3D WAVELET-BASED ALGORITHMS FOR THE COMPRESSION OF
GEOSCIENCE DATA

By

Justin Thomas Rucker

A Thesis
Submitted to the Faculty of
Mississippi State University
in Partial Fulfillment of the Requirements
for the Degree of Master of Science
in Engineering
in the Department of Electrical & Computer Engineering

Mississippi State, Mississippi
December 2005
Copyright by
Justin Thomas Rucker
2005
3D WAVELET-BASED ALGORITHMS FOR THE COMPRESSION OF
GEOSCIENCE DATA

By
Justin Thomas Rucker

Approved:

James E. Fowler
Associate Professor of
Electrical & Computer Engineering
(Director of Thesis)

Nicholas Younan
Professor of Electrical & Computer Engineering
and Graduate Coordinator
(Committee Member)

Roger L. King
Associate Dean of Research and Graduate Studies
James Worth Bagley College of Engineering
(Committee Member)
Name: Justin Thomas Rucker
Date of Degree: December 9, 2005
Institution: Mississippi State University
Major Field: Engineering (Electrical Engineering)
Major Professor: Dr. James E. Fowler
Title of Study: 3D WAVELET-BASED ALGORITHMS FOR THE COMPRESSION OF GEOSCIENCE DATA
Pages in Study: 108
Candidate for Degree of Master of Science

Geoscience applications generate large datasets; thus, compression is necessary to facilitate the storage and transmission of geoscience data. One focus is on the coding of hyperspectral imagery and the prominent JPEG2000 standard. Certain aspects of the encoder, such as rate-allocation between bands and spectral decorrelation, are not covered by the JPEG2000 standard. This thesis investigates the performance of several JPEG2000 encoding strategies. Additionally, a relatively low-complexity 3D embedded wavelet-based coder, 3D-tarp, is proposed for the compression of geoscience data. 3D-tarp employs an explicit estimate of the probability of coefficient significance to drive a nonadaptive arithmetic coder, resulting in a simple implementation suited to vectorized hardware acceleration. Finally, an embedded wavelet-based coder is proposed for the shape-adaptive coding of ocean-temperature data. 3D binary set-splitting with k-d trees, 3D-BISK, replaces the octree splitting structure of other shape-adaptive coders with k-d trees, a simpler set partitioning structure that is well-suited to shape-adaptive coding.
DEDICATION

I would like to dedicate this work to all my friends and family who have been affected by cancer and other life-threatening illnesses. Your strength through these trials and your unrelenting faith in God have been an inspiration to me.
ACKNOWLEDGMENTS

I thank God for continued blessings of knowledge, faith, and perseverance throughout my college career. I would also like to express my appreciation to my major professor, Dr. James E. Fowler, for taking me into his research group as an undergraduate, for providing me with an invaluable research experience, for always having patience with me, and for helping me make this thesis the best that it could possibly be. I would also like to thank the rest of my committee, Dr. Nicholas Younan and Dr. Roger L. King for serving on my committee, for their continued support, and their friendship. Finally, I thank my family, friends, and colleagues for their encouragement, patience, and support throughout my college career.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEDICATION</td>
<td>ii</td>
</tr>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>iii</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>vi</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>vii</td>
</tr>
<tr>
<td>CHAPTER</td>
<td></td>
</tr>
<tr>
<td>I. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>II. BACKGROUND</td>
<td>5</td>
</tr>
<tr>
<td>2.1 Discrete Wavelet Transform (DWT)</td>
<td>5</td>
</tr>
<tr>
<td>2.2 Bitplane Coding</td>
<td>7</td>
</tr>
<tr>
<td>2.3 Significance Map Coding</td>
<td>13</td>
</tr>
<tr>
<td>2.4 Refinement and Sign Coding</td>
<td>13</td>
</tr>
<tr>
<td>2.5 Arithmetic Coding</td>
<td>14</td>
</tr>
<tr>
<td>2.6 Prominent Significance Map Encoding</td>
<td>15</td>
</tr>
<tr>
<td>2.6.1 Runlength Coding</td>
<td>15</td>
</tr>
<tr>
<td>2.6.2 Zerotrees</td>
<td>16</td>
</tr>
<tr>
<td>2.6.3 Density Estimation</td>
<td>20</td>
</tr>
<tr>
<td>2.6.4 Spatial Partitioning</td>
<td>20</td>
</tr>
<tr>
<td>2.6.5 Conditional Coding</td>
<td>23</td>
</tr>
<tr>
<td>2.7 3D Shape-Adaptive Coding</td>
<td>27</td>
</tr>
<tr>
<td>2.7.1 Shape-Adaptive 3D-SPIHT</td>
<td>27</td>
</tr>
<tr>
<td>2.7.2 3D-OB-SPECK</td>
<td>28</td>
</tr>
<tr>
<td>2.7.3 3D-tarp</td>
<td>28</td>
</tr>
<tr>
<td>2.7.4 3D-WDR</td>
<td>29</td>
</tr>
<tr>
<td>2.7.5 EBCOT</td>
<td>29</td>
</tr>
<tr>
<td>CHAPTER</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>------</td>
</tr>
<tr>
<td>III. JPEG2000 ENCODING TECHNIQUES</td>
<td>31</td>
</tr>
<tr>
<td>3.1 JPEG2000 for Single-Component Images</td>
<td>32</td>
</tr>
<tr>
<td>3.2 Spectral Decorrelation for Multiple-Component Images</td>
<td>33</td>
</tr>
<tr>
<td>3.3 Rate-Allocation Strategies Across Multiple Image Components</td>
<td>34</td>
</tr>
<tr>
<td>3.3.1 JPEG2000-BIFR</td>
<td>35</td>
</tr>
<tr>
<td>3.3.2 JPEG2000-BIRA</td>
<td>35</td>
</tr>
<tr>
<td>3.3.3 JPEG2000-MC</td>
<td>36</td>
</tr>
<tr>
<td>IV. 3D-TARP</td>
<td>38</td>
</tr>
<tr>
<td>4.1 Estimation of Probability of Significance via Parzen Windows</td>
<td>39</td>
</tr>
<tr>
<td>4.2 Tarp Filtering</td>
<td>41</td>
</tr>
<tr>
<td>4.2.1 2D tarp Filtering</td>
<td>42</td>
</tr>
<tr>
<td>4.2.2 3D tarp Filtering</td>
<td>43</td>
</tr>
<tr>
<td>4.3 The 3D-tarp Coder</td>
<td>44</td>
</tr>
<tr>
<td>4.4 Parallelized Implementation of 3D-tarp</td>
<td>45</td>
</tr>
<tr>
<td>V. HYPERSPECTRAL COMPRESSION RESULTS</td>
<td>52</td>
</tr>
<tr>
<td>5.1 Data</td>
<td>52</td>
</tr>
<tr>
<td>5.2 Performance Metrics</td>
<td>58</td>
</tr>
<tr>
<td>5.3 Dyadic vs. Wavelet-Packet Transform</td>
<td>61</td>
</tr>
<tr>
<td>5.4 Evaluation of JPEG2000 Encoding Strategies</td>
<td>64</td>
</tr>
<tr>
<td>5.5 Hyperspectral Compression Results</td>
<td>69</td>
</tr>
<tr>
<td>VI. SHAPE-ADAPTIVE 3D BISK</td>
<td>74</td>
</tr>
<tr>
<td>6.1 3D binary set-splitting with k-d trees</td>
<td>75</td>
</tr>
<tr>
<td>6.1.1 Set Partitioning with k-d Trees</td>
<td>75</td>
</tr>
<tr>
<td>6.1.2 The 3D-BISK Algorithm</td>
<td>77</td>
</tr>
<tr>
<td>6.2 Experimental Results</td>
<td>81</td>
</tr>
<tr>
<td>VII. CONCLUSIONS</td>
<td>85</td>
</tr>
<tr>
<td>REFERENCES</td>
<td>88</td>
</tr>
<tr>
<td>APPENDIX</td>
<td></td>
</tr>
<tr>
<td>A. HYPERSPECTRAL UTILITIES</td>
<td>93</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>TABLE</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 POC Performance of 3D-SPIHT with Dyadic and Packet Transforms</td>
<td>63</td>
</tr>
<tr>
<td>5.2 SNR at 1.0 bpppb for SPIHT</td>
<td>63</td>
</tr>
<tr>
<td>5.3 SNR Performances at 1.0 bpppb</td>
<td>65</td>
</tr>
<tr>
<td>5.4 POC Performances for the Various JPEG2000 Encoding Strategies</td>
<td>68</td>
</tr>
<tr>
<td>5.5 POC Performance at 1.0 bpppb for the Various Coders</td>
<td>73</td>
</tr>
<tr>
<td>5.6 SNR at 1.0 bpppb for the Packet Transform</td>
<td>73</td>
</tr>
<tr>
<td>6.1 Comparison of 3D-BISK Set-Splitting Strategies at 1.0 bpv</td>
<td>83</td>
</tr>
<tr>
<td>6.2 Distortion Performance at 1.0 bpv</td>
<td>84</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Overview of modern 3D embedded wavelet-based coders.</td>
</tr>
<tr>
<td>2.2</td>
<td>One stage of 2D DWT decomposition composed of lowpass (LPF) and highpass (HPF) filters applied to the columns and rows independently.</td>
</tr>
<tr>
<td>2.3</td>
<td>A 3-scale, 2D DWT pyramid arrangement of subbands.</td>
</tr>
<tr>
<td>2.4</td>
<td>3-level, 3D dyadic DWT.</td>
</tr>
<tr>
<td>2.5</td>
<td>3D packet DWT, with $m = 3$ spatial decompositions and $n = 3$ spectral decompositions.</td>
</tr>
<tr>
<td>2.6</td>
<td>Bitplanes of the sign-magnitude representation of coefficients for bitplane coding.</td>
</tr>
<tr>
<td>2.7</td>
<td>Successive-approximation quantization of a coefficient magnitude $</td>
</tr>
<tr>
<td>2.8</td>
<td>Parent-child relationships between subbands of a 2D DWT.</td>
</tr>
<tr>
<td>2.9</td>
<td>Processing of sorted lists in SPIHT.</td>
</tr>
<tr>
<td>2.10</td>
<td>2D quadtree block partitioning as performed in 2D SPECK.</td>
</tr>
<tr>
<td>2.11</td>
<td>3D octree cube partitioning as performed in 3D SPECK.</td>
</tr>
<tr>
<td>2.12</td>
<td>Significance-state neighbors to $x_i$.</td>
</tr>
<tr>
<td>2.13</td>
<td>The AAC contexts for JPEG2000.</td>
</tr>
<tr>
<td>4.1</td>
<td>Pseudocode for the 2D tarp filter of [28]. The image is of size $N_1 \times N_2$.</td>
</tr>
<tr>
<td>4.2</td>
<td>Pseudocode for the 3D tarp filter. The volume is of size $N_1 \times N_2 \times N_3$.</td>
</tr>
</tbody>
</table>
4.3 The 3D Laplacian window for $\alpha = 0.5$. The boxed value indicates the window origin $(x_1, x_2, x_3 = 0)$.

4.4 Pseudocode for the vectorized 3D tarp filter for SIMD architectures. All buffer volumes initialized to zero at algorithm start. The “:” indicates vectorization along the corresponding dimension. $p'_1$ is $p_1$ offset by a one-column shift to the right; i.e., $p'_1[x_1, x_2, x_3] = p_1[x_1, x_2 - 1, x_3]$. This shift is accomplished during loading of the vector.

5.1 False-color image of Moffett

5.2 False-color image of Jasper Ridge

5.3 False-color image of Cuprite

5.4 False-color image of Low Altitude

5.5 False-color image of Lunar Lake

5.6 Classification map for the Moffett image using $k$-means classification. (a) Map for original image, (b) map after JPEG2000-BIFR compression.

5.7 Comparison of the typical rate-distortion performance for the dyadic transform of Fig. 2.4 versus that of the packet transform of Fig. 2.5. This plot is for the Moffett image using 3D-SPIHT.

5.8 Rate-distortion performance of JPEG2000 encoding techniques for Moffett.

5.9 Rate-distortion performance of JPEG2000 encoding techniques for Jasper Ridge.

5.10 Rate-distortion performance of JPEG2000 encoding techniques for Cuprite.

5.11 Rate-distortion performance of all techniques for Moffett.

5.12 Rate-distortion performance of all techniques for Cuprite.

5.13 Rate-distortion performance of all techniques for Jasper Ridge.

6.1 Set Partitioning in 2D-BISK

6.2 Set Partitioning in 3D-BISK
6.3 Rate-distortion performance for \texttt{adrtc} and \texttt{bisca}. ............ 83
Applications involving geoscience data, such as military surveillance, environmental monitoring, and classification, have become more prevalent. Geoscience datasets are often three-dimensional data cubes which require a significant amount of storage space. For instance, hyperspectral imagery is generated by collecting hundreds of contiguous bands; uncompressed hyperspectral imagery can be very large, with a single image potentially occupying hundreds of megabytes. The Airborne Visible InfraRed Imaging Spectrometer (AVIRIS) sensor, for example, is capable of collecting several gigabytes of data per day. Thus, compression is necessary to facilitate both the storage and the transmission of geoscience data.

Currently there are no standards, or even commonly accepted methodologies, for the compression of geoscience data. However, several techniques, such as vector quantization (VQ), principle component analysis (PCA), and trellis coding, have been proposed for the compression of hyperspectral imagery. These solutions perform well, however the nature of their bitstreams does not allow for successive reconstructions of the data, i.e. progressive transmission. In progressive transmission, the receiver can produce a low-quality representation of the data after having received only a small portion of the transmitted bitstream, and this “preview” of the data can be successively refined as more and more of the bitstream is received. Modern compression techniques support progressive transmission through the use of embedded coding, which is any
coding such that 1) any prefix of length $N$ bits of an $M$-bit coding is also a valid coding of the entire dataset, $0 < N \leq M$; and 2) if $N' > N$, then the distortion upon reconstructing from the length-$N'$ prefix is less than or equal to that associated with the length-$N$ prefix. To implement embedded coding, information must be organized in the bitstream in decreasing order of importance, where the most important information is that which produces the greatest reduction in distortion upon reconstruction. Although it is usually not possible to exactly achieve this ordering in practice, modern embedded image-compression algorithms can approximate this optimal embedded ordering [1].

Given the success of embedded wavelet-based coders such as JPEG2000 [2–4] and SPIHT [5] for 2D images, creating 3D versions of such coders constitutes a reasonable approach for the embedded compression of geoscience data.

In addition, the compression of imagery with arbitrary shape has become an important issue in several multimedia application areas, with the recent MPEG-4 video-coding standard [6] being the prime example. However, certain geoscience applications have also benefited from such shape-adaptive coding. For example, the US Naval Oceanographic Office (NAVOCEANO) generates three-dimensional oceanographic temperature datasets for rectangular regions of sea and land at standard ocean depths. Data that refers to land or points beyond the bathymetry are considered to have no valid data. Thus, the compression of such ocean-temperature data requires shape-adaptive coding. A key process in embedded wavelet-based coders is the mapping of the significance state of each wavelet coefficient (i.e., whether or not the coefficient is greater than or less than the current threshold) into a binary-valued significance map with the threshold decreasing for each successive pass through the dataset. Such coders can easily be made shape-adaptive and applied to the ocean-temperature compression problem by employing a 3D shape-adaptive wavelet transform [7] which transforms only the ocean regions; land regions, where no data exists, are permanently considered
insignificant in the significance map. The major difference between wavelet-based compression schemes lies in the method for coding the significance map; consequently, the key to shape-adaptive coding is to modify this significance-map encoding to accommodate the presence of land regions wherein no valid data lies.

This thesis considers several issues surrounding the embedded coding of geoscience data, with particular focus on 3D coding of geoscience data, with particular focus on 3D coding algorithms for hyperspectral imagery and arbitrarily shaped ocean-temperature volumes. A primary contribution of this thesis is the investigation of strategies for coding multiple-component data with JPEG2000. Since the JPEG2000 standard only covers the decoder, certain encoder issues, such as rate allocation and spectral decorrelation, are not addressed and thus, must be determined by the algorithm designer. Results, first presented in [8] and included in this thesis, indicate that performance varies greatly, depending on the implementation of these encoder design details.

Another primary contribution of this thesis is the development of a relatively low-complexity 3D coder. 3D-tarp, first proposed in [9, 10], offers a minimal reduction in performance as compared to more complex coders. In addition, 3D-tarp can be easily parallelized, which renders it amenable to, real-time implementation onboard data-collecting platforms, such as satellites.

A significant contribution of this thesis is a comprehensive body of results for 3D wavelet-based compression algorithms. While others have made similar comparisons on the same data used in this thesis, the results contained in Chap. V are, to our knowledge, the most comprehensive evaluation of 3D wavelet-based compression techniques in the literature. Additionally, the body of results in Sec. 6.2 is one of three known places where shape-adaptive coding has been considered for ocean-temperature data. The earlier study, [11], introduced the wavelets around land masses (WAVAL) algorithm,
while the results of Sec. 6.2 presents an updated, more complete version of results first appearing in [12].

The final contribution of this thesis is the development of 3D Binary Set Splitting in $k$-d Trees (3D-BISK) algorithm, a 3D shape-adaptive coder for ocean temperature data. 3D-BISK, which was initially published in [12], replaces the octree set-partitioning operation of 3D-SPECK with $k$-d trees [13], a simpler set decomposition particularly well-suited to shape-adaptive coding due to its greater flexibility at capturing arbitrarily shaped regions. Additionally, 3D-BISK aggressively discards land regions from consideration by shrinking the decomposed sets to the bounding box of their ocean regions. 3D-BISK has low complexity as compared with other techniques and almost always offers superior performance.

In the next chapter, we will discuss the major components of modern 3D embedded wavelet-based coders, discuss shape adaptive coding, and present previous work. We will see that most modern 3D embedded wavelet-based coders differ only in how the significance information is coded. We will explore several popular approaches to the coding of significance information in Sec. 2.6. In Chap. III, we will discuss using JPEG2000 for hyperspectral image compression and examine different encoding strategies for JPEG2000. In Chap. IV, 3D-tarp [9, 10], a low complexity 3D wavelet based coder is proposed for the compression of hyperspectral imagery. Data, performance metrics, and experimental results are presented in Chap. V. In Chap. VI, 3D Binary Set Splitting with $k$-d Trees (3DBISK) [12] is proposed for the shape-adaptive coding of ocean temperature data. Finally, concluding remarks are made in Chap. VII.
CHAPTER II
BACKGROUND

Modern wavelet-based coders are based upon four major precepts: a wavelet transform; significance-map encoding; successive-approximation coding, i.e., bit-plane coding; and some form of entropy coding, most often arithmetic coding. This coding process is depicted in Fig. 2.1, the components of which are described in detail below. We note that a significant portion of this discussion originated in [14].

2.1 Discrete Wavelet Transform (DWT)

Transforms aid the establishment of an embedded coding in that low-frequency components typically contain the majority of signal energy and are thus, more important than high-frequency components to reconstruction. Wavelet transforms are currently the transform of choice for modern 2D image coders, since they not only provide this partitioning of information in terms of frequency but also retain much of the spatial structure of the original image. Wavelet-based coders for geoscience data extend the 2D transform structure into three dimensions.

A 2D discrete wavelet transform (DWT) can be implemented as a filter bank as illustrated in Fig. 2.2. This filter bank decomposes the original image into horizontal \((H)\), vertical \((V)\), diagonal \((D)\), and baseband \((B)\) subbands, each being one-fourth the size of the original image. Wavelet theory provides filter-design methods such that the filter bank is perfectly reconstructing (i.e., there exists a reconstruction filter bank that will generate exactly the original image from the decomposed subbands \(H, V, D,\) and
and such that the lowpass and highpass filters have finite impulse responses (which aids practical implementation). Multiple stages of decomposition can be cascaded together by recursively decomposing the baseband; the subbands in this case are usually arranged in a pyramidal form as illustrated in Fig. 2.3.

For geoscience data, the 2D-transform decomposition of Fig. 2.3 is extended to three dimensions to accommodate the addition of the spectral dimension. A 3D wavelet transform, like the 2D transform, is implemented in separable fashion, employing 1D transforms separately in the spatial-row, spatial-column, and spectral-slice directions. However, the addition of a third dimension permits several options for the order of decomposition. For instance, we can perform one scale of decomposition along each direction, then do further decomposition in the baseband subband, leading to the dyadic decomposition, as is illustrated in Fig. 2.4. This dyadic decomposition structure is the most straightforward 3D generalization of the 2D dyadic decomposition of Fig. 2.3. However, in 3D, we can alternatively use a so-called wavelet-packet transform, in which we first decompose each spectral slice using a separable 2D transform and then follow with a 1D decomposition in the spectral direction. With this approach, we employ an $m$-scale decomposition spatially, followed by an $n$-scale decomposition spectrally, where it is possible for $m \neq n$. For example, the wavelet-packet transform depicted in Fig. 2.5 uses a three-scale decomposition ($m = n = 3$) in all directions. In comparing the two decomposition structures, the wavelet-packet transform is more flexible, because the spectral decomposition can be better tailored to the data at hand than in the dyadic transform. In Sec. 5.3, we will see that this wavelet-packet decomposition typically yields more efficient coding for hyperspectral datasets than the dyadic decomposition.

Wavelet-based coders, 2D or 3D, base their operation on the following observations about the DWT: 1) since most images are lowpass in nature, most signal energy is compacted into the baseband and lower-frequency subbands; 2) most coefficients are
zero in the higher-frequency subbands; 3) small- or zero-valued coefficients tend to be clustered together within a given subband; and 4) clusters of small- or zero-valued coefficients in one subband tend to be located in the same relative spatial position as similar clusters in subbands of the next decomposition scale. The techniques we describe in Sec. 2.6 exploit one or more of these DWT properties to achieve efficient coding performance.

2.2 Bitplane Coding

The partitioning of information into DWT subbands somewhat inherently supports embedded coding in that transmitting coefficients by ordering the subbands as $B_J$, $H_J$, $V_J$, $D_J$, $H_{J-1}$, $V_{J-1}$, $D_{J-1}$, . . . , implements a decreasing order of importance. However, more is needed to produce a truly embedded bitstream—even if coefficient $c_i \in S_j$ is more important than coefficient $c_k \in S_j$, not every bit of $c_i$ is necessarily more important than every bit of $c_k$. That is, not only should the coefficients be transmitted in decreasing order of importance, but also the individual bits that constitute the coefficients should be ordered as well.

Specifically, to effectuate an embedded coding of a set of coefficients, we represent the coefficients in sign-magnitude form as illustrated in Fig. 2.6 and code the sign and magnitude of the coefficients separately. For coefficient-magnitude coding, we transmit the most significant bit (MSB) of all coefficient magnitudes, then the next-most significant bit of all coefficient magnitudes, etc., such that each coefficient is successively approximated. This “bitplane-coding” scheme is contrary to the usual binary representation which would output all bits of $|c_0|$, then all bits of $|c_1|$, etc. The net effect of the bitplane coding is that each coefficient magnitude is successively quantized by dividing the interval in which it is known to reside in half and outputting a bit to designate the appropriate subinterval, as illustrated in Fig. 2.7.
Figure 2.1: Overview of modern 3D embedded wavelet-based coders.

Figure 2.2: One stage of 2D DWT decomposition composed of lowpass (LPF) and highpass (HPF) filters applied to the columns and rows independently.
Figure 2.3: A 3-scale, 2D DWT pyramid arrangement of subbands.
Figure 2.4: 3-level, 3D dyadic DWT.

Figure 2.5: 3D packet DWT, with $m = 3$ spatial decompositions and $n = 3$ spectral decompositions.
Coefficients

<table>
<thead>
<tr>
<th>Sign</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSB</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>LSB</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 2.6: Bitplanes of the sign-magnitude representation of coefficients for bitplane coding.

Figure 2.7: Successive-approximation quantization of a coefficient magnitude $|c|$ in interval $[0, T]$ where $T$ is an integer power of 2.
In practice, bitplane coding is usually implemented by performing two passes through the set of coefficients for each bitplane—the significance pass and the refinement pass. We define the significance state \( x_i \) with respect to threshold \( t \) of coefficient \( c_i \) as \( x_i = 1 \) if \( |c_i| \geq t \) (i.e., \( c_i \) is a significant coefficient), and \( x_i = 0 \) otherwise (i.e., \( c_i \) is insignificant). The significance pass describes \( x_i \) for all the coefficients in the DWT that are currently known to be insignificant but may become significant for the current threshold. On the other hand, the refinement pass produces a successive approximation to those coefficients that are already known to be significant by coding the current coefficient-magnitude bitplane for those significant coefficients. After each iteration of the significance and refinement passes, the significance threshold is divided in half, and the process is repeated for the next bitplane.
2.3 Significance Map Coding

The collection of $s[x_1, x_2, x_3]$ values for all the coefficients in the DWT of an image is called the significance map for a particular threshold value. Given our observations in Sec. 2.1 of the nature of DWT coefficients, we see that for most of the bitplanes (particularly for large $t$), the significance map will be only sparsely populated with nonzero values. Consequently, the task of the significance pass is to create an efficient coding of this sparse significance map at each bitplane; the efficiency of this coding will be crucial to the overall compression efficiency of the coder. Sec. 2.6 is devoted to reviewing approaches that prominent algorithms have taken for the efficient coding of significance-map information. These algorithms are largely 2D image coders which have been extended to 3D and modified to accommodate the addition of spectral information.

2.4 Refinement and Sign Coding

In most embedded image coders, after the significance map is coded for a particular bitplane, a refinement pass proceeds through the coefficients, coding the current bitplane value of each coefficient that is already known to be significant but did not become significant in the immediately preceding significance pass. These “refinement bits” permit the reconstruction of the significant coefficients with progressively greater accuracy. It is usually assumed that the occurrence of a 0 or 1 is equally likely in bitplanes other than the MSB for a particular coefficient; consequently, most algorithms take little effort to code the refinement bits and may simply output them unencoded into the bitstream. Recently, it has been recognized that the refinement bits possess some correlation to their neighboring coefficients [15], particularly for the more significant bitplanes. The significance and refinement passes encode the coefficient magnitudes; to reconstruct the wavelet coefficients, the coefficient signs must also be encoded. As with
the refinement bits, most algorithms assume that any given coefficient is equally likely to be positive or negative; however, recent work [15–17] has shown that there is some structure to the sign information that can be exploited to improve coding efficiency.

2.5 Arithmetic Coding

Most wavelet-based coders incorporate some form of lossless entropy coding at the final stage before producing the compressed bitstream. In essence, such entropy coders assign shorter bitstream codewords to more frequently occurring symbols in order to maximize the compactness of the bitstream representation.

Most wavelet-based coders use adaptive arithmetic coding (AAC) [18] for lossless entropy coding. AAC codes a stream of symbols into a bitstream with length very close to its theoretical minimum limit. Suppose source $X$ produces symbol $i$ with probability $p_i$. The entropy of source $X$ is defined to be

$$H(X) = - \sum_i p_i \log_2 p_i,$$

where $H(X)$ has units of bits per symbol (bps). One of the fundamental tenets of information theory is that the average bit rate in bps of the most efficient lossless (i.e., invertible) compression of source $X$ cannot be less than $H(X)$. In practice, AAC often produces compression quite close to $H(X)$ by estimating the probabilities of the source symbols with frequencies of occurrence, as it codes the symbol stream. Essentially, the better able AAC can estimate $p_i$, the closer it will come to the $H(X)$ lower bound on compression efficiency. Oftentimes, the efficiency of AAC can be improved by conditioning the coder with known context information and maintaining separate symbol-probability estimates for each context. That is, limiting the attention of AAC to a specific context usually reduces the variety of symbols, thus permitting better
estimation of the probabilities within that context and producing greater compression efficiency. From a mathematical standpoint, the conditional entropy of source $X$ with known information $Y$ is $H(X|Y)$. Since it is well known from information theory that

$$H(X|Y) \leq H(X),$$

conditioning AAC with $Y$ as the context will (usually) produce a bitstream with a smaller bit rate.

### 2.6 Prominent Significance Map Encoding Techniques

The primary difference between wavelet-based coding algorithms is how coding of the significance map is performed. Several significance-map coding techniques that have been used for hyperspectral imagery are discussed below. Typically, these techniques are originally developed for 2D images and then subsequently extended and modified for 3D coding. As a consequence, we briefly overview the original 2D algorithm—which is usually more easily conceptualized—before discussing its 3D extension for each of the techniques considered below.

#### 2.6.1 Runlength Coding

Since, for a given significance threshold, the significance map is essentially a binary image, techniques that have long been employed for the coding of bilevel images are applicable. Specifically, runlength coding is the fundamental compression algorithm behind the Group 3 fax standard; the Wavelet Difference Reduction (WDR) [19] combines runlength coding of the significance map with an efficient lossless representation of the runlength symbols to produce an embedded image coder. Originally developed for 2D imagery in [19], WDR was extended to 3D as an implementation in QccPack [20]; this 3D extension merely deploys runlength scanning
as a 3D raster scan of each subband of the 3D DWT, which is easily accomplished in either dyadic or packet DWT decompositions.

### 2.6.2 Zerotrees

Zerotrees are one of the most widely used techniques for coding significance maps in wavelet-based coders. Zerotrees capitalize on the fact that insignificant coefficients tend to cluster together within a subband, and clusters of insignificant coefficients tend to be located in the same location within subbands of different scales. As illustrated for a 2D DWT in Fig. 2.8, “parent” coefficients in a subband can be related to four “children” coefficients in the same relative spatial location in a subband at the next scale. A zerotree is formed when a coefficient and all of its descendants are insignificant with respect to the current threshold, while a zerotree root is defined to be a coefficient that is part of a zerotree yet is not the descendant of another zerotree root.

The Embedded Zerotree Wavelet (EZW) algorithm [21] was the first 2D image coder to make use of zerotrees for the coding of significance-map information. This coder is based on the observation that if a coefficient is found to be insignificant, it is likely that its descendants are also insignificant. Consequently, the occurrence of a zerotree root in the baseband or in the lower-frequency subbands can lead to substantial coding efficiency since we can denote the zerotree root as a special “Z” symbol in the significance map, and not code all of the descendants which are known then to be insignificant by definition. The EZW algorithm then proceeds to code the significance map in a raster scan within each subband, starting with the baseband and progressing to the high-frequency subbands. In this raster scan a significant coefficient is denoted by either a “+” or “−” symbol, depending on whether the coefficient value is positive or negative, while zerotree roots are denoted by the “Z” symbol and isolated insignificant coefficients (i.e., insignificant coefficients not forming a zerotree root) are
denoted by the “I” symbol. A lossless entropy coding of this symbol stream then produces a compact representation of the significance map. The significance threshold is halved, and the zerotree coding process is repeated for each successive bitplane. Note that, once a coefficient becomes significant and is coded with a “+” or “−,” no further information concerning that coefficient need be coded in the significance pass for subsequent bitplanes.

The Set Partitioning in Hierarchical Trees (SPIHT) algorithm [5] improves upon the zerotree concept by replacing the raster scan with a number of sorted lists that contain sets of coefficients (i.e., zerotrees) and individual coefficients. These lists are illustrated in Fig. 2.9. In the significance pass of the SPIHT algorithm, the list of insignificant sets (LIS) is examined in regard to the current threshold; any set in the list that is no longer a zerotree with respect to the current threshold is then partitioned into one or more smaller zerotree sets, isolated insignificant coefficients, or significant coefficients. Isolated insignificant coefficients are appended to the list of insignificant pixels (LIP), while significant coefficients are appended to the list of significant pixels (LSP). The LIP is also examined, and, as coefficients become significant with respect to the current threshold, they are appended to the LSP. Binary symbols are encoded to describe motion of sets and coefficients between the three lists. Since the lists remain implicitly sorted in an importance ordering, SPIHT achieves a high degree of embedding and compression efficiency.

Originally developed for 2D images, SPIHT has been extended to 3D in several contexts [22–27]. In the case of a dyadic transform such as in Fig. 2.4, the zerotree is a straightforward extension to 3D of the parent-child relationship of 2D zerotrees; that is, one coefficient is the parent to a $2 \times 2 \times 2$ cube of eight offspring coefficients in the next scale. However, in the case of a wavelet-packet transform, there are several approaches to fitting a zerotree structure to the wavelet coefficients. The first, proposed
in [24], recognizes that wavelet-packet subbands appear as “split” versions of their dyadic counterparts, consequently, one should “split” the $2 \times 2 \times 2$ offspring nodes of the dyadic zerotree structure appropriately. An alternative zerotree structure for packet transforms was proposed originally in [25], and was subsequently used in [26, 27]. In essence, this zerotree structure consists of 2D zerotrees within each “slice” of the subband-pyramid volume, with parent-child relationships setup between the tree-root coefficients of the 2D trees. Cho and Pearlman [26] called this alternative structure an “asymmetric” packet zerotree, with the original splitting-based packet structure in [24] then being a “symmetric” packet zerotree. The asymmetric structure usually offers somewhat more efficient compression performance than symmetric packet structures [25–27]. Additionally, the wavelet-packet transform can have the number of spectral decomposition levels different from the number of spatial decomposition levels when the asymmetric tree is used; whereas, the number of spatial and spectral decompositions must be the same to use the symmetric-packet zerotree structure.
Figure 2.8: Parent-child relationships between subbands of a 2D DWT.

Figure 2.9: Processing of sorted lists in SPIHT.
2.6.3 Density Estimation

An all-together different approach to significance-map coding was proposed in [28] wherein an explicit estimate of the probability of significance of wavelet coefficients is used to code the significance map. Specifically, the significance state of a set of coefficients for a given threshold is coded via a raster scan through the coefficients. For coding efficiency, an entropy coder codes the significance state for each coefficient, using the probability that that the coefficient is significant as determined by the density-estimation procedure. The density estimate is in the form of a multidimensional convolution implemented as a sequence of 1D filtering operations coined tarp filtering. In [28], the tarp-filtering procedure is originally described for 2D image coding; 3D-tarp, with the tarp-filtering procedure suitably extended to three dimensions, was proposed in [9, 10].

2.6.4 Spatial Partitioning

Another approach to significance-map coding is spatial partitioning:. The Set-Partitioning Embedded Block Coder (SPECK) [29, 30], originally developed as a 2D image coder, employs quadtree partitioning (see Fig. 2.10) in which the significance state of an entire block of coefficients is tested and coded. Then, if the block contains at least one significant coefficient, the block is subdivided into four subblocks of approximately equal size, and the significance-coding process is repeated recursively on each of the subblocks.

In 2D-SPECK, there are two types of sets: $S$ sets and $I$ sets. The first $S$ set is the baseband, and the first $I$ set contains everything that remains. There are also two linked lists in SPECK: the List of Insignificant Sets (LIS), which contains sorted lists of decreasing sizes that have not been found to contain a significant pixel as compared with the current threshold, and the List of Significant Pixels (LSP), which contains single
pixels that have been found to be significant through sorting and refinement passes. An $S$ set remains in the LIS until it is found to be significant against the current threshold. The set is then divided into four approximately equal-sized sets, and the significance of each of the resulting four sets is tested. If the set is not significant, then it is placed in its appropriate place in the LIS. If the set is significant and contains a single pixel, it is appended to the LSP; otherwise, the set is recursively split into four subsets. Following the significant pass, the coefficients in the LSP go through a refinement pass in which coefficients that have been previously found to be significant are refined.

The SPECK algorithm was extended to 3D in [31, 32] by replacing quadtrees with octrees as illustrated in Fig. 2.11. Unlike the original 2D-SPECK algorithm, the 3D-SPECK algorithm uses only one type of set, rather than having $S$ and $I$ sets as in 2D-SPECK. Consequently, each subband in the DWT decomposition is added to an LIS at the start of the 3D-SPECK algorithm, whereas the 2D algorithm initializes with only the baseband subband in an LIS. An advantage of the set-partitioning processing of 3D-SPECK is that sets are confined to reside within a single subband at all times throughout the algorithm, whereas sets in SPIHT (i.e., the zerotrees) span across scales. This is beneficial from a computational standpoint as the coder need only buffer a single subband at a given time, leading to reduced dynamic memory needed [30]. Furthermore, 3D-SPECK is easily applied to both the dyadic and packet transform structures of Figs. 2.4 and 2.5 with no algorithmic differences.
Figure 2.10: 2D quadtree block partitioning as performed in 2D SPECK.

Figure 2.11: 3D octree cube partitioning as performed in 3D SPECK.
2.6.5 Conditional Coding

Recent work [33] has indicated that typically the ability to predict the insignificance of a coefficient through parent-child relationships such as those employed by zerotree algorithms is somewhat limited compared to the predictive ability of neighboring coefficients within the same subband. Consequently, recent algorithms, such as SPECK, have focused on coding significance-map information using only within-subband information. Another approach to within-subband coding is to employ extensively conditioned, multiple-context AAC to capitalize on the theoretical advantages conditioning provides for entropy coding as discussed in Sec. 2.5.

The usual approach to employing AAC with context conditioning for the significance-map coding of an image is to use the known significance states of neighboring coefficients to provide the context for the coding of the significance state of the current coefficient. Assuming a 2D image, the eight neighboring significance states to $x_i$ are shown in Fig. 2.12. Given that each neighbor takes on a binary value, there are $2^8 = 256$ possible contexts.

JPEG2000 [2–4], the most prominent conditional-coding technique, uses contexts derived from the neighbors depicted in Fig. 2.12, but reduces the number of distinct contexts to nine, since not all possible contexts were found to be useful. The context definitions, which vary from subband to subband, are shown in Fig. 2.13. To further improve the context conditioning, as well as to increase the degree of embedding, JPEG2000 splits the coding of the significance map into two separate passes rather than employ one significance pass as do most other algorithms. Specifically, JPEG2000 uses a significance-propagation pass that codes those coefficients that are currently insignificant but have at least one neighbor that is already significant. This pass accounts for all coefficients that are likely to become significant in the current bitplane. The
remaining insignificant coefficients are coded in the cleanup pass; these coefficients, which are surrounded by insignificant coefficients, are likely to remain insignificant. Both passes use the same nine contexts depicted in Fig. 2.13. In addition, the cleanup pass includes one additional context used to encode four successive insignificant coefficients together with a single “insignificant run” symbol.

To code a single-component image, a JPEG2000 encoder first performs a 2D wavelet transform on the image and then partitions each transform subband into small, 2D rectangular blocks called codeblocks, which are typically of size $32 \times 32$ or $64 \times 64$ pixels. Subsequently, the JPEG2000 encoder independently generates an embedded bitstream for each codeblock. To assemble the individual codeblock bitstreams into a single, final bitstream, each codeblock bitstream is truncated in some fashion, and the truncated bitstreams are concatenated together to form the final bitstream. The method for codeblock-bitstream truncation is an implementation issue concerning only the encoder, as codeblock-bitstream lengths are conveyed to the decoder as header information. Consequently, this truncation process is not covered by the JPEG2000 standard.

It is highly likely that, for codeblocks residing in a single image component, any given JPEG2000 encoder with perform a Lagrangian rate-distortion optimal truncation as described as part of Taubman’s EBCOT algorithm [4, 15]. This optimal truncation technique, post-compression rate-distortion (PCRD) optimization, is a primary factor in the excellent rate-distortion performance of the EBCOT algorithm. PCDR optimization is performed simultaneously across all of the codeblocks from the image, producing an optimal truncation point for each codeblock. The truncated codeblocks are then concatenated together to form a single bitstream. The PCDR optimization, in effect, distributes the total rate for the image spatially across the codeblocks in a rate-distortion-
optimal fashion such that codeblocks with higher energy, which tend to more heavily influence the distortion measure, tend to receive greater rate.

As described in the standard, JPEG2000 is, in essence, a 2D image coder. Although the standard does make a few provisions for multicomponent/multiband, geoscience data, the core coding procedure is based on within band coding of 2D blocks as described above. Furthermore, the exact procedure employed for 3D imagery (e.g., 3D wavelet transform and PCRD optimization across multiple bands) largely entails design issues for the encoder and thus, lies outside the realm of the JPEG2000 standard, which covers the decoder only. Given the increasing prominence that JPEG2000 is garnering for the coding of hyperspectral imagery, we return to consider these encoder-centric issues in depth in Chap. III. Finally, we note that truly 3D coding, consisting of AAC coding of 3D codeblocks as in [34], has been proposed as JPEG2000 Part 10 (JP3D), an extension to the core JPEG2000 standard. However, at the time of this writing, this proposed extension is in the preliminary stages of development, and currently, JPEG2000 for hyperspectral imagery is employed as discussed in Chap. III.
Figure 2.12: Significance-state neighbors to $x_i$.

<table>
<thead>
<tr>
<th>$d_0$</th>
<th>$v_0$</th>
<th>$d_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$h_0$</td>
<td>$x_i$</td>
<td>$h_1$</td>
</tr>
<tr>
<td>$d_2$</td>
<td>$v_1$</td>
<td>$d_3$</td>
</tr>
</tbody>
</table>

Figure 2.13: The AAC contexts for JPEG2000.

<table>
<thead>
<tr>
<th>$B_j$ and $V_j$ subbands</th>
<th>$H_j$ subbands</th>
<th>$D_j$ subbands</th>
<th>Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\sum h_i \sum v_i \sum d_i$</td>
<td>$\sum h_i \sum v_i \sum d_i$</td>
<td>$\sum (h_i + v_i) \sum d_i$</td>
<td>$\geq 3$</td>
</tr>
<tr>
<td>2 $\geq 1$</td>
<td>2 $\geq 1$</td>
<td>2 $\geq 3$</td>
<td>8</td>
</tr>
<tr>
<td>1 1 1</td>
<td>0 1 1</td>
<td>$\geq 1$</td>
<td>7</td>
</tr>
<tr>
<td>1 0 $\geq 1$</td>
<td>0 1 0</td>
<td>0 2</td>
<td>6</td>
</tr>
<tr>
<td>1 0 0</td>
<td>0 1 0</td>
<td>$\geq 2$</td>
<td>5</td>
</tr>
<tr>
<td>0 2</td>
<td>2 0</td>
<td>1 1</td>
<td>4</td>
</tr>
<tr>
<td>0 1</td>
<td>1 0</td>
<td>0 1</td>
<td>3</td>
</tr>
<tr>
<td>0 0 $\geq 2$</td>
<td>0 0 $\geq 2$</td>
<td>$\geq 2$</td>
<td>2</td>
</tr>
<tr>
<td>0 0 1</td>
<td>0 0 1</td>
<td>1 0</td>
<td>1</td>
</tr>
<tr>
<td>0 0 0</td>
<td>0 0 0</td>
<td>0 0</td>
<td>0</td>
</tr>
</tbody>
</table>
2.7 3D Shape-Adaptive Coding

The problem of the shape-adaptive coding of ocean-temperature imagery was considered in [11, 35], wherein the modern paradigm of embedded wavelet-based coding—at the time quickly becoming the preferred approach to the compression of 2D images—was adapted to 3D ocean-temperature imagery with arbitrary shape. Since that time, a number of 3D embedded wavelet-based techniques with improved performance have been proposed, albeit in the context of traditional, rectangular imagery. Yet, these coders all have a common design which can be easily made shape-adaptive.

In this section, we review a number of prior approaches to 3D shape-adaptive coding. Essentially, 3D shape-adaptive coders are direct extensions to 3D of algorithms developed for 2D imagery with arbitrary shape. In the case of ocean temperature data, the straightforward approach to shape-adaptive coding involves applying a transform to only the valid ocean data and treating the remaining land regions as permanently “insignificant.” The bitplane-coding passes can then process these land regions in the same way as other insignificant coefficients. While most shape-adaptive coders are based on this general idea, a number of approaches employ various modifications to the significance-map encoding to increase performance.

2.7.1 Shape-Adaptive 3D-SPIHT

SPIHT [5] is one of the most prominent embedded wavelet-based coders for 2D images; it was extended to 3D in [24] and made shape adaptive in [36]. Significance-map encoding for 3D-SPIHT involves the coding of the insignificance of entire tree-structured sets (zerotrees) across multiple scales of a wavelet transform. The shape-adaptive version of 3D-SPIHT follows the straightforward approach described above by aggregating large land regions together with insignificant ocean regions into zerotree sets. Further refinement to the algorithm can be made by discarding subtrees consisting
entirely of land regions (which are permanently insignificant) from further consideration [36].

2.7.2 3D-OB-SPECK

The state-of-the-art SPECK algorithm [29, 30] algorithm eliminates the cross-scale aggregation of coefficients that occurs in SPIHT and other wavelet-based algorithms and instead applies partitioning to sets of contiguous coefficients within each individual subband. The shape-adaptive version, object-based SPECK (OB-SPECK) [37] is extended to 3D (3D-OB-SPECK) in [31], where the significance state of an entire set is tested and coded, then, if the set was known to contain at least one significant coefficient, the set is split into eight subsets (i.e., octree partitioning), and the process is repeated for each subset. 3D-OB-SPECK is similar to shape-adaptive 3D-SPIHT in that land is considered to be permanently insignificant, and, when a set contains only land coefficients, it is removed from further consideration.

2.7.3 3D-tarp

A unique approach to significance-map coding, tarp coding [28], uses a nonadaptive arithmetic coder coupled with an explicit probability estimate of the significance map. In the tarp coder, the density estimation is efficiently computed by a novel series of 1D filtering operations known as tarp filtering. In contrast to other prominent wavelet-based coders, tarp coding lacks complex context modeling or cross-subband, cross-scale aggregation of symbols such as zerotree structures. Tarp, originally developed for 2D rectangular images, was extended to 3D imagery in [9, 10] and to shape-adaptive coding in [38]. 3D-tarp is presented in detail in chapter IV. Shape-adaptive tarp coding calls for “skipping” over land regions without changing the current probability estimate. Because the tarp algorithm lacks context modeling and symbol aggregation, this skipping of land leads to efficient performance for shape-adaptive coding.
2.7.4 3D-WDR

The wavelet difference reduction (WDR) [19] algorithm combines runlength coding of the significance map with an efficient lossless representation of runlength symbols to produce an embedded image coder. WDR can be easily made shape-adaptive by “skipping” over land regions and not coding any significance information for them or including them in the runlengths. We note that the algorithm described in [11, 35], which is currently used by NAVOCEANO for the coding of ocean-temperature imagery, is very similar to shape-adaptive 3D-WDR.

2.7.5 EBCOT

The recent JPEG2000 standard [2] is the most prominent example of techniques that code the significance map, using known significance states of neighboring coefficients to provide a context for the coding of the significance state of the current coefficient with an adaptive arithmetic coder. While the JPEG2000 standard does not support arbitrarily shaped image coding, the underlying embedded block coding with optimized truncation (EBCOT) algorithm [15] is easily made shape-adaptive. In shape-adaptive EBCOT [2], land regions are ignored and not coded in all coding passes, while anytime that the context for an ocean coefficient overlaps the bathymetry boundary, land coefficients in the context are treated as insignificant.

The significance-map coding in EBCOT is strictly a 2D process. That is, a 2D image is transformed with a 2D wavelet transform, and each subband is partitioned into a number of codeblocks, with an embedded bitstream generated independently for each codeblock. EBCOT can be used for 3D datasets by applying this 2D codeblock-based procedure to each 2D “slice” of the 3D dataset, truncating each codeblock bitstream, and then concatenating the truncated bitstreams together, to form the final bitstream. In the 3D-EBCOT coding of [36], a Lagrangian rate-distortion optimal truncation procedure
is used as in the original 2D EBCOT formulation [15]. We note that, due to the 2D nature of its codeblock processing, the 3D-EBCOT algorithm must use the wavelet-packet transform, whereas the preceding techniques can use either the wavelet-packet or dyadic decomposition structures.

We have examined the problem of coding geoscience data and some of the solutions that have been previously proposed. In the next chapter, we will discuss using JPEG2000 for hyperspectral image compression. Since the JPEG2000 standard only covers the decoder, certain encoder implementation decisions are left up to the algorithm designer. We will explore several different encoding strategies for JPEG2000.
CHAPTER III
JPEG2000 ENCODING TECHNIQUES

JPEG2000 is an embedded, wavelet-based coder that has been increasingly considered for the coding of hyperspectral imagery as well as other types of volumetric data, such as medical imagery. JPEG2000 is attractive because of its proven state-of-the-art performance for the compression of grayscale and color photographic imagery. However, its performance for hyperspectral compression can vary greatly, depending on how the JPEG2000 encoder handles multiple-component images, i.e., images with multiple spectral bands.

In effect, the JPEG2000 standard specifies the syntax and semantics of the compressed bitstream and, consequently, the operation of the decoder. The exact architecture of the encoder, on the other hand, is left largely to the designer of the compression system. For example, in [39], Varma and Bell explore tradeoffs for several parameters to a JPEG2000 encoder, such as the color space, quantization step size, and the number of transform levels. However, the coding of multiple-component imagery is not considered.

In deploying JPEG2000 on multiple-component images, such as hyperspectral imagery, there are two primary issues that must be considered in the implementation of the JPEG2000 encoder: 1) spectral decorrelation and 2) rate allocation between image components. The first issue arises due to the fact that there tends to exist significant correlation between consecutive bands in a hyperspectral image. In this chapter, we
consider spectral decorrelation via a wavelet transform; results in Chap. V indicate significant performance improvement results from its use.

The second encoder-design issue—rate allocation between image components—arises from the fact that, essentially, JPEG2000 is a 2D compression algorithm. Consequently, given a specific target bitrate of $R$ bits per pixel per band (bpppb), the JPEG2000 encoder must determine how to allocate this total rate appropriately between bands. It is usually the case that certain bands have significantly higher energy than other bands and thus will weigh more heavily in distortion measures than the other, weaker-energy bands. Consequently, it is likely that the JPEG2000 encoder will need to allocate proportionally greater rate to the higher-energy bands in order to maximize distortion performance for a given total rate $R$. In this chapter, we explore several rate-allocation strategies; results in Chap. V demonstrates significant performance difference between them.

In the following sections, we first briefly overview JPEG2000 compression for single-component imagery. We then discuss the application of a wavelet transform for spectral decorrelation within a JPEG2000 encoder and describe three strategies for rate allocation between multiple image components. The evaluation of all considered techniques is presented in Sec. 5.4. We note that the following discussion was initially presented in [8].

### 3.1 JPEG2000 for Single-Component Images

To code a single-component image, a JPEG2000 encoder first performs a 2D wavelet transform on the image and then partitions each transform subband into small, 2D rectangular blocks called codeblocks, which are typically of size $32 \times 32$ or $64 \times 64$ pixels. Subsequently, the JPEG2000 encoder independently generates an embedded bitstream for each codeblock. To assemble the individual codeblock bitstreams into
a single, final bitstream, each codeblock bitstream is truncated in some fashion, and
the truncated bitstreams are concatenated together to form the final bitstream. The
method for codeblock-bitstream truncation is an implementation issue concerning only
the encoder as codeblock-bitstream lengths are conveyed to the decoder as header
information. Consequently, this truncation process is not covered by the JPEG2000
standard.

It is highly likely that, for codeblocks residing in a single image component, any
given JPEG2000 encoder will perform a Lagrangian rate-distortion optimal truncation
as described as part of Taubman’s EBCOT algorithm [4, 15]. This optimal truncation
technique, post-compression rate-distortion (PCRD) optimization, is a primary factor in
the excellent rate-distortion performance of the EBCOT algorithm. PCDR optimization
is performed simultaneously across all of the codeblocks from the image, producing
an optimal truncation point for each codeblock. The truncated codeblocks are then
concatenated together to form a single bitstream. The PCDR optimization, in effect,
distributes the total rate for the image spatially across the codeblocks in a rate-distortion-
optimal fashion such that codeblocks with higher energy, which tend to more heavily
influence the distortion measure, tend to receive greater rate.

3.2 Spectral Decorrelation for Multiple-Component Images

The JPEG2000 standard allows for up to 16,385 image components to be included in
a single bitstream; however, the standard does not specify how these image components
should be encoded for best performance. Whereas Part I of the JPEG2000 standard
[2] permits spectral decorrelation only in the case of three-band images (i.e., red-green-
blue), Annexes I and N of Part II of the standard [3] make provisions for arbitrary
spectral decorrelation, including wavelet transforms.
By applying a 1D wavelet transform spectrally, and then subsequently employing a 2D wavelet transform spatially within each component, we effectively implement the 3D wavelet-packet transform in Fig. 2.5. We note that many JPEG2000 implementations are not yet fully compliant with Part II of the standard. In this case, we can “simulate” the spectral decorrelation permitted under Part II by employing a 1D wavelet transform spectrally on each pixel in the scene before the image cube is sent to the Part-I-compliant JPEG2000 encoder. Such an external spectral transform has been used previously [10, 40] to implement a “2D spatial + 1D spectral” wavelet-packet transform with Part-I-compliant coders.

3.3 Rate-Allocation Strategies Across Multiple Image Components

The PCRD optimization procedure of EBCOT [4, 15] produces a rate-distortion-optimal bitstream for a single-component image by optimally truncating the independent codeblock bitstreams from the component. However, there are several ways that this single-component truncation procedure can be extended to the multiple-component case, and the resulting multiple-component truncation procedure, in effect, dictates how the total rate available for coding the hyperspectral image is allocated between the individual spectral bands.

That is, for a multiple-component image, a JPEG2000 encoder will partition each component, or spectral band, into 2D codeblocks which are coded into independent bitstreams as described above in Sec. 3.1 for single-component imagery. To assemble a final bitstream, these individual codeblock bitstreams are truncated and concatenated together. Although the method for codeblock-bitstream truncation is an implementation issue concerning only the encoder and is thus not covered by the JPEG2000 standard, it is highly likely that, any given multiple-component JPEG2000 encoder with perform PCRD optimization for at least the codeblocks originating from a single
image component. How this truncation process is extended across the multiple components may vary with encoder implementation. Below, we describe three possible multiple-component rate-allocation strategies and evaluate each for the compression of hyperspectral data. In the following, let a hyperspectral image volume $X$ be composed of $N$ bands $X_i$, i.e., $X = \{X_1, X_2, \ldots, X_N\}$. We code $X$ with a total rate of $R$ bpppb. Assume that $B_i = \text{JPEG2000\_Encode}(R_i, X_i)$ is a single-component JPEG2000 encoder that encodes component $X_i$ with rate $R_i$, using PCRD optimization, producing a bitstream $B_i$.

3.3.1 JPEG2000-BIFR

The most straightforward method of allocating rate between multiple image components is to simply code each component independently and assign to each an identical rate. This JPEG2000 band-independent fixed-rate (JPEG2000-BIFR), strategy operates as follows:

\[
\text{JPEG2000\_BIFR}(R, \{X_1, \ldots, X_N\})
\]

\[
B = \emptyset
\]

for $i = 1, 2, \ldots, N$

\[
B_i = \text{JPEG2000\_Encode}(R, X_i)
\]

\[
B = B \circ B_i
\]

return $B$

where the “$\circ$” operator denotes bitstream concatenation.

3.3.2 JPEG2000-BIRA

The next method, JPEG2000 band-independent rate allocation (JPEG2000-BIRA), also codes each band independently; however, rates are allocated explicitly so that more
important bands are coded with higher bitrate, and less important bands are coded at a lower bitrate.

\[
\text{JPEG2000\_BIRA}(R, \{X_1, \ldots, X_N\})
\]

\[
B = \emptyset
\]

for \(i = 1, 2, \ldots, N\)

\[
\sigma_i^2 = \text{variance } [X_i]
\]

for \(i = 1, 2, \ldots, N\)

\[
R_i = \frac{\log_2 \sigma_i}{\sum_{j=1}^{N} \log_2 \sigma_j} \cdot R_N
\]

\[
B_i = \text{JPEG2000\_Encode}(R_i, X_i)
\]

\[
B = B \circ B_i
\]

return \(B\)

The rates, \(R_i\), are determined so that bands with larger variances (i.e., higher energy) are coded at a higher bitrate than those with lower variances, while the total rate for the entire volume is \(R\). This approach is, in essence, an ad-hoc variant of classical optimal rate allocation for a set of quantizers based on log variances (chap. 8 of [41], [42]).

3.3.3 JPEG2000-MC

The final approach, what we will call JPEG2000 multi-component (JPEG2000-MC), can be employed when the JPEG2000 encoder is capable of performing PCRD optimization across multiple bands. That is, all of the spectral bands are input to the encoder which produces codeblock bitstreams for every codeblock in every subband of every image component. Then, PCRD optimal truncation is applied to all codeblock bitstreams from all bands simultaneously, rather than simply the codeblock bitstreams for a single band as in Sec. 3.1. In this way, the PCRD optimization performs to the
maximum of its potential, implicitly allocating rate in a rate-distortion fashion, not only spatially within each image component, but also spectrally across the multiple bands.

Since the JPEG2000 standard only specifies the decoder, encoder design is left up to the algorithm developer. In this chapter, we investigated several encoding strategies for handling multiple component images with JPEG2000, that take into account issues such as rate-allocation and spectral decorrelation. In the next chapter, 3D-tarp [9, 10], a low complexity 3D wavelet based coder is proposed for the compression of hyperspectral imagery. 3D-tarp’s complexity is greatly reduced from that of JPEG2000, and it can be easily parallelized.
CHAPTER IV

3D-TARP

Many embedded wavelet-based coding schemes utilize sophisticated processes such as context conditioning (JPEG2000), rate-distortion optimization (JPEG2000), or significance lists (SPIHT) which hinder scaling the algorithms into a third dimension and present significant difficulty for on-board implementations in hardware, particularly when parallel processing is considered.

In this chapter, we extend the recently proposed tarp coder [28], a 2D embedded wavelet-based coder with an exceedingly simple implementation, to 3D for the coding of hyperspectral imagery. The tarp technique employs an explicit estimate of the probability of wavelet-coefficient significance and a simple nonadaptive arithmetic coder, resulting in a still-image coder that is easily scaled to higher-dimensional datasets. While the probability estimate takes the form of Parzen windows, a well known nonparametric probability-estimation technique, the tarp coder implements this Parzen-window probability estimate as a novel sequence of 1D filtering operations coined tarp filtering. Experimental results show that our 3D version of tarp (3D-tarp) can achieve almost the same rate-distortion performance as a 3D version of SPIHT (3D-SPIHT) [24], while JPEG2000 exhibits slightly better performance for most data sets.

As pertaining to hardware implementation, we show that the most time-consuming operation of our tarp coder, the tarp filtering, can be highly vectorized for implementation on single-instruction-multiple-data (SIMD) architectures. Thus, the
proposed tarp coder can exploit the data-parallel capabilities of modern general-purpose processors, or, for greater concurrency, customized hardware with longer vectors could be used. In any event, the tarp coder benefits from the simplicity, elegance, and implicit synchronization of SIMD implementation, whereas other algorithms, such as 3D-SPIHT and JPEG2000, typically require a more complicated multiprocessor implementation to achieve a smaller amount of parallelism.

The remainder of this chapter is organized as follows. Next, in Sec. 4.1, we briefly review the theory of probability estimation by Parzen windows. Subsequently, in Sec. 4.2, we describe tarp filtering, first overviewing the 2D tarp filter from [28] which is then extended to 3D. In Sec. 4.3, we describe the incorporation of tarp filtering into a wavelet-based embedded coder to produce a 3D-tarp coder for hyperspectral imagery. Finally, we consider the vectorization of the 3D tarp filter in Sec. 4.4. This work originally appeared in [9] and was expanded to include classification results and the discussion on the parallelized implementation of 3D-tarp in [10].

4.1 Estimation of Probability of Significance via Parzen Windows

Consider an \( N \)-dimensional field of real-valued coefficients, \( c[x] \in \mathbb{R} \), where \( x \in \mathbb{Z}^N \), \( \mathbb{R} \) is the set of real numbers, and \( \mathbb{Z} \) is the set of integers. Given a threshold \( t \in \mathbb{R} \), the coefficient at location \( x \) is defined to be significant with respect to \( t \) if \( |c[x]| \geq t \), and is insignificant otherwise. Define the significance state with respect to \( t \) of \( c[x] \) to be

\[
v[x] = \begin{cases} 
1, & |c[x]| \geq t, \\
0, & \text{otherwise.}
\end{cases}
\]  

Suppose we know that coefficients at locations \( x_1, x_2, \ldots, x_m \) are significant with respect to some given threshold, and we would like to estimate the probability that the
coefficient at location \( \mathbf{x} \) is also significant. Parzen windows [43] is one approach to performing this probability estimate. Specifically, we estimate the probability that \( c[\mathbf{x}] \) is significant as

\[
p[\mathbf{x}] = \sum_{i=1}^{m} \phi[\mathbf{x} - \mathbf{x}_i],
\]

(4.2)

where \( \phi[\mathbf{x}] \) is an \( N \)-dimensional window sequence. A possible window sequence which is suited to the well known Laplacian distribution nature of wavelet-coefficient magnitudes in images is the Laplacian window,

\[
\phi[\mathbf{x}] = \beta \alpha^{||\mathbf{x}||}, \quad \mathbf{x} \in \mathcal{R},
\]

(4.3)

where \( \alpha \) is a parameter controlling the spread of the window, \( ||\mathbf{x}|| = \sum_{i=1}^{N} |x_i| \) is the \( l_1 \) norm of \( \mathbf{x} = [x_1, x_2, \ldots, x_N] \), and \( \beta \) is chosen so that

\[
\sum_{\mathbf{x} \in \mathcal{R}} \phi[\mathbf{x}] = 1,
\]

(4.4)

where \( \mathcal{R} \subseteq \mathbb{Z}^N \) is the region of support of the window. As a result, it can be shown [43] that \( p[\mathbf{x}] \) is guaranteed to be a valid probability mass function; i.e., \( p[\mathbf{x}] \geq 0, \forall \mathbf{x} \in \mathbb{Z}^N \), and \( \sum_{\mathbf{x} \in \mathbb{Z}^N} p[\mathbf{x}] = 1 \).

The density estimation of (4.2) can be considered to be the convolution of an \( N \)-dimensional filter of impulse response \( \phi[\mathbf{x}] \) with a field of Kronecker impulses situated at \( x_1, x_2, \ldots, x_m \). If the region of support \( \mathcal{R} \) of window \( \phi[\mathbf{x}] \) is causal, then this convolution can be calculated via a single raster scan through the coefficients. Below, we will define the causal region of support so as to not include \( \mathbf{x} = 0 \). By not including \( \mathbf{x} = 0 \) in \( \mathcal{R} \), both an encoder and its corresponding decoder in a compression system can make the same estimate of \( p[\mathbf{x}] \) by single raster scan since (4.2) depends on only values encountered strictly before the current location in the raster scan.
4.2 Tarp Filtering

In [28], Simard et al. propose using the density estimate of (4.2) to code the significance of wavelet coefficients for still-image coding. Specifically, the significance state of a set of coefficients for a given threshold is coded via a raster scan through the coefficients. For coding efficiency, an entropy coder codes $v[x]$ for each coefficient, using the probability that $v[x] = 1$ for the current coefficient as determined by the density-estimation procedure. The coder of [28] implements the $N$-dimensional convolution of (4.2) as a sequence of 1D filtering operations coined tarp filtering. This 1D-filtering approach is more efficient than a direct implementation of (4.2) in that only a limited number of probability estimates need be buffered, and that, because probability estimates are propagated from coefficient to coefficient, fewer arithmetic operations are performed.

Once the probability of significance of the coefficients is estimated for a given threshold, the tarp coder of [28] proceeds in the usual bitplane-coding paradigm common to modern embedded coders—significance and refinement passes are applied successively, and the significance threshold decreases after each pass. In [28], the significance pass uses the tarp filter to drive a nonadaptive binary arithmetic coder to code $v[x]$ in each subband, while coefficient-sign and refinement information is coded using a nonadaptive binary arithmetic coder on a uniform distribution.

Below, we describe the tarp-filtering procedure in greater detail, first concentrating on the $N = 2$ case, which was the only dimensionality considered in the original development [28]. However, since the focus of this paper is the coding of 3D data, we extend the tarp algorithm to the $N = 3$ case in Sec. 4.2.2. We consider using the 3D tarp-filtering operation to code hyperspectral imagery subsequently in Sec. 4.3.

1The name tarp filtering comes from the shape of the Laplacian window of (4.3) which resembles a tarp draped over a pole.
4.2.1 2D tarp Filtering

For $N = 2$, $\mathbf{x} = [x_1, x_2]$, where $x_1$ and $x_2$ are the row and column indices, respectively. The Laplacian window (4.3) in this case is

$$\phi[\mathbf{x}] = \beta \alpha^{|x_1|+|x_2|}, \quad \mathbf{x} = [x_1, x_2] \in \mathcal{R},$$

(4.5)

where the causal region of support is $\mathcal{R} = \mathcal{R}_1 \cup \mathcal{R}_2$,

$$\mathcal{R}_1 = \{ \mathbf{x} = [x_1, x_2] : x_1 = 0, x_2 > 0 \},$$

$$\mathcal{R}_2 = \{ \mathbf{x} = [x_1, x_2] : x_1 > 0, x_2 \in \mathbb{Z} \}.$$  

(4.6)

In order for (4.4) to hold for this $\phi[\mathbf{x}]$ and $\mathcal{R}$, it can be derived that

$$\beta = \frac{(1 - \alpha)^2}{2\alpha}.$$  

(4.7)

In essence, the tarp coder of [28] uses three 1D filters to implement the density estimate of (4.2)—one filter processes each row from left to right, another filter processes each row from right to left, and a third filter processes each column from top to bottom. Pseudocode for this filtering operation is given in Fig. 4.1. In Fig. 4.1, $p_1$ forms the left-to-right row filter, the updating of $p_3$ corresponds to the right-to-left row filter, and the updating of $p_2$ implements the top-to-bottom filter carried out on each column. We note that the memory overhead of these filtering operations is one row of $p_2$ values. For more detail on how tarp filtering is combined with bitplane coding to produce an embedded image coder, see [28, 38] and the tarp-coder implementation in QccPack [20].
4.2.2 3D tarp Filtering

In this section, we extend to hyperspectral datasets the 2D tarp filter described above. For hyperspectral imagery with \( N = 3 \), \( x = [x_1, x_2, x_3] \), where \( x_1 \), \( x_2 \), and \( x_3 \) are the spatial-row, spatial-column, and spectral-slice indices, respectively. The Laplacian window (4.3) in this case is

\[
\phi[x] = \beta_3 |x_1| + |x_2| + |x_3|, \quad x = [x_1, x_2, x_3] \in \mathcal{R},
\]

(4.8)

where the causal region of support is \( \mathcal{R} = \mathcal{R}_1 \cup \mathcal{R}_2 \cup \mathcal{R}_3 \),

\[
\mathcal{R}_1 = \{ x = [x_1, x_2, x_3] : x_1, x_3 = 0, x_2 > 0 \},
\]

\[
\mathcal{R}_2 = \{ x = [x_1, x_2, x_3] : x_1 > 0, x_2 \in \mathbb{Z}, x_3 = 0 \},
\]

(4.9)

\[
\mathcal{R}_3 = \{ x = [x_1, x_2, x_3] : x_1, x_2 \in \mathbb{Z}, x_3 > 0 \}.
\]

In order for (4.4) to hold for this \( \phi[x] \) and \( \mathcal{R} \), it can be derived that

\[
\beta = \frac{(1 - \alpha)^3}{3\alpha^3 + \alpha^3}.
\]

(4.10)

Fig. 4.4 shows a typical Laplacian window in 3D.

To estimate the probability of significance in 3D, we propagate information from three neighboring values, one at the left, one above, and one in the same spatial position in the previous spectral slice. Raster scanning proceeds in the order column, row, and then slice, and we use 1D filters to propagate probability estimates. Specifically, five 1D filtering steps are used. Three filters (\( p_1 \), \( p_2 \), and \( p_4 \)) essentially operate at the current spectral slice in a fashion similar to the 2D tarp filter. That is, in the current slice, one filter processes each row from left to right, another filter processes each row from right to left, and a third filter processes each column from top to bottom. Next we propagate
information in the spectral direction. To do so, we use buffers that hold probabilities for the entire previous slice, and after each slice is coded, the probabilities in the slice buffer are updated. Consequently, after a full slice is coded with the first three filters, another two 1D filtering steps \((p_3 \text{ and } p_5)\) update the probabilities for the current slice. Pseudocode for the 3D tarp filter is shown in Fig. 4.2.

The 3D tarp filtering operation requires somewhat greater buffer storage than its 2D counterpart. Specifically, single rows are stored for \(p_2\) and \(p_5\), while entire spectral slices are stored for \(p_1\), \(p_3\), and \(p_4\). Below, we describe how 3D tarp filter is combined with bitplane coding to create an embedded coder for hyperspectral imagery.

### 4.3 The 3D-tarp Coder

Wavelet-based embedded coders are usually built on two processing passes, the significance pass and refinement pass. In the significance pass, the significance state \(v[x]\) of each coefficient is encoded, and, when a coefficient transitions from insignificant to significant, the sign of the coefficient is also encoded. In the refinement pass, all the coefficients known to be significant (except those that became significant in the immediately preceding significance pass) are refined by coding the value of the bit in the current bitplane.

Contrary to most wavelet-based embedded coders, which use multiple-context adaptive arithmetic coding which is responsible for a significant portion of their rate-distortion performance, the tarp coder uses a relatively simple nonadaptive binary arithmetic coder. The tarp-filtering operation produces the estimate \(p[x]\) of the probability of significance of the current coefficient, and this probability estimate drives the arithmetic coder when coding the significance state \(v[x]\) in the significance pass. For the coding of sign bits in the significance pass, and for the coding of refinement bits in the refinement pass, we use a constant probability of 0.5 in the nonadaptive
arithmetic coder. Although it is possible to use more sophisticated codings of these sign and refinement bits [17], in practice, a nonuniform probability distribution would result in minimal rate-distortion improvement, while the use of the uniform distribution greatly simplifies the implementation and reduces computational complexity.

4.4 Parallelized Implementation of 3D-tarp

Although the DWT and arithmetic coder consume non-negligible computational resources, the tarp-filtering operation is responsible for an overwhelmingly large portion of the execution time of the software tarp coder used in the experimental results of the previous section. However, the tarp filter permits a significant amount of vectorization resulting in potentially substantial acceleration of the tarp coder when implemented in SIMD hardware. In the tarp-filtering operation, a large number of the filters are confined within one spectral slice, thereby allowing vectorization in the spectral direction; i.e., the filtering of multiple spectral slices in parallel. Specifically, the $p_1$, $p_2$, $p_4$, and $p_5$ filters support vectorization in the spectral direction, although the ordering of the computations must be rearranged somewhat from that originally presented in Fig. 4.2. Additionally, the spectral-direction filter, $p_3$, can be vectorized in the column direction. Finally, the calculation of the final probability $p$ can be vectorized in either the row, column, or spectral direction. Fig. 4.4 gives the resulting parallelized version of the 3D tarp filter. We note that the cost of the reordering of the algorithm from that of Fig. 4.2 is increased memory usage since one must maintain entire buffer volumes for $p_1, \ldots, p_5$ rather than the single spectral slices needed for $p_1, p_3,$ and $p_4$ originally. However, recall that the tarp coder employs tarp filtering on a subband-by-subband basis; consequently, buffer volumes need be only as big as the largest subband to be processed, specifically, $N_1N_2N_3/8$. Additionally, we note that, since the decoder needs $p$ for the current coefficient in order to decode $v$, the reordering of the tarp filter shown in Fig. 4.4
is suitable for only the encoder of a tarp-coder system. However, in hyperspectral applications, it is the encoder that is most likely to have access to parallelized hardware in time-critical on-board applications. We note that decoding following the original, non-parallelized filtering of Fig. 4.2 is possible even when the encoder uses parallelized filtering as in Fig. 4.4.

The degree of acceleration achieved by the vectorized tarp coder will depend on the amount of data-parallelism supported by the underlying SIMD architecture. To increase parallelization and reduce computational complexity, the tarp-filtering operations can be easily performed with fixed-point, rather than floating-point, arithmetic. Modern general-purpose processors typically support some integer-based SIMD processing. For example, assuming that 16-bit fixed-point representations are used, Motorola’s AltiVec [44] SIMD implementation would support eight parallel operations, while Intel’s MMX [45] would support four. Custom hardware implementation could conceivably employ longer vectors such that the acceleration obtainable would be limited by primarily the subband size.

Finally, we note that both SPIHT and JPEG2000 support parallelization to a certain extent; for example, see [46, 47] and Chap. 17 of [4]. However, these algorithms are highly sequential by nature and are difficult to make parallel. Additionally, the amount of parallelization is limited and typically relies on pipelining and multiprocessor, i.e., multiple-instruction-multiple-data (MIMD), architectures. Consequently, such implementations lack the simple and implicitly synchronized architecture of SIMD-based tarp filtering.

In this chapter, the 3D-tarp algorithm is proposed for the compression of hyperspectral imagery. 3D-tarp is a low-complexity, parallelizable algorithm, and, as can be seen in the next chapter, exhibits roughly equivalent rate-distortion performance to other, more complicated techniques. The next chapter describes the data, formulates
performance metrics, and presents results for several popular 3D wavelet-based compression algorithms.
for $x_1 = 0, \ldots, N_1 - 1$
    $p_1 = 0$
    for $x_2 = 0, \ldots, N_2 - 1$
        $p[x_1, x_2] = \alpha p_1 + \alpha p_2[x_2]$
        $p_1 = \alpha p_1 + \beta v[x_1, x_2]$
        $p_2[x_2] = p_1 + \alpha p_2[x_2]$
    endfor
    $p_3 = 0$
    for $x_2 = N_2 - 1, \ldots, 0$
        $p_2[x_2] = p_2[x_2] + \alpha p_3$
        $p_3 = \alpha p_3 + \beta v[x_1, x_2]$
    endfor
endfor

Figure 4.1: Pseudocode for the 2D tarp filter of [28]. The image is of size $N_1 \times N_2$. 
for $x_1 = 0, \ldots, N_1 - 1$
for $x_2 = 0, \ldots, N_2 - 1$
\[ p_3[x_1, x_2] = 0 \]
endfor
endfor

for $x_3 = 0, \ldots, N_3 - 1$
for $x_2 = 0, \ldots, N_2 - 1$
\[ p_2[x_2] = 0 \]
endfor
for $x_1 = 0, \ldots, N_1 - 1$
for $x_2 = 0, \ldots, N_2 - 1$
\[ p[x_1, x_2, x_3] = \alpha p_1[x_1, x_2 - 1] + \alpha p_2[x_2] + \alpha p_3[x_1, x_2] \]
\[ p_1[x_1, x_2] = \alpha p_1[x_1, x_2 - 1] + \beta v[x_1, x_2, x_3] \]
\[ p_2[x_2] = p_1[x_1, x_2] + \alpha p_2[x_2] \]
endfor
for $x_2 = N_2 - 1, \ldots, 0$
\[ p_2[x_2] = p_2[x_2] + \alpha p_4[x_1, x_2 + 1] \]
\[ p_3[x_1, x_2] = p_2[x_2] + \alpha p_3[x_1, x_2] \]
\[ p_4[x_1, x_2] = \alpha p_4[x_1, x_2 + 1] + \beta v[x_1, x_2, x_3] \]
endfor
for $x_2 = 0, \ldots, N_2 - 1$
\[ p_5[x_2] = 0 \]
endfor
for $x_1 = N_1 - 1, \ldots, 0$
for $x_2 = 0, \ldots, N_2 - 1$
\[ p_3[x_1, x_2] = p_3[x_1, x_2] + \alpha p_5[x_2] \]
\[ p_5[x_2] = p_1[x_1, x_2] + \alpha p_5[x_2] + \alpha p_4[x_1, x_2 + 1] \]
endfor
endfor
endfor

Figure 4.2: Pseudocode for the 3D tarp filter. The volume is of size $N_1 \times N_2 \times N_3$. 
Figure 4.3: The 3D Laplacian window for $\alpha = 0.5$. The boxed value indicates the window origin ($x_1, x_2, x_3 = 0$).
for $x_1 = 0, \ldots, N_1 - 1$
  for $x_2 = 0, \ldots, N_2 - 1$
    $p_1[x_1, x_2, :] = \alpha p_1[x_1, x_2 - 1, :] + \beta v[x_1, x_2, :]$
    $p_2[x_1, x_2, :] = p_1[x_1, x_2, :] + \alpha p_2[x_1 - 1, x_2, :]$
  endfor
  for $x_2 = N_2 - 1, \ldots, 0$
    $p_2[x_1, x_2, :] = p_2[x_1, x_2, :] + \alpha p_4[x_1, x_2 + 1, :]$
    $p_4[x_1, x_2, :] = \alpha p_4[x_1, x_2 + 1, :] + \beta v[x_1, x_2, :]$
  endfor
endfor
for $x_1 = N_1 - 1, \ldots, 0$
  for $x_2 = 0, \ldots, N_2 - 1$
    $p_6[x_1, x_2, :] = p_1[x_1, x_2, :] + \alpha p_6[x_1 + 1, x_2, :] + \alpha p_4[x_1, x_2 + 1, :]$
  endfor
endfor
for $x_3 = 0, \ldots, N_3 - 1$
  for $x_1 = 0, \ldots, N_1 - 1$
    $p_3[x_1, :, x_3] = p_2[x_1, :, x_3] + \alpha p_3[x_1, :, x_3 - 1] + \alpha p_6[x_1 + 1, :, x_3]$
  endfor
endfor
for $x_3 = 0, \ldots, N_3 - 1$
  for $x_1 = 0, \ldots, N_1 - 1$
    $p[x_1, :, x_3] = \alpha p_1[x_1, :, x_3] + \alpha p_2[x_1 - 1, :, x_3] + \alpha p_3[x_1, :, x_3 - 1]$
  endfor
endfor

Figure 4.4: Pseudocode for the vectorized 3D tarp filter for SIMD architectures. All buffer volumes initialized to zero at algorithm start. The “:” indicates vectorization along the corresponding dimension. $p_1'$ is $p_1$ offset by a one-column shift to the right; i.e., $p_1'[x_1, x_2, x_3] = p_1[x_1, x_2 - 1, x_3]$. This shift is accomplished during loading of the vector.
CHAPTER V
HYPERSPECTRAL COMPRESSION RESULTS

In this chapter, we evaluate the performance of several 3D wavelet-based compression algorithms for hyperspectral imagery. The hyperspectral datasets used in this work are described in Sec. 5.1. The performance metrics that will be used throughout the remainder of this thesis are presented in Sec. 5.2. The selection of the wavelet transform method, i.e. dyadic or wavelet-packet, is investigated in Sec. 5.3. In Sec. 5.4, the JPEG2000 encoding strategies discussed in Chap. III are evaluated. And Sec. 5.5 contains a comprehensive body of results and comparisons for several prominent wavelet-based compression techniques.

5.1 Data

All the datasets used in the experiments were collected by the Airborne Visible Infrared Imaging Spectrometer (AVIRIS), an airborne, hyperspectral sensor that collects data in 224 contiguous bands from 400 nm to 2500 nm. For the results here, we crop the first scene in each dataset to produce image cubes with the dimensions $512 \times 512 \times 224$. In all cases, the unprocessed radiance data was used. The datasets can be found at NASA’s AVIRIS website: http://aviris.jpl.nasa.gov/html/aviris.freedata.html. The false-color images of the datasets are shown below in Figs. 5.1-5.5. The false-color images use band 52 for red, 30 for green, and 5 for blue.
Figure 5.1: False-color image of Moffett
Figure 5.2: False-color image of Jasper Ridge
Figure 5.3: False-color image of Cuprite
Figure 5.4: False-color image of Low Altitude
Figure 5.5: False-color image of Lunar Lake
5.2 Performance Metrics

Traditionally, performance for lossy compression is determined by simultaneously measuring both distortion and rate. Distortion measures the fidelity of the reconstructed data to the original data, while rate essentially measures the amount of compression incurred. Distortion is commonly measured via a signal-to-noise ratio (SNR) between the original and reconstructed data. Let \( c[x_1, x_2, x_3] \) be an \( N_1 \times N_2 \times N_3 \) hyperspectral dataset with variance of \( \sigma^2 \). Let \( \tilde{c}[x_1, x_2, x_3] \) be the reconstructed dataset from the compressed bitstream. The mean square error (MSE) is defined as

\[
\text{MSE} = \frac{1}{N_1N_2N_3} \sum_{x_1,x_2,x_3} (c[x_1, x_2, x_3] - \tilde{c}[x_1, x_2, x_3])^2,
\]

while the SNR in decibels (dB) is defined in terms of the MSE as

\[
\text{SNR} = 10 \log_{10} \frac{\sigma^2}{\text{MSE}}.
\]

Both the MSE and SNR provide a measure of the performance of a coder in an average sense over the entire volume. Such an average measure may or may not be of the greatest use, depending on the application to be made of the reconstructed data. Hyperspectral imagery is often used in applications involving extensive analysis; consequently, it is paramount that the compression of hyperspectral data does not alter the outcomes of such analysis. As an alternative to the SNR measure for distortion, one can examine the difference between performance of application-specific analysis as applied to the original data and the reconstructed data. As an example, unsupervised classification of hyperspectral pixel vectors is representative of methods that segment an image into multiple constituent classes. To form a distortion measure, we can apply unsupervised classification on the original hyperspectral image as well as on the
reconstructed image, counting the number of pixels that change assigned class as a result of the compression. This distortion measure, *preservation of classification* (POC), is measured as the percentage of pixels that do not change class due to compression.

In the subsequent experimental results reported in Secs. 5.5 and 5.4, all POC results are calculated using the ISODATA and $k$-means unsupervised classification techniques as implemented in ENVI Version 4.0. A maximum of ten classes are used, and the POC performance is determined by applying the classification to the original dataset as well as to the reconstructed volume and comparing the classification map produced for reconstructed volume to that of the original dataset, i.e., the classification map of the original dataset becomes the “ground truth.” Fig. 5.6 depicts typical classification maps generated in this manner.
Figure 5.6: Classification map for the Moffett image using \(k\)-means classification. (a) Map for original image, (b) map after JPEG2000-BIFR compression.
In addition to distortion, it is necessary to gauge compression techniques according to the amount of compression incurred, due to the inherent tradeoff between distortion and compression—the more highly compressed a reconstructed dataset is, the greater is the expected distortion between the original and reconstructed data. Typically, for hyperspectral imagery, one measures the rate as the number of bits per pixel per band (bpppb), which gives the average number of bits to represent a single sample of the hyperspectral dataset. The compression ratio can then be determined as the ratio of the bpppb of the original dataset (usually 16 bpppb) to the bpppb of the compressed dataset.

5.3 Dyadic vs. Wavelet-Packet Transform

As was discussed in Sec. 2.1, there are two contending transform arrangements for the 3D DWT. The 3D dyadic transform (Fig. 2.4) is a direct extension of the dyadic transform of the 2D case, in which we transform once in each direction and then further decompose the baseband. In the case of the 3D packet transform (Fig. 2.5), the coefficients in each spectral slice are transformed with a 2D dyadic transform, which is then followed by a spectral transform. Fig. 5.7 depicts the typical rate-distortion performance achieved by a coder using these two transform structures. We see that the performance with the packet transform is greatly superior to that of the dyadic transform. Tables 5.1 show that the wavelet-packet transform yields higher POC results, with the exception of the Low Altitude dataset. Table 5.2 reports SNR results for each of the datasets at 1.0 bpppb. Again, the wavelet-packet transform performs significantly better (4-9 dB) than the dyadic transform for all datasets except the Low Altitude dataset. As we have observed similar results for other coders and other datasets, we will use the packet transform exclusively for all subsequent results, unless otherwise noted.
Figure 5.7: Comparison of the typical rate-distortion performance for the dyadic transform of Fig. 2.4 versus that of the packet transform of Fig. 2.5. This plot is for the Moffett image using 3D-SPIHT.
Table 5.1: POC Performance of 3D-SPIHT with Dyadic and Packet Transforms

<table>
<thead>
<tr>
<th>Dataset</th>
<th>ISODATA POC (%)</th>
<th>3D-SPIHT - Packet</th>
<th>3D-SPIHT - Dyadic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Moffett</td>
<td>99.7</td>
<td>98.6</td>
<td></td>
</tr>
<tr>
<td>Jasper Ridge</td>
<td>99.7</td>
<td>98.9</td>
<td></td>
</tr>
<tr>
<td>Cuprite</td>
<td>99.8</td>
<td>98.7</td>
<td></td>
</tr>
<tr>
<td>Low Altitude</td>
<td>97.9</td>
<td>99.2</td>
<td></td>
</tr>
<tr>
<td>Lunar Lake</td>
<td>99.7</td>
<td>99.5</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dataset</th>
<th>k-means POC (%)</th>
<th>3D-SPIHT - Packet</th>
<th>3D-SPIHT - Dyadic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Moffett</td>
<td>99.6</td>
<td>98.0</td>
<td></td>
</tr>
<tr>
<td>Jasper Ridge</td>
<td>99.5</td>
<td>98.3</td>
<td></td>
</tr>
<tr>
<td>Cuprite</td>
<td>99.6</td>
<td>98.0</td>
<td></td>
</tr>
<tr>
<td>Low Altitude</td>
<td>96.6</td>
<td>98.3</td>
<td></td>
</tr>
<tr>
<td>Lunar Lake</td>
<td>99.5</td>
<td>99.5</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.2: SNR at 1.0 bpppb for SPIHT

<table>
<thead>
<tr>
<th>Dataset</th>
<th>SNR (dB)</th>
<th>3D-SPIHT - Packet</th>
<th>3D-SPIHT - Dyadic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Moffett</td>
<td>45.3</td>
<td>37.8</td>
<td></td>
</tr>
<tr>
<td>Jasper Ridge</td>
<td>44.7</td>
<td>37.8</td>
<td></td>
</tr>
<tr>
<td>Cuprite</td>
<td>50.7</td>
<td>44.0</td>
<td></td>
</tr>
<tr>
<td>Low Altitude</td>
<td>27.4</td>
<td>31.8</td>
<td></td>
</tr>
<tr>
<td>Lunar Lake</td>
<td>46.1</td>
<td>42.2</td>
<td></td>
</tr>
</tbody>
</table>


5.4 Evaluation of JPEG2000 Encoding Strategies

For the results of this section, all JPEG2000 coding was done with Kakadu\(^1\) Version 4.3, with 5 levels of wavelet decomposition both spatially and spectrally and a quantization step size of 0.0000001. The popular 9-7 biorthogonal filter (included in Part I of the standard) was used for both spatial and spectral transforms. Since Kakadu is not yet fully compliant with Part II of the JPEG2000 standard, the spectral transform was applied externally as described in Sec. 3.2 and [10, 40].

We first examine rate-distortion performance of JPEG2000 encoding. In Fig. 5.8, we plot rate-distortion performance for a range of rates, while in Table 5.3, distortion performance at a single rate is tabulated. In these results, techniques labeled as “2D” do not use any spectral transform (i.e., only 2D wavelet transforms are applied spatially), while the other techniques use the 3D wavelet-packet transform which includes a spectral transform. For each dataset, we present performance for the three rate-allocation techniques described in Sec. 3.3, both with and without the spectral decorrelation transform. With the exception of JPEG2000-BIFR, all the rate-allocation techniques perform significantly better when a spectral transform is performed. We see that JPEG2000-MC substantially outperforms the other techniques by at least 5–10 dB. For the remainder of this thesis, JPEG2000-MC is used exclusively for JPEG2000 results.

We next turn our attention to classification performance. POC results are presented for ISODATA and \(k\)-means in Table 5.4. We see that the POC performances correlate well with SNR figures from Table 5.3—if one technique outperforms another in the rate-distortion realm, then it will mostly likely have higher POC as well. As expected, JPEG2000-MC performs substantially better than the other techniques in terms of POC.

\(^1\)http://www.kakadusoftware.com
Figure 5.8: Rate-distortion performance of JPEG2000 encoding techniques for Moffett.

Table 5.3: SNR Performances at 1.0 bpppb.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>2D</th>
<th>2D</th>
<th>2D</th>
<th>2D</th>
<th>2D</th>
<th>2D</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BIFR</td>
<td>BIFR</td>
<td>BIRA</td>
<td>BIRA</td>
<td>MC</td>
<td>MC</td>
</tr>
<tr>
<td>moffett</td>
<td>25.8</td>
<td>25.9</td>
<td>27.4</td>
<td>34.9</td>
<td>30.6</td>
<td>45.5</td>
</tr>
<tr>
<td>jasper ridge</td>
<td>24.0</td>
<td>23.8</td>
<td>25.7</td>
<td>33.4</td>
<td>29.8</td>
<td>44.8</td>
</tr>
<tr>
<td>cuprite</td>
<td>32.9</td>
<td>32.8</td>
<td>34.9</td>
<td>42.6</td>
<td>38.3</td>
<td>51.0</td>
</tr>
</tbody>
</table>
Figure 5.9: Rate-distortion performance of JPEG2000 encoding techniques for Jasper Ridge.
Figure 5.10: Rate-distortion performance of JPEG2000 encoding techniques for Cuprite.
Table 5.4: POC Performances for the Various JPEG2000 Encoding Strategies

<table>
<thead>
<tr>
<th>Dataset</th>
<th>ISODATA POC (%)</th>
<th>2D BIFR</th>
<th>2D BIRA</th>
<th>2D MC</th>
<th>2D BIFR</th>
<th>2D BIRA</th>
<th>2D MC</th>
<th>k-means POC (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>moffett</td>
<td>83.4</td>
<td>94.5</td>
<td>86.6</td>
<td>94.5</td>
<td>93.2</td>
<td>99.7</td>
<td></td>
<td>99.6</td>
</tr>
<tr>
<td>jasper ridge</td>
<td>77.3</td>
<td>75.5</td>
<td>82.2</td>
<td>93.7</td>
<td>93.9</td>
<td>99.7</td>
<td></td>
<td>99.5</td>
</tr>
<tr>
<td>cuprite</td>
<td>80.3</td>
<td>78.1</td>
<td>85.1</td>
<td>94.7</td>
<td>94.7</td>
<td>99.8</td>
<td></td>
<td>99.6</td>
</tr>
<tr>
<td>moffett</td>
<td>75.4</td>
<td>73.2</td>
<td>79.9</td>
<td>91.7</td>
<td>89.8</td>
<td>99.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jasper ridge</td>
<td>67.2</td>
<td>64.7</td>
<td>73.9</td>
<td>90.4</td>
<td>91.0</td>
<td>99.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cuprite</td>
<td>71.3</td>
<td>68.3</td>
<td>77.6</td>
<td>92.2</td>
<td>92.1</td>
<td>99.6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
5.5 Hyperspectral Compression Results

Rate-distortion performance for the 3D-WDR, 3D-tarp, 3D-SPECK, 3D-SPIHT, and JPEG2000 coders is shown in Figs. 5.11–5.13. In these results, we see that all five techniques provide largely similar rate-distortion performance for the datasets considered, with JPEG2000 usually slightly outperforming the others. Especially at low bit rates (less than 1 bpppb), all techniques give nearly identical rate-distortion performance. Since performances are relatively close, other implementation-based considerations should be made, such as complexity and resource usage.

SNR results at 1.0 bpppb for all techniques are presented in Table 5.6. JPEG2000 edges out 3D-SPIHT by a narrow margin on all datasets, followed closely by 3D-SPECK, 3D-WDR, and 3D-tarp, respectively. The POC performances at 1.0 bpppb in Table 5.5 further support that all of the algorithms exhibit roughly equal performance, with JPEG2000 slightly edging out the other techniques for most of the datasets. As was the case in Sec. 5.4, POC correlates well to SNR performance.

In this chapter, we evaluated the performance of several 3D wavelet-based algorithms for the compression of hyperspectral imagery, and SNR and POC performance metrics were established. In the next chapter, we propose 3D Binary Set Splitting with \(k\)-d Trees (3DBISK) [12] for the shape-adaptive coding of ocean temperature data.
Figure 5.11: Rate-distortion performance of all techniques for Moffett.
Figure 5.12: Rate-distortion performance of all techniques for Cuprite.
Figure 5.13: Rate-distortion performance of all techniques for Jasper Ridge.
Table 5.5: POC Performance at 1.0 bpppb for the Various Coders

<table>
<thead>
<tr>
<th>Dataset</th>
<th>ISODATA POC (%)</th>
<th>JPEG2000-MC</th>
<th>SPECK</th>
<th>SPIHT</th>
<th>TARP</th>
<th>WDR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Moffett</td>
<td>99.8</td>
<td>99.7</td>
<td>99.7</td>
<td>99.7</td>
<td>99.7</td>
<td>99.7</td>
</tr>
<tr>
<td>Jasper Ridge</td>
<td>99.8</td>
<td>99.7</td>
<td>99.7</td>
<td>99.7</td>
<td>99.7</td>
<td>99.7</td>
</tr>
<tr>
<td>Cuprite</td>
<td>99.8</td>
<td>99.8</td>
<td>99.8</td>
<td></td>
<td>99.8</td>
<td>99.8</td>
</tr>
<tr>
<td>Low Altitude</td>
<td>97.9</td>
<td>98.1</td>
<td>97.9</td>
<td>97.3</td>
<td>97.9</td>
<td></td>
</tr>
<tr>
<td>Lunar Lake</td>
<td>99.7</td>
<td>99.5</td>
<td>99.7</td>
<td>99.7</td>
<td>99.7</td>
<td>99.7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dataset</th>
<th>k-Means POC (%)</th>
<th>JPEG2000-MC</th>
<th>SPECK</th>
<th>SPIHT</th>
<th>TARP</th>
<th>WDR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Moffett</td>
<td>99.7</td>
<td>99.6</td>
<td>99.6</td>
<td>99.6</td>
<td>99.6</td>
<td>99.6</td>
</tr>
<tr>
<td>Jasper Ridge</td>
<td>99.6</td>
<td>99.5</td>
<td>99.5</td>
<td>99.5</td>
<td>99.5</td>
<td></td>
</tr>
<tr>
<td>Cuprite</td>
<td>99.7</td>
<td>98.7</td>
<td>99.6</td>
<td></td>
<td>99.7</td>
<td>99.7</td>
</tr>
<tr>
<td>Low Altitude</td>
<td>96.6</td>
<td>96.8</td>
<td>96.6</td>
<td>96.1</td>
<td>96.7</td>
<td></td>
</tr>
<tr>
<td>Lunar Lake</td>
<td>99.5</td>
<td>99.6</td>
<td>99.5</td>
<td>99.2</td>
<td>99.6</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.6: SNR at 1.0 bpppb for the Packet Transform

<table>
<thead>
<tr>
<th>Dataset</th>
<th>SNR (dB)</th>
<th>JPEG2000-MC</th>
<th>SPECK</th>
<th>SPIHT</th>
<th>TARP</th>
<th>WDR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Moffett</td>
<td>45.4</td>
<td>45.1</td>
<td>45.3</td>
<td>44.5</td>
<td>44.7</td>
<td></td>
</tr>
<tr>
<td>Jasper Ridge</td>
<td>44.9</td>
<td>44.4</td>
<td>44.7</td>
<td>43.7</td>
<td>44.2</td>
<td></td>
</tr>
<tr>
<td>Cuprite</td>
<td>50.8</td>
<td>50.5</td>
<td>50.7</td>
<td>50.3</td>
<td>50.4</td>
<td></td>
</tr>
<tr>
<td>Low Altitude</td>
<td>27.6</td>
<td>27.3</td>
<td>27.4</td>
<td>25.2</td>
<td>27.1</td>
<td></td>
</tr>
<tr>
<td>Lunar lake</td>
<td>46.4</td>
<td>45.9</td>
<td>46.1</td>
<td>43.7</td>
<td>45.9</td>
<td></td>
</tr>
</tbody>
</table>
In this chapter, we describe 3D binary set splitting with \( k \)-d trees (3D-BISK), which is a 3D extension of the 2D-BISK coder proposed in [48]. BISK is itself a variant of the well-known, state-of-the-art set-partitioning embedded block (SPECK) algorithm [29, 30]; its shape-adaptive version, object-based SPECK (OB-SPECK) [37]; and its 3D extension, 3D-OB-SPECK [31]. Experimental evidence has shown that 3D-SPECK demonstrates performance roughly equivalent to that of the prominent JPEG2000 standard [2] in tasks such as the compression of hyperspectral image cubes. JPEG2000, on the other hand, does not support shape-adaptive coding.

The main contribution of this chapter is the development of the 3D-BISK algorithm which replaces the octree set-partitioning operation of 3D-SPECK with \( k \)-d trees [13], a simpler set decomposition particularly well-suited to shape-adaptive coding due to its greater flexibility at capturing arbitrarily shaped regions. Additionally, 3D-BISK aggressively discards land regions from consideration by shrinking the decomposed sets to the bounding box of their ocean regions. Empirical results demonstrate that 3D-BISK consistently yields rate-distortion performance significantly superior to that of a number of other 3D shape-adaptive embedded coders for the coding of variety of ocean-temperature volumes.

The remainder of this chapter is organized as follows. In Sec. 6.1, we describe the 3D-BISK algorithm in detail and experimentally compare it to other techniques in
Sec. 6.2. We note that a preliminary description of 3D-BISK appeared in [12]; here, we give a more thorough presentation of the algorithm, as well as a more comprehensive experimental investigation.

6.1 3D binary set-splitting with \( k \)-d trees

In this section, we describe our 3D-BISK algorithm and the \( k \)-d tree set-partitioning structure upon which it is based. 3D-BISK is a variant of 3D-OB-SPECK that is well-suited to shape-adaptive coding. The octree-partitioning of 3D-OB-SPECK is replaced by binary set splitting of \( k \)-d trees which allows for a more flexible coding of arbitrarily shaped regions. An additional key difference between 3D-OB-SPECK and 3D-BISK is the aggressive shrinking of the sets to the bounding box of the ocean coefficients contained in the set, which is responsible for a large part of the performance gain.

6.1.1 Set Partitioning with \( k \)-d Trees

Octrees and \( k \)-d trees [13] are two well-known methods for the partitioning of 3D sets. In an octree, a set (a rectangular prism) is divided along all three dimensions to form eight equally sized subsets. On the other hand, in a \( k \)-d tree, a set is divided along a single dimension into two arbitrarily sized subsets. Figs. 6.1–6.2 depict set partitioning of BISK in 2D and 3D. Utilizing \( k \)-d trees for set partitioning allows us to extend BISK to higher dimensions without significantly increasing the implementation complexity. It is straightforward to see that \( k \)-d trees can achieve a partitioning of a set identical to that resulting from an octree decomposition, although usually a greater number of levels of decomposition are needed. However, we demonstrate, below, that the \( k \)-d trees decomposition is advantageous for shape-adaptive coding of the significance map.
Figure 6.1: Set Partitioning in 2D-BISK

Figure 6.2: Set Partitioning in 3D-BISK
6.1.2 The 3D-BISK Algorithm

Following a 3D wavelet transform, the 3D-BISK algorithm begins by placing the subbands of the transformed coefficients into a list of insignificant sets (LIS); subsequently, each subband is “shrunk” to the bounding volume of its ocean coefficients. As in 3D-OB-SPECK, each LIS is indexed, and a given set \( S \) resides in the LIS with index \( N(S) \), i.e., in LIS\( _{N(S)} \). The LIS index, \( N(S) \), is the total number of decompositions, or splits, that has produced the set. The algorithm continues in the usual bitplane-coding fashion with sorting and refinement passes. The algorithm is as follows:

\[
\text{procedure } \text{BISK}(\mathcal{X}) \\
\text{Initialization}(\mathcal{X}) \\
\text{Initialization}(\mathcal{X}) \\
n \leftarrow \text{max bitplane} \\
\text{while (true)} \\
\quad \text{SortingPass()} \\
\quad \text{RefinementPass()} \\
n \leftarrow n - 1 \\
\text{procedure } \text{Initialization}(\mathcal{X}) \\
\quad \text{for each subband } \mathcal{S} \text{ in } \mathcal{X} \\
\quad \quad N(\mathcal{S}) \leftarrow \text{total number of decompositions} \\
\quad \quad \text{in all dimensions} \\
\quad \quad \text{ShrinkSet}(\mathcal{S}) \\
\quad \quad \text{append } \mathcal{S} \text{ to LIS}_{N(\mathcal{S})} \\
\quad \quad \text{LSP } \leftarrow \emptyset \]
procedure SortingPass()

    \( l = \text{number of LIS lists} \)

    while \( l > 0 \)

        for each \( S \in \text{LIS}_{N(S)} \)

            \( \text{ProcessSet}(S) \)

        \( l \leftarrow l - 1 \)

procedure RefinementPass()

    for each \( S \in \text{LSP} \)

        output \( n^{th} \) bitplane value of coefficient magnitude

Like 3D-OB-SPECK, 3D-BISK tests the significance of all the sets in the all LIS lists. A set is considered to be significant if the magnitude of the largest ocean coefficient exceeds a threshold. If a set contains no significant ocean coefficients, it is placed into an LIS and will be processed at the next lower threshold. Since 3D-BISK employs \( k \)-d trees, when a set becomes significant, it is split into halves. Each half is then placed in an LIS and processed in the same manner until decomposed to a single pixel. If a set being processed contains no ocean coefficients, the set is removed from its LIS and discarded. The algorithms for processing and partitioning the sets is described below:

procedure ProcessSet(\( S \))

    if \( S = \emptyset \)

        remove \( S \) from \( \text{LIS}_{N(S)} \)

    else

        output \( \Gamma_n(S) \)

        if \( \Gamma_n(S) = 1 \)

            remove \( S \) from \( \text{LIS}_{N(S)} \)
if $|S| = 1$

output sign of $S$

append $S$ to LSP

else

CodeSet($S$)

procedure CodeSet($S$)

$\{S_1, S_2\} = \text{PartitionSet}(S)$

if $S_1 \neq \emptyset$

append $S_1$ to $\text{LIS}_{N(S_1)}$

ProcessSet($S_1$)

append $S_2$ to $\text{LIS}_{N(S_2)}$

ProcessSet($S_2$)

procedure PartitionSet($S$)

if $z(S) \geq y(S) \geq x(S)$

split $S$ depth-wise into $S_1$ and $S_2$:

$S_1: \lfloor z(S)/2 \rfloor \times y(S) \times x(S)$

$S_2: (z(S) - \lfloor z(S)/2 \rfloor) \times y(S) \times x(S)$

else

if $x(S) \geq y(S) > z(S)$

split $S$ horizontally into $S_1$ and $S_2$:

$S_1: z(S) \times \lfloor y(S)/2 \rfloor \times x(S)$

$S_2: z(S) \times (y(S) - \lfloor y(S)/2 \rfloor) \times x(S)$

else

if $y(S) > z(S) > x(S)$
split $S$ vertically into $S_1$ and $S_2$:

- $S_1$: $z(S) \times y(S) \times \lfloor x(S)/2 \rfloor$
- $S_2$: $z(S) \times y(S) \times \left(x(S) - \lfloor x(S)/2 \rfloor\right)$

Then:

- $N(S_1) = N(S) + 1$
- $N(S_2) = N(S) + 1$
- $\text{ShrinkSet}(S_1)$
- $\text{ShrinkSet}(S_2)$

Above, $\Gamma_n(S)$ is the significance state of set $S$, and $z(S), y(S), \text{and} \ x(S)$ are the number of ocean depths, rows, and columns, respectively, of set $S$.

The use of $k$-d trees in 3D-BISK is advantageous for the adaptive arithmetic coder used to code the significance information. Specifically, when set $S$ is split into $S_1$ and $S_2$, and $S_1$ is known to be insignificant (or empty), the significance state of $S_2$ is guaranteed to be significant. In this case, the significance state $\Gamma_n(S_2)$ is not coded into the bitstream. In the other case, the coding of $\Gamma_n(S_2)$ is conditioned with the knowledge that $S_1$ is significant.\(^1\) By contrast, the first seven subsets of an octree decomposition must be known to be insignificant for 3D-OB-SPECK to benefit from the same strategy.

The contexts used for set-significance coding within the BISK algorithm are as follows:

```plaintext
\begin{align*}
c(S_1) &\leftarrow \text{CONTEXT}_S1 \\
\text{if } \ S_1 = \emptyset \text{ or } \Gamma_n(S_1) = 0 \ \\
\quad c(S_2) &\leftarrow \text{CONTEXT}_\text{NOCODE} \\
\text{else} \ \\
\quad c(S_2) &\leftarrow \text{CONTEXT}_S2
\end{align*}
```

Above, $c(S_i)$ denotes the context that will be used to code $\Gamma_n(S_i)$.

\(^1\)Due to how sets are partitioned (see $\text{Partition}()$), $S_2$ is guaranteed to be nonempty while $S_1$ may or may not be empty.
In PartitionSet() above, a set is split into roughly equal-sized halves along the dimension of the set which is the longest. We refer to this strategy for determining the location and dimension of the set-splitting operation for the \( k \)-d tree as “longest dimension” (LD). We have investigated several alternative set-splitting approaches: 1) split a set into halves along the dimension that results in the smallest-sized sets after the set-shrinking operation (i.e., “smallest set” (SS)), 2) split a set along its longest dimension at the “center of mass” of its ocean points (i.e., “center of mass” (CM)), and 3) split a set along the dimension and at the center of mass that results in the smallest set sizes after the set-shrinking operation (i.e., “smallest set/center of mass” (SSCM)).

In the experimental results that follow in the next section, we empirically compare the performance of these set-splitting strategies.

### 6.2 Experimental Results

The performance of our 3D-BISK algorithm is measured using the ocean-temperature data from the study in [11, 35]. We use a three-level wavelet transform with the popular 9/7 biorthogonal filters, and find that, for all algorithms, the dyadic transform structure significantly outperforms the corresponding wavelet-packet transform. This is contrary to what we observed for hyperspectral imagery. Consequently, all algorithms use this dyadic transform, with the sole exception being 3D-EBCOT which cannot use the dyadic transform. All results for 3D-EBCOT are thus for the wavelet-packet transform. Throughout the experiments, rate is measured in bits per voxel (bpv) and distortion is measured as signal-to-noise ratio (SNR) in dB.

Table 6.1 compares the distortion performance at a given rate for the four set-splitting strategies for 3D-BISK outlined in Sec. 6.1.2. We see that, although the performances for all four approaches are rather similar, the LD strategy performs nearly
as well or better than all the other splitting policies. Consequently, we use the LD splitting strategy exclusively throughout the remainder of the text.

Fig. 6.3 presents the distortion obtained for two datasets over a range of rates. We see that 3D-BISK outperforms 3D-EBCOT by a wide margin (3–10 dB); this performance gap is due largely to the fact that 3D-EBCOT is constrained to use the wavelet-packet decomposition structure. Distortion performance for a given rate is tabulated in Table 6.2 for all the algorithms which use the dyadic transform. We see that 3D-BISK almost always yields the best distortion performance, being outperformed by tarp for only one dataset.

Interestingly, $k$-d trees often require more decompositions to represent a dataset than octrees. This suggests that binary set splitting allows for a simpler, more efficient entropy coding of the significance map. To confirm this hypothesis, we coded full datasets with no land so that ShrinkSet() had no effect on the coding process. In the case of coding these datasets, 3D-BISK and 3D-OB-SPECK exhibited virtually identical performance despite that fact that 3D-BISK incurred roughly seven times as many set-decomposition operations.

In this chapter, 3D-BISK is proposed for the shape-adaptive coding of ocean temperature volumes and experimental results are given. 3D-BISK replaces octrees of 3D-SPECK with $k$-d trees, a simpler set decomposition method that is well-suited to shape-adaptive coding. In addition, the ShrinkSet() provides additional performance gains by “shrinking” the sets to the bounding box of its significant coefficients. In the next chapter, some concluding remarks are made.
Figure 6.3: Rate-distortion performance for adrtc and bisca.

Table 6.1: Comparison of 3D-BISK Set-Splitting Strategies at 1.0 bpv

<table>
<thead>
<tr>
<th>Dataset</th>
<th>LD</th>
<th>CM</th>
<th>SS</th>
<th>SSCM</th>
</tr>
</thead>
<tbody>
<tr>
<td>bisca</td>
<td>53.1</td>
<td>53.1</td>
<td>53.2</td>
<td>53.1</td>
</tr>
<tr>
<td>nwlan</td>
<td>58.7</td>
<td>58.6</td>
<td>58.7</td>
<td>58.5</td>
</tr>
<tr>
<td>okina</td>
<td>61.6</td>
<td>61.6</td>
<td>61.5</td>
<td>61.7</td>
</tr>
<tr>
<td>ylsoj</td>
<td>54.2</td>
<td>54.1</td>
<td>54.1</td>
<td>54.2</td>
</tr>
</tbody>
</table>
Table 6.2: Distortion Performance at 1.0 bpv

<table>
<thead>
<tr>
<th>Dataset</th>
<th>BISK</th>
<th>SPECK</th>
<th>SPIHT</th>
<th>TARP</th>
<th>WDR</th>
</tr>
</thead>
<tbody>
<tr>
<td>adrtc</td>
<td>45.2</td>
<td>44.7</td>
<td>40.1</td>
<td>44.1</td>
<td>44.7</td>
</tr>
<tr>
<td>bisca</td>
<td>53.1</td>
<td>52.7</td>
<td>51.6</td>
<td>52.8</td>
<td>51.8</td>
</tr>
<tr>
<td>ginse</td>
<td>48.1</td>
<td>47.8</td>
<td>46.7</td>
<td><strong>48.1</strong></td>
<td>47.3</td>
</tr>
<tr>
<td>guama</td>
<td>70.7</td>
<td>69.7</td>
<td>68.7</td>
<td>70.3</td>
<td>70.1</td>
</tr>
<tr>
<td>hawai</td>
<td><strong>60.3</strong></td>
<td>59.9</td>
<td>59.0</td>
<td>59.7</td>
<td>60.1</td>
</tr>
<tr>
<td>med</td>
<td>49.3</td>
<td>48.9</td>
<td>46.7</td>
<td>48.9</td>
<td>48.8</td>
</tr>
<tr>
<td>nwlan</td>
<td>58.7</td>
<td>58.4</td>
<td>57.4</td>
<td>58.3</td>
<td>57.4</td>
</tr>
<tr>
<td>okina</td>
<td>61.6</td>
<td>60.7</td>
<td>57.3</td>
<td><strong>62.3</strong></td>
<td>61.1</td>
</tr>
<tr>
<td>socal</td>
<td>45.3</td>
<td>44.7</td>
<td>38.4</td>
<td>44.2</td>
<td>44.5</td>
</tr>
<tr>
<td>ylsoj</td>
<td><strong>54.2</strong></td>
<td>53.9</td>
<td>52.7</td>
<td>54.0</td>
<td>53.3</td>
</tr>
</tbody>
</table>
CHAPTER VII
CONCLUSIONS

The JPEG2000 standard covers only the decoder, so how encoders handle multiple spectral bands is left to the designer of the encoder. The results in this study demonstrate that different encoder designs may substantially affect performance. In Chap. III, we considered three different rate-allocation strategies for JPEG2000 and evaluated the performance of each, both with and without a spectral transform. We find that the technique that performs optimal rate-distortion truncation of bitstreams from all codeblocks from all image components—JPEG2000 multiple-component (JPEG2000-MC)—substantially outperforms the other techniques. Additionally, performance almost always benefits greatly from the application of a 1D spectral wavelet transform to remove correlation in the spectral direction. Also, we find that the POC results correlate well with the rate-distortion performance of the compression. We note that both Kakadu Version 4.3 and the JPEG2000 encoder in ENVI Version 4.1 (which uses the Kakadu coder) implement JPEG2000-MC rate allocation, and neither support the use of a spectral transform since they are not fully compliant with Part II of the JPEG2000 standard. Thus, the performance of these coders is equivalent to that of 2D JPEG2000-MC approach considered in Chap. III. As our results indicate, adding a spectral transform would significantly enhance the performance of these coders.

Our experimental observations indicate that all three techniques considered for the compression of hyperspectral imagery—3D-tarp, 3D-SPIHT, and JPEG2000—provide
largely similar rate-distortion performance for the datasets tested, with JPEG2000 often slightly outperforming the other two. Especially at low bit rates (less than 1 bpppb), all three techniques give nearly identical rate-distortion performance. However, given its simplicity of implementation and its ability to exploit a high degree of vectorization, 3D-tarp is perhaps the coder of the three that is best suited to on-board implementation, particularly when customized SIMD hardware with long vector lengths is possible.

In Chap. VI, we described 3D-BISK, an embedded, wavelet-based, 3D shape-adaptive coder for the compression of ocean-temperature data. We compared 3D-BISK to a variety of prominent 3D wavelet-based coding techniques, and experimental results demonstrated superior performance for 3D-BISK for a variety of ocean-temperature datasets. The performance gain was attributed to aggressive discarding of land-only sets and the simpler, more flexible partitioning of the sets that results from the binary $k$-d tree set-decomposition structure.

In this thesis, we have examined several 3D embedded wavelet-based compression algorithms for the storage and transmission of geoscience data, where the specific cases investigated are hyperspectral imagery and ocean temperature data. The body of results presented in Chap. V is, to our knowledge, the most comprehensive hyperspectral compression results in the literature. The results presented in Sec 6.2 are one of a few places in literature where compression results exist for ocean temperature data; they are the most extensive of the known publications.

Future work could include running results on hyperspectral data from other sensors; extending the techniques used in this work to compress other geoscience data; looking at the difference between compressing radiance, reflectance, and other hyperspectral data products; and implementing any of the compression schemes in hardware. Because JPEG2000 has gained so much popularity, there will most likely be few publications on 3D embedded wavelet-based coding techniques for geoscience data in the foreseeable
future. However, the problem of storing and transmitting the data will not disappear. Recently, there have been several promising techniques that involve unsupervised spectral-unmixing techniques to convert the data into the abundance domain, with the data being subsequently coded in the abundance domain. It is likely that spectral unmixing and other techniques that better preserve spectral information will be used in the future. However, some of the techniques used in this thesis may be used to compress the abundance volumes. For the time being, it appears 3D embedded wavelet-based coders are the best performing and most practical way to compress geoscience data while enabling progressive-transmission capabilities.
REFERENCES


APPENDIX A

HYPERSONTRAL UTILITIES
The following code was written to generate files compatible with QccPack [49], i.e.
icb files, from the raw sensor data, crop .icb files, and perform a spectral DWT and
inverse DWT. All of the following code is dependent upon QccPack to operate.

Code descriptions and syntax:

**rawtoicb.c** – Converts raw sensor data into a QccPack .icb file. The read order, i.e.
the precedent order of frames, rows, and columns, may need to be changed to conform
to a certain sensor format. It is assumed that the data is 16-bit. If the data is signed, then
include the -s switch.

usage: rawtoicb [-s] (num_rows) (num_cols) (num_frames) (raw file) (icb file)

**ImageCube23DData.c** - Converts QccPack .icb file to raw sensor data with the
formats BIL, BSQ, or BIP formatting.

usage: ImageCube23Data [-t (type)] (num_rows) (num_cols) (num_frames) (icb file)
(raw file)

**hypcrop.c** – Crops an .icb file to user specified dimensions.

usage: hypcrop (num_rows) (num_cols) (num_frames) (input file) (output file)

**hypdwt.c** – Performs a (num_scales) external 1D DWT, scales the data to occupy
the full range defined by the data width, i.e. (bits), and outputs the result into individual
files for use with the Kakadu v. 4.3 JPEG2000 coder. The output file names are
aviris.xxx.raw and out.aviris.xxx.raw. The out.aviris set is used for reconstruction with
kdu_expand.

usage: hypdwt [-ns (num_scales)] (bits) (input file)

**hypidwt.c** – Performs an external 1D IDWT on the output of kdu_expand and scales
the data appropriately.

usage: hypidwt [-ns (num_scales)] (num.rows) (num.cols) (num.frames) (bits) (output file)
run_experiment1 – A bash shell script that demonstrates how to read in a raw AVIRIS dataset, crop it, perform JPEG2000 compression on it, reconstructs the data, and calculates the distortion.
//rawtoicb.c: Converts raw sensor data into a QccPack .icb file. The read order of //frames rows and columns may need to changed to suite the raw data format

#include "libQccPack.h"

#define USG_STRING "[-s \t:] \t:d:num_rows \t:d:num_cols \t:d:num_frames \t:s:rawfile \t:s:icbfile"

QccString RawFile;

int NumRows;
int NumCols;
int NumFrames;
QccIMGImageCube ImageCube;

int Signed = 0;

int Read3DData(QccString filename,
                QccIMGImageCube *image_cube,
                int signed_data)
{
    FILE *file_ptr;
    int frame, row, col;
    int temp;
    char ch;
    int msb;
    int lsb;

    if (image_cube == NULL)
        return(1);

    if ((file_ptr = QccFileOpen(filename, "r")) == NULL)
    {
        QccErrorAddMessage("{Read3DData}: Error calling QccFileOpen()");
        return(1);
    }

    for (frame = 0; frame < image_cube->num_frames; frame++)
        for (row = 0; row < image_cube->num_rows; row++)
            for (col = 0; col < image_cube->num_cols; col++)
            {
                if (QccFileReadChar(file_ptr, &ch))
                {
                    QccErrorAddMessage("{Read3DData}: Error calling QccFileReadChar()");
                    return(1);
                }
                if (signed_data)
                    msb = ch << 8;
                else
                    msb = ((unsigned char)ch) << 8;
                if (QccFileReadChar(file_ptr, &ch))
                {
                    QccErrorAddMessage("{Read3DData}: Error calling QccFileReadChar()");
                    return(1);
                }
                lsb = (unsigned char)ch;
                temp = msb | lsb;
                image_cube->volume[frame][row][col] = (double)temp;
            }

    QccFileClose(file_ptr);
    return(0);
}

int main(int argc, char *argv[])
{
    QccInit(argc, argv);
    QccIMGImageCubeInitialize(&ImageCube);
}
if (QccParseParameters(argc, argv,
    USG_STRING,
    &Signed,
    &NumRows,
    &NumCols,
    &NumFrames,
    RawFile,
    ImageCube.filename))
    QccErrorExit();

ImageCube.num_rows = NumRows;
ImageCube.num_cols = NumCols;
ImageCube.num_frames = NumFrames;
if (QccIMGImageCubeAlloc(&ImageCube))
    {
    QccErrorAddMessage("\@s: Error calling QccIMGImageCubeAlloc()",
                        argv[0]);
    QccErrorExit();
    }

if (Read3DDData(RawFile, &ImageCube, Signed))
    {
    QccErrorAddMessage("\@s: Error calling Read3DDData()",
                        argv[0]);
    QccErrorExit();
    }

if (QccIMGImageCubeSetMinMax(&ImageCube))
    {
    QccErrorAddMessage("\@s: Error calling QccIMGImageCubeSetMinMax()",
                        argv[0]);
    QccErrorExit();
    }

if (QccIMGImageCubeWrite(&ImageCube))
    {
    QccErrorAddMessage("\@s: Error calling QccIMGImageCubeWrite()",
                        argv[0]);
    QccErrorExit();
    }

QccIMGImageCubeFree(&ImageCube);
QccExit;
//
// ImageCube23DData: Writes a QccPack Formatted file(.ich) to raw data in either BIL, BSQ, or BIP format.
/*
*/

#include "libQccPack.h"
#include "libQccPackIMG.h"

#define USG_STRING "[-t %s:type] %i:rows %i:cols %i:frames %s:infile %s:outfile"

QccString OutputFilename;
FILE *ofile;

QccIMGImageCube InputImage;
int NumFrames, NumRows, NumCols;

QccString BandOrder = "BSQ";

int main(int argc, char *argv[])
{
    unsigned short int image_word = 0;
    unsigned char bytensb;
    unsigned char bytelsh;
    int frame, row, col;
    int i;
    
    QccInit(argc, argv);
    QccIMGImageCubeInitialize(&InputImage);

    if (QccParseParameters(argc, argv,
        USG_STRING,
        BandOrder,
        &NumFrames,
        &NumRows,
        &NumCols,
        &InputImage.filename,
        &OutputFilename))
        QccErrorExit();

    InputImage.num_frames = NumFrames;
    InputImage.num_rows = NumRows;
    InputImage.num_cols = NumCols;

    QccIMGImageCubeAlloc(&InputImage);
    if (QccIMGImageCubeWrite(&InputImage))
    {
        QccErrorAddMessage("%s: Error calling QccIMGImageCubeWrite()", argv[0]);
        QccErrorExit();
    }

    ofile = QccFileOpen(OutputFilename,"w");
if (!strcmp(BandOrder, "BSQ", 3))
{
  printf("BandOrder is BSQ\n");
  for(frame = 0; frame < NumFrames; frame++)
    for(row = 0; row < NumRows; row++)
      for(col = 0; col < NumCols; col++)
        
        image_word = InputImage.volume[frame][row][col];
        bytelsb = (char)(image_word/256);
        bytelsb = (char)(image_word & 0X0F);
        if (QccFileWriteChar(ofile, bytelsb))
          
          QccErrorAddMessage("%s: Error calling QccFileReadChar()",
          argv[0]);
          QccErrorExit();
    }

    if (QccFileWriteChar(ofile, bytelsb))
    
    QccErrorAddMessage("%s: Error calling QccFileReadChar()",
    argv[0]);
    QccErrorExit();
}

/* //if((i==0)&(k<10)&&(j==0)){
          printf("%02x0 %02x0\n",bytelsb,bytelsb);
        }*/

/* //if(col == 10 & row == 2)
        printf(" %i", image_word);*/
}

//printf("\n");

if (!strcmp(BandOrder, "BIL", 3))
{
  printf("BandOrder is BIL\n");
  for(row = 0; row < NumRows; row++)
    for(frame = 0; frame < NumFrames; frame++)
      for(col = 0; col < NumCols; col++)
    
        image_word = InputImage.volume[frame][row][col];
        bytelsb = (char)(image_word/256);
        bytelsb = (char)(image_word & 0X0F);
        if (QccFileWriteChar(ofile, bytelsb))
          
          QccErrorAddMessage("%s: Error calling QccFileReadChar()",
          argv[0]);
          QccErrorExit();
    }

    if (QccFileWriteChar(ofile, bytelsb))
    
    QccErrorAddMessage("%s: Error calling QccFileReadChar()",
    argv[0]);
    QccErrorExit();
}

/* //if((i==0)&(k<10)&&(j==0)){
          printf("%02x0 %02x0\n",bytelsb,bytelsb);
        }*/

/* //if(col == 10 & row == 2)
        printf(" %i", image_word);*/
}

//printf("\n");
if (!strncmp(BandOrder, "BIP", 3))
{
  printf("BandOrder is BIP\n");
  for(row = 0; row < NumRows; row++)
    for(col = 0; col < NumCols; col++)
      for(frame = 0; frame < NumFrames; frame++)
      {
        image_word = InputImage.volume[frame][row][col];
        bytemsB = (char)(image_word/256);
        bytelsb = (char)(image_word & 0xFF);
        if (QccFileWriteChar(ofile, bytemsB))
          { QccErrorAddMessage("\$s: Error calling QccFileReadChar()",
            argv[0]);
            QccErrorExit();
          }
        if (QccFileWriteChar(ofile, bytelsb))
          { QccErrorAddMessage("\$s: Error calling QccFileReadChar()",
            argv[0]);
            QccErrorExit();
          }
        /* if((i==0)&&(k<=10)&&(j==0))
          printf("%02x %02x\n", bytemsB, bytelsb);
        */
        /* if(col == 10 & row == 2)
          printf(" %d", image_word);*/
      }
  /* for(frame = 0; frame < NumFrames; frame++)
    { printf(" %10.1f", InputImage.volume[frame][2][10]);
  */
  printf("\n");/*/  

  /* if (QccIMGImageCubePrint(&InputImage))
    { QccErrorAddMessage("\$s: Error calling QccIMGImageCubeWrite()", argv[0]);
      QccErrorExit();
    } */

QccIMGImageCubeFree(&InputImage);
QccFileClose(ofile);
QccExit;
}
// hypercrop.c
/* This program:
 * Reads in a .lib file
 * and crops it to the size specified in the command line
 * arguments
 */
#include "libQccPack.h"
#include "libQccPackAVAV.h"

#define USG_STRING "%i:rows %i:cols %i:frames %s:infile %s:outfile"

QccImageCube InputImage, OutputImage;

int main (int argc, char *argv[])
{
    int num_rows,num_cols,num_frames;
    int frame, row, col;
    QccImageCubeInitialize(&InputImage);
    QccImageCubeInitialize(&OutputImage);

    QccInit(argc, argv);

    if (QccParseParameters(argc, argv,
                      USG_STRING,
                      &num_rows,
                      &num_cols,
                      &num_frames,
                      InputImage.filename,
                      OutputImage.filename))
        QccErrorExit();

    if (QccImageCubeRead(&InputImage))
    {
        QccErrorAddMessage("%s: Error calling QccImageCubeWrite()", argv[0]);
        QccErrorExit();
    }

    if(num_rows <= InputImage.num_rows)
        OutputImage.num_rows = num_rows;
    if(num_frames <= InputImage.num_frames)
        OutputImage.num_frames = num_frames;
    if(num_cols <= InputImage.num_cols)
        OutputImage.num_cols = num_cols;

    if (QccImageCubeAlloc(&OutputImage))
    {
        QccErrorAddMessage("%s: Error calling QccImageCubeAlloc()", argv[0]);
        QccErrorExit();
    }

    for(frame=0; frame < num_frames; frame++)
        for(row=0; row < num_rows; row++)
            for(col=0; col < num_cols; col++)
                OutputImage.volume[frame][row][col] = InputImage.volume[frame][row][col];

    if (QccImageCubeWrite(&OutputImage))
    {
        QccErrorAddMessage("%s: Error calling QccImageCubeWrite()", argv[0]);
        QccErrorExit();
    }

    QccImageCubeFree(&InputImage);
    QccImageCubeFree(&OutputImage);

    QccExit;
}
//bygda.c
/* This program:
 * Reads in a .ibc file
 * performs a NumScales level spectral 1D DWT
 * outputs each individual component as a file
 * specifically for use with the Kakadu implementation
 * of JPEG2000
 * */
#define USG_STRING "[-ns %i:dwt_num_scales] %i:bits %s:infile"
QccString outfile;
QccString scalefilename = "scale.raw";
FILE *scalefile;
FILE *outfile;
QccIMGImageCube InputImage;

// contains the value of the next value to be written out to a file
int scaled_coeff=0;

// initial DWT vars
int Length = 384;
int NumScales = 5;
int StartOdd = 0;

QccWAVWavelet Wavelet;
QccString WaveletFilename = "CohenDaubechiesFeauveau.9-7.fbk";
QccString Boundary = "symmetric";
QccVector x = NULL;
QccVector y = NULL;

int main (int argc, char *argv[])
{
    double dwt_max_coeff = ~MAXDOUBLE;
    double mean;
    int row,col,frame,frame2;
    int NumCols,NumRows;
    int file_num = 0;
    int num_bits = 0;
    int max_scale = 1073741823;

    QccIMGImageCubeInitialize(&InputImage);
    QccInit(argc, argv);

    if (QccParseParameters(argc, argv,
           USG_STRING,
           &NumScales,
           &num_bits,
           InputImage.filename))
        QccErrorExit( );

    // used in scaling data to the range of maximum bit output.
    // NOTE: The highest precision kakadu will tolerate is 31 bits.
    max_scale = pow(2,(num_bits-1))-1;
    printf("num bits: %i , max scale: %i \n", num_bits, max_scale);
    if (QccIMGImageCubeRead(&InputImage))
        {
        QccErrorAddMessage("%s: Error calling QccIMGImageCubeWrite()", argv[0]);
        QccErrorExit();
        }
    Length = InputImage.num_frames;
    NumRows = InputImage.num_rows;
    NumCols = InputImage.num_cols;
}
for(row=0; row < NumRows; row++)
for(col=0; col < NumCols; col++)
for(frame=0; frame < Length; frame++)
{
    mean += InputImage.volume[frame][row][col];
}
mean /= Length*NumCols*NumRows;

printf("num_scales: %d, NumScales\n");
//print("num rows: %d, num cols: %d num frames: %d
//\n",InputImage.num_rows,InputImage.num_cols,InputImage.num_frames);
// Set the values of image cube to Vectors to pass to the QccWaveletDWTID
for(row=0; row < NumRows; row++)
{
    for(col=0; col < NumCols; col++)
    {
        for(frame=0; frame < Length; frame++)
        {
            x[frame] = InputImage.volume[frame][row][col] - (float)mean;
        }
        // for each row and col
        if (QccWaveletDWTID(x,
             y,
             Length,
             StartOdd,
             0,
             NumScales,
             &Wavelet))
        {
            QccErrorAddMessage("ks: Error Calling QccWaveletDWTID()", argv[0]);
            QccErrorExit();
        }
    for(frame2=0; frame2 < Length; frame2++)
    {
        if(fabs(y[frame2]) > dwt_max_coeff)
        {
            dwt_max_coeff = fabs(y[frame2]);
        }
        InputImage.volume[frame2][row][col] = y[frame2];
    }
}
// file initialization for read and write
scalefile = QccFileOpen(scalefilename, "w");
//writes the scale value out to a file for later use.
dwt_max_coeff = (float)dwt_max_coeff;
QccFileWriteDouble(scalefile, dwt_max_coeff);
QccFileWriteDouble(scalefile, mean);
QccFileClose(scalefile);
printf("max coeff: %f \n",dwt_max_coeff);
printf("mean: %f \n",mean);
for (file_num = 0; file_num < Length; file_num++)
{
    QccStringSprintf(outfilename, "aviris.%03d.raw", file_num);
    outfile = QccFileOpen(outfilename, "w");
    for (row=0; row < NumRows; row++)
    {
        for (col=0; col < NumCols; col++)
        {
            // this part does the scaling
            scaled_coeff = (int)rint((InputImage.volume[file_num][row][col] /
                dwt_max_coeff) * max_scale);
            /*if ((row==5) && (col==10) && (frame<20))
               {
                printf(" %i ", scaled_coeff);
               }*/
            // this part writes the new values out to a file
            QccFileWriteInt(outfile, scaled_coeff);
        }
        QccFileClose(outfile);
    }
}

printf("\n");
for (file_num = 0; file_num < Length; file_num++)
{
    QccStringSprintf(outfilename, "out.aviris.%03d.raw", file_num);
    outfile = QccFileOpen(outfilename, "w");
    for (row=0; row < NumRows; row++)
    {
        for (col=0; col < NumCols; col++)
        {
            // this part does the scaling
            scaled_coeff = (int)rint((InputImage.volume[file_num][row][col] /
                dwt_max_coeff) * max_scale);
            // this part writes the new values out to a file
            QccFileWriteInt(outfile, scaled_coeff);
        }
        QccFileClose(outfile);
    }
}
QccIMGImageCubeFree(&InputImage);
QccExit;
//hypidwt.c
/* This program:
   * Takes in a hyperspectral image of size row, column, height as 384 ind. files
   * converts back to doubles and unscales
   * performs a NumScales level 3D IDWT
   * puts back into a single volume
*/
#include "libQccPack.h"
#include "libQccPackKAV.h"
#include "malloc.h"
#include "values.h"
#include "limits.h"

#define USG_STRING "[-ns %i:dwt_num_scales] %i:row %i:col %i:depth %i:bits %s:outfile"

QccString infilename;
QccString scalefilename = "scale.raw";

FILE *infilen;
FILE *scalefile;
QccIMGImageCube OutputImage;
int image_row_dim = 0;
int image_col_dim = 0;
int image_depth_dim = 0;
double dwt_max_coeff = 0;
// contains the value of the next value to be written out to a file
int iscaled_coeff=0;

//DWT vars
int Length = 384;
int NumScales = 5;
int StartOdd = 0;
QccKAVWavelet Wavelet;
QccString WaveletFilename = "CohenDaubechiesFeauveau.9-7.fbk";
QccString Boundary = "symetric";
QccVector x = NULL;
QccVector y = NULL;

int main (int argc, char *argv[]) {
    double mean=0.0;
    int row,col,frame,frame2;
    int file_num = 0;
    int num_bits = 0;
    int max_scale = 1073741823;
    printf("Program Started\n");
    QccIMGImageCubeInitialize(&OutputImage);
    QccInit(argc, argv);

    if (QccParseParameters(argc, argv,
        USG_STRING,
        &NumScales,
        &image_row_dim,
        &image_col_dim,
        &image_depth_dim,
        &num_bits,
        scalefilename))
        QccErrorExit(  );

//used in scaling data to the range of maximum bit output.
//NOTE: The highest precision kakadu will tolerate is 31 bits.
    max_scale = pow(2,(num_bits-1))-1;
    scalefile = QccFileOpen(scalefilename, "r");
    QccFileReadDouble(scalefile, &dwt_max_coeff);
    QccFileReadDouble(scalefile, &mean);
    printf("max scale: %f max coeff: %f mean: %f \n",max_scale, dwt_max_coeff, mean);
    QccFileClose(scalefile);
// reads in the individual files after the JPEG2000 coding/decoding.
numfiles = image_depth_dim;
for(file_num = 0; file_num < numfiles; file_num++)
{
    QccStringSprint(infilename,"avirisn.k03d.raw", file_num);
    infile = QccFileOpen(infilename,"r");
    for(row=0; row < image_row_dim; row++)
        for(col=0; col < image_col_dim; col++)
        {
            QccFileReadChar(infile, &bytemsh);
            QccFileReadChar(infile, &byteelsb);
            image_word=(short int)(bytemsh << 8 | byteelsb);
            //QccFileReadInt(infile, &iscaled_coeff);
            //image_cube[row][col][file_num] = ((double)iscaled_coeff / max_scale) *
            abs(dwt_max_coeff);
            image_cube[row][col][file_num] = (float)image_word;
            //QccFileClose(infile);
        }
    printf("files read in and scaled\n");
}
// Inverse DWT
if(QccWAVWaveletCreate(&Wavelet, WaveletFilename, Boundary))
{
    QccErrorAddMessage("%s: Error Calling QccWAVWaveletCreate()", argv[0]);
    QccErrorExit();
}
else
{
    QccErrorAddMessage("%s: Error Calling QccWAVWaveletCreate()", argv[0]);
    QccErrorExit();
}
if((x = QccVectorAlloc(Length)) == NULL)
{
    QccErrorAddMessage("%s: Error Calling QccVectorAlloc()", argv[0]);
    QccErrorExit();
}
else
{
    QccErrorAddMessage("%s: Error Calling QccVectorAlloc()", argv[0]);
    QccErrorExit();
}
/* Read in slices inverse scale*/
//QccVectorPrint(x,Length);
/* for each row and col */
for(row=0; row < image_row_dim; row++)
    for(col=0; col < image_col_dim; col++)
    {
        for(frame=0; frame < image_dep_dim; frame++)
        {
            y[frame]=(double)image_cube[row][col][frame];
        }
        if(QccWAVWaveletInverseDWTID(y, x, Length, StartOdd, 0, NumScales, &Wavelet))
        {
            QccErrorAddMessage("%s: Error Calling QccWAVWaveletInverseDWTID()", argv[0]);
            QccErrorExit();
        }
    }
for(row=0; row < image_row_dim; row++)
{
    for(col=0; col < image_col_dim; col++)
    {
        for(frame=0; frame < image_dep_dim; frame++)
        {
            y[frame] = OutputImage.volume[frame][row][col];
        }
        if (QccWAVWaveletInverseDWT1D(y, 
            x, 
            image_dep_dim, 
            StartOdd, 
            0, 
            NumScales, 
            sWavelet))
        {
            QccErrorAddMessage("%s: Error Calling QccWAVWaveletInverseDWT1D()", argv[0]);
            QccErrorExit();
        }
        for(frame2=0; frame2 < image_dep_dim; frame2++)
        {
            // assigns the idwt coefficients to image_cube and then adds .5 rounding factor
            // for the int cast.
            OutputImage.volume[frame2][row][col] = x[frame2] + mean;
            /*if((frame2<10) && (row==5) && (col==10))
             {
                printf("%s, OutputImage.volume[frame2][row][col]\n");
            }*/
        }
    }
}
printf("\n");
printf("Inverse DWT complete\n");
if (QccIMGImageCubeWrite(&OutputImage))
{
    QccErrorAddMessage("%s: Error calling QccIMGImageCubeWrite()", argv[0]);
    QccErrorExit();
}
printf("File Written\n");
QccExit;
#!/bin/sh

This script shows an example of how to generate a .icb file from raw AVIRIS data. Modifications will need to be made to rawtoicb for other sensors.

crop_rows=512
crop_cols=512
crop_frames=224

# convert to icb. An AVIRIS dataset is 512 lines x 614 columns x 224 bands
# and the data is signed 16-bit integers.
.rawtoicb -s 512 614 224 ./moffett_1.a.rfl ./moffett_1.icb

# crop to user specified dimension
./hypcrop crop_rows crop_cols crop_frames ./moffett_1.icb ./moffett_1_crop.icb

# perform a 4 level 1D spectral DWT and scale to occupy the full 30 bit range
./hypdwt -ns 4 30 ./moffett_1_crop.icb

# get the input and output strings
files=`echo aviris.*.raw|sed -e"s/ //,//g"`
filesos=`echo out.aviris.*.raw|sed -e"s/ //,//g"`

# JPEG2000 Compression on transformed components.
# Actual Rate is 1.0 bits per pixel per band (bppb).
# Rate input to kdu_compress is crop_frame * desired bitrate, e.g. 224*1.0
./kdu_compress -1 $files -o jpeg2000.raw \
-Sprecision=31 $signed=yes $dims=$(${rows},${cols}) \
-rate 224 -no_weights Qstep=.0000001 Clevels=4

# Uncompress the files into individual components
./kdu_expand -i jpeg2000.raw -o $filesos -raw_components

# combine the components into an image cube, perform IDWT, and
# output reconstructed icb file
./hypidwt -ns 4 $rows $cols $frames 30 ./moffett_1_crop.jp2kmc.icb

# calculate the distortion with icbdist, a utility in QccPack.
icbdist -vo -snr ./moffett_1_crop.icb ./moffett_1_crop.jp2kmc.icb