RED           4
41: #define MAGENTA       5
42: #define BROWN         6
43: #define LIGHTGRAY     7    /* Color Definition */
44: #define DARKGRAY      8
45: #define LIGHTBLUE     9
46: #define LIGHTGREEN   10
47: #define LIGHTCYAN    11
48: #define LIGHTRED     12
49: #define LIGHTMAGENTA 13
50: #define YELLOW       14
51: #define WHITE        15
52:
53: #define FALSE           1
54: #define TRUE            0
55:
```

```
 56: void Wrt_Str(int,int,char*,int,int);
 58: char *int2ch(int);
 59:
 60: #include "complex.h"
 61:
 62: #define PIXELS          256
 63: #define M               16
 64: #define N               PIXELS/M
 65: #define pi              4.0*atan(1.0)
 66: #define SWAP(a,b)       tempr=(a); (a)=(b); (b)=tempr
 67:
 68: struct  SMALL_REAL {
 69:                     float   sr1[2*PIXELS+1];
 70:                    };
 71: typedef struct SMALL_REAL small_real;
 72:
 73: struct  BIG_REAL {
 74:                     float   br1[2*PIXELS*PIXELS+1];
 75:                    };
 76: typedef struct BIG_REAL big_real;
 77:
 78: struct  BIG_COMPLEX {
 79:                     complex bc1[PIXELS*PIXELS];
 80:                    };
 81: typedef struct BIG_COMPLEX big_complex;
 82:
 83: struct  ARRAY_2_R {
 84:                     float   x2[PIXELS][PIXELS];
 85:                    };
 86: typedef struct ARRAY_2_R array_2_r;
 87:
 88: struct  ARRAY_4_R {
 89:                     float   x4[M][N][M][N];
 90:                    };
 91: typedef struct ARRAY_4_R array_4_r;
 92:
 93: struct  ARRAY_4_C {
 94:                     complex c4[M][N][M][N];
 95:                    };
 96: typedef struct ARRAY_4_C array_4_c;
 97:
 98: typedef unsigned char byte;
 99:
100: void put_window(int,int,int,int,int,int,char*,int,int);
101:
102: int herm(side, imageI, imageR, deg, width, THRm, THRn, M1, N1, imageC, imageS)
103: int side;
104: char imageI[13];
105: char imageR[13];
106: int deg;
107: float width;
108: int THRm, THRn;
109: int M1,N1;
110: char imageC[13];
111: char imageS[13];
```

```
112: {
114:
115:     FILE            *DIS, *t, *z_t, *ftempout, *spectrOut;
116:     char            ch,
117:                     filename_in[80],
118:                     filename_out[80];
119:
120:     int             val2, val3;
121:     int             i, j, m, n, r, s, arg, ro, sig;
122:     int             count = 0;
123:     int             dim[5];
124:     float           val1, image_size, x1, x2,prn;
125:
126:     array_2_r       *f2, *g2;
127:     array_2_r           *f_R;
128:
129:     array_4_r       *f4, *g4;
130:
131:     big_real        *fdata;
132:     complex         z, z1, z2;
133:     array_4_c       *zf4, *zg4, *zf_R;
134:     byte            ftemp[M*N];
135:     char                    *mess_do[3];
136:     char                    *mess_info[30];
137:     char                    *mess_error[15];
138:     char                    number1[5];
139:     char                    number2[5];
140:     int                     q=0;
141:     char                    *numbert;
142:     char            *none="                                                                 ";
143:     int                     no_file_name;
144:
145:     int                             gab[M][N][M][N];
146:     int                     a,b;
147:     float                   intensity[PIXELS][PIXELS];
148:     byte                    spectr[M*N];
149:
150:
151:     mess_do[0] = "Hit <return> to continue...";
152:
153:     mess_error[0]  = "Cannot Open Image File...";
154:     mess_error[1]  = "Cannot Open File to Store Processed Image...";
155:     mess_error[2]  = "Cannot Allocate Memory for f2 Array...";
156:     mess_error[3]  = "Cannot Allocate Memory for g2 Array...";
157:     mess_error[4]  = "Cannot Allocate Memory for f4 Array...";
158:     mess_error[5]  = "Cannot Allocate Memory for g4 Array...";
159:     mess_error[6]  = "Cannot Allocate Memory for zf4 Array...";
160:     mess_error[7]  = "Cannot Allocate Memory for zg4 Array...";
161:     mess_error[8]  = "Cannot Allocate fdata Array...";
162:     mess_error[9]  = "Cannot Allocate zf_R Array...";
163:     mess_error[10] = "Cannot Allocate f_R Array...";
164:
165:     mess_info[0]  = "Starting process...";
166:     mess_info[1]  = "Opening files...";
167:     mess_info[2]  = "Allocating memory for real 2D arrays...";
```

```
168:     mess_info[3]  = "Allocating memory for real 4D arrays...";
170:     mess_info[4]  = "Reading original image...";
171:     mess_info[5]  = "Changing 2D arrays to 4D arrays...";
172:     mess_info[6]  = "Freeing memory used by real 2D array for the image...";
173:     mess_info[7]  = "Allocating memory for complex 4D array for the image..";
174:     mess_info[8]  = "Doing the Zak transforms on the image...";
175:     mess_info[9]  = "Freeing memory used by real 4D array of the image...";
176:     mess_info[10] = "Doing the Zak transforms on the arrays...";
177:     mess_info[11] = "Allocating Memory for a Big Real Array...";
178:     mess_info[12] = "Computing zf4/zg4 for Inverse FFT...";
179:     mess_info[13] = "Freeing Memory used by Zak transform of the image...";
180:     mess_info[14] = "Computing Coefficients...";
181:     mess_info[15] = "Writing Coefficients to the file -> ";
182:     mess_info[16] = "Starting to Discard Coefficients...";
183:     mess_info[17] = "Number of Coefficients Kept:  ";
184:     mess_info[18] = "Number of Coefficients Discarded:  ";
185:     mess_info[19] = "Starting to Recover the Image...";
186:     mess_info[20] = "Allocating Memory for a New Complex 4D Arrays...";
187:     mess_info[21] = "Recovering result of the FFT and multiplying it...";
188:     mess_info[22] = "Freeing memory used by the big real array...";
189:     mess_info[23] = "Allocating memory for processed image...";
190:     mess_info[24] = "Doing the inverse Zak transform on the processed image...";
191:     mess_info[25] = "Freeing memory used by Zak of the processed image...";
192:     mess_info[26] = "End Image synthesis, Writing processed image to file...";
193:     mess_info[27] = "Image processed and stored in file, Freeing memory...";
194:     mess_info[28] = "All Done! ... Hit any Key to Exit to Main Menu ...";
195:
196:     put_window(12,2,65,20,0x1f,3," Running Hermite Image Decomposition ", RED, YELLOW);
197:
198:     strcpy( filename_in, imageI);
199:
200:     strcpy( filename_out, imageR);
201:
202:     Wrt_Str(17,3,mess_info[0],WHITE, BLUE);
203:
204:     dim[1] = dim[2] = M1;
205:     dim[3] = dim[4] = N1;
206:
207:     image_size = (float)(side) * (float)(side);
208:
209:     /* original image */
210:
211:     Wrt_Str(17,4,mess_info[1],WHITE,BLUE);
212:
213:     _pmode = 0x8000;
214:     t = fopen( filename_in, "r" );
215:     if ( t == NULL ) {
216:         Wrt_Str(17,5,mess_error[0],WHITE,BLUE);
217:         Wrt_Str(17,6,mess_do[0],WHITE,BLUE);
218:         pauseb();
219:         exit(1);
220:     }
221:
222:     /* processed image */
223:     z_t = fopen( filename_out, "w" );
```

```
224:     if ( z_t == NULL ) {
226:         Wrt_Str(17,5,mess_error[1],WHITE,RED);
227:         Wrt_Str(17,6,mess_do[0],WHITE,RED);
228:         pauseb();
229:         exit(1);
230:     }
231:
232:     Wrt_Str(17,5,mess_info[2],WHITE,BLUE);
233:
234:     f2 = (array_2_r *)( malloc( sizeof( array_2_r ) ) );
235:     if ( f2 == NULL ) {
236:         Wrt_Str(17,6,mess_error[2],WHITE,BLUE);
237:         Wrt_Str(17,7,mess_do[0],WHITE,BLUE);
238:         pauseb();
239:         exit(1);
240:     }
241:
242:     g2 = (array_2_r *)( malloc( sizeof( array_2_r ) ) );
243:     if ( g2 == NULL ) {
244:         Wrt_Str(17,6,mess_error[3],WHITE,BLUE);
245:         Wrt_Str(17,7,mess_do[0],WHITE,BLUE);
246:         free( f2 );
247:         pauseb();
248:         exit(1);
249:     }
250:
251:     Wrt_Str(17,6,mess_info[3],WHITE,BLUE);
252:
253:     f4 = (array_4_r *)( malloc( sizeof( array_4_r ) ) );
254:     if ( f4 == NULL ) {
255:         Wrt_Str(17,7,mess_error[4],WHITE,BLUE);
256:         Wrt_Str(17,8,mess_do[0],WHITE,BLUE);
257:         free( f2 );
258:         free( g2 );
259:         pauseb();
260:         exit(1);
261:     }
262:
263:     g4 = (array_4_r *)( malloc( sizeof( array_4_r ) ) );
264:     if ( g4 == NULL ) {
265:         Wrt_Str(17,7,mess_error[5],WHITE,BLUE);
266:         Wrt_Str(17,8,mess_do[0], WHITE,BLUE);
267:         free(f2);
268:         free(g2);
269:         free(f4);
270:         pauseb();
271:         exit(1);
272:     }
273:
274:     Wrt_Str(17,7,mess_info[4],WHITE,BLUE);
275:
276:     for (i = 0; i < side; i++ )
277:     {
278:         fread(ftemp, 1, side, t);
279:
```

```
280:         for (j = 0; j < side; j++ )
282:         {
283:             f2->x2[i][j] = (float) ftemp[j];
284:         }
285:     }
286:     fclose( t );
287:
288:     get_values( side, width, deg, g2->x2 );
289:
290:     Wrt_Str(17,8,mess_info[5],WHITE,BLUE);
291:
292:     /*** Converting 2-D array to 4-D array ***/
293:     twotofour( f2->x2, f4->x4, M1, N1 );  /* containing image */
294:     twotofour( g2->x2, g4->x4, M1, N1 );  /* containing coefficients */
295:
296:     free( f2 );
297:     free( g2 );
298:
299:     Wrt_Str(17,9,mess_info[6],WHITE,BLUE);
300:
301:
302:     Wrt_Str(17,10,mess_info[7],WHITE,BLUE);
303:
304:     zf4 = (array_4_c *)( malloc( sizeof( array_4_c ) ) );
305:     if ( zf4 == NULL ) {
306:         Wrt_Str(17,11,mess_error[6],WHITE,BLUE);
307:         free( f2 );
308:         free( f4 );
309:         free( g4 );
310:         Wrt_Str(17,12,mess_do[0],WHITE,BLUE);
311:         pauseb();
312:         exit(1);
313:     }
314:
315:     Wrt_Str(17,11,mess_info[8],WHITE,BLUE);
316:
317:     ZAK( f4->x4, zf4->c4, M1, N1 );
318:
319:     Wrt_Str(17,12,mess_info[9],WHITE,BLUE);
320:
321:     free( f4 );
322:
323:     zg4 = (array_4_c *)( malloc( sizeof( array_4_c ) ) );
324:     if ( zg4 == NULL ) {
325:         Wrt_Str(17,13,mess_error[7],WHITE,BLUE);
326:         free( f2 );
327:         free( g4 );
328:         free( zf4 );
329:         Wrt_Str(17,14,mess_do[0],WHITE,BLUE);
330:         pauseb();
331:         exit(1);
332:     }
333:
334:     Wrt_Str(17,13,mess_info[10],WHITE,BLUE);
335:
```

```
336:     ZAK( g4->x4, zg4->c4, M1, N1 );
338:
339:     free( g4 );
340:
341:     Wrt_Str(17,14,mess_info[11],WHITE,BLUE);
342:
343:     fdata = (big_real *)( malloc( sizeof( big_real ) ) );
344:     if ( fdata == NULL ) {
345:         Wrt_Str(17,15,mess_error[8],WHITE,BLUE);
346:         free( f2 );
347:         free( zf4 );
348:         free( zg4 );
349:         Wrt_Str(17,16,mess_do[0],WHITE,BLUE);
350:         pauseb();
351:         exit(1);
352:     }
353:
354:     Wrt_Str(17,15,mess_info[12],WHITE,BLUE);
355:
356:
357:     for (m = 0; m < M1; m++ )
358:     {
359:         for (n = 0; n < M1; n++ )
360:         {
361:             for (r = 0; r < N1; r++ )
362:             {
363:                 for (s = 0; s < N1; s++ )
364:                 {
365:                     z1 = zf4->c4[m][n][r][s];
366:                     z2 = zg4->c4[m][n][r][s];
367:                     z = cdiv( z1, z2 );
368:                     x1 = z.x / image_size;
369:                     x2 = z.y / image_size;
370:                     arg = m*M1*N1*N1 + n*N1*N1 + r*N1 + s;
371:                     fdata->br1[2*arg+1] = x1;
372:                     fdata->br1[2*arg+2] = x2;
373:                 }
374:             }
375:         }
376:     }
377:
378:     Wrt_Str(17,16,mess_info[13],WHITE,BLUE);
379:
380:     free( zf4 );
381:
382:     Wrt_Str(17,17,mess_info[14],WHITE,BLUE);
383:
384:     fourn( fdata->br1, dim, 4, -1 );          /* Call 4-dim IFFT routine */
385:
386: /******** Saving Coefficients in to File **********/
387: /******** Saving Coefficients in to File **********/
388:  _pmode = 0x4000;
389: ftempout = fopen(imageC,"wt");
390:
391: if (ftempout != NULL)
```

```
392: {
394:      Wrt_Str(17,18,mess_info[15],WHITE,BLUE);
395:      Wrt_Str(55,18,imageC,WHITE,RED);
396: }
397:
398: /*************************/
399: /* ro is   a variable  representing the row coordinate
400: of the coefficient array and it goes from 0 to 15 */
401: for(ro=0;ro<N1; ro++)
402: {
403:  fprintf(ftempout,"\n*** coeff row coordinate ro # %d", ro);
404:  for(sig=0;sig<2*N1;sig+=2)
405:  {
406:   fprintf(ftempout,"\n*** Vertical pos sig  # %d\n", sig/2);
407:   for(m=0;m<M1;m++)  /* m is horizontal spatial frequency******/
408:   {
409:    fprintf(ftempout,"\n");
410:    for(n=0;n<M1;n++) /* n is vertical spatial frequency ********/
411:    {
412:
413:      arg=2*m*M1*N1*N1 + 2*n*N1*N1 + 2*N1*ro + sig; ++++++++++++
414:     /* prn is the absolute value of the coefficients */
415:      prn = sqrt( fdata->br1[arg+1]* fdata->br1[arg+1] +
416:                    fdata->br1[arg+2]*fdata->br1[arg+2]);
417:
418:      if (strlen(imageC) >= 5)
419:      {
420:         no_file_name = FALSE;
421:         fprintf(ftempout,"%8.4f", prn );
422:      }
423:
424:    }
425:   }
426:  }
427: }
428: fclose(ftempout);
429: /*************************/
430: _pmode = 0x8000;
431:     spectrOut = fopen(imageS , "wb" );
432:     if ( spectrOut == NULL )
433:     {
434:         Wrt_Str(17,5," Failed to Create Coefficient Image..",WHITE,RED);
435:         Wrt_Str(17,6,mess_do[0],WHITE,RED);
436:         pauseb();
437:         exit(1);
438:     }
439:
440:     if (strlen(imageC) < 5)
441:     {
442:         no_file_name  = TRUE;
443:         mess_info[15] = "Coefficients are not saved, File Name was not entered..";
444:         Wrt_Str(17,18,mess_info[15],WHITE,BLUE);
445:     }
446:
447: arg=0;
```

```
448: m=0;
450: n=0;
451: ro=0;
452: sig=0;
453:
454:
455:
456: /* THIS THE END OF write to file !!!!!! *****************/
457:
458:     Wrt_Str(17,19,mess_info[16],WHITE,BLUE);
459:
460:     for ( m = 0; m < M1; m++ ) {
461:         for ( n = 0; n < M1; n++ ) {
462:                 {
463:                     for ( ro = 0; ro < N1; ro++ ) {
464:                         for ( sig = 0; sig < 2*N1; sig += 2) {
465: /* sig is vertical column, ro is horizontal row */
466: /*
467:   if ( !(
468:       (
469:       ((m ==1) && (n ==0))||
470:       ((m ==2) && (n ==0))||
471:       ((m ==3) && (n ==0))||
472:       ((m ==4) && (n ==0))||
473:       ((m ==5) && (n ==0))||
474:       ((m ==6) && (n ==0))||
475:       ((m ==7) && (n ==0))||
476:       ((m ==8) && (n ==0))||
477:       ((m ==0) && (n ==0))   )
478:       &&
479:       (
480:       (  (ro==0) && (sig==0))   ||
481:       (  (ro==0) && (sig==30))  ||
482:       (  (ro==0) && (sig==28))  ||
483:       (  (ro==0) && (sig==26))  ||
484:       (  (ro==0) && (sig==24))  ||
485:       (  (ro==1) && (sig==0))   ||
486:       (  (ro==1) && (sig==30))  ||
487:       (  (ro==1) && (sig==28))  ||
488:       (  (ro==1) && (sig==26))  ||
489:       (  (ro==1) && (sig==24))  ||
490:       (  (ro==2) && (sig==0))   ||
491:       (  (ro==2) && (sig==30))  ||
492:       (  (ro==2) && (sig==28))  ||
493:       (  (ro==2) && (sig==26))  ||
494:       (  (ro==2) && (sig==24))  ||
495:       (  (ro==3) && (sig==0))   ||
496:       (  (ro==3) && (sig==30))  ||
497:       (  (ro==3) && (sig==28))  ||
498:       (  (ro==3) && (sig==26))  ||
499:       (  (ro==3) && (sig==24))  ||
500:       (  (ro==15) && (sig==0))  ||
501:       (  (ro==15) && (sig==30)) ||
502:       (  (ro==15) && (sig==28)) ||
503:       (  (ro==15) && (sig==26)) ||
```

```
504:       (  (ro==15) && (sig==24))
506:       )
507:                 ){
508:            arg =  2*m*M1*N1*N1 + 2*n*N1*N1 + 2*N1*ro + sig;
509:            fdata->br1[arg+1] = fdata->br1[arg+2] = 0.;
510:                       count++;
511:                    }
512: */
513:                    }
514:                 }
515:              }
516:           }
517:        }
518:
519:        Wrt_Str(17,20,mess_info[17],WHITE,BLUE);
520:        Wrt_Str(51,20,int2ch(M1*M1*N1*N1 - count),WHITE,RED);
521: /* here I insert the write to file  code ************/
522:        /*** Scroll Messages Up ***/
523:        for (q=3;q<21;q++)
524:           Wrt_Str(17,q,none,BLUE,BLUE);
525:
526:        for (q=1;q<15;q++)
527:           Wrt_Str(17,q+2,mess_info[q],WHITE,BLUE);
528:
529:        if (no_file_name == FALSE)
530:        {
531:            Wrt_Str(17,17,none,BLUE,BLUE);
532:            Wrt_Str(17,17,mess_info[15],WHITE,BLUE);
533:            Wrt_Str(55,17,imageC,WHITE,RED);
534:        } else
535:            {
536:                Wrt_Str(17,17,none,BLUE,BLUE);
537:                Wrt_Str(17,17,mess_info[15],WHITE,BLUE);
538:            }
539:
540:        Wrt_Str(17,18,none,BLUE,BLUE);
541:        Wrt_Str(17,18,mess_info[16],WHITE,BLUE);
542:
543:        Wrt_Str(17,19,none,BLUE,BLUE);
544:        Wrt_Str(17,19,mess_info[17],WHITE,BLUE);
545:        Wrt_Str(51,19,int2ch(M1*M1*N1*N1 - count),WHITE,RED);
546:
547:        Wrt_Str(17,20,none,BLUE,BLUE);
548:        Wrt_Str(17,20,mess_info[18],WHITE,BLUE);
549:        Wrt_Str(51,20,int2ch(count),WHITE,RED);
550:        /************************************/
551:
552:        /* start image synthesis*/
553:
554:        fourn(fdata->br1, dim, 4, 1);
555:
556:        /*** Scroll Messages Up ***/
557:        for (q=3;q<21;q++)
558:           Wrt_Str(17,q,none,BLUE,BLUE);
559:
```

```
560:     for (q=2;q<15;q++)
562:        Wrt_Str(17,q+1,mess_info[q],WHITE,BLUE);
563:
564:     if (no_file_name == FALSE)
565:     {
566:         Wrt_Str(17,16,none,BLUE,BLUE);
567:         Wrt_Str(17,16,mess_info[15],WHITE,BLUE);
568:         Wrt_Str(55,16,imageC,WHITE,RED);
569:     } else
570:         {
571:             Wrt_Str(17,16,none,BLUE,BLUE);
572:             Wrt_Str(17,16,mess_info[15],WHITE,BLUE);
573:         }
574:
575:     Wrt_Str(17,17,none,BLUE,BLUE);
576:     Wrt_Str(17,17,mess_info[16],WHITE,BLUE);
577:
578:     Wrt_Str(17,18,none,BLUE,BLUE);
579:     Wrt_Str(17,18,mess_info[17],WHITE,BLUE);
580:     Wrt_Str(51,18,int2ch(M1*M1*N1*N1 - count),WHITE,RED);
581:
582:     Wrt_Str(17,19,none,BLUE,BLUE);
583:     Wrt_Str(17,19,mess_info[18],WHITE,BLUE);
584:     Wrt_Str(51,19,int2ch(count),WHITE,RED);
585:
586:     Wrt_Str(17,20,none,BLUE,BLUE);
587:     Wrt_Str(17,20,mess_info[19],WHITE,BLUE);
588:     /************************************/
589:
590:     zf_R = (array_4_c *)( malloc( sizeof( array_4_c ) ) );
591:     if ( zf_R == NULL ) {
592:
593:         Wrt_Str(17,19,mess_error[9],WHITE,BLUE);
594:
595:         free( f2 );
596:         free( fdata );
597:         free( zg4 );
598:
599:         Wrt_Str(17,20,mess_do[0],WHITE,BLUE);
600:         pauseb();
601:         exit(1);
602:     }
603:
604:     /*** Scroll Messages Up ***/
605:     for (q=3;q<21;q++)
606:        Wrt_Str(17,q,none,BLUE,BLUE);
607:
608:     for (q=3;q<15;q++)
609:        Wrt_Str(17,q,mess_info[q],WHITE,BLUE);
610:
611:     if (no_file_name == FALSE)
612:     {
613:         Wrt_Str(17,15,none,BLUE,BLUE);
614:         Wrt_Str(17,15,mess_info[15],WHITE,BLUE);
615:         Wrt_Str(55,15,imageC,WHITE,RED);
```

```
616:     }  else
618:         {
619:             Wrt_Str(17,15,none,BLUE,BLUE);
620:             Wrt_Str(17,15,mess_info[15],WHITE,BLUE);
621:         }
622:
623:     Wrt_Str(17,16,none,BLUE,BLUE);
624:     Wrt_Str(17,16,mess_info[16],WHITE,BLUE);
625:
626:     Wrt_Str(17,17,none,BLUE,BLUE);
627:     Wrt_Str(17,17,mess_info[17],WHITE,BLUE);
628:     Wrt_Str(51,17,int2ch(M1*M1*N1*N1 - count),WHITE,RED);
629:
630:     Wrt_Str(17,18,none,BLUE,BLUE);
631:     Wrt_Str(17,18,mess_info[18],WHITE,BLUE);
632:     Wrt_Str(51,18,int2ch(count),WHITE,RED);
633:
634:     Wrt_Str(17,19,none,BLUE,BLUE);
635:     Wrt_Str(17,19,mess_info[19],WHITE,BLUE);
636:
637:     Wrt_Str(17,20,none,BLUE,BLUE);
638:     Wrt_Str(17,20,mess_info[20],WHITE,BLUE);
639:     /*************************************/
640:
641:     for (m = 0; m < M1; m++)
642:     {
643:         for (n = 0; n < M1; n++)
644:         {
645:             for (ro = 0; ro < N1; ro++)
646:             {
647:                 for (sig = 0; sig < 2*N1; sig += 2)
648:                 {
649:                     arg = 2*m*M1*N1*N1 + 2*n*N1*N1 + 2*N1*ro + sig + 1;
650:                     x1 = fdata->br1[arg];
651:                     x2 = fdata->br1[arg+1];
652:                     z1 = cmplx( x1, x2 );
653:                     r = ro;
654:                     s = sig / 2;
655:                     z2 = zg4->c4[m][n][r][s];
656:                     z = cmul( z1, z2 );
657:                     zf_R->c4[m][n][r][s] = z;
658:                 }
659:             }
660:         }
661:     }
662:
663:     /*** Scroll Messages Up ***/
664:     for (q=3;q<21;q++)
665:        Wrt_Str(17,q,none,BLUE,BLUE);
666:
667:     for (q=4;q<15;q++)
668:        Wrt_Str(17,q-1,mess_info[q],WHITE,BLUE);
669:
670:     if (no_file_name == FALSE)
671:     {
```

```
672:          Wrt_Str(17,14,none,BLUE,BLUE);
674:          Wrt_Str(17,14,mess_info[15],WHITE,BLUE);
675:          Wrt_Str(55,14,imageC,WHITE,RED);
676:      }  else
677:          {
678:              Wrt_Str(17,14,none,BLUE,BLUE);
679:              Wrt_Str(17,14,mess_info[15],WHITE,BLUE);
680:          }
681:
682:      Wrt_Str(17,15,none,BLUE,BLUE);
683:      Wrt_Str(17,15,mess_info[16],WHITE,BLUE);
684:
685:      Wrt_Str(17,16,none,BLUE,BLUE);
686:      Wrt_Str(17,16,mess_info[17],WHITE,BLUE);
687:      Wrt_Str(51,16,int2ch(M1*M1*N1*N1 - count),WHITE,RED);
688:
689:      Wrt_Str(17,17,none,BLUE,BLUE);
690:      Wrt_Str(17,17,mess_info[18],WHITE,BLUE);
691:      Wrt_Str(51,17,int2ch(count),WHITE,RED);
692:
693:      Wrt_Str(17,18,none,BLUE,BLUE);
694:      Wrt_Str(17,18,mess_info[19],WHITE,BLUE);
695:
696:      Wrt_Str(17,19,none,BLUE,BLUE);
697:      Wrt_Str(17,19,mess_info[20],WHITE,BLUE);
698:
699:      Wrt_Str(17,20,none,BLUE,BLUE);
700:      Wrt_Str(17,20,mess_info[21],WHITE,BLUE);
701:      /**********************************/
702:
703:
704:      /*** Scroll Messages Up ***/
705:      for (q=3;q<21;q++)
706:         Wrt_Str(17,q,none,BLUE,BLUE);
707:
708:      for (q=5;q<15;q++)
709:         Wrt_Str(17,q-2,mess_info[q],WHITE,BLUE);
710:
711:      if (no_file_name == FALSE)
712:      {
713:          Wrt_Str(17,13,none,BLUE,BLUE);
714:          Wrt_Str(17,13,mess_info[15],WHITE,BLUE);
715:          Wrt_Str(55,13,imageC,WHITE,RED);
716:      }  else
717:          {
718:              Wrt_Str(17,13,none,BLUE,BLUE);
719:              Wrt_Str(17,13,mess_info[15],WHITE,BLUE);
720:          }
721:
722:      Wrt_Str(17,14,none,BLUE,BLUE);
723:      Wrt_Str(17,14,mess_info[16],WHITE,BLUE);
724:
725:      Wrt_Str(17,15,none,BLUE,BLUE);
726:      Wrt_Str(17,15,mess_info[17],WHITE,BLUE);
727:      Wrt_Str(51,15,int2ch(M1*M1*N1*N1 - count),WHITE,RED);
```

```
728:
730:      Wrt_Str(17,16,none,BLUE,BLUE);
731:      Wrt_Str(17,16,mess_info[18],WHITE,BLUE);
732:      Wrt_Str(51,16,int2ch(count),WHITE,RED);
733:
734:      Wrt_Str(17,17,none,BLUE,BLUE);
735:      Wrt_Str(17,17,mess_info[19],WHITE,BLUE);
736:
737:      Wrt_Str(17,18,none,BLUE,BLUE);
738:      Wrt_Str(17,18,mess_info[20],WHITE,BLUE);
739:
740:      Wrt_Str(17,19,none,BLUE,BLUE);
741:      Wrt_Str(17,19,mess_info[21],WHITE,BLUE);
742:
743:      Wrt_Str(17,20,none,BLUE,BLUE);
744:      Wrt_Str(17,20,mess_info[22],WHITE,BLUE);
745:      /************************************/
746:
747:      free( fdata );
748:      free( zg4 );
749:
750:      /*** Scroll Messages Up ***/
751:      for (q=3;q<21;q++)
752:         Wrt_Str(17,q,none,BLUE,BLUE);
753:
754:      for (q=6;q<15;q++)
755:         Wrt_Str(17,q-3,mess_info[q],WHITE,BLUE);
756:
757:      if (no_file_name == FALSE)
758:      {
759:          Wrt_Str(17,12,none,BLUE,BLUE);
760:          Wrt_Str(17,12,mess_info[15],WHITE,BLUE);
761:          Wrt_Str(55,12,imageC,WHITE,RED);
762:      }  else
763:          {
764:              Wrt_Str(17,12,none,BLUE,BLUE);
765:              Wrt_Str(17,12,mess_info[15],WHITE,BLUE);
766:          }
767:
768:      Wrt_Str(17,13,none,BLUE,BLUE);
769:      Wrt_Str(17,13,mess_info[16],WHITE,BLUE);
770:
771:      Wrt_Str(17,14,none,BLUE,BLUE);
772:      Wrt_Str(17,14,mess_info[17],WHITE,BLUE);
773:      Wrt_Str(51,14,int2ch(M1*M1*N1*N1 - count),WHITE,RED);
774:
775:      Wrt_Str(17,15,none,BLUE,BLUE);
776:      Wrt_Str(17,15,mess_info[18],WHITE,BLUE);
777:      Wrt_Str(51,15,int2ch(count),WHITE,RED);
778:
779:      Wrt_Str(17,16,none,BLUE,BLUE);
780:      Wrt_Str(17,16,mess_info[19],WHITE,BLUE);
781:
782:      Wrt_Str(17,17,none,BLUE,BLUE);
783:      Wrt_Str(17,17,mess_info[20],WHITE,BLUE);
```

```
784:
786:      Wrt_Str(17,18,none,BLUE,BLUE);
787:      Wrt_Str(17,18,mess_info[21],WHITE,BLUE);
788:
789:      Wrt_Str(17,19,none,BLUE,BLUE);
790:      Wrt_Str(17,19,mess_info[22],WHITE,BLUE);
791:
792:      Wrt_Str(17,20,none,BLUE,BLUE);
793:      Wrt_Str(17,20,mess_info[23],WHITE,BLUE);
794:      /*************************************/
795:
796:      f_R = (array_2_r *)( malloc( sizeof( array_2_r ) ) );
797:      if ( f_R == NULL ) {
798:          Wrt_Str(17,19,mess_error[10],WHITE,BLUE);
799:          free( f2 );
800:          free( zf_R );
801:          Wrt_Str(17,20,mess_do[0],WHITE,BLUE);
802:          pauseb();
803:          exit(1);
804:      }
805:
806:      /*** Scroll Messages Up ***/
807:      for (q=3;q<21;q++)
808:         Wrt_Str(17,q,none,BLUE,BLUE);
809:
810:      for (q=7;q<15;q++)
811:         Wrt_Str(17,q-4,mess_info[q],WHITE,BLUE);
812:
813:      if (no_file_name == FALSE)
814:      {
815:          Wrt_Str(17,11,none,BLUE,BLUE);
816:          Wrt_Str(17,11,mess_info[15],WHITE,BLUE);
817:          Wrt_Str(55,11,imageC,WHITE,RED);
818:      }  else
819:          {
820:              Wrt_Str(17,11,none,BLUE,BLUE);
821:              Wrt_Str(17,11,mess_info[15],WHITE,BLUE);
822:          }
823:
824:      Wrt_Str(17,12,none,BLUE,BLUE);
825:      Wrt_Str(17,12,mess_info[16],WHITE,BLUE);
826:
827:      Wrt_Str(17,13,none,BLUE,BLUE);
828:      Wrt_Str(17,13,mess_info[17],WHITE,BLUE);
829:      Wrt_Str(51,13,int2ch(M1*M1*N1*N1 - count),WHITE,RED);
830:
831:      Wrt_Str(17,14,none,BLUE,BLUE);
832:      Wrt_Str(17,14,mess_info[18],WHITE,BLUE);
833:      Wrt_Str(51,14,int2ch(count),WHITE,RED);
834:
835:      Wrt_Str(17,15,none,BLUE,BLUE);
836:      Wrt_Str(17,15,mess_info[19],WHITE,BLUE);
837:
838:      Wrt_Str(17,16,none,BLUE,BLUE);
839:      Wrt_Str(17,16,mess_info[20],WHITE,BLUE);
```

```
840:
842:     Wrt_Str(17,17,none,BLUE,BLUE);
843:     Wrt_Str(17,17,mess_info[21],WHITE,BLUE);
844:
845:     Wrt_Str(17,18,none,BLUE,BLUE);
846:     Wrt_Str(17,18,mess_info[22],WHITE,BLUE);
847:
848:     Wrt_Str(17,19,none,BLUE,BLUE);
849:     Wrt_Str(17,19,mess_info[23],WHITE,BLUE);
850:
851:     Wrt_Str(17,20,none,BLUE,BLUE);
852:     Wrt_Str(17,20,mess_info[24],WHITE,BLUE);
853:     /************************************/
854:
855:     INV_ZAK( zf_R->c4, f_R->x2, M1, N1 );
856:
857:     /*** Scroll Messages Up ***/
858:     for (q=3;q<21;q++)
859:        Wrt_Str(17,q,none,BLUE,BLUE);
860:
861:     for (q=8;q<15;q++)
862:        Wrt_Str(17,q-5,mess_info[q],WHITE,BLUE);
863:
864:     if (no_file_name == FALSE)
865:     {
866:         Wrt_Str(17,10,none,BLUE,BLUE);
867:         Wrt_Str(17,10,mess_info[15],WHITE,BLUE);
868:         Wrt_Str(55,10,imageC,WHITE,RED);
869:     }  else
870:         {
871:             Wrt_Str(17,10,none,BLUE,BLUE);
872:             Wrt_Str(17,10,mess_info[15],WHITE,BLUE);
873:         }
874:
875:     Wrt_Str(17,11,none,BLUE,BLUE);
876:     Wrt_Str(17,11,mess_info[16],WHITE,BLUE);
877:
878:     Wrt_Str(17,12,none,BLUE,BLUE);
879:     Wrt_Str(17,12,mess_info[17],WHITE,BLUE);
880:     Wrt_Str(51,12,int2ch(M1*M1*N1*N1 - count),WHITE,RED);
881:
882:     Wrt_Str(17,13,none,BLUE,BLUE);
883:     Wrt_Str(17,13,mess_info[18],WHITE,BLUE);
884:     Wrt_Str(51,13,int2ch(count),WHITE,RED);
885:
886:     Wrt_Str(17,14,none,BLUE,BLUE);
887:     Wrt_Str(17,14,mess_info[19],WHITE,BLUE);
888:
889:     Wrt_Str(17,15,none,BLUE,BLUE);
890:     Wrt_Str(17,15,mess_info[20],WHITE,BLUE);
891:
892:     Wrt_Str(17,16,none,BLUE,BLUE);
893:     Wrt_Str(17,16,mess_info[21],WHITE,BLUE);
894:
895:     Wrt_Str(17,17,none,BLUE,BLUE);
```

```
896:     Wrt_Str(17,17,mess_info[22],WHITE,BLUE);
898:
899:     Wrt_Str(17,18,none,BLUE,BLUE);
900:     Wrt_Str(17,18,mess_info[23],WHITE,BLUE);
901:
902:     Wrt_Str(17,19,none,BLUE,BLUE);
903:     Wrt_Str(17,19,mess_info[24],WHITE,BLUE);
904:
905:     Wrt_Str(17,20,none,BLUE,BLUE);
906:     Wrt_Str(17,20,mess_info[25],WHITE,BLUE);
907:     /**********************************/
908:
909:     free( zf_R );
910:
911:     /* End Image synthesis */
912:
913:     /* output the synthesized image */
914:
915:     /*** Scroll Messages Up ***/
916:     for (q=3;q<21;q++)
917:        Wrt_Str(17,q,none,BLUE,BLUE);
918:
919:     for (q=9;q<15;q++)
920:        Wrt_Str(17,q-6,mess_info[q],WHITE,BLUE);
921:
922:     if (no_file_name == FALSE)
923:     {
924:         Wrt_Str(17,9,none,BLUE,BLUE);
925:         Wrt_Str(17,9,mess_info[15],WHITE,BLUE);
926:         Wrt_Str(55,9,imageC,WHITE,RED);
927:     }  else
928:         {
929:             Wrt_Str(17,9,none,BLUE,BLUE);
930:             Wrt_Str(17,9,mess_info[15],WHITE,BLUE);
931:         }
932:
933:     Wrt_Str(17,10,none,BLUE,BLUE);
934:     Wrt_Str(17,10,mess_info[16],WHITE,BLUE);
935:
936:     Wrt_Str(17,11,none,BLUE,BLUE);
937:     Wrt_Str(17,11,mess_info[17],WHITE,BLUE);
938:     Wrt_Str(51,11,int2ch(M1*M1*N1*N1 - count),WHITE,RED);
939:
940:     Wrt_Str(17,12,none,BLUE,BLUE);
941:     Wrt_Str(17,12,mess_info[18],WHITE,BLUE);
942:     Wrt_Str(51,12,int2ch(count),WHITE,RED);
943:
944:     Wrt_Str(17,13,none,BLUE,BLUE);
945:     Wrt_Str(17,13,mess_info[19],WHITE,BLUE);
946:
947:     Wrt_Str(17,14,none,BLUE,BLUE);
948:     Wrt_Str(17,14,mess_info[20],WHITE,BLUE);
949:
950:     Wrt_Str(17,15,none,BLUE,BLUE);
951:     Wrt_Str(17,15,mess_info[21],WHITE,BLUE);
```

```
952:
954:     Wrt_Str(17,16,none,BLUE,BLUE);
955:     Wrt_Str(17,16,mess_info[22],WHITE,BLUE);
956:
957:     Wrt_Str(17,17,none,BLUE,BLUE);
958:     Wrt_Str(17,17,mess_info[23],WHITE,BLUE);
959:
960:     Wrt_Str(17,18,none,BLUE,BLUE);
961:     Wrt_Str(17,18,mess_info[24],WHITE,BLUE);
962:
963:     Wrt_Str(17,19,none,BLUE,BLUE);
964:     Wrt_Str(17,19,mess_info[25],WHITE,BLUE);
965:
966:     Wrt_Str(17,20,none,BLUE,BLUE);
967:     Wrt_Str(17,20,mess_info[26],WHITE,BLUE);
968:     /************************************/
969:
970:     for (i = 0; i < side; i++)
971:     {
972:         for (j = 0; j < side; j++)
973:         {
974:             val1 = f_R->x2[i][j];
975:             val2 = (int)( val1 + 0.5 );
976:             if (val2 > 255)
977:                 val2 = 255;
978:             ftemp[j] = (byte)(val2);
979:            spectr[j] = (byte)intensity[i][j];
980:         }
981:         fwrite( ftemp, 1, side, z_t );
982:         fwrite( spectr, 1, side, spectrOut );
983:
984:     }
985:     fclose( z_t );
986:     fclose( spectrOut );
987:
988:     /*** Scroll Messages Up ***/
989:     for (q=3;q<21;q++)
990:        Wrt_Str(17,q,none,BLUE,BLUE);
991:
992:     for (q=10;q<15;q++)
993:        Wrt_Str(17,q-7,mess_info[q],WHITE,BLUE);
994:
995:     if (no_file_name == FALSE)
996:     {
997:         Wrt_Str(17,8,none,BLUE,BLUE);
998:         Wrt_Str(17,8,mess_info[15],WHITE,BLUE);
999:         Wrt_Str(55,8,imageC,WHITE,RED);
1000:    }  else
1001:        {
1002:            Wrt_Str(17,8,none,BLUE,BLUE);
1003:            Wrt_Str(17,8,mess_info[15],WHITE,BLUE);
1004:        }
1005:
1006:    Wrt_Str(17,9,none,BLUE,BLUE);
1007:    Wrt_Str(17,9,mess_info[16],WHITE,BLUE);
```

```
1008:
1010:     Wrt_Str(17,10,none,BLUE,BLUE);
1011:     Wrt_Str(17,10,mess_info[17],WHITE,BLUE);
1012:     Wrt_Str(51,10,int2ch(M1*M1*N1*N1 - count),WHITE,RED);
1013:
1014:     Wrt_Str(17,11,none,BLUE,BLUE);
1015:     Wrt_Str(17,11,mess_info[18],WHITE,BLUE);
1016:     Wrt_Str(51,11,int2ch(count),WHITE,RED);
1017:
1018:     Wrt_Str(17,12,none,BLUE,BLUE);
1019:     Wrt_Str(17,12,mess_info[19],WHITE,BLUE);
1020:
1021:     Wrt_Str(17,13,none,BLUE,BLUE);
1022:     Wrt_Str(17,13,mess_info[20],WHITE,BLUE);
1023:
1024:     Wrt_Str(17,14,none,BLUE,BLUE);
1025:     Wrt_Str(17,14,mess_info[21],WHITE,BLUE);
1026:
1027:     Wrt_Str(17,15,none,BLUE,BLUE);
1028:     Wrt_Str(17,15,mess_info[22],WHITE,BLUE);
1029:
1030:     Wrt_Str(17,16,none,BLUE,BLUE);
1031:     Wrt_Str(17,16,mess_info[23],WHITE,BLUE);
1032:
1033:     Wrt_Str(17,17,none,BLUE,BLUE);
1034:     Wrt_Str(17,17,mess_info[24],WHITE,BLUE);
1035:
1036:     Wrt_Str(17,18,none,BLUE,BLUE);
1037:     Wrt_Str(17,18,mess_info[25],WHITE,BLUE);
1038:
1039:     Wrt_Str(17,19,none,BLUE,BLUE);
1040:     Wrt_Str(17,19,mess_info[26],WHITE,BLUE);
1041:
1042:     Wrt_Str(17,20,none,BLUE,BLUE);
1043:     Wrt_Str(17,20,mess_info[27],WHITE,BLUE);
1044:     /************************************/
1045:
1046:     free( f_R );
1047:
1048:     /*** Scroll Messages Up ***/
1049:     for (q=3;q<21;q++)
1050:        Wrt_Str(17,q,none,BLUE,BLUE);
1051:
1052:     for (q=11;q<15;q++)
1053:        Wrt_Str(17,q-8,mess_info[q],WHITE,BLUE);
1054:
1055:     if (no_file_name == FALSE)
1056:     {
1057:         Wrt_Str(17,7,none,BLUE,BLUE);
1058:         Wrt_Str(17,7,mess_info[15],WHITE,BLUE);
1059:         Wrt_Str(55,7,imageC,WHITE,RED);
1060:     }  else
1061:         {
1062:             Wrt_Str(17,7,none,BLUE,BLUE);
1063:             Wrt_Str(17,7,mess_info[15],WHITE,BLUE);
```

```
1064:          }
1066:
1067:      Wrt_Str(17,8,none,BLUE,BLUE);
1068:      Wrt_Str(17,8,mess_info[16],WHITE,BLUE);
1069:
1070:      Wrt_Str(17,9,none,BLUE,BLUE);
1071:      Wrt_Str(17,9,mess_info[17],WHITE,BLUE);
1072:      Wrt_Str(51,9,int2ch(M1*M1*N1*N1 - count),WHITE,RED);
1073:
1074:      Wrt_Str(17,10,none,BLUE,BLUE);
1075:      Wrt_Str(17,10,mess_info[18],WHITE,BLUE);
1076:      Wrt_Str(51,10,int2ch(count),WHITE,RED);
1077:
1078:      Wrt_Str(17,11,none,BLUE,BLUE);
1079:      Wrt_Str(17,11,mess_info[19],WHITE,BLUE);
1080:
1081:      Wrt_Str(17,12,none,BLUE,BLUE);
1082:      Wrt_Str(17,12,mess_info[20],WHITE,BLUE);
1083:
1084:      Wrt_Str(17,13,none,BLUE,BLUE);
1085:      Wrt_Str(17,13,mess_info[21],WHITE,BLUE);
1086:
1087:      Wrt_Str(17,14,none,BLUE,BLUE);
1088:      Wrt_Str(17,14,mess_info[22],WHITE,BLUE);
1089:
1090:      Wrt_Str(17,15,none,BLUE,BLUE);
1091:      Wrt_Str(17,15,mess_info[23],WHITE,BLUE);
1092:
1093:      Wrt_Str(17,16,none,BLUE,BLUE);
1094:      Wrt_Str(17,16,mess_info[24],WHITE,BLUE);
1095:
1096:      Wrt_Str(17,17,none,BLUE,BLUE);
1097:      Wrt_Str(17,17,mess_info[25],WHITE,BLUE);
1098:
1099:      Wrt_Str(17,18,none,BLUE,BLUE);
1100:      Wrt_Str(17,18,mess_info[26],WHITE,BLUE);
1101:
1102:      Wrt_Str(17,19,none,BLUE,BLUE);
1103:      Wrt_Str(17,19,mess_info[27],WHITE,BLUE);
1104:
1105:      Wrt_Str(17,20,none,BLUE,BLUE);
1106:      Wrt_Str(17,20,mess_info[28],WHITE,RED);
1107:      /**********************************/
1108:
1109:      pauseb();
1110:
1111:   return 0;
1112: }
1113: /*****************************************************************************/
1114:
1115:
1116: /*****************************************************************************/
1117: ZAK( f4, zf4, m1, n1 )
1118: complex zf4[M][M][N][N];
1119: float   f4[M][N][M][N];
```

```
1120: int            m1, n1;
1122: {
1123:     int     i, j, m, n, r, s, ro, sig, arg, dim[3];
1124:     float   data[N*N], fdata[2*N*N+1];
1125:     complex z;
1126:
1127:     dim[1] = dim[2] = n1;
1128:
1129:     for (m = 0; m < m1; m++)
1130:     {
1131:         for (n = 0; n < m1; n++)
1132:         {
1133:             for (r = 0; r < n1; r++)
1134:             {
1135:                 for (s = 0; s < n1; s++)
1136:                 {
1137:                     data [s + r*n1] = f4[m][r][n][s];
1138:                 }
1139:             }
1140:
1141:             for ( i = 0; i < 2*n1*n1; i += 2 )
1142:             {
1143:                 fdata[i] = 0;
1144:                 fdata[i+1] = data[i/2];
1145:             }
1146:             fdata[2*n1*n1] = 0;
1147:
1148:             fourn(fdata, dim, 2, -1);
1149:
1150:             for (ro = 0; ro < n1; ro++)
1151:             {
1152:                 for (sig = 0; sig < 2 * n1; sig = sig + 2)
1153:                 {
1154:                     arg = 2*n1*ro + sig + 1;
1155:                     z = cmplx(fdata[arg], fdata[arg+1]);
1156:                     zf4[m][n][ro][sig/2] = z;
1157:                 }
1158:             }
1159:
1160:         }
1161:     }
1162:
1163: }
1164: /*****************************************************************************/
1165:
1166: /*****************************************************************************/
1167: twotofour( in, ou, m1, n1 )
1168: float in[M*N][M*N], ou[M][N][M][N];
1169: int   m1, n1;
1170: {
1171:     int     m, r, n, s;
1172:
1173:     for (m = 0; m < m1; m++)
1174:     {
1175:         for (r = 0; r < n1; r++)
```

```
1176:         {
1178:             for (n = 0; n < m1; n++)
1179:             {
1180:                 for (s = 0; s < n1; s++)
1181:                 {
1182:                     ou[m][r][n][s] = in[m+r*m1][n+s*m1];
1183:
1184:                 }
1185:             }
1186:         }
1187:     }
1188: }
1189: /*****************************************************************************/
1190:
1191: /*****************************************************************************/
1192: INV_ZAK( f4, if2, m1, n1 )
1193: complex f4[M][M][N][N];
1194: float   if2[M*N][M*N];
1195: int             m1, n1;
1196: {
1197:     int     i, j, kl, m, n, r, s, ro, sig, arg;
1198:     int             dim[3];
1199:     float   size;
1200:     double  val;
1201:     float   fdata[2*N*N+1];
1202:     complex data[N*N];
1203:     complex z;
1204:
1205:     dim[1] = dim[2] = n1;
1206:     size = (float)( n1 * n1 );
1207:
1208:     for (m = 0; m < m1; m++) {
1209:         for (n = 0; n < m1; n++) {
1210:             for (r = 0; r < n1; r++) {
1211:                 for (s = 0; s < n1; s++) {
1212:                     data [s + r*n1] = f4[m][n][r][s];
1213:                 }
1214:             }
1215:
1216:             for ( i = 1; i <= 2*n1*n1; i += 2 ) {
1217:                 fdata[i] = data[(i-1)/2].x;
1218:                 fdata[i+1] = data[(i-1)/2].y;
1219:             }
1220:
1221:             fourn(fdata, dim, 2, 1);
1222:
1223:             for (ro = 0; ro < n1; ro++) {
1224:                 for (sig = 0; sig < 2*n1; sig += 2) {
1225:                     arg = 2*n1*ro + sig + 1;
1226:                     r = ro;
1227:                     s = sig / 2;
1228:                     z = cmplx( fdata[arg], fdata[arg+1] );
1229:                     val = cabs( z ) / size;
1230:                     if2 [m+r*m1][n+s*m1] = val;
1231:                 }
```

```
1232:             }
1234:
1235:         }
1236:     }
1237:
1238: }
1239: /*****************************************************************************/
1240:
1241: /*****************************************************************************/
1242: get_values( n, a, deg, g )
1243: float   g[M*N][M*N];
1244: float   a;
1245: int     deg, n;
1246: {
1247:     double  ex, fac, ii, jj, val, n_dbl;
1248:     int     i, j;
1249:
1250:     n_dbl = (double)( n );
1251:
1252:     for (i = 0; i < n; i++)
1253:     {
1254:     ii = ( 2. * ((double)( i )) - n_dbl + 1. ) / n_dbl;
1255:         for (j = 0; j < n; j++)
1256:         {
1257:         jj = ( 2. * ((double)( j )) - n_dbl + 1. ) / n_dbl;
1258:
1259:                 switch( deg )
1260:                 {
1261:                     case 0: fac = 1.0;
1262:                             break;
1263:                     case 1: fac = ( 5.01326*ii )*( 5.01326*jj );
1264:                             break;
1265:                     case 2: fac = ( 25.1327*ii*ii - 2.0 ) * ( 25.1327*jj*jj
1266:                                     - 2.0 );
1267:                             break;
1268:                     case 3: fac = ( -30.0795*ii + 125.997*ii*ii*ii )
1269:                                     * ( -30.0795*jj + 125.997*jj*jj*jj );
1270:                             break;
1271:                     case 4: fac = ( 631.6547*ii*ii*ii*ii - 301.5929*ii*ii
1272:                                     + 12. ) * ( 631.6547*jj*jj*jj*jj
1273:                                     - 301.5929*jj*jj +12. );
1274:                             break;
1275:                     case 5: fac = ( 3166.6470*ii*ii*ii*ii*ii - 2519.9376*ii*ii*ii
1276:                                     + 300.7954*ii )*( 3166.6470*jj*jj*jj*jj*jj
1277:                                     -2519.9376*jj*jj*jj + 300.7954*jj );
1278:                             break;
1279:                     default: fac = 1.0;
1280:                             break;
1281:                 }
1282:             val = -a * a * ( ii*ii + jj*jj ) / 2.;
1283:             ex = fac * exp( val );
1284:             g[i][j] = (float)( ex );
1285:         }
1286:     }
1287: }
```

```
1288: /*****************************************************************************/
1290:
1291: /*****************************************************************************/
1292: fourn( data, nn, ndim, isign )
1293: float   data[];
1294: int     nn[], ndim, isign;
1295: {
1296:     int     i1, i2, i3, i2rev, i3rev, ip1, ip2, ip3, ifp1, ifp2;
1297:     int     ibit, idim, k1, k2, n, nprev, nrem, ntot;
1298:     float   tempi, tempr;
1299:     double  theta, wi, wpi, wpr, wr, wtemp;
1300:
1301:     ntot = 1;
1302:     for (idim = 1; idim <= ndim; idim++)
1303:         ntot *= nn[idim];
1304:     nprev = 1;
1305:     for (idim = ndim; idim >= 1; idim--)
1306:     {
1307:         n = nn[idim];
1308:         nrem = ntot / (n * nprev);
1309:         ip1 = nprev << 1;
1310:         ip2 = ip1 * n;
1311:         ip3 = ip2 * nrem;
1312:         i2rev = 1;
1313:         for (i2 = 1; i2 <= ip2; i2 += ip1)
1314:         {
1315:             if (i2 < i2rev)
1316:             {
1317:                 for (i1 = i2; i1 <= i2 + ip1 - 2; i1 += 2)
1318:                 {
1319:                     for (i3 = i1; i3 <= ip3; i3 += ip2)
1320:                     {
1321:                         i3rev = i2rev + i3 - i2;
1322:                         SWAP(data[i3], data[i3rev]);
1323:                         SWAP(data[i3+1], data[i3rev+1]);
1324:                     }
1325:                 }
1326:             }
1327:             ibit = ip2 >> 1;
1328:             while (ibit >= ip1 && i2rev > ibit)
1329:             {
1330:                 i2rev -= ibit;
1331:                 ibit >>= 1;
1332:             }
1333:             i2rev += ibit;
1334:         }
1335:         ifp1 = ip1;
1336:         while (ifp1 < ip2)
1337:         {
1338:             ifp2 = ifp1 << 1;
1339:             theta = isign * 6.28318530717959 / (ifp2 / ip1);
1340:             wtemp = sin(0.5 * theta);
1341:             wpr = -2.0 * wtemp * wtemp;
1342:             wpi = sin(theta);
1343:             wr = 1.0;
```

```
1344:             wi = 0.0;
1346:             for (i3 = 1; i3 <= ifp1; i3 += ip1)
1347:             {
1348:                 for (i1 = i3; i1 <= i3 + ip1 - 2; i1 += 2)
1349:                 {
1350:                     for (i2 = i1; i2 <= ip3; i2 += ifp2)
1351:                     {
1352:                         k1 = i2;
1353:                         k2 = k1 + ifp1;
1354:                         tempr = wr * data[k2] - wi * data[k2+1];
1355:                         tempi = wr * data[k2+1] + wi * data[k2];
1356:                         data[k2] = data[k1] - tempr;
1357:                         data[k2+1] = data[k1+1] - tempi;
1358:                         data[k1] += tempr;
1359:                         data[k1+1] += tempi;
1360:                     }
1361:                 }
1362:                 wr = (wtemp = wr) * wpr - wi * wpi + wr;
1363:                 wi = wi * wpr + wtemp * wpi + wi;
1364:             }
1365:             ifp1 = ifp2;
1366:         }
1367:         nprev *= n;
1368:     }
1369: }
```

```
 1: /*
 2:         FILENAME:           bdv.c
 3:         CREATED:            01-AUG-88
 4:         LAST MODIFIED:      19-SEP-91
 5:         CREATE BY:          Christopher Voltz - UDRI
 6:         MODIFIED BY:        Craig A. Vrana - UDRI
 7:         INTERFACE PROTOCOL: Microsoft C 5.1
 8:         REQUIREMENTS:       VGA
 9:
10:         LAST MODIFIED:      16-SEPT-93
11:         MODIFIED BY:        Alex Firdman - UDRI
12:         INTERFACE PROTOCOL: NDP-C V. 4.2.1
13:
14:         This program recieves as input, from the DOS command line, the
15:     filename of an image, with or without the .IMG extension, presumed to be
16:     in the PCVision IMAGEACTION format and displays it on the VGA screen
17:     using 64 gray shades.  However, only the first 200 lines of the
18:     image are displayed.  Images with a width more than 320 will be
19:     truncated when displayed.  When the user presses any key, the screen
20:     returns to the same mode as before this program was called.
21:
22:         BIOS calls are used.  While this does not help portability, it
23:     was the only method available to utilize the full capability of the
24:     VGA at the time this program was developed, ie. the VGA was only
25:     partially supported by the compiler.
26:
27:
28: */
29:
30: # include <stdio.h>
31: # include <stdlib.h>
32: # include <os.h>
33: # include <conio.h>
34: # include <string.h>
35: # include <grex.h>
36:
37:
38: # define MAX_X              640   /* maximum absolute x coordinate          */
39: # define MAX_Y              480   /* maximum absolute y coordinate          */
40: # define NUM_GRAY_LEVELS    64    /* number of gray-scale levels            */
41: # define VIDEO_INT          0x10  /* interrupt number for video functions   */
42:
43: # define _MRES256COLOR      19
44: # define _DEFAULTMODE        3
45:
46: disp_image(img_name, img_size)
47: char img_name[64];
48: unsigned img_size;
49: {
50:     FILE        * file_in;
51:     unsigned char   val;
52:     unsigned int    x;
53:     unsigned int    x_size;
54:     unsigned int    y;
55:     unsigned int    y_size;
```

```
56:     int             flag=0;
58:     union REGS16    hw_reg,
59:                     out_reg;
60:
61:
62:         /* calculate image name */
63:     if (!strchr (img_name, '.'))
64:         strcat (img_name, ".img");
65:
66:         /* try and open image file */
67:     if ((file_in=fopen (img_name, "rb")) == NULL)
68:     {
69:         graphic_text("Cannot open Input File",80,190,250);
70:         exit(1);
71:     }
72:
73:         /* get image size */
74:     x_size = img_size;
75:     y_size = x_size;
76:
77:         /* set mode to 320x200, 256 colors  */
78:     _setvideomode(_MRES256COLOR);               /* function code    */
79:
80:         /* program pallette to display 64 gray scales */
81:     for (val=NUM_GRAY_LEVELS, x=0; x<256; x++)
82:     {
83:         if (val < NUM_GRAY_LEVELS-1)        /* map 256 colors to 64 shades  */
84:             val++;
85:         else
86:             val = 0;
87:         hw_reg.h.dh = val;              /* red intensity        */
88:         hw_reg.h.cl = val;              /* blue intensity       */
89:         hw_reg.h.ch = val;              /* green intensity        */
90:         hw_reg.x.bx = x;                /* color number         */
91:         hw_reg.x.ax = 0x1010;           /* function code        */
92:         int86(VIDEO_INT,&hw_reg,&out_reg);      /* set palette entry      */
93:     }
94:
95:     move(0,0);
96:     set_color(250);
97:     draw(319,0);
98:     draw(319,199);
99:     draw(0,199);
100:    draw(0,0);
101:
102:        /* read in image and display */
103:    for (y=1; y<200 && y<MAX_Y && !feof (file_in); y++)
104:    {
105:        if(y%3)
106:        {
107:            if(flag==1)
108:            {
109:                y--;
110:                flag = 0;
111:            }
```

```
112:            for (x=0; x<x_size && x<MAX_X && !feof (file_in); x++)
114:            {
115:                val = fgetc (file_in) / (256/NUM_GRAY_LEVELS);
116:                hw_reg.x.dx = y;              /* row        */
117:                hw_reg.x.cx = x+1;            /* column     */
118:                hw_reg.h.bh = 0;              /* page number    */
119:                hw_reg.h.ah = 0x0C;           /* function code  */
120:                hw_reg.h.al = val;            /* intensity    */
121:                int86(VIDEO_INT,&hw_reg,&out_reg);   /* write the pixel  */
122:            }   /* for */
123:        }
124:        else
125:        {
126:            for (x=0; x<x_size && x<MAX_X && !feof (file_in); x++)
127:            {
128:                val = fgetc (file_in) / (256/NUM_GRAY_LEVELS);
129:                hw_reg.x.dx = y;              /* row        */
130:                hw_reg.x.cx = x+1;              /* column     */
131:                hw_reg.h.bh = 0;              /* page number    */
132:                hw_reg.h.ah = 0x0C;           /* function code  */
133:                hw_reg.h.al = val;            /* intensity    */
134:            }   /* for */
135:            flag = 1;
136:        }
137:        for (; x<x_size && !feof (file_in); x++)
138:            fgetc (file_in);
139:    }  /* for */
140:
141:    graphic_text(" Displaying Image :", 20, 187, 255);
142:    graphic_text(img_name,180,187,255);
143:
144:        /* wait for user to press a key */
145:    getch ();
146:
147:        /* clear screen */
148:    _setvideomode(_DEFAULTMODE);
149:
150: fclose(file_in);
151: return(0);
152: }
```

```
 1: /*
 2:         LAST MODIFIED BY :  Alex Firdman
 3:         MODIFIED ON      :  15-SEPT-1993
 4:         COMPILER USED    :  NDP-C, Ver 4.2.1
 5:
 6:         This program recieves as input, from the DOS command line, the
 7:     filename of an image, with or without the .IMG extension, presumed to be
 8:     in the PCVision IMAGEACTION format and displays it on the VGA screen
 9:     using 64 gray shades.
10:
11:     Modified for higher resolution available on SVGA cards. The purpose
12:     of this modification is to display two images - original and processed
13:     on the same screen simultaneously.
14:
15:         BIOS calls are used.  While this does not help portability, it
16:     was the only method available to utilize the full capability of the
17:     VGA at the time this program was developed, ie. the VGA was only
18:     partially supported by the compiler.
19:
20:         Text has to be bit-mapped, since Trident SVGA mode is not supported.
21: */
22:
23: # include <stdio.h>
24: # include <stdlib.h>
25: # include <os.h>
26: # include <conio.h>
27: # include <string.h>
28: # include <grex.h>
29:
30: # include "5x8.inc"   /** Text bit map to print in Trident SVGA mode **/
31:
32: # define RES256_640_400   0x05c  /** Trident 640x400x256 SVGA mode **/
33: # define TEXT_MODE        0x03   /** Trident 80x40 color text mode **/
34: # define NUM_GRAY_LEVELS  64
35:
36: # define MAX_COL_PIX      640
37: # define MAX_ROW_PIX      400
38:
39:
40: # define MAX_PIX            300
41: # define MAX_X              640   /* maximum absolute x coordinate          */
42: # define MAX_Y              480   /* maximum absolute y coordinate          */
43: # define NUM_GRAY_LEVELS    64    /* number of gray-scale levels            */
44: # define VIDEO_INT          0x10  /* interrupt number for video functions   */
45:
46: # define _DEFAULTMODE        3
47:
48: char *int2ch(int);
49:
50: void disp_comp(int, char [], char []);
51:
52: typedef unsigned char BYTE;
53:
54: FILE    *file_in1, *file_in2, *file_in3, *fin1, *fin2;
55:
```

```
 56: int comp_image(img_name1,img_name2,img_size,img_name3)
 58: char img_name1[13], img_name2[13], img_name3[13];
 59: unsigned int img_size;
 60: {
 61:     unsigned char   val;
 62:     unsigned int    x;
 63:     unsigned int    x_size;
 64:     unsigned int    y;
 65:     unsigned int    y_size;
 66:     int             flag=0;
 67:     union REGS16    hw_reg,
 68:                     out_reg;
 69:
 70:     int             i,j;
 71:     int             p=0,q=0,m=0;
 72:     float           percent;
 73:
 74:     int             limx, limy, max_color;
 75:     int             mode, device[4];
 76:
 77:         /* calculate image name */
 78:
 79:     if (!strchr (img_name1, '.'))
 80:         strcat (img_name1, ".img");
 81:
 82:     if (!strchr (img_name2, '.'))
 83:         strcat (img_name1, ".img");
 84:
 85:         /* try and open image file */
 86:     if ((file_in1=fopen (img_name1, "rb")) == NULL)
 87:     {
 88:         _setvideomode(_DEFAULTMODE);
 89:         printf(" !!!! Cannot Open 1st Input Image File -> %s\n", file_in1);
 90:         exit(1);
 91:     }
 92:
 93:     if ((file_in2=fopen (img_name2, "rb")) == NULL)
 94:     {
 95:         _setvideomode(_DEFAULTMODE);
 96:         printf(" !!!! Cannot Open 2nd Input Image File -> %s\n", file_in2);
 97:         exit(1);
 98:     }
 99:
100:     x_size = img_size;
101:
102:     y_size = x_size;
103:
104:     /* Displaying in 640x400x256 Super Vga Mode */
105:
106:     init_trident();
107:
108:     make_board(0,0,318,199,250);
109:
110:         /* read in image and display */
111:     for (y=1; y<400 && y<MAX_Y && !feof (file_in1); y++)
```

```
112:    {
114:        if(y%3)
115:        {
116:            if(flag==1)
117:            {
118:                y--;
119:                flag = 0;
120:            }
121:            for (x=0; x<x_size && x<MAX_X && !feof (file_in1); x++)
122:            {
123:                val = fgetc (file_in1) / (256/NUM_GRAY_LEVELS);
124:                hw_reg.x.dx = y;              /* row        */
125:                hw_reg.x.cx = x+1;              /* column     */
126:                hw_reg.h.bh = 0;              /* page number    */
127:                hw_reg.h.ah = 0x0C;           /* function code  */
128:                hw_reg.h.al = val;            /* intensity    */
129:                int86(VIDEO_INT,&hw_reg,&out_reg);   /* write the pixel  */
130:            }   /* for */
131:        }
132:        else
133:        {
134:            for (x=0; x<x_size && x<MAX_X && !feof (file_in1); x++)
135:            {
136:                val = fgetc (file_in1) / (256/NUM_GRAY_LEVELS);
137:                hw_reg.x.dx = y;              /* row        */
138:                hw_reg.x.cx = x+1;              /* column     */
139:                hw_reg.h.bh = 0;              /* page number    */
140:                hw_reg.h.ah = 0x0C;           /* function code  */
141:                hw_reg.h.al = val;            /* intensity    */
142:            }   /* for */
143:            flag = 1;
144:        }
145:        for (; x<x_size && !feof (file_in1); x++)
146:            fgetc (file_in1);
147:    }  /* for */
148:
149:    wr_gr_str(20,191,"original image : ");
150:    wr_gr_str(130,191,img_name1);
151:
152:    make_board(320,0,316,199,250);
153:
154:        /* read in image and display */
155:    for (y=1; y<400 && y<MAX_Y && !feof (file_in2); y++)
156:    {
157:        if(y%3)
158:        {
159:            if(flag==1)
160:            {
161:                y--;
162:                flag = 0;
163:            }
164:            for (x=0; x<x_size && x<MAX_X && !feof (file_in2); x++)
165:            {
166:                val = fgetc (file_in2) / (256/NUM_GRAY_LEVELS);
167:                hw_reg.x.dx = y;              /* row        */
```

```
168:                hw_reg.x.cx = x+321;            /* column     */
170:                hw_reg.h.bh = 0;              /* page number    */
171:                hw_reg.h.ah = 0x0C;           /* function code  */
172:                hw_reg.h.al = val;            /* intensity    */
173:                int86(VIDEO_INT,&hw_reg,&out_reg);   /* write the pixel  */
174:            }   /* for */
175:        }
176:        else
177:        {
178:            for (x=0; x<x_size && x<MAX_X && !feof (file_in2); x++)
179:            {
180:                val = fgetc (file_in2) / (256/NUM_GRAY_LEVELS);
181:                hw_reg.x.dx = y;              /* row        */
182:                hw_reg.x.cx = x+321;            /* column     */
183:                hw_reg.h.bh = 0;              /* page number    */
184:                hw_reg.h.ah = 0x0C;           /* function code  */
185:                hw_reg.h.al = val;            /* intensity    */
186:            }   /* for */
187:            flag = 1;
188:        }
189:        for (; x<x_size && !feof (file_in2); x++)
190:            fgetc (file_in2);
191:    }  /* for */
192:
193:    wr_gr_str(340,191,"processed image : ");
194:    wr_gr_str(500,191,img_name2);
195:
196:    make_board(0,201,318,197,250);
197:
198:        /* read in Spectrum image and display */
199: if (strlen(img_name3) > 5)
200: {
201:    if ((file_in3=fopen (img_name3,"rb")) == NULL)
202:    {
203:        _setvideomode(_DEFAULTMODE);
204:        printf(" !!!! Cannot Open 3d Input Image File -> %s\n", file_in3);
205:        exit(1);
206:    }
207:
208:    for (y=202; y<400 && y<MAX_Y && !feof (file_in3); y++)
209:    {
210:        if(y%3)
211:        {
212:            if(flag==1)
213:            {
214:                y--;
215:                flag = 0;
216:            }
217:            for (x=0; x<x_size && x<MAX_X && !feof (file_in3); x++)
218:            {
219:                val = fgetc (file_in3) / (256/NUM_GRAY_LEVELS);
220:                hw_reg.x.dx = y;              /* row        */
221:                hw_reg.x.cx = x+1;              /* column     */
222:                hw_reg.h.bh = 0;              /* page number    */
223:                hw_reg.h.ah = 0x0C;           /* function code  */
```

```
224:                hw_reg.h.al = val;          /* intensity    */
226:                int86(VIDEO_INT,&hw_reg,&out_reg);   /* write the pixel  */
227:            }   /* for */
228:        }
229:        else
230:        {
231:            for (x=0; x<x_size && x<MAX_X && !feof (file_in3); x++)
232:            {
233:                val = fgetc (file_in3) / (256/NUM_GRAY_LEVELS);
234:                hw_reg.x.dx = y;            /* row        */
235:                hw_reg.x.cx = x+1;            /* column     */
236:                hw_reg.h.bh = 0;            /* page number    */
237:                hw_reg.h.ah = 0x0C;         /* function code  */
238:                hw_reg.h.al = val;          /* intensity    */
239:            }   /* for */
240:            flag = 1;
241:        }
242:        for (; x<x_size && !feof (file_in3); x++)
243:            fgetc (file_in3);
244:    }  /* for */
245:
246:    wr_gr_str(20,390,"coefficient spectrum : ");
247:    wr_gr_str(216,390,img_name3);
248:
249:
250: fclose(file_in3);
251: }
252: else
253:    {
254:        wr_gr_str(16,300,"coefficient spectrum  was not entered...");
255:    }
256:
257:    make_board(320,201,316,197,250);
258:
259:    disp_comp(img_size, img_name1, img_name2);
260:
261: fclose(file_in1);
262: fclose(file_in2);
263:
264:
265:        /* wait for user to press a key */
266:    pauseb();
267:
268:        /* clear screen */
269:    text_trident_mode();
270:
271: return 0;
272: }
273:
274: void disp_comp(size, name1, name2)
275: int size;
276: char name1[13];
277: char name2[13];
278: {
279:    int i,j;
```

```
280:     unsigned int diff=0;
282:     unsigned int same=0;
283:     int m=0;
284:     BYTE image1[MAX_PIX];
285:     BYTE image2[MAX_PIX];
286:     float percent;
287:
288:     if ((fin1 = fopen(name1,"rb")) == NULL)
289:     {
290:         _setvideomode(_DEFAULTMODE);
291:         printf(" !!!! Cannot Open 1st Input Image File -> %s\n", name1);
292:         exit(1);
293:     }
294:
295:
296:     if ((fin2 = fopen(name2,"rb")) == NULL)
297:     {
298:         _setvideomode(_DEFAULTMODE);
299:         printf(" !!!! Cannot Open 2nd Input Image File -> %s\n", name2);
300:         exit(1);
301:     }
302:
303:     wr_gr_str(324,250,"total number of pixels processed :");
304:
305:     for (i=0;i<size;i++)
306:     {
307:       fread(image1, 1, size, fin1);
308:       fread(image2, 1, size, fin2);
309:
310:       for (j=0;j<size;j++)
311:       {
312:          m++;
313:          if (image1[j] == image2[j])
314:             same++;
315:          else
316:             diff++;
317:       }
318:     }
319:
320:         wr_gr_str(550,250,int2ch(m));
321:
322:         percent = (float) diff/(same+diff)*100;
323:
324:         wr_gr_str(324,280,"number of identical pixels   ");
325:         wr_gr_str(550,280,(char*)int2ch(same));
326:
327:         wr_gr_str(324,300,"number of different pixels   ");
328:         wr_gr_str(550,300,(char*)int2ch(diff));
329:
330:         wr_gr_str(324,320,"image 1 and image 2 differ   ");
331:
332:         wr_gr_str(550,320,(char*)int2ch((int) (percent)));
333:
334:         wr_gr_str(595,320," % ");
335:
```

```
336: fclose(fin1);
338: fclose(fin2);
339:
340: return;
341: }
342:
343: int make_board(colS,rowS,width,hight,intens)
344: int rowS;
345: int colS;
346: int width;
347: int hight;
348: int intens;
349: {
350: int j, i;
351: union REGS16  inregs, outregs;
352:
353:         j=0;
354:       /*** Draw Upper Horizontal Line ***/
355:       for (j=colS; j<(colS+width); j++)
356:       {
357:         inregs.x.dx = rowS;                  /* row             */
358:         inregs.x.cx = j;                     /* column          */
359:         inregs.h.bh = 0;                     /* page number     */
360:         inregs.h.ah = 0x0c;                  /* function code   */
361:         inregs.h.al = intens;                /* intensity       */
362:         int86(VIDEO_INT,&inregs,&outregs);   /* write the pixel */
363:       }
364:         j=0;
365:       /*** Draw lower Horizontal Line ***/
366:       for (j=colS; j<(colS+width+1); j++)
367:       {
368:         inregs.x.dx = rowS+hight;            /* row             */
369:         inregs.x.cx = j;                     /* column          */
370:         inregs.h.bh = 0;                     /* page number     */
371:         inregs.h.ah = 0x0c;                  /* function code   */
372:         inregs.h.al = intens;                /* intensity       */
373:         int86(VIDEO_INT,&inregs,&outregs);   /* write the pixel */
374:       }
375:         j=0;
376:       /*** Draw Left Vertical line ***/
377:       for (j=rowS; j<(rowS+hight); j++)
378:       {
379:         inregs.x.dx = j;                     /* row             */
380:         inregs.x.cx = colS;                  /* column          */
381:         inregs.h.bh = 0;                     /* page number     */
382:         inregs.h.ah = 0x0c;                  /* function code   */
383:         inregs.h.al = intens;                /* intensity       */
384:         int86(VIDEO_INT,&inregs,&outregs);   /* write the pixel */
385:       }
386:         j=0;
387:       /*** Draw Right Vertical line ***/
388:       for (j=rowS; j<(rowS+hight); j++)
389:       {
390:         inregs.x.dx = j;                     /* row             */
391:         inregs.x.cx = colS+width;            /* column          */
```

```
392:         inregs.h.bh = 0;                    /* page number      */
394:         inregs.h.ah = 0x0c;                 /* function code    */
395:         inregs.h.al = intens;               /* intensity        */
396:         int86(VIDEO_INT,&inregs,&outregs);  /* write the pixel  */
397:       }
398: return 0;
399: }
400:
401: int text_trident_mode(void)
402: {
403: int val;
404: int color;
405: union REGS16  inregs, outregs;
406:
407:     /*** Set Video Mode to SVGA 640x400x256 ***/
408:     inregs.h.ah = 0x0;
409:     inregs.h.al = TEXT_MODE;
410:     int86(VIDEO_INT,&inregs,&outregs);
411:
412: return 0;
413: }
414:
415: int init_trident(void)
416: {
417: int val;
418: int color;
419: union REGS16  inregs, outregs;
420:
421:     /*** Set Video Mode to SVGA 640x400x256 ***/
422:     inregs.h.ah = 0x0;
423:     inregs.h.al = RES256_640_400;
424:     int86(VIDEO_INT,&inregs,&outregs);
425:
426:     /*** Set Palette to convert 256 Colors to 64 Shades ***/
427:
428:     for (val=NUM_GRAY_LEVELS, color=0; color<256; color++)
429:     {
430:         if (val < NUM_GRAY_LEVELS-1)        /* map 256 colors to 64 shades  */
431:             val++;
432:         else
433:             val = 0;
434:
435:         inregs.h.dh = val;                      /* red intensity        */
436:         inregs.h.cl = val;                      /* blue intensity       */
437:         inregs.h.ch = val;                      /* green intensity      */
438:         inregs.x.bx = color;                    /* color number         */
439:
440:         inregs.h.al = 0x10;
441:         inregs.h.ah = 0x10;                     /* function code        */
442:         int86(VIDEO_INT,&inregs,&outregs);      /* set palette entry    */
443:     }
444: return 0;
445: }
446:
447: int write_gr_ch(row,col,ch,sec)
```

```
448: int row;
450: int col;
451: int ch[25][8][6];
452: int sec;
453: {
454: union REGS16  inregs, outregs;
455: int i, j;
456: int added;
457:
458:     for (i=row; i<=row+6; i++)
459:     {
460:       for (j=col; j<col+8; j++)
461:       {
462:           inregs.h.ah = 0x0c;
463:           inregs.h.al = ch[sec][j-col][i-row];
464:           inregs.h.bh = 0;
465:           inregs.x.cx = i;
466:           inregs.x.dx = j;
467:           int86(VIDEO_INT,&inregs,&outregs);      /* set palette entry    */
468:       }
469:     }
470: return 0;
471: }
472:
473: int wr_gr_str(row,col,str)
474: int row;
475: int col;
476: char str[40];
477: {
478: int i,j,m=0,n=0;
479: char alfa[42] = {'a','b','c','d','e','f','g','h','i','j','k','l','m','n','o','p','q','r','s','t','u','v','w','x','y',',','.',':','-','\\','%',' ','1
480:
481:     for(i=0; i<strlen(str); i++)
482:     {
483:         for(j=0; j<42; j++)
484:         {
485:             if(str[i] == alfa[j])
486:             {
487:                 write_gr_ch(row+m,col,ABC,j);
488:                 m += 6;
489:             }
490:         }
491:     }
492:
493: return 0;
494: }
```

```
int ABC[25][8][6] = {
                    {       {       0,  0,  0,250,250,  0},
                            {       0,  0,250,  0,250,  0},
                            {       0,250,  0,  0,250,  0},
                            {       0,250,  0,  0,250,  0},
                            {       0,250,250,250,250,  0},
                            {       0,250,  0,  0,250,  0},
                            {       0,250,  0,  0,250,  0},
                            {       0,250,  0,  0,250,  0} },

                    {       {   0,250,250,250,  0,  0},
                            {       0,250,  0,  0,250,  0},
                            {       0,250,  0,  0,250,  0},
                            {       0,250,250,250,  0,  0},
                            {       0,250,  0,  0,250,  0},
                            {       0,250,  0,  0,250,  0},
                            {       0,250,  0,  0,250,  0},
                            {       0,250,250,250,  0,  0} },

                    {       {   0,  0,250,250,  0,  0},
                            {       0,250,  0,  0,250,  0},
                            {       0,250,  0,  0,250,  0},
                            {       0,250,  0,  0,  0,  0},
                            {       0,250,  0,  0,  0,  0},
                            {       0,250,  0,  0,250,  0},
                            {       0,250,  0,  0,250,  0},
                            {       0,  0,250,250,  0,  0} },

                    {       {   0,250,250,250,  0,  0},
                            {       0,250,  0,  0,250,  0},
                            {       0,250,  0,  0,250,  0},
                            {       0,250,  0,  0,250,  0},
                            {       0,250,  0,  0,250,  0},
                            {       0,250,  0,  0,250,  0},
                            {       0,250,250,250,  0,  0} },

                    {       {   0,250,250,250,250,  0},
                            {       0,250,  0,  0,  0,  0},
                            {       0,250,  0,  0,  0,  0},
                            {       0,250,250,250,250,  0},
                            {       0,250,  0,  0,  0,  0},
                            {       0,250,  0,  0,  0,  0},
                            {       0,250,  0,  0,  0,  0},
                            {       0,250,250,250,250,  0} },

                    {       {   0,250,250,250,250,  0},
                            {       0,250,  0,  0,  0,  0},
                            {       0,250,  0,  0,  0,  0},
                            {       0,250,250,250,250,  0},
                            {       0,250,  0,  0,  0,  0},
                            {       0,250,  0,  0,  0,  0},
                            {       0,250,  0,  0,  0,  0},
                            {       0,250,  0,  0,  0,  0} },

FILE=5x8.inc    Thu Apr 21 12:07:05 1994    PAGE=1
```

```
56:                 {       {   0,  0,250,250,  0,  0},
58:                         {       0,250,  0,  0,250,  0},
59:                         {       0,250,  0,  0,250,  0},
60:                         {       0,250,  0,  0,  0,  0},
61:                         {       0,250,  0,250,250,  0},
62:                         {       0,250,  0,  0,250,  0},
63:                         {       0,250,  0,  0,250,  0},
64:                         {       0,  0,250,250,  0,  0} },
65:
66:                 {       {   0,250,  0,  0,250,  0},
67:                         {       0,250,  0,  0,250,  0},
68:                         {       0,250,  0,  0,250,  0},
69:                         {       0,250,250,250,250,  0},
70:                         {       0,250,  0,  0,250,  0},
71:                         {       0,250,  0,  0,250,  0},
72:                         {       0,250,  0,  0,250,  0},
73:                         {       0,250,  0,  0,250,  0} },
74:
75:                 {       {   0,  0,  0,250,  0,  0},
76:                         {       0,  0,  0,250,  0,  0},
77:                         {       0,  0,  0,250,  0,  0},
78:                         {       0,  0,  0,250,  0,  0},
79:                         {       0,  0,  0,250,  0,  0},
80:                         {       0,  0,  0,250,  0,  0},
81:                         {       0,  0,  0,250,  0,  0},
82:                         {       0,  0,  0,250,  0,  0} },
83:
84:                 {       {   0,  0,  0,  0,250,  0},
85:                         {       0,  0,  0,  0,250,  0},
86:                         {       0,  0,  0,  0,250,  0},
87:                         {       0,  0,  0,  0,250,  0},
88:                         {       0,  0,  0,  0,250,  0},
89:                         {       0,250,  0,  0,250,  0},
90:                         {       0,250,  0,  0,250,  0},
91:                         {       0,  0,250,250,  0,  0} },
92:
93:                 {       {   0,250,  0,  0,250,  0},
94:                         {       0,250,  0,  0,250,  0},
95:                         {       0,250,  0,250,  0,  0},
96:                         {       0,250,250,  0,  0,  0},
97:                         {       0,250,250,  0,  0,  0},
98:                         {       0,250,  0,250,  0,  0},
99:                         {       0,250,  0,  0,250,  0},
100:                        {       0,250,  0,  0,250,  0} },
101:
102:                {       {   0,250,  0,  0,  0,  0},
103:                        {       0,250,  0,  0,  0,  0},
104:                        {       0,250,  0,  0,  0,  0},
105:                        {       0,250,  0,  0,  0,  0},
106:                        {       0,250,  0,  0,  0,  0},
107:                        {       0,250,  0,  0,  0,  0},
108:                        {       0,250,  0,  0,250,  0},
109:                        {       0,250,250,250,250,  0} },
110:
111:                {       {   0,250,  0,  0,  0,250},
```

```
112:                        {       0,250,250,  0,250,250},
114:                        {       0,250,250,  0,250,250},
115:                        {       0,250,  0,250,  0,250},
116:                        {       0,250,  0,  0,  0,250},
117:                        {       0,250,  0,  0,  0,250},
118:                        {       0,250,  0,  0,  0,250},
119:                        {       0,250,  0,  0,  0,250} },
120:
121:                  {     {   0,250,  0,  0,250,  0},
122:                        {       0,250,250,  0,250,  0},
123:                        {       0,250,250,  0,250,  0},
124:                        {       0,250,250,  0,250,  0},
125:                        {       0,250,  0,250,250,  0},
126:                        {       0,250,  0,250,250,  0},
127:                        {       0,250,  0,250,250,  0},
128:                        {       0,250,  0,  0,250,  0} },
129:
130:                  {     {   0,  0,250,250,  0,  0},
131:                        {       0,250,  0,  0,250,  0},
132:                        {       0,250,  0,  0,250,  0},
133:                        {       0,250,  0,  0,250,  0},
134:                        {       0,250,  0,  0,250,  0},
135:                        {       0,250,  0,  0,250,  0},
136:                        {       0,250,  0,  0,250,  0},
137:                        {       0,  0,250,250,  0,  0} },
138:
139:                  {     {   0,250,250,250,  0,  0},
140:                        {       0,250,  0,  0,250,  0},
141:                        {       0,250,  0,  0,250,  0},
142:                        {       0,250,  0,  0,250,  0},
143:                        {       0,250,  0,  0,250,  0},
144:                        {       0,250,250,250,  0,  0},
145:                        {       0,250,  0,  0,  0,  0},
146:                        {       0,250,  0,  0,  0,  0} },
147:
148:                  {     {   0,  0,250,250,  0,  0},
149:                        {       0,250,  0,  0,250,  0},
150:                        {       0,250,  0,  0,250,  0},
151:                        {       0,250,  0,  0,250,  0},
152:                        {       0,250,  0,  0,250,  0},
153:                        {       0,250,250,250,250,  0},
154:                        {       0,250,  0,250,250,  0},
155:                        {       0,  0,250,250,  0,250} },
156:
157:                  {     {   0,250,250,250,  0,  0},
158:                        {       0,250,  0,  0,250,  0},
159:                        {       0,250,  0,  0,250,  0},
160:                        {       0,250,  0,  0,250,  0},
161:                        {       0,250,  0,  0,250,  0},
162:                        {       0,250,250,250,  0,  0},
163:                        {       0,250,  0,250,  0,  0},
164:                        {       0,250,  0,  0,250,  0} },
165:
166:                  {     {   0,  0,250,250,250,  0},
167:                        {       0,250,  0,  0,  0,  0},
```

```
168:                        {       0,250,  0,  0,  0,  0},
170:                        {       0,  0,250,250,  0,  0},
171:                        {       0,  0,  0,  0,250,  0},
172:                        {       0,  0,  0,  0,250,  0},
173:                        {       0,  0,  0,  0,250,  0},
174:                        {       0,250,250,250,  0,  0} },
175:
176:                  {     {   0,250,250,250,250,250},
177:                        {       0,  0,  0,250,  0,  0},
178:                        {       0,  0,  0,250,  0,  0},
179:                        {       0,  0,  0,250,  0,  0},
180:                        {       0,  0,  0,250,  0,  0},
181:                        {       0,  0,  0,250,  0,  0},
182:                        {       0,  0,  0,250,  0,  0},
183:                        {       0,  0,  0,250,  0,  0} },
184:
185:                  {     {   0,250,  0,  0,250,  0},
186:                        {       0,250,  0,  0,250,  0},
187:                        {       0,250,  0,  0,250,  0},
188:                        {       0,250,  0,  0,250,  0},
189:                        {       0,250,  0,  0,250,  0},
190:                        {       0,250,  0,  0,250,  0},
191:                        {       0,250,  0,  0,250,  0},
192:                        {       0,  0,250,250,  0,  0} },
193:
194:                  {     {   0,250,  0,  0,  0,250},
195:                        {       0,250,  0,  0,  0,250},
196:                        {       0,250,  0,  0,  0,250},
197:                        {       0,250,  0,  0,  0,250},
198:                        {       0,250,  0,  0,  0,250},
199:                        {       0,250,  0,  0,  0,250},
200:                        {       0,  0,250,  0,250,  0},
201:                        {       0,  0,  0,250,  0,  0} },
202:
203:                  {     { 250,  0,  0,  0,  0,250},
204:                        { 250,  0,  0,  0,  0,250},
205:                        { 250,  0,  0,  0,  0,250},
206:                        { 250,  0,  0,  0,  0,250},
207:                        { 250,  0,  0,  0,  0,250},
208:                        { 250,  0,250,250,  0,250},
209:                        { 250,  0,250,250,  0,250},
210:                        {       0,250,  0,  0,250,  0} },
211:
212:                  {     {   0,250,  0,  0,  0,250},
213:                        {       0,250,  0,  0,  0,250},
214:                        {       0,250,  0,  0,  0,250},
215:                        {       0,  0,250,  0,250,250},
216:                        {       0,  0,  0,250,  0,  0},
217:                        {       0,  0,250,  0,250,  0},
218:                        {       0,250,  0,  0,  0,250},
219:                        {       0,250,  0,  0,  0,250} },
220:
221:                  {     {   0,250,  0,  0,  0,250},
222:                        {       0,250,  0,  0,  0,250},
223:                        {       0,250,  0,  0,  0,250},
```

```
224:                        {       0,  0,250,  0,250,250},
226:                        {       0,  0,  0,250,  0,  0},
227:                        {       0,  0,  0,250,  0,  0},
228:                        {       0,  0,  0,250,  0,  0},
229:                        {       0,  0,  0,250,  0,  0} }
230: };
231:
232: int ANUM[17][8][6] = {
233:                    {       {   0,  0,  0,  0,  0,  0},
234:                            {       0,  0,  0,  0,  0,  0},
235:                            {       0,  0,  0,  0,  0,  0},
236:                            {       0,  0,  0,  0,  0,  0},
237:                            {       0,  0,250,250,  0,  0},
238:                            {       0,  0,250,250,  0,  0},
239:                            {       0,  0,  0,250,  0,  0},
240:                            {       0,  0,250,  0,  0,  0} },
241:
242:                    {       {   0,  0,  0,  0,  0,  0},
243:                            {       0,  0,  0,  0,  0,  0},
244:                            {       0,  0,  0,  0,  0,  0},
245:                            {       0,  0,  0,  0,  0,  0},
246:                            {       0,  0,  0,  0,  0,  0},
247:                            {       0,  0,  0,  0,  0,  0},
248:                            {       0,  0,250,250,  0,  0},
249:                            {       0,  0,250,250,  0,  0} },
250:
251:                    {       {   0,  0,250,250,  0,  0},
252:                            {       0,  0,250,250,  0,  0},
253:                            {       0,  0,  0,  0,  0,  0},
254:                            {       0,  0,  0,  0,  0,  0},
255:                            {       0,  0,  0,  0,  0,  0},
256:                            {       0,  0,  0,  0,  0,  0},
257:                            {       0,  0,250,250,  0,  0},
258:                            {       0,  0,250,250,  0,  0} },
259:
260:                    {       {   0,  0,  0,  0,  0,  0},
261:                            {       0,  0,  0,  0,  0,  0},
262:                            {       0,  0,  0,  0,  0,  0},
263:                            {       0,  0,  0,  0,  0,  0},
264:                            {       0,250,250,250,250,  0},
265:                            {       0,  0,  0,  0,  0,  0},
266:                            {       0,  0,  0,  0,  0,  0},
267:                            {       0,  0,  0,  0,  0,  0} },
268:
269:                    {       {   0,250,  0,  0,  0,  0},
270:                            {       0,250,  0,  0,  0,  0},
271:                            {       0,  0,250,  0,  0,  0},
272:                            {       0,  0,250,  0,  0,  0},
273:                            {       0,  0,  0,250,  0,  0},
274:                            {       0,  0,  0,250,  0,  0},
275:                            {       0,  0,  0,  0,250,  0},
276:                            {       0,  0,  0,  0,250,  0} },
277:
278:                    {       { 250,250,  0,  0,  0,250},
279:                            { 250,250,  0,  0,250,  0},
```

```
                        {       0,  0,  0,250,250,  0},
                        {       0,  0,  0,250,  0,  0},
                        {       0,  0,250,  0,  0,  0},
                        {       0,  0,250,  0,  0,  0},
                        {   0,250,  0,  0,250,250},
                        { 250,250,  0,  0,250,250} },

                {       {   0,  0,  0,  0,  0,  0},
                        {       0,  0,  0,  0,  0,  0},
                        {       0,  0,  0,  0,  0,  0},
                        {       0,  0,  0,  0,  0,  0},
                        {       0,  0,  0,  0,  0,  0},
                        {       0,  0,  0,  0,  0,  0},
                        {       0,  0,  0,  0,  0,  0} },

                {       {   0,  0,  0,250,250,  0},
                        {       0,  0,  0,250,250,  0},
                        {       0,  0,250,  0,250,  0},
                        {       0,  0,250,  0,250,  0},
                        {       0,  0,  0,  0,250,  0},
                        {       0,  0,  0,  0,250,  0},
                        {       0,  0,  0,  0,250,  0} },

                {       {   0,  0,250,250,  0,  0},
                        {       0,250,  0,  0,250,  0},
                        {       0,250,  0,  0,250,  0},
                        {       0,  0,  0,250,  0,  0},
                        {       0,  0,  0,250,  0,  0},
                        {       0,  0,250,  0,  0,  0},
                        {       0,250,  0,  0,  0,  0},
                        {       0,250,250,250,250,  0} },

                {       {   0,  0,250,250,  0,  0},
                        {       0,250,  0,  0,250,  0},
                        {       0,250,  0,  0,250,  0},
                        {       0,  0,  0,250,  0,  0},
                        {       0,  0,  0,250,  0,  0},
                        {       0,250,  0,  0,250,  0},
                        {       0,250,  0,  0,250,  0},
                        {       0,  0,250,250,  0,  0} },

                {       {   0,  0,  0,250,250,  0},
                        {       0,  0,250,  0,250,  0},
                        {       0,  0,250,  0,250,  0},
                        {       0,250,  0,  0,250,  0},
                        {       0,250,  0,  0,250,  0},
                        {       0,250,250,250,250,  0},
                        {       0,  0,  0,  0,250,  0},
                        {       0,  0,  0,  0,250,  0} },

                {       {   0,250,250,250,250,  0},
                        {       0,250,  0,  0,  0,  0},
                        {       0,250,250,250,  0,  0},
```

```
336:                        {       0,  0,  0,  0,250,  0},
338:                        {       0,  0,  0,  0,250,  0},
339:                        {       0,250,  0,  0,250,  0},
340:                        {       0,250,  0,  0,250,  0},
341:                        {       0,  0,250,250,  0,  0} },
342:
343:                  {     {   0,  0,250,250,  0,  0},
344:                        {       0,250,  0,  0,250,  0},
345:                        {       0,250,  0,  0,  0,  0},
346:                        {       0,250,250,250,  0,  0},
347:                        {       0,250,  0,  0,250,  0},
348:                        {       0,250,  0,  0,250,  0},
349:                        {       0,250,  0,  0,250,  0},
350:                        {       0,  0,250,250,  0,  0} },
351:
352:                  {     {   0,250,250,250,250,  0},
353:                        {       0,250,  0,  0,250,  0},
354:                        {       0,250,  0,  0,250,  0},
355:                        {       0,  0,  0,250,  0,  0},
356:                        {       0,  0,  0,250,  0,  0},
357:                        {       0,  0,  0,250,  0,  0},
358:                        {       0,  0,250,  0,  0,  0},
359:                        {       0,  0,250,  0,  0,  0} },
360:
361:                  {     {   0,  0,250,250,  0,  0},
362:                        {       0,250,  0,  0,250,  0},
363:                        {       0,250,  0,  0,250,  0},
364:                        {       0,  0,250,250,  0,  0},
365:                        {       0,250,  0,  0,250,  0},
366:                        {       0,250,  0,  0,250,  0},
367:                        {       0,250,  0,  0,250,  0},
368:                        {       0,  0,250,250,  0,  0} },
369:
370:                  {     {   0,  0,250,250,  0,  0},
371:                        {       0,250,  0,  0,250,  0},
372:                        {       0,250,  0,  0,250,  0},
373:                        {       0,  0,250,250,250,  0},
374:                        {       0,  0,  0,  0,250,  0},
375:                        {       0,  0,  0,  0,250,  0},
376:                        {       0,250,  0,  0,250,  0},
377:                        {       0,  0,250,250,  0,  0} },
378:
379:                  {     {   0,  0,250,250,  0,  0},
380:                        {       0,250,  0,  0,250,  0},
381:                        {       0,250,  0,250,250,  0},
382:                        {       0,250,  0,250,250,  0},
383:                        {       0,250,250,  0,250,  0},
384:                        {       0,250,250,  0,250,  0},
385:                        {       0,250,  0,  0,250,  0},
386:                        {       0,  0,250,250,  0,  0} }
387: };
```

```
 1: /*              Program:        p2.c
 2:                 Created by:     Chris Voltz
 3:                 Date Created:   Unknown
 4:                 Modified by:    Craig A. Vrana - UDRI
 5:                 Last Modified:  07-DEC-89
 6:
 7:                 Modified by   : Alex Firdman
 8:                 Last Modified : 07-22-93
 9:                 Compiler used : NDP-C, Ver. 4.2.1
10:
11:         Modifications are made to port this code to NDP-C V.4.2.1
12:         and to improve the quality of the printed image.
13:
14:             This program recieves as input, from the DOS command line, the
15:         filename of an image, without the .IMG extension, presumed to be in
16:         the PCVision IMAGEACTION format and prints it on an HP Laserjet+.
17:         However, it only uses a limited number of shades.  This version has
18:         been altered to directly display the density patterns that are listed
19:         below instead of converting them to a triangle.  A smoother gray
20:         shade results but the shades tend to be darker than those of the
21:         original version of the program.
22:
23:         Modification was made to the format reading routine. This utility
24:         program will read binary image files; the image files with the header
25:         stripped from the IMG image file. It is done to print IBG (image binary
26:         graphic) files on laser printer. The IBG files are processed using
27:         Hermite, Wavelet and other image transformation techniques.
28:
29: */
30:
31: #include <stdio.h>
32: #include <stdlib.h>
33: #include <os.h>
34: #include <string.h>
35:
36: #define BLACK         0
37: #define BLUE          1
38: #define GREEN         2
39: #define CYAN          3
40: #define RED           4
41: #define MAGENTA       5
42: #define BROWN         6
43: #define LIGHTGRAY     7    /* Color Definition */
44: #define DARKGRAY      8
45: #define LIGHTBLUE     9
46: #define LIGHTGREEN    10
47: #define LIGHTCYAN     11
48: #define LIGHTRED      12
49: #define LIGHTMAGENTA  13
50: #define YELLOW        14
51: #define WHITE         15
52:
53: #define OFF      0
54: #define ON_NORM  2
55: #define ON_BLOCK 1
```

```
 56:
 58: #define MAX_IMAGE_SIZE  256
 59: #define PIN_SIZE        8
 60:
 61: void error_message(int,int,int,int,char*);
 62: void cursoronof(int);
 63: void Wrt_Str(int,int,char*,int,int);
 64: void put_window(int,int,int,int,int,int,char*,int,int);
 65:
 66: int prnt_ima(img_name, img_size)
 67: char img_name[13];
 68: int img_size;
 69: {
 70: unsigned char   ch_h[256][PIN_SIZE] = {
 71:       {  0,   0,   0,   0,   0,   0,   0,   0},
 72:       {  0,   0,   0,   0,   0,   0,   0,   0},
 73:       {  0,   0,   0,   0,   0,   0,   0,   0},
 74:       {  0,   0,   0,   0,   0,   0,   0,   0},
 75:       {  1,   0,   0,   0,   0,   0,   0,   0},
 76:       {  1,   0,   0,   0,   0,   0,   0,   0},
 77:       {  1,   0,   0,   0,   0,   0,   0,   0},
 78:       {  1,   0,   0,   0,   0,   0,   0,   0},
 79:       {  7,   0,   0,   0,   0,   0,   0,   0},
 80:       {  7,   0,   0,   0,   0,   0,   0,   0},
 81:       {  7,   0,   0,   0,   0,   0,   0,   0},
 82:       {  7,   0,   0,   0,   0,   0,   0,   0},
 83:       { 15,   0,   0,   0,   0,   0,   0,   0},
 84:       { 15,   0,   0,   0,   0,   0,   0,   0},
 85:       { 15,   0,   0,   0,   0,   0,   0,   0},
 86:       { 15,   0,   0,   0,   0,   0,   0,   0},
 87:       { 31,   0,   0,   0,   0,   0,   0,   0},
 88:       { 31,   0,   0,   0,   0,   0,   0,   0},
 89:       { 31,   0,   0,   0,   0,   0,   0,   0},
 90:       { 31,   0,   0,   0,   0,   0,   0,   0},
 91:       { 63,   0,   0,   0,   0,   0,   0,   0},
 92:       { 63,   0,   0,   0,   0,   0,   0,   0},
 93:       { 63,   0,   0,   0,   0,   0,   0,   0},
 94:       { 63,   0,   0,   0,   0,   0,   0,   0},
 95:       {127,   0,   0,   0,   0,   0,   0,   0},
 96:       {127,   0,   0,   0,   0,   0,   0,   0},
 97:       {127,   0,   0,   0,   0,   0,   0,   0},
 98:       {127,   0,   0,   0,   0,   0,   0,   0},
 99:       {255,   0,   0,   0,   0,   0,   0,   0},
100:       {255,   0,   0,   0,   0,   0,   0,   0},
101:       {255,   0,   0,   0,   0,   0,   0,   0},
102:       {255,   0,   0,   0,   0,   0,   0,   0},
103:       {255,   1,   0,   0,   0,   0,   0,   0},
104:       {255,   1,   0,   0,   0,   0,   0,   0},
105:       {255,   1,   0,   0,   0,   0,   0,   0},
106:       {255,   1,   0,   0,   0,   0,   0,   0},
107:       {255,   3,   0,   0,   0,   0,   0,   0},
108:       {255,   3,   0,   0,   0,   0,   0,   0},
109:       {255,   3,   0,   0,   0,   0,   0,   0},
110:       {255,   3,   0,   0,   0,   0,   0,   0},
111:       {255,   7,   0,   0,   0,   0,   0,   0},
```

```
112:      {255,   7,   0,   0,   0,   0,   0,   0},
114:      {255,   7,   0,   0,   0,   0,   0,   0},
115:      {255,   7,   0,   0,   0,   0,   0,   0},
116:      {255,  15,   0,   0,   0,   0,   0,   0},
117:      {255,  15,   0,   0,   0,   0,   0,   0},
118:      {255,  15,   0,   0,   0,   0,   0,   0},
119:      {255,  15,   0,   0,   0,   0,   0,   0},
120:      {255,  31,   0,   0,   0,   0,   0,   0},
121:      {255,  31,   0,   0,   0,   0,   0,   0},
122:      {255,  31,   0,   0,   0,   0,   0,   0},
123:      {255,  31,   0,   0,   0,   0,   0,   0},
124:      {255,  63,   0,   0,   0,   0,   0,   0},
125:      {255,  63,   0,   0,   0,   0,   0,   0},
126:      {255,  63,   0,   0,   0,   0,   0,   0},
127:      {255,  63,   0,   0,   0,   0,   0,   0},
128:      {255, 127,   0,   0,   0,   0,   0,   0},
129:      {255, 127,   0,   0,   0,   0,   0,   0},
130:      {255, 127,   0,   0,   0,   0,   0,   0},
131:      {255, 127,   0,   0,   0,   0,   0,   0},
132:      {255, 255,   0,   0,   0,   0,   0,   0},
133:      {255, 255,   0,   0,   0,   0,   0,   0},
134:      {255, 255,   0,   0,   0,   0,   0,   0},
135:      {255, 255,   0,   0,   0,   0,   0,   0},
136:      {255, 255,   1,   0,   0,   0,   0,   0},
137:      {255, 255,   1,   0,   0,   0,   0,   0},
138:      {255, 255,   1,   0,   0,   0,   0,   0},
139:      {255, 255,   1,   0,   0,   0,   0,   0},
140:      {255, 255,   3,   0,   0,   0,   0,   0},
141:      {255, 255,   3,   0,   0,   0,   0,   0},
142:      {255, 255,   3,   0,   0,   0,   0,   0},
143:      {255, 255,   3,   0,   0,   0,   0,   0},
144:      {255, 255,   7,   0,   0,   0,   0,   0},
145:      {255, 255,   7,   0,   0,   0,   0,   0},
146:      {255, 255,   7,   0,   0,   0,   0,   0},
147:      {255, 255,   7,   0,   0,   0,   0,   0},
148:      {255, 255,  15,   0,   0,   0,   0,   0},
149:      {255, 255,  15,   0,   0,   0,   0,   0},
150:      {255, 255,  15,   0,   0,   0,   0,   0},
151:      {255, 255,  15,   0,   0,   0,   0,   0},
152:      {255, 255,  31,   0,   0,   0,   0,   0},
153:      {255, 255,  31,   0,   0,   0,   0,   0},
154:      {255, 255,  31,   0,   0,   0,   0,   0},
155:      {255, 255,  31,   0,   0,   0,   0,   0},
156:      {255, 255,  63,   0,   0,   0,   0,   0},
157:      {255, 255,  63,   0,   0,   0,   0,   0},
158:      {255, 255,  63,   0,   0,   0,   0,   0},
159:      {255, 255,  63,   0,   0,   0,   0,   0},
160:      {255, 255, 127,   0,   0,   0,   0,   0},
161:      {255, 255, 127,   0,   0,   0,   0,   0},
162:      {255, 255, 127,   0,   0,   0,   0,   0},
163:      {255, 255, 127,   0,   0,   0,   0,   0},
164:      {255, 255, 255,   0,   0,   0,   0,   0},
165:      {255, 255, 255,   0,   0,   0,   0,   0},
166:      {255, 255, 255,   0,   0,   0,   0,   0},
167:      {255, 255, 255,   0,   0,   0,   0,   0},
```

```
168:      {255, 255, 255,   1,   0,   0,   0,   0},
170:      {255, 255, 255,   1,   0,   0,   0,   0},
171:      {255, 255, 255,   1,   0,   0,   0,   0},
172:      {255, 255, 255,   1,   0,   0,   0,   0},
173:      {255, 255, 255,   3,   0,   0,   0,   0},
174:      {255, 255, 255,   3,   0,   0,   0,   0},
175:      {255, 255, 255,   3,   0,   0,   0,   0},
176:      {255, 255, 255,   3,   0,   0,   0,   0},
177:      {255, 255, 255,   7,   0,   0,   0,   0},
178:      {255, 255, 255,   7,   0,   0,   0,   0},
179:      {255, 255, 255,   7,   0,   0,   0,   0},
180:      {255, 255, 255,   7,   0,   0,   0,   0},
181:      {255, 255, 255,  15,   0,   0,   0,   0},
182:      {255, 255, 255,  15,   0,   0,   0,   0},
183:      {255, 255, 255,  15,   0,   0,   0,   0},
184:      {255, 255, 255,  15,   0,   0,   0,   0},
185:      {255, 255, 255,  31,   0,   0,   0,   0},
186:      {255, 255, 255,  31,   0,   0,   0,   0},
187:      {255, 255, 255,  31,   0,   0,   0,   0},
188:      {255, 255, 255,  31,   0,   0,   0,   0},
189:      {255, 255, 255,  63,   0,   0,   0,   0},
190:      {255, 255, 255,  63,   0,   0,   0,   0},
191:      {255, 255, 255,  63,   0,   0,   0,   0},
192:      {255, 255, 255,  63,   0,   0,   0,   0},
193:      {255, 255, 255, 127,   0,   0,   0,   0},
194:      {255, 255, 255, 127,   0,   0,   0,   0},
195:      {255, 255, 255, 127,   0,   0,   0,   0},
196:      {255, 255, 255, 127,   0,   0,   0,   0},
197:      {255, 255, 255, 255,   0,   0,   0,   0},
198:      {255, 255, 255, 255,   0,   0,   0,   0},
199:      {255, 255, 255, 255,   0,   0,   0,   0},
200:      {255, 255, 255, 255,   0,   0,   0,   0},
201:      {255, 255, 255, 255,   1,   0,   0,   0},
202:      {255, 255, 255, 255,   1,   0,   0,   0},
203:      {255, 255, 255, 255,   1,   0,   0,   0},
204:      {255, 255, 255, 255,   1,   0,   0,   0},
205:      {255, 255, 255, 255,   3,   0,   0,   0},
206:      {255, 255, 255, 255,   3,   0,   0,   0},
207:      {255, 255, 255, 255,   3,   0,   0,   0},
208:      {255, 255, 255, 255,   3,   0,   0,   0},
209:      {255, 255, 255, 255,   7,   0,   0,   0},
210:      {255, 255, 255, 255,   7,   0,   0,   0},
211:      {255, 255, 255, 255,   7,   0,   0,   0},
212:      {255, 255, 255, 255,   7,   0,   0,   0},
213:      {255, 255, 255, 255,  15,   0,   0,   0},
214:      {255, 255, 255, 255,  15,   0,   0,   0},
215:      {255, 255, 255, 255,  15,   0,   0,   0},
216:      {255, 255, 255, 255,  15,   0,   0,   0},
217:      {255, 255, 255, 255,  31,   0,   0,   0},
218:      {255, 255, 255, 255,  31,   0,   0,   0},
219:      {255, 255, 255, 255,  31,   0,   0,   0},
220:      {255, 255, 255, 255,  31,   0,   0,   0},
221:      {255, 255, 255, 255,  63,   0,   0,   0},
222:      {255, 255, 255, 255,  63,   0,   0,   0},
223:      {255, 255, 255, 255,  63,   0,   0,   0},
```

```
224:      {255, 255, 255, 255,  63,   0,   0,   0},
226:      {255, 255, 255, 255, 127,   0,   0,   0},
227:      {255, 255, 255, 255, 127,   0,   0,   0},
228:      {255, 255, 255, 255, 127,   0,   0,   0},
229:      {255, 255, 255, 255, 127,   0,   0,   0},
230:      {255, 255, 255, 255, 255,   0,   0,   0},
231:      {255, 255, 255, 255, 255,   0,   0,   0},
232:      {255, 255, 255, 255, 255,   0,   0,   0},
233:      {255, 255, 255, 255, 255,   0,   0,   0},
234:      {255, 255, 255, 255, 255,   1,   0,   0},
235:      {255, 255, 255, 255, 255,   1,   0,   0},
236:      {255, 255, 255, 255, 255,   1,   0,   0},
237:      {255, 255, 255, 255, 255,   1,   0,   0},
238:      {255, 255, 255, 255, 255,   3,   0,   0},
239:      {255, 255, 255, 255, 255,   3,   0,   0},
240:      {255, 255, 255, 255, 255,   3,   0,   0},
241:      {255, 255, 255, 255, 255,   3,   0,   0},
242:      {255, 255, 255, 255, 255,   7,   0,   0},
243:      {255, 255, 255, 255, 255,   7,   0,   0},
244:      {255, 255, 255, 255, 255,   7,   0,   0},
245:      {255, 255, 255, 255, 255,   7,   0,   0},
246:      {255, 255, 255, 255, 255,  15,   0,   0},
247:      {255, 255, 255, 255, 255,  15,   0,   0},
248:      {255, 255, 255, 255, 255,  15,   0,   0},
249:      {255, 255, 255, 255, 255,  15,   0,   0},
250:      {255, 255, 255, 255, 255,  31,   0,   0},
251:      {255, 255, 255, 255, 255,  31,   0,   0},
252:      {255, 255, 255, 255, 255,  31,   0,   0},
253:      {255, 255, 255, 255, 255,  31,   0,   0},
254:      {255, 255, 255, 255, 255,  63,   0,   0},
255:      {255, 255, 255, 255, 255,  63,   0,   0},
256:      {255, 255, 255, 255, 255,  63,   0,   0},
257:      {255, 255, 255, 255, 255,  63,   0,   0},
258:      {255, 255, 255, 255, 255, 127,   0,   0},
259:      {255, 255, 255, 255, 255, 127,   0,   0},
260:      {255, 255, 255, 255, 255, 127,   0,   0},
261:      {255, 255, 255, 255, 255, 127,   0,   0},
262:      {255, 255, 255, 255, 255, 255,   0,   0},
263:      {255, 255, 255, 255, 255, 255,   0,   0},
264:      {255, 255, 255, 255, 255, 255,   0,   0},
265:      {255, 255, 255, 255, 255, 255,   0,   0},
266:      {255, 255, 255, 255, 255, 255,   1,   0},
267:      {255, 255, 255, 255, 255, 255,   1,   0},
268:      {255, 255, 255, 255, 255, 255,   1,   0},
269:      {255, 255, 255, 255, 255, 255,   1,   0},
270:      {255, 255, 255, 255, 255, 255,   3,   0},
271:      {255, 255, 255, 255, 255, 255,   3,   0},
272:      {255, 255, 255, 255, 255, 255,   3,   0},
273:      {255, 255, 255, 255, 255, 255,   3,   0},
274:      {255, 255, 255, 255, 255, 255,   7,   0},
275:      {255, 255, 255, 255, 255, 255,   7,   0},
276:      {255, 255, 255, 255, 255, 255,   7,   0},
277:      {255, 255, 255, 255, 255, 255,   7,   0},
278:      {255, 255, 255, 255, 255, 255,  15,   0},
279:      {255, 255, 255, 255, 255, 255,  15,   0},
```

```
280:      {255, 255, 255, 255, 255, 255,  15,   0},
282:      {255, 255, 255, 255, 255, 255,  15,   0},
283:      {255, 255, 255, 255, 255, 255,  31,   0},
284:      {255, 255, 255, 255, 255, 255,  31,   0},
285:      {255, 255, 255, 255, 255, 255,  31,   0},
286:      {255, 255, 255, 255, 255, 255,  31,   0},
287:      {255, 255, 255, 255, 255, 255,  63,   0},
288:      {255, 255, 255, 255, 255, 255,  63,   0},
289:      {255, 255, 255, 255, 255, 255,  63,   0},
290:      {255, 255, 255, 255, 255, 255,  63,   0},
291:      {255, 255, 255, 255, 255, 255, 127,   0},
292:      {255, 255, 255, 255, 255, 255, 127,   0},
293:      {255, 255, 255, 255, 255, 255, 127,   0},
294:      {255, 255, 255, 255, 255, 255, 127,   0},
295:      {255, 255, 255, 255, 255, 255, 255,   0},
296:      {255, 255, 255, 255, 255, 255, 255,   0},
297:      {255, 255, 255, 255, 255, 255, 255,   0},
298:      {255, 255, 255, 255, 255, 255, 255,   0},
299:      {255, 255, 255, 255, 255, 255, 255,   1},
300:      {255, 255, 255, 255, 255, 255, 255,   1},
301:      {255, 255, 255, 255, 255, 255, 255,   1},
302:      {255, 255, 255, 255, 255, 255, 255,   1},
303:      {255, 255, 255, 255, 255, 255, 255,   3}, /*6*/
304:      {255, 255, 255, 255, 255, 255, 255,   3},
305:      {255, 255, 255, 255, 255, 255, 255,   3},
306:      {255, 255, 255, 255, 255, 255, 255,   3},
307:      {255, 255, 255, 255, 255, 255, 255,   7}, /*5*/
308:      {255, 255, 255, 255, 255, 255, 255,   7},
309:      {255, 255, 255, 255, 255, 255, 255,   7},
310:      {255, 255, 255, 255, 255, 255, 255,   7},
311:      {255, 255, 255, 255, 255, 255, 255,  15}, /*4*/
312:      {255, 255, 255, 255, 255, 255, 255,  15},
313:      {255, 255, 255, 255, 255, 255, 255,  15},
314:      {255, 255, 255, 255, 255, 255, 255,  15},
315:      {255, 255, 255, 255, 255, 255, 255,  31}, /*3*/
316:      {255, 255, 255, 255, 255, 255, 255,  31},
317:      {255, 255, 255, 255, 255, 255, 255,  31},
318:      {255, 255, 255, 255, 255, 255, 255,  31},
319:      {255, 255, 255, 255, 255, 255, 255,  63}, /*2*/
320:      {255, 255, 255, 255, 255, 255, 255,  63},
321:      {255, 255, 255, 255, 255, 255, 255,  63},
322:      {255, 255, 255, 255, 255, 255, 255,  63},
323:      {255, 255, 255, 255, 255, 255, 255, 127},
324:      {255, 255, 255, 255, 255, 255, 255, 127},
325:      {255, 255, 255, 255, 255, 255, 255, 127},
326:      {255, 255, 255, 255, 255, 255, 255, 127},
327:      {255, 255, 255, 255, 255, 255, 255, 255},
328:      {255, 255, 255, 255, 255, 255, 255, 255},
329:      {255, 255, 255, 255, 255, 255, 255, 255},
330:      {255, 255, 255, 255, 255, 255, 255, 255}, /*1*/
331: };     /* ch_h initializer */
332:
333: static  unsigned char   ch_v[256][PIN_SIZE];
334:     FILE            *file_in, *file_out;
335:
```

```
336: static unsigned char   in_buffer[MAX_IMAGE_SIZE];
338:     unsigned int    index;
339:     unsigned int    index_2;
340:     unsigned int    index_3;
341:
342: static unsigned char   out_buffer[MAX_IMAGE_SIZE];
343:     unsigned char   val;
344:     int             x_size=0;
345:     int             y_size=0;
346:
347:
348: x_size = img_size;
349: y_size = x_size;
350:
351: /*  strcat(img_name, ".IMG");  */
352:
353:         /* open image file */
354:     file_in=fopen(img_name, "rb");
355:
356:     file_out=fopen("prn", "wb");
357:
358:         /* check if image too big */
359:
360:     if (x_size>MAX_IMAGE_SIZE)
361:     {
362:         error_message(30,12,30,5," Image Size is too big..");
363:         exit(T);
364:     }
365:
366:     cursoronof(OFF);
367:     put_window(28,10,31,5,0x4f,2,"",BLUE,WHITE);
368:     Wrt_Str(30,12," Please, Wait. Printing... ",RED,WHITE);
369:
370:         /* calculate vertical pin spots */
371:     for (index=0; index<256; index++)
372:     {
373:         for (index_2=0; index_2<PIN_SIZE; index_2++)
374:         {
375:             ch_v[index][index_2] = 0;
376:             for (val=0; val<PIN_SIZE; val++)
377:             {
378:               if ((index_2-val) < 0)
379:                 ch_v[index][index_2] += (ch_h[index][val] & (0x80 >> index_2))
380:                                         >> -(index_2-val);
381:               else
382:                 ch_v[index][index_2] += (ch_h[index][val] & (0x80 >> index_2))
383:                                         << (index_2-val);
384:             }   /* for each pin */
385:         }   /* for each column */
386:     }   /* for each intensity */
387:
388:         /* initialize Laserjet */
389:
390:     fprintf(file_out, "%cE", ESC);      /* reset printer           */
391:     fprintf(file_out, "%c*t300R", ESC); /* set 300 dpi graphics    */

FILE=prnt_ima.c    Thu Apr 21 12:08:13 1994    PAGE=7
```

```
392:     fprintf(file_out, "%c&a6C", ESC);       /* move cursor to column 5  */
394:     fprintf(file_out, "%c*r1A", ESC);       /* start raster graphics    */
395:
396:     fprintf(file_out,"\nImage Name : %s ** Image Size : %d x %d\n",
397:                                              img_name, x_size, y_size);
398:
399:         /* read in image and display */
400:     for (index=0; index<y_size && !feof(file_in); index++) /* 1=y_size */
401:     {
402:         fread(in_buffer, 1, x_size, file_in);       /* get a line of pixels */
403:         for (index_2=0; index_2<PIN_SIZE; index_2++)
404:         {
405:             for (index_3=0; index_3<x_size; index_3++)
406:             {
407:                 out_buffer[index_3] = ch_h[255-in_buffer[index_3]][index_2];
408:             }       /* for each pixel */
409:             fprintf(file_out, "%c*b%1dW", ESC, x_size); /* start transfer   */
410:             fwrite(out_buffer, 1, x_size, file_out);        /* transfer line    */
411:         }   /* for each line of gray scale pattern */
412:     }   /* for each row of pixels */
413:
414:         /* print page and eject */
415:     fprintf(file_out, "%c*rB", ESC);        /* end raster graphics  */
416:     fprintf(file_out, "%cE", ESC);      /* reset printer        */
417:
418: fclose(file_in);
419: fclose(file_out);
420:
421: return 0;
422: }   /* main */
```

```
 1:
 2:
 3: #include <stdio.h>
 4:
 5: void nrerror(error_text)
 6: char error_text[];
 7: {
 8:     void exit();
 9:
10:     fprintf(stderr,"Numerical Recipes run-time error...\n");
11:     fprintf(stderr,"%s\n",error_text);
12:     fprintf(stderr,"...now exiting to system...\n");
13:     exit(1);
14: }
15:
16:
17:
18: unsigned char  *unsigned_vector(nl,nh)
19: int nl,nh;
20: {
21:     unsigned char *v;
22:
23:     v=(unsigned char *)malloc((unsigned) (nh-nl+1)*sizeof(unsigned char));
24:     if (!v) nrerror("allocation failure in vector()");
25:     return v-nl;
26: }
27:
28: float *vector(nl,nh)
29: int nl,nh;
30: {
31:     float *v;
32:
33:     v=(float *)malloc((unsigned) (nh-nl+1)*sizeof(float));
34:     if (!v) nrerror("allocation failure in vector()");
35:     return v-nl;
36: }
37:
38: int *ivector(nl,nh)
39: int nl,nh;
40: {
41:     int *v;
42:
43:     v=(int *)malloc((unsigned) (nh-nl+1)*sizeof(int));
44:     if (!v) nrerror("allocation failure in ivector()");
45:     return v-nl;
46: }
47:
48: double *dvector(nl,nh)
49: int nl,nh;
50: {
51:     double *v;
52:
53:     v=(double *)malloc((unsigned) (nh-nl+1)*sizeof(double));
54:     if (!v) nrerror("allocation failure in dvector()");
55:     return v-nl;
```

FILE=nrutil.c Thu Apr 21 12:11:41 1994 PAGE=1

```
}

short **short_matrix(nrl,nrh,ncl,nch)
int nrl,nrh,ncl,nch;
{
    int i;
        short **m;

    m=(short **) malloc((unsigned) (nrh-nrl+1)*sizeof(short*));
    if (!m) nrerror("allocation failure 1 in matrix()");
    m -= nrl;

    for(i=nrl;i<=nrh;i++) {
        m[i]=(short *) malloc((unsigned) (nch-ncl+1)*sizeof(short));
        if (!m[i]) nrerror("allocation failure 2 in matrix()");
        m[i] -= ncl;
    }
    return m;
}


float **matrix(nrl,nrh,ncl,nch)
int nrl,nrh,ncl,nch;
{
    int i;
    float **m;

    m=(float **) malloc((unsigned) (nrh-nrl+1)*sizeof(float*));
    if (!m) nrerror("allocation failure 1 in matrix()");
    m -= nrl;

    for(i=nrl;i<=nrh;i++) {
        m[i]=(float *) malloc((unsigned) (nch-ncl+1)*sizeof(float));
        if (!m[i]) nrerror("allocation failure 2 in matrix()");
        m[i] -= ncl;
    }
    return m;
}

double **dmatrix(nrl,nrh,ncl,nch)
int nrl,nrh,ncl,nch;
{
    int i;
    double **m;

    m=(double **) malloc((unsigned) (nrh-nrl+1)*sizeof(double*));
    if (!m) nrerror("allocation failure 1 in dmatrix()");
    m -= nrl;

    for(i=nrl;i<=nrh;i++) {
        m[i]=(double *) malloc((unsigned) (nch-ncl+1)*sizeof(double));
        if (!m[i]) nrerror("allocation failure 2 in dmatrix()");
        m[i] -= ncl;
    }
    return m;
```

```
112: }
114:
115: int **imatrix(nrl,nrh,ncl,nch)
116: int nrl,nrh,ncl,nch;
117: {
118:     int i,**m;
119:
120:     m=(int **)malloc((unsigned) (nrh-nrl+1)*sizeof(int*));
121:     if (!m) nrerror("allocation failure 1 in imatrix()");
122:     m -= nrl;
123:
124:     for(i=nrl;i<=nrh;i++) {
125:         m[i]=(int *)malloc((unsigned) (nch-ncl+1)*sizeof(int));
126:         if (!m[i]) nrerror("allocation failure 2 in imatrix()");
127:         m[i] -= ncl;
128:     }
129:     return m;
130: }
131:
132:
133:
134: float **submatrix(a,oldrl,oldrh,oldcl,oldch,newrl,newcl)
135: float **a;
136: int oldrl,oldrh,oldcl,oldch,newrl,newcl;
137: {
138:     int i,j;
139:     float **m;
140:
141:     m=(float **) malloc((unsigned) (oldrh-oldrl+1)*sizeof(float*));
142:     if (!m) nrerror("allocation failure in submatrix()");
143:     m -= newrl;
144:
145:     for(i=oldrl,j=newrl;i<=oldrh;i++,j++) m[j]=a[i]+oldcl-newcl;
146:
147:     return m;
148: }
149:
150:
151:
152: void free_vector(v,nl,nh)
153: float *v;
154: int nl,nh;
155: {
156:     free((char*) (v+nl));
157: }
158:
159: void free_ivector(v,nl,nh)
160: int *v,nl,nh;
161: {
162:     free((char*) (v+nl));
163: }
164:
165: void free_dvector(v,nl,nh)
166: double *v;
167: int nl,nh;
```

```
168: {
170:     free((char*) (v+nl));
171: }
172:
173:
174:
175: void free_matrix(m,nrl,nrh,ncl,nch)
176: float **m;
177: int nrl,nrh,ncl,nch;
178: {
179:     int i;
180:
181:     for(i=nrh;i>=nrl;i--) free((char*) (m[i]+ncl));
182:     free((char*) (m+nrl));
183: }
184:
185: void free_dmatrix(m,nrl,nrh,ncl,nch)
186: double **m;
187: int nrl,nrh,ncl,nch;
188: {
189:     int i;
190:
191:     for(i=nrh;i>=nrl;i--) free((char*) (m[i]+ncl));
192:     free((char*) (m+nrl));
193: }
194:
195: void free_imatrix(m,nrl,nrh,ncl,nch)
196: int **m;
197: int nrl,nrh,ncl,nch;
198: {
199:     int i;
200:
201:     for(i=nrh;i>=nrl;i--) free((char*) (m[i]+ncl));
202:     free((char*) (m+nrl));
203: }
204:
205:
206:
207: void free_submatrix(b,nrl,nrh,ncl,nch)
208: float **b;
209: int nrl,nrh,ncl,nch;
210: {
211:     free((char*) (b+nrl));
212: }
213:
214:
215:
216: float **convert_matrix(a,nrl,nrh,ncl,nch)
217: float *a;
218: int nrl,nrh,ncl,nch;
219: {
220:     int i,j,nrow,ncol;
221:     float **m;
222:
223:     nrow=nrh-nrl+1;
```

```
224:     ncol=nch-ncl+1;
226:     m = (float **) malloc((unsigned) (nrow)*sizeof(float*));
227:     if (!m) nrerror("allocation failure in convert_matrix()");
228:     m -= nrl;
229:     for(i=0,j=nrl;i<=nrow-1;i++,j++) m[j]=a+ncol*i-ncl;
230:     return m;
231: }
232:
233:
234:
235: void free_convert_matrix(b,nrl,nrh,ncl,nch)
236: float **b;
237: int nrl,nrh,ncl,nch;
238: {
239:     free((char*) (b+nrl));
240: }
```

Appendix 2.

Description of the Discrete Cosine Transform (DCT) as Implemented by the Joint Photographic Experts Group (JPEG)

The Discrete Cosine Transform (DCT): Definitions and Rationale

Compression using the DCT is carried out in three stages: 1) DCT transform, 2) coefficient quantization, and 3) lossless coding. The two-dimensional DCT (cf., Pennebaker & Mitchell, 1993; Rao & Yip, 1990) is defined as follows:

$$DCT(i,j) = \frac{1}{\sqrt{2N}} C(i) \cdot C(j) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} I(x,y) \cdot \cos\left[\frac{(2x+1)i\pi}{2N}\right] \cdot \cos\left[\frac{(2y+1)j\pi}{2N}\right] \quad , \qquad \text{(A)}$$

where $C(x) = 1/\sqrt{2}$ when x=0 and $C(x) = 1$, otherwise, for an image of size NxN pixels. The DCT transform yields an NxN (i.e., square) matrix of frequency coefficients.

The inverse DCT is defined similarly as:

$$I(x,y) = \frac{1}{\sqrt{2N}} \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} C(i) \cdot C(j) \cdot DCT(i,j) \cdot I(x,y) \cdot \cos\left[\frac{(2x+1)i\pi}{2N}\right] \cdot \cos\left[\frac{(2y+1)j\pi}{2N}\right] \quad . \qquad \text{(B)}$$

After taking the DCT of an image, all the elements in row 0 have a zero frequency component in one direction, and all the elements in column 0 have a frequency component of zero in the other direction. As the zeros and columns move away from the origin, the coefficients in the transformed DCT matrix begin to represent higher frequencies, with the highest frequencies at position N-1 of the matrix. This fact is significant in image compression. For most images, the frequency components in row 0 and column 0 (the DC components) are relatively large. As we move away from the DC components toward the higher frequencies, we find that the coefficients not only tend to have lower values, but they become less important perceptually for image description. So the DCT can help to identify pieces of information in the image that can be effectively removed without seriously compromising image quality. The quantitative description of image quality is an important issue. So far, there is no single, good measure for specifying the perceptual quality of an image. The procedure for deciding how to discard the relatively insignificant image information (i.e., coefficients) will be described later.

Before taking the DCT, the image is broken down into small blocks of size 8x8 or 16x16 pixels, and the DCT is taken on these smaller blocks. The upper left corner in each block represents the DC term for that block. The lower right corner of the block represents the highest frequency for the block. The block size is a parameter, however, the conventional practice is to use an 8x8 block.

Quantization

The next step in image compression is quantization. The original image is represented by 8 bits per pixel resulting in 256 different gray scale levels. After the DCT, each frequency point takes on values in the range from -1,024 to +1,023, and thus occupies 11 bits. Quantization is a procedure that reduces the number of bits required to represent each coefficient. When we reduce the number of bits needed to store the coefficients, we also reduce precision. Since the coefficients far from the low frequency components contribute little to perceptual image quality, the precision of these coefficients can be low (i.e., they can be represented with a small number of bits). The farther away from the DC-point of the image, the less a given element contributes to image quality, and the less we are concerned with its precision.

Quantization is implemented using a quantization matrix. For every element position in the DCT matrix, a corresponding value in the quantization matrix gives a quantization value step-size. This step-size ranges from 1 to 255. The most important elements in the image will be encoded with a small step size, with size 1 giving the most precision. The less important values, corresponding to high frequencies, will be encoded with a large step size. The quantization formula is as follows:

$$Quantized\_value(i,j) = Round\_to\_nearest\_integer\left\{\frac{DCT(i,j)}{step\_size(i,j)}\right\}, \qquad (C)$$

where, *step_size (i,j)* is obtained from the quantization matrix. From the above formula, it is clear that for large step-size values, small *DCT (i,j)* values will be quantized to zero. This is the case, for instance, for most high frequency components. The smaller the step size, the higher is the precision of the quantization . Thus, for example, a high value of *DCT (i,j)* with a small step-size will have a large quantized value. A small value of *DCT (i,j)* with a large step-size will be

quantized to zero. Thus, only if high frequency coefficients reach unusually large values, will they be quantized to non-zero values.

The dequantization formula operates in reverse order:

$$DCT(i,j) = Quantized\_value(i,j) \cdot step\_size(i,j) \quad . \tag{D}$$

Obviously, large step-sizes can generate large errors.

Selection of a Quantization Matrix

The selection of an appropriate quantization matrix is the most important step in determining the quality of a decompressed image. Although there is no rigorous mathematical theory by which to choose the quantization matrix, there are many experimental approaches. One of the approaches is to measure the error between the original image and the decompressed image for each quantization matrix, and then choose the quantization matrix that gives the minimal error. Another approach is to evaluate the decompressed image using perceptual criteria. Of course, the perceptual criterion will not always correspond exactly to the minimal error criterion. Because the quantization matrix determines the decompressed image quality, choosing extraordinarily large step-sizes for the DCT coefficients results in excellent compression ratios but poor image quality. By choosing small step-sizes, we get low compression ratio but very good decompressed image quality. This tradeoff allows a great deal of flexibility in choosing the required picture quality based on imaging requirements and available storage.

The quantization matrix desired is computed as follows:

$$Step\_size[i][j] = 1 + [(1 + (i + j)) \cdot quality - factor] \tag{E}$$

First, the user selects a quality factor from 1 to 25. Quality factor 1 corresponds to best quality, while 25 corresponds to worst quality. Thus, using Equation E, the quantization matrix for quality 2 would be:

$$\begin{matrix} 3 & 5 & 7 & 9 \\ 5 & 7 & 9 & 11 \\ 7 & 9 & 11 & 13 \\ 9 & 11 & 13 & 15 \end{matrix} \qquad \text{(M1)}$$

As an example, if the DCT matrix before quantization were:

$$\begin{matrix} 92 & 3 & -9 & -7 \\ -84 & 62 & 1 & -18 \\ -52 & -36 & -10 & 14 \\ -39 & -58 & 12 & 17 \end{matrix} , \qquad \text{(M2)}$$

after dequantization using M1, we would have:

$$\begin{matrix} 90 & 0 & -7 & 0 \\ -35 & -56 & 9 & 11 \\ -84 & 54 & 0 & -13 \\ -45 & -33 & 0 & 0 \end{matrix} \qquad \text{(M3)}$$

The difference matrix is:

$$\begin{matrix} 2 & 3 & -2 & -7 \\ -4 & -2 & 3 & 6 \\ 0 & 6 & 1 & -6 \\ -7 & -3 & -10 & 14 \end{matrix}$$

which are rather small entries for a quality factor of 2. Thus, we would conclude that the quantization error affects the image quality.

Coding

The final step in compression is coding the quantized images. Recall that the coefficient block (0,0) represents the DC-value (i.e., the mean intensity) of the corresponding image block. Usually, the adjacent image block has a similar mean intensity value, and, therefore, for encoding purpose, the absolute value of the (0,0) coefficient-block is replaced by a relative value. Since adjacent blocks in an image are usually highly correlated, coding a given DC-value as the difference from the previous DC-value typically produces a very small number. Coefficients are encoded using run-length encoding of zero-value. Since a large number of coefficients are truncated to zero, this encoding scheme is very efficient. Finally, a zig-zag reordering of coefficients is performed to increase the run length of zero coefficients (see e.g., Pennebaker & Mitchell, 1993).