Image Processing with Wreath Products

William Y. Chang

Michael E. Orrison, Advisor

Weiqing Gu, Reader

May 7, 2004

HARVEY MUDD
COLLEGE
Department of Mathematics

Abstract

We present a technique to decompose digital images using generalizations of the Discrete Fourier Transform (DFT). This method arises from the representation theory of iterated wreath product groups, which describe symmetries of the quad tree decomposition of an image. In addition, we describe applications to compression, edge detection, and de-noising, as well as experimental results to evaluate the performance of these operations.

Contents

List of Figures

List of Tables

Acknowledgments

I'd like to thank my advisor, Michael Orrison, for his warm encouragement and focused guidance throughout my thesis research, and my second reader, Weiqing Gu, for her helpful comments. Many thanks to my colleagues in representation theory: Melissa Banister and Grant Clifford, and thanks to Eric Malm for helping me with LaTeX issues. This thesis could not possibly exist without all of you!

Chapter 1

Introduction

Image processing refers to the various operations performed on pictures that are digitally stored as an aggregate of pixels. There are many problems in image processing, including enhancing or degrading the quality of the image, artistically transforming the image, and finding and recognizing objects in an image.

Our approach traces back to Cooley and Tukey's seminal paper on the Fast Fourier Transform (FFT) algorithm [4]. Subsequent efforts to formulate the discrete Fourier transform (DFT) in terms of the representation theory of finite cyclic groups provides the basis of our approach. In two recent papers, Foote et. al. and Mirchandani et. al. ([8], [18]) link the spectral analysis approach for data on finite groups with the world of wavelets. Their framework realizes the DFT and the Haar wavelet transform as special cases of a more general construction. In particular, the use of iterated wreath products as automorphism groups of spherically homogeneous rooted trees (SHRTs) provides the vital link between wavelets, spectral analysis, and image processing.

Following in the same direction, this paper explores the spectral decomposition algorithm from [8]. The main contribution of this paper is the application of iterated wreath products of different symmetry groups, which is previously unexplored in the field. We derive individual decompositions based on choices of symmetry groups, noting the differences of the Haar, Fourier, dihedral, alternating, and symmetric decompositions. In addition, we investigate the applicability of classical image processing techniques to our image spectra by demonstrating compression, edge detection, and de-noising techniques.

Our approach in the following chapters is as follows. First, we give the necessary background tools from representation theory, and we explain how to generalize the Discrete Fourier Transform (DFT) using these tools. Second, we introduce wreath products and describe their structure. Third, we derive a wreath product invariant decomposition with respect to several iterated wreath product groups. Lastly, we describe approaches for compression, edge detection, and de-noising as a proof-of-concept that our spectral analysis is applicable to standard image processing problems.

Chapter 2

Representation Theory

In this chapter, we briefly cover the basic tools of representation theory. We assume basic knowledge of groups, rings, and modules. A more complete treatment of representation theory can be found in the later chapters of [6]. As a concrete application of representation theory, we derive the classical DFT. This material is found (concisely presented) in [8], [17], and [21].

2.1 Representations of Finite Groups

Let G be a finite group, let F be a field and let V be a vector space over F.

Definition 2.1. Let $n \in \mathbb{Z}^+$.

(1) A *linear representation* of G is any homomorphism φ from G to $GL(V)$.

(2) A *matrix representation* of G is any homomorphism φ from G to $GL_n(F)$.

A representation for a group is simply a map from group elements in G to linear transformations in $GL(V)$. When V is a finite dimensional vector space, we can fix a basis for V and obtain an isomorphism from $GL(V)$ to $GL_n(F)$. Since this paper only concerns finite dimensional vector spaces, we will use linear and matrix representations interchangeably.

Although we can think of representations as maps, we can also think of them as modules over group rings. Recall the definitions of group rings and modules:

Definition 2.2. The *group ring* of G over F, denoted FG, is the set of all formal sums of the form

$$\sum_{g \in G} \alpha_g g, \qquad \alpha_g \in F$$

where addition and multiplication are performed in the usual distributive fashion.

Definition 2.3. Let R be a ring (not necessarily commutative nor with 1). A *left R-module* or a *left module over R* is a set M together with

(1) a binary operation $+$ on M under which M is an abelian group, and

(2) an action of R on M (that is, a map $R \times M \to M$) denoted by rm such that for all $r, s \in R$ and for all $m, n \in M$

- **(a)** $(r+s)m = rm + sm$,
- **(b)** $(rs)m = r(sm)$,
- **(c)** $r(m+n) = rm + rn$,
- **(d)** if the ring R has a 1, then $1m = m$.

Now, suppose $\varphi : G \to GL(V)$ is a representation of G. Amazingly enough, we can consider the vector space V (over F) as a module over the ring FG, where the action of FG on V is defined by

$$\left(\sum_{g\in G} \alpha_g g\right) \cdot v = \sum_{g\in G} \alpha_g \varphi(g)(v) \quad \text{for all } \sum_{g\in G} \alpha_g g \in FG, v \in V.$$

Conversely, given an FG-module V, we can obtain a representation $\varphi : G \to GL(V)$ using the action of FG on V:

$$\varphi(g)(v) = g \cdot v \quad \text{for all } v \in V,$$

where $g \cdot v$ is the action of the group element g on the element v of V. Therefore, a representation $\varphi : G \to GL(V)$ is equivalent to specifying an FG-module V. Here, we say that V *affords* the representation φ.

This correspondence between representations and modules is very useful because now we can apply module theory to explore properties of representations. In particular, we are interested in the basic building blocks of representations by considering *submodules*, which correspond to ”sub”-representations. Furthermore, we can define the notion of a *simple* or *irreducible* representation analogously to how a simple module is defined. To refresh the reader’s memory, let us recall the definitions of a submodule and a simple module:

Definition 2.4. Let M be a nonzero R-module for a ring R.

(1) An *R-submodule* of M is a subgroup N of M which is closed under the action of ring elements, i.e., $rn \in N$ for all $r \in R, n \in N$.

(2) M is said to be *irreducible* or *simple* if its only submodules are 0 and M; otherwise, M is called *reducible*.

One of the most important problems in group theory is the *Hölder Program*, which attempts to classify all finite simple groups and to find all ways of putting simple groups together to form other groups. In module theory, an analogous problem is the *extension problem*, in which we study exact sequences of modules, direct sums of modules, and tensor products of modules. In the representation theory of finite groups, *Wedderburn's theorem* (Theorem 2.2) tells us how to break representations up into smaller irreducible representations. This astounding result forms the foundation for our approach in decomposing signals and images. For the remainder of this section, we will focus on Wedderburn's theorem and other important objects and tools in representation theory that allow us to use Wedderburn's theorem.

The first important theorem we will discuss is Maschke's theorem. Intuitively, this theorem states that for any submodule of an FG-module, you can always find a complementary submodule. This enables us to split the original module into a direct sum of two submodules.

Definition 2.5. Let 1_F be the multiplicative identity of the field F. The *characteristic* of a field F is the smallest positive integer p such that $p \cdot 1_F = 0$ if such a p exists, and 0 otherwise.

Theorem 2.1. (Maschke's Theorem) *Let G be a finite group and let F be a field whose characteristic does not divide $|G|$. If V is any FG-module and U is any submodule of V, then V has a submodule W such that $V = U \oplus W$.*

In terms of representation theory, if we are given a representation φ for a subspace of V which is closed the action of φ, then we can find a *complement* which is also closed under φ. In addition, for complex representations, you can find a *orthogonal complement* to the original subspace with respect to some inner product. Maschke's theorem will be important in proving Theorem 4.2. In addition, this theorem is useful in proving Wedderburn's theorem. But first we state a quick corollary from Maschke's theorem:

Corollary. *If G is a finite group and F is a field whose characteristic does not divide $|G|$, then every finitely generated FG-module M is* completely reducible, *i.e. M is isomorphic to a direct sum of irreducible submodules.*

This is a direct consequence of Maschke's theorem: given any non-trivial submodule of an FG-module M, we can recursively apply Maschke's theorem to the submodule and its complement until we cannot break them down any further (i.e. until we obtain irreducible submodules). In addition, note that for any finite group G and $F = \mathbb{C}$, every $\mathbb{C}G$-module over a finite dimensional vector space V is finitely generated, because we can always choose a finite basis which spans V. Therefore, all finite dimensional $\mathbb{C}G$-modules are completely reducible.

Now we are ready to state Wedderburn's theorem. We will conveniently apply this theorem to $\mathbb{C}G$-modules, since they are guaranteed to be completely reducible.

Theorem 2.2. (Wedderburn's Theorem) *Let R be a nonzero ring with* 1*. If every R-module is completely reducible, then the ring R considered as a left R-module is a direct sum:*

$$R \cong L_1 \oplus L_2 \oplus \cdots \oplus L_n,$$

where each L_i is a simple module with $L_i = Re_i$, for $e_i \in R$ which satisfy

(1) $e_i{}^2 = e_i$ *for all i (the e_i are* idempotents*),*

(2) $e_i e_j = 0$ *if $i \neq j$ (the idempotents are* orthogonal*),*

(3) *and* $\sum_{i=1}^{n} e_i = 1$.

This theorem guarantees us a decomposition of any $\mathbb{C}G$-module M as a direct sum of irreducible submodules:

$$M \cong M_1 \oplus M_2 \oplus \cdots \oplus M_l.$$

Note that the M_i are not necessarily distinct; in fact, there may be many irreducible submodules that are isomorphic. Thus, this decomposition is not unique. However, we can group the isomorphic components to form a *isotypic decomposition* of M:

$$M \cong a_1N_1 \oplus a_2N_2 \oplus \cdots \oplus a_kN_k,$$

where each a_i is a nonnegative integer indicating the multiplicity of the irreducible submodule N_i, i.e.

$$a_iN_i = \overbrace{N_i \oplus \cdots \oplus N_i}^{a_i \text{ times}}.$$

Note that $N_i = M_j$ for some j, and each N_i is distinct. We refer to the submodules a_iN_i as *isotypic subspaces*.

Now, recall in Wedderburn's theorem that we obtained the irreducible modules L_i by projection using the idempotents e_i. If there were a simple way of obtaining these idempotents, then applying Wedderburn's theorem would be easy. It turns out that we can use the theory of *characters* to find these idempotents.

Definition 2.6. If φ is a matrix representation of G afforded by the FG-module V, the *character* of φ is the function

$$\chi : G \to F \quad \text{defined by} \quad \chi(g) = \operatorname{tr}\varphi(g),$$

where $\operatorname{tr}\varphi(g)$ is the trace of $\varphi(g)$.

Recall that $\operatorname{tr} AB = \operatorname{tr} BA$ for two $n \times n$ matrices A and B. For an invertible matrix P, we see that the trace is invariant under conjugation by P:

$$\operatorname{tr} P^{-1}AP = \operatorname{tr} P.$$

Thus, the character of a representation is invariant under any change of basis. In other words, characters are *class functions*, i.e. constant on the conjugacy classes of G:

$$\chi(g^{-1}xg) = \operatorname{tr}\left(\varphi(g^{-1}xg)\right) = \operatorname{tr}\left(\varphi(g^{-1})\varphi(x)\varphi(g)\right) = \operatorname{tr}\varphi(x) = \chi(x).$$

We now apply Wedderburn's theorem to characters. Suppose we have a character ψ for the complex representation complex φ, which is afforded by some $\mathbb{C}G$-module V. Then, we can decompose the matrix of φ as a block diagonal matrix, where each block corresponds to an irreducible representation of φ or an irreducible submodule of V. Then, ψ is actually the sum of the characters of these irreducible representations:

$$\psi = a_1\chi_1 + a_2\chi_2 + \cdots + a_r\chi_r,$$

where each χ_i is the character of the representation corresponding to the i^{th} irreducible submodule of V.

We can do even more with complex characters; in fact, we can put a inner product structure on the space of complex class functions as follows (recall that characters are just particular class functions):

Definition 2.7. For class functions θ and ψ, define their inner product to be

$$\langle\theta,\psi\rangle = \overline{\langle\psi,\theta\rangle} = \frac{1}{|G|}\sum_{g\in G}\theta(g)\overline{\psi(g)},$$

where the bar denotes complex conjugation.

To simplify the calculation of this inner product, we can take advantage of the fact that class functions are constant over conjugacy classes. So suppose there are l conjugacy classes with representatives $g_1, g_2, \ldots, g_l$. Then, the inner product of θ and ψ can be expressed as

$$\langle\theta,\psi\rangle = \sum_{i=1}^{l}\frac{\chi(g_i)\overline{\psi(g_i)}}{|C_G(g_i)|},$$

where $|C_G(g_i)|$ is the order of the centralizer of g_i in G. With respect to this formula, we have an important orthogonality relation between the irreducible characters:

Theorem 2.3. (The First Orthogonality Relation for Group Characters) *Let G be a finite group and let $\chi_1, \dots, \chi_l$ be the irreducible characters of G over $\mathbb{C}$. Then, with respect to the inner product $\langle\, , \rangle$ above we have*

$$\langle \chi_i, \chi_j \rangle = \delta_{ij}$$

and the irreducible characters are an orthonormal basis for the space of class functions. In particular, if θ is any class function then

$$\theta = \sum_{i=1}^{l} \langle \theta, \chi_i \rangle \chi_i. \tag{2.1}$$

In addition, we can define a notion of a *norm* of a class function:

Definition 2.8. For θ any class function on G the *norm* of θ is $\langle \theta, \theta \rangle^{1/2}$ and is denoted by $\|\theta\|$.

Applying Theorem 2.3, we see that when the class function θ is expressed using irreducible characters, $\theta = \sum_{i=1}^{l} \alpha_i \chi_i$, then the norm is simply

$$\|\theta\| = \left(\sum_{i=1}^{l} {\alpha_i}^2 \right)^{1/2}.$$

Thus it follows that a character has norm 1 if and only if it is irreducible.

Now, we arrive at the long awaited formula for obtaining idempotents. Although we have briefly stated definitions for idempotents in Wedderburn's theorem, let us formally define idempotents and some of their properties.

Definition 2.9. Let R be a ring, and let $Z(R)$ be the center of R. Then,

(1) An element $e \in R$ is called an *idempotent* if $e^2 = e$.

(2) Idempotents e_1 and e_2 are orthogonal if $e_1 e_2 = e_2 e_1 = 0$.

(3) An idempotent e is called a *primitive central idempotent* if $e \in Z(R)$ and e cannot be written as a sum of two other orthogonal idempotents in the ring $Z(R)$.

And finally, the formula for calculating idempotents:

Theorem 2.4. *Let M be a $\mathbb{C}G$-module, and let M_i be the irreducible submodules of the Wedderburn decomposition of M. Let $e_1, \dots, e_l$ be the orthogonal primitive central idempotents in $\mathbb{C}G$ such that e_i restricted to the irreducible M_i is the identity map. Furthermore, let χ_i be the character afforded by M_i. Then*

$$e_i = \frac{\chi_i(1)}{|G|} \sum_{g \in G} \chi_i(g^{-1}) g. \tag{2.2}$$

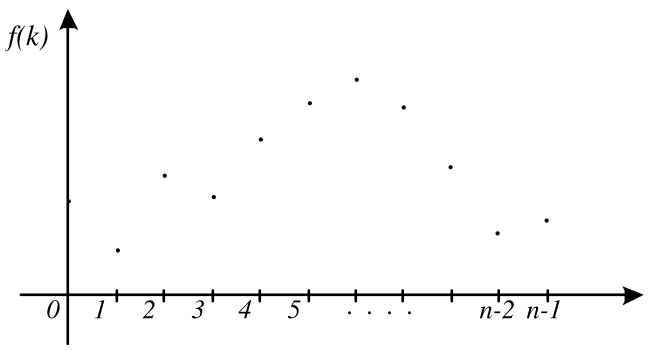


Figure 2.1: A Function Sampled at n Points.

2.2 Applying Representation Theory: A Derivation of the DFT

In this section, we will apply the tools from the previous section to derive the classical DFT. In order to do this, we will need a representation of some group G, a corresponding $\mathbb{C}G$-module, and the use of Wedderburn's theorem to decompose this module.

Suppose we have a complex-valued function $f(k)$ that is sampled at (or defined on) n points; i.e. $k \in \{0, \dots, n-1\}$, e.g., Figure 2.1. Then the classical DFT of this function is given by

$$\widehat{f}(j) = \frac{1}{n} \sum_{k=0}^{n-1} f(k) e^{-2\pi i k \frac{j}{n}}. \tag{2.3}$$

Similarly, the inverse of this transform, the IDFT, is given by

$$f(k) = \sum_{j=0}^{n-1} \widehat{f}(j) e^{2\pi i k \frac{j}{n}}.$$

Now, denote the cyclic group on n elements as Z_n, and let $G = Z_n$. Suppose that the generator for Z_n is the element $x \in Z_n$. Amazingly, for our function f, we can identify the n points on the domain of f with group elements of Z_n. Therefore each f simply becomes an element of $\mathbb{C}G$:

$$f = \sum_{g \in G} \alpha_g g = \sum_{i=0}^{n-1} \alpha_i x^i, \quad \text{where } \alpha_i = f(i).$$

Therefore the space of all functions defined on this domain can be represented as a $\mathbb{C}G$-module over itself, $\mathbb{C}G$. Let us proceed to decompose this space using Wedderburn's theorem.

Let $\{\varphi_i\}$ be the irreducible representations of φ. For a finite abelian group, all of its irreducible complex representations φ_i are 1-dimensional, since the matrices $\varphi_i(g)$ for all $g \in G$ need be commutative as well. Also, all 1-dimensional complex representations of a finite cyclic group maps the group elements to the n^{th} roots of unity in $\mathbb{C}$. Thus, for a primitive n^{th} root of unity $\omega = e^{\frac{-2\pi i}{n}}$, the irreducible characters of Z_n are

$$\chi_k(x^j) = \omega^{jk}.$$

To show that these characters are indeed irreducible, we calculate the norm of χ_k:

$$\begin{aligned}
\langle \chi_k, \chi_k \rangle &= \frac{1}{|G|} \sum_{g \in G} \chi_k(g)\overline{\chi_k(g)} \\
&= \frac{1}{n} \left(\sum_{j=0}^{n-1} \chi_k(x^j)\overline{\chi_k(x^j)} \right) \\
&= \frac{1}{n} \left(\sum_{j=0}^{n-1} \omega^{jk}\omega^{-jk} \right) = 1.
\end{aligned}$$

From this information, we can derive the orthogonal primitive central idempotents using Theorem 2.4:

$$\begin{aligned}
e_k &= \frac{\chi_k(1)}{|G|} \sum_{g \in G} \chi_k(g^{-1})g \\
&= \frac{1}{n} \sum_{j=0}^{n-1} \chi_k(x^n x^{-j})x^j \\
&= \frac{1}{n} \sum_{j=0}^{n-1} \omega^{-jk}x^j.
\end{aligned}$$

Then, the individual idempotents look like

$$\begin{aligned}
e_0 &= \frac{1}{n}\left(1 + x + x^2 + x^3 + \cdots + x^{n-1}\right) \\
e_1 &= \frac{1}{n}\left(1 + \omega^{-1}x + \omega^{-2}x^2 + \omega^{-3}x^3 + \cdots + \omega^{-(n-1)}x^{n-1}\right) \\
e_2 &= \frac{1}{n}\left(1 + \omega^{-2}x + \omega^{-4}x^2 + \omega^{-6}x^3 + \cdots + \omega^{-2(n-1)}x^{n-1}\right) \\
&\vdots \\
e_{n-1} &= \frac{1}{n}\left(1 + \omega^{-(n-1)}x + \omega^{-2(n-1)}x^2 + \omega^{-3(n-1)}x^3 + \cdots + \omega^{-(n-1)^2}x^{n-1}\right).
\end{aligned}$$

Now, let us project $\mathbb{C}G$ into its irreducible submodules according to Wedderburn's theorem. For any function $f = \sum_{k=0}^{n-1} \alpha_k x^k \in \mathbb{C}G$, where each $\alpha_k \in \mathbb{C}$, we can project f onto the i^{th} irreducible submodule by calculating $e_i \cdot f$. As an example, consider the projection of f to the second irreducible submodule, $e_1 \cdot f$:

$$\begin{aligned} e_1 \cdot f &= \frac{1}{n}\left(1 + \omega^{-1}x + \omega^{-2}x^2 + \cdots + \omega^{-(n-1)}x^{n-1}\right) \cdot f \\ &= \frac{1}{n}\left[1 \cdot f + (\omega^{-1}x) \cdot f + (\omega^{-2}x^2) \cdot f + \cdots + (\omega^{-(n-1)}x^{n-1}) \cdot f\right]. \end{aligned}$$

For sake of notational simplicity, let us use vectors to denote elements in $\mathbb{C}G$ by

$$f = \sum_{k=0}^{n-1} \alpha_k x^k = \begin{bmatrix} \alpha_0 \\ \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_{n-1} \end{bmatrix}.$$

Then, we can write $e_1 \cdot f$ as

$$\begin{aligned} e_1 \cdot f &= \frac{1}{n}\left(1 \cdot \begin{bmatrix} \alpha_0 \\ \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_{n-1} \end{bmatrix} + (\omega^{-1}x) \cdot \begin{bmatrix} \alpha_0 \\ \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_{n-1} \end{bmatrix} + \cdots + (\omega^{-(n-1)}x^{n-1}) \cdot \begin{bmatrix} \alpha_0 \\ \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_{n-1} \end{bmatrix}\right) \\ &= \frac{1}{n}\left(\begin{bmatrix} \alpha_0 \\ \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_{n-1} \end{bmatrix} + \omega^{-1}\begin{bmatrix} \alpha_{n-1} \\ \alpha_0 \\ \alpha_1 \\ \vdots \\ \alpha_{n-2} \end{bmatrix} + \omega^{-2}\begin{bmatrix} \alpha_{n-2} \\ \alpha_{n-1} \\ \alpha_0 \\ \vdots \\ \alpha_{n-3} \end{bmatrix} + \cdots + \omega^{-(n-1)}\begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \\ \vdots \\ \alpha_0 \end{bmatrix}\right). \end{aligned}$$

Using matrices, we can simplify the notation even further:

$$e_1 \cdot f = \frac{1}{n}\begin{bmatrix} 1 & \omega^{-(n-1)} & \omega^{-(n-2)} & & \omega^{-1} \\ \omega^{-1} & 1 & \omega^{-(n-1)} & \cdots & \omega^{-2} \\ \omega^{-2} & \omega^{-1} & 1 & & \omega^{-3} \\ & \vdots & & \ddots & \vdots \\ \omega^{-(n-1)} & \omega^{-(n-2)} & \omega^{-(n-3)} & \cdots & 1 \end{bmatrix}\begin{bmatrix} \alpha_0 \\ \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_{n-1} \end{bmatrix}.$$

Multiplying by $\omega^n = 1$, we get

$$\begin{aligned}
e_1 \cdot f &= \frac{1}{n} \begin{bmatrix} 1 & \omega & \omega^2 & & \omega^{n-1} \\ \omega^{n-1} & 1 & \omega & \cdots & \omega^{n-2} \\ \omega^{n-2} & \omega^{n-1} & 1 & & \omega^{n-3} \\ & \vdots & & \ddots & \vdots \\ \omega & \omega^2 & \omega^3 & \cdots & 1 \end{bmatrix} \begin{bmatrix} \alpha_0 \\ \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_{n-1} \end{bmatrix} \\
&= \frac{1}{n} \begin{bmatrix} 1 \\ \omega^{n-1} \\ \omega^{n-2} \\ \vdots \\ \omega \end{bmatrix} \begin{bmatrix} 1 & \omega & \omega^2 & \cdots & \omega^{n-1} \end{bmatrix} \begin{bmatrix} \alpha_0 \\ \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_{n-1} \end{bmatrix} \\
&= \frac{1}{n} \sum_{k=0}^{n-1} \omega^k \alpha_k \begin{bmatrix} 1 \\ \omega^{n-1} \\ \omega^{n-2} \\ \vdots \\ \omega \end{bmatrix}.
\end{aligned}$$

Notice that the vector above only has one degree of freedom. Thus, we are projecting onto a one-dimensional space, and we only need to store one coefficient. We pick the coefficient of 1_G, and thus we finally obtain

$$e_1 \cdot f = \frac{1}{n} \sum_{k=0}^{n-1} \alpha_k \omega^k.$$

This looks exactly like the 2nd Fourier coefficient from the formula we saw earlier in Equation 2.3.

In general, for any projection $e_i \cdot f$, we obtain (after much calculation)

$$e_i \cdot f = \frac{1}{n} \sum_{k=0}^{n-1} \alpha_k \omega^{ik} \qquad \text{for} \quad 0 \le i \le n-1.$$

This is precisely Equation 2.3. Furthermore, we can express all of the projections $e_i \cdot f$ succinctly in matrix form, denoted as DFT_n:

$$\mathrm{DFT}_n = \begin{bmatrix} e_0 \cdot f \\ e_1 \cdot f \\ e_2 \cdot f \\ \vdots \\ e_{n-1} \cdot f \end{bmatrix} = \frac{1}{n} \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)^2} \end{bmatrix} \begin{bmatrix} \alpha_0 \\ \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_{n-1} \end{bmatrix}. \tag{2.4}$$

Chapter 3

Wreath Products

We can use wreath product groups to describe permutations of special types of trees called “spherically homogeneous rooted trees,” or SHRTs for short. The inspiration to use SHRTs comes from [8], and we will draw on its definitions and insights for this chapter. To clarify some basic terminology, a *rooted* tree is a tree with a designated *vertex* (or *node*) called the *root*. These rooted trees are typically drawn like an upside-down plant with the root at the very top, as in Figure 3.1. In these trees, the vertices are arranged according to their distance from the root, which is referred to as the *level* of a vertex. A *child* of a specified vertex v is a vertex that is connected to v and one node further from the root than v. The *leaves* of the tree are the vertices that are of maximal distance from the root, i.e., vertices that have no children. For the sake of convenience, let the maximum level of any vertex in a tree be called the *height* of the tree.

A SHRT is a rooted tree such that all vertices at a given level have the same number of children. We denote SHRTs with an h-tuple $(m_0, m_1, \ldots, m_{h-1})$ where h is the height of the tree and m_i denotes the number of children at each vertex of level i. Note here that all of the leaves in this tree are of the same distance to the root, at the maximum level h.

Figure 3.1 is an example of a SHRT described by the 3-tuple $(2,2,2)$. This type of tree, where every vertex (except the leaves) has exactly two children, is called a *binary tree.* Similarly, a *quad tree* refers to a SHRT where every non-leaf vertex has exactly four children. In general, these trees are called *regularly branching r-ary trees*, where r specifies the number of children at each vertex.

In an image processing setting, we would like to structure our data so that it is convenient to define a notion of a group acting on our image. In other words, we would like to apply symmetry groups to our image in order to find symmetries within the image. Furthermore, we would like to find decompositions of the image

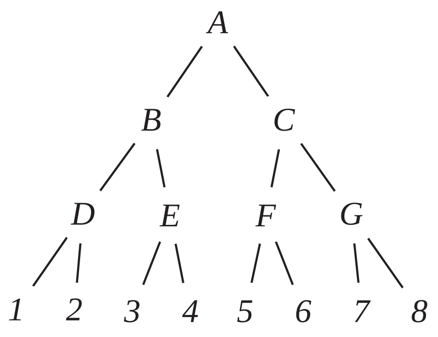


Figure 3.1: A Graph of a Binary Tree of 3 Levels.

data with respect to these symmetries. In fact, it is particularly useful to identify the pixels in the image with the leaves of a SHRT, because it allows for wreath product groups to act on the image by permuting the corresponding SHRT.

Now, how would one describe permutations of trees? Intuitively, we would like some way to describe a reordering of vertices in the SHRT. Since we want to preserve the adjacency of the vertices of the tree (otherwise, we would be creating a completely different tree), we would like to permute closely connected vertices amongst themselves. We can do this by only permuting children (of a particular vertex) amongst themselves, for all vertices in the tree.

This is equivalent to picking an element from the symmetric group for each vertex in the tree, where the element describes how the vertex permutes its children while keeping the subtrees below the children intact. But recall that for SHRTs, the number of children is exactly the same for vertices at a given level. So the i^{th}in the tree, we can pick an element of the same group S_{m_i} for each vertex of that level. This is precisely the algebraic structure offered by wreath products of symmetric groups. Note that for regularly branching r-ary trees, the number of vertices gets exponentially large as the height of the tree increases, so we would expect these wreath product groups to become very large very fast.

Now, another way to interpret these permutations is to see how they pertain to permutations of just the leaves. In fact, the wreath products describe a *structured permutation* of the leaves. This aspect proves to be especially useful when defining the wreath product group action on an image.

In addition to this interpretation of wreath products, there has been much theoretical work with other uses of wreath products. For further reading, Kerber ([12], [13]) talks about the representation theory of wreath products in more detail. Rockmore ([20]) gives a FFT for wreath products. Furthermore, Eldredge ([7]) describes an approach to finding isotypic decompositions of data in the binary tree case by using separating sets. In addition, wreath products are used in a variety of different settings. Straubing ([23]) applies wreath products in finite automata theory, deriving a "wreath product principle" for formal languages. Finally, Schoolfield analyzes random walks on wreath products of groups [22].

3.1 Wreath Products

Definition 3.1. Let G be a finite group, and H a permutation group on n elements ($H \leq S_n$). Let $G^n = G \times G \times \cdots \times G$ (n times) be the set of ordered n-tuples of elements of G. Let $g, h \in G^n$, $\sigma, \pi \in H$, and let h_i denote the ith coordinate in the n-tuple h. Then the *wreath product* of G with H, denoted $G \wr H$, is the set $G^n \times H$ with multiplication defined as

$$(g,\sigma)(h,\pi) = \left(g\left(\sigma^{-1}h\right), \sigma\pi\right)$$

where

$$\left(\sigma^{-1}h\right)_i = h_{\sigma^{-1}(i)}. \tag{3.1}$$

We illustrate the wreath product with a simple example, using the tree in Figure 3.2. The group that describes all adjacency preserving permutations of this tree is $S_2 \wr S_3$. Let $(g,\sigma), (h,\pi) \in S_2 \wr S_3$ be the permutations $\Big(\big(1,(12),1\big),(132)\Big)$ and $\Big(\big((12),1,1\big),(12)\Big)$, respectively. Then, the composition of these two permutations (by multiplication in $S_2 \wr S_3$) is

$$\begin{aligned}
(g,\sigma)(h,\pi) &= \Big(\big(1,(12),1\big),(132)\Big)\Big(\big((12),1,1\big),(12)\Big) \\
&= \Big(\big(1,(12),1\big)\left[(132)^{-1}\big((12),1,1\big)\right],(132)(12)\Big) \\
&= \Big(\left[\big(1,(12),1\big)\big(1,1,(12)\big)\right],(23)\Big) \\
&= \Big(\big(1,(12),(12)\big),(23)\Big).
\end{aligned}$$

The resulting permutation should be equivalent to applying the permutations (h,π) and then (g,σ) in that order. To check, we can apply (h,π) and then (g,σ) as shown in Figure 3.3. Here, we applied the permutations in a top-down fashion, first applying π (or σ) by swapping the whole subtrees with roots b,c,d and then applying h (or g) to the leaves, where the ith coordinate of h applies to the children of the ith subtree. We see in this example that the final permutation does in fact correspond to the one obtained by the multiplication formula for the wreath product.

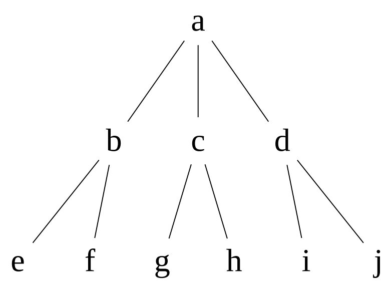


Figure 3.2: Graph of a (3,2) SHRT.

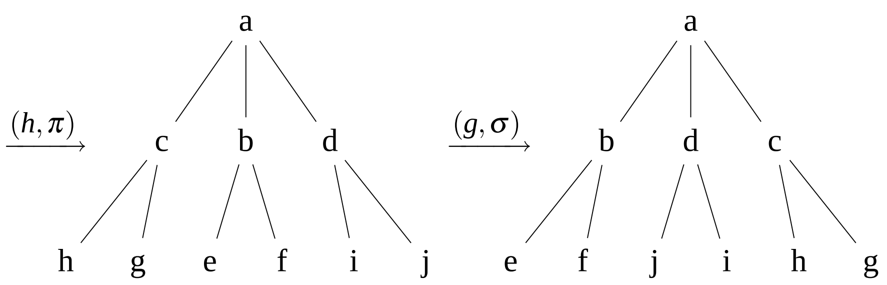


Figure 3.3: Subsequent Permutations of Figure 3.2.

An intuitive way to see why this works is to observe that the permutations of vertices higher up the tree will affect the permutations below. Therefore, in composing permutations of trees, the permutations below have to be "untwisted" in order to be applied correctly. This reveals the semidirect product structure of the wreath product, where the action of H on G^n is to permute the ordering of the n-tuples. More precisely, $G \wr H$ is a semidirect product of H and G^n with respect to the permutation representation from H into $\mathrm{Aut}(G^n)$ defined by Equation 3.1, where $\mathrm{Aut}(G^n)$ denotes the full automorphism group of G^n.

In addition, we can also create a correspondence between the permutations of the leaves and permutations of the tree. However, not all permutations of the leaves can be constructed using permutations of the tree. For example, in Figure 3.2, since the adjacency of the vertices needs to be preserved, we would not be able to permute the leaves e and f so that they have different parents, since they are connected to a common parent. But there exists a unique wreath product permutation for each permutation of the leaves that is realizable using permutations from the wreath product group.

Proposition 3.1. *Let $(m_0, m_1, \ldots, m_{h-1})$ be a SHRT of height h. Let π be a permutation of the $\prod_{i=0}^{h-1} m_i$ leaves of this tree, which is realizable by adjacency preserving permutations of the tree. Then there exists a unique element in $S_{m_{h-1}} \wr S_{m_{h-2}} \wr \cdots \wr S_{m_0}$ that gives rise to π.*

Proof. We prove this by induction on the height of the tree. First, for $h = 1$, there are m_0 leaves connected directly to the root of the tree, and clearly there exists a unique permutation in S_{m_0} that permutes the leaves of this tree. Now, suppose that our proposition is true for some height h. Consider the SHRT of height $h+1$. Since the permutations of the leaves is adjacency-preserving, we can partition this set of leaves into m_{h-1} sets of m_h vertices of distance 2 from each other. For each of these sets, there exists a unique element in S_{m_h} that describes the permutation of its vertices. Thus, there exists a unique element in $\sigma \in S_{m_h} \times \cdots \times S_{m_h}$ (m_{h-1} times) that describes each permutation of vertices in all m_{h-1} sets. By the induction hypothesis, there exists a unique element in $\tau \in S_{m_{h-1}} \wr S_{m_{h-2}} \wr \cdots \wr S_{m_0}$ that describes the ordering of the m_{h-1} groups. Thus, we have a unique element $(\sigma, \tau) \in S_{m_h} \wr S_{m_{h-1}} \wr \cdots \wr S_{m_0}$ that corresponds to this permutation of the leaves. By the principle of induction, the proposition follows. $\square$

This proposition tells us that there exists a bijective correspondence between realizable permutations of the leaves and wreath product permutations of the tree. In this manner, the wreath product gives rise to a *structured permutation* of the leaves. In the next chapter, we will see that this structure will allow us to identify pixels in the image with a unique SHRT. In addition, we will be able to use the wreath product to permute the pixels in the image in a very organized manner. Furthermore, we can investigate whether there exists a transformation that is invariant under the wreath product group action, which will in turn lead to a decomposition.

Chapter 4

Image Decomposition using Wreath Product Groups

The idea of applying wreath products to image processing was first introduced in [10]. Further developing this idea, Foote et. al. and Mirchandani et. al. ([8], [18]) describe the quad tree decomposition of an image, which involves a scanning method to identify pixels with quad trees. In this chapter, we will look in greater detail at the algorithm given by [8], and we will introduce a decomposition of the image data which is invariant under the wreath product group action.

4.1 Quad Tree Decompositions of Images

How can automorphisms of trees help us with image processing? We can construct a tree-like structure from a digital image by recursively breaking the image into pieces, creating a hierarchy within the image. This structure allows us to permute pixels in an image just like how we would permute the leaves of a tree using iterated wreath product groups.

Procedure 4.1. Suppose we are given a image with dimensions $2^h \times 2^h$.

(1) Break the image up into four quadrants, each of dimension $2^{h-1} \times 2^{h-1}$.

(2) Order the four quadrants clockwise, starting from the top left quadrant.

(3) Continue the process recursively, breaking quadrants into sub-quadrants (of dimension $2^{h-2} \times 2^{h-2}$) and sub-quadrants into sub-sub-quadrants (of dimension $2^{h-3} \times 2^{h-3}$), and so on.

(4) Stop at the hth recursive step, when quadrants cannot be broken up further since they are made of individual pixels.

Example 4.1. Suppose we have an 4×4 image, represented by the matrix

$$M = \begin{pmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{pmatrix}.$$

We first break the image into four quadrants:

$$\left(\begin{array}{cc|cc} a & b & c & d \\ e & f & g & h \\ \hline i & j & k & l \\ m & n & o & p \end{array}\right) \sim \left(\begin{array}{c|c} A & B \\ \hline D & C \end{array}\right).$$

And, for each of the four sub-quadrants, we can break them down again into sub-sub-quadrants:

$$\left(\begin{array}{c:c|c:c} a & b & c & d \\ \hdashline e & f & g & h \\ \hline i & j & k & l \\ \hdashline m & n & o & p \end{array}\right).$$

At this point, we must stop the recursive process since we cannot subdivide the matrix any further. Now, using the ordering scheme, we can order the vertices as follows:

$$\left(\begin{array}{cc|cc} 1 & 2 & 5 & 6 \\ 4 & 3 & 8 & 7 \\ \hline 13 & 14 & 9 & 10 \\ 16 & 15 & 12 & 11 \end{array}\right).$$

This gives rise to the quad-tree in Figure 4.1.

We note that this scanning method can be generalized to other trees as well, however, the size of the image must be adjusted to match the number of leaves in the desired tree. There are also variations within quad trees. For any tree $\mathscr{T}_{\mathbf{m}}$ where $\mathbf{m}$ is an n-tuple $(4^k, 4^k, \dots, 4^k)$, we obtain different quad trees for different choices of k and n. We can then generalize our scanning procedure as well. For example, in the 4×4 matrix given above, we could have scanned 16 vertices at once, giving the ordering

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 12 & 13 & 14 & 5 \\ 11 & 16 & 15 & 6 \\ 10 & 9 & 8 & 7 \end{pmatrix}.$$

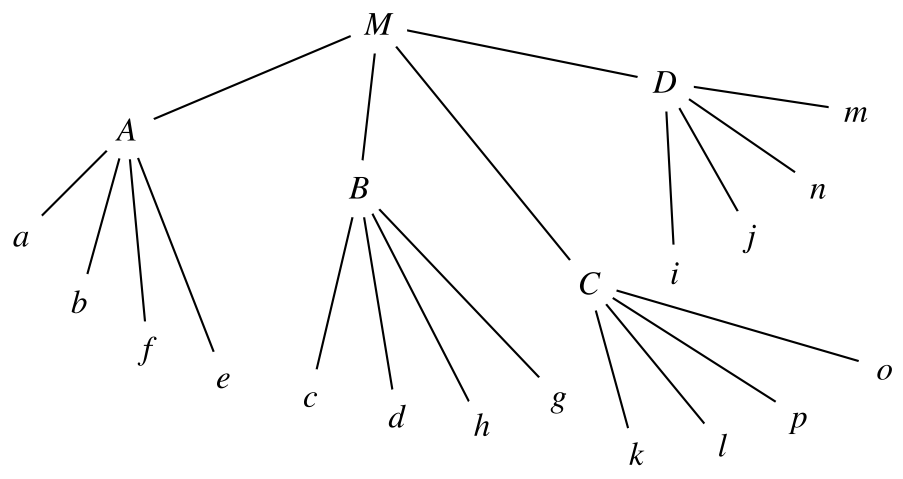


Figure 4.1: A Graph of the Quad Tree Decomposition.

Although we can generalize in this fashion, we will consider the case of scanning only 4 vertices at once, i.e., the case of a quad tree.

4.2 Spectral Decomposition

Given the scanning algorithm that constructs a function on the leaves of a quad tree, the wreath product group can act on the tree, resulting in permutations of the original image. In this section, we describe a wreath product group invariant decomposition from [8]. In addition, we describe an iterative algorithm that speeds up the decomposition.

4.2.1 Multiresolution Decomposition

Consider any tree $\mathscr{T}_{\mathbf{m}}$ given by the n-tuple $\mathbf{m} = (m_0, m_1, m_2, \ldots, m_{n-1})$. Let the set of leaves of $\mathscr{T}_{\mathbf{m}}$ be denoted X, and let $L(X)$ denote the space of all complex valued functions on X. Let G be any wreath product group acting on this tree. Recall that the i^{th} level denotes the set of all vertices in the tree that are of distance i from the root.

First, we can filter $L(X)$ into an ascending chain of subsets of $L(X)$. The following is stated as Theorem 4.4 in [8] without proof, so we provide a proof of this theorem.

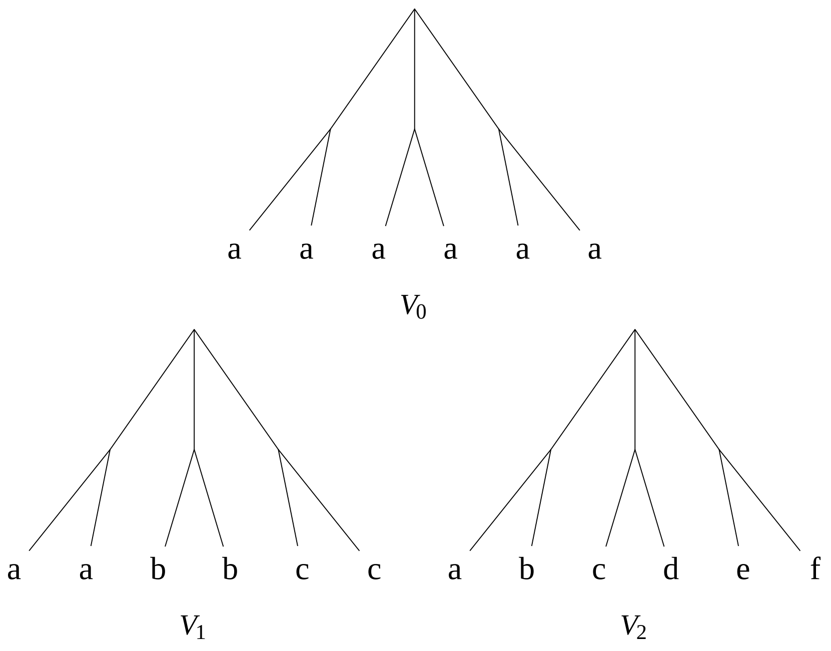


Figure 4.2: A Filtration of the $(3,2)$-tree from Figure 3.2.

Theorem 4.1. *Let $V_i \in L(X)$ be the space of all functions that are constant on each subset of leaves A_i that descend from a common node at level i. Then $V_n = L(X)$, and the subspaces V_i form a filtration of $L(X)$:*

$$0 \subset V_0 \subset V_1 \subset \cdots \subset V_n = L(X).$$

Moreover, each V_i is G-invariant, i.e., for any permutation in G, $g \cdot v \in V_i$.

Proof. First, we show $V_n = L(X)$. By definition, V_n denotes all the functions that are constant on the leaves descending from each vertex that is distance n away from the root. Since the vertices that are distance n away are exactly the leaves of the tree, and since each leaf can take on only one value, any function in $L(X)$ is constant on each leaf, i.e., $V_n = L(X)$.

Note that V_0 is the set of functions that are constant on all leaves of the tree, and as the index increases, we allow for functions of progressively further detail. In fact, for each V_i, we can partition X into $k = \prod_{j=0}^{i-1} m_j$ subsets A_i (with $m_{-1} = 1$), where each subset corresponds to the leaves that descend from one of the k vertices at level i. Then, V_i describes all of the functions that are constant within each of those subsets. For example, we can filter the $(3,2)$-tree from the previous chapter as shown in Figure 4.2.

From this, it's clear that $V_i \subset V_{i+1}$ since V_{i+1} consists of the functions that are constant on a smaller sets of leaves, compared to V_i. Thus the V_i form a filtration simply because they form a chain of subsets.

Finally, to show that each V_i is G-invariant, recall that G provides adjacency preserving transformations of the tree. As a result, no matter how we permute the tree, each A_i will contain the same elements (up to a reordering of the A_i's), because they descend from a common vertex in the tree. Therefore, each set will still have the same constant value; thus the permuted function will remain in V_i. $\square$

This filtration is called a multiresolution filtration ([8]), because of the way that each V_i subsequent allows for further detail. If we consider $L(X)$ as a $\mathbb{C}G$-module, where elements in $\mathbb{C}G$ permute and scale functions defined on the leaves of a tree (such a correspondence is possible because of Proposition 3.1), each V_i is a submodule of $L(X)$ because it is G-invariant and thus closed under permuting and scaling by elements of $\mathbb{C}G$. Furthermore, V_i is a submodule of V_j for all $j \geq i$. Applying Maschke's theorem (Theorem 2.1), we can decompose each V_i into two submodules V_{i-1} and its orthogonal complement. This results in the following theorem:

Theorem 4.2. *For each $\mathbb{C}G$-module V_i, V_{i-1} is a submodule of V_i with orthogonal complement W_i such that*

$$V_i = V_{i-1} \oplus W_i.$$

Now, we need a method to filter constant values for each V_i in order to construct a G-invariant transformation in a multiresolution fashion. Foote et. al. ([8]) constructs such a transform based on the *Radon transform*. This takes the average function value on each subset of leaves A_i (descending from a common node at level i), decomposing the V_i into two components: the average value component and its complement.

Definition 4.2. Let the Radon transform $\mathscr{R}_i : L(X) \to V_i$ be defined by

$$\mathscr{R}_i(f)(x) = \frac{1}{|A_i(x)|} \sum_{x_n \in A_i} f(x_n), \tag{4.1}$$

where $A_i(x)$ is the subset of leaves that contains x and descends from some common node at level i.

Since the Radon transform gives rise to functions in V_i, we can apply Theorem 4.2 to obtain a transformation that filters $L(X)$ (starting from V_n down to V_0). This is the general multiresolution decomposition of $L(X)$ for any wreath product group G.

4.2.2 Recursive Algorithm

Theorem 4.2 gives us a decomposition of $L(X)$, but in a recursive manner. In other words, we decompose $L(X) = V_n$ by starting with V_n and working down to V_0:

$$\begin{aligned} L(X) = V_n &= V_{n-1} \oplus W_n \\ &= (V_{n-2} \oplus W_{n-1}) \oplus W_n \\ &= V_{n-3} \oplus W_{n-2} \oplus W_{n-1} \oplus W_n \\ &= \cdots \\ &= V_0 \oplus \left(\bigoplus_{i=1}^{n} W_i \right). \end{aligned}$$

Thus, we can apply the Radon transform recursively at each level from n to 0, successively averaging pixel values and finding the complement to those values.

Now, let us construct decompositions for a specific group. Consider the case with a quad tree $\mathscr{T}_{(4,4,\ldots,4)}$ and G the iterated wreath product of cyclic groups on four elements, i.e.,

$$G = Z_4 \wr Z_4 \wr \cdots \wr Z_4.$$

The invariant transform for Z_4 is the classical Discrete Fourier Transform (DFT) on four points. Note that Theorem 4.2 only gave a constant-value filtration of the image, and the DFT actually gives rise to three more coefficients. In other words, Theorem 4.2 enables us to break Z_4 down into a one-dimensional constant-value subspace and a three-dimensional space describing variance from the average value. By using the DFT, we have made a specific choice to decompose the three-dimensional space in a certain way. In fact, the DFT breaks V_i into four one-dimensional spaces, where one of the projections is a average-value filtration and the rest describe differences of the data from this average with respect to the group action. Thus $L(X)$ becomes

$$L(X) = V_0 \oplus \left(\bigoplus_{i=1}^{n} W_{i1} \oplus W_{i2} \oplus W_{i3} \right),$$

where each subspace is one-dimensional and invariant under the action of G. Here, W_{ij} corresponds to the j^{th} DFT coefficient performed on the sets A_i. Thus we have a recursive algorithm:

Algorithm 4.3. **(1)** Divide the image into quadrants.

(2) Recursively divide quadrants into sub-quadrants, until we cannot divide the image any further into quadrants that contain more than one element.

(3) Perform the DFT on all 2×2 blocks of pixel values by placing them in a 4×1 vector, where the values are ordered according to our ordering scheme.

(4) Recursively perform DFTs on the first coefficients of these blocks (the average values) until we cannot gather coefficients any longer.

Example 4.2. To visually illustrate this algorithm, consider the 4×4 matrix earlier from Section 4.1 that was divided into four blocks:

$$\begin{pmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{pmatrix},$$

$$A = \begin{pmatrix} a & b \\ e & f \end{pmatrix}, \quad B = \begin{pmatrix} c & d \\ g & h \end{pmatrix}, \quad C = \begin{pmatrix} k & l \\ o & p \end{pmatrix}, \quad D = \begin{pmatrix} i & j \\ m & n \end{pmatrix}.$$

We cannot divide the matrix up any further. Now we can perform the DFT separately on each of these blocks:

$$\mathrm{DFT}_4(A) = \mathrm{DFT}_4 \begin{pmatrix} a \\ b \\ f \\ e \end{pmatrix} = \begin{pmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \\ \alpha_4 \end{pmatrix} \qquad \mathrm{DFT}_4(B) = \mathrm{DFT}_4 \begin{pmatrix} c \\ d \\ h \\ g \end{pmatrix} = \begin{pmatrix} \beta_1 \\ \beta_2 \\ \beta_3 \\ \beta_4 \end{pmatrix}$$

$$\mathrm{DFT}_4(C) = \mathrm{DFT}_4 \begin{pmatrix} k \\ l \\ p \\ o \end{pmatrix} = \begin{pmatrix} \gamma_1 \\ \gamma_2 \\ \gamma_3 \\ \gamma_4 \end{pmatrix} \qquad \mathrm{DFT}_4(D) = \mathrm{DFT}_4 \begin{pmatrix} i \\ j \\ n \\ m \end{pmatrix} = \begin{pmatrix} \delta_1 \\ \delta_2 \\ \delta_3 \\ \delta_4 \end{pmatrix}.$$

Now, we store the DFT coefficients in a new matrix, storing all of the first coefficients in the first quadrant, the second coefficients in the second quadrant, etc.:

$$\left(\begin{array}{cc:cc} \alpha_1 & \beta_1 & \alpha_2 & \beta_2 \\ \delta_1 & \gamma_1 & \delta_2 & \gamma_2 \\ \hdashline \alpha_4 & \beta_4 & \alpha_3 & \beta_3 \\ \delta_4 & \gamma_4 & \delta_3 & \gamma_3 \end{array}\right).$$

We recursively perform the DFT on the first coefficients of the four DFTs; i.e., we perform the DFT on the first quadrant of this matrix:

$$\mathrm{DFT}_4 \begin{pmatrix} \alpha_1 \\ \beta_1 \\ \gamma_1 \\ \delta_1 \end{pmatrix} = \begin{pmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \varepsilon_3 \\ \varepsilon_4 \end{pmatrix}.$$

Now we store this result in place of the first coefficients, so our decomposition matrix becomes:

$$\left(\begin{array}{cc|cc} \varepsilon_1 & \varepsilon_2 & \beta_1 & \beta_2 \\ \varepsilon_4 & \varepsilon_3 & \beta_4 & \beta_3 \\ \hline \delta_1 & \delta_2 & \gamma_1 & \gamma_2 \\ \delta_4 & \delta_3 & \gamma_4 & \gamma_3 \end{array}\right).$$

There is only one average value coefficient left (ε_1), and thus we are done.

4.2.3 Iterative Algorithm

Although we can perform the recursive algorithm, we can speed up the performance of the decomposition by using an iterative method. This method relies on a quick way of calculating the ordering of the pixels in the image. Recall from Section 4.1 that an ordering of the pixel values is

$$\begin{pmatrix} 1 & 2 & 5 & 6 \\ 4 & 3 & 8 & 7 \\ 13 & 14 & 9 & 10 \\ 16 & 15 & 12 & 11 \end{pmatrix}.$$

Simply "marching" through the pixel values and performing DFTs along the way is faster than the recursive method. This is because the recursive method allocates memory for each recursive call and has to divide the image to find the relevant subsets $\{A_i\}$ at each level.

A quick march is obtained as follows. Suppose we would like to scan a $2^n \times 2^n$ image. We start on the upper-left-most pixel and attach ordering numbers to the pixel, starting from 1. In addition to this counter, denoted by i, we have three more variables: the row and column position of the pixel (r, c), and the current level l of the pixel (we begin at the 0th level).

Procedure 4.4. Initialize $i = 0$, $(r, c) = (1, 1)$, and $l = 0$. At each pixel, perform a test on the pixel number i:

(1) Find the largest non-negative integer k such that 4^k divides i.

(2) If $k = l$, we have *completed* a level and must move on to the next level. Thus, reset $(r, c) = (1, 2^l + 1)$, and increment l by 1. However, if $k = n$, then we have reached all pixels in the image and we are done.

(3) If $k \neq l$, compute $m = \big(i/4^k\big) \bmod 4$. Here, k represents a *local level*, and m represents the relative position within that local level. First, we reset our row and column positions back to the first pixel in our local level; this can be accomplished by decrementing r by $2^k - 1$ and leaving c fixed. Now, we change the row and column positions based on the value of m, which has three possible values:

(a) If $m = 1$, then increment c by 2^k.

(b) If $m = 2$, then increment r by 2^k.

(c) If $m = 3$, then decrement c by 2^k.

Notice that the case where $m = 4$ cannot occur, since k was defined to be the *largest* integer such that 4^k divides i.

This procedure gives the row and column position in the image given the index in the quad tree scan, and makes it possible to march through the image quickly. Then, to recursively break down the first coefficients of the computed DFTs, the algorithm runs again on the first coefficients of the spectrum above. We repeat this process until we are left with one number: the average value of the entire image.

4.3 Decompositions with Other Iterated Wreath Products

For the cyclic case, our spectral decomposition involved the recursive use of the DFT. However, the DFT can also be thought of as a change of basis particular to cyclic groups, as in Equation 2.4. Carrying this idea further, we can investigate whether other choices of bases give rise to different decompositions.

Instead of wreath products of Z_4, consider wreath products of the group $Z_2 \times Z_2$. The invariant decomposition in this case corresponds exactly with the Haar wavelet transform. From the DFT matrix obtained in Section 2.2, we can find the DFT matrix for Z_2:

$$\mathrm{DFT}_2 = \frac{1}{2}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}.$$

To find the DFT-like transformation for $Z_2 \times Z_2$, we may simply take the Kronecker product of the DFT matrices from each direct summand. Thus, we arrive at the Haar transform:

$$\mathrm{DFT}_{2\times 2} = \frac{1}{4}\left[\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \otimes \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}\right] = \frac{1}{4}\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{pmatrix}.$$

To apply this transformation, instead of the DFT matrix DFT_4, we can substitute $\mathrm{DFT}_{2\times 2}$. Unlike DFT_4, the Haar transform does not yield complex coefficients, so the decomposition itself is very different.

We can also derive a DFT-like matrix for other iterated wreath product groups as well. We use the term "DFT-like matrix" to refer to the group invariant transformations in this section, because we obtain them considering the permutation representations of the groups. For the remainder of this section, we will derive the DFT-like transformation for all subgroups of S_4 that act transitively on four points: the dihedral group D_8, the alternating group A_4, and finally the symmetric group S_4.

4.3.1 The Dihedral Group

Using the method outlined in Section 2.2, we can use a similar approach to derive the DFT-like matrix for D_8. First, we need to define how D_8 acts on a block of four data points. Recall D_8 has eight elements, where a represents a cyclic shift and b represents a flip:

$$D_8 = \{1, a, a^2, a^3, b, ab, a^2b, a^3b\}.$$

Consider the representation of D_8 acting on a 2×2 matrix by way of permutation, where

$$a\cdot\begin{pmatrix}1 & 2\\ 4 & 3\end{pmatrix} = \begin{pmatrix}4 & 1\\ 3 & 2\end{pmatrix}, \qquad b\cdot\begin{pmatrix}1 & 2\\ 4 & 3\end{pmatrix} = \begin{pmatrix}4 & 3\\ 1 & 2\end{pmatrix}.$$

Then, we can realize each element of D_8 as a permutation matrix; for example

$$a = \begin{pmatrix}0 & 0 & 0 & 1\\ 1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0\end{pmatrix}, \qquad b = \begin{pmatrix}0 & 0 & 0 & 1\\ 0 & 0 & 1 & 0\\ 0 & 1 & 0 & 0\\ 1 & 0 & 0 & 0\end{pmatrix}.$$

Denote this permutation representation by ρ. We need to first find the irreducible representations within ρ. This can be accomplished by finding the characters for each conjugacy class in D_8 and computing the inner product of this character with the irreducible characters of D_8. This way we are able to determine which irreducible representations are contained in ρ.

Recall from Section 2.1 that the character is simply the trace of a matrix. A permutation matrix has a 1 in the diagonal if an element is fixed, and 0 otherwise. Therefore, we can calculate the character table of our representation by counting the number of elements fixed as a result of applying the permutation. Thus, we

g_i	1	(a,a^3)	a^2	(b,a^2b)	(ab,a^3b)
$\lvert C_G(g_i)\rvert$	8	4	8	4	4
χ_1	1	1	1	1	1
χ_2	1	1	1	-1	-1
χ_3	1	1	-1	1	-1
χ_4	1	1	-1	-1	1
χ_5	2	-2	0	0	0

Table 4.1: Irreducible Characters for D_8

obtain the following character table:

Conjugacy Classes	1	(a,a^3)	a^2	(b,a^2b)	(ab,a^3b)
ρ	4	0	0	0	2

Now, we compute the inner product of ρ and χ_i, according to Equation 2.1 and referring to Table 4.1 obtained from [11]:

$$\langle \theta, \psi \rangle = \sum_{i=1}^{l} \frac{\chi(g_i)\overline{\psi(g_i)}}{|C_G(g_i)|},$$

$$\begin{aligned}
\langle \chi_1, \rho \rangle &= \frac{4(1)}{8} + \frac{0(1)}{4} + \frac{0(1)}{8} + \frac{0(1)}{4} + \frac{2(1)}{4} = 1, \\
\langle \chi_2, \rho \rangle &= \frac{4(1)}{8} - \frac{2(1)}{4} = 0, \\
\langle \chi_3, \rho \rangle &= 0, \\
\langle \chi_4, \rho \rangle &= 1, \\
\langle \chi_5, \rho \rangle &= 1.
\end{aligned}$$

Thus χ_1, χ_4, χ_5 are the irreducible representations in ρ, each with multiplicity 1. Now we find the projection matrices for these irreducible representations by computing their corresponding idempotents, using the formula from Section 2.1:

$$e_i = \frac{\chi_i(1)}{|G|} \sum_{g \in G} \chi_i(g^{-1})g,$$

$$e_1 = \frac{1}{8}\left(\sum_{g\in G} g\right) = \frac{1}{4}\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix},$$

$$\begin{aligned} e_4 &= \frac{1}{8}\left(\sum_{g\in G} \chi_4(g^{-1})g\right) \\ &= \frac{1}{8}\left(1(1) - 1(a) + 1(a^2) - 1(a^3) - 1(b) + 1(ab) - 1(a^2b) + 1(a^3b)\right) \\ &= \frac{1}{4}\begin{pmatrix} 1 & -1 & 1 & -1 \\ -1 & 1 & -1 & 1 \\ 1 & -1 & 1 & -1 \\ -1 & 1 & -1 & 1 \end{pmatrix}, \end{aligned}$$

$$e_5 = \frac{1}{2}\begin{pmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \\ -1 & 0 & 1 & 0 \\ 0 & -1 & 0 & 1 \end{pmatrix}.$$

From this, we can pick out the first $\chi_i(1)$ rows from each idempotent (since the dimension of the projection of χ_i is $\chi_i(1)$), and we obtain the DFT-like matrix for D_8:

$$\frac{1}{4}\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 2 & 0 & -2 & 0 \\ 0 & 2 & 0 & -2 \end{pmatrix}.$$

4.3.2 The Alternating and Symmetric Groups

We follow exactly the same procedure for A_4. A_4 is the group of even transpositions and has exactly 12 elements:

$$\begin{aligned} A_4 = \{&1, (12)(34), (13)(24), (14)(23), \\ &(123), (124), (134), (234), \\ &(132), (142), (143), (243)\}. \end{aligned}$$

g_i	1	$(12)(34)$	(123)	(132)
$\|C_G(g_i)\|$	12	4	3	3
χ_1	1	1	1	1
χ_2	1	1	ω	ω^2
χ_3	1	1	ω^2	ω
χ_4	3	-1	0	0

Table 4.2: Irreducible Characters for A_4

The action here is more explicit than the case for D_8: for both A_4 and S_4, we directly apply the permutation to the four data points. For example,

$$(12)(34)\cdot\begin{pmatrix}1 & 2\\ 4 & 3\end{pmatrix} = \begin{pmatrix}2 & 1\\ 3 & 4\end{pmatrix}, \qquad (123)\cdot\begin{pmatrix}1 & 2\\ 4 & 3\end{pmatrix} = \begin{pmatrix}2 & 3\\ 4 & 1\end{pmatrix}.$$

Again, we can realize each element of A_4 as a permutation matrix, for instance

$$(12)(34) = \begin{pmatrix}0 & 1 & 0 & 0\\ 1 & 0 & 0 & 0\\ 0 & 0 & 0 & 1\\ 0 & 0 & 1 & 0\end{pmatrix}, \qquad (123) = \begin{pmatrix}0 & 0 & 1 & 0\\ 1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 0 & 1\end{pmatrix}.$$

Now we calculate the character table of the permutation representation of A_4 by finding the number of elements fixed:

g_i	1	$(12)(34)$	(123)	(132)
ρ	4	0	1	1

Computing inner products using Table 4.2 (where ω is a primitive 3rd root of unity), we find that χ_1 and χ_4 are in ρ, and thus ρ projects the four data points into a 1-dimensional subspace and a 3-dimensional subspace, where the idempotents

g_i	1	(12)	(123)	(12)(34)	(1234)
$\|C_G(g_i)\|$	24	4	3	8	4
χ_1	1	1	1	1	1
χ_2	1	-1	1	1	-1
χ_3	2	0	-1	2	0
χ_4	3	1	0	-1	-1
χ_5	3	-1	0	-1	1

Table 4.3: Irreducible Characters for S_4

are:

$$e_1 = \frac{1}{4}\left(\sum_{g\in G} g\right) = \frac{1}{4}\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix},$$

$$e_4 = \frac{\chi_4(1)}{12}\left(\sum_{g\in G} \chi_4(g^{-1})g\right) = \frac{1}{4}\begin{pmatrix} 3 & -1 & -1 & -1 \\ -1 & 3 & -1 & -1 \\ -1 & -1 & 3 & -1 \\ -1 & -1 & -1 & 3 \end{pmatrix}.$$

Therefore, a DFT-like matrix is:

$$\frac{1}{4}\begin{pmatrix} 1 & 1 & 1 & 1 \\ -1 & 3 & -1 & -1 \\ -1 & -1 & 3 & -1 \\ -1 & -1 & -1 & 3 \end{pmatrix}.$$

The case for the symmetric group S_4 is similar. We exhibit the character tables, the idempotents, and the resulting DFT matrix. The character table for the permutation representation is:

g_i	1	(12)	(123)	(12)(34)	(1234)
ρ	4	2	1	0	0

We see from Table 4.3 that $\rho = \chi_1 + \chi_4$. The idempotents for the irreducible representations χ_1 and χ_4 are:

$$e_1 = \frac{1}{24}\left(\sum_{g\in G} g\right) = \frac{1}{4}\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix},$$

$$e_4 = \frac{\chi_4(1)}{24}\left(\sum_{g\in G} \chi_4(g^{-1})g\right) = \frac{1}{4}\begin{pmatrix} 3 & -1 & -1 & -1 \\ -1 & 3 & -1 & -1 \\ -1 & -1 & 3 & -1 \\ -1 & -1 & -1 & 3 \end{pmatrix}.$$

And the resulting DFT-like matrix is exactly the same as the alternating group:

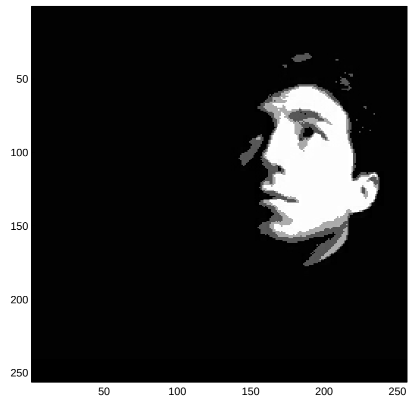
$$\frac{1}{4}\begin{pmatrix} 1 & 1 & 1 & 1 \\ -1 & 3 & -1 & -1 \\ -1 & -1 & 3 & -1 \\ -1 & -1 & -1 & 3 \end{pmatrix}.$$

4.4 Examples

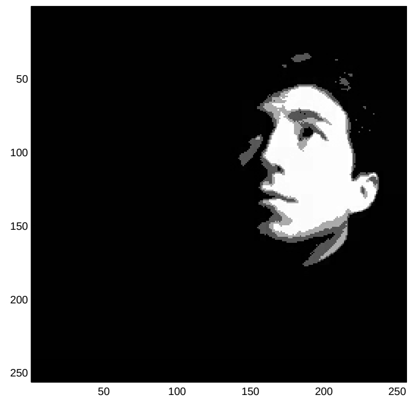
In Figure 4.3, we ran the decomposition algorithm on a black and white photo. In addition to the decomposition, we constructed an inverse decomposition algorithm in order to recover the original information from the spectrum. The maximum error (of a point-by-point pixel value difference) in this case was the negligible quantity 6.6613×10^{-16}. The colors of the spectrum amplitude and phase were inverted for clarity (white means low magnitude, darker means higher magnitude).

Notice that in the spectrum, the image seems to be repeated four times, and the first quadrant appears to be subdivided more and more. This is due to our method of storing the spectrum. We store the spectrum by storing all the first coefficients resulting from the DFT in the first quadrant, the second coefficients in the second quadrant, and so forth. Since we recursively compute decompositions of the first coefficients, the first quadrant is subdivided into subsequent recursive decompositions.

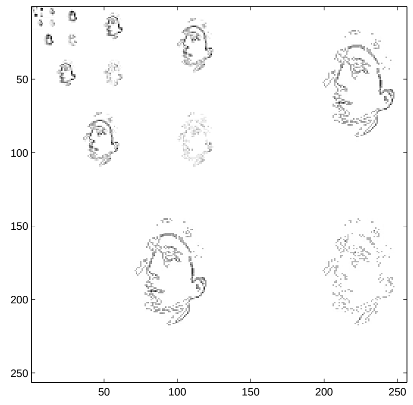
We illustrate the idea of progressive inverse decomposition in Figure 4.4 with a color photo. In this figure, we only use a portion of the spectrum to reconstruct the image. The first image (top left) represents the reconstruction with only four pixels in the spectrum (the top left four). Subsequent images use higher "levels" of pixels in the spectrum, i.e., the top left $16, 64, 256, 1024, \ldots$ pixels. Each of these



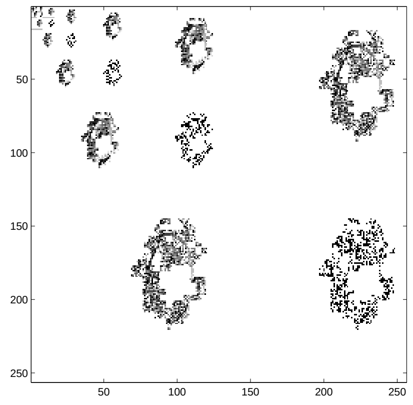
(a) Original Image



(b) Recovered Image



(c) Spectrum Amplitude



(d) Spectrum Phase

Figure 4.3: Decomposition

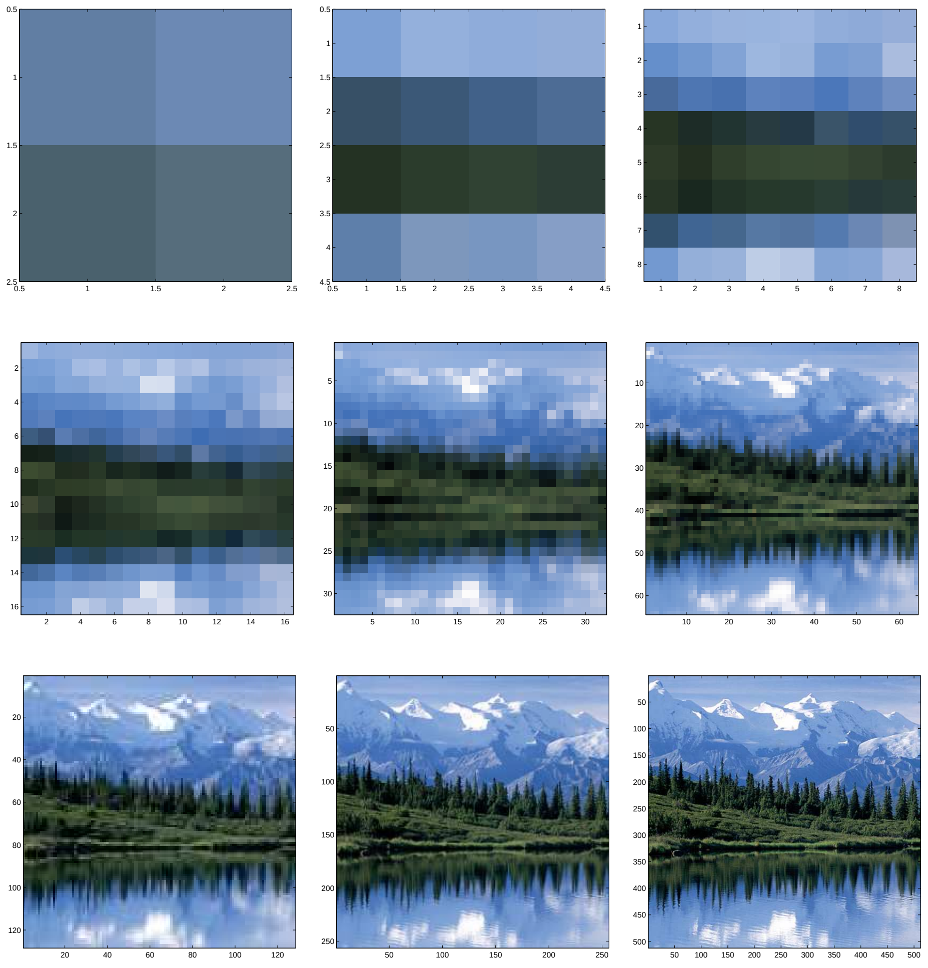


Figure 4.4: Multiresolution

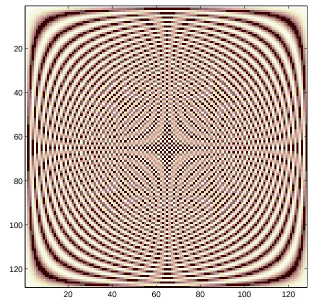
levels correspond to traversing the quad tree spectrum only up to a certain depth. Notice here that the quad-tree scan can be employed to give a progressive stream of reconstructions, capturing finer and finer details as more and more information is available.

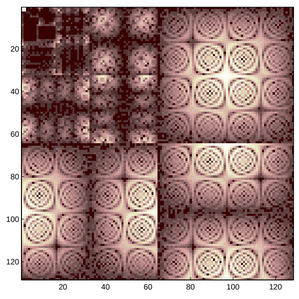
Finally, we can compare the different transformations of the same image to build an intuition of what these transformations do. We ran the algorithm on three images: geometry, sinsin, and tartan, shown in Figure 4.5, Figure 4.6, and Figure 4.7. These images were obtained from MATLAB's Wavelet Toolbox, and they are interesting because they contain regular patterns and symmetries.

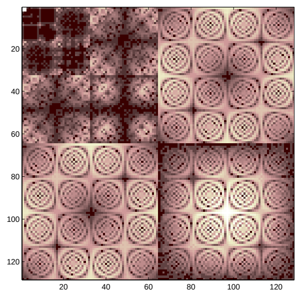
Overall, the Haar transform and the Fourier transform yield similar results. Notice in the dihedral, alternating, and symmetric transforms that some of the projections are two or three dimensional. In the dihedral case, the third and fourth quadrants comprise a two dimensional projection, and in the alternating/symmetric case the second, third, and fourth quadrants represent a three dimensional projection. The darker parts of the spectra represent places where the variance component (the orthogonal complement to the average) is small. This difference component is based on how the image differs from the down-sampled average image, depending on the particular symmetry.

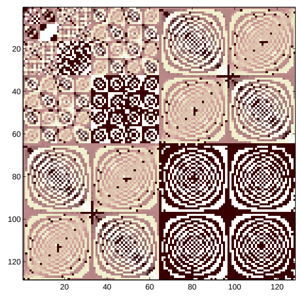
Examining the geometry example more closely, notice that around the far edges and corners of the original image, the pixel values are nearly the same. This is reflected in the Haar, Fourier, and dihedral spectrum, where there is a quadrant that has darker parts around the edges of the quadrant. In addition, the Haar transform has an "H" shaped quadrant and an "I" shaped quadrant. This is reminiscent of the vertical and horizontal flips of $Z_2 \times Z_2$, and since the original image is symmetric horizontally and vertically down the midpoint, the difference component is small in those areas where there are greater amounts of symmetry. For the Fourier transform, there seem to be lower (darker) values where there is rotational symmetry (cyclic shifts); and for the dihedral transform, there are dark diagonals going across two of the quadrants, probably representing shift-and-flip symmetry. The alternating transform does not seem to have as many darker parts, because we are decomposing with respect to a greater number of symmetries, which is harder to satisfy. Finally, the symmetric transform is exactly the same as the alternating transform, since the matrix we derived was exactly the same. The sinsin example seems to yield similar results in comparison to the geometry example, in terms of which symmetries are encoded as darker parts of the spectrum. The Fourier phase of the sinsin example is quite strange, because it appears almost three-dimensional.

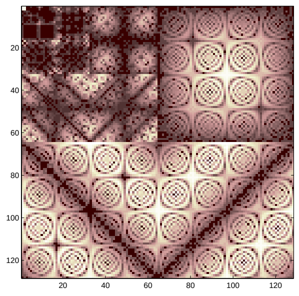
Comparing these different transforms, we see that some transforms are better than others in encapsulating more information using less space, based upon the particular symmetries that the group presents. This is especially important when compressing data using these transforms; the darker the image, the better

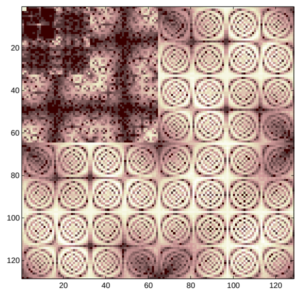

(a) Original Image


(b) Haar Transform


(c) Fourier Amplitude


(d) Fourier Phase


(e) Dihedral Transform


(f) Alternating Transform

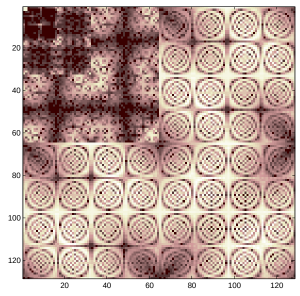
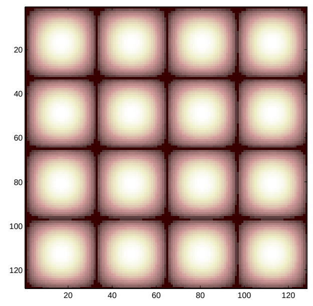
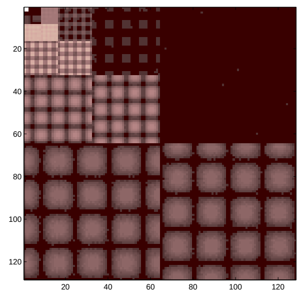
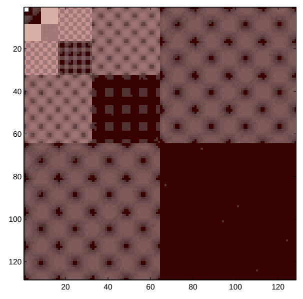
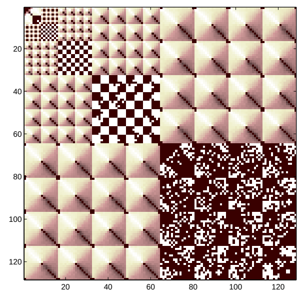

(g) Symmetric Transform

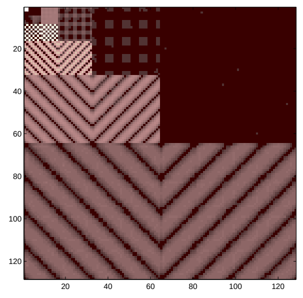
Figure 4.5: Decompositions of the Geometry Image

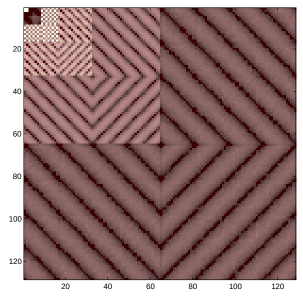

(a) Original Image


(b) Haar Transform


(c) Fourier Amplitude


(d) Fourier Phase


(e) Dihedral Transform


(f) Alternating Transform

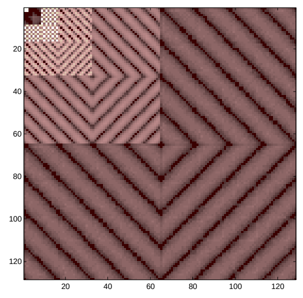
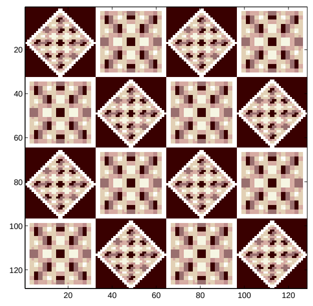
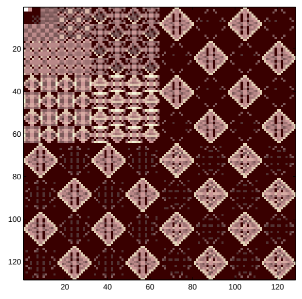
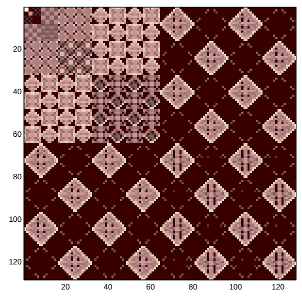
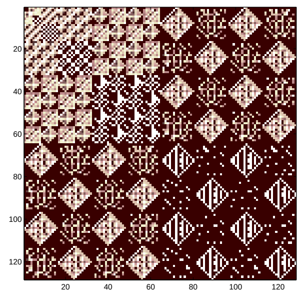

(g) Symmetric Transform

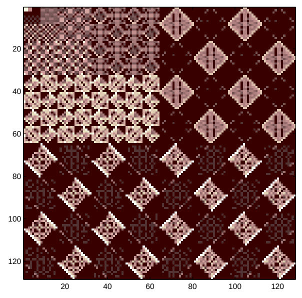
Figure 4.6: Decompositions of the Sinsin Image

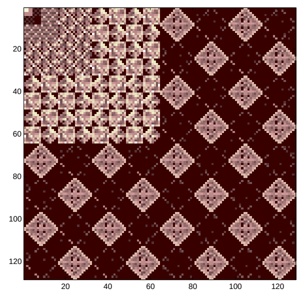

(a) Original Image


(b) Haar Transform


(c) Fourier Amplitude


(d) Fourier Phase


(e) Dihedral Transform


(f) Alternating Transform

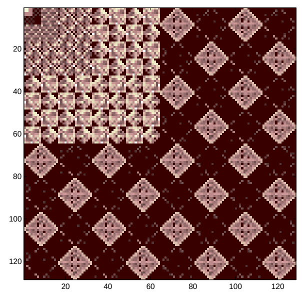

(g) Symmetric Transform

Figure 4.7: Decompositions of the Tartan Image

the compression rate. We will explore compression and other applications of these decompositions in the next chapter.

Chapter 5

Image Processing with Wreath Product Groups

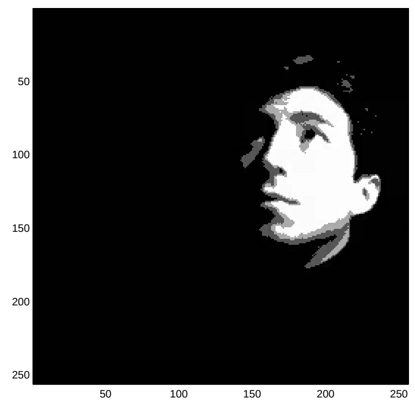
In this chapter we investigate applications of the wreath product decomposition from the last chapter. We will consider three main applications: compression, convolution, and de-noising. By discarding small values in the image spectrum, we can achieve image compression. In addition, we can construct filters that change the image spectrum in certain ways in order to achieve certain affects. A high-pass filter, for example, would be able to filter out edges in the data. These and more approaches are described with examples in this chapter.

5.1 Compression

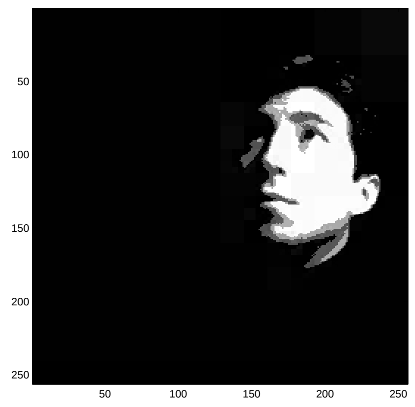
5.1.1 Near-Zero Thresholding

In the spectrum of Figure 4.3, the information is very sparse, since most of the spectrum is white (meaning that the Fourier coefficients are zero at those locations). We can use this to our advantage to achieve compression. For a given spectrum, we can discard small values by setting them equal to zero. The results of this method are in Figure 5.1. Notice that this image results in a high level of compression, due to the sparse amount of information in the original image. The maximum and minimum error (denoted by Δf) increases as more information is discarded in the spectrum, as expected. Also, the average error of the image seems to increase as well.

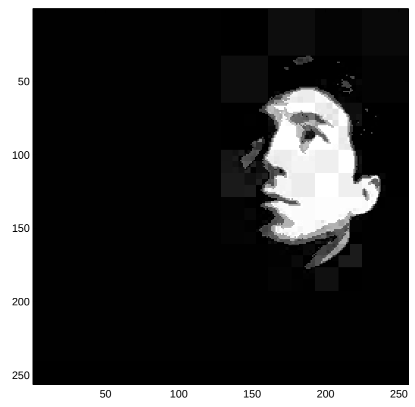
Now compare this to Figure 5.2. We see here that parts of the image with finer detail are left relatively untouched, while sections with less detail are adversely affected. This is because the places with fine detail have larger coefficients in the



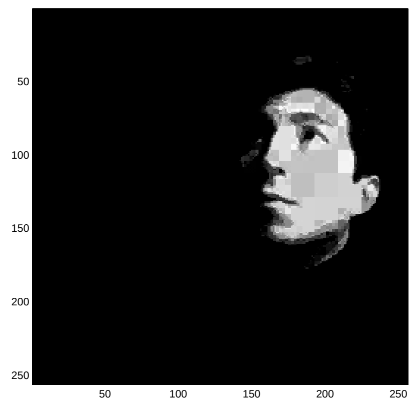
(a) Amplitude > 0.0001
90.5273% Compressed
$-4.4409 \times 10^{-16} \leq \Delta f \leq 6.6613 \times 10^{-16}$
1.4494×10^{-17} Average Δf



(b) Amplitude > 0.03
93.1854% Compressed
$-0.078163 \leq \Delta f \leq 0.086008$
-0.00019505 Average Δf

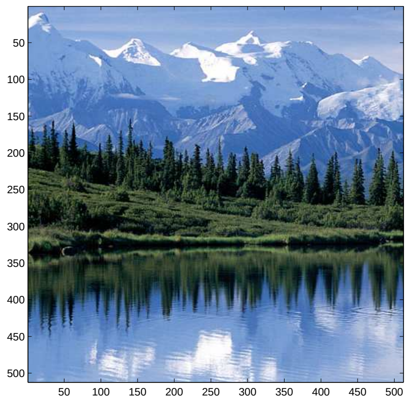


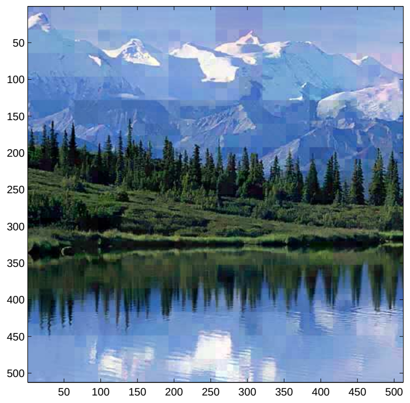
(c) Amplitude > 0.06
93.4128% Compressed
$-0.17512 \leq \Delta f \leq 0.28371$
-0.0017895 Average Δf

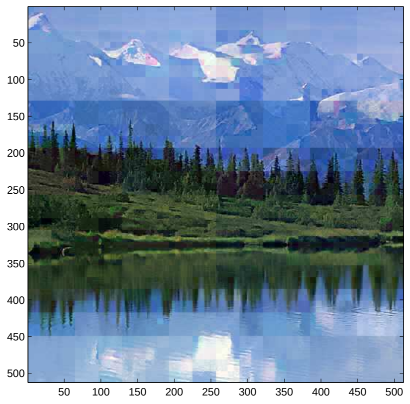


(d) Amplitude > 0.09
94.5633% Compressed
$-0.28052 \leq \Delta f \leq 0.56501$
0.024224 Average Δf

Figure 5.1: Compression


(a) Amplitude > 0.0001
5.5216% Compressed
$-9.1912\times 10^{-5} \leq \Delta f \leq 0.00013021$
2.24×10^{-9} Average Δf


(b) Amplitude > 0.03
49.2598% Compressed
$-0.18411 \leq \Delta f \leq 0.20542$
-0.00013933 Average Δf


(c) Amplitude > 0.06
55.0279% Compressed
$-0.32995 \leq \Delta f \leq 0.45321$
-3.8713×10^{-5} Average Δf

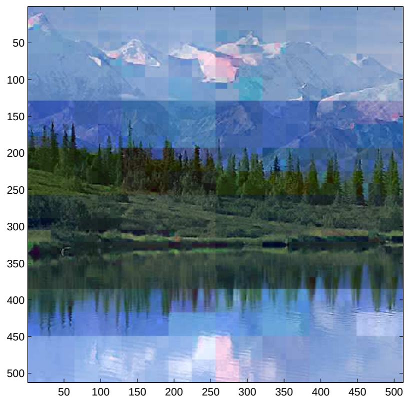

(d) Amplitude > 0.09
56.1526% Compressed
$-0.41145 \leq \Delta f \leq 0.58572$
-1.1713×10^{-5} Average Δf

Figure 5.2: More Compression

spectrum (due to high local variance of the data), and subsequently the coefficients are not set to zero. However, we have discarded relatively small coefficients, causing errors that propagate through each level-of-detail subset in the spectrum. The first key difference is that less information is compressed. This is because the information in the image varies quite rapidly for adjoining pixels. Thus, the average amplitude of the spectrum is much higher than in Figure 5.1. There is another strange difference between the two experiments, and that is the average error. The average error does not get very high, even when we throw away amplitudes that are less than or equal to 0.09. In addition, the average error and the compression rate seems to be correlated, because the overall brightness of the picture increases as image is more compressed. However, this seems to be image-dependent, since in Figure 5.1 the average error is small with $\approx$ 90% compression.

5.1.2 Sparsity-Norm Balance Thresholding

With the near-zero thresholding method, it is hard to determine exactly where the optimal threshold is. If possible, we would like to construct an algorithm that guesses an appropriate threshold for an image. One such method is the sparsity-norm balancing method. Under this heuristic, we test successive thresholds and calculate the norm of the spectrum after each threshold. The p-norm of a vector $\mathbf{x}$ is given by ([14]):

$$|\mathbf{x}|_p = \left(\sum_i |x_i|^p \right)^{1/p}.$$

From this, we can define the p-norm of a matrix A ([14]):

$$\|A\|_p = \max_{\mathbf{x} \text{ s.t. } |\mathbf{x}|_p = 1} \|A\mathbf{x}\|_p .$$

It turns out that computing the matrix norm is quite difficult for values of $p > 1$. We instead choose a simpler method, modeled after the vector norm:

$$\|A\| = \left(\sum_i \sum_j |a_{ij}|^2 \right)^{1/2}.$$

Our goal in calculating the norm of the spectrum is to balance thresholding with loss in image quality. The metric we use to measure loss of image quality is the norm, which actually calculates how much total “energy” we lose after compression. Now, if we were to use the matrix p-norm, we would use the 2-norm $\|A\|_2$, which is the square root of the maximum eigenvalue of $A^H A$ where A^H is the conjugate transpose. However, for large matrices, the norm is difficult to compute.

Since we just want to calculate how much total energy is lost in the image, the simpler heuristic works well and is much easier to compute To find the right balance, we calculated the norm of the Fourier spectrum of the image for uniformly distributed threshold values between 0 and 1. For each threshold, we calculated the ratio

$$\frac{\text{norm of the compressed spectrum}}{\text{norm of the original spectrum}}.$$

When the threshold is 0, the percent number of zeros should be close to 0%, while the percent norm is 100%. However, when the threshold is at 1, we would expect the percent zeros to be 100%, while the percent norm is 0%. Thus, the curves of the two quantities intersect at some point, and we set the global threshold to be at this intersection. This method, borrowed from MATLAB's Wavelet Toolbox, is called *balanced sparsity-norm.*

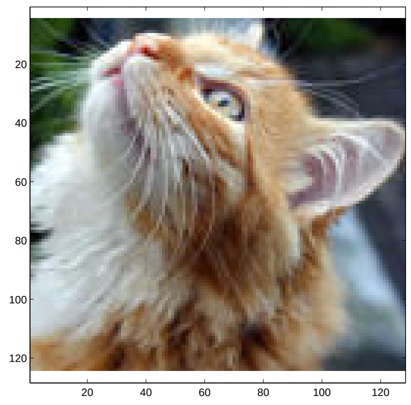
In addition, we recognize that a loss of coefficients at the lower levels of the decomposition (the smaller quadrants in the upper left corner) makes a much greater impact on the quality of the image than the coefficients at higher levels of the decomposition. A solution to this problem is to only decompose the image to a certain level, instead of decomposing the image down to the last pixel. In fact, depending on the recursive depth of the image spectrum, one obtains drastically different compression rates. In the previous examples of near-zero thresholding, we decomposed the image up to the full recursive depth, but here we decompose the image only up to the third recursive level.

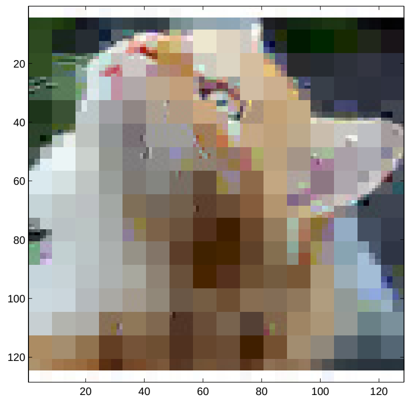
We present examples of the described method in Figure 5.3 and Figure 5.4. We performed the decomposition (only up to level 3) and ran the sparsity-norm balance routine. In most cases, the threshold seemed to be too high in terms of preserving image quality. To decrease the threshold in proportion to the calculated threshold, for fixed c and threshold value t we calculated the *square-root balance sparsity-norm* threshold, $\sqrt{(t/c)}/c$ (also from MATLAB's Wavelet Toolbox). In our case, $c = 128$. In both cases, the square-root balance proves to be the better trade off between image quality and compression rate. Although the balance threshold gives us phenomenal compression rates (better than 1:20 compression), the image quality is sacrificed.

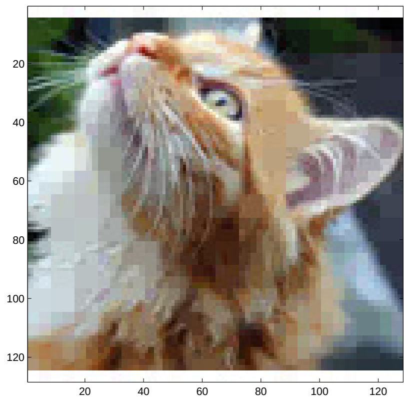
5.2 Convolution

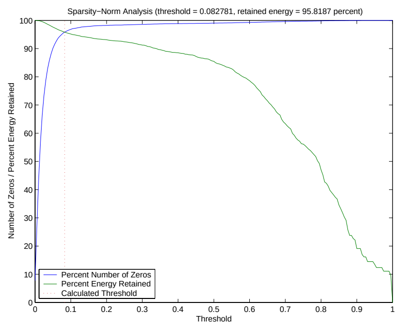
Let G be a finite group acting on a set $X = \{x_0, x_1, \ldots, x_{n-1}\}$. The cyclic convolution of signals f and h (on X) is defined by ([18]):

$$(f * h)(x_k) = \sum_{i=0}^{n-1} f(x_i)h(x_{k-i}) = \sum_{i=0}^{n-1} f(\sigma^i x_0)h(\sigma^{-i} x_k),$$

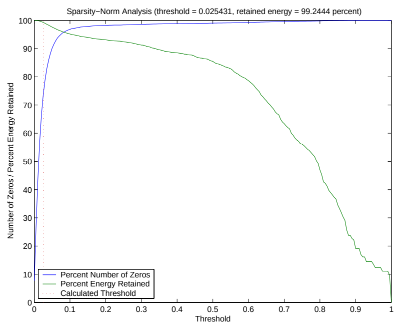

(a) Original Image


(b) Balance Sparsity-Norm
95.7947% Compressed


(c) Square-root Balance Sparsity-Norm
73.8037% Compressed

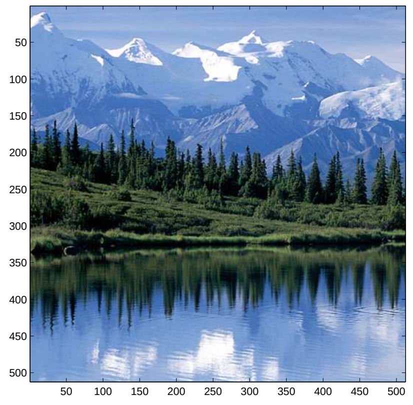


(d) Sparsity-Norm Graph

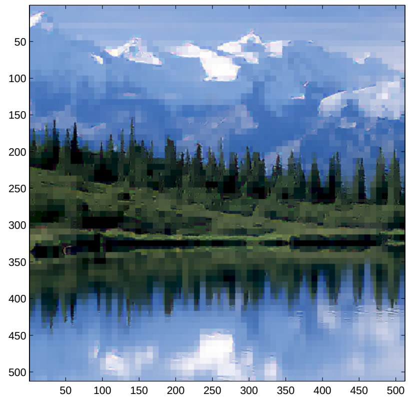


(e) Sparsity-Norm Graph

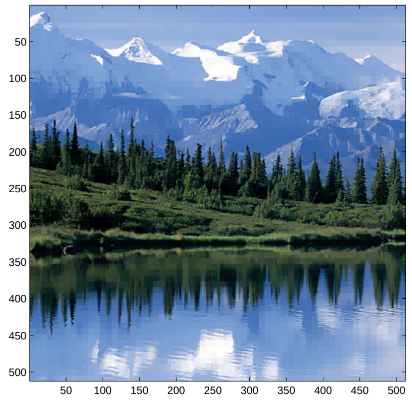
Figure 5.3: Balance Sparsity-Norm with the Cat Image



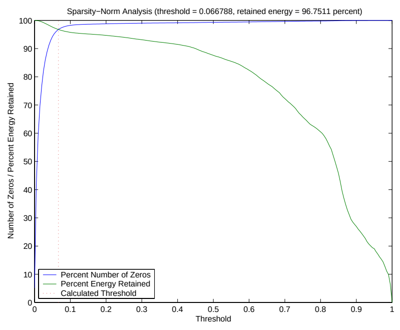
(a) Original Image



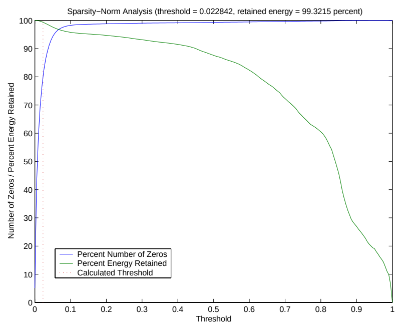
(b) Balance Sparsity-Norm
96.7696% Compressed



(c) Square-root Balance Sparsity-Norm
79.9764% Compressed



(d) Sparsity-Norm Graph



(e) Sparsity-Norm Graph

Figure 5.4: Balance Sparsity-Norm with the Mountain Lake Image

where σ cyclically permutes X, and the underlying group G is the cyclic group Z_n generated by σ. We can generalize this formula for an arbitrary group G acting on X. Again, for signals f and h defined on X, their group-based convolution is ([18]):

$$(f*h)(x_k) = \frac{|X|}{|G|}\sum_{\beta\in G} f(\beta x_0)h(\beta^{-1}x_k).$$

Mirchandani et. al. ([18]) develop a convolution for iterated wreath product cyclic groups based on cyclic convolution. They describe both convolution in the image spectrum and in the signal space as well. We will state their important results: Theorem 2.5 and Theorem 2.6 in [18].

Theorem 5.1. *For signals f and h defined on the leaves of an SHRT, let $Q(f)$ and $Q(h)$ denote their spectra, respectively. Then the convolution of these signals may be computed in the spectral domain by multiplying each entry in each nested grid irreducible sub-matrix of $Q(f)$ by the upper left-hand entry of the corresponding block matrix of $Q(h)$.*

To explain this in more familiar terms, recall example Example 4.2 where we demonstrated how to explicitly construct the image spectrum. The matrix we obtained in that case was:

$$\left(\begin{array}{cc:cc} \epsilon_1 & \epsilon_2 & \beta_1 & \beta_2 \\ \epsilon_4 & \epsilon_3 & \beta_4 & \beta_3 \\ \hdashline \delta_1 & \delta_2 & \gamma_1 & \gamma_2 \\ \delta_4 & \delta_3 & \gamma_4 & \gamma_3 \end{array}\right).$$

The nested grid irreducible submatrices of this spectrum refer to each and every dotted square block in the spectrum. Now suppose we have another spectrum:

$$\left(\begin{array}{cc:cc} e_1 & e_2 & b_1 & b_2 \\ e_4 & e_3 & b_4 & b_3 \\ \hdashline d_1 & d_2 & c_1 & c_2 \\ d_4 & d_3 & c_4 & c_3 \end{array}\right).$$

Suppose that the first spectrum is $Q(f)$ and the second $Q(h)$. Then the convolution of these two signals is obtained by simply scaling each dotted block in $Q(f)$ with the upper left-hand entry of the corresponding dotted block in $Q(h)$:

$$\left(\begin{array}{cc:cc} \varepsilon_1 & \varepsilon_2 & \beta_1 & \beta_2 \\ \varepsilon_4 & \varepsilon_3 & \beta_4 & \beta_3 \\ \hdashline \delta_1 & \delta_2 & \gamma_1 & \gamma_2 \\ \delta_4 & \delta_3 & \gamma_4 & \gamma_3 \end{array}\right) * \left(\begin{array}{cc:cc} e_1 & e_2 & b_1 & b_2 \\ e_4 & e_3 & b_4 & b_3 \\ \hdashline d_1 & d_2 & c_1 & c_2 \\ d_4 & d_3 & c_4 & c_3 \end{array}\right)$$

$$= \left(\begin{array}{cc:cc} e_1\varepsilon_1 & e_1\varepsilon_2 & b_1\beta_1 & b_1\beta_2 \\ e_1\varepsilon_4 & e_1\varepsilon_3 & b_1\beta_4 & b_1\beta_3 \\ \hdashline d_1\delta_1 & d_1\delta_2 & c_1\gamma_1 & c_1\gamma_2 \\ d_1\delta_4 & d_1\delta_3 & c_1\gamma_4 & c_1\gamma_3 \end{array}\right).$$

Intuitively, this multiplication seems to be a sort of selective or graduated thresholding, since we are scaling each distinct block separately. We now proceed to describe convolution in the signal space itself, for the case of a quad-tree of height n and for the iterated wreath product cyclic group $Z_4 \wr Z_4 \wr \cdots \wr Z_4$ with the following theorem ([18]):

Theorem 5.2. *Let $X = \{x_0, x_1, \ldots, x_{4^n-1}\}$ be the set of leaves of the quad tree $\mathcal{Q}(n) = \mathcal{T}_{(4,4,\ldots,4)}$ acted upon by the group $G = Z_4 \wr Z_4 \wr \cdots \wr Z_4$. Let G_0 be the subgroup of G fixing the leaf x_0. For any index i, let $h_i \in L(X)$ be the unit impulse delta function supported at x_i; i.e. $h_i(x_j) = \delta_{ij}$ for $0 \le i \le 4^n - 1$. Define f_0 to be h_0. Then the following hold:*

(1) For any index $i > 0$, the convolution $f_0 * h_i$ is the function defined by

$$(f_0 * h_i)(x_k) = \begin{cases} 0 & \textit{if } x_i \textit{ and } x_k \textit{ do not lie in a common subtree} \\ & \quad \textit{not containing } x_0 \\ 4^{s-n} & \textit{where the largest subtree containing } x_i \textit{ and } x_k \\ & \quad \textit{but not } x_0 \textit{ has root on level } s \end{cases}$$

where $f_0 * h_0 = f_0$.

(2) Let α_j be an element of G such that $\alpha_j x_0 = x_j$. Then

$$(f_i * h_i)(x_k) = \sum_{j=0}^{4^n-1} f(x_j)\alpha_j(f_0 * h_i)(x_k) = \sum_{j=0}^{4^n-1} f(x_j)(f_0 * h_i)(\alpha_j{}^{-1} x_k).$$

(3) The linearity of convolution in the second variable reduces the computation of $f * h$ for an arbitrary h to linear combinations of convolutions of the types in parts (1) and (2).

To provide a rough sketch of the proof of this theorem, part (1) of this theorem comes from the group-based convolution formula defined at the beginning of this section, and Parts (2) and (3) follow as a result of some other properties of group-based convolution. Part (2) comes mainly from the fact that $\alpha(f * h) = (\alpha f) * h$ for all $f, h \in L(X)$ and all $\alpha \in G$, then showing that $h_i({\alpha_j}^{-1}x_k) = \alpha_j(f_0 * h_i)(x_k)$. Part (3) results from the property that convolution in $L(X)$ is bilinear and associative. For more details of the proof of this and the previous theorem, consult [18].

To interpret this convolution, we would prefer to think of it as scaling each quadrant in the decomposition separately. Now, the question is, how do we apply convolution for image processing? We present two applications of convolution: edge detection and de-noising.

5.2.1 Edge Detection and De-noising

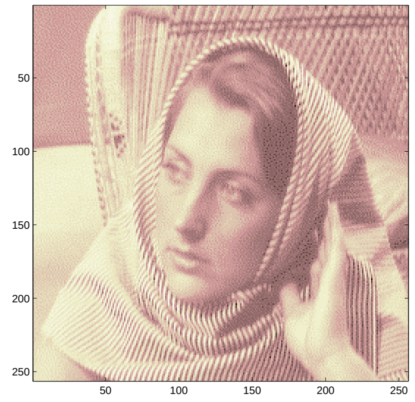
The idea behind detecting edges in the image is to filter out all the details in the image except for where the pixel values sharply change. This is equivalent to using a high-pass filter in order to pick out sudden changes. To accomplish this, we selectively filled in the spectrum with zeros (represented by black quadrants), including the upper left-most corner, which contains a small, average-value filtration version of the original image.

We obtained two more test images from MATLAB's Wavelet Toolbox: a picture of a woman and a noisy closeup. Figure 5.5 is an example where some quadrants were filled with zeros. The edges in the image are picked up remarkably well in the filtered image. In Figure 5.6, the exact same filtration was performed in the mountain lake image. However, this image was a color image with all three RGB channels, and as a result of filtering out the upper left-most corner, too much pixel intensity was lost. To make up for that loss, a constant value of 1 was inserted in the upper left-most corner. Again, the filtered image seems to have picked up the edges quite well.

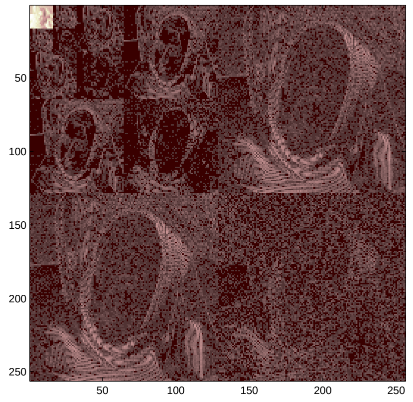
To de-noise an image, instead of trying to retain the highest details as we have done with edge detection, we remove the highest details in order to smooth the image out. In Figure 5.7 and Figure 5.8, we removed two of the three detail quadrants at the highest recursive level (largest quadrants).

5.3 Further applications

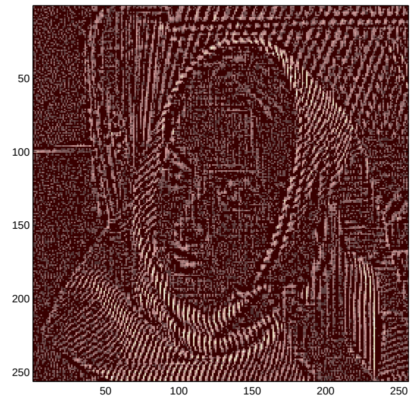
In addition to edge detection and de-noising, a variety of other image processing techniques can be employed with our decomposition. Mirchandani et. al. ([18]) describe a method for using the image spectrum to determine image similarity. In



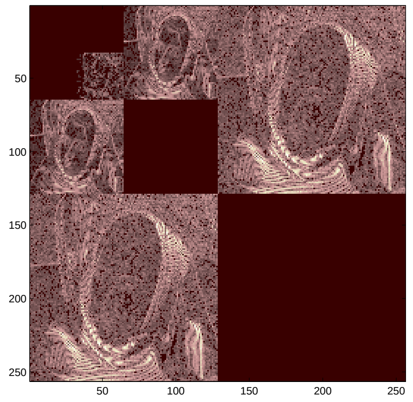
(a) Original Image



(b) Original Spectrum

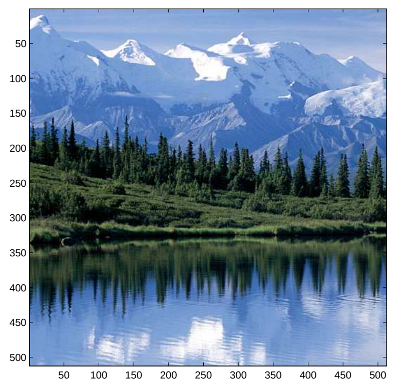


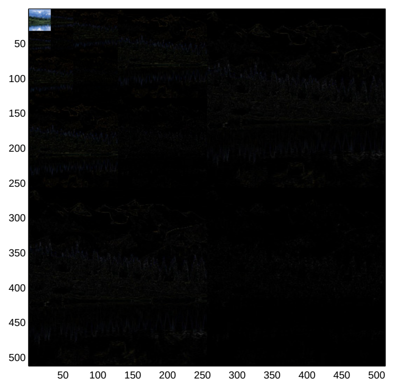
(c) Filtered Image

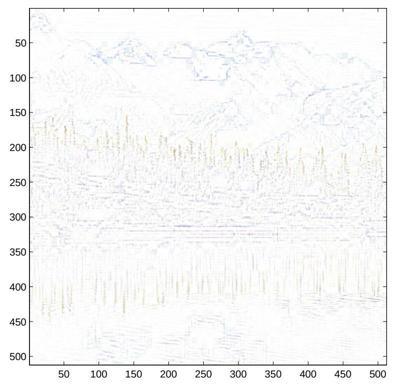


(d) Filtered Spectrum

Figure 5.5: Edge Detection on the Woman Image


(a) Original Image


(b) Original Spectrum


(c) Filtered Image (Inverted)

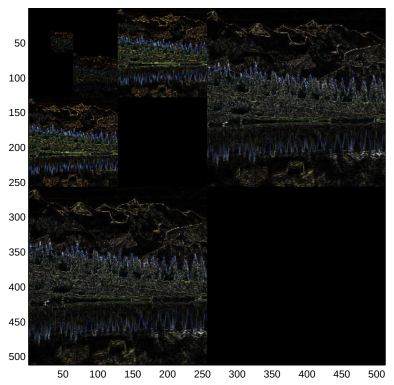
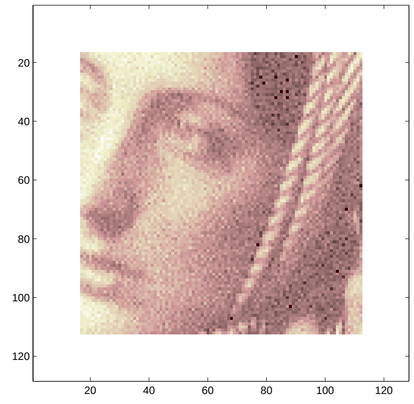
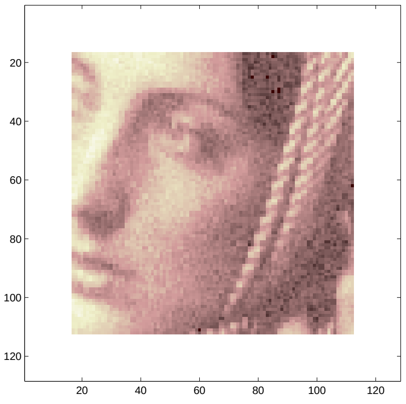

(d) Filtered Spectrum

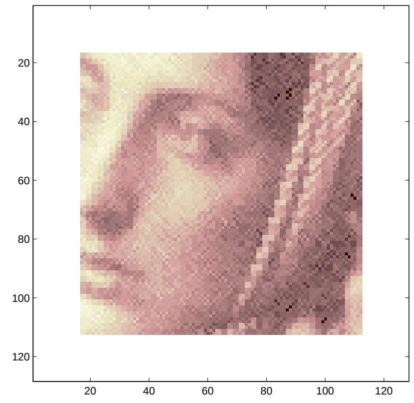
Figure 5.6: Edge Detection on the Mountain Lake Image



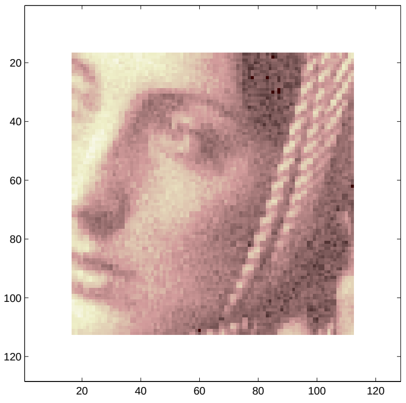
(a) Original Image



(b) Kept 2nd Quadrant

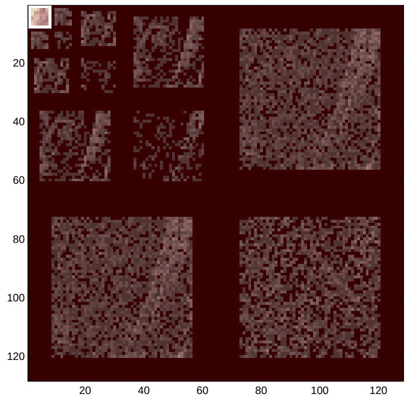


(c) Kept 3rd Quadrant

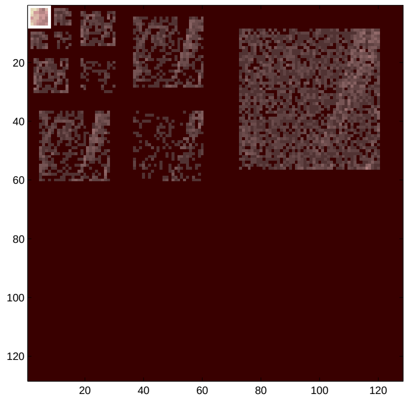


(d) Kept 4th Quadrant

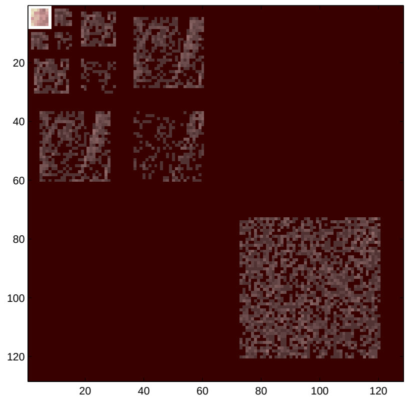
Figure 5.7: De-noising the Noisy Woman Image



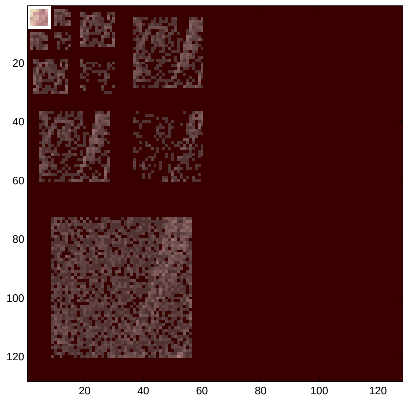
(a) Original Spectrum



(b) Kept 2nd Quadrant



(c) Kept 3rd Quadrant



(d) Kept 4th Quadrant

Figure 5.8: Spectra of the Noisy Woman Image

their approach, they compare two image spectra using standard linear correlation, which seems to exhibit significantly higher correlation after a low-pass or high-pass transformation of the spectrum. An application of determining image similarity is pattern matching in images. Given an input image, and a target image that specifies a particular shape or object, the goal of pattern matching is to find where the target is located in the input image. A reliable method to determine similarity between images would facilitate pattern matching.

Other applications in image processing include sharpening - is it possible to make an image less blurry using a transformation of the image decomposition? This problem falls under the more general category of image restoration, along with many other problems such as unsharp-masking (accentuating edges in the image), noise suppression, and distortion suppression. These applications and more are described in [26].

Chapter 6

Conclusion and Future Work

In this paper, we have described a general representation theoretical framework surrounding iterated wreath product groups, resulting in decompositions of digital images. We first presented tools from representation theory in Chapter 2, followed by a derivation of the classical DFT. Next, we described the wreath products and their structure in Chapter 3, which provided the background for Chapter 4. We then described a multiresolution filtration that provided us with a wreath product group invariant decomposition. After presenting examples of image decompositions with respect to various iterated wreath products, in Chapter 5 we applied our decomposition to compression, edge detection, and de-noising.

6.1 Future Work

6.1.1 Image Decomposition

In Chapter 4, we described the quad-tree scanning method and derived the spectral decomposition of a function on the leaves of a quad tree, based on the multiresolution filtering of the function space. We have considered the case for a quad-tree, but it would be worthwhile to investigate decompositions for other SHRTs and their automorphism groups. Furthermore, our decomposition is with respect to iterated wreath products of the same group; what happens for decompositions of iterated wreath products of different automorphism groups for each level of the SHRT? In addition, how could we better explain the role of each symmetry group in the decomposition? Further research in this field may result in a catalog of decompositions of functions defined on SHRTs.

In addition, it would be interesting to investigate decompositions with respect to finite field-valued functions instead of complex-valued functions. In this case, it may not be necessarily true that the characteristic of the field does not divide the order of the group, and thus new representation theoretical tools would be needed.

Finally, as a natural extension to the decomposition, we would like to see our methods applied to 3D volumetric data. Is it worthwhile to construct an analogous *oct-tree* scanning method and decompose it with respect to this oct-tree? What kind of information would the volume spectrum encapsulate in this case?

6.1.2 Image Processing

The thresholding methods described in Section 5.1 use the sparsity-norm balancing method to determine an appropriate threshold. In JPEG compression, the image is transformed to YUV space, which separates the image into luminance and chrominance components. The human eye is apparently less sensitive to changes in chrominance than to changes in luminance values. We could run the same compression algorithm, but instead of running it on an RGB image, we could run it on the chrominance components of the YUV version of the image. This might achieve better compression results in terms of perceived image quality.

Lastly, a promising direction of research would be to determine which wavelet-based image processing methods could be effectively applied to our decomposition scheme. So far, the three application areas of compression, edge detection, and denoising seem comparable with analogous wavelet-based methods. What are the comparative advantages of wavelet methods versus wreath product methods?

6.2 In Closing

Researchers are making many connections between practical algorithms and mathematical theory. In particular, I have attempted to connect ideas from algebra, engineering, and computer science in this paper, by linking representation theory, wavelets, and image processing. I hope that my paper has inspired further interest in this fascinating area of mathematics.

Appendix A

Manual for my MATLAB Programs

I developed the image processing software entirely in MATLAB. The scripts are setup so that some global variables (variables in the workspace) are used among many scripts, so it is imperative to set these global variables up before attempting other image processing operations.

In order to initially read an image, first, the image must be located in the current directory. Next, the script `run.m` should be modified so that the `imagefile` variable is set to the name of the desired image. Next, you can modify a dazzling number of different global options to suit your needs, and the purpose of each option is described as comments in the code.

When you are ready to begin processing, type `run` in the command window, which will execute the `run.m` script. This will automatically generate the spectrum and run the inverse decomposition routine, displaying everything concisely in one window. After running this command, you are able to run any other image processing script. These scripts are `compression.m`, `isocurve.m`, `multiresolution.m`, and `replacement.m`.

Appendix B

Programs

B.1 autocrop.m

Routine for automatically cropping an image so that its dimensions are powers of 2.

```
function cropped = autocrop(image)

% autocrop     automatically crops or pads and image so that it's suitable
%           for processing.  It rounds the smaller dimension to the
%           nearest power of 2, which compromises between minimizing
%           the amount of whitespace and preserving features in the
%           original image.  The algorithm also crops pictures a fourth
%           of the way down in height, because this is where people's
%           faces usually end up.
%
% Usage: autocrop(matrix) where matrix represents a loaded image.  The
%      program can accomodate up to four dimensions.
%

sizearray = size(image);
height = sizearray(1);
width = sizearray(2);
otherdims = sizearray(3:end);

new_height = height;
new_width = width;
log_height = log2(height);
log_width = log2(width);

% test "power of 2"-ness
powertest = [log_height == floor(log_height), log_width == floor(log_width)];

% in case we need to reset both width and height,
% do some calculations based on the larger dimension
suggested_pwr = ceil(log2(width));
if (height < width)
  suggested_pwr = round(log2(height));
```

```
end
if (suggested_pwr > 10)
  suggested_pwr = 10;
end

if height ~= width
disp 'Warning: The number of rows do not match the number of columns.';

% if both are powers of 2, then set the new dimensions to the higher value
if (powertest(1) && powertest(2))
  if (height >= width)
    new_width = height;
  else
    new_height = width;
  end
  elseif powertest(1)
    new_width = height;
  elseif powertest(2)
    new_height = width;
  else
    % just set it to 256 times 256
    new_height = 2^suggested_pwr;
    new_width = 2^suggested_pwr;
end
else
  % if the rows and columns match, then test only one
  if ~ powertest(1)
    % just set it
    new_height = 2^suggested_pwr;
    new_width = 2^suggested_pwr;
  end
end


changetest = [new_height ~= height, new_width ~= width];

if (changetest(1) || changetest(2))
  disp(['New dimesions are: ' num2str(new_height) ' by '  num2str(new_width) ' by '
      num2str(otherdims)]);
end

% if we need to change width
if changetest(2)
  % test if we need to pad with ones or eliminate some lines
  if width < new_width
    % number of lines to pad
    nlines = new_width - width;
    % distribute both left and right
    leftpad = floor(nlines / 2);
    rightpad = nlines - leftpad;

    % assume the height is constant and pad it
    image = [ones([height leftpad otherdims]), image, ones([height rightpad otherdims
        ])];
    width = new_width;
  else
```

```
    % let’s try to crop either side of it
    % number of lines to crop
    nlines = width - new_width;
    % distribute both left and right
    leftcrop = floor(nlines / 2) + 1;
    rightcrop = leftcrop + new_width - 1;

    % assume the height is constant and pad it
    image = image(:, leftcrop:rightcrop, :, :);
    width = new_width;
  end
end

% if we need to change height (copied directly from the width case with minor changes)
if changetest(1)
  % test if we need to pad with ones or eliminate some lines
  if height < new_height
    % number of lines to pad
    nlines = new_height - height;
    % distribute both top and bottom
    toppad = floor(nlines / 2);
    bottompad = nlines - toppad;

    % assume the width is constant and pad it
    image = [ones([toppad width otherdims]); image; ones([bottompad width otherdims])
        ];
    height = new_height;
  else
    % let’s try to crop either side of it
    % number of lines to crop
    nlines = height - new_height;
    % distribute both top and bottom
    % ... let’s skew it up a bit, because people’s faces tend to be
    % ... near the top
    topcrop = floor(nlines / 4);
    bottomcrop = topcrop + new_height - 1;

    % assume the width is constant and pad it
    image = image(topcrop:bottomcrop, :, :, :);
    height = new_height;
  end
end

% assign the result
cropped = image;
```

B.2 batchread.m

Script for reading in a file from the variable `imageloc`.

```
% % batchread.m     reads a file using imread and saves
original = imread(imageloc);

original = double(original)/255;
```

```

% autocrop the image
original = autocrop(original);
```

B.3 batchrun.m

Script for reading and automatically generating image decompositions in ppm format.

```
% % batchrun.m - calculation and display script in batch mode
batchread;

% global options
spectSelected = 'fourier';                % choose the basis
decompLevel = 2;                    % maximum spectral decomposition level
[basis, usesImaginary] = getBasis(spectSelected); % get the basis
invertMagnitude = false;                % invert magnitude colors
invertPhase = false;                   % invert phase colors
brightness = 0.3;                   % defines the brightness of the colormap
% myColormap = brighten(gray, brightness);       % assign the colormap with a certain brightness
myColormap = pink;                     % assign the colormap with a certain brightness

% compression options
compressionRestoreHiLevel = true;           % restores the avg pixel values at the highest level
                               % ... (smallest remaining upper left quadrant)
                               % ... after the compression step.
selectedThreshold = 'balance';            %
% selectedThreshold = 'sqrtBalance';          % thresholding method selection
% selectedThreshold = 'nearZero';           %


% Now, run through the decomposition and inverse decomposition
% display the original
figure; close
imagesc(original); axis square; colormap gray;
print(gcf, '-dppm', [imageloc '-orig.ppm']);

% do the decomposition
spect = quad_spectrum2(original, decompLevel, spectSelected);

% display the decomposition
figure; close
if (invertMagnitude) imagesc(1-normalize(spect));
else imagesc(normalize(spect)); end
axis square; colormap gray;
print(gcf, '-dppm', [imageloc '-amp.ppm']);

% display phase information if we have imaginary numbers
figure; close
if (usesImaginary)
  if (invertPhase) imagesc(1-normalize(angle(spect)));
  else imagesc(normalize(angle(spect))); end
else imagesc(zeros(size(original))); end
axis square; colormap(myColormap);
```

```
print(gcf, '-dppm', [imageloc '-phase.ppm']);

% do the inverse decomposition
recovered = quad_inverse(spect, decompLevel, spectSelected);

% fix numerical errors
test = double(recovered >= 0); recovered = recovered.*test;          % cast all values
      less than 0 as 0
test = double(recovered <= 1); recovered = recovered.*test + (1-test);   % cast all values
       greater than 1 as 1

% display recovered image
figure; close
imagesc(recovered); axis square; colormap(myColormap);
print(gcf, '-dppm', [imageloc '-recov.ppm']);

% find error between original and recovered image
error = original-recovered;
disp(['Maximum Error: ' num2str(max(max(max(error))))]);
disp(['Maximum Recovered Value: ' num2str(max(max(max(recovered))))]);
disp(['Minimum Recovered Value: ' num2str(min(min(min(recovered))))]);
```

B.4 blowup.m

Script for enlarging a smaller quadrant of the spectrum to replace a larger quadrant.

```
function result = blowup(M, factor)

height = size(M, 1);
width = size(M, 2);
colors = size(M, 3);

result = zeros(height*factor, width*factor, colors);

for i = 1:height
    for j = 1:width
        for c = 1:colors
            result(factor*(i-1)+1:factor*(i), factor*(j-1)+1:factor*(j), c) = M(i, j,
                c)*ones(factor, factor);
        end
    end
end
```

B.5 compression.m

Script for compression. Handles the input and output of figures.

```
% compression.m      script to run compression routines

numpixels = numel(spect);
ratio = 1;
```

```
avgangle = sum(sum(sum(angle(spect))))/numpixels;

% calculate all the constants necessary
dim = size(spect, 1);                % the number of rows in the image
height = log2(dim);              % the "height" of the quad-tree based on this image
highestLevel = 2^(height - decompLevel - 1);

[threshold, energy] = sparsityNorm(spect, selectedThreshold);

% nearzeroamp = (normalize(spect) <= threshold);
% bigangle = (angle(spect) >= avgangle);
% nearzeros = nearzeroamp.*(1-bigangle);
nearzeros = (normalize(spect) <= threshold);
ratio = sum(sum(sum(nearzeros))) / numpixels;
pruned = spect.*(1-nearzeros);

% we don't want to threshold the highest level where the average pixels are
% stored - restore that part of the spectrum.
if (compressionRestoreHiLevel)
  pruned(1:highestLevel, 1:highestLevel, :) = spect(1:highestLevel, 1:highestLevel, :)
      ;
end

% reconstruct the image
result = quad_inverse(pruned, decompLevel, spectSelected);
% fix numerical errors
fix = double(result >= 0); result = result.*fix;           % cast all values less than 0 as 0
fix = double(result <= 1); result = result.*fix + (1-fix);   % cast all values greater than 1
    as 1

error = original-result;
disp(['Pruning amplitudes less than or equal to ' num2str(threshold)]);
disp(['Compression Rate; ~ (# pixels pruned)/(# pixels total) : ' num2str(ratio*100) '
    %']);
disp(['Maximum Error: ' num2str(max(max(max(error))))]);
disp(['Minimum Error: ' num2str(min(min(min(error))))]);
disp(['Average Error: ' num2str(sum(sum(sum(error)))/numel(error))]);
disp(['Maximum Recovered Value: ' num2str(max(max(max(result))))]);
disp(['Minimum Recovered Value: ' num2str(min(min(min(result))))]);
disp ' ';

if (displayCompress)
  % display the original image
  subplot(2, 2, 1); imagesc(original); title('Original Image'); axis square; colormap(
      myColormap);

  % display the result
  subplot(2, 2, 2); imagesc(result); title('Compression Result'); axis square;
      colormap(myColormap);

  % display the thresholded decomposition
  subplot(2, 2, 3);
  if (invertMagnitude)
    imagesc(1-normalize(pruned));
    title(['Spectrum Magnitude After Thresholding (Inverted, ' num2str(ratio*100) '%
        Compression Rate)']);
  else
    imagesc(normalize(pruned));
    title(['Spectrum Magnitude After Thresholding (' num2str(ratio*100) '% Compression
         Rate)']);
  end
  axis square; colormap(myColormap);
else
  figure; imagesc(result); axis square; colormap(myColormap);
end
```

B.6 isocurve.m

Draws lines around pixels of equal color value.

```
function lines = isocurve(M, values)

% from Schroeder, et. al. "The Visualization Toolkit."
%
% Marching Squares / Cubes Algorithm (p. 160)
%
%  1. Select a cell (a 2-by-2 submatrix, or a 2-by-2-by-2 subcube)
%  2. Calculate the inside / outside state of each vertex of the cell
%  3. Create an index by storing the binary state of each vertex
%    in a separate bit
%  4. Use the index to look up the topological state of the cell
%    in a case table
%  5. Calculate the contour location (via interpolation) for each edge
%    in the case table.
%

% test case
% M = [0 1 1 3 2; 1 3 6 6 3; 3 7 9 7 3; 2 7 8 6 2; 1 2 3 4 3];
% value = 5;
for value = values
drawline = zeros(2, 2);
drawline2 = zeros(2, 2);

% for now, assume M is a two-dimensional matrix.
height = size(M, 1);
width = size(M, 2);

for i = 1 : height-1
  for j = 1 : width - 1
    % first we select the 2-by-2 cell where (i, j) specifies the upper
    % left hand corner ((i,j), (i+1,j), (i,j+1), (i+1,j+1)
    cell = M(i:i+1, j:j+1);

    % next, we calculate the inside / outside state
    % a vertex is in if its value is greater or equal than the value given
    inNout = (cell >= value);

    % next, we associate each case with a index (binary, 16 cases)
    index = inNout(1, 1) + 2*inNout(1, 2) + 4*inNout(2, 2) + 8*inNout(2, 1);

    top = [i, j + interp(value, cell(1, 1), cell(1, 2))];
    bottom = [i+1, j + interp(value, cell(2, 1), cell(2, 2))];
    left = [i + interp(value, cell(1, 1), cell(2, 1)), j];
    right = [i + interp(value, cell(1, 2), cell(2, 2)), j+1];

    % now, go through the cases for the topology & make contours
    % lines are represented: height location first, then width location
    switch (index)
      case {0, 15}
      case {1, 14}
        drawline = [top; left];
      case {2, 13}
        drawline = [top; right];
      case {3, 12}
        drawline = [left; right];
      case {4, 11}
        drawline = [bottom; right];
      case 5
        drawline = [left; bottom];
        drawline2 = [top; right];
      case {6, 9}
        drawline = [top; bottom];
      case {7, 8}
        drawline = [left; bottom];
      case 10
        drawline = [bottom; right];
        drawline2 = [top; left];
    end

    % draw the isocurve
    line(drawline(:, 1), drawline(:, 2));
    if ((index == 5) || (index == 10))
      line(drawline2(:, 1), drawline2(:, 2));
    end
  end
end

end

function ans = interp(v, a, b)
if (a ~= b)
  ans = abs((v - a) / (a - b));
else
  ans = 0;
end
```

B.7 getBasis.m

Function that returns the appropriate DFT matrix for a group.

```
function [M, usesImaginary] = getBasis(selection)

% define change of basis here
w = -j;
```

```
spectTypes = {'fourier', [1 1 1 1; 1 w w^2 w^3; 1 w^2 w^4 w^6; 1 w^3 w^6 w^9]/4;...
      'haar', [1 1 1 1; 1 -1 1 -1; 1 1 -1 -1; 1 -1 -1 1]/4;...
      'dihedral', [1 1 1 1; 1 -1 1 -1; 2 0 -2 0; 0 2 0 -2]/4;...
      'alternating', [1 1 1 1; -1 3 -1 -1; -1 -1 3 -1; -1 -1 -1 3]/4;...
      'symmetric', [1 1 1 1; -1 3 -1 -1; -1 -1 3 -1; -1 -1 -1 3]/4;};


% basis selction
numTypes = size(spectTypes, 1);
M = 0;

for i=1:numTypes
    % compare strings to choose basis
    if (strcmp(spectTypes{i, 1}, selection))
        M = spectTypes{i, 2};
        if (imag(M) == 0)
            usesImaginary = false;
        else
            usesImaginary = true;
        end
        break;
    elseif (i == numTypes)
        disp(['No match found for basis of name ' selection '.  Please try again.']);
    end
end
```

B.8 multiresolution.m

Script to progressively recover the image from portions of the spectrum.

```
height = log2(size(spect, 1));

for i=height:-1:height-decompLevel
  G = spect(1:2^i, 1:2^i, :);
  H = quad_inverse(G, decompLevel+(i-height), spectSelected);
%   F = double(H >= 0); H = H.*F;            % cast all values less than 0 as 0
%   F = double(H <= 1); H = H.*F + (1-F);    % cast all values greater than 1 as 1
  figure; imagesc(normalize(H)); axis square;
end
```

B.9 normalize.m

Normalizes the values of the recovered image so that it falls under 0 and 1.

```
function result = normalize(matrix)

result = abs(matrix);
% result = matrix;

for i=1:size(matrix, 3);
    result(:, :, i) = (1/max(max(result(:, :, i))))*result(:, :, i);
end
```

B.10 quad_inverse.m

Performs the inverse decomposition.

```
function result = quad_inverse(spectrum, spectLevel, basis)

% quad_inverse     quad-tree/DFT inverse decomposition of an image
%
% Usage: quad_inverse(M)

% change of basis matrix
% w = -j;
% d = [1 1 1 1; 1 w w^2 w^3; 1 w^2 w^4 w^6; 1 w^3 w^6 w^9];
% d = [1 1 1 1; 1 -1 1 -1; 1 1 -1 -1; 1 -1 -1 1];
[d, usesImaginary] = getBasis(basis);

% invert the matrix
d = inv(d);

% split the change of basis matrix into real and imaginary parts
dr = real(d);
di = imag(d);

% calculate all the constants necessary
dim = size(spectrum, 1);                % the number of rows in the image
cols = size(spectrum, 2);               % the number of columns in the image
height = log2(dim);             % the "height" of the quad-tree based on this image
colors = size(spectrum, 3);             % the number of colors in the image

% special factor for calculating spectrum
% discontinued
%factor = 4;

if dim ~= cols
  disp 'Error: The number of rows do not match the number of columns.';
  spectrum = 0;
  return
elseif height ~= floor(height)
  disp 'Error: Each dimension of the image must be a power of 2.';
  spectrum = 0;
  return
end

rspectrum = real(spectrum);
roriginal = zeros(dim, dim, colors);
rfft_values = zeros(4, 1);
rpixel_values = zeros(4, 1);
ispectrum = imag(spectrum);
ipixel_values = zeros(4, 1);
if (usesImaginary)
  ioriginal = zeros(dim, dim, colors);
  ifft_values = zeros(4, 1);
end

% % do the first quadrant! (the top 4 pixels).
% for color=1:colors
%    % pick out the pixel values for each color
%    rpixel_values(1) = rspectrum(1, 1, color);
%    rpixel_values(2) = rspectrum(1, 2, color);
%    rpixel_values(3) = rspectrum(2, 2, color);
%    rpixel_values(4) = rspectrum(2, 1, color);
%
%    ipixel_values(1) = ispectrum(1, 1, color);
%    ipixel_values(2) = ispectrum(1, 2, color);
%    ipixel_values(3) = ispectrum(2, 2, color);
%    ipixel_values(4) = ispectrum(2, 1, color);
%
%    % calculate the dft by multiplying the matrix
%    % do this for both the real and imaginary parts
%    rfft_values = dr*rpixel_values*factor - di*ipixel_values*factor;
%    % This line is unnecessary because the recovered image should only be
%    % real, and the imaginary part should just be zero.
% %    ifft_values = dr*ipixel_values*factor + di*rpixel_values*factor;
%
%    % assign the resulting values
%    roriginal(1, 1, color) = rfft_values(1);      % 1st quadrant
%    roriginal(1, 2, color) = rfft_values(2);      % 2nd quadrant
%    roriginal(2, 2, color) = rfft_values(3);      % 3rd quadrant
%    roriginal(2, 1, color) = rfft_values(4);      % 4th quadrant
%
% %    ioriginal(1, 1, color) = ifft_values(1);      % 1st quadrant
% %    ioriginal(1, 2, color) = ifft_values(2);      % 2nd quadrant
% %    ioriginal(2, 2, color) = ifft_values(3);      % 3rd quadrant
% %    ioriginal(2, 1, color) = ifft_values(4);      % 4th quadrant
% end
%
% for color=1:colors
%    rspectrum(1:2, 1:2, color) = roriginal(1:2, 1:2, color);
%    %ispectrum(1:2, 1:2, color) = ioriginal(1:2, 1:2, color);
%    ispectrum(1:2, 1:2, color) = zeros(2, 2);
% end

% quad ordering routine copied here.
for current_height=max(2,height-spectLevel):height

  maxlevel = 1;
  row = 1;
  col = 1;
  counter = 1;

  while maxlevel <= (current_height-1)

    % do stuff here
    for color=1:colors
      % pick out the pixel values for each color
      rpixel_values(1) = rspectrum(row, col, color);
      rpixel_values(2) = rspectrum(row, col + 2^(current_height-1), color);
      rpixel_values(3) = rspectrum(row + 2^(current_height-1), col + 2^(current_height
          -1), color);
      rpixel_values(4) = rspectrum(row + 2^(current_height-1), col, color);

      if (usesImaginary)
```

```
        ipixel_values(1) = ispectrum(row, col, color);
        ipixel_values(2) = ispectrum(row, col + 2^(current_height-1), color);
        ipixel_values(3) = ispectrum(row + 2^(current_height-1), col + 2^(
            current_height-1), color);
        ipixel_values(4) = ispectrum(row + 2^(current_height-1), col, color);
      end

      % calculate the dft by multiplying the matrix
      % do this for both the real and imaginary parts
      if (usesImaginary)
        rfft_values = dr*rpixel_values - di*ipixel_values;
      else
        rfft_values = dr*rpixel_values;
      end

      % This line is unnecessary because the recovered image should only be
      % real, and the imaginary part should just be zero.
%     if (usesImaginary)
%       ifft_values = dr*ipixel_values + di*rpixel_values;
%     end

      % assign the resulting values
      roriginal(2*row - 1, 2*col - 1, color) = rfft_values(1);    % 1st quadrant
      roriginal(2*row - 1, 2*col, color) = rfft_values(2);      % 2nd quadrant
      roriginal(2*row, 2*col, color) = rfft_values(3);        % 3rd quadrant
      roriginal(2*row, 2*col - 1, color) = rfft_values(4);      % 4th quadrant

%       if (usesImaginary)
%         ioriginal(2*row - 1, 2*col - 1, color) = ifft_values(1);     % 1st quadrant
%         ioriginal(2*row - 1, 2*col, color) = ifft_values(2);       % 2nd quadrant
%         ioriginal(2*row, 2*col, color) = ifft_values(3);         % 3rd quadrant
%         ioriginal(2*row, 2*col - 1, color) = ifft_values(4);       % 4th quadrant
%       end
    end

    % the rest of the counting program
    if counter == 4^maxlevel
            row = 1;
            col = 2^maxlevel + 1;
            maxlevel = maxlevel + 1;
    else
      % initial starting values
      level = 0;
      quad = mod(counter, 4);

      while quad == 0
        level = level + 1;
        quad = mod(counter, 4^(level+1));
      end

      if mod(counter, 4^level) == 0
        quad = mod(counter / 4^level, 4);
      end

      row = row - (2^level - 1);

      switch quad
        case 1
          col = col + 2^level;
        case 2
          row = row + 2^level;
        case 3
          col = col - 2^level;
        otherwise
          % turns out that putting in debugging statments like this
          % slows things down significantly
%             disp(['Error: quad = ' num2str(quad) ' should have never happened...']);
      end
    end

    % increment pixel counter
    counter = counter + 1;
  end

  % After we are done with this level, re-assign the spectrum so that we can analyze
  % it in a recursive fashion
  % In the classical DFT case, the top left quadrant will be real, so we
  % only need to pass on the real values.
  for color=1:colors
    rspectrum(1:2^current_height, 1:2^current_height, color) = roriginal(1:2^
        current_height, 1:2^current_height, color);
%     if (usesImaginary)
%       ispectrum(1:2^current_height, 1:2^current_height, color) = ioriginal(1:2^current_height, 1:2^
    current_height, color);
%       ispectrum(1:2^current_height, 1:2^current_height, color) = zeros(2^current_height, 2^current_height);
%     end
  end
end

% make it complex
% if (usesImaginary)
%   result = complex(roriginal, ioriginal);
% end
result = roriginal;
```

B.11 quad_ordering.m

Routine that handles the quad-tree scan. This routine is embedded in the decomposition and inverse decomposition routines.

```
function matrix = quad_ordering(height)

% quad_ordering      Returns a quad-tree like ordering of a matrix
%
% Usage: quad_ordering(height)
%
% For a given height, this function produces a matrix that traverses the
% image in a quad-tree like fashion. The algorithm is simply based on
% detecting when we have finished traversing the current quadrant and then
% moving our position to the next quadrant. It's very fast, and I don't
% think it can get much faster.
%
% Written 11/16/03, Will Chang

dim = 2^height;
matrix = zeros(dim, dim);

maxlevel = 1;
row = 1;
col = 1;
counter = 1;

% in this program, a "level" signifies the level in the quad tree. Unlike
% the quad_spectrum program, level 0 means that we are on the pixel level;
% level 1 on the quad-pixel level, level 2 on the 16-pixel level, and so on
% and so forth.

while maxlevel <= height
    matrix(row, col) = counter;

    % we know when we have hit the last pixel of the current level when our
    % pixel counter has reached number 4^maxlevel. Move to the first pixel
    % in the next level, and increment our maxlevel counter.
    if counter == 4^maxlevel
        row = 1;
        col = 2^maxlevel + 1;
        maxlevel = maxlevel + 1;
    else
        % initial starting values
        level = 0;
        quad = mod(counter, 4);

        % find the local level, that is, find out if we're on the pixel
        % level, the 4-pixel level, the 16-pixel level, etc. by testing
        % the divisibility of our pixel counter by powers of 4.
        while quad == 0
            level = level + 1;
            quad = mod(counter, 4^(level+1));
        end

        % given that we're on some level, we want to find our the relative
        % position of our quadrant, because we'll have to find out a way to
        % move to the next quadrant.
        % This is a very cool trick.
        if mod(counter, 4^level) == 0
            quad = mod(counter / 4^level, 4);
        end

        % go back to position 1 in our local quadrant
        % This is also a very cool trick.
        row = row - (2^level - 1);

        switch quad
            case 1
                col = col + 2^level;
            case 2
```

```
                row = row + 2^level;
            case 3
                col = col - 2^level;
            otherwise
                % turns out that putting in debugging statments like this
                % slows things down significantly
%                disp(['Error: quad = ' num2str(quad) ' should have never happened...']);
        end
    end

    % increment pixel counter
    counter = counter + 1;
end
```

B.12 quad_spectrum2.m

Performs the decomposition.

```
function result = quad_spectrum2(M, userMaxLevel, basis)

% quad_spectrum2     quad-tree/DFT decomposition of an image
%
% Usage: quad_spectrum2(M)
%
%    M is the image to be decomposed. This is a fast version of the
%    recursive quad_spectrum program.
%
%    The way I made it fast was to use the quick ordering method developed in
%    quad_ordering so that I can march through the image and do a constant
%    amount of work per pixel in the image.
%    Since MATLAB hates dealing with complex numbers, I split up the
%    decomposition into separate real and imaginary parts so that every line
%    in the processing loop is "accelerated."

% get change of basis matrix
% w = -j;
% d = [1 1 1 1; 1 w w^2 w^3; 1 w^2 w^4 w^6; 1 w^3 w^6 w^9];
% d = [1 1 1 1; 1 -1 1 -1; 1 1 -1 -1; 1 -1 -1 1];
[d, usesImaginary] = getBasis(basis);


% split the change of basis matrix into real and imaginary parts
dr = real(d);
di = imag(d);

% calculate all the constants necessary
dim = size(M, 1);    % the number of rows in the image
cols = size(M, 2);    % the number of columns in the image
height = log2(dim);   % the "height" of the quad-tree based on this image
colors = size(M, 3);  % the number of colors in the image

% special factor for calculating spectrum
% discontinued
%factor = 4;
```

```

if dim ~= cols
  disp 'Error: The number of rows do not match the number of columns.';
  spectrum = 0;
  return
elseif height ~= floor(height)
  disp 'Error: Each dimension of the image must be a power of 2.';
  spectrum = 0;
  return
end

pixel_values = zeros(4, 1);
rspectrum = zeros(dim, dim, colors);
rfft_values = zeros(4, 1);
ifft_values = zeros(4, 1);
if (usesImaginary)
  ispectrum = zeros(dim, dim, colors);
end

% quad ordering routine copied here.
for current_height=height:-1:max(2, height-userMaxLevel)  % this needs to go up only to 2
    because maxlevel starts at 1.
  maxlevel = 1;
  row = 1;
  col = 1;
  counter = 1;

  while maxlevel <= (current_height-1)

    % do stuff here
    for color=1:colors
      % pick out the pixel values for each color
      pixel_values(1) = M(2*row - 1, 2*col - 1, color);
      pixel_values(2) = M(2*row - 1, 2*col, color);
      pixel_values(3) = M(2*row, 2*col, color);
      pixel_values(4) = M(2*row, 2*col - 1, color);

      % calculate the dft by multiplying the matrix
      % do this for both the real and imaginary parts
      rfft_values = dr*pixel_values;
      if (usesImaginary)
        ifft_values = di*pixel_values;
      end

      % assign the resulting values
%         rspectrum(row, col, color) = counter;                              % 1st quadrant
%         rspectrum(row, col + 2^(current_height-1), color) = counter;               % 2nd quadrant
%         rspectrum(row + 2^(current_height-1), col + 2^(current_height-1), color) = counter; % 3rd quadrant
%         rspectrum(row + 2^(current_height-1), col, color) = counter;               % 4th quadrant

      rspectrum(row, col, color) = rfft_values(1);                    % 1st quadrant
      rspectrum(row, col + 2^(current_height-1), color) = rfft_values(2);
          % 2nd quadrant
      rspectrum(row + 2^(current_height-1), col + 2^(current_height-1), color) =
          rfft_values(3);  % 3rd quadrant
      rspectrum(row + 2^(current_height-1), col, color) = rfft_values(4);     % 4th quadrant

      if (usesImaginary)
        ispectrum(row, col, color) = ifft_values(1);                                   % 1st quadrant
        ispectrum(row, col + 2^(current_height-1), color) = ifft_values(2);     % 2nd quadrant
        ispectrum(row + 2^(current_height-1), col + 2^(current_height-1), color) = ifft_values(3);  % 3rd quadrant
        ispectrum(row + 2^(current_height-1), col, color) = ifft_values(4);     % 4th quadrant
      end
    end

    % the rest of the counting program
    if counter == 4^maxlevel
            row = 1;
            col = 2^maxlevel + 1;
            maxlevel = maxlevel + 1;
    else
      % initial starting values
      level = 0;
      quad = mod(counter, 4);

      while quad == 0
        level = level + 1;
        quad = mod(counter, 4^(level+1));
      end

      if mod(counter, 4^level) == 0
        quad = mod(counter / 4^level, 4);
      end

      row = row - (2^level - 1);

      switch quad
        case 1
          col = col + 2^level;
        case 2
          row = row + 2^level;
        case 3
          col = col - 2^level;
        otherwise
          % turns out that putting in debugging statments like this
          % slows things down significantly
%             disp(['Error: quad = ' num2str(quad) ' should have never happened...']);
      end
    end

    % increment pixel counter
    counter = counter + 1;
  end

  % After we are done with this level, re-assign M so that we can analyze
  % it in a recursive fashion
  % In the classical DFT case, the top left quadrant will be real, so we
  % only need to pass on the real values.
  M = rspectrum;
end


% do the last quadrant! (the remaining top 4 pixels are not done yet).
% for color=1:colors
%   % pick out the pixel values for each color
%   pixel_values = [M(1, 1, color); M(1, 2, color); M(2, 2, color); M(2, 1, color)];
%
%   % calculate the dft by multiplying the matrix
%   % do this for both the real and imaginary parts
%   rfft_values = dr*pixel_values/factor;
%   ifft_values = di*pixel_values/factor;
%
%   % assign the resulting values
%   rspectrum(1, 1, color) = rfft_values(1);      % 1st quadrant
%   rspectrum(1, 2, color) = rfft_values(2);      % 2nd quadrant
%   rspectrum(2, 2, color) = rfft_values(3);      % 3rd quadrant
%   rspectrum(2, 1, color) = rfft_values(4);      % 4th quadrant
%
%   ispectrum(1, 1, color) = ifft_values(1);      % 1st quadrant
%   ispectrum(1, 2, color) = ifft_values(2);      % 2nd quadrant
%   ispectrum(2, 2, color) = ifft_values(3);      % 3rd quadrant
%   ispectrum(2, 1, color) = ifft_values(4);      % 4th quadrant
% end

% make it complex
if (usesImaginary)
  result = complex(rspectrum, ispectrum);
else
  result = rspectrum;
end
```

B.13 read.m

Reads the image file.

```
% % read.m     reads a file using imread
original = imread(imagefile);

original = double(original)/255;
% R = original(:, :, 1);
% G = original(:, :, 2);
% B = original(:, :, 3);
% original = B;

% figure; imagesc(R); colormap(gray); axis square;
% figure; imagesc(G); colormap(gray); axis square;
% figure; imagesc(B); colormap(gray); axis square;

% autocrop the image
original = autocrop(original);
```

B.14 replacement.m

Replaces a quadrant with zeros.

```
function result = replacement(spect, levels, replaces, basis, mymap)
% replacement scheme - I'm experimenting with replacing certain
% decomposition levels with lower-resolution equivalents.

height = size(spect, 1);
width = size(spect, 2);
colors = size(spect, 3);

% height and width should be the same
dim = height;
counter = 0;

result = spect;

for level=levels
  counter = counter + 1;
  lo = dim / (2^(level+1));
  hi = dim / (2^(level));

  % quad_a = spect(1:lo, lo+1:2*lo, :);
  % quad_b = spect(lo+1:2*lo, lo+1:2*lo, :);
  % quad_c = spect(lo+1:2*lo, 1:lo, :);
  quad_0 = ones(lo, lo, colors);
  quad_a = zeros(lo, lo, colors);
  quad_b = zeros(lo, lo, colors);
  quad_c = zeros(lo, lo, colors);

  for replace=replaces{counter}
    switch (replace)
      case 1
        result(1:hi, 1:hi, :) = blowup(quad_0, 2);
      case 2
        result(1:hi, hi+1:2*hi, :) = blowup(quad_a, 2);
      case 3
        result(hi+1:2*hi, hi+1:2*hi, :) = blowup(quad_b, 2);
      case 4
        result(hi+1:2*hi, 1:hi, :) = blowup(quad_c, 2);
    end
  end
end

figure;imagesc(normalize(result)); axis square; colormap(mymap);

recovered = quad_inverse(result, 3, basis);
test = double(recovered >= 0); recovered = recovered.*test;           % cast all values
    less than 0 as 0
test = double(recovered <= 1); recovered = recovered.*test + (1-test);   % cast all values
     greater than 1 as 1
figure; imagesc(recovered); axis square; colormap(mymap);
```

B.15 run.m

Script that runs the decomposition and inverse decomposition.

```
% run.m - calculation and display script
% My Examples
imagefile = 'cat.bmp';
% imagefile = 'strange6-256.bmp';
% imagefile = 'oh_my-256.bmp';
% imagefile = 'mountain-lake.tif';

% Examples from the wavelet toolbox
% imagefile = 'geometry.bmp';
% imagefile = 'sinsin.bmp';
% imagefile = 'tartan.bmp';
% imagefile = 'woman.bmp';
% imagefile = 'noisewom.bmp';
% imagefile = 'tire.bmp';
% imagefile = 'finger.bmp';
% imagefile = 'finger256.bmp';
% imagefile = 'chess.bmp';
% imagefile = 'facets.bmp';
read;

% global options
spectSelected = 'fourier';                  % choose the basis
decompLevel = 2;                      % maximum spectral decomposition level
[basis, usesImaginary] = getBasis(spectSelected); % get the basis
invertMagnitude = false;                % invert magnitude colors
invertPhase = false;                    % invert phase colors
brightness = 0.3;                     % defines the brightness of the colormap
myColormap = brighten(pink, brightness);       % assign the colormap with a certain brightness

% Display Options
displayRun = true;                      % displays the decomposition / inverse decomposition
displayCompress = true;                 % displays all the information about the compression

% compression options
compressionRestoreHiLevel = false;           % restores the avg pixel values at the highest level
                                  % ... (smallest remaining upper left quadrant)
                                  % ... after the compression step.
% selectedThreshold = 'balance';             %
selectedThreshold = 'sqrtBalance';          % thresholding method selection
% selectedThreshold = 'nearZero';            %


% Now, run through the decomposition and inverse decomposition

% do the decomposition
spect = quad_spectrum2(original, decompLevel, spectSelected);

%figure; imagesc(normalize(spect)); axis square; colormap(myColormap);


if (displayRun)
  % display the original
  title(['Results of ' spectSelected ' transform']);
  subplot(2, 2, 1); imagesc(original); title('Original Image'); axis square; colormap(
      myColormap);

  % display the decomposition
  subplot(2, 2, 3);
  if (invertMagnitude) imagesc(1-normalize(spect)); title('Spectrum Magnitude (
      Inverted)');
  else imagesc(normalize(spect)); title('Spectrum Magnitude'); end
  axis square; colormap(myColormap);

  % display phase information if we have imaginary numbers
  subplot(2, 2, 4);
  if (usesImaginary)
    if (invertPhase) imagesc(1-normalize(angle(spect))); title('Spectrum Phase (
        Inverted)');
    else imagesc(normalize(angle(spect))); title('Spectrum Phase'); end
  else imagesc(zeros(size(original))); title('Spectrum Phase'); end
  axis square; colormap(myColormap);

  % do the inverse decomposition
  recovered = quad_inverse(spect, decompLevel, spectSelected);

  % fix numerical errors
  test = double(recovered >= 0); recovered = recovered.*test;           % cast all values
      less than 0 as 0
  test = double(recovered <= 1); recovered = recovered.*test + (1-test);   % cast all
      values greater than 1 as 1

  % display recovered image
  subplot(2, 2, 2); imagesc(recovered); title('Recovered Image'); axis square;
      colormap(myColormap);

  % find error between original and recovered image
  error = original-recovered;
  disp(['Maximum Error: ' num2str(max(max(max(error))))]);
  disp(['Maximum Recovered Value: ' num2str(max(max(max(recovered))))]);
  disp(['Minimum Recovered Value: ' num2str(min(min(min(recovered))))]);
end
```

B.16 runall.m

Script that runs many different decompositions for many images.

```
% runall.m -- script to execute all of the different types of decompositions
% My Examples
% imagefile = 'cat.bmp';
% imagefile = 'strange6-256.bmp';
% imagefile = 'oh_my-256.bmp';

% Examples from the wavelet toolbox
% imagefile = 'geometry.bmp';
% imagefile = 'sinsin.bmp';
% imagefile = 'tartan.bmp';
% imagefile = 'woman.bmp';
% imagefile = 'tire.bmp';
% imagefile = 'finger.bmp';
% imagefile = 'finger256.bmp';
% imagefile = 'chess.bmp';
% imagefile = 'facets.bmp';

imagefiles = {'oh_my-256'};
% imagefiles = {'geometry', 'sinsin', 'tartan'};
% spectra = {'alternating', 'symmetric'};

numFiles = size(imagefiles, 2);
% numSpectra = size(spectra, 2);


% global options
spectSelected = 'fourier';
% decompLevel = 9;                        % maximum spectral decomposition level
invertMagnitude = true;                 % invert magnitude colors
invertPhase = false;                    % invert phase colors
brightness = 0.3;                       % defines the brightness of the colormap
myColormap = brighten(pink, brightness);       % assign the colormap with a certain brightness

% compression options
compressionRestoreHiLevel = true;           % restores the avg pixel values at the highest level
                                  % ... (smallest remaining upper left quadrant)
                                  % ... after the compression step.
selectedThreshold = 'balance';            %
% selectedThreshold = 'sqrtBalance';          % thresholding method selection
% selectedThreshold = 'nearZero';            %

% Now, run through the decomposition and inverse decomposition
% display the original
% figure;
% subplot(2, numSpectra, 1); imagesc(original); title('Original Image'); axis square; colormap(myColormap);

for i=1:numFiles
%   for j=1:numSpectra
  for j=0:6;
    imagefile = [imagefiles{i} '.bmp'];
    read;
%      spectSelected = spectra{j};                % choose the basis
    [basis, usesImaginary] = getBasis(spectSelected); % get the basis
    decompLevel = j;

    % do the decomposition
    spect = quad_spectrum2(original, decompLevel, spectSelected);

    % display the decomposition
%      subplot(2, numSpectra, i+numSpectra);
    figure;
    if (invertMagnitude) imagesc(1-normalize(spect));
    else imagesc(normalize(spect)); end
    axis square; colormap(myColormap);
    print(gcf, '-depsc2', '-r150', [imagefiles{i} '-' spectSelected '.eps']);

    % display phase information if we have imaginary numbers
%      subplot(2, 2, 4);
%      if (usesImaginary)
%        if (invertPhase) imagesc(1-normalize(angle(spect))); title('Spectrum Phase (Inverted)');
%        else imagesc(normalize(angle(spect))); title('Spectrum Phase'); end
%      else imagesc(zeros(size(original))); title('Spectrum Phase'); end
%      axis square; colormap(myColormap);
  end
end
```

B.17 sparsityNorm.m

Function that calculates sparsity-norm balancing.

```
function [finalThreshold, finalEnergy] = sparsityNorm(spectrum, threshold)

thresholdMethod = {'balance', 'sqrtBalance', 'nearZero'};
sqrtFactor = 128;
index = 1;
numThresholds = 200;
normSpect = normalize(spectrum);
zerosExceed = 0;

% size information
rows = size(spectrum, 1);   % the number of rows in the image
cols = size(spectrum, 2);   % the number of columns in the image
colors = size(spectrum, 3);   % the number of colors in the image

numElements = rows * cols;
if (colors > 0) numElements = numElements * colors; end
% totalEnergy = norm(spectrum, 2);
totalEnergy = sum(sum(sum(spectrum.*conj(spectrum))));

comparison = zeros(size(spectrum));
thresholds = zeros(1, numThresholds + 1);
percentZeros = zeros(1, numThresholds + 1);
percentEnergy = zeros(1, numThresholds + 1);

for currentThreshold = 0:1/numThresholds:1
  comparison = double(normSpect > currentThreshold);
  numZeros = sum(sum(sum(comparison)));

  % apply the threshold
  spectrum = spectrum .* comparison;
%   energy = norm(spectrum, 2);
  energy = sum(sum(sum(spectrum.*conj(spectrum))));

  thresholds(index) = currentThreshold;
  percentZeros(index) = (numElements - numZeros) / numElements * 100;
%   percentEnergy(index) = (energy / totalEnergy) * 100;
  percentEnergy(index) = sqrt(energy / totalEnergy) * 100;
  if ((percentZeros(index) >= percentEnergy(index)) && (zerosExceed == 0))
    zerosExceed = index;
  end
  index = index + 1;
end

% compute final threshold by linear interpolation
a = thresholds(zerosExceed - 1); b = thresholds(zerosExceed);
e1 = percentEnergy(zerosExceed - 1); e2 = percentEnergy(zerosExceed);
z1 = percentZeros(zerosExceed - 1); z2 = percentZeros(zerosExceed);

% assume that e1, e2 and z1, z2 each define a line, and that the two lines
% cross between a and b.  If lines don't cross, then it might give a funky result.

balanceThreshold = (e1-z1)*(b-a)/((z2-z1)-(e2-e1)) + a;
balanceEnergy = (z2-z1)/(b-a)*(balanceThreshold-a) + z1;

sqrtBalanceThreshold = sqrt(balanceThreshold*sqrtFactor)/sqrtFactor;
nearZeroThreshold = 0.01;

% find these values by linear interpolation
nearZeroIndex = 1;
sqrtBalanceIndex = 1;
for i=1:numThresholds+1
  if (thresholds(i) < nearZeroThreshold)
    nearZeroIndex = nearZeroIndex + 1;
  end
  if (thresholds(i) < sqrtBalanceThreshold)
    sqrtBalanceIndex = sqrtBalanceIndex + 1;
  end
  if ((thresholds(i) >= nearZeroThreshold) && (thresholds(i) >= sqrtBalanceThreshold))
    break;
  end
end

% now we know that the thresholds in the indices are greater than the given thresholds
if (sqrtBalanceIndex > 1)
  sqrtBalanceEnergy = percentEnergy(sqrtBalanceIndex-1) +...
            (percentEnergy(sqrtBalanceIndex)-percentEnergy(sqrtBalanceIndex-1)) *...
            (sqrtBalanceThreshold-thresholds(sqrtBalanceIndex-1)) /...
            (thresholds(sqrtBalanceIndex)-thresholds(sqrtBalanceIndex-1));
else
  sqrtBalanceEnergy = percentEnergy(sqrtBalanceIndex);
end

if (nearZeroIndex > 1)
  nearZeroEnergy = percentEnergy(nearZeroIndex-1) +...
          (percentEnergy(nearZeroIndex)-percentEnergy(nearZeroIndex-1)) *...
          (nearZeroThreshold-thresholds(nearZeroIndex-1)) /...
          (thresholds(nearZeroIndex)-thresholds(nearZeroIndex-1));
else
  nearZeroEnergy = percentEnergy(nearZeroIndex);
end

% select the threshold
select = strcmp(threshold, thresholdMethod);

if (select(1))
  finalThreshold = balanceThreshold;
  finalEnergy = balanceEnergy;
elseif (select(2))
  finalThreshold = sqrtBalanceThreshold;
  finalEnergy = sqrtBalanceEnergy;
elseif (select(3))
  finalThreshold = nearZeroThreshold;
  finalEnergy = nearZeroEnergy;
else
  disp(['Specified Threshold ' threshold ' not found. Defaulting to balance sparsity-
      norm.']);
  finalThreshold = balanceThreshold;
  finalEnergy = balanceEnergy;
end

% display result
% this is meant to display in the compression script.
figure; subplot(2, 2, 4);
plot(thresholds, percentZeros, thresholds, percentEnergy);
line([finalThreshold, finalThreshold], [0.0, 100.0], 'LineStyle', ':', 'Color', [0.8,
    0.1, 0.1]);
title(['Sparsity-Norm Analysis (threshold = ' num2str(finalThreshold) ', retained
    energy = ' num2str(finalEnergy) ' percent)']);
xlabel('Threshold'); ylabel('Number of Zeros / Percent Energy Retained');
legend('Percent Number of Zeros', 'Percent Energy Retained', 'Calculated Threshold',
    3);
```

Bibliography

[1] Ulrich Baum and Michael Clausen. Computing Irreducible Representations of Supersolvable Groups. *Mathematics of Computation*, 63(207):351–359, July 1994.

[2] Michael Clausen and Ulrich Baum. *Fast Fourier Transforms*. BI-Wissenschaftsverlag, Mannheim, 1993.

[3] Michael Clausen and Ulrich Baum. Fast Fourier Transforms for Symmetric Groups: Theory and Implementation. *Mathematics of Computation*, 61(204):833–847, October 1993.

[4] James W. Cooley and John W. Tukey. An Algorithm for the Machine Calculation of Complex Fourier Series. *Mathematics of Computation*, 19:297–301, April 1965.

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, second edition, 2001.

[6] David S. Dummit and Richard M. Foote. *Abstract Algebra*. Upper Saddle River, N.J., Prentice Hall, second edition, 1999.

[7] Nathaniel Eldredge. An Eigenspace Approach to Isotypic Projections for Data on Binary Trees. Undergraduate Thesis, Harvey Mudd College, 2003.

[8] R. Foote, G. Mirchandani, D. Rockmore, D. Healy, and T. Olson. A Wreath Product Group Approach to Signal and Image Processing. I. Multiresolution Analysis. *IEEE Transactions on Signal Processing*, 48(1):102–132, January 2000.

[9] David M. Goldschmidt. *Group Characters, Symmetric Functions, and the Hecke Algebra*. American Mathematical Society, University Lecture Series, 1993.

[10] D. Healy, G. Mirchandani, T. Olson, and D. Rockmore. Wreath Products for Image Processing. *Proceedings of the ICASSP*, 6:3582–3586, 1996.

[11] Gordon James and Martin Liebeck. *Representations and Characters of Groups*. Cambridge University Press, second edition, 2001.

[12] Adalbert Kerber. *Representations of Permutation Groups I*, volume 240 of *Lecture Notes in Mathematics*. Springer-Verlag, Berlin, 1971.

[13] Adalbert Kerber. *Representations of Permutation Groups II*, volume 495 of *Lecture Notes in Mathematics*. Springer-Verlag, Berlin, 1975.

[14] Peter D. Lax. *Linear Algebra*. John Wiley & Sons, New York, 1997.

[15] Haeyoung Lee, Mathieu Desbrun, and Peter Schröder. Progressive Encoding of Complex Isosurfaces. In *Proceedings of ACM SIGGRAPH*, volume 22, pages 471–476, July 2003.

[16] David K. Maslen and Daniel N. Rockmore. Separation of Variables and the Computation of Fourier Transforms on Finite Groups I. *Journal of the American Mathematical Society*, 10(1):169–214, January 1997.

[17] David K. Maslen and Daniel N. Rockmore. The Cooley-Tukey FFT and Group Theory. *Notices of the American Mathematical Society*, 48(10):1151–1160, November 2001.

[18] G. Mirchandani, R. Foote, D. Rockmore, D. Healy, and T. Olson. A Wreath Product Group Approach to Signal and Image Processing. II. Convolution, Correlation, and Applications. *IEEE Transactions on Signal Processing*, 48(3):749–767, March 2000.

[19] Elizabeth Norton. Data Compression on the Symmetric Group. Undergraduate Thesis, Harvey Mudd College, 2002.

[20] Daniel N. Rockmore. Fast Fourier Transforms for Wreath Products. *Journal of Applied and Computational Harmonic Analysis*, 2(3):279–292, July 1995.

[21] Daniel N. Rockmore. Recent Progress and Applications in Group FFTs. In *Proceedings of the Conference on Computational Noncommutative Algebra and Applications*. NATO Advanced Study Institute, July 2003.

[22] C. Schoolfield. Random Walks on Wreath Products of Groups. *Journal of Theoretical Probability*, 15:667–693, 2002.

[23] H. Straubing. The Wreath Product and Its Applications. In *Formal Properties of Finite Automata and Applications*, 1990.

[24] R. Tolimieri and M. An. Group Filters and Image Processing. In *Proceedings of the Conference on Computational Noncommutative Algebra and Applications*. NATO Advanced Study Institute, July 2003.

[25] James S. Walker. Fourier Analysis and Wavelet Analysis. *Notices of the American Mathematical Society*, 44(6):658–670, June/July 1997.

[26] I. T. Young, J. J. Gerbrands, and L. J. van Vliet. *Fundamentals of Image Processing*. Diktatenverkoop Faculteit Technische Natuurkunde, http://www.ph.tn.tudelft.nl/Courses/FIP/noframes/fip.html, 1995.